Bo Kågström   Erik Elmroth
Jack Dongarra   Jerzy Waśniewski (Eds.)

Applied Parallel Computing

State of the Art in Scientific Computing

8th International Workshop, PARA 2006
Umeå, Sweden, June 18-21, 2006
Revised Selected Papers

Springer

Volume Editors

Bo Kågström
Erik Elmroth
Umeå University, Department of Computing Science
and High Performance Computing Center North (HPC2N)
90187 Umeå, Sweden
E-mail: {bokg, elmroth}@cs.umu.se

Jack Dongarra
University of Tennessee, Department of Computer Science
1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
E-mail: dongarra@cs.utk.edu

Jerzy Waśniewski
Technical University of Denmark, Informatics and Mathematical Modelling
Building 321, 2800 Kongens Lyngby, Denmark
E-mail: jw@imm.dtu.dk

Library of Congress Control Number: 2007936931

CR Subject Classification (1998): G.1-4, F.1-2, D.1-3, J.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-75754-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75754-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12175321 06/3180 5 4 3 2 1 0

Preface

The Eighth International Workshop on Applied Parallel Computing (PARA 2006) was held in Umeå, Sweden, June 18–21, 2006. The workshop was organized by the High Performance Computing Center North (HPC2N) and the Department of Computing Science at Umeå University. The general theme for PARA 2006 was "State of the Art in Scientific and Parallel Computing." Topics covered at PARA 2006 included basic algorithms and software for scientific, parallel and grid computing, tools and environments for developing high-performance computing applications, as well as a broad spectrum of applications from science and engineering.

The workshop included 7 plenary keynote presentations, 15 invited minisymposia organized in 30 sessions, and 16 sessions of contributed talks. The minisymposia and the contributed talks were held in five to six parallel sessions. The main workshop program was preceded by two half-day tutorials. In total, 205 presentations were held at PARA 2006, by speakers representing 28 countries. Extended abstracts for all presentations were made available at the PARA 2006 Web site (www.hpc2n.umu.se/para06).

The reviewing process was performed in two stages for evaluation of originality, appropriateness, and significance. In the first stage, extended abstracts were reviewed for selection of contributions to be presented at the workshop. In the second stage the full papers submitted after the workshop were reviewed. In total, 120 papers were selected for publication in this peer-reviewed post-conference proceedings.

A number of people contributed in different regards to the organization and the accomplishment of PARA 2006. First of all the Local Organization Committee did a greatly appreciated and enthusiastic job. We also acknowledge the following people for the assistance and support during the workshop days: Yvonne Löwstedt and Anne-Lie Persson; Niklas Edmundsson, Roger Oscarsson, and Mattias Wadenstein. A special thanks goes to the PARA 2006 secretary, Lena Hellman, to Anders Backman and Björn Torkelsson for designing and managing the PARA 2006 Web site including the electronic paper submission system, powered by Commence, and to Mats Nylén and Mikael Rännar for their professional assistance in compiling and editing the PARA 2006 program, the booklet of extended abstracts, and the final proceedings. PARA 2006 would not have been possible without the personal involvement of all these fine people. We also greatly acknowledge all minisymposia organizers, the review coordinators and all the referees for their evaluations in the second review stage, which included several rounds and resulted in these professionally peer-reviewed post-workshop proceedings. Finally, we would also like to thank the sponsoring institutions for their generous financial support.

Since 1996 the international PARA conferences have become biennial and are organized by one of the Nordic countries. The three first workshops including PARA 1996 and the last PARA 2004 were held in Lyngby, Denmark. The other three, besides this one, were held in Umeå, Sweden (PARA 1998), in Bergen, Norway (PARA 2000), and in Espoo, Finland (PARA 2002). The PARA 2008 workshop will take place in Trondheim, Norway, May 13–16, 2008.

March 2007

Bo Kågström
Erik Elmroth
Jack Dongarra
Jerzy Wasniewski

In Memoriam and Dedication

Every day we are reminded of the perishables of life and that our individual lives are only a gift for a finite time. Unfortunately, at PARA 2006 this was no exception. Our colleague Amelia De Vivo, Università degli Studi della Basilicata, Italy, passed away during the conference on June 21, 2006 in Umeå, Sweden. Another of our colleagues, Olav Beckmann, Imperial College London, UK died on November 9, 2006; this workshop was the last professional engagement of his career. As a token of our friendship and admiration of Amelia De Vivo and Olav Beckmann, we dedicate the PARA 2006 conference proceedings to them, from all PARA 2006 participants and with our regards to their families.

Organization

PARA 2006 was organized by the High Performance Computing Center North (HPC2N) and the Department of Computing Science at Umeå University.

Organization and Program Committee

Bo Kågström	Umeå University and HPC2N, Sweden *PARA 2006 Chairman*
Erik Elmroth	Umeå University and HPC2N, Sweden *PARA 2006 Coordinator*
Jack Dongarra	University of Tennessee and Oak Ridge National Laboratory, USA
Jerzy Wasniewski	Technical University of Denmark

Local Organization Committee

Anders Backman
Erik Elmroth (Coordinator)
Lena Hellman (Secretary)
Bo Kågström (Chairman)
Daniel Kressner
Mats Nylén
Mikael Rännar
Björn Torkelsson

Review Coordinators

Peter Arbenz
Siegfried Benkner
Ann-Charlotte Berglund-Sonnhammer
Xing Cai
Raimondas Ciegis
Zlatko Drmac
Anne C. Elster
Michael Grønager
Fred Gustavson
Sverker Holmgren
Mats Holmström
Paul Kelly
Christoph Kessler
David Kincaid
Sinisa Krajnovic
Daniel Kressner
Aatto Laaksonen
Claude Lacoursiére
Julien Langou
Hans Petter Langtangen
Mats G. Larson
Osni Marques
Lars Nordström
Frederik Orellana
Kjell Rönnmark
Jennifer Scott
Sameer Shende
Oxana Smirnova
Hans Stockinger
Jerzy Wasniewski
Jan Westerholm
Felix Wolf
Anders Ynnerman

Sponsoring Institutions

The Swedish Research Council (VR) via the Swedish National Infrastructure for Computing (SNIC).

Umeå University via the Vice-Chancellor, the High Performance Computing Center North (HPC2N), and the Department of Computing Science.

PARA 2006 Tutorials

Python in High-Performance Computing
Organizers and Lecturers: *Xing Cai*, *Hans Petter Langtangen*, and *Kent-Andre Mardal*, Simula Research Laboratory and Oslo University, Norway

Introduction to Object-Oriented Modeling and Simulation with Modelica
Organizers and Lecturers: *Peter Fritzson* and *Anders Sandahl*, Linköping University, Sweden

PARA 2006 Keynote Presentations

Implementing Advanced Force Fields for Simulation of Physical and Biological Processes
Tom Darden, National Institute of Environmental Health Science, North Carolina, USA

The Future of LAPACK and ScaLAPACK
James Demmel, University of California, Berkeley, USA

Supercomputers and Clusters and Grids, Oh My!
Jack Dongarra, University of Tennessee, Knoxville and Oak Ridge National Laboratory, USA

Large-Scale Ill-Posed Problems and Applications
Per Christian Hansen, Technical University of Denmark, Lyngby, Denmark

Mixed-Language Programming for HPC Applications
Hans Petter Langtangen, Simula Research Laboratory and Oslo University, Norway

Parallel MATLAB
Cleve Moler, The MathWorks Inc., USA

Recent Advances in Rendering and Interaction for Volumetric Data in Medical Applications
Anders Ynnerman, Linköping University, Sweden

PARA 2006 Minisymposia

Stretching Time and Length Scales in Biomolecular Modelling (18 presentations)
Organizer: *Aatto Laaksonen*, Stockholm University, Sweden

Recent Advances in Dense Linear Algebra (11 presentations)
Organizers: *Daniel Kressner*, University of Zagreb, Croatia and Umeå University, Sweden; *Julien Langou*, The University of Tennessee, Knoxville, USA

CFD Applications for High-Performance Computing (6 presentations)
Organizer: *Sinisa Krajnovic*, Chalmers, Sweden

HPC Environments: Visualization and Parallelization Tools (12 presentations)
Organizers: *Anne C. Elster*, Norwegian University of Science and Technology, Norway; *Otto Anshus*, University of Tromsø, Norway

Tools, Frameworks and Applications for High-Performance Computing (17 presentations)
Organizer: *Osni Marques*, Lawrence Berkeley National Laboratory, Berkeley, USA

Grid Data Management (5 presentations)
Organizers: *Siegfried Benkner*, University of Vienna, Austria; *Heinz Stockinger*, Swiss Institute of Bioinformatics, Lausanne, Switzerland

Simulations in Geophysics and Space Physics (9 presentations)
Organizers: *Mats Holmström*, Swedish Institute of Space Physics, Kiruna, Sweden; *Kjell Rönnmark*, Umeå University, Sweden

Tools for Parallel Performance Analysis (12 presentations)
Organizer: *Felix Wolf*, Forschungszentrum Jülich GmbH, Germany

Grids for Scientific Computing (9 presentations)
Organizer: *Oxana Smirnova*, Lund University, Sweden

Simulations of Materials (12 presentations)
Organizer: *Lars Nordström*, Uppsala University, Sweden

Novel Data Formats and Algorithms for Dense Linear Algebra Computations (13 presentations)
Organizers: *Fred Gustavson*, IBM T.J. Watson Research Center, New York, USA and Umeå University; *Jerzy Wasniewski*, Technical University of Denmark, Lyngby, Denmark

Bioinformatics and Computational Biology (5 presentations)
Organizers: *Ann-Charlotte Berglund Sonnhammer* and *Sverker Holmgren*, Uppsala University, Sweden

Scientific Visualization and HPC Applications (4 presentations)
Organizers: *Matt Cooper* and *Anders Ynnerman*, Linköping University, Sweden

Software Tools for Parallel CFD Applications (7 presentations)
Organizers: *Xing Cai* and *Hans Petter Langtangen*, Simula Research Laboratory and Oslo University, Norway

Multi-scale Physics (6 presentations)
Organizer: *Mats G. Larson*, Umeå University, Sweden

PARA 2006 Speakers

The keynote, minisymposia and contributed papers that are published in the proceedings were presented by the speakers listed below (the names are in the same order as the papers appear in the Table of Contents). We refer to the PARA 2006 program for a complete listing of all presentations and speakers.

Keynote Papers

Jack Dongarra
James W. Demmel
Per Christian Hansen
Hans Petter Langtangen

Minisymposia Papers

Peter G. Kusalik
Peter Ahlström
Christophe Labbez
Mikael Lund
Mikael Peräkylä
Paweł Sałek
Yaoquan Tu
David van der Spoel
Björn Adlerborn
Robert Granat
Craig Lucas
Jakub Kurzak
Enrique S. Quintana-Ortí
Håkan Nilsson
John Markus Bjørndalen
Espen Skjelnes Johnsen
Jan C. Meyer
Thorvald Natvig
Ingar Saltvik
Daniel Stødle
Peter Arbenz
L. Anthony Drummond
Viral B. Shah
Lutz Gross
Zsolt I. Lázár
Sameer Shende
Siegfried Benkner
Brian Coghlan
Carmela Comito
Francisco Almeida
Werner Benger
Pablo López
Madelene Jeanette Parviainen
Jörgen Vedin
Genaro Costa
Karl Fürlinger
Felix Wolf
Josep Jorba
Edmond Kereku
Andreas Knüpfer
Sameer Shende
Hung-Hsun Su
Brian J. N. Wylie
Oxana Smirnova
Sigve Haug
Roman Wyrzykowski
Pooja M. Panchmatia
Levente Vitos
Michael Bader
Isak Jonsson
Fred G. Gustavson
Lars Karlsson

Tadeusz Swirszcz
Jerzy Waśniewski
José R. Herrero
Przemysław Stpiczyński
Jennifer A. Scott
Enrique S. Quintana-Ortí
Tetsuya Sakurai
Mahen Jayawardena
Paul Sjöberg
Jeanette Tångrot
Paul R. Woodward
Markus Blatt
Bruno Carpentieri
Xing Cai
Hiroshi Okuda
Kent-Andre Mardal
Christophe Prud'homme
Eugene V. Shilnikov
Fredrik Bengzon

Contributed Papers

Juan A. Acebrón
Torsten Adolph
Nikolaos M. Missirlis
Scott B. Baden
Michael Spevak
Jorge Andrade
Amy Krause
Krzysztof Benedyczak
Jan Kwiatkowski
Gabriele Pierantoni
Olaf Schneider
Marilton S. de Aguiar
Tamito Kajiyama
Dries Kimpe
Carmen B. Navarrete
Gerhard Zumbusch
Jan Westerholm
Cevdet Aykanat
Kamen Yotov
Pascal Hénon
Takahiro Katagiri
Mariana Kolberg
Claude Lacoursière
Marzio Sala
Marzio Sala
Teruo Tanaka
René Heinzl
Christoph Kessler
Margreet Nool
Constantine Bekas
Nils Smeds
Vladimir A. Tcheverda
Bartlomiej Jacek Kubica
Tomas Lindén
Paul R. Woodward
Juan-Pedro Martínez-Gallar
Stéphane Vialle
Yurong Chen
Raimondas Čiegis
Jacek Dąbrowski
Euloge Edi
Thomas Fischer
Massimiliano Rak
Myungho Lee
Daisuke Takahashi

Table of Contents

The Impact of Multicore on Math Software*

Alfredo Buttari[1], Jack Dongarra[1,2], Jakub Kurzak[1], Julien Langou[3], Piotr Luszczek[1], and Stanimire Tomov[1]

[1] Innovative Computing Laboratory, University of Tennessee Knoxville, TN 37996, USA
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge National Laboratory, TN 37831, USA
dongarra@cs.utk.edu
[3] Department of Mathematical Sciences, University of Colorado at Denver and Health Sciences Center, Campus Box 170, P.O. Box 173364, CO 80217-3364, USA

Abstract. Power consumption and heat dissipation issues are pushing the microprocessors industry towards multicore design patterns. Given the cubic dependence between core frequency and power consumption, multicore technologies leverage the idea that doubling the number of cores and halving the cores frequency gives roughly the same performance reducing the power consumption by a factor of four. With the number of cores on multicore chips expected to reach tens in a few years, efficient implementations of numerical libraries using shared memory programming models is of high interest. The current message passing paradigm used in ScaLAPACK and elsewhere introduces unnecessary memory overhead and memory copy operations, which degrade performance, along with the making it harder to schedule operations that could be done in parallel. Limiting the use of shared memory to fork-join parallelism (perhaps with OpenMP) or to its use within the BLAS does not address all these issues.

1 Introduction

The idea that computational modeling and simulation represents a new branch of scientific methodology, alongside theory and experimentation, was introduced about two decades ago. It has since come to symbolize the enthusiasm and sense of importance that people in our community feel for the work they are doing. But when we try to assess how much progress we have made and where things stand along the developmental path for this new "third pillar of science," recalling some history about the development of the other pillars can help keep things in perspective. For example, we can trace the systematic use of experiment back to Galileo in the early seventeenth century. Yet for all the incredible successes it enjoyed over its first three centuries, and the considerable contributions from many outstanding scientists such as G. Mendel or C. R. Darwin, the experimental

* This work was supported in part by the National Science Foundation and Department of Energy.

method arguably did not fully mature until the elements of good experimental design and practice were finally analyzed and described in detail by R. A. Fisher and others in the first half of the twentieth century. In that light, it seems clear that while Computational Science has had many remarkable youthful successes, it is still at a very early stage in its growth.

Many of us today who want to hasten that growth believe that the most progressive steps in that direction require much more community focus on the vital core of Computational Science: *software and the mathematical models and algorithms it encodes.* Of course the general and widespread obsession with hardware is understandable, especially given exponential increases in processor performance, the constant evolution of processor architectures and supercomputer designs, and the natural fascination that people have for big, fast machines. But when it comes to advancing the cause of computational modeling and simulation as a new part of the scientific method, there is no doubt that the complex software "ecosystem" it requires must take its place on the center stage.

At the application level the science has to be captured in mathematical models, which in turn are expressed algorithmically and ultimately encoded as software. Accordingly, on typical projects the majority of the funding goes to support this translation process that starts with scientific ideas and ends with executable software, and which over its course requires intimate collaboration among domain scientists (physicists, chemists, biologists, etc), computer scientists and applied mathematicians. This process also relies on a large infrastructure of mathematical libraries, protocols and system software that has taken years to build up and that must be maintained, ported, and enhanced for many years to come if the value of the application codes that depend on it are to be preserved and extended. The software that encapsulates all this effort, energy, and thought, routinely outlasts (usually by years, sometimes by decades) the hardware it was originally designed to run on, as well as the individuals who designed and developed it.

Thus the life of Computational Science revolves around a multifaceted software ecosystem. But today there is (and should be) a real concern that this ecosystem of Computational Science, with all its complexities, is not ready for the major challenges that will soon confront the field. Domain scientists now want to create much larger, multi-dimensional applications in which a variety of previously independent models are coupled together, or even fully integrated. They hope to be able to run these applications on Petascale systems with tens of thousands of processors, to extract all the performance that these platforms can deliver, to recover automatically from the processor failures that regularly occur at this scale, and to do all this without sacrificing good programmability. This vision of what Computational Science wants to become contains numerous unsolved and exciting problems for the software research community. Unfortunately, it also highlights aspects of the current software environment that are either immature or under funded or both [3].

2 The Challenges of Multicore

It is difficult to overestimate the magnitude of the discontinuity that the high performance computing (HPC) community is about to experience because of the emergence of the next generation of multi-core and heterogeneous processor designs [4]. For at least two decades, HPC programmers have taken for granted that each successive generation of microprocessors would, either immediately or after minor adjustments, make their old software run substantially faster. But three main factors are converging to bring this "free ride" to an end. First, system builders have encountered intractable physical barriers – too much heat, too much power consumption, and too much leaking voltage – to further increases in clock speeds. Second, physical limits on the number and bandwidth of pins on a single chip means that the gap between processor performance and memory performance, which was already bad, will get increasingly worse. Finally, the design trade-offs being made to address the previous two factors will render commodity processors, absent any further augmentation, inadequate for the purposes of tera- and peta-scale systems for advanced applications. This daunting combination of obstacles has forced the designers of new multi-core and hybrid systems, searching for more computing power, to explore architectures that software built on the old model are unable to effectively exploit without radical modification [5].

But despite the rapidly approaching obsolescence of familiar programming paradigms, there is currently no well understood alternative in whose viability the community can be confident. The essence of the problem is the dramatic increase in complexity that software developers will have to confront. Dual-core machines are already common, and the number of cores is expected to roughly double with each processor generation. But contrary to the assumptions of the old model, programmers will not be able to consider these cores independently (i.e. multi-core is *not* "the new SMP") because they share on-chip resources in ways that separate processors do not. This situation is made even more complicated by the other non-standard components that future architectures are expected to deploy, including mixing different types of cores, hardware accelerators, and memory systems. Finally, the proliferation of widely divergent design ideas shows that the question of how to best combine all these new resources and components is largely unsettled. When combined, these changes produce a picture of a future in which programmers must overcome software design problems that are vastly more complex and challenging than in the past in order to take advantage of the much higher degrees of concurrency and greater computing power that new architectures will offer.

The work that we currently pursue is the *initial phase* of a larger project in *Parallel Linear Algebra for Scalable Multi-Core Architectures*(*PLASMA*) that aims to address this critical and highly disruptive situation. While PLASMA's ultimate goal is to create software frameworks that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures, the current high levels of disorder and uncertainty in the field processor design make it premature to attack this goal directly. More experimentation is needed with these new designs in order to see how prior techniques can be made useful by recombination or creative application and to discover what novel approaches can be developed into making our programming models sufficiently flexible and adaptive for the new regime.

Preliminary work we have already done on available multi-core and heterogeneous systems, such as the IBM CELL processor, shows that techniques for increasing parallelism and exploiting heterogeneity can dramatically accelerate application performance on these types of systems. Other researchers have already begun to utilize these results. Under this early PLASMA project, we are leveraging our initial work in the following three-pronged research effort:

– *Experiment with techniques* – Building on the model of large grain data flow analysis, we are exploring techniques that exploit dynamic and adaptive out-of-order execution patterns on multi-core and heterogeneous systems. Early experiences with matrix factorization techniques have already led us to the idea of dynamic look-ahead, and our preliminary experiments show that this technique can yield great improvements in performance.
– *Develop prototypes* – We are testing the most promising techniques through highly optimized (though neither flexible nor portable and thus not general enough) implementations that we, and other researchers in the community, can use to study their limits and gain insight into potential problems. These prototypes are also enabling us to assess how well suited these approaches are to dynamic adaptation and automated tuning.
– *Provide a design draft for the PLASMA framework* – An initial design plan for PLASMA frameworks for multi-core and hybrid architectures is being developed and, in combination with PLASMA software prototypes, will be distributed for community feedback.

Though it is clear that the impact of the multi-core revolution will be ubiquitous, we believe that, in developing a programming model for this radically different environment, there are clear advantages to focusing on Linear Algebra (LA) in general and Dense Linear Algebra (DLA) in particular, as PLASMA does. For one thing, DLA libraries are critically important to Computational Science across an enormous spectrum of disciplines and applications, so a programming framework of the type we envision for PLASMA will certainly be indispensable and needs to be achieved as quickly as possible. But DLA also has strategic advantages as a research vehicle, because the methods and algorithms that underlie it have been so thoroughly studied and are so well understood. This background understanding will allow us to devise techniques that maximally exploit the resources of the microprocessor platforms under study.

As a third point, we claim that the techniques developed for LA are general enough to be utilized in other software libraries. In this respect, the research performed on the PLASMA project is expected to be beneficial for other libraries. Historically this has been the case with LAPACK and its use of the BLAS. Nowadays several libraries outside the LA discipline have followed the LAPACK example and their performance heavily relies on the BLAS. The inverse is not true, and we can not find such a generality in other disciplines.

2.1 Main Factors Driving the Multi-core Discontinuity

Among the various factors that are driving the momentous changes now occurring in the design of microprocessors and high end systems, three stand out as especially notable: 1) *the number of transistors on the chip will continue to double roughly every 18 months, but the speed of processor clocks will not continue to proportionally increase*; 2) *the number and bandwidth of pins on CPUs are reaching their limits* and 3) *there will be a strong drift toward hybrid installations for petascale (and larger) systems.* The first two involve fundamental physical limitations that nothing currently on the horizon is likely to overcome. The third is a consequence of the first two, combined with the economic necessity of using many thousands of CPUs to scale up to petascale and larger systems. Each of these factors has a somewhat different effect on the design space for future programming:

1. *More transistors and slower clocks means multi-core designs and more parallelism required* – The *modus operandi* of traditional processor design – increase the transistor density, speed up the clock rate, raise the voltage – has now been blocked by a stubborn set of physical barriers – too much heat produced, too much power consumed, too much voltage leaked. Multi-core designs are a natural response to this situation. By putting multiple processor cores on a single die, architects can continue to increase the number of gates on the chip without increasing the power densities. But since excess heat production means that frequencies can not have a sustained increase, deep-and-narrow pipeline models will tend to recede as shallow-and-wide pipeline designs become the norm. Moreover, despite obvious similarities, multi-core processors are not equivalent to multiple-CPUs or to SMPs. Multiple cores on the same chip can share various caches (including TLB!) and they certainly share the bus. Extracting performance from this configuration of resources means that programmers must exploit increased thread-level parallelism (TLP) and efficient mechanisms for inter-processor communication and synchronization to manage resources effectively. The complexity of parallel processing will no longer be hidden in hardware by a combination of increased instruction level parallelism (ILP) and deep-and-narrow pipeline techniques, as it was with superscalar designs. It will have to be addressed in software.
2. *Thicker "memory wall" means that communication efficiency will be even more essential* – The pins that connect the processor to main memory have become a strangle point, with both the rate of pin growth and the bandwidth per pin slowing down, if not flattening out. Thus the processor to memory performance gap, which is already approaching a thousand cycles, is expected to grow, by 50% per year according to some estimates. At the same time, the number of cores on a single chip is expected to continue to

double every 18 months, and since limitations on space will keep the cache resources from growing as quickly, cache per core ratio will continue to go down. Problems of memory bandwidth, memory latency, and cache fragmentation will, therefore, tend to get worse.

3. *Limitations of commodity processors will further increase heterogeneity and system complexity* – Experience has shown that tera- and petascale systems must, for the sake of economic viability, use commodity off-the-shelf (COTS) processors as their foundation. Unfortunately, the trade-offs that are being (and will continue to be) made in the architecture of these general purpose multi-core processors are unlikely to deliver the capabilities that leading edge research applications require, even if the software is suitably modified. Consequently, in addition to all the different kinds of multithreading that multi-core systems may utilize – at the core-level, socket-level, board-level, and distributed memory level – they are also likely to incorporate some constellation of special purpose processing elements. Examples include hardware accelerators, GPUs, off-load engines (TOEs), FPGAs, and communication processors (NIC-processing, RDMA). Since the competing designs (and design lines) that vendors are offering are already diverging, and mixed hardware configurations (e.g. Los Alamos Roadrunner, Cray BlackWidow) are already appearing, the hope of finding a common target architecture around which to develop future programming models seems at this point to be largely forlorn.

We believe that these major trends will define, in large part at least, the design space for scientific software in the coming decade. But while it may be important for planning purposes to describe them in the abstract, to appreciate what they mean in practice, and therefore what their strategic significance may be for the development of new programming models, one has to look at how their effects play out in concrete cases. Below we describe our early experience with these new architectures, both how they render traditional, cornerstone numerical libraries obsolete, and how innovative techniques can exploit their parallelism and heterogeneity to address these problems.

2.2 Free Ride Is over for HPC Software: Case of LAPACK/ScaLAPACK

One good way to appreciate the impact and significance of the multi-core revolution is to examine its effect on software packages that are comprehensive and widely used. The LAPACK/ScaLAPACK libraries fit that description. These libraries, which embody much of our work in the adaptation of block partitioned algorithms to parallel linear algebra software design, have served the HPC and Computational Science community remarkably well for twenty years. Both LAPACK and ScaLAPACK apply the idea of blocking in a consistent way to a wide range of algorithms in linear algebra (LA), including linear systems, least square problems, singular value decomposition, eigenvalue decomposition, etc., for problems with dense and banded coefficient matrices. ScaLAPACK also addresses the much harder problem of implementing these routines on top of distributed memory architectures. Yet it manages to keep close correspondence to LAPACK in the way the code is structured or organized. The design of these packages has had a major impact on how mathematical software has been written and used successfully during that time. Yet when you look at how these foundational libraries can be expected to fair on large-scale multi-core systems, it becomes clear that we are on the verge of a transformation in software design at least as potent as the change engendered a decade ago by message passing architectures, when the community had to rethink and rewrite many of its algorithms, libraries, and applications.

Historically, LA methods have put a strong emphasis on *weak scaling or isoscaling* of algorithms, where speed is achieved when the number of processors is increased while the problem size per processor is kept constant, effectively increasing the overall problem size. This measure tells us when we can exploit parallelism to solve larger problems. In this approach, increasing speed of a single processing element should decrease the time to solution. But in the emerging era of multiprocessors, although the number of processing elements (i.e., cores) in systems will grow rapidly (exponentially, at least for a few generations), the computational power of individual processing units is likely to be reduced. Many problems in scientific computing reach their scaling limits on a certain number of processors determined by the ratio of computation/communication. With the speed of individual cores in the system on a decline, those problems will require *increased time to solution on the next generation of architectures.* In order to address the issue, emphasis has to be shifted from weak to *strong scaling*, where speed is achieved when the number of processors is increased while *the overall problem size is kept constant*, which effectively decreases the problem size per processor. In other words we need to seek more parallelism in algorithms and push their existing scaling limitations by investigating parallelization at a much finer levels of granularity.

The standard approach to parallelization of numerical linear algebra algorithms for both shared and distributed memory systems, utilized by the LAPACK/ScaLAPACK libraries, is to rely on a parallel implementation of BLAS - threaded BLAS for shared memory systems and PBLAS for distributed memory systems. Historically, this approach made the job of writing hundreds of routines in a consistent and accessible manner doable. But although this approach solves numerous complexity problems, it also enforces a very rigid and inflexible software structure, where, *at the level of LA, the algorithms are expressed in a serial way.* This obviously inhibits the opportunity to exploit inherently parallel algorithms at finer granularity. This is shown by the fact that the traditional method is successful mainly in extracting parallelism from Level 3 BLAS; in the case of most of the Level 1 and 2 BLAS, however, it usually fails to achieve speedups and often results in slowdowns. It relies on the fact that, for large enough problems, the $O(n^3)$ cost of Level 3 BLAS dominates the computation and renders the remaining operations negligible. The problem with encapsulating parallelization in the BLAS/PBLAS in this way is that it requires a heavy synchronization model on a shared memory system and a heavily synchronous and blocking form of collective communication on distributed memory systems with message passing. This paradigm will break down on the next generation architectures, because it relies on coarse grained parallelization and emphasizes *weak scaling*, rather than *strong scaling.*

2.3 Preliminary Work: Exploiting Parallelism on Multi-core

We used the forgoing analysis of the problems of LAPACK/ScaLAPACK on multi-core systems as the basis of some preliminary tests of techniques for doing fast and efficient LA on multi-core. LA operations are usually performed as a sequence of smaller tasks; it is possible to represent the execution flow of an operation as a Directed Acyclic Graph (DAG) where the nodes represent the sub-tasks and the edges represent the dependencies among them. Whatever the execution order of the sub-tasks is, the result will be correct as long as these dependencies are not violated. This concept has been used in the past to define "look-ahead" techniques that have been extensively applied to the LU factorization . Such methods can be used to remedy the problem of synchronizations introduced by non-parallelizable tasks by overlapping their execution with the execution of more efficient ones [1]. Although the traditional technique of look-ahead usually provides only a static definition of the execution flow that is hardwired in the source code, the idea of out-of-order execution it embodies can be extended to broader range of cases, where the execution flow is determined at run time in a fully dynamic fashion. With this dynamic approach, the subtasks that contribute to the result of the operation can be scheduled dynamically depending on the availability of resources and on the constraints defined by the dependencies among them (i.e., edges in the DAG).

Our recent work shows how the one-sided factorizations, LU, QR and Cholesky can benefit from the application of this technique [2]. Block formulations of these three factorizations, as well as many other one-sided transformations, follow a common scheme. In a single step of each algorithm, first operations are applied to a single block of rows or columns, referred to as the panel, then the result is applied to the remaining portion of the matrix. The panel operations are usually implemented with Level 1 and 2 BLAS and, in most cases, achieve the best performance when executed on a single processor or a small subset of all the processors used for the factorization.

It is well known that matrix factorizations have left-looking and right-looking formulations. The transition between the two can be done by automatic code transformations, although this requires more powerful methods than simple dependency analysis. In particular, the technique of look-ahead can be used to significantly improve the performance of matrix factorizations by performing panel factorizations in parallel with the update to the remaining submatrix from a previous step of the algorithm. The look-ahead can be of arbitrary depth, as was shown, for example, in the High Performance LINPACK benchmark (HPL). The look-ahead simply alters the order of operations in the factorization. A great number of permutations is legal, as long as algorithmic dependencies are not vi-

```
while(1)
   fetch_task();
   switch(task.type) {
      case PANEL:            // reduce the panel
         dgetf2();
         update_progress();
      case COLUMN:           // update a block-column
         dlaswp();           // of the trailing submatrix
         dtrsm();
         dgemm();
         update_progress();
      case END:              // perform left swap and return
         for()
            dlaswp();
         return;
   }
}
```

Fig. 1. Pseudo-code showing the execution flow for the LU factorization. The same execution scheme applies to the other one-sided transformations Cholesky and QR.

olated. From this point of view, right-looking and left-looking formulations of a matrix factorization are on two opposite ends of a wide spectrum of possible execution paths, with the look-ahead providing a transition between them. If the straight right-looking formulation is regarded as one with the look-ahead of zero, then the left-looking formulation is equivalent to the right looking formulation with the maximum possible look-ahead for a given problem.

Applying the idea of dynamic execution flow definition to LU factorization leads to the implementation of the left-looking variant of the algorithm, where the panel factorizations are performed as soon as possible, with the modification that if the panel factorization introduces a stall, then an update to a block of columns (or rows) of the right submatrix is performed instead. The updating continues only until next panel factorization is possible. Figure 1 (above) shows the simplified code that defines the execution flow. Here the steps of checking dependencies and making a transition are merged into the step of fetching the next task (the `fetch_task()` subroutine), where the choice of transition is made dynamically at run-time depending on the progress of the execution.

Experimental results show how the dynamic workflow technique is capable of improving the overall performance while providing an extremely high level of portability. Figure 2 shows that by applying dynamic task scheduling to the QR factorization, it is possible to out perform a standard LAPACK implementation with threaded BLAS.

3 The Future

Advancing to the next stage of growth for computational simulation and modeling will require us to solve basic research problems in Computer Science and Applied Mathematics at the same time as we create and promulgate a new paradigm for the development of scientific software. To make progress on both fronts simultaneously will require a level of sustained, interdisciplinary collaboration among the core research communities that, in the past, has only been

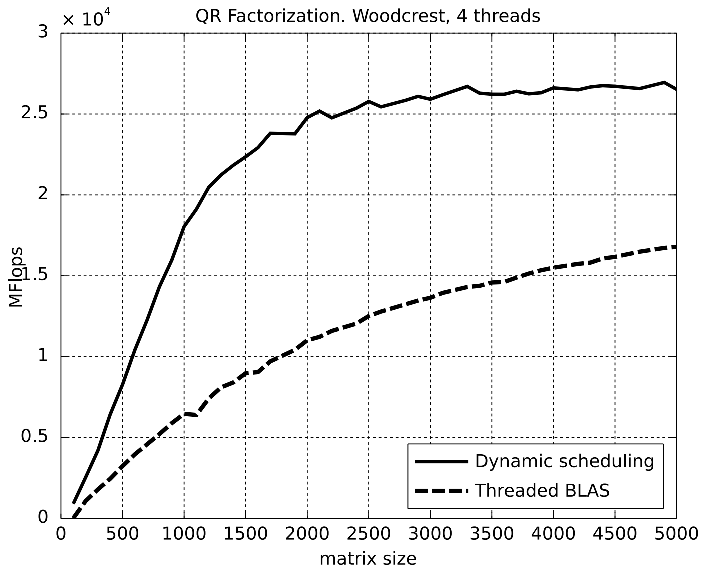


Fig. 2. Comparison of parallelization techniques for QR factorization (Two dual core Intel 3.0GHz Woodcrest processors (four cores total), GOTO BLAS 1.05, blocksize NB=64)

achieved by forming and supporting research centers dedicated to such a common purpose. We believe that the time has come for the leaders of the Computational Science movement to focus their energies on creating such software research centers to carry out this indispensable part of the mission.

References

1. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: Past, Present, and Future. Concurrency and Computation: Practice and Experience 15(9), 803–820 (2003), `http://www.netlib.org/benchmark/hpl/`
2. Kurzak, J., Dongarra, J.J.: Implementation of Linear Algebra Routines with Lookahead - LU, Cholesky, QR. In: Workshop on State-of-the-Art in Scientific and Parallel Computing, June, 2006, Umea, Sweden (2006)
3. Post, D.E., Votta, L.G.: Computational Science Demands a New Paradigm. Physics Today 58(1), 35–41 (2005)
4. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (March 2005)
5. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley, Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183 (December 18, 2006), `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`

Prospectus for the Next LAPACK and ScaLAPACK Libraries

James W. Demmel[1], Jack Dongarra[2,3], Beresford Parlett[1], William Kahan[1], Ming Gu[1], David Bindel[1], Yozo Hida[1], Xiaoye Li[1], Osni Marques[1], E. Jason Riedy[1], Christof Vömel[1], Julien Langou[2], Piotr Luszczek[2], Jakub Kurzak[2], Alfredo Buttari[2], Julie Langou[2], and Stanimire Tomov[2]

[1] University of California, Berkeley CA 94720, USA
`demmel@cs.berkeley.edu`
[2] University of Tennessee, Knoxville TN 37996, USA
[3] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Abstract. New releases of the widely used LAPACK and ScaLAPACK numerical linear algebra libraries are planned. Based on an on-going user survey (www.netlib.org/lapack-dev) and research by many people, we are proposing the following improvements: Faster algorithms, including better numerical methods, memory hierarchy optimizations, parallelism, and automatic performance tuning to accommodate new architectures; More accurate algorithms, including better numerical methods, and use of extra precision; Expanded functionality, including updating and downdating, new eigenproblems, etc. and putting more of LAPACK into ScaLAPACK; Improved ease of use, e.g., via friendlier interfaces in multiple languages. To accomplish these goals we are also relying on better software engineering techniques and contributions from collaborators at many institutions.

1 Introduction and Motivation

LAPACK and ScaLAPACK are widely used software libraries for numerical linear algebra. There have been over 68M web hits at www.netlib.org for the associated libraries LAPACK, ScaLAPACK, CLAPACK and LAPACK95. LAPACK and ScaLAPACK are used to solve leading edge science problems and they have been adopted by many vendors and software providers as the basis for their own libraries, including AMD, Apple (under Mac OS X), Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, several Linux distributions (such as Debian), NAG, IMSL, the MathWorks (producers of MATLAB), Interactive Supercomputing, and PGI. Future improvements in these libraries will therefore have a large impact on users.

The ScaLAPACK and LAPACK development is mostly driven by algorithm research, the result of the user/vendor survey, the demands and opportunities of new architectures and programming languages, and the enthusiastic participation of the research community in developing and offering improved versions of existing Sca/LAPACK codes [51].

Brief outline of the paper: Section 2 discusses challenges in making current algorithms run efficiently, scalably, and reliably on future architectures. Section 3 discusses two kinds of improved algorithms: faster ones and more accurate ones. Since it is hard to improve both simultaneously, we choose to include a new faster algorithm if it is about as accurate as previous algorithms, and we include a new more accurate algorithm if it is at least about as fast as the previous algorithms. Section 4 describes new linear algebra functionality that will be included in new Sca/LAPACK releases. Section 5 describes our proposed software structure for Sca/LAPACK. Section 6 describes initial performance results.

2 Challenges of Future Architectures

Parallel computing is becoming ubiquitous at all scales of computation. It is no longer just exemplified by the TOP 500 list of the fastest computers in the world. In a few years typical laptops are predicted to have 64 cores per multicore processor chip, and up to 256 hardware threads per chip. So unless all algorithms (not just numerical linear algebra!) can exploit this parallelism, they will cease to speed up, and in fact slow down compared to machine peak.

Furthermore, the gap between processor speed and memory speed continues to grow exponentially: processor speeds are improving at 59% per year, main memory bandwidth at only 23%, and main memory latency at a mere 5.5% [39]. This means that an algorithm that is efficient today, because it does enough floating point operations per memory reference to mask slow memory speed, may not be efficient in the near future. The same story holds for parallelism, with communication network bandwidth improving at just 26%, and network latency unimproved since the Cray T3E in 1996 until recently.

The largest scale target architectures of importance for LAPACK and ScaLAPACK include platforms now installed at NSF and DOE sites, as well as near term procurements. Longer term the High Productivity Computing Systems (HPCS) program [45] is supporting the construction of petascale computers by Cray (Cascade) and IBM (PERCS).

LAPACK and ScaLAPACK will have to run efficiently and correctly on a much wider array of platforms than in the past. In addition to the above architecturally diverse set of supercomputers and multicore chips in laptops, some future architectures are expected to be heterogeneous. For example, a cluster purchased over time will consist of some old, slow processors and some new, fast ones. Some processors may have higher loads from multiple users than others. Even single machines will be heterogenous, consisting of a CPU and other, faster, special purposes processors like GPUs, SSE units, etc. These will not just be heterogeneous in performance, but possibly in floating point semantics, with different units treating exceptions differently, or only computing in single precision. In a cluster, if one processor runs fastest handling denormalized numbers according to the IEEE 754 floating point standard [46], and another is fastest when flushing them to zero, then sending a number from one processor to another may change its value or even lead to a trap. Either way, correctness is a challenge, and not just for linear algebra.

It will be a challenge to map LAPACK's and ScaLAPACK's current software hierarchy of BLAS/BLACS/PBLAS/LAPACK/ScaLAPACK efficiently to all these platforms. For example, on a platform with multiple levels of parallelism (multicores, SMPs, distributed memory) would it be better to treat each SMP node as a ScaLAPACK process, calling parallelized BLAS or should each processor within the SMP be mapped to a process, or something else?

A more radical departure from current practice would be to make our algorithms asynchronous. Currently our algorithms are block synchronous, with phases of computation followed by communication with (implicit) barriers. But on networks that can overlap communication and computation, or on multi-threaded shared memory machines, block synchrony can leave a significant fraction of the platform idle at any time. For example, the LINPACK benchmark version of LU decomposition exploits such asynchrony and can run 2x faster than its block synchronous ScaLAPACK counterpart (see Section 5).

3 Better Algorithms

Three categories of routines are going to be addressed in Sca/LAPACK: (1) faster and/or more accurate algorithms for functions in LAPACK, which also need to be put in ScaLAPACK (discussed here) (2) functions now in LAPACK but not in ScaLAPACK (discussed in Section 4), and (3) functions in neither LAPACK nor ScaLAPACK (also discussed in Section 4). The following are lists of planned improvements.

Linear Systems and Least Squares Problems. Possible improvements include (1) iterative refinement using portable extra precision BLAS [52,5,14,13] to get guaranteed accuracy in linear systems [24] and least squares problems; (2) iterative refinement where the LU factorization is computed in single precision even though all the data is in double precision, in order to exploit the fact that single can be from 2x faster than double (on an SSE unit) to 10x faster (on an IBM Cell) – refinement is used to try to make the answer as accurate as standard double precision LU factorization; (3) recursive data structures of Gustavson, Kågström et al [35] to improve memory locality, in particular for symmetric packed matrix factorizations, but keeping the usual columnwise matrix interface; (4) a more stable pivoting scheme for symmetric indefinite matrices proposed by Ashcraft, Grimes and Lewis [4], that keeps the L factor more bounded than the current Bunch-Kaufman factorization; (5) Cholesky factorization with diagonal pivoting [44] that avoids a breakdown if the matrix is nearly indefinite/rank-deficient, which is useful both for optimization problems and computing high accuracy symmetric positive definite eigenvalue decompositions (EVD); (6) improved condition estimators for tridiagonal [25,43] or triangular [34] matrices; and (7) "latency avoiding" variations on the LU and QR decompositions that reduce the number of messages sent by factor equal to the block size in the 2D block-cyclic layout, which may be advantageous when the latency is large.

Eigenvalue Problems. Possible improvements include (1) the 2003 SIAM Linear Algebra Prize winning work of Braman, Byers, and Mathias [16,17] for solving the nonsymmetric eigenvalue problem up to 10x faster, as well as extensions to QZ in collaboration with Kågström and Kressner [50]; (2) a more complete implementation of the 2006 SIAM Linear Algebra Prize winning work of Dhillon and Parlett [57] on the Multiple Relatively Robust Representations (MRRR) algorithm for the symmetric eigenvalue problem, including work by Parlet and Vömel [58] to deal with tight clusters of eigenvalues and by Bientensi, Dhillon and can de Geihn on load balancing in the parallel version [9]; (3) extensions of the MRRR algorithm to the SVD [41], though some potential obstacles to guaranteed stability remain [70]; (4) a faster reduction to bidiagonal form for the SVD by Howell, Fulton, et al [37] that uses new BLAS [13] to achieve better memory locality; (5) a different faster bidiagonal reduction suitable for the case when only left or only right singular vectors are desired, but with possible less numerical stability [7,59]; (6) when only a few eigen- or singular vectors are desired, the successive band reduction approach of Bischof and Lang [11] can move most flops from level 2 to level 3 BLAS; (7) an efficent algorithm by Drmač and Veselić for computing the SVD with high relative accuracy [33]; and (8) analogous high accuracy algorithms for the symmetric indefinite EVD by Slapničar [61] and by Dopico, Molera and Moro [32].

4 Added Functionality

Putting More of LAPACK into ScaLAPACK. Numerous matrix data types supported by LAPACK are not in ScaLAPACK. The most important omissions are as follows: (1) There is no support for packed storage of symmetric (SP,PP) or Hermitian (HP,PP) matrices, nor the triangular packed matrices (TP) resulting from their factorizations (using $\approx n^2/2$ instead of n^2 storage); these have been requested by users. The interesting question is what data structure to support. One possibility is recursive storage as discussed in Sec. 3 [35,2]. Alternatively the packed storage may be partially expanded into a 2D array in order to apply Level 3 BLAS (GEMM) efficiently. Some preliminary ScaLAPACK prototypes support packed storage for the Cholesky factorization and the symmetric eigenvalue problem [12]. (2) ScaLAPACK only offers limited support of band matrix storage and does not specifically take advantage of symmetry or triangular form (SB,HB,TB). (3) ScaLAPACK does not support data types for the standard (HS) or generalized (HG, TG) nonsymmetric EVDs; see further below.

The table below compares the available functions in LAPACK and ScaLAPACK. The relevant user interfaces ('drivers') are listed by subject and acronyms are used for the software in the respective libraries. The table also shows that in the ScaLAPACK library the implementation of some driver routines and their specialized computational routines are currently missing. The highest priority ones to include are marked "add". We also want expert drivers that compute error bounds.

Extending Current Functionality. We outline possible extensions of Sca/LAPACK functionality, motivated by users and research progress: (1) efficient

	LAPACK	SCALAPACK
Linear Equations	GESV (LU) POSV (Cholesky) SYSV (LDL^T)	PxGESV PxPOSV missing, add
Least Squares (LS)	GELS (QR) GELSY (QR w/pivoting) GELSS (SVD w/QR) GELSD (SVD w/D&C)	PxGELS missing missing missing
Generalized LS	GGLSE (GRQ) GGGLM (GQR)	missing missing
Symmetric EVD	SYEV (QR) SYEVD (D&C) SYEVR (RRR)	PxSYEV PxSYEVD missing, add
Nonsymmetric EVD	GEES (HQR) GEEV (HQR + vectors)	missing driver, add missing driver, add
SVD	GESVD (QR) GESDD (D&C)	PxGESVD (missing complex C/Z) missing
Generalized Symmetric EVD	SYGV (inverse iteration) SYGVD (D&C)	PxSYGVX missing, add
Generalized Nonsymmetric EVD	GGES (HQZ) GGEV (HQZ + vectors)	missing, add missing, add
Generalized SVD	GGSVD (Jacobi)	missing

updating of factorizations like Cholesky, LDL^T, LU and QR, either using known unblocked techniques [38,26] or more recent blocked ones; (2) an $O(n^2)$ eigenvalue routine for companion matrices, i.e. to find roots of polynomials, which would replace the `roots()` function in Matlab, based on recent work of Gu, Bini and others on semiseparable matrices [20,66,10]; (3) recent structure-preserving algorithms for matrix polynomial eigenvalue problems, especially quadratic eigenvalue problems [63]; (4) new algorithm for matrix functions like the square root, exponential and sign function [23]; (5) algorithms for various Sylvester-type matrix equations (recursive, RECSY; parallel, SCASY) [48,49,40]. (6) product and quotient eigenvalue algorithms now in SLICOT [8] are being considered, using the improved underlying EVD algorithms; and (7) out-of-core algorithms [12,28,27].

5 Software

Improving Ease of Use. "Ease of use" can be classified as follows: ease of programming (which includes easy conversion from serial to parallel, from LAPACK to ScaLAPACK and the possiblity to use high level interfaces), ease of obtaining predictable results in dynamic environments (for debugging and performance), and ease of installation (including performance tuning).

There are tradeoffs involved in each of these subgoals. In particular, ultimate ease of programming, exemplified by typing $x = A\backslash b$ in order to solve $Ax = b$

(paying no attention to the data type, data structure, memory management or algorithm choice) requires an infrastructure and user interface best left to the builders of systems like MATLAB and may come at a significant performance and reliability penalty. In particular, many users now exercise, and want to continue to exercise, detailed control over data types, data structures, memory management and algorithm choice, to attain both peak performance and reliability (e.g., not running out of memory unexpectedly). But some users also would like Sca/LAPACK to handle workspace allocation automatically, make it possible to call Sca/LAPACK on a greater variety of user-defined data structures, and pick the best algorithm when there is a choice.

To accomodate these "ease of programming" requests as well as requests to make the Sca/LAPACK code accessible from other languages than Fortran, the following steps are considered: (1) Produce new F95 modules for the LAPACK drivers, for workspace allocation and algorithm selection. (2) Produce new F95 modules for the ScaLAPACK drivers, which convert, if necessary, the user input format (e.g., a simple block row layout across processors) to the optimal one for ScaLAPACK (which may be a 2D block cyclic layout with block sizes that depend on the matrix size, algorithm and architecture). Allocate memory as needed. (3) Produce LAPACK and ScaLAPACK wrappers in other languages. Based on current user surveys, these languages will tentatively be C, C++, Python and MATLAB. See below for software engineering details.

Ease of conversion from serial code (LAPACK) to parallel code (ScaLAPACK) is done by making the interfaces (at least at the driver level) as similar as possible. This includes expanding ScaLAPACK's functionality to include as much of LAPACK as possible (see Section 4).

Obtaining predictable results in a dynamic environment is important for debugging (to get the same answer when the code is rerun), for reproducibility, auditability (for scientific or legal purposes), and for performance (so that runtimes do not vary widely and unpredictably). Reproducibility in the face of asynchronous algorithms and heterogeneous systems will come with a performance penalty but is important for debugging and when auditability is critical.

To ease installation, we will use tools like autoconf and automatic performance tuning, supporting users from those who want to download and use one routine as quickly and simply as possible, to those who want an entire library, and to test and performance tune it carefully.

Improved Software Engineering. We describe our software engineering (SWE) approach. The main goals are to keep the substantial code base maintainable, testable and evolvable into the future as architectures and languages change. Maintaining compatibility with other software efforts and encouraging 3rd party contributions to the efforts of the Sca/LAPACK team are also goals [51].

These goals involve tradeoffs. One could explore starting "from scratch", using higher level ways to express the algorithms from which specific implementations could be generated. This approach yields high flexibility allowing the generation of code that is optimized for future computing platforms with different layers of parallelism, different memory hierarchies, different ratios of computation rate to bandwidth to latency, different programming languages and compilers, etc. Indeed, one can think of the problem as implementing the following *meta-program*:

```
(1) for all linear algebra problems
    (linear systems, eigenproblems, ...)
(2)   for all matrix types (general, symmetric, banded, ...)
(3)     for all data types (real/complex, different precisions)
(4)       for all machine architectures, communication topologies
(5)         for all programming interfaces
(6)           provide the best algorithm(s) available in terms of
              performance and accuracy (``algorithms'' is plural
              because sometimes no single one is always best)
```

This potential scope is quite large, requiring a judicious mixture of prioritization and automation. Indeed, there is prior work in automation [42], but so far this work has addressed only part of the range of algorithmic techniques Sca/LAPACK needs (e.g., not eigenproblems), it may not easily extend to more asynchronous algorithms and still needs to be coupled to automatic performance tuning techniques. Still, some phases of the meta-program are at least partly automatable now, namely steps (3) through (5) (see below).

Note that line (5) of the meta-program is "programming interfaces" not "programming languages," because the question of the best implementation language is separate from providing ways to call it (from multiple languages). Currently Sca/LAPACK is written in F77. Over the years, the Sca/LAPACK team and others have built on this to provide interfaces or versions in other languages: LAPACK95 [6] and LAPACK3E [3] for F95 (LAPACK3E providing a straightforward wrapper, and LAPACK95 using F95 arrays to simplify the interfaces at some memory and performance costs), CLAPACK in C [21] (translated, mostly automatically, using f2c [36]), LAPACK++ [29], TNT [64] in C++, and JLAPACK in Java [47] (translated using f2j).

First we summarize the SWE development plan and then the SWE research plan. The *development plan* includes (1) maintaining the core in Fortan, adopting those features of F95 that most improve ease-of-use and ease-of-development (recursion, modules, environmental enquiries) but do not prevent the most demanding users from attaining the highest performance and reliable control over the run-time environment (so not automatic memory allocation). Keeping Fortran is justified for cost and continuity reasons, as well as the fact that the most effective optimizing compilers still work best on Fortran, even when they share "back ends" with the C compiler, because of the added difficulty of discerning the absence of aliasing in C [22]; (2) F95 wrappers for the drivers to improve ease of use, via automatic workspace allocation and automatic algorithm selection; (3) F95 wrappers for the drivers that use performance models to determine the best layout for the user's matrix (which may be 2D block-cyclic and/or recursive instead of the 1D blocked layouts most natural to users) and which convert to that layout and back invisibly to the user; (4) wrappers for the drivers in other languages, like C, Python and Matlab; (5) using the new BLAS standard [15,13,5], for new Sca/LAPACK routines, which also provides new high precision functionality needed for iterative refinement [24], and systematically ensuring thread-safety; (6) using tools like autoconf, bugzilla, svn, automatic overnight build and test, etc. to streamline installation and development and encourage third party contributions, and using modules to provide extra precise versions based on one source; and (7) doing systematic performance tuning, not just the BLAS [69,68] and the BLACS [54,65], but the 1300 calls to the ILAENV routine in LAPACK that provides various blocking parameters, and exploiting modeling techniques that let the user choose how much tuning effort to expend [67,56].

The following are *SWE research tasks*: (1) exploring the best mappings of the Sca/LAPACK software layers (BLAS, BLACS, PBLAS, LAPACK, ScaLAPACK) to the hardware layers of emerging architectures, including deciding that different layers entirely are needed; (2) exploring the use of new parallel programming languages being funded by NSF, DOE and the DARPA HPCS program, namely UPC [19], Titanium [71], CAF [55], Fortress [1], X10 [60] and Cascade [18]; (3) exploring further automation of the production of library software (existing work such as [42] still needs to address two-sided factorizations, iterative algorithms for the EVD and SVD, more asynchronous algorithms, novel data layouts, how multiple levels of parallelism map to multiple levels of hardware, and so on); (4) using statistical models to accelerate performance tuning at installation time [67]; and (5) choosing the right subset of a cluster to run on at run-time, depending on the dynamically changing load.

Multicore and Multithreading. Message passing as used in ScaLAPACK introduces memory overhead unnecessary on multicore platforms, degrading performance and making it harder to schedule potentially parallel operations. Limiting shared memory parallelism to fork-join parallelism (e.g., OpenMP) or the BLAS is inadequate. Shared memory would let us replace data partitioning by work partitioning, although cache locality requirements mean we still need either two dimensional block cyclic (or perhaps recursive) layouts.

Shared memory systems have used client/server, work-crew, and pipelining for parallelism, but because of data dependencies pipelining is most appropriate, as well as being a good match for streaming hardware like the IBM Cell. For efficiency we must avoid pipeline stalls when data dependencies block execution.

We illustrate this for LU factorization. LU has left-looking and right-looking formulations [30]. Transition between the two can be done by automatic code transformations [53], although more than simple dependency analysis is needed. Lookahead can improve performance by performing panel factorizations in parallel with updates to the trailing matrix from the previous algorithm steps [62]. The lookahead can be of arbitrary depth (as exploited by the LINPACK benchmark [31]).

In fact lookahead provides a spectrum of implementations from right-looking (no lookahead) to left-looking (maximum lookahead). Less lookahead avoids pipeline stalls at the beginning of the factorization, but may introduce them at the end; more lookahead provides more work at the end of the factorization but may stall at the beginning.

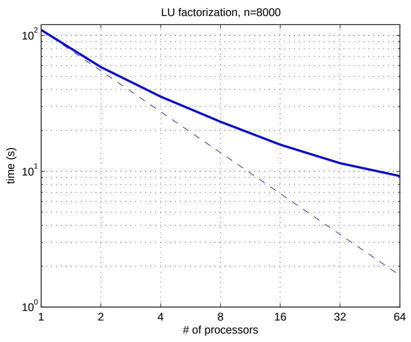
Recent experiments show that pipeline stalls can be greatly reduced if unlimited lookahead is allowed and the lookahead panel factorizations are dynamically scheduled, which is much easier in shared memory than distributed memory, in part because there is no storage overhead.

6 Performance

We give a few recent performance results for ScaLAPACK driver routines on recent architectures. We discuss strong scalability, i.e. we keep the problem size constant while increasing the number of processors.

Figure 1(left) gives the time to solve $Ax = b$ for $n = 8,000$ on a cluster of dual processor 64 bit AMD Opterons with a Gigabit ethernet. As the number of processors increases from 1 to 64, the time decreases from 110 sec (3.1 GFlops) to 9 sec (37.0 GFlops) (thanks to Emmanuel Jeannot for sharing the result.)

Figure 1(right) gives the time for the symmetric eigendecomposition with $n = 12,000$ on a 64 processor cluster of dual processor 64 bit Intel Xeon EMTs with a Myrinet MX interconnect. The matrices are generated randomly using the same generator as in the Linpack Benchmark, so there are no tight eigenvalue clusters.



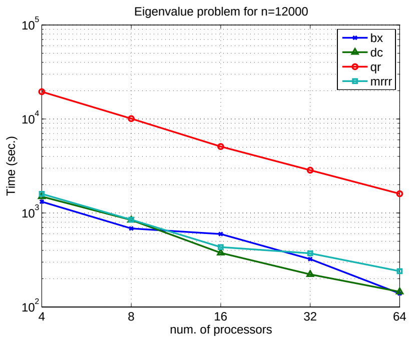


Fig. 1. Left: Scalability of ScaLAPACK's LU (`pdgetrf`) for $n = 8,000$. Right: Scalability of the ScaLAPACK's symmetric eigensolvers with $n = 12,000$. Four eigensolvers are shown: BX (`pdsyevx`), QR (`pdsyev`), DC (`pdsyevd`) and MRRR (`pdsyevr`).

References

1. Steele, A., et al.: The Fortress language specification, version 0.707, `research.sun.com/projects/plrg/fortress0707.pdf`
2. Andersen, B.S., Wazniewski, J.: A recursive formulation of Cholesky factorization of a matrix in packed storage. ACM Trans. Math. Soft. 27(2), 214–244 (2001)

3. Anderson, E.: LAPACK3E (2003), http://www.netlib.org/lapack3e
4. Ashcraft, C., Grimes, R.G., Lewis, J.G.: Accurate symmetric indefinite linear equation solvers. SIAM J. Matrix Anal. Appl. 20(2), 513–561 (1998)
5. Bailey, D., Demmel, J., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S., Kapur, A., Li, X., Martin, M., Thompson, B., Tung, T., Yoo, D.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Soft. 28(2), 152–205 (2002)
6. Barker, V., Blackford, S., Dongarra, J., Du Croz, J., Hammarling, S., Marinova, M., Wasniewski, J., Yalamov, P.: LAPACK95 Users' Guide. SIAM (2001), http://www.netlib.org/lapack95
7. Barlow, J., Bosner, N., Drmač, Z.: A new stable bidiagonal reduction algorithm (2004), www.cse.psu.edu/~barlow/fastbidiag3.ps
8. Benner, P., Mehrmann, V., Sima, V., Van Huffel, S., Varga, A.: SLICOT - a subroutine library in systems and control theory. Applied and Computational Control, Signals, and Circuits 1, 499–539 (1999)
9. Bientinisi, P., Dhillon, I.S., van de Geijn, R.: A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. Technial Report TR-03-26, Computer Science Dept., University of Texas (2003)
10. Bini, D., Eidelman, Y., Gemignani, L., Gohberg, I.: Fast QR algorithms for Hessenberg matrices which are rank-1 perturbations of unitary matrices. Dept. of Mathematics report 1587, University of Pisa, Italy (2005), http://www.dm.unipi.it/~gemignani/papers/begg.ps
11. Bischof, C.H., Lang, B., Sun, X.: A framework for symmetric band reduction. ACM Trans. Math. Soft. 26(4), 581–601 (2000)
12. Blackford, L.S., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D.W., Whaley, R.C.: Scalapack prototype software. Netlib, Oak Ridge National Laboratory (1997)
13. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of Basic Linear Algebra Subroutines (BLAS). ACM Trans. Math. Soft., 28(2) (June 2002)
14. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C., Maany, Z., Krough, F., Corliss, G., Hu, C., Keafott, B., Walster, W., Gudenberg, J.W.v.: Basic Linear Algebra Subprograms Techical (BLAST) Forum Standard. Intern. J. High Performance Comput. 15(3-4) (2001)
15. Blackford, S., Corliss, G., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Hu, C., Kahan, W., Kaufman, L., Kearfott, B., Krogh, F., Li, X., Maany, Z., Petitet, A., Pozo, R., Remington, K., Walster, W., Whaley, C., Gudenberg, J.W.v., Lumsdaine, A.: Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. Intern. J. High Performance Comput. 15(3-4), 305 (2001), also available at www.netlib.org/blas/blast-forum/
16. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part I: Maintaining well-focused shifts and Level 3 performance. SIAM J. Matrix Anal. Appl. 23(4), 929–947 (2001)
17. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part II: Aggressive early deflation. SIAM J. Matrix Anal. Appl. 23(4), 948–973 (2001)

18. Callahan, D., Chamberlain, B., Zima, H.: The Cascade high-productivity language. In: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), pp. 52–60. IEEE Computer Society Press, Los Alamitos (2004), `www.gwu.edu/~upc/publications/productivity.pdf`
19. Cantonnet, F., Yao, Y., Zahran, M., El-Ghazawi, T.: Productivity analysis of the UPC language. In: IPDPS 2004 PMEO workshop (2004), `www.gwu.edu/~upc/publications/productivity.pdf`
20. Chandrasekaran, S., Gu, M.: Fast and stable algorithms for banded plus semiseparable systems of linear equations. SIAM J. Matrix Anal. Appl. 25(2), 373–384 (2003)
21. CLAPACK: LAPACK in C, `http://www.netlib.org/clapack/`
22. Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Chavarria-Miranda, D., Contonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y.: An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In: Proc. 10th ACM SIGPLAN Symp. on Principles and Practice and Parallel Programming (PPoPP 2005), ACM Press, New York (2005), `www.hipersoft.rice.edu/caf/publications/index.html`
23. Davies, P., Higham, N.J.: A Schur-Parlett algorithm for computing matrix functions. SIAM J. Matrix Anal. Appl. 25(2), 464–485 (2003)
24. Demmel, J., Hida, Y., Kahan, W., Li, X.S., Mukherjee, S., Riedy, E.J.: Error bounds from extra precise iterative refinement. ACM TOMS 32(2), 325–351 (2006)
25. Dhillon, I.S.: Reliable computation of the condition number of a tridiagonal matrix in O(n) time. SIAM J. Matrix Anal. Appl. 19(3), 776–796 (1998)
26. Dongarra, J., Bunch, J., Moler, C., Stewart, G.W.: LINPACK User's Guide. SIAM, Philadelphia, PA (1979)
27. Dongarra, J., D'Azevedo, E.: The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. Computer Science Dept. Technical Report CS-97-347, University of Tennessee, Knoxville, TN (January 1997), `http://www.netlib.org/lapack/lawns/lawn118.ps`
28. Dongarra, J., Hammarling, S., Walker, D.: Key concepts for parallel out-of-core LU factorization. Computer Science Dept. Technical Report CS-96-324, University of Tennessee, Knoxville, TN (April 1996), `www.netlib.org/lapack/lawns/lawn110.ps`
29. Dongarra, J., Pozo, R., Walker, D.: Lapack++: A design overview of ovject-oriented extensions for high performance linear algebra. In: Supercomputing 93, IEEE Computer Society Press, Los Alamitos (1993), `math.nist.gov/lapack++`
30. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia, PA (1998)
31. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency Computat.: Pract. Exper. 15, 803–820 (2003)
32. Dopico, F.M., Molera, J.M., Moro, J.: An orthogonal high relative accuracy algorithm for the symmetric eigenproblem. SIAM. J. Matrix Anal. Appl. 25(2), 301–351 (2003)
33. Drmač, Z., Veselić, K.: New fast and accurate Jacobi SVD algorithm. Technical report, Dept. of Mathematics, University of Zagreb (2004)
34. Duff, I.S., Vömel, C.: Incremental Norm Estimation for Dense and Sparse Matrices. BIT 42(2), 300–322 (2002)
35. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46(1), 3–45 (2004)

36. f2c: Fortran-to-C translator, `http://www.netlib.org/f2c`
37. Fulton, C., Howell, G., Demmel, J., Hammarling, S.: Cache-efficient bidiagonalization using BLAS 2.5 operators, p. 28 (2004) (in progress)
38. Golub, G., Van Loan, C.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
39. Graham, S., Snir, M., Patterson, C. (eds.): Getting up to Speed: The Future of Supercomputing. National Research Council (2005)
40. Granat, R., Jonsson, I., Kågström, B.: Combining Explicit and Recursive Blocking for Solving Triangular Sylvester-Type Matrix Equations in Distrubuted Memory Platforms. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 742–750. Springer, Heidelberg (2004)
41. Grosser, B.: Ein paralleler und hochgenauer O(n^2) Algorithmus für die bidiagonale Singulärwertzerlegung. PhD thesis, University of Wuppertal, Wuppertal, Germany (2001)
42. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal Linear Algebra Methods Environment. ACM Trans. Math. Soft. 27(4), 422–455 (2001)
43. Hargreaves, G.I.: Computing the condition number of tridiagonal and diagonal-plus-semiseparable matrices in linear time. Technical Report submitted, Department of Mathematics, University of Manchester, Manchester, England (2004)
44. Higham, N.J.: Analysis of the Cholesky decomposition of a semi-definite matrix. In: Cox, M.G., Hammarling, S. (eds.) Reliable Numerical Computation, ch. 9, pp. 161–186. Clarendon Press, Oxford (1990)
45. High productivity computing systems (hpcs), `http://www.highproductivity.org`
46. IEEE Standard for Binary Floating Point Arithmetic Revision (2002), `grouper.ieee.org/groups/754`
47. JLAPACK: LAPACK in Java, `http://icl.cs.utk.edu/f2j`
48. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems. I. one-sided and coupled Sylvester-type matrix equations. ACM Trans. Math. Software 28(4), 392–415 (2002)
49. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems. II. Two-sided and generalized Sylvester and Lyapunov matrix equations. ACM Trans. Math. Software 28(4), 416–435 (2002)
50. Kågström, B., Kressner, D.: Multishift Variants of the QZ Algorithm with Aggressive Early Deflation. SIAM J. Matrix Anal. Appl. 29(1), 199–227 (2006)
51. LAPACK Contributor Webpage, `http://www.netlib.org/lapack-dev/contributions.html`
52. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Soft. 28(2), 152–205 (2002)
53. Menon, V., Pingali, K.: Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring. Int. J. Parallel Comput. 32(6), 501–523 (2004)
54. Nishtala, R., Chakrabarti, K., Patel, N., Sanghavi, K., Demmel, J., Yelick, K., Brewer, E.: Automatic tuning of collective communications in MPI. In: Poster at SIAM Conf. on Parallel Proc., San Francisco, `www.cs.berkeley.edu/~rajeshn/poster_draft_6.ppt`
55. Numrich, R., Reid, J.: Co-array Fortran for parallel programming. Fortran Forum, 17 (1998)

56. OSKI: Optimized Sparse Kernel Interface, `http://bebop.cs.berkeley.edu/oski/`
57. Parlett, B.N., Dhillon, I.S.: Orthogonal eigenvectors and relative gaps. SIAM J. Matrix Anal. Appl. 25(3), 858–899 (2004)
58. Parlett, B.N., Vömel, C.: Tight clusters of glued matrices and the shortcomings of computing orthogonal eigenvectors by multiple relatively robust representations. University of California, Berkeley (2004) (In preparation)
59. Ralha, R.: One-sided reduction to bidiagonal form. Lin. Alg. Appl. 358, 219–238 (2003)
60. Saraswat, V.: Report on the experimental language X10, v0.41. IBM Research technical report (2005)
61. Slapničar, I.: Highly accurate symmetric eigenvalue decomposition and hyperbolic SVD. Lin. Alg. Appl. 358, 387–424 (2002)
62. Strazdins, P.E.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Int. J. Parallel Distrib. Systems Networks 4(1), 26–35 (2001)
63. Tisseur, F., Meerbergen, K.: A survey of the quadratic eigenvalue problem. SIAM Review 43, 234–286 (2001)
64. TNT: Template Numerical Toolkit, `http://math.nist.gov/tnt`
65. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Towards an accurate model for collective communications. Intern. J. High Perf. Comp. Appl., special issue on Performance Tuning 18(1), 159–167 (2004)
66. Vandebril, R., Van Barel, M., Mastronardi, M.: An implicit QR algorithm for semiseparable matrices to compute the eigendecomposition of symmetric matrices. Report TW 367, Department of Computer Science, K.U. Leuven, Leuven, Belgium (2003)
67. Vuduc, R., Demmel, J., Bilmes, J.: Statistical models for automatic performance tuning. In: Intern. Conf. Comput. Science (May 2001)
68. Whaley, R.C., Dongarra, J.: The ATLAS WWW home page, `http://www.netlib.org/atlas/`
69. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing 27(1-2), 3–25 (2001)
70. Willems, P.: personal communication (2006)
71. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performnace Java dialect. Concurrency: Practice and Experience 10, 825–836 (1998)

Large-Scale Methods in Image Deblurring

Per Christian Hansen[1] and Toke Koldborg Jensen[2]

[1] Informatics and Mathematical Modelling,
Technical University of Denmark, DK-2800 Lyngby, Denmark
pch@imm.dtu.dk
[2] TNM Consult, Marielundvej 48, DK-2730 Herlev, Denmark

Abstract. We use the two-dimensional DCT to study several properties of reconstructed images computed by regularizing iterations, that is, Krylov subspace methods applied to discrete ill-posed problems. The regularization in these methods is obtained via the projection onto the associated Krylov subspace. We focus on CGLS/LSQR, GMRES, and RRGMRES, as well as MINRES and MR-II in the symmetric case.

1 Introduction

In the setting of matrix computations, the model for the blurring of the image is $A\,x = b$, where the vectors x and b represent the exact and blurred images, and the matrix A represents the blurring process. Since image deblurring is a discrete ill-posed problem, it is necessary to use regularization in order to compute stable solutions [6]. Moreover, it is often advantageous to impose boundary conditions on the reconstruction, which is achieved by a simple modification of the coefficient matrix [9], [12].

This paper focuses on *regularizing iterations* where we apply a Krylov subspace method directly to the problem $Ax = b$. The regularization comes from the projection of the solution on the Krylov subspace associated with the method, and the number of iterations plays the role of the regularization parameter.

We use the two-dimensional discrete cosine transform (2D-DCT) to perform a spectral analysis of the solutions to the image deblurring problem, computed by means of regularizing iterations, and we focus on CGLS/LSQR and GMRES and their variants MINRES, RRGMRES and MR-II. To the best of our knowledge, a thorough study of the spectral and visual quality of the reconstructions computed by these methods has not been carried out.

2 The Image Deblurring Problem and Its Regularization

Underlying the image deblurring problem is a 2D Fredholm integral equation of the first kind which has the generic form

$$\int_0^1 \int_0^1 K(s,t;x,y)\, f(x,y)\, dx\, dy = g(s,t), \qquad 0 \le s,t \le 1 \tag{1}$$

where f and g represent the exact and blurred images, and the kernel K is the *point spread function* (PSF) for the blurring. Our work here is restricted to square $n \times n$ images, and to the case where the PSF is spatially invariant and separates in the variables, i.e., $K(s,t,;x,y) = k_{\rm c}(s-x)\,k_{\rm r}(t-y)$ where $k_{\rm c}$ and $k_{\rm r}$ are given functions. More general PSFs can be studied by the same approach, at the cost of a much higher computational complexity, but the separable PSFs suffice to illustrate our points.

Discretization of the integral equation (1) then leads to the linear model

$$A_{\rm c}\,X\,A_{\rm r}^T = B \tag{2}$$

in which X and B are the exact and blurred images, and the two $n \times n$ Toeplitz matrices $A_{\rm c}$ and $A_{\rm r}$ perform blurring in the direction of the columns and rows of the image, respectively. By introducing the vectors $x = \mathrm{vec}\,(X)$ and $b = \mathrm{vec}\,(B)$, where $\mathrm{vec}\,(\cdot)$ stacks the columns of the matrix, we can rewrite the above system in the form $A\,x = b$, in which the *PSF matrix* A is the $n^2 \times n^2$ Kronecker product

$$A = A_{\rm r} \otimes A_{\rm c}. \tag{3}$$

We emphasize that while the Kronecker-product system is useful for studying the discrete ill-posed problem, all computations are performed with the system in (2). See, e.g., [7] for details about deconvolution algorithms.

In addition to the above choice of PSF model, we assume that the noise in the discrete problem appears as additive Gaussian noise in the right-hand side,

$$B = A_{\rm c}\,X\,A_{\rm r}^T + E,$$

where X is the exact image, and the elements of the noise matrix E are statistically independent, uncorrelated with X, and coming from a normal distribution with zero mean and standard deviation η.

The Kronecker form of the PSF matrix (3) lets us compute the SVD of large matrices, due to the fact that given the SVDs of the two matrices $A_{\rm c}$ and $A_{\rm r}$,

$$A_{\rm c} = U_{\rm c}\,\Sigma_{\rm c}\,V_{\rm c}^T, \qquad A_{\rm r} = U_{\rm r}\,\Sigma_{\rm r}\,V_{\rm r}^T,$$

we can write the SVD of the PSF matrix $A = A_{\rm r} \otimes A_{\rm c}$ as

$$A = U\,\Sigma\,V^T = \big((U_{\rm r} \otimes U_{\rm c})\,\Pi\big)\big(\Pi^T(\Sigma_{\rm r} \otimes \Sigma_{\rm c})\,\Pi\big)\big((V_{\rm r} \otimes V_{\rm c})\,\Pi\big)^T, \tag{4}$$

where the $n^2 \times n^2$ permutation matrix Π ensures that the diagonal elements of $\Pi^T(\Sigma_{\rm r} \otimes \Sigma_{\rm c})\,\Pi$ appear in decreasing order. We emphasize that our analysis of the iterative methods is not restricted to Kronecker products – it holds for all PSF matrices.

It is well know that the SVD of the coefficient matrix A provides a natural basis for the study of regularization methods. Many regularization methods, including regularizing iterations, lead to regularized solutions in the form

$$x_{\rm reg} = \sum_{k=1}^{n^2} \phi_i\,\frac{u_k^T b}{\sigma_k}\,v_k. \tag{5}$$

These methods are often referred to as spectral filtering methods; we avoid this term, in order not to confuse it with our DCT-based spectral analysis.

With the notation $x_{\mathrm{reg}} = \mathrm{vec}\,(X_{\mathrm{reg}})$, (5) immediately leads to the expression

$$X_{\mathrm{reg}} = \sum_{k=1}^{n^2} \phi_k \, \frac{u_k^T b}{\sigma_k} \, V^{[k]}, \tag{6}$$

where ϕ_k are the filter factors, σ_k are the singular values of A, u_k are the left singular vectors, and $V^{[k]}$ are $n \times n$ matrices such that $v_k = \mathrm{vec}\big(V^{[k]}\big)$ are the right singular vectors. This relation shows that we can express the regularized solution X_{reg} as a weighted sum over the *basis images* $V^{[k]}$.

With the Kronecker-product form of the PSF matrix A, Eq. (4) shows that there are simple expressions for the singular values and vectors. If $\sigma_{\mathrm{r}i}$ and $\sigma_{\mathrm{c}j}$ are the singular values of A_{r} and A_{c}, then their products are the singular values of A. Moreover, if $u_{\mathrm{r}i}$, $u_{\mathrm{c}j}$, $v_{\mathrm{r}i}$ and $v_{\mathrm{c}j}$ are the left and right singular vectors of A_{r} and A_{c}, then the left and right singular vectors of A are $u_{\mathrm{c}j} \otimes u_{\mathrm{r}i}$ and $v_{\mathrm{c}j} \otimes v_{\mathrm{r}i}$, respectively. Then (6) takes the form

$$X_{\mathrm{reg}} = \sum_{i=1}^{n} \sum_{j=1}^{n} \phi_{ij} \, \frac{u_{\mathrm{r}i}^T \, B \, u_{\mathrm{c}j}}{\sigma_{\mathrm{r}i} \, \sigma_{\mathrm{c}j}} \, V^{[ij]}$$

where ϕ_{ij} is the filter factor associated with the product $\sigma_{\mathrm{r}i} \, \sigma_{\mathrm{c}j}$, and the basis images are given by $V^{[ij]} = v_{\mathrm{r}i} \, v_{\mathrm{c}j}^T$.

3 Spectral Properties of the PSF Matrix

The two-dimensional discrete cosine transform (2D-DCT) is a simple frequency transform that is often used in image processing. The transformed image $\widehat{X}$ is given by

$$\widehat{X} = \mathcal{C} \, X \, \mathcal{C}^T,$$

where $\mathcal{C}$ is an orthogonal matrix that represents the one-dimensional DCT. The elements of $\mathcal{C} \in \mathbb{R}^{n \times n}$ are given by

$$\mathcal{C}_{ij} = \begin{cases} \sqrt{1/n} & i = 1 \\ \sqrt{2/n} \, \cos\big(\pi(i-1)(2j-1)/(2n)\big), & i > 1. \end{cases}$$

The 2D-DCT transformed image $\widehat{X}$ provides a frequency representation of the image X, where each element $\hat{x}_{ij}$ corresponds to a specific basis image (see e.g, [11, p. 136]). The element $\hat{x}_{11}$ represents a constant (the DC term), and the elements $\hat{x}_{1j}$ and $\hat{x}_{i1}$ correspond to simple cosine waves of varying frequency in horizontal or vertical direction over the entire image. The remaining elements of $\widehat{X}$ represent combinations of frequencies in the two directions. The lowest spatial frequencies are represented in one corner, and the highest in the opposite corner.

For one-dimensional discrete ill-posed problems arising from discretizations of Fredholm integral equations, we know from the analysis in [8] that the singular vectors u_i and v_i of the coefficient matrix A tend to have an increasing number of sign changes in their elements as the index i increases. I.e., the smaller the singular value σ_i, the more high-frequent the appearance of the corresponding singular vectors u_i and v_i.

In the two-dimensional case we expect a similar behavior, but the concept of frequency is more complicated because the singular vectors now correspond to two-dimensional basis images. The correct way to study the spectral behavior of the singular vectors is therefore to study the two-dimensional spectral behavior of the basis images $V^{[k]}$, e.g., by means of the 2D-DCT. We need to sort the latter basis images according to decreasing singular values $\sigma_{\mathrm{r}i}\,\sigma_{\mathrm{c}j}$ using the permutation matrix Π from Eq. (4); the sorted basis images are then equal to $V^{[k]}$, $k = 1, 2, \ldots, n^2$.

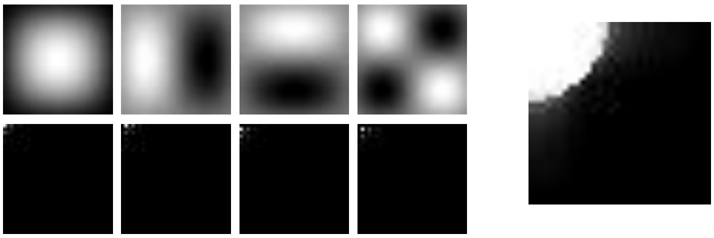


Fig. 1. Top left: The four basis images $V^{[k]}$, $k = 1, 2, 3, 4$ for the PSF matrix for isotropic blur. Bottom left: the corresponding $|\widehat{V}^{[k]}|$. Right: Plot of $(\sum_{i=1}^{150} |\widehat{V}^{[i]}|^2)^{1/2}$; the main contribution is located in a sphere in the upper left corner.

We now construct two coefficient matrices $A \in \mathbb{R}^{1024\times1024}$ and $\widetilde{A} \in \mathbb{R}^{1024\times1024}$, both being Kronecker products of two 32×32 Toeplitz matrices. The matrix A describes isotropic blurring while $\widetilde{A}$ describes non-isotropic blurring. The first four basis images $V^{[1]}$ to $V^{[4]}$ of A are shown in Fig. 1, together with $|\widehat{V}^{[1]}|$ to $|\widehat{V}^{[4]}|$ which denote the absolute values of their 2D-DCT. The rightmost picture in Fig. 1 shows the "RMS image" $\left(\sum_{i=1}^{150} |\widehat{V}^{[i]}|^2\right)^{1/2}$ of the first 150 basis images, which illustrates all the dominating frequency components in the first 150 basis images. We see that the main contributions lie in the upper left corner of the spectrum; thus all the first basis images are low-frequent.

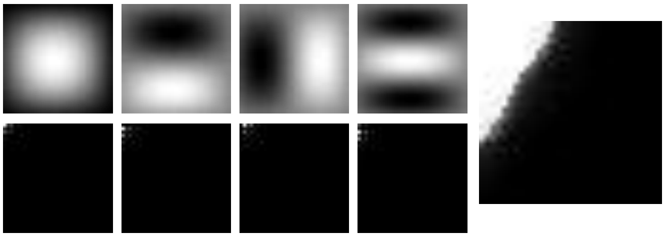


Fig. 2. Similar to Fig. 1, but for the coefficient matrix $\widetilde{A}$ for non-isotropic blur. Frequencies in one direction appear before similar frequencies in the other direction.

Figure 2 shows the first four basis images of $\widetilde{A}$ for non-isotropic blur. We see that frequencies in one direction tend to appear before the corresponding frequencies in the other direction in the image. This is clearly seen in the fourth basis image $V^{[4]}$ which is dominated by the 2D-DCT component $\hat{v}_{3,2}^{[4]}$ – while in the isotropic case, $V^{[4]}$ is dominated by $\hat{v}_{2,2}^{[4]}$ We also see from the plot of the "RMS image" $\left(\sum_{i=1}^{150} |\widehat{V}^{[i]}|^2\right)^{1/2}$ that the frequency contents is no longer confined to a spherical region, but the main contributions are still low-frequent.

We emphasize that the boundary conditions influence the basis images, see [9], but the frequency decomposition property is similar for all boundary conditions. In this work we use zero boundary conditions, and all basis images are therefore zero at the border. Other choices of boundary conditions, such as periodic and reflexive boundary conditions, lead to a different behavior of the basis images at the boundaries; but discussions of these issues are outside the scope of this paper.

4 Krylov Subspace Methods

The properties of Krylov subspaces have been studied extensively in the literature [10] and they form the basic element for understanding the intrinsic regularizing properties of LSQR, GMRES, etc.

4.1 The CGLS and LSQR Algorithms

The two algorithms CGLS and LSQR, which are designed to solve the least squares problem $\min \|Ax - b\|_2$, are mathematically equivalent; we use the LSQR formulation here because it better allows us to carry out the desired spectral analysis. Both methods work implicitly with the normal equations $A^TAx = A^Tb$ and are thus based on the Krylov subspace

$$\mathcal{K}_k\left(A^TA, A^Tb\right) = \operatorname{span}\left\{A^Tb, (A^TA)A^Tb, \ldots, (A^TA)^{k-1}A^Tb\right\}. \tag{7}$$

LSQR constructs explicit basis vectors for this Krylov subspace via the Lanczos bidiagonalization algorithm, which gives the partial decomposition

$$AV_k = U_{k+1}B_k, \tag{8}$$

where $V_k \in \mathbb{R}^{n^2 \times k}$ is a matrix with orthonormal columns that span the Krylov subspace (7). The matrix $U_{k+1} \in \mathbb{R}^{n^2 \times (k+1)}$ also has orthonormal columns, and its first column is chosen as $u_1 = b/\|b\|_2$ which simplifies the implementation considerably. The matrix $B_k \in \mathbb{R}^{(k+1)\times k}$ is a lower bidiagonal matrix. The LSQR iterate $x^{(k)}$ minimizes the 2-norm of the residual in the Krylov subspace, i.e., $x^{(k)} = \operatorname{argmin}_x \|Ax - b\|_2$ s.t. $x \in \mathcal{K}_k\left(A^TA, A^Tb\right)$, and it follows that

$$x^{(k)} = V_k\xi_k, \qquad \xi_k = \operatorname{argmin}_{\xi\in\mathbb{R}^k} \|B_k\xi - \rho e_1\|_2, \tag{9}$$

where e_1 is the first canonical unit vector in $\mathbb{R}^{k+1}$ and $\rho = \|b\|_2$. This algorithm can be implemented using short recurrences and thus one can avoid storing the partial decomposition (8).

4.2 The GMRES Algorithm and Its Relatives

These algorithms are based on the Arnoldi process that iteratively constructs an orthonormal basis for the Krylov subspace

$$\mathcal{K}_k\left(A, b\right) = \operatorname{span}\left\{b, Ab, A^2 b, \ldots, A^{k-1} b\right\}, \tag{10}$$

and results in the partial decomposition

$$A\, W_k = W_{k+1} H_k, \qquad W_{k+1} = (\, W_k\, ,\, w_{k+1}\,), \tag{11}$$

where the columns of $W_k \in \mathbb{R}^{n^2 \times k}$ provide an orthonormal basis for the Krylov subspace in (10), and $H_k \in \mathbb{R}^{(k+1)\times k}$ is an upper Hessenberg matrix. The first column in W_k is again chosen as $w_1 = b/\|b\|_2$. If A is symmetric then H_k reduces to tridiagonal form, and the construction of the partial decomposition (11) can be done by a three-term recurrence as implemented in MINRES. In this case the solution can be updated without explicitly storing the partial decomposition. For general matrices, however, no such short recurrences exist and GMRES needs to save all the constructed Krylov vectors.

The GMRES/MINRES iterate $\hat{x}^{(k)}$ minimizes the residual norm with respect to the above Krylov subspace, i.e., $\hat{x}^{(k)} = \operatorname{argmin}\|b - Ax\|_2$ s.t. $x \in \mathcal{K}_k\left(A, b\right)$, which leads to the relation

$$\hat{x}^{(k)} = W_k \zeta_k, \qquad \zeta_k = \operatorname{argmin}_{\zeta \in \mathbb{R}^k} \|H_k \zeta - \rho e_1\|_2, \tag{12}$$

where again e_1 is the first canonical unit vector and $\rho = \|b\|_2$.

There exists a variant of GMRES that uses Ab as the starting vector for the Krylov subspace, instead of b as in (10). This gives the "shifted" Krylov subspace

$$\mathcal{K}_k\left(A, Ab\right) = \operatorname{span}\left\{Ab, A^2 b, A^3 b, \ldots, A^k b\right\}. \tag{13}$$

The algorithm MR-II [4], [5] is an efficient short-term recurrence implementation of this method for symmetric matrices. In the non-symmetric case the algorithm is called RRGMRES [1], [2]. The partial decomposition when using RRGMRES or MR-II is written as

$$A\, \widehat{W}_k = \widehat{W}_{k+1} \widehat{H}_k, \qquad \widehat{W}_{k+1} = (\, \widehat{W}_k\, ,\, \widehat{w}_{k+1}\,), \tag{14}$$

where $\widehat{W} \in \mathbb{R}^{n^2 \times k}$ provides a basis for (13).

4.3 Examples of Iterates

We illustrate the typical behavior of the iterative methods using two examples. The true image X is a part of the image Barbara of size 175×175. Two PSF matrices are used: a symmetric A for isotropic blur, and a non-symmetric $\widetilde{A}$ for non-isotropic blur. In both cases, we use additive white noise E, scaled such that $\|E\|_{\mathrm{F}}/\|B\|_{\mathrm{F}} = 0.05$. The true image and the two blurred and noisy images are shown in Fig. 3.

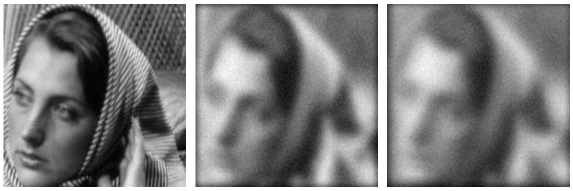


Fig. 3. Left: True image X. Middle: Blurred image B due to the symmetric PSF matrix A. Right: Blurred image B due to the non-symmetric PSF matrix $\widetilde{A}$.

Figure 4 shows the LSQR, MINRES, and MR-II solutions after 5, 10, and 25 iterations for the symmetric PSF matrix, and it is seen that the algorithms give very different solutions. The LSQR solutions slowly improve, but after 25 iterations some noise has appeared as small circular "freckles." The MINRES solutions get dominated very fast by high-frequency noise; with the noise level defined above, the iterates are clearly dominated by noise after only 5 iterations. The MR-II solutions show artifacts similar to the LSQR solutions, but the convergence seems faster as the solution after 10 MR-II iterations is visually similar to the LSQR solution after 25 iterations.

The solutions for the non-symmetric PSF matrix are also shown in Fig. 4. Here, the LSQR algorithm is again seen to generate some structured artifacts, clearly visible after 25 iterations. The artifacts are no longer circular, due to the non-isotropic blurring. But they are still band-limited and certainly not high-frequent as the noise seen in the GMRES solutions. The RRGMRES solutions again show artifacts similar to the artifacts for LSQR. But the difference in convergence speed is not as large as for LSQR and MR-II.

For a smaller noise level, GMRES and MINRES seem to exhibit faster convergence than LSQR, and converge to visually pleasing solutions as studied, e.g., in [3]. Figure 5 shows the 10th iteration of LSQR, MINRES, and MR-II for the symmetric blurring with the noise level reduced to $\|E\|_\mathrm{F}/\|B\|_\mathrm{F} = 5 \cdot 10^{-4}$. The MINRES solution visually gives the highest level of detail and it is at least as good as the MR-II solution. The LSQR solution is a bit more blurred, which indicates that MINRES, behind the noise, converges faster than LSQR.

5 Study of Solution and Noise Components

We now study how the regularization behaves and how it treats the wanted signal contents, as well as the additive noise. This analysis is only possible when the additive noise $e = \mathrm{vec}(E)$ is explicitly known, and we can split the noisy right-hand side as $b = b^\mathrm{exact} + e$ with $b^\mathrm{exact} = Ax^\mathrm{exact}$.

In the case of LSQR, the filter factors ϕ_k can be computed by a three-term recurrence [6, Theorem 6.3.2], [13]. But for GMRES, no short recurrence exists and the filters are are not simple to compute. Hence, to split the solution components, we must return to the basic relations for the algorithms (8), (11), and (14). We stress that MINRES and MR-II cannot be used in this connection,

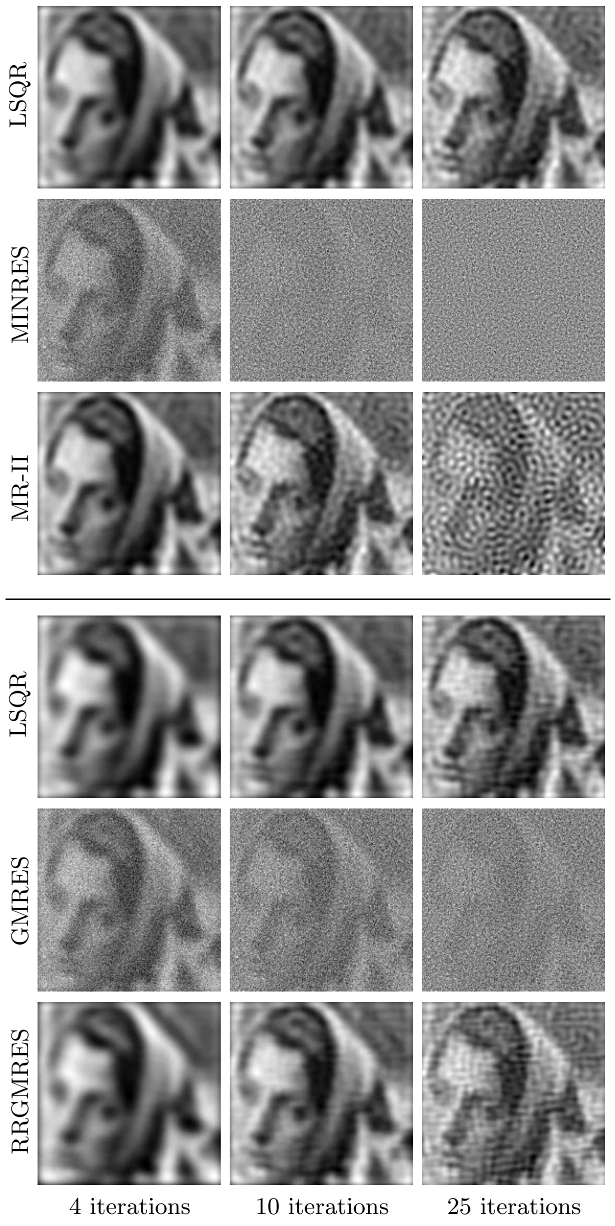


Fig. 4. Top: LSQR, MINRES, and MR-II iterates $x^{(k)}$, $\hat{x}^{(k)}$ and $\tilde{x}^{(k)}$ with the symmetric PSF matrix A, for $k = 5$, 10, and 20 iterations. Bottom: LSQR, GMRES, and RRGMRES iterates $x^{(k)}$, $\hat{x}^{(k)}$ and $\tilde{x}^{(k)}$ with the nonsymmetric PSF matrix $\widetilde{A}$, for $k = 5$, 10, and 25 iterations.

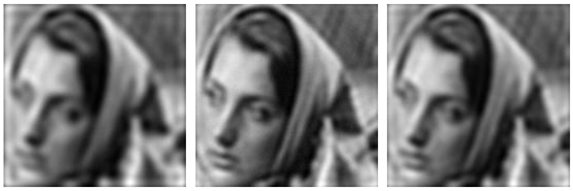


Fig. 5. LSQR, MINRES, and MR-II solutions for symmetric problem with noise level $\|E\|_F/\|B\|_F = 5 \cdot 10^{-4}$. The lower noise level makes the MINRES solution useful, and we see that the MINRES solution visually seems to have the highest level of details.

while the partial Arnoldi decompositions are explicitly needed; therefore GMRES and RRGMRES have been used in the following also for the symmetric problem.

Using the splitting of the right-hand side into the signal and noise components, for LSQR we have $x^{(k)} = x_b^{(k)} + x_e^{(k)}$ with

$$x_b^{(k)} = V_k B_k^\dagger U_{k+1} b^{\mathrm{exact}}, \quad x_e^{(k)} = V_k B_k^\dagger U_{k+1} e,$$

and similarly $\hat{x}_b^{(k)}$ and $\hat{x}_e^{(k)}$ for GMRES, and $\tilde{x}_b^{(k)}$ and $\tilde{x}_e^{(k)}$ for RRGMRES. This procedure enforces the solution components to lie in the correct Krylov subspaces, generated from A and b, and spanned by the columns of V_k, W_k, and $\widetilde{W}_k$.

These splittings are shown in Fig. 6 for LSQR, GMRES, and RRGMRES for the Barbara image. Both the symmetric and the non-symmetric PSF matrix are studied. We see how the noise propagates very differently due to the differences in the Krylov subspaces. The LSQR algorithm produces ringing effects in the symmetric case, and low-frequent but more unstructured ringing artifacts in the non-symmetric case. It is interesting to see how the ringing effects follow the contours of the image.

GMRES propagates the noise more or less as white noise for both PSF matrices. An interesting observation is that also the signal component is noisy, especially for the non-symmetric problem. This is caused by the fact that the noise is directly present in the Krylov basis from which the solutions are reconstructed.

The RRGMRES solution components behave much like the LSQR solution components, but appear to carry more details after the same number of iterations. After 10 iterations, the "freckles" are smaller in diameter, but more intense in the reconstructed images. The solution components of the non-symmetric problem are even more similar to the LSQR solution components. Avoiding the noisy right-hand side from the GMRES subspace is seen to improve the quality of the RRGMRES solutions.

To study the "freckles" in more detail, we study how the noise in the LSQR solutions behaves in the 2D-DCT domain. For the three LSQR solutions after 5, 10, and 25 iterations from Fig. 4, we show in Fig. 7 the error components $X_e^{(k)}$ and their 2D-DCT spectra. We see that the frequencies are primarily represented in

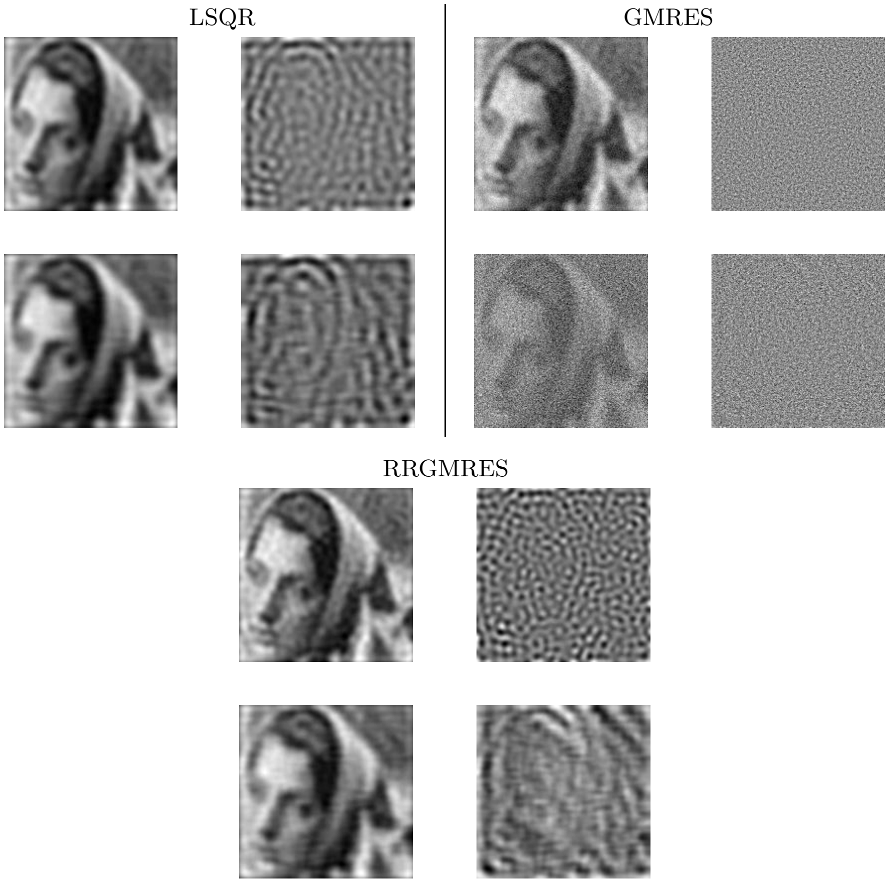


Fig. 6. For each method, we show the splitting of the solution after 10 iterations into the signal part (left) and the noise part (right), for both the symmetric problem (top images) and the nonsymmetric problem (bottom images)

the upper left corner — corresponding to the situation for the frequency contents of the singular vectors in Figs. 1 and 2. We also observe that the “freckles” appear as a bandlimited ring of frequencies in the 2D-DCT domain, and that this ring gets larger and more intense when increasing the number of iterations. Due to the SVD filtering, we know that the components remaining to be restored are all higher-frequent and contaminated by noise. Thus all the extractable information about the true solution lies in the lower frequencies which are all reconstructed when the “freckles” start to dominate.

The higher convergence speed observed by Calvetti et al. [3] for GMRES with less noise is verified by our MR-II and RRGMRES solutions that exclude the noisy component from the Krylov subspace. An interesting observation is that removing the noise from the GMRES subspace results in “freckle” artifacts similar to LSQR if the iterations are run to far.

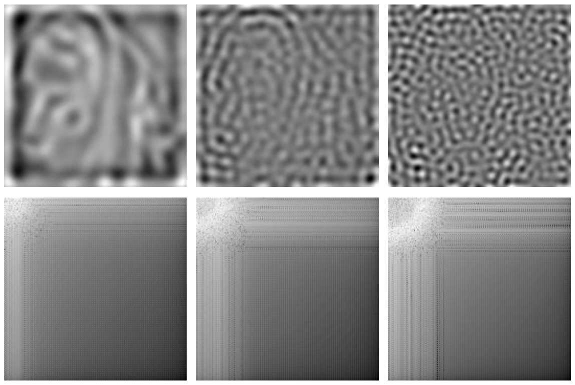


Fig. 7. Top: The noise components $X_e^{(k)}$ of the LSQR solution for iteration 4, 10 and 25. Bottom: The 2D-DCT of these noise components.

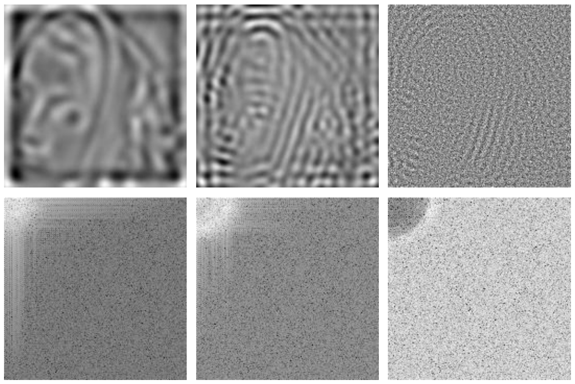


Fig. 8. Top: The noise components $X_e^{(k)}$ of the GMRES solution for iteration 4, 10 and 25. Bottom: The 2D-DCT of these noise components.

The reason that some GMRES/MINRES solutions appear to be visually better than the LSQR solutions in a low-noise setting is simply because the "freckles" do not appear to the same extend. The "freckles" are very disturbing for for the visual quality of the solutions, while the propagated noise component is not — provided its size is sufficiently small. The extra noise can even create an illusion of higher resolution even though no true information can be reliably reconstructed beyond the "freckles." Figure 8 shows how the noise is propagated by GMRES in the low-noise setting actually also produce ringing effects, but of much smaller magnitude than the wanted signal contents. The main contributions from the subspace is low-frequent, but the higher frequencies are slowly boosted to cover the freckles.

6 Conclusion

We have provided some insight into the use of regularizing iterations in connection with image restoration. We showed that – similar to the one-dimensional case – the SVD provides a frequency (or spectral) decomposing, and that the projection onto the Krylov subspace of LSQR/CGLS has a regularizing effect. We also showed that the GMRES and MINRES Krylov subspaces contain noise components that can severely deteriorate the solutions, and hence the MR-II and RRGMRES variants are to be preferred. Finally we demonstrated that all the methods can produce visually disturbing artifacts which appear as "freckles."

References

1. Calvetti, D., Lewis, B., Reichel, L.: GMRES-type methods for inconsistent systems. Linear Algebra and its Applications 316, 157–169 (2000)
2. Calvetti, D., Lewis, B., Reichel, L.: On the choice of subspace for iterative methods for linear discrete ill-posed problems. International Journal of Applied Mathematics and Computer Science 11, 1069–1092 (2001)
3. Calvetti, D., Lewis, B., Reichel, L.: GMRES, L-curves, and discrete ill-posed problems. BIT 42, 44–65 (2002)
4. Hanke, M.: The minimal error conjugate gradient method is a regularization method. Proc. Amer. Math. Soc. 123, 3487–3497 (1995)
5. Hanke, M., Brakhage, H.: Conjugate gradient type methods for ill-posed problems. SIAM Review 38, 682 (1996)
6. Hansen, P.C.: Rank-Deficient and Discrete Ill-Posed Problems. SIAM, Philadephia (1998)
7. Hansen, P.C.: Deconvolution and regularization with Toeplitz matrices. Numer. Algo. 29, 323–378 (2002)
8. Hansen, P.C., Kilmer, M.E., Kjeldsen, R.H.: Exploiting residual information in the parameter choice for discrete ill-posed problems. BIT 46, 41–59 (2006)
9. Hansen, P.C., Nagy, J.G., O'Leary, D.P.: Deblurring Images: Matrices, Spectra, and Filtering. SIAM, Philadelphia (2006)
10. Ipsen, I.C.F., Meyer, C.D.: The idea behind Krylov methods. American Mathematical Monthly 105, 889–899 (1998)
11. Jain, A.K.: Fundamentals of Digital Image Processing. Prentice Hall International Editions (1989)
12. Nagy, J.G., Palmer, K.M., Perrone, L.: Iterative Methods for Image Deblurring: A Matlab Object Oriented Approach. Numerical Algorithms 36, 73–93 (2004), See also `http://www.mathcs.emory.edu/~nagy/RestoreTools`
13. Vogel, C.R.: Solving ill-conditioned linear systems using the conjugate gradient method. Technical report, Department of Mathematical Sciences, Montana State University, Bozeman, MT 59717 (1987)

A Case Study in High-Performance Mixed-Language Programming

Hans Petter Langtangen[1,2]

[1] Simula Research Laboratory, Oslo, Norway
hpl@simula.no
http://folk.uio.no/hpl
[2] Dept. of Informatics, University of Oslo, Norway

Abstract. Several widely used and promising programming tools and styles for computational science software are reviewed and compared. In particular, we discuss function/subroutine libraries, object-based programming, object-oriented programming, generic (template) programming, and iterators in the context of a specific example involving sparse matrix-vector products. A key issue in the discussion is to hide the storage structure of the sparse matrix in application code. The role of different languages, such as Fortran, C, C++, and Python, is an integral part of the discussion. Finally, we present performance measures of the various designs and implementations. These results show that high-level Python programming, with loops migrated to compiled languages, maintains the performance of traditional implementations, while offering the programmer a more convenient and efficient tool for experimenting with designs and user-friendly interfaces.

1 Introduction

During the last fifty years we have experienced a tremendous speed-up of hardware and algorithms for scientific computing. An unfortunate observation is that the development of programming techniques suitable for high-performance computing has lagged significantly behind the development of numerical methods and hardware. This is quite surprising since difficulties with handling the complexity of scientific software applications is widely recognized as an obstacle for solving complicated computational science problems. As a consequence, technical software development is a major money consumer in science projects. The purpose of this paper is to review and compare common scientific programming techniques and to outline some new promising approaches.

The first attempts to improve scientific software development took place in the 1970s, when it became common to collect high-quality and extensively verified implementations of general-purpose algorithms in Fortran subroutine libraries. The subroutines then constitute the algorithmic building blocks in more advanced applications, where data represented by array, integer, real, and string variables are shuffled in and out of subroutines. Such libraries are still perhaps the most important building blocks in scientific software (LAPACK being the primary example).

In the 1980s and 1990s some transition from Fortran to C took place in the scientific computing community. C offers the possibility to group primitive variables together to form new compound data types. Fortran 90 also provides this functionality. The result is that the function arguments are fewer and reflect a higher abstraction level.

With C++ or Fortran 90, and the class or module concept, compound data types could also be equipped with functions operating on the data. Application code may then define some class objects and call their functions instead of shuffling long lists of variables in and out of function/subroutine libraries as in C and Fortran 77. C++ also offers object-oriented programming and templates, two constructs that can be used to parameterize types away such that one single function may work with different types of input data. This reduces the amount of code in libraries significantly and increases the flexibility when using the library in a specific application code. The upcoming Fortran 2003/2008 will enable the same constructs.

An important feature of the present paper is that we, along with the above mentioned programming strategies in Fortran, C, and C++, also illustrate the use of the Python language. Python supports all major programming styles and serves as a more convenient and more productive implementation environment than the traditional compiled languages used in high-performance computing. A particularly important feature of Python is the clean syntax, which by many is referred to as "executable pseudocode". This feature helps to bridge the gap between a mathematical expression of a solution algorithm and the corresponding computer implementation. Add-on modules equip Python with functionality close to that of basic Matlab. Contrary to Matlab, Python offers an advanced, full-fledged programming language with strong support for clean programming with user-defined objects. Another advantage of Python is its rich set of libraries for non-numerical/administrative tasks, and such tasks tend to fill up large portions of scientific codes.

The paper is divided into two parts. First, we exemplify major programming techniques, tools, and languages used in scientific computing and demonstrate how Python is both a complement and an alternative to Fortran, C, and C++. The second part presents some computational performance assessments of the various programming techniques. We remark that the paper primarily addresses readers with some technical knowledge of C, C++, and Python.

2 A Review of Programming Tools and Techniques

Programming tools and techniques are often best illustrated by applying them to a specific problem. We have chosen to focus on the matrix-vector product because the numerics of this problem is simple, the codes become short, and yet different implementation styles exhibit diverse properties.

Mathematically, we want to compute $\boldsymbol{r} = \boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{A}$ is a square $n \times n$ matrix, and $\boldsymbol{x}$ and $\boldsymbol{r}$ are n-vectors. This operation is a key ingredient in many numerical algorithms. In particular, if $\boldsymbol{A}$ arises from discretization of a partial differential equation with finite difference, element, or volume methods, $\boldsymbol{A}$ is sparse in the sense that most matrix entries are zero. Various matrix storage schemes enable the programmer to take computational advantage of the fact that $\boldsymbol{A}$ is sparse. One such storage scheme is the *compressed row storage* format. Given a matrix

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & 0 & 0 & a_{0,3} & 0 \\ 0 & a_{1,1} & a_{1,2} & 0 & a_{1,4} \\ 0 & a_{2,1} & a_{2,2} & 0 & 0 \\ a_{3,0} & 0 & 0 & a_{3,3} & a_{3,4} \\ 0 & a_{4,1} & 0 & a_{4,3} & a_{4,4} \end{pmatrix},$$

the compressed row storage scheme consists of a one-dimensional array, `a`, of all the nonzeroes, stored row by row, plus two indexing integer arrays, where the first, `rowp` ("row pointer"), tells where each row begins, and the second, `coli`, holds the column indices:

$$\begin{aligned} \texttt{a} &= (a_{0,0}, a_{0,3}, a_{1,1}, a_{1,2}, a_{1,4}, a_{2,1}, a_{2,2}, a_{3,0}, a_{3,3}, \ldots, a_{4,4}) \\ \texttt{rowp} &= (0, 2, 5, 7, 10, 13) \\ \texttt{coli} &= (0, 3, 1, 2, 4, 1, 2, 0, 3, 4, 1, 3, 4) \end{aligned}$$

Notice here that we index $\boldsymbol{A}$ with base zero, i.e., the entries of $\boldsymbol{A}$ are $a_{i,j}$ for $i, j = 0, \ldots, n-1$. Also notice that `rowp` has length $n+1$, with last entry equal to the number of nonzeroes.

The straightforward algorithm for computing the matrix-vector product goes as follows:

```
r = 0
for i = 0, ..., n-1:
    s = 0
    for j = rowp[i], ..., rowp[i+1]-1:
        s = s + a[j]*x[coli[j]]
    r[i] = s
```

The next subsections exemplify how different programming techniques result in different levels of complexity in codes that need to call matrix-vector product functionality for different matrix storage schemes. Two schemes are exemplified: a two-dimensional array for a (full) $n \times n$ matrix and compressed row storage of a sparse matrix.

2.1 Classical Fortran 77 and C Implementations

In Fortran 77 the matrix-vector product is naturally implemented as a subroutine where $\boldsymbol{A}$, $\boldsymbol{x}$, $\boldsymbol{r}$, and the array size n are subroutine arguments. Typically, the product is computed by

```
call matvec_dense(A, n, x, r)
call matvec_sparse(a, rowp, coli, n, nz, x, r)
```

for the two cases that $\boldsymbol{A}$ is stored in a two-dimensional $n \times n$ array `A` or $\boldsymbol{A}$ is a sparse matrix stored in the `a`, `rowp`, and `coli` arrays (`nz` is the number of nonzeroes, i.e., the length of a and coli). As seen, the calls are quite different although the mathematics is conceptually identical. The reason is that Fortran 77 exposes all details of the underlying data structures for matrix storage. An implication for a linear solver, say some Conjugate Gradient-like method, where the matrix-vector product is a key ingredient in the algorithm, is that the code depends explicitly on the way we store the matrix. Adding a new storage scheme implies that all matrix-vector product calls in the whole code need modification.

A similar C implementation differs mainly by its use of pointers instead of proper arrays:

```
void matvec_dense(double** A, int n, double* x, double* r);
matvec_dense(A, n, x, r);  /* call */

void matvec_sparse(double* a, int* rowp, int* coli, int n,
                   int nz, double* x, double* r);
matvec_sparse(a, rowp, coli, n, nz, x, r);  /* call */
```

2.2 Object-Based Implementation in Terms of C-Like Structs

A significant language feature of C is the struct type, which allows several basic data types to be grouped together. It makes sense that the arrays a, rowp, and coli used to represent a sparse matrix, are grouped together and equipped with information on the sizes of the array:

```
struct MatSparseC {
    double* a; int* rowp; int* coli; int n; int nz;
};

MatSparseC A = MatSparseC();
```

We can now access the internal sparse matrix arrays through the dot notation: A.a, A.rowp, etc.

A dense matrix representation of $\boldsymbol{A}$ in terms of a C array can be implemented similarly:

```
struct MatDenseC {
    double** A; int n;
};
```

The calls to matrix-vector product functions

```
void matvec_dense (MatDenseC*  A, double* x, double* r);
void matvec_sparse(MatSparseC* A, double* x, double* r);
```

can now be made more uniform:

```
matvec_dense (&A, x, r);
matvec_sparse(&A, x, r);
```

That is, only the fundamental mathematical objects $\boldsymbol{A}$, $\boldsymbol{x}$, and $\boldsymbol{r}$ enter the function that computes $\boldsymbol{r} = \boldsymbol{A}\boldsymbol{x}$.

Applying C++ as a "better C", we may use the same name for the matrix-vector product function (known as *function overloading*) such that the call hides the way $\boldsymbol{A}$ is stored:

```
void matvec(const MatDenseC*  A, const double* x, double* r);
void matvec(const MatSparseC* A, const double* x, double* r);

matvec(&A, x, r);  // same syntax when A is MatDenseC or MatSparseC
```

The `const` keyword contributes to increased readability (input arguments) and safety (no possibility to change `A` and `x` inside the functions).

Actually, in C++ we would use references instead of pointers and probably also wrap primitive arrays in a class. A natural choice is to employ the ready-made vector class `std::valarray<double>` from the C++ Standard Template Library (STL) [6]. Alternatively, a home-made vector class can easily be implemented. For the rest of the paper we introduce the short form `Vecd` for `std::valarray<double>`. A new vector of length `n` is then created by `Vecd(n)`, while `Vecd(a,n)` borrows data from an existing `double* a` array of length `n`.

With help of the `Vecd` type the function signatures and associated call become

```
void matvec(const MatDenseC&  A, const Vecd& x, Vecd& r);
void matvec(const MatSparseC& A, const Vecd& x, Vecd& r);

matvec(A, x, r);  // same syntax when A is MatDenseC or MatSparseC
```

When data are grouped together using a struct or class, one often speaks about *object-based programming*, since the fundamental variables are objects (here `A`) encapsulating more primitive data types.

In Python we may mirror all the implementations above. The relevant array data structure is supported by Numerical Python [3], and this structure contains both the array data and the length of each array dimension. Fortran-style functions can be implemented in Python and called as

```
matvec_dense2(A, x, r)
matvec_sparse2(a, rowp, coli, x, r)
```

A more "Pythonic" implementation would return the result `r` rather than having it as an argument (and performing in-place modifications of `r`):

```
r = matvec_dense(A, x)
r = matvec_sparse(a, rowp, coli, x)
```

This latter design requires the `matvec_*` routines to allocate a new `r` vector in each call. This work introduces some overhead that we will quantify in Section 3.

Python's class construction can be used to group data structures into objects as one does in C with a `struct`. Here is a possible example:

```
class MatSparseC:
    pass  # start with empty class

A = MatSparseC()
A.a = some_a; A.rowp = some_rowp; A.coli = some_coli
r = matvec_sparse_C(A, x)

class MatDenseC:
    pass

A = MatDenseC(); A.a = some_a
r = matvec_dense_C(A, x)
```

Function overloading is not supported in Python so each function must have a unique name. Also observe that in Python, we may declare an empty class and later add desired attributes. This enables run time customization of interfaces.

2.3 Object-Based Implementation in Terms of Classes

The *class* concept, originally introduced in SIMULA and made popular through languages such as Smalltalk, C++, Java, C#, and Python, groups data structures together as in C structs, but the class may also contain functions (referred to as *methods*) operating on these data structures. An obvious class implementation of sparse and dense matrices would have the matrix-vector product as a method. The idea is that the internal data structures are invisible outside the class and that the methods provide algorithms operating on these invisible data. In the present example, the user of the class is then protected from knowing how a matrix is stored in memory and how the algorithm takes advantage of the storage scheme.

A typical C++ implementation of a sparse matrix class may be outlined as follows.

```
class MatSparse
{
private:
  Vecd a;
  Vec<int>    rowp;
  Vec<int>    coli;
  int n, nz;
public:
  MatSparse(double* a_, int* rowp_, int* coli_, int nz, int n) :
    a(a_, nz), rowp(rowp_, n+1), coli(coli_, nz) {}

  Vecd matvec (const Vecd& x) const {
    Vecd r = Vecd(x.size());
    matvec2(x, r); return r;
  }
  void matvec2 (const Vecd& x, Vecd& r) const {
    matvec_sparse(a.ptr(), rowp.ptr(), coli.ptr(),
                  rowp.size()-1, a.size(), x.ptr(), r.ptr());
  }

  Vecd operator* (const Vecd& x) const {
    return matvec(x);
  }
};
```

There are three matrix-vector multiplication methods: one that takes a pre-allocated `r` array as argument, one that returns the result `r` in a new array, and one (similar) that overloads the multiplication operator such that $\boldsymbol{r} = \boldsymbol{A}\boldsymbol{x}$ can be written in C++ code as `r=A*x`. Observe that if we have the plain C function `matvec_sparse` from page 38, class `MatSparse` above may be thought of as a wrapper of an underlying C function library.

The idea is now that all matrix storage formats are encapsulated in classes such as `MatSparse`, i.e., we may have classes `MatDense`, `MatTridiagonal`, `MatBanded`, etc. All of these have the same interface from an application programmer's point of view:

```
// A is MatSparse, MatDense, etc., x is Vecd
Vecd r(n);

r = A*x;
r = A.matvec(x);
A.matvec2(x, r);
```

This means that we may write a solver routine for a Conjugate Gradient-like method by using the same matrix-vector product syntax regardless of the matrix format:

```
void some_solver(const MatSparse& A, const Vecd& b, Vecd& x)
{
  ...
  r = A*x;
  ...
}
```

Nevertheless, the arguments to the routine must be accompanied by object types, and here the explicit name `MatSparse` appears and thereby ties the function to a specific matrix type. Object-oriented programming, as explained in the next section, can remove the explicit appearance of the matrix object type.

What we did above in C++ can be exactly mirrored in Python:

```
class MatSparse:
    """Class for sparse matrix with built-in arithmetics."""
    def __init__(self, a, rowp, coli):
        self.a, self.rowp, self.coli = a, rowp, coli  # store data

    def matvec(self, x):
        return matvec_sparse(self.a, self.rowp, self.coli, x)

    def matvec2(self, x, r):
        matvec_sparse2(self.a, self.rowp, self.coli, x, r)

    def __mul__(self, x):
        return self.matvec(x)
```

The `__mul__` method defines the multiplication operator in the same way as `operator*` does in C++. Also this Python class merely acts as a wrapper of an underlying function library.

Contrary to C++, there is no explicit typing in Python so a single solver routine like

```
def some_solver(A, x, b):
    ...
    r = A*x
    ...
```

will work for all matrix formats that provide the same interface. In other words, there is no need for object orientation or template constructs as we do in C++

to reach the same goal of having one solver that works with different matrix classes.

We should add that the matrix classes shown above will normally be equipped with a more comprehensive interface. For example, it is common to provide a subscription operator such that one can index sparse matrices by a syntax like `A(i,j)` or `A[i,j]`. More arithmetic operators and I/O support are also natural extensions.

2.4 Object-Oriented Implementation

The key idea behind object-oriented programming is that classes are organized in hierarchies and that an application code can treat all classes in a hierarchy in a uniform way. For example, we may create a class `Matrix` that defines (virtual) methods for carrying out the matrix-vector product (like `matvec`, `matvec2`, and the multiplication operator), but there are no data in class `Matrix` so it cannot be used to store a matrix. Subclasses implement various matrix formats and utilize the specific formats to write optimized algorithms in the matrix-vector product methods. Say `MatDense`, `MatSparse`, and `MatTridiagonal` are such subclasses. We may now write a solver routine where the matrix object is parameterized by the base class,

```
void some_solver(const Matrix& A, const Vecd& b, Vecd& x)
{
  ...
  r = A*x;
  ...
}
```

Through the `Matrix` reference argument we may pass any instance of any subclass in the `Matrix` hierarchy, and C++ will at run-time keep track of which class type the instance corresponds to. If we provide a `MatSparse` instance as the argument `A`, C++ will automatically call the `MatSparse::operator*` method in the statement `r=A*x`. This hidden book-keeping provides a kind of "magic" which simplifies the implementation of the flexibility we want, namely to write a single function that can work with different types of matrix objects.

Python offers the same mechanism as C++ for object-oriented programming, but in the present case we do not need this construction to parameterize the matrix type away. The dynamic typing (i.e., the absence of declaring variables with types) solves the same problem as object orientation in C++ aims at. However, in some cases where some matrix classes can share data and/or code, it may be beneficial to use inheritance and even object-oriented programming. This is exemplified in Section 2.6.

2.5 Generic Programming with Templates in C++

C++ offers the template mechanism to parameterize types. With templates we can obtain much of the same flexibility as dynamic typing provides in Python.

For example, we can write the solver function with a template for the matrix type:

```
template <typename Matrix>
void some_solver(const Matrix& A, const Vecd& b, Vecd& x)
{
  ...
  r = A*x;
  ...
}
```

Now we may feed in totally independent classes for matrices as long as these classes provide the interface required by the `some_solver` function. Sending (say) a `MatSparse` instance as `A` to this function causes the compiler to generate the necessary code where `Matrix` is substituted by `MatSparse`. Writing algorithms where the data are parameterized by templates is frequently referred to as generic programming. This style encourages a split between data and algorithms (see Section 2.7), while object-oriented programming ties data and algorithms more tightly and enforces classes to be grouped together in hierarchies. A consequence is that libraries based on object-oriented programming are often harder to reuse in new occasions than libraries based on generic programming. The class methods in object-oriented programming are virtual, which implies a (small) overhead for each method call since the methods to call are determined at run-time. With templates, the methods to call are known at compile time, and the compiler can generate optimally efficient code.

2.6 Combining Python and Compiled Languages

Python offers very compact and clean syntax, which makes it ideal for experimenting with interfaces. The major deficiency is that loops run much more slowly than in compiled languages, because agressive compiler optimizations are impossible when the variables involved in a loop may change their type during execution of the loop. Fortunately, Python was originally designed to be extended in C, and this mechanism can be used to easily migrate time-consuming loops to C, C++, or Fortran.

Integration of Python and Fortran is particularly easy with the aid of F2PY [4]. Say we have the `matvec_sparse` routine from page 38. After putting this (and other desired routines) in a file `somecode.f`, it is merely a matter of running

```
Unix/DOS> f2py -m matvec -c somecode.f
```

to automatically parse the Fortran code and create a module `matvec` that can be loaded into Python as if it were written entirely in Python. Either in the Fortran source code, or in an intermediate module specification file, the programmer must specify whether the `r` parameter is a pure output parameter or if it is both input and output (all parameters are by default input). Because all arguments are classified as input, output, or both, F2PY may provide both the interfaces `matvec_sparse` and `matvec_sparse2` on page 40 by calling the same underlying

Fortran routine. It is the F2PY-generated wrapper code that, in the former case, allocates a new result array in each call.

Class MatSparse can now be modified to call Fortran routines for doing the heavy work. The modification is best implemented as a subclass that can inherit all the functionality of MatSparse that we do not need to migrate to Fortran:

```
class MatSparse_compiled(MatSparse):
    def __init__(self, a, rowp, coli):
        MatSparse.__init__(self, a, rowp, coli)
        import matvec;  self.module = matvec

    def matvec(self, x):
        # call Fortran routine, let wrapper allocate result:
        return self.module.matvec_sparse(self.a, self.rowp,
                                         self.coli, x)

    def matvec2(self, x, r):
        # call Fortran routine, use pre-allocated result r:
        self.module.matvec_sparse2(self.a, self.rowp,
                                   self.coli, x, r)
```

Note that it is not necessary to redefine __mul__ since the inherited version (from MatSparse) calls matvec, which behaves as a virtual method, i.e., the correct version of matvec in the subclass will be automatically called (we thus rely on object-oriented programming when we do not repeat __mul__ in the derived class). To summarize, the classes MatSparse and MatSparse_compiled have the same interface and can be used interchangeably in application code.

F2PY can also be used to wrap C code. However, it is hard to parse C code and detect which function arguments are arrays because (i) a pointer can be both an array and the address of a scalar, and (ii) there is no mechanism to associate array dimensions with a pointer. By expressing the function signatures in C in a Fortran 90-like module (an *interface* or .pyf file in F2PY terminology), F2PY can automatically generate extension modules from C sources. It is a simple matter to extend class MatSparse_compiled so that self.module either refers to a Fortran or to a C extension module. We have done this in the experiments to see if there are performance differences between Fortran and C.

There exist several techniques for migrating the matrix-vector product to C++. One technique is to apply a tool like SWIG [7] to automatically wrap the C++ class MatSparse such that it is mirrored in Python. Another alternative is to make a straight C function taking pointers and integers as arguments and then pass the arrays to a C++ function that creates a MatSparse instance for computation. The latter strategy has been implemented and will be referred to as *Python calling C, calling C++* MatSparse::matvec2. The C function was made callable from Python by F2PY. A third technique is to use tools like Weave, Pyrex, PyInline, or Instant to write inline C or C++ code in Python.

When Python is combined with Fortran, C, or C++, it is natural and convenient to allocate memory on the Python side. Numerical Python arrays have contiguous memory layout and can be efficiently shuffled to compiled code as pointers.

2.7 Iterators

Classical Fortran and C code applies do or for loops over integer indices to traverse array data structures. This coding style works in most other languages as well, but C, C++, and Fortran compilers are very good at optimizing such constructions.

In recent years, *iterators* have become a popular alternative to loops with integer indices. The standard template library (STL) [6] in C++ promoted the use of iterators in the mid 1990s, and the construction has gained wide use also in scientific computing (three extensive examples being Boost [1], MTL [5], and Dune [2]). One may view iterators as generalized loops over data structures. A matrix-vector product can be implemented as something like

```
template <class Matrix>
void matvec(const Matrix& A, const Vecd& x, Vecd& r)
{
  r = 0;
  Matrix::iterator element;
  for (element = A.begin(); element != A.end(); ++element) {
      r(element.i()) += (*element)*x(element.j());
      //  index i          value        index j
}
```

The idea is to have an iterator instance, here Matrix::iterator element, which acts as generalized pointer to the elements of the data structure. The for loop starts with setting element to point to the beginning of the data structure A. The loop continues as long as element has not reached one step beyond the data in A. In each pass we update the element "pointer" by the unary Matrix::iterator::operator++ method. For a sparse matrix, this operator++ must be able to pass through all the nonzeroes in each row of A. The element instance also provides a pointer dereferencing, through the unary operator*, for accessing the corresponding data element A. The mathematical matrix indices are available through the methods element.i() and element.j().

We emphasize that there are many design considerations behind a particular iterator implementation. The one shown here is just an example. High-quality numerical libraries based on iterators provide different designs (MTL [5] applies separate row and column iterators, for instance).

The idea of using iterators to implement a matrix-vector product is that the same function can be used for different matrix formats, as long as the provided matrix instance A offers an iterator class with the described interface and functionality. Using inline functions for the operator++, operator*, i, and j methods, the iterator class may with a clever implementation be as efficient as classical loops with integers. In the present example with one iterator running through the whole sparse matrix, optimal efficiency is non-trivial.

In some way iterators are both complementary and an alternative to templates, object-oriented programming, or dynamic typing. Iterators are here used to implement the matrix-vector product such that the same code, and the same call of course, can be reused for various matrix objects. The previous constructions in Sections 2.1–2.6 aimed at encapsulating *different* matrix-vector implementations in terms of loops tailored to the underlying representation array of the matrix, such that the product looks the same in application code and the matrix type is parameterized away. Iterators and templates in combination (generic programming) provide a quite different solution to the same problem.

Iterators may be somewhat comprehensive to implement in C++. To play around with iterators to see if this is the desired interface or not, we strongly propose to use Python because the language offers very convenient constructs to quickly prototype iterators. For example, the sparse matrix iterator can be compactly implemented through a so-called Python *generator* function as follows:

```
class MatSparse:
    ...
    def __iter__(self):
        """Iterator: for value, i, j in A:"""
        for i in xrange(len(self.rowp)-1):
            for c in xrange(self.rowp[i],self.rowp[i+1]):
                yield self.a[c], i, self.coli[c]
```

We can now write the matrix-vector product function in a generic way as

```
def matvec(A, x):
    r = zeros(len(x), x.dtype)
    for value, i, j in A:
        r[i] += value*x[j]
    return r
```

This implementation works well with all matrix instances that implement an iterator that returns a 3-tuple with the matrix value and the two indices. Observe how this compact `for` loop replaces the traditional nested loops with integer indices, and also notice how simple the mapping between the traditional loops and the new iterator is (in the `MatSparse.__iter__` method).

A faster but also more memory consuming implementation can be made by precomputing a (new sparse matrix) storage structure that holds the `(value,i,j)` tuple for each matrix entry:

```
    def __iter__(self):
        try:
            return self._ap  # use a precomputed list as iterator
        except AttributeError:  # first time, precompute:
            self._ap = [(value, i, j) for value, i, j in self]
            return self._ap
```

3 Performance Experiments

We have run experiments comparing the various implementations sketched in Section 2 on an IBM laptop X30 machine with 1.2 GHz Pentium III processor and 500 Mb memory, using Linux Ubuntu 2.6.12, GNU compilers (`gcc`, `g++`, `g77`) version 4.0 with `-O3` optimization, Python version 2.4, `numpy` version 1.0, and `Numeric` version 24.2.

The results in Table 1 show that there is no significant difference between implementations of the matrix-vector product `r=A*x` in C, C++, and Fortran (and especially no overhead in wrapping arrays in the C++ class `MatSparse`). Using a preallocated result array `r` is as expected most efficient in these tests where a large number (160) of matrix-vector products are performed. Letting the result array be allocated in each call gives 10-15% overhead, at the gain of getting more "mathematical" code, either `r=matvec_sparse(A,x)` or `r=A*x`.

The pure Python implementation may be useful for experimenting with alternative software designs, but the CPU time is increased by two orders of magnitude, so for most applications of Python in scientific computing one must migrate loops to compiled code. We also note that the new Numerical Python implementation `numpy` is still slower than the old `Numeric` for pure Python loops, but the efficiency of `numpy` is expected to increase in the future.

Iterators are very conveniently implemented in Python, but these run slower than the plain loops (which we used internally in the iterator implementation). Even a straight `for` loop through a precomputed data structure tailored for the iterator is as slow as plain loops. The overhead of loops in Python also drowns the expected positive effect of preallocating arrays.

When our Python class for sparse matrices has its loops migrated to compiled code, there is no loss of efficiency compared to a pure Fortran or C function working with plain arrays only. We have from additional experiments estimated that a Python call to a Fortran function costs about 70 Fortran floating-point multiplications, which indicates the amount of work that is needed on the Fortran side to make the overhead negligible.

Table 1. Scaled CPU times for various implementations of a matrix-vector product with vector length 50,000 and 231,242 nonzeroes

Implementation	`numpy`	`Numeric`
Python calling `matvec_sparse2`, calling C, no array returned	1.0	1.0
Python calling `matvec_sparse2`, calling F77, no array returned	1.0	1.0
Python calling C, calling C++ `MatSparse::matvec2`	1.0	1.0
Python `MatSparse_compiled.matvec2`	1.0	1.0
Python calling `matvec_sparse`, calling F77 , new array returned	1.10	1.12
Python calling `matvec_sparse`, calling C, new array returned	1.13	1.15
Python `MatSparse_compiled.matvec`	1.13	1.12
Python `MatSparse_compiled: r=A*x`	1.16	1.13
Python iterator with new precomputed matrix data	136	89
Python `MatSparse.matvec`	153	92
Python `MatSparse.matvec2`	153	92
Python `MatSparse: r=A*x`	153	92
Python function `matvec_sparse`	153	92
Python iterator based on generator method	228	156

4 Concluding Remarks

We have in this paper reviewed the ideas of function libraries, object-based programming, object-oriented programming, generic programming and iterators, using a simple example involving the implementation of a matrix-vector product. The role of the traditional high-performance computing languages Fortran and C, along with the more recently accepted language C++ and the potentially interesting Python language, has also been discussed in the context of the example. A driving force from "low level" languages like Fortran and C to higher-level languages such as C++ and Python is to offer interfaces to matrix operations where the specific details of the sparse matrix storage are hidden from the application programmer. That is, one goal is to write one matrix-vector call and make it automatically work for all matrix types. Object-oriented programming, generic programming with templates and iterators, as well as the dynamic typing in Python, provide different means to reach this goal.

A performance study shows that low level (and fast) Fortran or C code can be wrapped in classes, either in C++ or Python, with negligible overhead when the arrays are large as in the present tests. There is neither any difference in speed between loops in Fortran, C, and C++ on the author's machine with GNU v4.0 compilers.

The author's most important experience from this case study is that scientific programming is much more convenient in Python than in the more traditional languages. Therefore, Python shows a great potential in scientific computing, as it promotes quick and convenient prototyping of user-friendly interfaces, while (with a little extra effort of migrating loops to compiled code) maintaining the performance of traditional implementations.

References

1. Boost C++ libraries, `http://www.boost.org/libs/libraries.htm`
2. Dune library, `http://www.dune-project.org`
3. Numerical Python software package, `http://sourceforge.net/projects/numpy`
4. Peterson, P.: F2PY software package, `http://cens.ioc.ee/projects/f2py2e`
5. Siek, J., Lumsdaine, A.: A modern framework for portable high-performance numerical linear algebra. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Advances in Software Tools for Scientific Computing, Springer, Heidelberg (1999)
6. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley, Reading (1997)
7. SWIG software package, `http://www.swig.org`

Stretching Time and Length Scales in Biomolecular Modelling: Minisymposium Abstract

Aatto Laaksonen

Arrhenius Laboratory, Stockholm University, Sweden

Molecular modelling/simulation techniques have in three decades evolved to a powerful tool and scientific discipline of its own now used in many areas of physics, chemistry, and biology with applications from materials science to biotechnology. These techniques have naturally become applicable for more and complex systems largely thanks to the rapid development in computer technology. However in recent years a variety of new advanced and innovative techniques have been presented to push the time and length scales further towards nano/meso scale applications and soft matter. More efficient computational schemes have been proposed to treat long-ranged interactions, parallel algorithms are proposed to run on high-end fast computers, pc-clusters and heterogeneous GRID environment. Ab initio and hybrid QM/MM methods are becoming routine and developed to treat large systems. Multi-scale modelling schemes across several physical descriptions of matter from quantum mechanical systems with nuclei and electrons all the way to nano/meso/micro/macro levels are maturing rapidly. This minisymposium will highlight several of these latest techniques.

Averaged Configurations from Molecular Dynamics Simulations

K. Gillis[1], J. Vatamanu[1], M.S. Gulam Razul[2], and Peter G. Kusalik[1]

[1] Department of Chemistry, University of Calgary,
Calgary, Alberta, T2N 1N4, Canada
peter.kusalik@ucalgary.ca

[2] Department of Physics, Saint Francis Xavier University,
Antigonish, Nova Scotia, Canada

Abstract. One of the challenges in the large scale simulations required for biomolecular system is the recording, monitoring and visualization of configurational information from molecular dynamics trajectories. A detailed record of instantaneous configuration along the full trajectory can quickly become unmanageable. In this paper we will describe an alternative approach where configurations averaged over trajectory segments are used to follow the detailed molecular behaviour of a system over multiple-nanosecond simulations. We will then discuss the successful application of this approach to molecular dynamics simulations of crystal growth.

1 Introduction

Analysis of system configurations can be a key component in the computer simulation of molecular systems. These configurations, which supply full molecular details of the (relative) positions of the particles that make up the system, are frequently used to provide a means of exploring the microscopic behaviour captured by a molecular simulation. It is typical for these configurations to consist simply of the positions of all particles at a particular instant in time along the dynamical trajectory of the system. A set of such instantaneous configurations from a simulation can be saved and later analyzed further or visualized.

Complications arise in simulations, such as those required for biomolecular systems, where large length and time scales are necessary to capture the behaviour of interest [1,2,3,4]. In these cases, the storage required to maintain this detailed record of the systems evolution (for example, as instantaneous configurations recorded every 50fs) can be prohibitive (possibly requiring 10^6 configurations, or more). Moreover, the visualization of a system at this resolution over a multiple nanosecond trajectory becomes quite impractical, and may well contain detailed information (for example, due to the thermal motion of the atoms or molecules of interest) that tends to obscure the most relevant behaviour. One remedy to these problems is to select and record instantaneous configurations on a far coarser grid in time (perhaps every 50 ps, for example). However, while such an approach may retain some basic aspects of the dynamics exhibited by the system, considerable information is lost nonetheless. For instance, it may not be clear if the instantaneous configuration recorded at the end of a relatively long trajectory segment is representative of the behaviour exhibited by the system during that segment. Below we will demonstrate that an alternative approach, that employs averaged configurations to provide a true time coarse-graining of a simulation trajectory, captures considerably more detailed molecular information and allows larger scale behaviour in such systems to be tracked more readily.

2 Generation of Averaged Configurations

Spatial coarse-graining has become a widely used approach [3,4,5,6] in molecular simulations of large systems (such as membrane system) where by the detailed behaviour of groups of atoms are represented by (averaged into) single effective interaction sites. The present approach is similar in character in that to will rely on a coarse-graining, but now in time. Specifically, an averaged configuration can be produced for any particular trajectory segment by averaging molecular (or atomic) coordinates,

$$\bar{x} = \frac{1}{N_s} \sum_{t=1}^{N_s} x_t \ , \tag{1}$$

over the N_s time steps of the segment spanning a time τ. In Eq. (1), x can represent a positional coordinate; the case of orientational coordinates will be discussed below. The choice of segment length can be an important consideration; it typically is made as long as possible while still providing a reasonable vantage point (i.e., frequency of sampling) from which to observe the complex processes characterizing the molecular behaviour of interest. With an appropriate selection of trajectory segment length over which to average, extraneous particle motion (e.g. thermal motion) can be effectively removed so that any net (more gross) motion can be more easily observed.

For each trajectory segment one has in principle a distribution of values for each degree of freedom of the system. To provide additional information into the nature of these distributions (beyond their means, or first-moments), we also find it advantageous to monitor their second-moments, or widths. The root mean-squared (RMS) deviations,

$$\sigma = \left(\overline{x^2} - \bar{x}^2\right)^{1/2} \ , \tag{2}$$

are also measures of the diffusive motion exhibited by a molecule during the trajectory segment and hence are clearly related to Debye-Waller factors. The treatment of molecular orientations requires specific attention. At least for small molecules (like water), we find it advantageous to separate and average their degrees of freedom as positions and orientations. This approach allows for the conservation of molecular geometry during the averaging process. However, Eq. (1) cannot be applied directly to orientational coordinates due to the lack of commutativity of finite rotations. Fortunately, an averaging procedure for orientations has been developed recently [7] in terms of an average quaternion, or orientational centroid. In this procedure the orientational centroid, q$_c$ minimizes the function

$$G(q_c) = \sum_{t=1}^{N_s} \Gamma^2(q_c, q_t) \;, \tag{3}$$

where $\Gamma\,(q_c, q_t) = 2cos^{-1}(q_c \cdot q_t)$ is the arc length between the centroid (average) orientation and the orientation q_t. A simple Monte Carlo search algorithm can be used [7] to determine a value of q_c for a set of N_s orientations.

3 Applications to Homogeneous Nucleation

In simulation studies of homogeneous crystal growth, where one attempts to observe the spontaneous formation of crystalline order in an otherwise bulk liquid, the identification and characterization of the critical nucleus is a crucial aspect. Local structural order parameters have been developed and used for this purpose [8,9,10], although the challenge has been to identify order parameters that are both sensitive and generic. The local structural order parameters utilized by Frenkel and co-workers [9,10], based on spherical harmonics, are the most widely used. Here we will only briefly outline how these order parameters are constructed. One starts by defining for each particle i

$$\bar{q}_{lm}(i) = \frac{1}{N_1(i)} \sum_{j=1}^{N_1(i)} Y_{lm}(\hat{r}_{ij}) \tag{4}$$

where $N_1(i)$ is the number of first neighbors of i, Y_{lm} is a spherical harmonics and $\hat{r}_{ij}$ is the unit vector representing the direction of the separation vector joining particles i and j. It has been previously shown [9,10] that the choice of l=6 provides a rather robust measure of local order in most systems. One then generates the normalized 13-dimensional complex vector $\mathbf{q_6}(i)$ from the components $\bar{q}_{6m}(i)$ for each particle i. We subsequently consider the coherence between the measures $\mathbf{q_6}(i)$ and $\mathbf{q_6}(j)$ for the neighboring particles i and j. Specifically, if $\mathbf{q}_6(i) \cdot \mathbf{q}_6(j) > a_{con}$, where a_{con} is some fixed value, then the pair is labeled as connected. The number of connected neighbors of each i, $N_{con}(i)$, is obtained and if $N_{con}(i)$ is greater than some threshold, i is labeled as solid-like.

In utilizing this order parameter, previous workers [9,10,11] have employed the coordinates from instantaneous configurations from their simulations. Here we will test the impact of using (rolling) average coordinates as obtained from Eq. 1. Since the goal is to detect solid-like particles in systems undergoing homogenous nucleation, where the character of particles might be expected to change rapidly, relatively short trajectory segments, composed of N_s=50, 100, 200 and 400 timesteps, were examined. Molecular dynamics simulations of systems of 4000 Lennard-Jones (LJ) particles were carried out at a reduced density of 0.95. Particles were considered to be first neighbors if their separation was less than 1.5σ.

It is first necessary to identify appropriate values for the parameters a_{con} and $N_{con}(i)$. For this purpose simulations were performed at a reduced temperature of 0.65 for a LJ liquid and an FCC crystalline solid. Fig. 1 compares probability distributions functions for $\mathbf{q}_6(i) \cdot \mathbf{q}_6(j)$ recorded when instantaneous coordinates are used with those obtained from averaged coordinates. It can be seen that while the distributions from the liquid systems are relatively unaffected by local time averaging (coarse-graining), there is considerable sharpening and shifting of the distributions towards their ideal value (1.0) for crystalline systems. Consequently, the overlap between liquid and solid distributions is significantly reduced (by at least two orders of magnitude) even when configurations averaged over as few as 100 timesteps are employed. The crossover points in Fig. 1 were utilized to provide appropriate values for the connection threshold, a_{con}, in each case.

Fig. 2 shows probability distributions functions for N_{con} obtained when coordinates from instantaneous and averaged configurations are used. We see a rather dramatic improvement in the resolution of the distributions from crystalline and liquid systems as local time averaging (coarse-graining) is enabled. The distributions from the liquid simulations consistently shift to the left (to lower values) as more averaging is performed. While the solid distribution appears somewhat broad and spans the full range of values when instantaneous coordinates are utilized, all the levels of averaging shown in Fig. 2 produce a distribution that has essentially become a delta function, centered at its ideal value of 12. The obvious conclusion is that the sensitivity of this order-parameter, and hence ones ability

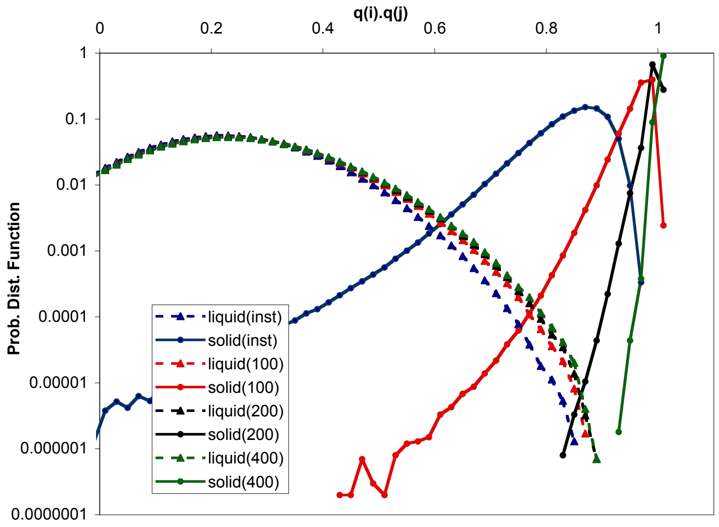


Fig. 1. Probability distribution functions for values of $\mathbf{q}_6(i) \cdot \mathbf{q}_6(j)$ from simulations of liquid (dashed lines) and FCC solid (solid lines) LJ systems at a reduced temperature of 0.65. The blue lines represent data obtained from instantaneous configurations, while the red, black and green lines are results obtained from coordinates averages over 100, 200 and 400 timesteps, respectively.

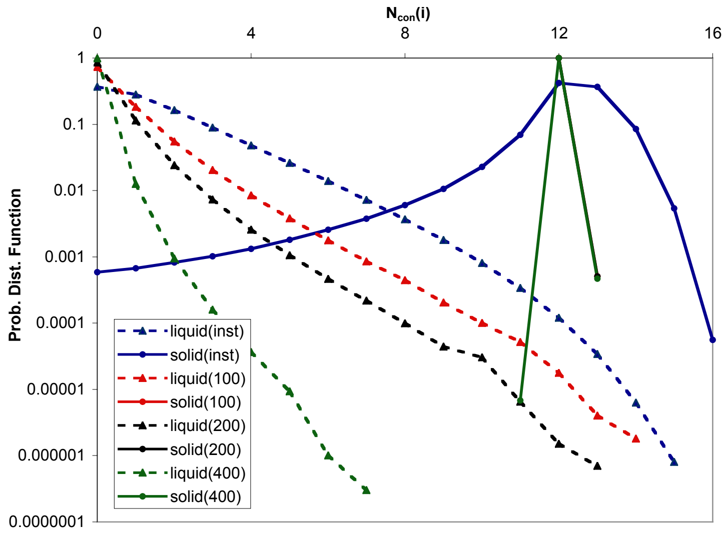


Fig. 2. Probability distribution functions for values of N_{con} from simulations of liquid and FCC solid LJ systems at a reduced temperature of 0.65. The lines are defined as in Fig. 1. It should be noted that all the lines from averaged configurations for an FCC solid are superimposed on this plot.

to distinguish between solid-like and liquid-like particles in these simulations, has been greatly enhanced. Even with only modest amounts of averaging (i.e. 100 timesteps) the overlap between liquid and solid distributions is significantly reduced, i.e. by two orders of magnitude.

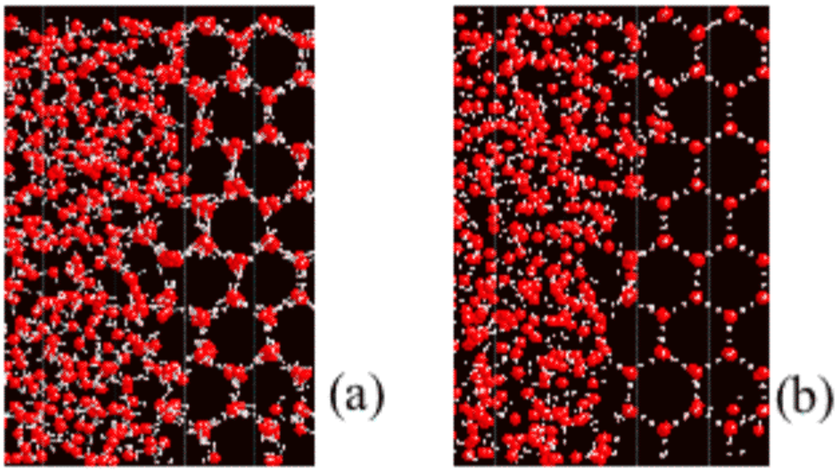


Fig. 3. (a) Instantaneous and (b) averaged configurations of the same interfacial region of an ice/water system during crystal growth. The averaging in (b) is over 20 ps.

4 Applications to Heterogeneous Crystal Growth

We have employed averaged configurations extensively in our molecular simulation studies of heterogeneous crystal growth [12,13,14], where the detailed analysis of multiple nanosecond trajectories is required to uncover the underlying processes associated with crystal growth. The details of the simulation methodology we have employed can be found elsewhere [12,15]. Fig. 3 compares an averaged configuration from a 20 ps trajectory segment with the instantaneous configuration from the end of this trajectory segment. The ice/water system pictured in Fig. 3 is looking down the c-axis of hexagonal ice (I) during its crystal growth. Perhaps the most striking aspect of the averaged configuration is the clarity of its crystalline structure and the distinctiveness of the interfacial layer. Clearly even at this level, the averaged configuration is providing a superior view of the systems behaviour. We point that while some molecular overlaps can oc-

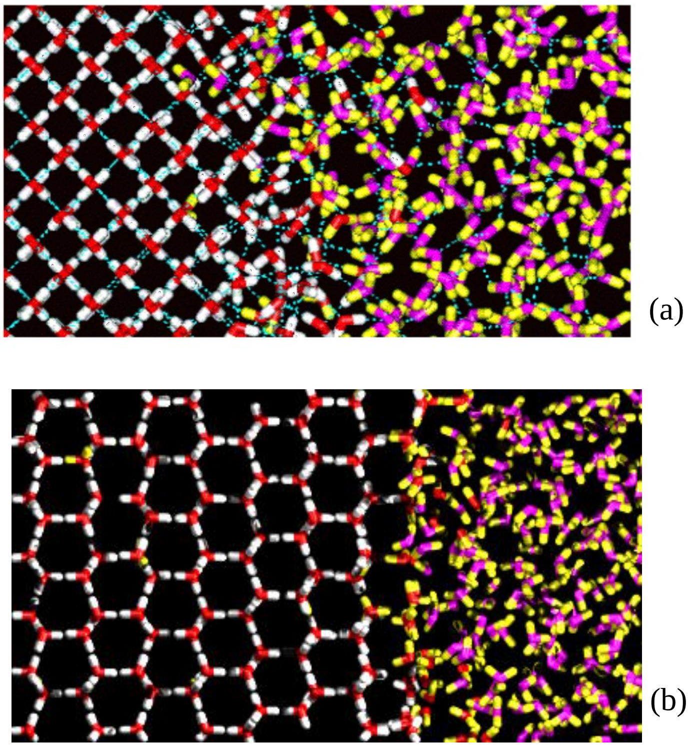


Fig. 4. (a) Averaged configuration of the interfacial region of the [001] face of a growing cubic ice (I) crystal. A trajectory segment of 25 ps was used to produce the averaged positions and orientations of the molecules. The blue dashed lines represent hydrogen-bonded molecules. (b) Averaged configuration of the interfacial region of the [0001] face of a growing hexagonal ice (I) crystal. The length of averaging trajectory segment was 75 ps. The molecules are colored as discussed in the text.

cur in an averaged configuration within the liquid region of a system (due to the diffusive motion of these particles), these are not problematic in our analysis.

To enhance further our ability to extract visual information from averaged configurations, we have labeled water molecules within each averaged configuration as being translationally solid-like or liquid-like, and similarly as being rotationally solid-like or liquid-like. This was done by identifying appropriate thresholds for the RMS deviations (i.e., diffusive behaviour) for both positions and orientations that are consistent with values found in the bulk crystal; molecules with values above these thresholds are identified as being liquid-like. The molecules could be then colored according to their solid-like/liquid-like labels. Specifically, if a molecular was labeled translationally solid-like, its oxygen was colored red, otherwise the oxygen was colored magenta. If a molecule was labeled rotationally solid-like, its hydrogens were colored white, otherwise the hydrogens were colored yellow. We can see from Fig. 4, where the interfacial region of an averaged configuration for two ice/water systems are shown, that these labels provide considerable insight it the molecular behaviour at the interfaces. It should also be noted that the qualitative characteristics observed in Fig. 4 do not apparently change in any significant way if the length of the trajectory segment is changed somewhat (e.g. increased from 25 to 75 ps, as in Figs. 4(a) and 4(b), respectively). This indicates that as long as a reasonable choice for the length of time coarse-graining is utilized, the results obtained are rather insensitive to this value.

5 Conclusions

We have shown that averaged configurations, representing a time coarse-graining over the trajectory of a system, can be very useful for following the molecular mechanisms of crystal growth. The inclusion of second-moment information was also observed to add significant depth to the information contained in these averaged configurations. We have used this approach extensively [12,13,14,16] to studying crystal growth of pure and mixed crystals of both atomic and molecular systems. We would expect that it would prove similarly useful in simulations of other systems characterized by rather slow processes, for example in the folding of a protein or in the transport of an ion across a membrane.

References

1. Bergethon, P.R.: The Physical Basis of Biochemistry: The Foundations of Molecular Biophysics. Springer, Heidelberg (2000)
2. Becker, O.M., MacKerell, A.D., Roux Jr., B., Watanabe, M.: Computational Biochemistry and Biophysics, Marcel Dekker, USA (2001)
3. Lyubartsev, A.P., Karttunen, M., Vattulainen, I., Laaksonen, A.: Soft Mater. 1, 121–137 (2003)
4. Nielsen, S.O., Lopez, C.F., Srinivas, G., Klein, M.L.: J. of Phys.: Cond. Matter 16, R481–R512 (2004)

5. Ortiz, V., Nielsen, S.O., Klein, M.L., Discher, D.E.: J. of Polymer Sc., B: Polymer Phys. 44, 1907–1918 (2006)
6. Srinivas, G., Discher, D.E., Klein, M.L.: Nature Materials 3, 638–644 (2004)
7. de la Peña, L.H., Kusalik, P.G.: Mol. Phys. 102, 927–937 (2004)
8. Steinhardt, P.J., Nelson, D.R., Ronchetti, M.: Phys. Rev. B 28, 784 (1983)
9. ten Wolde, P.R., Ruiz-Montero, M.J., Frenkel, D.: Phys. Rev. Lett. 75, 2714 (1995)
10. ten Wolde, P.R., Ruiz-Montero, M.J., Frenkel, D.: J. Chem. Phys. 104, 9932 (1996)
11. Desgranges, C., Delhommelle, J.: J. Am. Chem. Soc. 128, 10368–10369 (2006)
12. Razul, M.S.G., Tam, E.V., Lam, M.E., Kusalik, P.G.: Mol. Phys, 103, 1929 (2005)
13. Razul, M.S.G., Hendry, J.G., Kusalik, P.G.: J. Chem. Phys. 123, 204722 (2005)
14. Vatamanu, J., Kusalik, P.G.: J. Phys. Chem. B. 110, 15896 (2006)
15. Vatamanu, J., Kusalik, P.G.: J. Chem. Phys. (accepted)
16. Vatamanu, J., Kusalik, P.G.: J. Am. Chem. Soc. 128, 15588 (2006)

Atomistic Simulation Studies of Polymers and Water

Erik Johansson and Peter Ahlström

School of Engineering, University College of Borås, SE-501 90 Borås, Sweden
peter.ahlstrom@hb.se

Abstract. A Monte Carlo simulation study of water and hydrocarbons aiming at understanding the degradation of polyethylene cable insulation is presented. The equilibrium distributions and clustering of water in vapour and in hydrocarbons was investigated using Gibbs ensemble Monte-Carlo simulations. Different combinations of water and hydrocarbon models are investigated in order to reproduce experimental densities of water and hydrocarbons in both the water phase and the hydrocarbon phase.

1 Introduction

The degradation of polyethylene insulation of high voltage DC cables in the presence of water is an important yet incompletely understood problem. It is found to proceed via the formation of water trees, in which water penetrates into the cable insulation [1]. In order to separate the different possible causes for the degradation of polyethylene in real cables, a series of simulations of increasing complexity is performed. We start with simple systems like water and hydrocarbons, to fine-tune simulation parameters to experimental data before the complexity of long polymers is introduced.

In an earlier study [2] we focused on the properties of liquid water in equilibrium with its own vapour since these properties will be of high relevance to the behaviour of water in hydrophobic hydrocarbons where the water properties could be expected to be similar to those of water vapour. This study gave valuable information about the structure of water vapour. The occurence of water clusters with different sizes was monitored and some results are summarized in Fig. 1.

In the present work we study the equilibrium between pure water and decane (which acts as a model hydrocarbon). In the first, preliminary study we compare the properties of pure water vapour with water in decane, in both cases in equilibrium with pure water liquid. Especially the clustering of water molecules was compared.

In these studies water and decane are modelled by the well-known SPC/E [3] and TraPPE [4] potentials, respectively. Since these force fields, with a simple Lorentz-Berthelot mixing rule for water-decane interactions, did not accurately reproduce water densities in the hydrocarbon phase, the second part of this

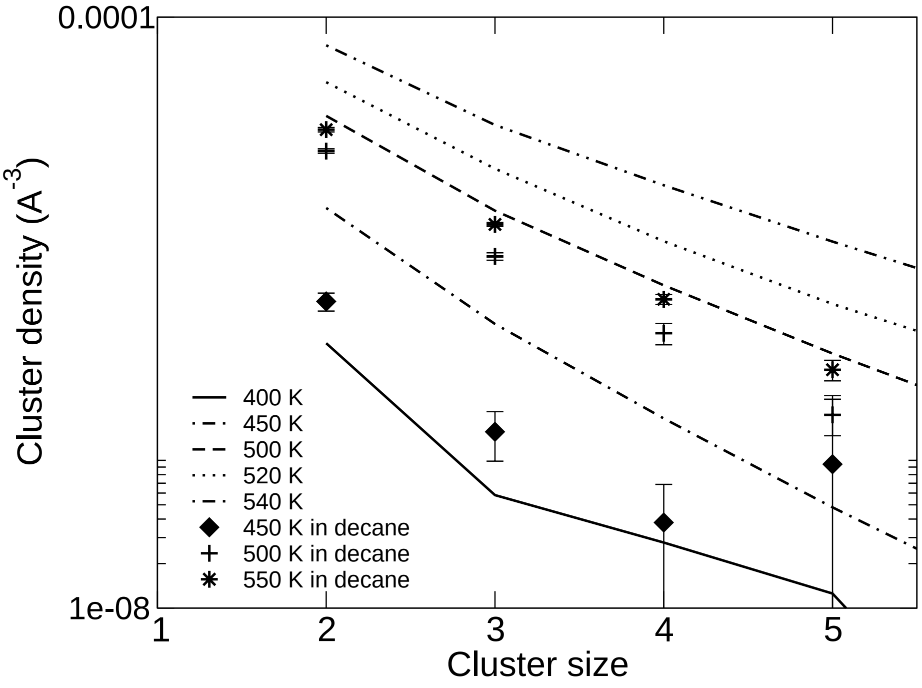


Fig. 1. Density of (SPC/E) water clusters (in number of clusters/Å^3) as a function of cluster size. Note the logarithmic scale.
The curves refer to water clusters in pure vapour whereas the symbols refer to water clusters in decane (modelled by the TraPPE potential). The water densities in decane are smaller than in pure water vapour but tendencies are similar, cf. text.

paper is devoted to optimizing the interaction parameters between water and hydrocarbons to reproduce the equilibrium distribution of water and short chain hydrocarbons (hexane to hexadecane). The SPC [5] water force field and the Karayannis [6] alkane force field are also included in the study.

2 Methods

2.1 Simulation Methods

The choice of simulation method depends both on the system (i.e. what methods are possible) and the properties of interest (i.e. what methods are desirable). In this work different Monte Carlo (MC) methods have been used to study the interaction between alkanes and water.

In order to study phase equilibria the Gibbs Ensemble Monte Carlo (GEMC) method was developed by Panagiatopoulos [7]. In this method, the two phases in equilibrium with each other are simulated in each of two boxes. The pressure is kept constant whereas the volumes of the two boxes are allowed to vary.

In addition to ordinary translation, rotation and regrowth moves inside each box, it also includes trial moves where the molecules are transferred between the boxes. The GEMC method has frequently been used to simulate liquid-vapour equilibria but in the present study it is used to study a distribution equilibrium with two liquid boxes, one intially containing a liquid hydrocarbon and one containing liquid water. The configurational bias method [8] was used to enhance the acceptance rate for insertions in dense phases. All simulations presented here were performed using the Gibbs Ensemble program by Errington and Panagiatopoulos [9] which was slightly modified to incorporate an analysis of water clusters.

2.2 Force Fields

For all studies in this work pair-additive empirical force fields were used. The reason for this choice was their simplicity and the possibility to compare with several other studies of the pure compounds.

There exist several simple force fields that well reproduce properties of water like the SPC/E [3] and TIP4P [10] force fields. These force fields are optimized to reproduce the thermodynamic properties of liquid water. This means, among others, that the dipole moment is higher than *in vacuo* in order to account for the effective polarization of the water molecules in the liquid. The SPC/E model was used for most of the studies reported here since it allows for comparison with previous simulations of liquid-vapour phase equilibria [2].

Most simulations presented here were made using the the TraPPE united-atom force field [4] for the alkane molecules. This force field is known to reproduce experimental phase properties for different alkanes and alkane mixtures. Another force field, that reproduces experimental properties of polyethylene, was developed by Karayannis *et al.* [6] and is here called the Karayannis model. It is a hybrid of the Asymmetric United Atom (AUA) model [11] and the TraPPE model. These two models overestimate the density of the polymer by a few percent [6], while the Karayannis model shows very good agreement with experimental results. One significant difference between the Karayannis model and the TraPPE model is that the Lennard-Jones well-depth (ϵ) of the CH_3 end groups in the TraPPE model is approximately twice as large as for the CH_2 groups, where it is the same for all carbon groups in Karayannis model. This works well in a polymer, with long chains and only few end groups, but we have found that the cohesive energy of the Karayannis model is too small when it is used for shorter alkanes. For example the liquid density of hexane, where one third of the carbon groups are end groups, is only one eighth of the experimental density. The TraPPE hexane density is in very good agreement with experiment. To improve the Karayannis model we have used the ϵ of the TraPPE model for the end groups (98K instead of the original 46K), which transforms the model to one that gives densities that only differs a few percent from experinental values. This means that the results presented here from the “Karayannis” model are, in fact, from a slightly modified version of the model so that it can be used in alkane systems.

None of the force fields contain interaction parameters between water and hydrocarbons. A common way to estimate these parameters is to use the Lorentz-Berthelot rules [12]

$$\sigma_{ij} = \frac{\sigma_{ii} + \sigma_{jj}}{2} \tag{1}$$

$$\epsilon_{ij} = \sqrt{\epsilon_{ii}\epsilon_{jj}} \tag{2}$$

where σ_{ij} and ϵ_{ij} are the Lennard-Jones parameters for the interaction between species i and j whereas σ_{ii} and ϵ_{ii} are the parameters for two atoms of the same species.

However, Equation 2 does not accurately represent the interaction when some of the molecules are polar and in order to better reproduce the experimental distribution of water and decane between the phases Equation 2 was modified to

$$\epsilon_{ij} = B\sqrt{\epsilon_{ii}\epsilon_{jj}} \tag{3}$$

where B is an *ad-hoc* parameter which is fitted to reproduce the experimental partition coefficient.

3 Results and Discussion

In the exploratory study the SPC/E-model of Berendsen *et al.* [3] was used for water. For the alkanes, the TraPPE [4] united atom model was used and the Lorentz-Berthelot combination rules, Eqs. (1)-(2) were used for the interaction parameters between different species. In Fig. 1 the number densities of water clusters of different sizes in the decane phase are shown as symbols. A comparison with the data for pure water vapour at the same temperatures shows that the density of water clusters of a given size is similar but lower in decane. In addition, the ratio between the clustering in neat vapour and in decane increases with increasing cluster size, probably due to the lack of large voids in the decane liquid. This ratio increases more rapidly at lower temperatures since the number of voids of a given size in pure decane decreases with temperature and the energy barrier to increase the size of a void is more easily overcome as the temperature increases.

The overall density of water in the decane phase was much lower than the experimental values at all temperatures, *c.f.*, open diamonds in Fig. 2, which shows the solubility of SPC/E water in decane at different temperatures. This can be seen as an effect of either the attractive part of the alkane/water interaction (i.e., ϵ_{ij}) being too weak or the water-water attraction being too strong.

Using a stronger water-alkane attraction ($B = 1.68$ in Eq. 3) improved the water in decane solubility obtained from the TraPPE and SPC/E force fields (open squares in Fig. 2). However, at 550 K a one phase system was obtained

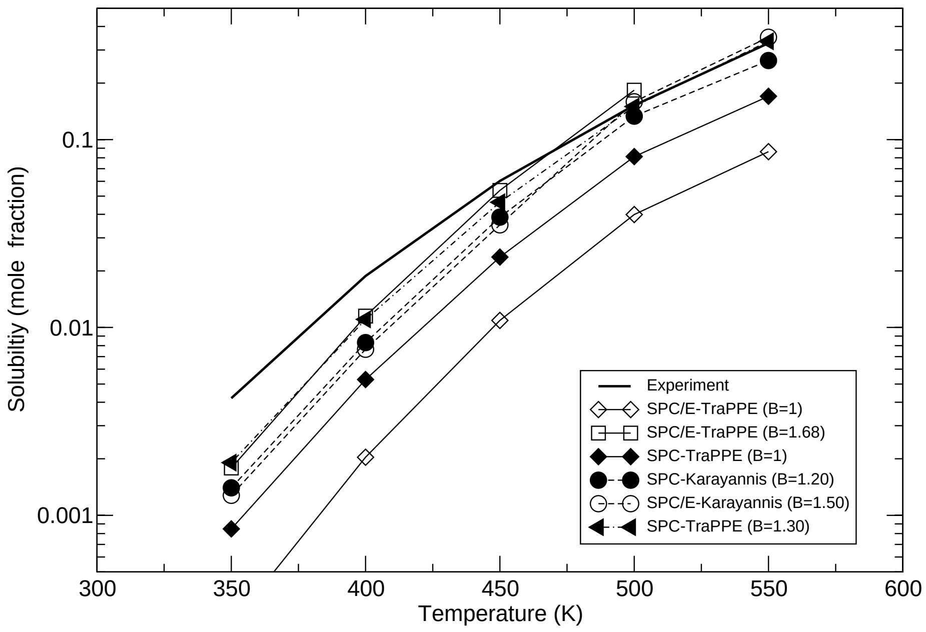


Fig. 2. Solubility (in mole fractions) of water in decane for different combinations of models as a function of temperature. Note the logarithmic scale. B is defined in Eq. 2. $B = 1$ corresponds to the original Lorentz-Berthelot rules. Experimental data are from Tsonopoulos [13].

so the critical temperature for this combination of models is too low. According to Tsonopoulos [13] the experimental critical solution point for a water - decane system is close to 630 K and 20 MPa. In addition, this model ($B = 1.68$) led to a far too high solubility of decane in water. Therefore these parameters were not used for the water-alkane simulations.

Instead, the model was optimized by diminishing the charges on water by a few percent to those of the SPC model, since the polarization of water in alkanes could be expected to be notably smaller than the polarization in pure water (for which the SPC/E model was optimized). This, together with a value of $B = 1.3$, yields good solubilities in both phases (*cf.* Fig. 2 for water in decane).

A comparison was also made for the solubilities of water in alkanes of different chain lengths at 450 K, cf. Fig. 3. In agreement with experiment the mole fraction dissolved water is essentially constant for the short chain alkanes. Also here it was found that the use of the TraPPE alkane model together with the SPC/E water model (with $B = 1.68$) yields a good representation of the mutual solubilities of water and alkanes.

The results for the Karayannis model are very similar to those for the TraPPE model, which is expected due to the large similarity of the models, especially for longer chain alkanes where the influence of the terminal groups is small.

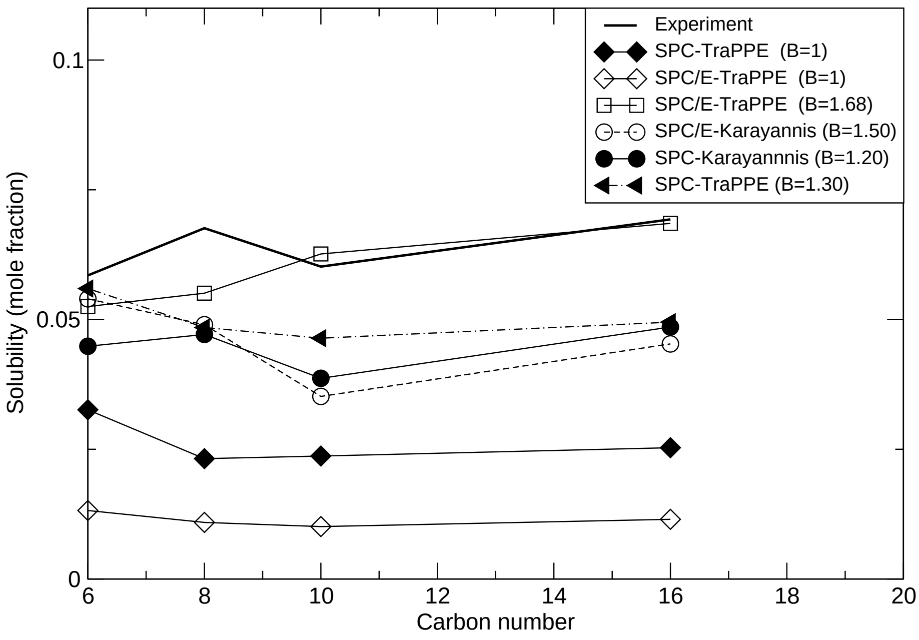


Fig. 3. Solubilities of water in alkanes as a function of chain length at 450 K and 30 bar (expressed as mole fractions) B is defined in Eq. 2. $B = 1$ corresponds to the original Lorentz-Berthelot rules. Experimental data are from Tsonopoulos [13].

4 Conclusions

An important conclusion of this study is that water clustering is similar in alkanes and neat vapour, especially when the water concentration in the oil (or vapour) is low. Similar to the neat vapour, the fraction of clusters dissolved in alkane decreases with increasing cluster size. However, the number of clusters of a given size is always smaller in decane than in neat vapour and this deviation increases with cluster size. This is expected since the number of voids in the pure alkane decreases rapidly with increasing void size.

In order to properly model the solubility of water in alkanes, the simple combination of the SPC/E model and good alkane models is not sufficent. This is partly due to the fact that water is a highly polarizable molecule and it is polarized in the liquid phase which is implicitly included in in the SPC/E model. Water in alkane, on the other hand, is not expected to be as polarized and thus the less polarized SPC model yields a good representation of the water partition if the van der Waals attraction between water and alkane also is increased by about 30%. An even better solution might be to explicitly include polarizabilities, but this is not feasible yet for modelling large water-polymer systems.

Acknowledgements

The authors wish to thank Prof. Kim Bolton for fruitful discussions. The hospitality of Prof. Doros N. Theodorou, National Technical University, Athens, Greece and discussions with him and his group are gratefully acknowledged as are discussions with Prof I. Economou of National Centre for Scientific Research Demokritos, Athens, Greece. Financial support from the Swedish National Research School in Materials Science is gratefully acknowledged.

References

1. Dissado, L.A., Fothergill, J.C.: Electrical degradation and breakdowm in polymers. Peter Peregrinus Ltd, London (1992)
2. Johansson, E., Bolton, K., Ahlström, P.: Simulations of Vapor Water Clusters at vapor-liquid equilibrium. Journal of Chemical Physics 123, 24504 (2005)
3. Berendsen, H.J.C., Grigera, J.R., Straatsma, T.P.: The Missing Term in Effective Pair Potentials. J. Phys. Chem. 91, 6269–6271 (1987)
4. Martin, M.G., Siepmann, J.I.: Transferable potentials for phase equilibria. 1. United-atom description of n-alkanes. J. Phys. Chem. B 102, 2569–2577 (1998)
5. Berendsen, H.J.C., Postma, J.P.M., van Gunsteren, W.F., Hermans, J.: In: Pullman, B. Intermolecular Forces, Reidel, Dordrecht, p. 331 (1981)
6. Karayannis, N.C., Giannousaki, A.E., Mavrantzas, V.G., Theodorou, D.N.: Atomistic Monte Carlo simulation of strictly monodisperse long polyethylene melts through a generalized chain bridging algoritm. Journal of Chemical Physics 117, 5465–5479 (2002)
7. Panagiotopoulos, A.Z.: Direct Determination of Phase Coexistence Properties of Fluids by Monte-Carlo Simulation in a new ensemble. Molec. Phys. 61, 813–826 (1987)
8. Siepmann, J.I., Frenkel, D.: Configurational Bias Monte-Carlo - A New Sampling Scheme for Flexible Chains. Mol. Phys. 75, 59–70 (1992)
9. Errington, J., Panagiotopoulos, A.Z.: Gibbs Ensemble Monte Carlo Program fetched (October 2005),
   available at `http://kea.princeton.edu/jerring/gibbs/index.html`
10. Jorgensen, W.L., Chandrasekhar, J., Madura, J.D., Impey, R.W., Klein, M.L.: Comparison of simple potential functions for simulating liquid water. J. Chem. Phys. 79, 926–935 (1983)
11. Toxvaerd, S.: Molecular-Dynamics Calculation of the Equation of State of Alkanes. J. Chem. Phys. 93, 4290–4295 (1990)
12. Lorentz, H.A.: Ann. Phys. 12, 127 (1881), Berthelot, D.C.R.: Hebd. Seances Acad. Sci. 126, 1703 (1898)
13. Tsonopolous, C.: Thermodynamic analysis of the mutual solubilities of normal alkanes and water. Fluid Phase Equilibria 156, 11–34 (1999)

A New Monte Carlo Method for the Titration of Molecules and Minerals

Christophe Labbez[1,2] and Bo Jönsson[2]

[1] Institut Carnot de Bourgogne, UMR 5209 CNRS,
Université de Bourgogne, F-21078 Dijon Cedex, France
[2] Theoretical Chemistry, Chemical Center, POB 124, SE-221 00 Lund, Sweden
Christophe.labbez@teokem.lu.se

Abstract. The charge state of molecules and solid/liquid interfaces is of paramount importance in the understanding of the reactivity and the physico-chemical properties of many systems. In this work, we porpose a new Monte Carlo method in the grand canonical ensemble using the primitive model, which allows us to simulate the titration behavior of macromolecules or solids at constant pH. The method is applied to the charging process of colloidal silica particles dispersed in a sodium salt solution for various concentrations and calcium silicate hydrate nanoparticles in a calcium hydroxide solution. An excellent agreement is found between the experimental and simulated results.

1 Introduction

For the sake of simplicity we shall consider a planar solid/liquid interface. The solid surface is defined by explicit titratable sites (hard spheres or points) distributed on a square lattice with the same intrinsic dissociation constant K_0. The liquid phase is described by the so called primitive model [1] where the ions are charged hard spheres embedded in a dielectric continuum that represent the solvent (for more details see [2]. The titration can be depicted in the usual way,

$$M - OH \rightleftarrows M - O^- + H^+ \tag{1}$$

from which the intrinsic dissociation contant can be expressed,

$$K_0 = \frac{a_{M-OH}a_H}{a_{M-OH}} \tag{2}$$

where a is the activity. The ordinary method to conduct a titration simulation [3] consists of many attempts of deprotonation and protonation, where the trial energy for these can be expressed as,

$$\beta \Delta U = \beta \Delta U^{el} \pm \ln 10(pH - pK_0) \tag{3}$$

where ΔU^{el} is the change in electrostatic free energy when a site is protonated or deprotonated and $\beta = 1/k_B T$. The proton is implicitely treated through the constant pH. For this purpose a negative charge is assigned to a titrating site when it is deprotonated and vice versa. In order to satisfy the electroneutrality of the system a free charge is added/removed from the simulation cell according to the charge state of the surface site. That is, when a site is ionised the electroneutrality can be maintained by following two distinct procedures either adding a cation or removing an anion, that we shall call in what is following addition cation (ACP) and deletion anion procedure (DAP). From this and as it will be illustrated later, we immediately note that the ordinary titration method, Eq. 3, is wrong by an energy corresponding to the use of the free positive or negative charge and as a result, the two procedures gives different results. Indeed, the chemical potential of the surface site, $\mu = k_B T \ln 10(pH - pK_0)$, is mixed with that of simple ions. Ullner et al. [6] have first developed a self consistent method which consists in calculating $pH - pK_0$ at a set ionisation fraction in the canonical ensemble avoiding the simple ion contribution. In this work, a grand canonical titration method (GCT) is proposed that properly correct for the latter and enable us to perform titration simulation at constant pH. It will be shown that, in the framework of the ordinary titration method and depending on the charge state of the surface, the ACP or DAP can lead to results close to the exact solution obtained through the GCT.

2 Grand Canonical Titration

In this section the two possible procedures, i.e. ACP and DAP, in the framework of the GCT for a surface next to a 1:1 salt (AB) are described. Similar procedures for multivalent salts can be easily extrapolated from this work.

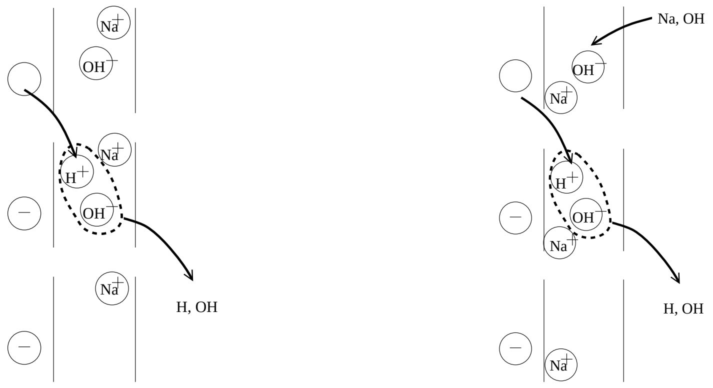


Fig. 1. Illustration of the ionisation procedure in the framework of the GCT by: a) the deletion of an anion, here OH^-; b) by the addition of a cation, here Na^+

2.1 Deletion of an Anion

In the case of DAP the ionisation of a surface site can be written as two steps. That is, the deprotonation of the surface and the exchange of the ion couple (H^+, B^-) with the bulk. This is illustrated in Figure 2-a. The corresponding Boltzmann factor of the trial energy can be expressed as,

$$\exp(-\beta\Delta U) = \frac{N_B}{V}\exp(-\beta\mu_B)\exp(-\beta\Delta U^{el})\exp\{+\ln 10(pH - pK_0)\} \quad (4)$$

where μ represents the chemical potential of the considered ion. A similar expression can be found for protonation as,

$$\exp(-\beta\Delta U) = \frac{V}{N_B+1}\exp(+\beta\mu_B)\exp(-\beta\Delta U^{el})\exp\{-\ln 10(pH - pK_0)\} \quad (5)$$

In these two equations, one can remark that only a prefactor involving the anion, B^-, appears since the contribution of the proton is already taken into account through their last exponential term.

2.2 Addition of a Cation

In the case of ACP, the ionisation of a surface site can be described in three steps, see Fig. 2-b. That is, the addition of the ion couple (A^+, B^-), the deprotonation and the exchange of the ion couple (H^+, B^-) with the bulk. The corresponding Boltzmann factor of the trial energy can be expressed as,

$$\begin{aligned}\exp(-\beta\Delta U) = &\frac{V}{N_A+1}\frac{V}{N_B+1}\exp(\beta\mu_{AB}) \quad (6)\\ &\frac{N_B+1}{V}\exp(-\beta\mu_B)\exp(-\beta\Delta U^{el})\exp\{+\ln 10(pH - pK_0)\}\end{aligned}$$

After some simplifications this equation can be rewritten as,

$$\exp(-\beta\Delta U) = \frac{V}{N_A+1}\exp(+\beta\mu_A)\exp(-\beta\Delta U^{el})\exp\{+\ln 10(pH - pK_0)\} \quad (7)$$

and for protonation,

$$\exp(-\beta\Delta U) = \frac{N_A}{V}\exp(-\beta\mu_A)\exp(-\beta\Delta U^{el})\exp\{-\ln 10(pH - pK_0)\} \quad (8)$$

From eqs (4,5,7,8) one can notice that the GCT method amounts to simply correct the trial energy of titration, Eq. (3), by the excess chemical potential of the "unwanted" ion (the ion used to maintain electroneutrality of the simulation box.

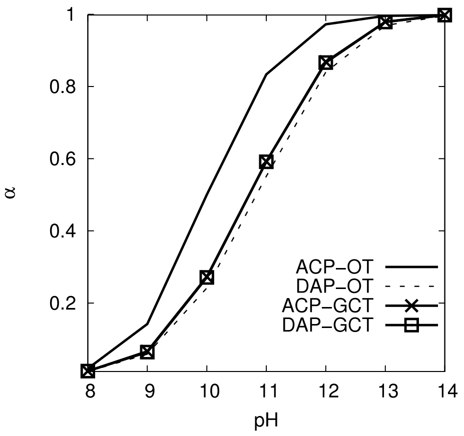


Fig. 2. Ionisation fraction for a surface next to a 20 mM CaX_2 solution varying the pH. The surface site density is set to 4.8/nm^2 and pK_0 to 9.8. ACP: addition cation procedure; DAP: deletion anion procedure; OT: ordinary titration; GCT: grand canonical titration.

3 Results and Discussion

As an example, Fig. 2 presents the simulated ionisation fraction (α) for a surface next to a 20 mM CaX_2 solution as a function of pH. The surface sites are chosen with a density of 4.8 /nm^2 and pK_0 is set to 9.8 in order to correspond to calcium silicate hydrate. The ionisation fraction is obtained using ACP and DAP in the framework of the ordinary titration (Eq. 3) and GCT. In the case of addition of a divalent calcium, an attempt is made to simultaneoulsy deprotonate two sites selected at random. One can immediately notice that in the framework of GCT both procedure, addition of Ca^{2+} and deletion of X^-, gives exactly the same results illustrating the self-consistency of our approach. Interestingly, it can be seen that in the framework of the ordinary procedure and for this pH range, DAP slightly understimates α while ACP largely overestimates it. The opposite takes place, not shown here, for a surface of positive charge density at pH values lower than the pK_0 (for example, considering the equilibrium $M - OH^{+1/2} \rightleftarrows M - O^{-1/2} + H^+$. This is simply explained by the large excess density of counterions near the charged wall, see Eq. (5-9).

In Fig. 3 the simulated and experimental surface charge density against pH for various ionic strengths of silica particles next to a 1-1 salt solution are presented. Experimental details are given in ref. [4]. For simulations, the surface site density and pK_0 is set to 4.8/nm^2 and 7.5, respectively. The value of pK_0 has been taken from the literature [4,5,7,8,9,10,11]. The site diameter is set equal to that of free charges, i.e. 0.4 nm. As expected, α increases with the ionic strength. Except for high salt concentration at the lowest pH values, the simulation predictions are in very good agreement with the experiments.

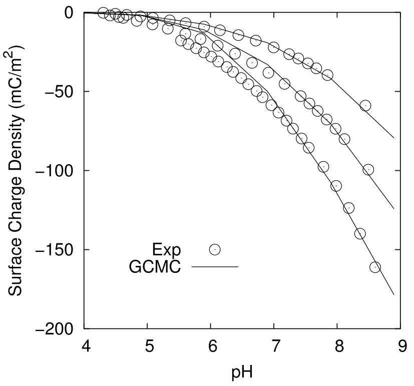


Fig. 3. Surface charge density of silica next to a sodium salt varying the *pH* and for increasing concentration: from top to bottom 10 mM, 100 mM, 1000 mM. empty circles: experiments; full curves: simulations

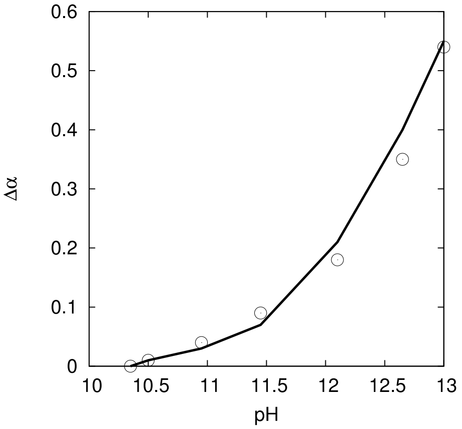


Fig. 4. Comparison between experimental (points) and simulated (line) net increase of the ionization fraction ($\Delta\alpha$) as a function of the *pH* for C-S-H nano-particles dispersed in a solution containing a low bulk calcium concentration

The GCT method has been further confrontated to titration data of calcium silicate hydrate (C-S-H), which is the main constituent of cement paste. The experimental and simulated titrating curves of C-S-H particles dispersed in a 2 mM calcium hydroxide solution are presented on Fig. 4. C-S-H does not exist below *pH* 10 and since its surface is partially ionised for this pH, the net increase of α ($\Delta\alpha$) instead of α is reported here. In simulations, except for pK_0 the parameters are the same as before. The latter is set to 9.8, which corresponds to the first ionisation of silisic acid. For more details see ref. [2]. Again, a very good agreement is found between simulated and experimental $\Delta\alpha$. This indicates that our microscopic model is capable of describing the electrostatic interactions that govern the titration of individual silanol groups leading to the macroscopically observable surface charge.

4 Conclusion

We have proposed a new Monte Carlo method in the grand canonical ensemble called Grand Canonical Titration, which allows us to simulate the titration behavior of molecules or solids at constant pH. The self-consistency of the method has been verified. In addition, the method has been applied to the charging process of both colloidal silica particles in a sodium salt solution and C-S-H nano-particles in a $Ca(OH)_2$ solution. An excellent agreement has been found between the experimental and simulated results. This result is particulary important in the field of colloid science as it shows that the many parameters of the commonly used "classical Stern model" can be reduced to two, namely the surface site density and pK_0. In addition to being more simple, our approach appears to have a clearer physical basis.

Acknowledgments. A generous grant from the Swedish Institute and the Swedish Foundation for Strategic Research is gratefully acknowledged. We also acknowledge the support of the CRI from the University of Bourgogne (France) and the LUNARC (Sweden) for access to their computer facilities.

References

1. Frenkel, D.: Understanding molecular simulation. Computational Science Series, vol. 1. Elsevier, Amsterdam (2001)
2. Labbez, C., Jönsson, B., Pochard, I., Nonat, A., Cabane, B.: Surface charge density and electrokinetic potential of highly charged minerals: Experiments and Monte Carlo simulations on calcium silicate hydrate. J. Phys. Chem. B 110, 9219–9230 (2006)
3. Kesvatera, T., Jönsson, B., Thulin, E., Linse, S.: Ionization behavior of acidic residues in calbindin D_9k. Proteins 37, 106–115 (1999)
4. Kobayashi, M., Skarba, M., Galletto, P., Cakara, D., Borkovec, M.: Effects of heat treatment on the aggregation and charging of Stöber-type silica. J. Colloid Interface Sci. 292, 139–147 (2005)
5. Hiemstra, T., Van Riemsdijk, W.H., Bolt, G.H.: Multisite proton adsorption modeling at the solid/solution interface of (hydr)oxides: A new approach: I. Model description and evaluation of intrinsic reaction constants. J. Colloid Interface Sci. 133, 91–104 (1989)
6. Ullner, M., Woodward, C.: Simulations of the titration of linear polyelectrolytes with explicit simple ions: Comparison with screened coulombs models and experiments. Mocromolecules 33, 7144–7156 (2000)
7. Sverjensky, D.A.: Physical surface complexation models for sorption at the mineral/water interface. Nature 364, 776–780 (1993)
8. Iler, R.K.: The Chemistry of Silica. Wiley, New York (1979)

9. Foissy, A., Persello, J.: Surface group ionization on silicas. In: Legrand, A.P. (ed.) The surface properties of silica, pp. 402–411. Wiley, New York (1998)
10. Dove, P.M., Craven, C.M.: Surface charge density on silica in alkali and alkaline earth chloride electrolyte solutions. Geochemica et Cosmochimica Acta 69, 4963–4970 (2004)
11. Hunter, R.J.: Foundations of colloid science, 2nd edn. Oxford University Press, Oxford (1993)

Coarse Graining Biomolecular Systems

Mikael Lund

Institute of Organic Chemistry and Biochemistry,
Academy of Sciences of the Czech Republic,
Flemingovo nam. 2, Prague 6, CZ-16610, Czech Republic
`mlund@mac.com`
`http://www.molecular.cz`

Abstract. Proteins in the living cell can interact with a wide variety of solutes, ranging from ions, peptides, other proteins, DNA to membranes. Charged groups play a major role and solution conditions such as pH and ionic strength can modulate the interactions significantly. Describing these systems in a statistical mechanical framework involves thousands of pair-interactions and therefore a certain amount of coarse graining is often required. We here present a conceptually simple "mesoscopic" protein model where the detailed charge distribution and surface topology is well preserved. Monte Carlo simulations based on this model can be used to accurately reproduce second virial coeffients, pH titration curves and binding constants of proteins.

1 Blurry Proteins

Scientists from different fields use different approximate descriptions of how a protein or biomolecule looks like. This is illustrated in Figure 1 where it is sketched how the same protein may be presented in various disciplines. What is shown is in fact a kind of coarse graining where the most detailed model (quantum level) is transformed into simpler alternatives, tailored to capture only properties of interest. This is a powerful technique as the mathematical complexity can be drastically reduced (or even removed) yet still provide useful insight. It can also be a pitfall, though. It is not uncommon to see one kind of coarse graining applied to problems where it is not suitable. For example, it would be silly to measure protein aggregation using the "cartoon model" as it provides no controlled way of quantifying intermolecular interactions. Likewise it would seem overly excessive to use a full quantum mechanical model as we deal with long range interactions and a detailed description of internal electron densities is not really needed (nor tractable for that matter). The trick is to find the right level for the problem at hand, and at the same time ensure that the results are not artificially affected by an excessive amount of input parameters.

The "biologist", "chemist", and "physicist" models shown in Figure 1 all illustrate – some better than others – that proteins have a detailed surface topology and indeed occupy a volume from which other molecules are excluded. This gives rise to significant intermolecular interactions and any quantifying model should

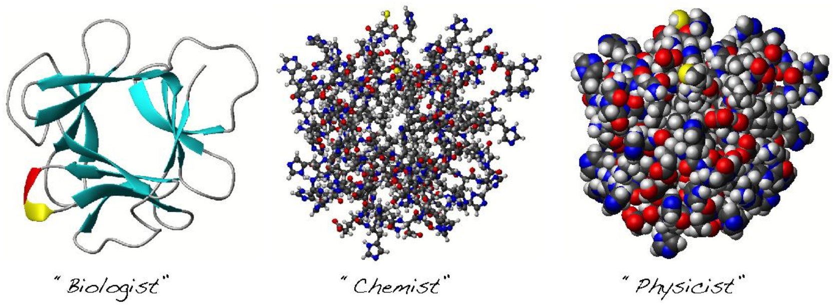


Fig. 1. A protein as seen from three different scientific view points: The "cartoon" model (left) showing secondary structure, the ball-and-stick model (middle) illustrating individual atoms and chemical bonds, and finally a space filling model (right), where each atom is presented by a sphere

capture this effect. From a computational point of view it is appealing to coarse grain as much as possible while still preserving physical properties essential to the problem at hand – in this case the excluded volume. The "colloid scientist" would (possibly) replace the protein with a sphere matching the protein volume so as to reduce the number of particles from thousands to merely one. For many globular proteins this can be a reasonable approximation but is not for more elongated molecules. Another disadvantage of the spherical model is that the detailed charge distribution is usually replaced by a single point charge, thus neglecting possible electric multipole moments. Figure 2 shows other alternatives;

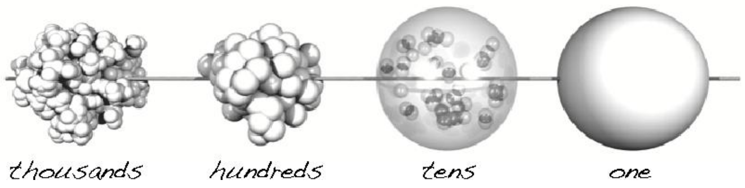


Fig. 2. Various protein models and the number of particles involved. From left to right: 1) Atomistic model where all atoms are represented by spheres. 2) Amino acid model where entire amino acid residues are approximated by spheres. 3) Point charges in an encapsulating sphere. 4) A sphere with a point charge in the middle (DLVO type).

in particular we note the amino acid model where all amino acid residues in the protein are replaced by spheres. This method was scrutinized by Lund and Jönsson[1] and – while seemingly crude – it captures many protein details and is advantageous for several reasons: 1) the surface topology is well preserved (see Figure 3), 2) the detailed charge distribution is maintained, 3) it can incorporate detailed short range interactions (van der Waals, for example), and 4) the number of particles is reduced by one order of magnitude compared to the atomistic representation, allowing even large proteins to be handled.

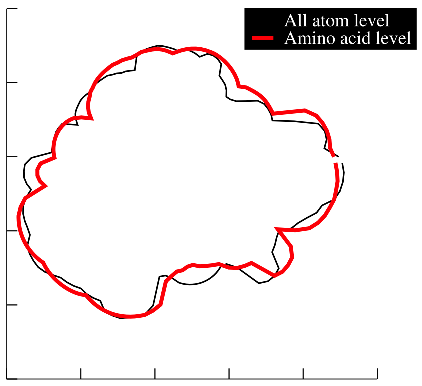


Fig. 3. A cross section of the protein ribonuclease using both an atomistic- and an amino acid representation

1.1 Rigidity

The models proposed so far all neglect structural degrees of freedom within the protein molecule. Clearly this is an approximation as side chains do have some flexibility and certain proteins are known to function via structural perturbations. To incorporate this we need a set of energy functions for stretching, twisting and bending intra-molecular bonds in the protein. Such *force fields* are typically constructed in a semi-empirical manner and may contain hundreds of parameters and it can be difficult to make sure that these do not artificially influence the final results.

In cases where structural fluctuations are minor or irrelevant for the properties of interest the molecule can be kept rigid. For a number of proteins the structure is invariant to even large changes in solution conditions (pH, salt). In theoretical calculations the (fixed) structural coordinates are acquired from either NMR or X-ray crystallography, but these are only partly experimental: After the data collection the experimental constraints (electron density maps or NOE's) are – together with a force field – used in a simulation, so as to obtain a the most probable structure. Therefore the "experimental" structure do hold a reminiscence of theory and should be regarded as an average structure. This is especially true for solution NMR structures where the flexible side chains can appear in a multitude of conformations, but are typically averaged to just one. These facts support the usage of a rigid but *likely* structure.

1.2 More Coarse Graining

So far we have discussed the macromolecule only but protein interactions usually take place in a condensed phase in the presence of solvent and other solutes. One obvious but important feature of an aqueous solution is that the particle density is *high*; the concentration of water in water is roughly 55 mol/l and this will significantly influence any interaction taking place. Computationally this is challenging as one needs to consider thousands of solvent molecules and their mutual interactions throughout phase space. This is manageable by "averaging out" configurational variables[2], leaving us with *effective potentials* that has the character of free energies and can be performed in a step wise manner,

$$\beta w(x) = -\ln \int e^{-\beta U(x,y)} dy = -\ln \left\langle e^{-\beta U(x,y)} \right\rangle_y \tag{1}$$

$$\beta w = -\ln \left\langle e^{-\beta U(x,y)} \right\rangle_{x,y} = -\ln \left\langle e^{-\beta w(x)} \right\rangle_x . \tag{2}$$

where β is the inverse of the thermal energy. For example, for polar fluids structural degrees of freedom can be integrated out to reduce the properties of the enormous number of solvent molecules to a single, macroscopic number, namely the relative *dielectric constant*, ϵ_r. Very appropriately, the solvent is now referred to as a dielectric continuum, indicating that structural details are nonexistent. The (effective) interaction energy between two charges, i and j in the solution is now given by Coulombs law, reduced by a factor of $1/\epsilon_r$:

$$\langle \beta w_{ij} \rangle_{solv} \approx \frac{1}{\epsilon_r} \frac{e^2 q_i q_j}{4\pi\epsilon_0 r_{ij} kT} = \frac{l_B z_i z_j}{r_{ij}} \tag{3}$$

where q is the charge, $z = q/e$ the charge number, e_0 the permittivity of vacuum, k Boltzmann's constant and e is the electron charge. Note the introduction of the Bjerrum length, l_B. In a similar fashion we can continue (Figure 4) to treat salt particles implicitly and arrive at the classic Debye-Hückel result [3],

$$\langle \beta w_{ij} \rangle_{solv,salt} \approx \frac{l_B z_i z_j}{r_{ij}} e^{-\kappa r_{ij}} \tag{4}$$

where the inverse Debye screening length, κ is proportional to the square root of the ionic strength. Equation 3 and 4 are valid for spherical symmetric charges, but for dipolar molecules (such as proteins) angular averaging produce additional terms – dipole-dipole interactions (Keesom energy) for example:

$$\left\langle \beta w_{ij}^{dip-dip} \right\rangle_{solv,angles} \approx -\frac{(l_B \mu_i \mu_j)^2}{3 r_{ij}^6} \tag{5}$$

To derive these expressions, assumptions and mathematical approximations are almost always applied and this of course has consequences. For example, in the Debye-Hückel theory correlations between ions[4] are partly neglected and unphysical results usually appear for strongly coupled systems (highly charged

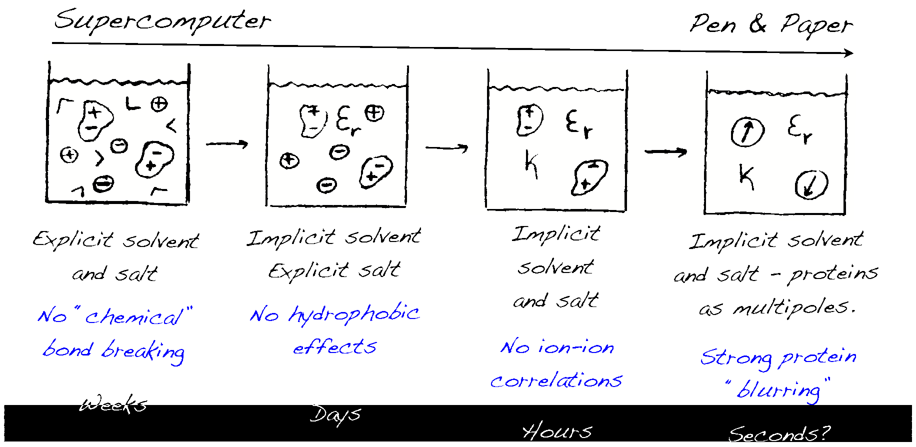

Supercomputer
Pen & Paper
Explicit solvent and salt
No "chemical" bond breaking
Implicit solvent Explicit salt
No hydrophobic effects
Implicit solvent and salt
No ion-ion correlations
Implicit solvent and salt - proteins as multipoles.
Strong protein "blurring"
Weeks
Days
Hours
Seconds?

Fig. 4. Simplifying an aqueous protein solution using statistical mechanical averaging resulting in faster and faster calculations. The blue text indicates the implication of the approximations.

molecules, multivalent salt etc.). It suffices to say that effective potentials reduce the computational cost significantly – usually at the expense of molecular detail; for further information the reader should consult the textbook of Israelachvili [5].

1.3 Dielectric Boundaries

As just outlined averaging out solvent degrees of freedom can drastically reduce the computational cost when simulating liquids. However, this continuum approach brings about another concern: The deep interior of proteins consists mainly of non-polar matter, while the surrounding solvent is highly polar. The effect of this can be captured by solving the Poisson(-Boltzmann) equation for the dielectric boundary between the low- and high dielectric regions. However, this boundary is not well defined nor sharp; surface groups of water soluble proteins are partially charged, polar and polarizable, stemming from proton fluctuations, structural flexibility of amino acid side chains and backbone dipole moments. Further, since the dielectric constant is a macroscopic property influenced by distant molecules, setting up boundaries within a few nanometers may be to push the model too far. Note that it is not uncommon (in fact it is normal) to let the boundary include charges in the low-dielectric protein interior even though the Born energy suggests costs of $\sim$10-100 kT per charge relative to positions just a few Ångströms away. This is typically remedied by partly solvating charges using surface accessible areas and by doing so good agreement can be obtained with experimental data for properties such as side-chain pK_a-values and overall titration behavior.

The above complications suggest a different and simpler approach. If we can agree that surface charges are established in relatively high dielectric regions, electrostatic interactions between them can be conveniently estimated using a uniform dielectric constant equal to that of water. Mathematically, this is trivial as the pair interaction between any two charges will be of the type shown in Eq. 3 or 4 and hence, many-particle systems can be studied relatively easily. Using this approach a number of researchers[1,6,7,8,9,10,11] have obtained good agreement with experimental data – even though effects from the low dielectric interior have been ignored. This indicates that the uniform dielectric approximation is applicable for the overall electrostatic environment but may become less applicable for deeply buried charges.

1.4 A Few Examples

We shall here mention just a few of the applications in which the presented techniques can be used to predict protein solution properties and will not go into great detail of how the systems are solved numerically. Using Metropolis' Monte Carlo (MC) simulation technique[12,13] coordinate space of a given model system can be efficiently sampled, yielding statistical mechanical averages. Coarse graining proteins to the amino acid level and treating salt explicitly, but solvent implicitly (uniform dielectric) a number of properties have been studied as shown in Fig. 5 and Table 1. The virial coefficients, B_2 (Fig. 5, left) are obtained from two-body simulations[1] where the inter-protein potential of mean force, $w(r)$ can be sampled and later integrated to yield the virial coefficient,

$$B_2 = -2\pi \int_0^{\infty} \left(e^{-\beta w(r)} - 1 \right) r^2 dr. \tag{6}$$

When B_2 is negative there is a net attraction between the proteins and is positive they repel each other. Thus, the virial coefficient is interesting in connection with protein aggregation and crystallization and is strongly influenced by solution pH, salt concentration and type. In the case of lysozyme there is a strong repulsion at low pH and salt concentration, where the protein net-charge if positive. As pH is increased this charge gradually diminishes and as the protein goes through its isoelectric point, an attraction occurs. Addition of salt effectively screens out electrostatic interactions and hence the virial coefficient is much smaller.

Due to acidic and alkaline side chains on the protein surface the overall protein charge varies with solution conditions. This can be studied in a simulation of a single protein by letting protons fluctuate between the macromolecule and the bulk solution[15,16]. The resulting average protein charge (Fig. 5, right), dipole moment and charge capacitance[16] can be used to construct an approximate expression[17] for the inter-protein potential of mean force. In addition to being very fast, this approach to protein-protein interactions has an important advantage in that the result can – mathematically – be partitioned into contributions from ion-ion, ion-dipole interactions etc. and as such can be more informative than a "brute-force" two-body simulation. Not only can the overall protonation state be analyzed but it is also possible to examine individual, titrating sites. Table 1 shows calculated values for various residues in the protein ovomucoid

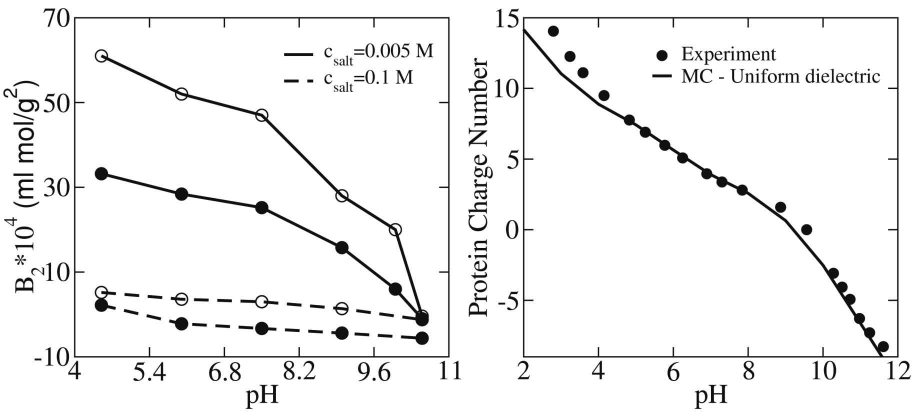


Fig. 5. Comparison between experimental data and results obtained from Monte Carlo simulations using the uniform dielectric model. Left: The virial coefficient for lysozyme at different pH. Open circles represents MC data. Right: Titration behavior of ribonuclease – experimental data from ref. [14].

Table 1. Measured and calculated pK_a values for turkey ovomucoid third domain at low salt concentrations. The MC and MTK calculations are both based on the crystal structure[18] (1PPF). In the simulations, the amino acid model was applied together with a uniform dielectric constant.

	Ideal	MTK[19]		MC	Exp.[20]
		ϵ_p=4, relaxed	ϵ_p=20	ϵ_p=80	
Asp7	4.0	2.1	3.0	3.2	< 2.6
Glu10	4.4	4.0	3.5	3.9	4.1
Glu19	4.4	3.1	2.7	3.4	3.2
Asp27	4.0	2.9	3.7	2.7	< 2.3
Glu43	4.4	5.6	4.7	4.1	4.7
Ctr56	3.8	2.6	3.2	2.5	< 2.5
rms	1.2	0.5	0.7	0.4	

third domain. Besides experimental data the comparison is against an ideal calculation (no electrostatics) and a modified Tanford-Kirkwood (MTK) approach. Despite the high level of coarse graining in the MC simulations, it is on average closer to the experimental results than what is obtained with the MTK method, based on Poisson–Boltzmann electrostatics with both a dielectric boundary and side-chain flexibility.

1.5 Final Remarks

We have here outlined some methods useful for simplifying proteins and their surrounding media – going from tens of thousands to merely hundreds of particles.

Computationally this is a tremendous advantage and despite their naive nature these models indeed are capable of capturing the essential physics of biomolecular systems. An appealing feature of moderating the level of sophistication is that the demand for input parameters is diminished. Yet – as illustrated by the above examples – very competitive results can be obtained.

References

1. Lund, M., Jönsson, B.: A mesoscopic model for protein-protein interactions in solution. Biophys. J. 85, 2940–2947 (2003)
2. Hill, T.L.: An Introduction to Statistical Thermodynamics. Dover Publications Inc., New York (1986)
3. Debye, P., Huckel, E.: Z. Physik 24, 185 (1923)
4. Guldbrand, L., Jönsson, B., Wennerström, H., Linse, P.: Electric double layer forces. a monte carlo study. J. Chem. Phys. 80, 2221 (1984)
5. Israelachvili, J.N.: Intermolecular and Surface Forces, 2nd edn. Academic Press, London (1992)
6. Warshel, A., Russel, S.T., Churg, A.K.: Macroscopic models for studies of electrostatic interactions in proteins: Limitations and applicability. Proc. Natl. Acad. Sci. 81, 4785–4789 (1984)
7. Kesvatera, T., Jönsson, B., Thulin, E., Linse, S.: Binding of ca2+ to calbindin d9k: Structural stability and function at high salt concentration. Biochemistry 33, 14170–14176 (1994)
8. Demchuk, E., Wade, C.: Improving the continuum dielectric approach to calculating pk_a's of ionizable groups in proteins. J. Phys. Chem. 100, 17373–17387 (1996)
9. Kesvatera, T., Jönsson, B., Thulin, E., Linse, S.: Measurement and modelling of sequence-specific pK_a values of calbindin D_{9k}. J. Mol. Biol. 259, 828 (1996)
10. Penfold, R., Warwicker, J., Jönsson, B.: Electrostatic models for calcium binding proteins. J. Phys. Chem. B 102, 8599–8610 (1998)
11. Kesvatera, T., Jönsson, B., Thulin, E., Linse, S.: Ionization behaviour of acidic residues in calbindin d_{9k}. Proteins 37, 106–115 (1999)
12. Metropolis, N.A., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A., Teller, E.: Equation of state calculations by fast computing machines. J. Chem. Phys. 21, 1087–1097 (1953)
13. Frenkel, D., Smit, B.: Understanding Molecular Simulation. Academic Press, London (1996)
14. Tanford, C., Hauenstein, J.D.: Hydrogen ion equilibria of ribonuclease. J. Am. Chem. Soc. 78, 5287–5291 (1956)
15. Ullner, M., Jönsson, B., Söderberg, B., Peterson, C.: A Monte Carlo study of titrating polyelectrolytes. J. Chem. Phys. 104, 3048–3057 (1996)
16. Lund, M., Jönsson, B.: On the charge regulation of proteins. Biochemistry 44(15), 5722–5727 (2005)
17. Bratko, D., Striolo, A., Wu, J.Z., Blanch, H.W., Prausnitz, J.M.: Orientation-averaged pair potentials between dipolar proteins or colloids. J. Phys. Chem. 106, 2714–2720 (2002)

18. Bode, W., Wei, A.Z., Huber, R., Meyer, E., Travis, J., Neumann, S.: X-ray crystal structure of the complex of human leukocyte elastase (pmn elestase) and the third domain of the turkey ovomucoid inhibitor. EMBO 5, 2453 (1986)
19. Havranek, J.J., Harbury, P.B.: Tanford-kirkwood electrostatics for protein modeling. Proc. Natl. Acad. Sci. 96, 11145–11150 (1999)
20. Shaller, W., Robertson, A.D.: ph, ionic strength, and temperature dependences of ionization equilibria for the carboxyl groups in turkey ovomucoid thirs domain. Biochemistry 34, 4714–4723 (1995)

Molecular Dynamics Simulation Studies on the Modulation of Vitamin D Receptor Activity by Agonists and Antagonists

Mikael Peräkylä

Department of Biosciences, University of Kuopio,
P.O. BOX 1627, FIN-70211, Kuopio, Finland
Mikael.Peräkylä@uku.fi

Abstract. Molecular dynamics (MD) simulations of vitamin D receptor (VDR) ligand complexes have been carried out to explain and predict ligands' functional behavior. Elevated simulation temperature, simulated annealing, locally enhanced sampling method, and targeted dynamics were used to speed up the sampling of the conformational space in MD simulations. In addition, self-organizing map and Sammon's mapping algorithm was applied to group and visualize receptor movement upon ligand binding. It was shown that the degree of structural order in the carboxy-terminal α-helix inversely correlated with the strength of the antagonistic activity of the ligand and that a two-side chain analog of vitamin D functions as a potent agonist to the VDR despite its significantly increased volume. Binding of novel nonsteroidal VDR agonists was also investigated. Simulation results were combined with extensive experimental data. In this work theoretical and experimental studies were fruitfully combined to investigate complex receptor regulation.

1 Introduction

The nuclear receptor (NR) for 1α,25-dihydroxyvitamin D3 ($1\alpha,25(OH)_2D_3$), the vitamin D receptor (VDR), binds its ligands with high affinity (K_d is 1 nM or lower). The ligand binding domain (LBD) of VDR is formed of 12 α-helices and its overall architecture is similar for all NRs. A crucial step in the regulation of the biological activity of VDR is the stabilization of the agonistic conformation of the LBD via repositioning of the most carboxy-terminal α-helix (helix 12, Fig. 1. This conformational change is initiated by agonist binding. Helix 12 contains a short transactivation function 2 (AF-2) domain [1]. A glutamate residue of the AF-2 domain forms, together with a lysine residue of helix 3, a charge clamp that positions the nuclear receptor interaction domain of coactivator proteins to a hydrophobic cleft on the surface of the LBD [2]. The correct position of helix 12 is important for the correct distance between the glutamate and lysine residue and influences in this way the efficacy of the NR-coactivator interaction, and eventually, resulting in an enhanced transcription of NR target genes

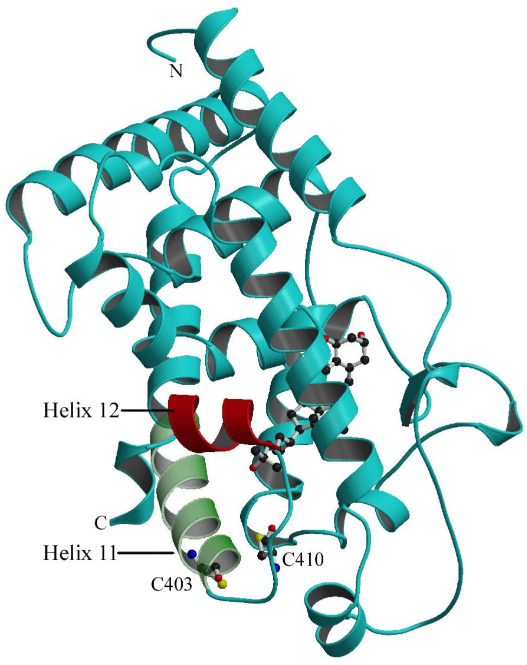


Fig. 1. The structure of the VDR-LBD-1α,25(OH)$_2$D$_3$ complex

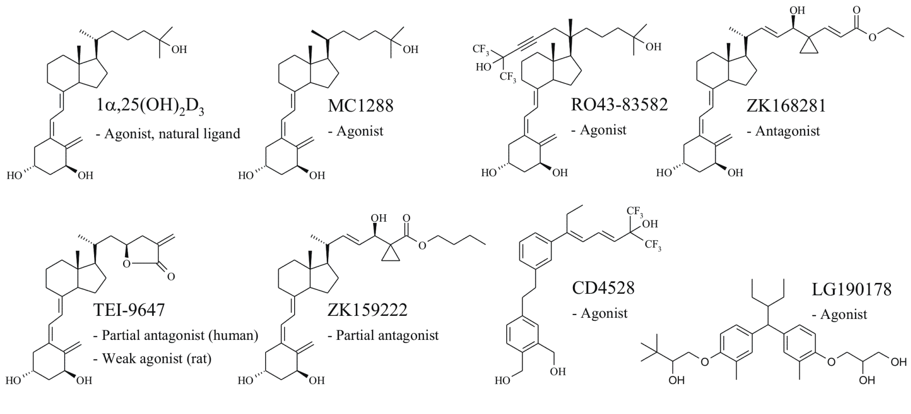


Fig. 2. Structures of agonistic and antagonistic VDR ligands. RO43-83582 (Gemini) is a double side chain agonist, TEI-9647 a partial antagonist for human, but a weak agonist for rodent VDR, and CD4528 and LG190178 are potent nonsteroidal VDR agonists.

[3]. Natural and synthetic molecules that selectively activate or inhibit VDR or other NRs are of considerable biological significance and may have important clinical applications. The complexes formed between a number structurally and

functionally different ligands (Fig. 2) and the LBD of VDR were studied using MD simulations. Special emphasis was placed on finding connections between the structural changes induced by ligand binding and ligand's functional properties (agonism/antagonism). Because the structural changes taking place upon ligand binding are slow compared to the simulation time scale different computational tricks were used to speed up the sampling of the conformational space. To this end, elevated simulation temperature, simulated annealing alone and in combination with locally enhanced sampling (LES) method and targeted dynamics were applied. Similar problems were faced when different ligands (Fig. 2) were docked to the LBP. Simulation results were used to suggest specific mutagenesis experiments and to provide structural data that could be used to rationalize experimental observations. The extensive combination of computational and experimental results was found to results in particularly fruitful insights into the complex regulation of the VDR that would have otherwise been difficult to obtain.

2 Computational Details

The initial coordinates of VDR were obtained from the X-ray crystal structure of the VDR-LBD-1α,25(OH)$_2$D$_3$ complex (Protein Data Bank code 1DB1) [4]. Coactivator peptide KNHPMLMNLLKDN was added to the simulation system and placed on the surface of the VDR-LBD on the basis of (rat) VDR-LBD-1α,25(OH)$_2$D$_3$ complex x-ray structure (1RK3) [5]. Ligands were placed to the ligand-binding site using the VDR-LBD-1α,25(OH)$_2$D$_3$ crystal structure as a model. The two-side chain analog RO43-83582 (Gemini) was docked to the ligand-binding pocket using the locally enhanced sampling (LES) method [6] with five copies of ligand side chains. The structures obtained from the LES simulations were studied further with long MD simulations. Targeted MD method was used to generate an antagonistic receptor conformation. In this method, an additional term is added to the energy function of the system based on the mass-weighted root mean square deviation of a set of atoms in the current structure compared with a reference structure. For the molecular dynamics simulations VDR complexes were solvated by TIP3P water molecules in a periodic box of $61\times69\times86$ Å. Crystallographic water molecules were included in the simulation systems. The water molecules of the complexes were first energy-minimized for 1000 steps, heated to 300 K in 5 ps and equilibrated by 10 ps at constant volume and temperature of 300 K. After that, the simulation systems were minimized for 1000 steps, the temperature of the systems was increased to 300 K in 5 ps and equilibrated for 100 ps The equilibration was carried out at constant pressure (1 atm) conditions. In the production simulations of 2-10 ns the electrostatics were treated using the particle-mesh Ewald method. A timestep of 1.5 fs was used and bonds involving hydrogen atoms were constrained to their equilibrium lengths using the SHAKE algorithm. The simulations were done using the AMBER 7.0/8.0 simulation package [7] and the parm99 parameter set of AMBER. The parameters of the ligands were generated with the Antechamber suite of AMBER in conjunction with the general amber force field [8]. The atomic point charges of the ligands were calculated with the two-stage RESP [9] fit at the HF/6-31G* level using ligand geometries optimized with the semi-empirical PM3 method using the Gaussian03 program [10].

3 Results and Discussion

The carboxy-terminal α-helix, helix 12, of VDR contains a critical ligand-modulated interface for the interaction with coactivator proteins. MD simulation were done for the natural VDR agonist 1α,25(OH)$_2$D$_3$, a partial antagonist ZK159222 and a complete antagonist ZK168281. Because displacement of the helix is a slow process which is difficult to observe in MD simulations, a simulation of 1 ns at 300 K was first carried out followed by another 1 ns at 340 K. After the 2 ns MD the structural differences induced to the LBD by the three functionally different ligands were clear. It was observed that, as expected, helix 12 stayed in the agonistic conformation during the VDR-1α,25(OH)$_2$D$_3$ simulation, even after 1 ns at 340 K. The average structure of the last 500 ps of the simulation was still close to the x-ray structure of the same complex [4]. In contrast, the structure of helix 12 of the ZK159222 and ZK168281 complexes was clearly distorted compared to the experimental structure. In the case of the complete antagonist, the root-mean square deviation (RMSD) of the Cα atoms (compared to the x-ray structure) of the helix 12 (residues 417-423) was during the 1 ns simulation at 340 K about 3 Å, while it was about 1 Å in the 1α,25(OH)$_2$D$_3$ simulation, confirming the antagonistic nature of ZK168281. In the case of ZK159222 the RMSD varied between 1-3 Å during the simulation, as would have been expected for a partial antagonist. The distance between two charged residues, Glu420 of the helix 12 and Lys246 is critical for the interaction between the VDR ligand-binding domain and coactivator. In the case of 1α,25(OH)$_2$D$_3$ simulation the distance between the Cα, atoms of Lys246 and Glu420 was on average 19.0 Å, which is close to 19.1 Å of the x-ray structure. Again, the distance of the VDR-ZK168281 simulation deviated clearly more from the reference distance of the agonistic structure than the distance of the ZK159222 simulation demonstrating the more complete antagonistic properties of the former ligand. Thus, MD simulations could explain the different action of the two antagonists by demonstrating a more drastic displacement of helix 12 through ZK168281 than through ZK159222 [11] [12].

The large majority of 1α,25(OH)$_2$D$_3$ analogs which have been synthesized with the goal of improving the potency and specificity of the physiological effects of vitamin D are simply side chain modifications of the natural hormone. RO43-83582 (Gemini) is an exceptional vitamin D3 analog with two side chains that, despite about 25 % increased volume, acts as a potent agonist and seems to bind the VDR in its agonistic conformation. When Gemini was docked to the ligand-binding pocket of the VDR x-ray structures, it became far from obvious how a ligand with two side chains and considerably increased volume could fit into the binding pocket of the LBD without disturbing the agonistic receptor conformation. To dock the Gemini into the ligand-binding pocket twelve different LBD-Gemini conformations were simulated with the LES method and five copies of the Gemini side chains. The starting conformations for these simulations were generated by rotating the side chains of Gemini in steps of 30°. Simulated annealing was used in conjunction with LES resulting, after cooling to 0 K, an ensamble of 72 side chain placements. These placements formed two narrow clusters predicting two possible positions for the second side chain of Gemini. In both conformations the first side chain keeps the same position than the single side chain of 1α,25(OH)$_2$D$_3$. Gemini's 25-OH also makes the important hydrogen bonds with His305 and His397, like the natural hormone. However, there were two different positions for the second side chain. In one of the positions the extra side chain points in the same direction as the C21-methyl group of 1α,25(OH)$_2$D$_3$, whereas in the other position it is rotated 120 compared to the first position. This structural prediction was challenged by mutating residues closest to the binding positions of the extra side chains into bulky phenylalanines. It could be demonstrated that filling both binding sites of the extra side chain with one bulky phenylalanine is more severe than placing two phenylalanines together into one or the other binding site. In addition, mutations were found to disturb the action of Gemini significantly more than that of 1α,25(OH)$_2$D$_3$. Thus, it was demonstrated that the second side chain can choose between two binding positions within the LBP of the VDR [13] [14] [15]. At the time of these computational and experimental studies there were no crystallographic analysis available of VDR-Gemini complex. Later, a preliminary report on the zebrafish VDR-LBD - two side chain analog x-ray structure confirmed that the second side chain occupies one of the predicted binding pockets with minimal disturbance in the structure of the LBD [16]. An analysis based on neural network algorithms and visualization of the analysis using the Sammon's mapping method of several VDR-ligand x-ray structures and MD simulations showed that the expansion of the ligand-binding pocket to accommodate the second side chain of Gemini, and also the shrinkage of the pocket upon binding of superagonist MC1288, is a combined results of minor movements of more than 30 residues and major movements of a few critical residues [15].

The 26,23-lactone derivative of 1α,25(OH)$_2$D$_3$, TEI-9647, is a partial antagonist of the human VDR, but in rat cells it behaves as a weak agonist. Comparison of the amino acid sequences of the human and rat VDR-LBD and inspection of the VDR crystal structure suggested that there are only two residues, at positions 403 (Cys in human, Ser in rat) and 410 (Cys in human, Asn in rat), which are the most likely candidates for the differing action of TEI-9647. This action could be mimicked in human cells by the double mutagenesis (Cys403Ser and Cys410Asn) to the human VDR. MD simulations of 6 ns of the wild type human VDR and human VDR double mutant (Cys403Ser and Cys410Asn) complexed with 1α,25(OH)$_2$D$_3$ and TEI-9647 were run to study the structural consequences of the mutations. It was seen that TEI-9647 decreases the stability of helix 12 of the wild type VDR compared to 1α,25(OH)$_2$D$_3$. In contrast, the RMSDs of helix 12 of VDR double mutant simulations were similar for both 1α,25(OH)$_2$D$_3$ and

TEI-9647. In addition, the RMSDs of these two simulations were similar to those of VDR-1α,25(OH)$_2$D$_3$. Thus, MD simulations reproduced the experimentally observed behavior of 1α,25(OH)$_2$D$_3$ and TEI-9647 in human and rat cells. Simulation structures showed that Asn410 of rat VDR (Cys in human) formed helix 12 stabilizing interactions between helices 11 and 12. Because of the additional stabilization TEI-9647 acted as a weak agonist of rat VDR but as an partial antagonist of the human VDR [17].

MD simulations have also been used to understand how four selected nonsteroidal VDR agonists bind to the LBD of the VDR. Ligand docking and MD simulations showed that the nonsteroidal ligands take a shape within the LBP that is very similar to that of the natural ligand and that each of the three hydroxyl groups of the ligands formed hydrogen bonds with the residues of the LBP. The natural hormone, 1α,25(OH)$_2$D$_3$, forms six hydrogen bonds with the LDB. Therefore, the six hydrogen bond distances were measured from the MD simulations of the nonsteroidal ligands and 1α,25(OH)$_2$D$_3$. The measurements were compared to the point mutatagenesis analysis in which the same six critical residues were replaced by alanines. Comparison of the calculated distances and experimental data showed good corrrelation between the extend of hydrogen bonds and loss of functional activity. In other words, the more exactly the nonsteroidal ligands place their hydroxyl groups, the more potent VDR agonists they seem to be [18].

The MD simulations of the modulation of the vitamin D receptor activity by various ligands have demonstrated that although the task, the prediction and description of receptor activity is challenging, interesting new insight may be gained if computational experiments are tightly linked with laboratory experiments. However, simulation of ligand-induced receptor activities is far more difficult than simulation of ligand binding affinities, which nowadays may often be considered as a standard simulation task. MD simulations of receptor activities requires often innovative use of computer simulation techniques, application of special analysis methods and, importantly, detailed knowledge about the relationship between receptor structure and its functional properties.

References

1. Bourguet, W., Germain, P., Gronemeyer, H.: Nuclear Receptor Ligand-Binding Domains Three-Dimensional Structures, Molecular Interactions and Pharmacological Implications. Trends Pharmacol. Sci. 21, 381–388 (2000)
2. Feng, W.J., Ribeiro, R.C.J., Wagner, R.L., Nguyen, H., Apriletti, J.A., Fletterick, R.J., Baxter, J.D., Kushner, P.J., West, B.L.: Hormone-Dependent Coactivator Binding to a Hydrophobic Cleft on Nuclear Receptors. Science 280, 1747–1749 (1998)
3. Freedman, L.P.: Increasing the Complexity of Coactivation in Nuclear Receptor Signaling. Cell 97, 5–8 (1999)
4. Rochel, N., Wurtz, J.M., Mitschler, A., Klaholz, B., Moras, D.: The Crystal Structure of the Nuclear Receptor for Vitamin D Bound to its Natural Ligand. Mol. Cell 5, 173–179 (2000)

5. Vanhooke, J.L., Benning, M.M., Bauer, C.B., Pike, J.W., DeLuca, H.F.: Molecular Structure of the Rat Vitamin D Receptor Ligand Binding Domain Complexed with 2-Carbon-Substituted Vitamin D-3 Hormone Analogues and a LXXLL-Containing Coactivator Peptide. Biochemistry 43, 4101–4110 (2004)
6. Simmerling, C., Elber, R.: Hydrophobic Collapse in a Cyclic Hexapeptide – Computer Simulations of CHDLFC and CAAAAC in Water. J. Am. Chem. Soc. 116, 2534–2547 (1994)
7. Case, D.A., Darden, T.A., Cheatham III, T.E., Simmerling C.L., Wang, J., Duke, R.E., Luo, R., Merz, K.M., Wang, B., Pearlman, D.A., Crowley, M., Brozell, S., Tsui, V., Gohlke, H., Mongan, J., Hornak, V., Cui, G., Beroza, P., Schafmeister, C., Caldwell, J.W., Ross, W.D., Kollman, P.A.: AMBER 8. University of California, San Francisco (2004)
8. Wang, J., Wolf, R.M., Caldwell, J.W., Kollman, P.A., Case, D.A.: Development and Testing of a General Amber Force Field. J. Comput. Chem. 25, 1157–1174 (2004)
9. Bayly, C.I., Cieplak, P., Cornell, W.D., Kollman, P.A.: A Well-Behaved Electrostatic Potential Based Method Using Charge Restraints for Deriving Atomic Charges - the RESP Model. J. Phys. Chem. 97, 10269–10280 (1993)
10. Frisch, M.J., Trucks, G.W., Schlegel, H.B., Scuseria, G.E., Robb, M.A., Cheeseman, J.R., Montgomery Jr., J.A., Vreven, T., Kudin, K.N., Burant, J.C., Millam, J.M., Iyengar, S.S., Tomasi, J., Barone, V., Mennucci, B., Cossi, M., Scalmani, G., Rega, N., Petersson, G.A., Nakatsuji, H., Hada, M., Ehara, M., Toyota, K., Fukuda, R., Hasegawa, J., Hratchian, H.P., Cross, J.B., Adamo, C., Jaramillo, J., Gomperts, R., Stratmann, R.E., Yazyev, O., Austin, A.J., Cammi, R., Pomelli, C., Ochterski, J., Ayala, P.Y., Morokuma, K., Voth, G., Salvador, P., Dannenberg, J.J., Zakrzewski, V.G., Dapprich, S., Daniels, A.D., Strain, M.C., Farkas, O., Malick, D.K., Rabuck, A.D., Raghavachari, K., Foresman, J.B., Ortiz, J.V., Cui, Q., Baboul, A.G., Clifford, S., Cioslowski, J., Stefanov, B.B., Liu, G., Liashenko, A., Piskorz, P., Komaromi, I., Martin, R.L., Fox, D.J., Keith, T., Al-Laham, M.A., Peng, C.Y., Nanayakkara, A., Challacombe, M., Gill, P.M.W., Johnson, B., Chen, W., Wong, M.W., Gonzalez, C., Pople, J.A.: Gaussian, Inc., Pittsburgh, PA (2003)
11. Väisänen, S., Peräkylä, M., Kärkkäinen, J.I., Steinmeyer, A., Carlberg, C.: Critical Role of Helix 12 of the Vitamin D3 Receptor for the Partial Agonism of Carboxylic Ester Antagonists. J. Mol. Biol. 315, 229–238 (2002)
12. Lempiäinen, H., Molnár, F., Gonzalez, M.M., Peräkylä, M., Carlberg, C.: Antagonist- and Inverse Agonist-Driven Interactions of the Vitamin D Receptor and the Constitutive Androstane Receptor with Co-Repressor Protein. Mol. Endocrinol. 19, 2258–2272 (2005)
13. Väisänen, S., Peräkylä, M., Kärkkäinen, J.I., Uskokovic, M.R., Carlberg, C.: Structural Evaluation of the Agonistic Action of a Vitamin D Analog with Two Side Chains Binding to the Nuclear Vitamin D Receptor. Mol. Pharmacol. 63, 1230–1237 (2003)
14. Gonzalez, M.M., Samenfeld, P., Peräkylä, M., Carlberg, C.: Corepressor Excess Shifts the Two Side Chain Vitamin D Analogue Gemini from an Agonist to an Inverse Agonist of the Vitamin D Receptor. Mol. Endocrinol. 17, 2028–2038 (2003)
15. Molnár, F., Peräkylä, M., Carlberg, C.: Vitamin D Receptor Agonists Specifically Modulate the Volume of the Ligand-Binding Pocket. J. Biol. Chem. 281, 10516–10526 (2006)

16. Ciesielski, F., Rochel, N., Mitschler, A., Kouzmenko, A., Moras, D.: Structural Investigation of the Ligand Binding Domain of the Zebrafish VDR in Complexes with 1 Alpha,25(OH)(2)D-3 and Gemini: Purification, Crystallization and Preliminary X-ray Diffraction Analysis. J. Steroid Biochem. Mol. Biol. 89(90), 55–59 (2004)
17. Peräkylä, M., Molnár, F., Carlberg, C.: A Structural Basis for the Species-Spesific Antagonism of 26,23-Lactones on Vitamin D Signaling. Chem. Biol. 11, 1147–1156 (2004)
18. Peräkylä, M., Malinen, M., Herzig, K.H., Carlberg, C.: Gene Regulatory Potential of Non-Steroidal Vitamin D Receptor Ligands. Mol. Endocrinol. 19, 2060–2073 (2005)

Sparse Matrix Algebra for Quantum Modeling of Large Systems

Emanuel H. Rubensson[1,2], Elias Rudberg[1,3], and Paweł Sałek[1]

[1] Department of Theoretical Chemistry,
Royal Institute of Technology, SE-10691 Stockholm, Sweden
pawsa@theochem.kth.se
[2] Department of Physics and Chemistry,
University of Southern Denmark, DK-5230 Odense M, Denmark
[3] Department of Chemistry,
University of Warwick, Coventry CV4 7AL, UK

Abstract. Matrices appearing in Hartree–Fock or density functional theory coming from discretization with help of atom–centered local basis sets become sparse when the separation between atoms exceeds some system–dependent threshold value. Efficient implementation of sparse matrix algebra is therefore essential in large–scale quantum calculations. We describe a unique combination of algorithms and data representation that provides high performance and strict error control in blocked sparse matrix algebra. This has applications to matrix–matrix multiplication, the Trace–Correcting Purification algorithm and the entire self–consistent field calculation.

1 Introduction

The properties of matter as we see it are parametrized by the behavior of single atoms. On one hand, atoms that strongly bind to each other will often make a strong material. On the other, few atoms missing from the crystal structure will introduce strain on microscopic scale that can considerably affect macroscopic mechanical or electric properties of samples consisting of many orders of magnitude more atoms. A doping concentration of 1 atom per 10^5 can make the difference between an insulator and a semiconductor [1]. Unfortunately, direct investigation of processes at this level is often difficult and sometimes outright impossible since the act of measurement itself can affect its result [2,3]. Computer modeling of such processes becomes very important for the understanding of such systems. Since creation and breaking of chemical bonds often involves redistribution of the electronic wave function, the system must be described at the quantum level. There are many processes that do not involve considerable electron redistributions and they can be simulated at a simpler, classical level – this is however out of scope of this article. Each electron – and interesting systems consist of thousands of them – is described by its wave function determining the probability that the electron can be found at some point in space. The total electronic wave function must fulfill additional conditions. Since the electrons are indistinguishable fermions, the total wave function must change only sign when two electrons are swapped. This constraint leads to the simple Slater determinant representation ansatz and in turn to the Hartree–Fock (HF) model of the Schrödinger equation if only one determinant is used. The HF model effectively simulates electron movement in an approximate, averaged field of all other electrons and nuclei, and all motion correlation effects are neglected. There have been many attempts to improve this by – for example – including more determinants in the expansion, but they lead to considerably increased computational cost [4]. One of the main advantages of the HF theory is that the practical implementation can be made to scale linearly with the size of the modeled system. An alternative to wave function theory is the so–called Density Functional Theory (DFT). This theory avoids to some extent the complexity associated with wave function theory by using the electronic density $\rho(\boldsymbol{r})$ as the primary variable. This variable choice automatically makes the electrons indistinguishable – only the total density can be determined. The major challenge of DFT is to determine the energy associated with a given density. The initial obstacle related to the kinetic energy term was overcome by the theory of Kohn and Sham (KS). This theory exploits the success of HF and introduces a concept of orbitals so that the kinetic energy functional is evaluated in a manner analogous to HF theory. Currently, many approximations exist for the remaining exchange and correlation terms.

This paper is concerned with an effective way of representing electron density in HF and KS theories. The density $\rho(\boldsymbol{r})$ is usually represented by a matrix expansion

$$\rho(\boldsymbol{r}) = \sum_{p,q=1}^{K} D_{pq} b_p(\boldsymbol{r}) b_q(\boldsymbol{r}) \tag{1}$$

where D_{pq} are elements of the density matrix D and $\{b_p(\boldsymbol{r})\}$ is a set of K basis functions. Common choices for the basis set include plane wave functions, Slater functions $e^{-\alpha r}$, or Gaussian functions $e^{-\alpha r^2}$, the two latter ones multiplied by an angular part and centered at the atoms. The objective of such calculations is to minimize the total HF/KS energy E expressed in terms of the one–electron operator matrix H_1, the two–electron matrix $F_{2\mathrm{el}}$ and – in case of DFT – an exchange–correlation term E_{xc}

$$E = \mathrm{Tr}\,[H_1 D] + \frac{1}{2}\mathrm{Tr}\,[F_{2\mathrm{el}} D] + E_{\mathrm{xc}}\,. \tag{2}$$

The matrix H_1 that includes kinetic energy and nuclear attraction terms depends only on the chosen basis set and atom charges and positions. The matrix $F_{2\mathrm{el}}$ and the scalar E_{xc} both depend on the trial electron density. From now on, the total HF/KS potential matrix $F = H_1 + F_{2\mathrm{el}}$ will be referred to as the Fock matrix. In case of DFT, an additional exchange–correlation term F_{xc} is added to F. HF/KS calculations are done in cycles, each of them involving two time consuming steps: a) evaluation of the Fock matrix for a trial electron density and b) search for the corresponding density matrix. These cycles are performed until self–consistence is reached. The formation of the new density matrix in step b) traditionally uses the so–called aufbau principle: The Fock matrix F is diagonalized to obtain its eigenpairs. The eigenvectors C^{occ} associated with the smallest eigenvalues are combined to obtain the density matrix D:

$$FC^{\text{occ}} = \varepsilon SC^{\text{occ}} \qquad \rightarrow \qquad D = C^{\text{occ}}(C^{\text{occ}})^T \tag{3}$$

where S is the basis set overlap matrix. This operation scales cubically with the problem size and becomes the bottleneck for large systems. A method that takes advantage of the existing sparsity in the F matrix is needed. Several algorithms have been proposed for this purpose [5,6,7,8,9,10,11,12]. All of them compute the solution iteratively by repeated matrix–matrix multiplications. The performance is therefore closely connected to the efficiency of the multiplications. The multiplications can scale linearly if multiplications by zeros which are present in sparse matrices are avoided. Multiplication may result in new small elements appearing but this fill–in can be prevented by filtering out small elements. A systematic filtering algorithm will control the error propagation and contain it under the user–requested threshold.

This paper presents a unique combination of algorithms that provide a robust framework for sparse matrix operations and enable linear scaling in density purification methods. First, a trace–correcting purification (TC2) algorithm is reviewed and some of its computational aspects are described. We then discuss sparsity properties of the involved matrices and review shortly an efficient method for the error control crucial for the TC2 algorithm. Next, we introduce the Hierarchic Matrix Library that was used to implement this algorithm. Finally, we present some benchmarks of this library.

The benchmark systems presented in this article are water droplets with up to 888 molecules, generated by molecular dynamics at 300 K. All benchmarks were performed on an Intel Xeon EM64T 3.4 GHz and 3–21G basis set was used unless stated otherwise.

2 Trace–Correcting Purification

Our work with sparse matrix algebra was motivated by the need for fast and reliable matrix operations in density matrix purification methods. Density matrix purification has been proposed in a multitude of variants [8,9,10,11,12]. The purification algorithms rely on the fact that the Fock matrix and the density matrix share a common set of eigenvectors but have different eigenvalues. One therefore applies a series of eigenvector conserving transformations to the Fock matrix so that the eigenvalues corresponding to occupied eigenvectors converge to 1 and the remaining eigenvalues converge to 0.

The trace–correcting purification algorithm (TC2), developed by Niklasson [9], is not only the simplest one but is also very competitive when it comes to performance measured in number of matrix–matrix multiplications required to converge [9,11]. The TC2 algorithm assumes orthogonal basis set, i.e. the overlap matrix $S = I$. Therefore, the generalized eigenvalue problem in Eq. 3 has to be transformed to standard form. This can for example be achieved by a Cholesky decomposition of the overlap matrix $S = U^T U$ [13]. The purification algorithm is then applied to $F_{ort} = U^{-T} F U^{-1}$ resulting in a density D_{ort} in orthogonal basis which can be transformed back to the original basis by $D = U^{-1} D_{ort} U^{-T}$. This results in an additional cost of four extra matrix–matrix multiplications per self–consistent field cycle plus the cost of one inverse Cholesky decomposition of the overlap matrix. TC2 also requires upper and lower bounds `lmax` and `lmin` of the eigenvalue spectrum which can be obtained using eg. the Gershgorin theorem [14]. The algorithm is as follows:

```
compute P = (lmax*I - F)/(lmax - lmin)
while not converged
    if(trace(P) > N) then
        P := P*P
    else
        P := 2*P - P*P
end while
```

The final result of the purification is contained in `P`. The efficiency and reliability of the purification algorithm depend on the representation of matrices and the algorithms used for the matrix manipulations. In particular, two important operations are repeated in each iteration of the procedure: 1) The symmetric matrix square and 2) Truncation of small matrix elements. Both these operations are described later on in detail. The simplicity of the TC2 algorithm is very appealing but one has to be aware of its drawbacks. One potential deficiency of the TC2 algorithm is that the accumulated error grows exponentially during a number of iterations – see eg. [10] and [15]. It is therefore crucial to diligently control the truncation error. Also, degenerate eigenvalues at the band gap – for which the diagonalization method would randomly pick some to be occupied – cannot be automatically handled and require separate treatment.

3 Sparsity

One feature that distinguishes matrices appearing in HF and KS computations is their rather limited sparsity. Sparsity patterns have previously been investigated for linear alkanes [16]. These systems are very sparse and linear scaling can therefore be achieved for relatively small systems. Three–dimensional dense systems like the water droplets we are using for benchmarks in this article are considerably more difficult to handle. The left panel of Figure 1 shows that the matrices have thousands of nonzeros per row and that this value is still increasing for 610 water molecules. While sparsity is larger in some cases, algorithms have to be able to handle semi–dense cases as well. In all computations presented in this article, the truncation of matrices was done so that, for given matrix A, truncated matrix $\widetilde{A}$, and threshold τ, $\|A - \widetilde{A}\|_F \leq \tau$ as described in section 4. A threshold of $\tau = 10^{-4}$ gives errors in total HF/KS energies of about 10^{-5} Hartree.

Previous work within this field use a blocked compressed sparse row representation [17] to store the non–vanishing submatrices. In this setting, basis

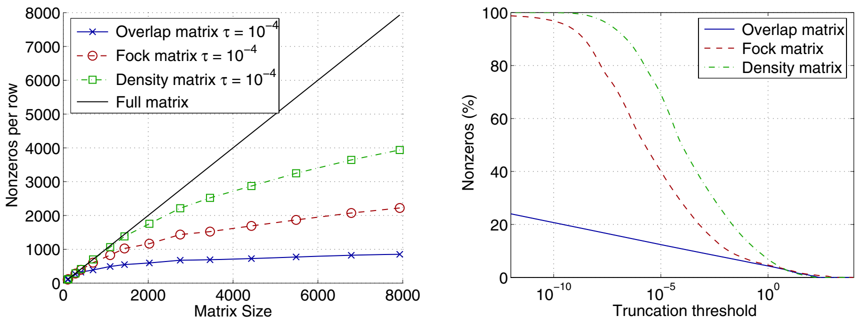


Fig. 1. Left panel: Sparsity in Hartree–Fock computations. The droplet size ranges from 8 to 610 water molecules. Full matrix given as a reference. Right panel: Percentage of nonzeros in the overlap matrix, the Fock matrix, and the density matrix for varying truncation threshold τ. Matrices were computed for a water droplet with 610 water molecules. The matrix size is 7930.

functions are grouped into atom [6] or multi–atom [18] blocks where atoms and associated basis functions are reordered to group matrix elements associated with atoms close in space. Sparsity is then considered at the block level and dense block multiplications are performed using hardware–optimized linear algebra libraries [19,20]. As a consequence, the block sizes are determined by the chosen basis set and the molecular geometry. A problem with this approach is that many linear algebra libraries perform considerably better for selected matrix sizes. Additionally, different block sizes appearing in atom and multi–atom based approaches lead to more substantial heap memory fragmentation which makes it more difficult to optimally utilize available resources. These problems can be avoided by choosing uniform block sizes as described later.

The most important matrices involved in HF and KS computations are the overlap matrix, the Fock matrix, and the density matrix. The right panel of Figure 1 shows an example of the sparsity of these three matrices for different values of the truncation threshold τ. The matrices used in the right panel of Figure 1 correspond to the largest matrix size in the left panel. We note that the change in sparsity with varying truncation threshold is different for the three matrices. This is because the sparsity in the overlap matrix is determined by the basis set only, while for the other matrices, sparsity is also dependent on the physical properties of the system.

4 Systematic Truncation of Small Elements

There exist several ways to maintain sparsity by dropping small matrix elements. One appealing way is to explicitly take advantage of the matrix element magnitude dependence on the distance between the corresponding basis function centers [18,21]. A submatrix is dropped when the shortest distance between the two atom groups is greater than a predefined cutoff radius. It is, however, rarely known which cutoff radius that will correspond to a certain accuracy and this approach will therefore cause severe difficulties with error control. Apart from risking larger errors than expected, one will usually also include submatrices with negligible contribution, reducing in this way the efficiency in subsequent matrix operations [15,22]. Another way to remove unnecessary submatrices is to look at the maximum absolute element in the submatrix. The entire submatrix is dropped if this element is smaller than a preselected threshold. In this way one is able to strictly control the error. However, the error estimate obtained with this approach is far from optimally tight[15].

Based on these observations, we have come to the conclusion that it is better to formulate the truncation in terms of the norm of the entire matrix that one is interested in. For example, if one is interested in a certain accuracy in the total HF/KS energy, one should employ a truncation method that is based on the Frobenius norm of the entire matrix. The idea is simple; while keeping the error in the chosen norm below some requested threshold, as many submatrices as possible should be removed. Also, the truncation should be fast. Our method which we call a Systematic Small–Submatrix Selection Algorithm (SSSA) realizes this idea [15]. The SSSA outline is the following: The norms of all nonzero submatrices are computed and placed into a vector. Subsequently, the norms are sorted in descending order. Finally, small submatrices are removed from the end of this sorted vector as long as the sum of their norms is below the requested threshold. Figure 2 shows that the careful filtering obtained by using the SSSA algorithm together with the Frobenius norm keeps the total HF/KS energy error at a predictable level.

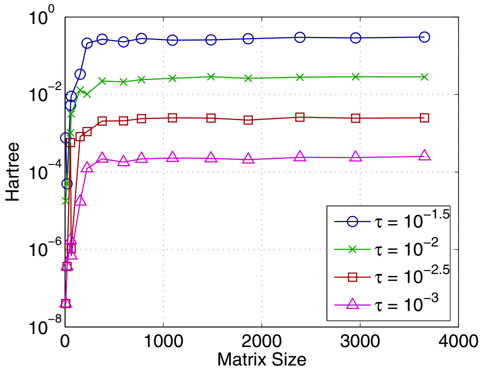


Fig. 2. Error in the final HF energy as a function of selected SSSA threshold and system size. For systems affected by truncation errors, SSSA algorithm keeps their impact at a strictly controlled level. The benchmark systems are water droplets. Basis set: STO–3G. The largest system, with 3654 basis functions consists of 522 water molecules.

5 Hierarchic Matrix Library

An optimal data structure for representation of sparse matrices appearing in HF/KS calculations has to fulfill several conditions. Since the matrices are semi–dense, the representation must not introduce much overhead. The representation must allow for quick evaluation of norms as needed for SSSA and be generally flexible to handle many matrix operations performed on matrices, like matrix multiplications by ordinary and on–the–fly transposed matrices. We propose a hierarchic matrix data structure that treats the matrix in several levels. At the lowest level in the hierarchy, the matrix elements are real numbers just like in the case of an ordinary full matrix. At higher levels, each matrix element is a hierarchic matrix. This makes it possible to consider sparsity at several levels in the hierarchy. If a submatrix is zero at a certain level in the hierarchy, it is unnecessary to reference lower levels. We have realized this idea using generic programming in C++ and called the resulting code library the Hierarchic Matrix Library (HML). A somewhat simplified building block illustrating the idea of HML is the following template:

```
template<typename Telement>
class Matrix {
  Telement* elements;
};
```

Using this template one may define an ordinary full matrix type and a hierarchic three-level matrix type as follows:

```
typedef Matrix<double> MatrixType;
typedef Matrix<Matrix<Matrix<double> > > ThreeLevelMatrixType;
```

At the lowest level we use a specialization that calls the Basic Linear Algebra Subprograms (BLAS). In this way high performance is obtained if the program is linked to a highly tuned BLAS implementation. The matrix–matrix multiplication at higher levels was implemented by the straightforward triple for–loop. Recursive algorithms have previously been used in dense matrix operations to optimally utilize deep memory hierarchies[23]. Also in HML, cache hit rate could possibly be improved by using some kind of cache oblivious algorithm combined with a more careful selection of submatrix block sizes at higher levels. In all benchmarks presented in this article the program was linked to Intel's Math Kernel Library (MKL).

Our reordering of basis functions is similar to the one proposed in Ref. [18] in that it is based on distances in space between atom centers. We have, however, chosen to keep the block size fixed rather than constraining it to match the number of basis functions on a set of atom centers. The reason for this is twofold: 1) At the matrix sizes we are interested in – typically in the range 0 to 200 – BLAS performance is often highly dependent on the matrix size. 2) Use of a uniform block size reduces the probability of memory fragmentation. In Figure 3, timings for two important matrix operations are plotted as functions of block size. The investigated operations are general matrix–matrix multiply (gemm) and symmetric matrix square (sysq). The gemm operation is $C = \alpha\, \mathrm{op}(A)\, \mathrm{op}(B) + \beta C$

and the sysq operation is $S = \alpha T^2 + \beta S$ where α and β are scalars, A, B, and C are general matrices, S and T are symmetric matrices, and $\mathrm{op}(A) = A$ or $\mathrm{op}(A) = A^T$. Note that in this figure the block sizes have been chosen as a multiple of 4 since MKL's gemm operation is optimized for such matrix sizes. Scanning block sizes with step 1 would result in a highly jagged plot, with performance drops up to 40%. This plot may look different for other underlying BLAS libraries but suggests that the multiplication performance is fairly immune to changes of the block size as long as the block size is in the range 24 to 64.

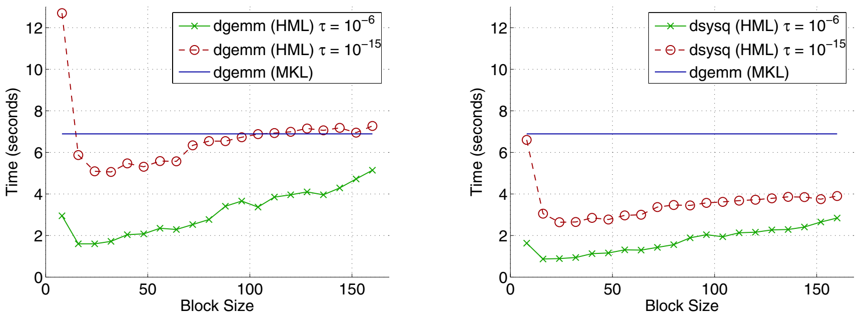


Fig. 3. Sparse matrix–matrix multiplication performance for different choices of block size and truncation threshold with fixed matrix size = 2756, with MKL's full matrix dgemm implementation used as reference. The benchmarks were performed on an overlap matrix from a HF/KS calculation on a water droplet containing 212 water molecules. Left panel: Matrix–matrix multiply (dgemm). Right panel: Symmetric matrix square (dsysq).

6 Performance

An advantage with the hierarchic data structure compared to the blocked compressed sparse row representation is that symmetry in matrices can easily be utilized, often increasing the computational speed and reducing the memory usage by a factor of 2. This can also be seen in Figure 3 where the time for a sysq operation for all block sizes is less than 55% of the time for a gemm operation. In the context of hierarchic matrices this is achieved by recursive computation of symmetric matrix squares (sysq), symmetric rank–k matrix updates (syrk), symmetric matrix–matrix multiplies (symm), and matrix–matrix multiplies (gemm). Figure 4 displays the performance of the TC2 algorithm using HML compared to the traditional diagonalization method. It can be seen that purification can be used for all matrix sizes, not only for large ones, without significant loss of performance. This is achieved thanks to the limited overhead in the HML implementation.

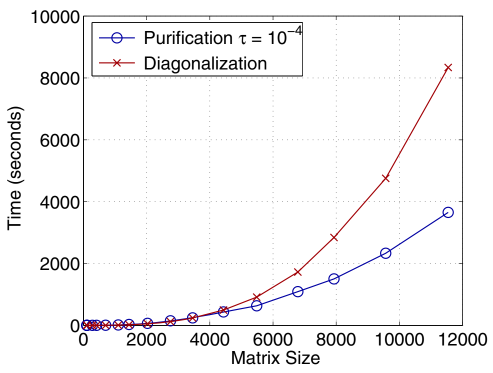


Fig. 4. Benchmark of the trace–correcting purification algorithm implemented with routines from the Hierarchic Matrix Library (HML) (truncation threshold $\tau = 10^{-4}$) compared to diagonalization with the Math Kernel Library's dsygv

7 Summary and Conclusions

We have presented a unique combination of theoretical methods, algorithms and data representation that allows to efficiently store and process matrices appearing in HF and KS theories. Our implementation uses a hierarchic data structure to efficiently store and access elements of sparse matrices, maintaining the truncation error under a requested threshold. This data structure makes it possible to implement many interesting algorithms in an efficient and transparent manner. As an example of such an algorithm, we have chosen the purification algorithm that thanks to our implementation can be used for robust and efficient calculations of electron density matrices in the HF and KS theories. Benchmarks that we present show that purification as implemented by us is competitive to the diagonalization method, becoming superior already for 4000 basis functions, even for dense, three–dimensional systems like water droplets, while maintaining high accuracy.

Acknowledgment. This research has been supported by the Sixth Framework Programme Marie Curie Research Training Network under contract number MRTN–CT–2003–506842.

References

1. Kittel, C.: 8. In: Introduction to Solid State Physics, 7th edn., John Wiley&Sons, Chichester (1996)
2. Schrödinger, E.: Die gegenwartige situation in der quantenmechanik. Naturwissenschaften 23, 807 (1935)
3. Schrödinger, E.: The present situation in quantum mechanics: a translation of Schrödinger's "cat paradox". Proc. Am. Phil. Soc. 124, 323 (1980)

4. See, e.g., Cramer, C.J.: Essentials of computational chemistry, 2nd edn. Wiley, Chichester (2004)
5. Goedecker, S.: Linear scaling electronic structure methods. Rev. Mod. Phys. 71, 1085 (1999)
6. Challacombe, M.: A simplified density matrix minimization for linear scaling self–consistent field theory. J. Chem. Phys. 110, 2332 (1999)
7. Shao, Y., Saravanan, C., Head–Gordon, M., White, C.A.: Curvy steps for density matrix–based energy minimization: Application to large–scale self–consistent–field calculations. J. Chem. Phys. 118, 6144 (2003)
8. Palser, A.H.R., Manolopoulos, D.E.: Canonical purification of the density matrix in electronic structure theory. Phys. Rev. B 58, 12704 (1998)
9. Niklasson, A.M.N.: Expansion algorithm for the density matrix. Phys. Rev. B 66, 155115 (2002)
10. Niklasson, A.M.N., Tymczak, C.J., Challacombe, M.: Trace resetting density matrix purification in O(N) self–consistent–field theory. J. Chem. Phys. 118, 8611 (2003)
11. Mazziotti, D.A.: Towards idempotent reduced density matrices via particle–hole duality: McWeeny's purification and beyond. Phys. Rev. E 68, 66701 (2003)
12. Holas, A.: Transforms for idempotency purification of density matrices in linear–scaling electronic–structure calculations. Chem. Phys. Lett. 340, 552 (2001)
13. Millam, J.M., Scuseria, G.E.: Linear scaling conjugate gradient density matrix search as an alternative to diagonalization for first principles electronic structure calculations. J. Chem. Phys. 106, 5569 (1997)
14. Demmel, J.W.: Applied Numerical Linear Algebra, SIAM, Philadelphia (1997)
15. Rubensson, E.H., Sałek, P.: Systematic sparse matrix error control for linear scaling electronic structure calculations. J. Comp. Chem. 26, 1628–1637 (2005)
16. Maslen, P.E., Ochsenfeld, C., White, C.A., Lee, M.S., Head–Gordon, M.: Locality and Sparsity of Ab Initio One–Particle Density Matrices and Localized Orbitals. J. Phys. Chem. A 102, 2215 (1998)
17. Gustavsson, F.G.: Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. ACM Trans. Math. Softw. 4, 250 (1978)
18. Saravanan, C., Shao, Y., Baer, R., Ross, P.N., Head–Gordon, M.: Sparse matrix multiplications for linear scaling electronic structure calculations in an atom–centered basis set using multiatom blocks. J. Comp. Chem. 24, 618 (2003)
19. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide, release 2.0 SIAM, Philadelphia (1994)
20. Whaley, R.C., Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. Softw. Pract. Exper. 35, 101 (2005)
21. Li, X.-P., Nunes, R.W., Vanderbilt, D.: Density–matrix electronic–structure method with linear system–size scaling. Phys. Rev. B 47, 10891 (1993)
22. Challacombe, M.: A general parallel sparse–blocked matrix multiply for linear scaling SCF theory. Comp. Phys. Comm. 128, 93 (2000)
23. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Rev. 46, 3 (2004)

A Highly Efficient *Ab Initio* Tight-Binding-Like Approximate Density-Functional Quantum Mechanical Method

Yaoquan Tu[1], Lennart Nilsson[2], and Aatto Laaksonen[3]

[1] Department of Theoretical Chemistry,
Royal Institute of Technology,
SE-106 91 Stockholm, Sweden
Tu@theochem.kth.se
[2] Karolinska Institutet,
Department of Biosciences at NOVUM,
Center for Structural Biochemistry,
SE-141 57 Huddinge, Sweden
Lennart.Nilsson@biosci.ki.se
[3] Division of Physical Chemistry,
Arrhenius Laboratory, Stockholm University,
SE-106 91 Stockholm Sweden
aatto@physc.su.se

Abstract. A highly efficient *ab initio* tight-binding-like approximate density-functional quantum mechanical method has recently been developed by us. In this method, the integrals related to the exchange-correlation part are obtained by higher order many-center expansions and all the integrals can be obtained by the interpolation of the look-up tables. The speed of the calculation is also enhanced by using a better way to choose the integrals in the look-up tables. It is shown that the calculated molecular equilibrium geometries and the reaction energies for hydrogenation reactions are very close to those from the usual density functional theory calculations.

1 Introduction

Electronic structure calculations are essential in the study of the properties of molecules and materials. In the past decades, many electronic structure calculation methods have been developed. Of the methods, those developed from the density functional theory (DFT)[1,2,3] have become very popular. Currently, DFT methods can be used to study a system of up to about 100 atoms with considerable accuracy. Despite such great advance, DFT calculations are still very time-demanding. For systems of several hundred atoms or more, such calculations are far too slow to be applied in practice. In recent years, tremendous progresses have been made in developing efficient and reliable approximate electronic structure calculation methods that can be used for a wide variety of purposes, for example, for modeling the forces on atoms in atomistic molecular dynamics computer simulations. One of such advances is the development of *ab initio* tight-binding-like (AITB) electronic structure methods which can be hopefully used as a general tool for electronic structure calculations[4,5,6]. In this paper, we outline our recent work on the development of such an accurate yet highly efficient AITB method.

2 Theoretical Development

AITB methods usually start from the Harris-Foulkes functional[7,8] which is equivalent to expanding the electron-electron interactions in the Kohn-Sham energy functional with respect to a reference density $\tilde{\rho}$, keeping only the zero-th order approximation and the first order correction, and neglecting the second order and higher order corrections. In order to make the calculations simpler and faster, we made a further simplification of the Harris-Foulkes functional. The simplified functional is given as

$$\begin{aligned} E_{HF} = \sum_{i}^{N_{occ}} f_i \varepsilon_i - \frac{1}{2} \int \int \frac{\tilde{\rho}(\mathbf{r}_1)\, \tilde{\rho}(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_1 d\mathbf{r}_2 \\ + E_{xc}\left[\rho^{(0)}\right] - \int \rho^{(0)}(\mathbf{r}) V_{xc}\left(\rho^{(0)}(\mathbf{r})\right) d\mathbf{r} + V_{I-I}, \end{aligned} \tag{1}$$

with

$$V_{xc}(\mathbf{r}) = \frac{\delta E_{xc}[\rho(\mathbf{r})]}{\delta \rho(\mathbf{r})}, \tag{2}$$

where f_i is the occupation number on orbital i. E_{xc} is the exchange-correlation energy functional and V_{xc} is the so-called exchange-correlation potential. ρ is the electron density. V_{I-I} is the ion-ion interaction term. The orbital energy ε_i is found from the following one-electron orbital equation:

$$\left[-\tfrac{1}{2}\nabla_1^2 + V_{ext}(\mathbf{r}_1) + \int \frac{\tilde{\rho}(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_2 + V_{xc}\left(\rho^{(0)}(\mathbf{r}_1)\right)\right] \mid \psi_i(\mathbf{r}_1)\rangle = \varepsilon_i \mid \psi_i(\mathbf{r}_1)\rangle\,, \tag{3}$$

where V_{ext} is the external potential. Compared to the original Harris-Foulkes functional[7,8], our simplification is in the exchange-correlation part. Eq.(1) is equivalent to expanding the Coulomb and exchange-correlation interaction terms in the Kohn-Sham energy functional with respect to $\tilde{\rho}$ and $\rho^{(0)}$, respectively, and neglecting all the second order and higher order corrections. Thus, the error in the total energy caused by such simplification is also only in the second order and would have only a minor effect on the calculation results.

The reference electron densities $\tilde{\rho}$ and $\rho^{(0)}$ we used are super-positions of spherically distributed atomic-like densities having the forms

$$\tilde{\rho}(\mathbf{r}) = \sum_I \tilde{\rho}_I(r_I), \tag{4}$$

and

$$\rho^{(0)}(\mathbf{r}) = \sum_I \rho_I^{(0)}(r_I), \tag{5}$$

respectively, with

$$\tilde{\rho}_I(r_I) = \rho_I^{(0)}(r_I) + \Delta n_I f_I(r_I), \tag{6}$$

where $\rho_I^{(0)}$ is the valence electron density of neutral atom I and $f_I(r_I)$ corresponds to the density of a single electron in the highest occupied atomic orbital. $r_I = |\mathbf{r} - \mathbf{R}_\mathrm{I}|$ is the distance between $\mathbf{r}$ and atomic site $\mathbf{R_I}$. Δn_I can be considered as the net number of electrons that atom I obtains in a molecular system and is determined by

$$\frac{\partial E_{HF}}{\partial \Delta n_I} = 0. \tag{7}$$

Eq.(7) leads to a set of $\{\Delta n_I\}$ required in solving eq.(3). Therefore, in practical calculation, $\{\Delta n_I\}$ and eq.(3) are solved self-consistently.

Under the LCAO-MO approximation, molecular orbitals are linear combinations of atomic orbitals, that is,

$$\psi_i = \sum_\mu C_{\mu i}\phi_\mu, \tag{8}$$

where $\{\phi_\mu\}$ represent atomic orbitals. The coefficients $\{C_{\mu i}\}$ and the orbital energies ϵ_i can be obtained by solving the following equation self-consistently:

$$\mathbf{FC} = \mathbf{SC}\varepsilon, \tag{9}$$

with

$$S_{\mu v} = \langle \phi_\mu | \phi_v \rangle, \tag{10}$$

and

$$\begin{aligned} F_{\mu v} \;=\; & \langle \phi_\mu | -\tfrac{1}{2}\nabla^2 | \phi_v \rangle + \langle \phi_\mu | V_{ion} | \phi_v \rangle \\ + \; & \langle \phi_\mu(\mathbf{r}_1) | \textstyle\int \frac{\tilde{\rho}(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_2 | \phi_v(\mathbf{r}_1) \rangle + \langle \phi_\mu(\mathbf{r}) | V_{XC}\left(\rho^{(0)}(\mathbf{r})\right) | \phi_v(\mathbf{r}) \rangle \\ = \; & \langle \phi_\mu | -\tfrac{1}{2}\nabla^2 | \phi_v \rangle + \langle \phi_\mu | \sum_I V_I^{(PP)} | \phi_v \rangle + \langle \phi_\mu(\mathbf{r}_1) | \sum_I \textstyle\int \frac{\rho_I^{(0)}(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_2 | \phi_v(\mathbf{r}_1) \rangle \\ + \; & \sum_I \Delta n_I \langle \phi_\mu(\mathbf{r}_1) | \textstyle\int \frac{f_I(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_2 | \phi_v(\mathbf{r}_1) \rangle + \langle \phi_\mu(\mathbf{r}) | V_{XC}\left(\rho^{(0)}(\mathbf{r})\right) | \phi_v(\mathbf{r}) \rangle, \end{aligned} \tag{11}$$

where $V_I^{(PP)}$ is the pseudo-potential due to the nucleus and core electrons of atom I.

In this work, $\rho_I^{(0)}$ and $f_I(\mathbf{r})$ are all expanded as linear combinations of 1S type gaussians by least-square fittings. V_{ext} corresponds to the norm-conserving separable dual-space pseudo-potential devised by Goedecker, Teter, and Hutter[9] for DFT calculations. The atomic orbitals $\{\phi_\mu\}$ are also expressed as linear combinations of primitive gaussians. Therefore, it is clear that all the integrals, except for those related to the exchange-correlation potential and energy functional, can be expressed as closed forms and calculated analytically.

For the calculation of integrals related to the exchange-correlation potential or energy functional, the approximation of many-center expansion is adopted. For the off-center integrals, the same expansion formula as that given by Horsfield[5] is used, that is

$$\begin{aligned}\left\langle \phi_{I\alpha} \left| V_{xc}\left(\rho^{(0)}\right) \right| \phi_{J\beta} \right\rangle &= \left\langle \phi_{I\alpha} \left| V_{xc}\left(\rho_I^{(0)} + \rho_J^{(0)}\right) \right| \phi_{J\beta} \right\rangle \\ + \sum_{K(\neq I,J)} \Big\langle \phi_{I\alpha} &\Big| V_{xc}\left(\rho_I^{(0)} + \rho_J^{(0)} + \rho_K^{(0)}\right) - V_{xc}\left(\rho_I^{(0)} + \rho_J^{(0)}\right) \Big| \phi_{J\beta} \Big\rangle. \end{aligned} \tag{12}$$

For the on-site integrals, we developed an improved many-center expansion scheme by including higher order terms in the original expressions of Horsfield. The improved integrals are given as

$$\begin{aligned}\left\langle \phi_{I\alpha} \left| V_{xc}\left(\rho^{(0)}\right) \right| \phi_{I\beta} \right\rangle = &\left\langle \phi_{I\alpha} \left| V_{xc}\left(\rho_I^{(0)}\right) \right| \phi_{I\beta} \right\rangle \\ &+ \sum_{J(\neq I)} \left\langle \phi_{I\alpha} \left| V_{xc}\left(\rho_I^{(0)} + \rho_J^{(0)}\right) - V_{xc}\left(\rho_I^{(0)}\right) \right| \phi_{I\beta} \right\rangle \\ &+ \delta V_{I\alpha,I\beta}, \end{aligned} \tag{13}$$

$$\begin{aligned}\int \rho_I^{(0)}(\mathbf{r})\, \varepsilon_{XC}\left(\rho^{(0)}(\mathbf{r})\right) d\mathbf{r} = &\int \rho_I^{(0)}(\mathbf{r})\, \varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r})\right) d\mathbf{r} \\ &+ \sum_{J(\neq I)} \int \rho_I^{(0)}(\mathbf{r}) \left[\varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r}) + \rho_J^{(0)}(\mathbf{r})\right) - \varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r})\right)\right] d\mathbf{r} \\ &+ \delta E_{XC,I}, \end{aligned} \tag{14}$$

with

$$\begin{aligned}\delta V_{I\alpha,I\beta} \approx \frac{1}{2} \sum_{\substack{J(\neq I) \\ K(\neq I,J)}} &\langle \phi_{I\alpha} | V_{xc}\left(\rho_I^{(0)} + \rho_J^{(0)} + \rho_K^{(0)}\right) + V_{xc}\left(\rho_I^{(0)}\right) \\ &- V_{xc}\left(\rho_I^{(0)} + \rho_J^{(0)}\right) - V_{xc}\left(\rho_I^{(0)} + \rho_K^{(0)}\right) | \phi_{I\beta} \rangle, \end{aligned} \tag{15}$$

and

$$\begin{aligned}\delta E_{XC,I} \approx \frac{1}{2} \sum_{\substack{J(\neq I) \\ K(\neq I,J)}} \int &\rho_I^{(0)}(\mathbf{r}) [\varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r}) + \rho_J^{(0)}(\mathbf{r}) + \rho_K^{(0)}(\mathbf{r})\right) + \varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r})\right) \\ &- \varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r}) + \rho_J^{(0)}(\mathbf{r})\right) - \varepsilon_{XC}\left(\rho_I^{(0)}(\mathbf{r}) + \rho_K^{(0)}(\mathbf{r})\right)] d\mathbf{r}. \end{aligned} \tag{16}$$

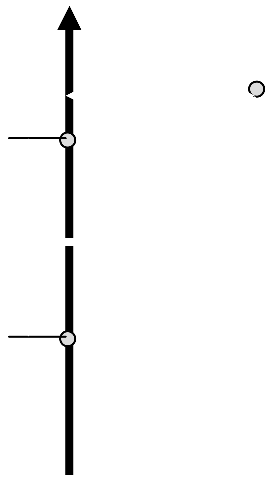


Fig. 1. Geometry used in the look-up tables for three-center integrals. A, B, and C denote atoms.

In the above equations, α and β denote atomic orbitals and I, J, and K denote atoms. ε_{XC} is the exchange-correlation energy density. $\delta V_{I\alpha,I\beta}$ is the correction for the on-site potential integral and $\delta E_{XC,I}$ is the correction for the energy integral.

In practical AITB calculations, the integrals required are often obtained by finding the corresponding integrals in local coordinate systems through the interpolation of the look-up tables and then by transforming the integrals from the local coordinate systems to the molecular coordinate system. Except in those for the one-center integrals, in look-up tables are integrals calculated on a predefined mesh of inter-atomic distances in the local coordinate systems. The geometry used in the tables is shown in Figure 1. The parameters r_i, s_j, and t_k in the mesh are determined by the following equations

$$r_i = r_{\max} - (r_{\max} - r_{\min}) \frac{\ln(N_r + 1 - i)}{\ln(N_r)}, \tag{17}$$

$$s_j = s_{\max} - (s_{\max} - s_{\min}) \frac{\ln(N_s + 1 - j)}{\ln(N_s)}, \tag{18}$$

and

$$t_k = t_{\max} - (t_{\max} - t_{\min}) \frac{\ln(N_t + 1 - k)}{\ln(N_t)} \tag{19}$$

Usually, when $r_i = r_{\max}$, or $s_j = s_{\max}$, or $t_k = t_{\max}$, most of the integrals approach to some values that can be neglected. The exceptions are those of the form $\langle \phi_\mu (\mathbf{r}_1)| \int \frac{f_J(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_2 \, |\phi_v (\mathbf{r}_1)\rangle$. For these integrals, a similar treatment to that outlined in Ref([10]) is used.

3 Results

Based on the method outlined above , we have written a Fortran AITB calculation program using BLYP [11,12] Generalized Gradient Approximation (GGA) for the exchange-correlation functional. In our current implementation, the minimal basis set using only the valence atomic orbitals as basis functions is adopted. Some preliminary results are listed in Tables 1 to 3.

Table 1. Equilibrium geometries (bond lengths in Å and angles in degrees) for selected molecules

Molecule	Symmetry	Parameter	DFT [a]	This Work	Exp.
H_2	$D_{\infty h}$	R(HH)	0.823	0.823	0.742
CH_4	T_d	R(CH)	1.164	1.156	1.092
C_2H_2	$D_{\infty h}$	R(CC)	1.297	1.255	1.203
		R(CH)	1.154	1.139	1.061
C_2H_4	D_{2h}	R(CC)	1.409	1.422	1.339
		R(CH)	1.169	1.158	1.085
		∠(HCH)	115.9	116.4	117.8
C_2H_6	D_{3d}	R(CC)	1.583	1.572	1.531
		R(CH)	1.171	1.162	1.096
		∠(HCH)	107.1	107.1	107.8
NH_3	C_{3v}	R(NH)	1.104	1.118	1.012
		∠(HNH)	105.7	100.7	106.7
HCN	$C_{\infty v}$	R(CN)	1.258	1.232	1.153
		R(CH)	1.119	1.146	1.065
CH_2NH	C_s	R(CN)	1.362	1.338	1.273
		R(CH_{syn})	1.190	1.170	1.103
		R(CH_{anti})	1.181	1.167	1.081
		R(NH)	1.141	1.133	1.023
		∠($H_{syn}CN$)	128.2	127.4	123.4
		∠($H_{anti}CN$)	118.3	118.9	119.7
		∠(HNC)	109.0	109.0	110.5
CH_3NH_2	C_s	R(CN)	1.498	1.533	1.471
		R(CH_{tr})	1.194	1.171	1.099
		R(CH_g)	1.177	1.165	1.099
		R(NH_a)	1.100	1.118	1.010
		∠(NCH_{tr})	117.7	116.7	113.9
		∠($H_gCH'_g$)	106.4	107.0	108.0
		∠(HNH)	108.7	101.1	107.1

[a] Frozen core

Table 2. Equilibrium geometries (bond lengths in Å and angles in degrees) for selected molecules

Molecule	Symmetry	Parameter	DFT[a]	This Work	Exp.
N_2	$D_{\infty h}$	R(NN)	1.257	1.195	1.098
N_2H_2	C_{2h}	R(NN)	1.342	1.310	1.252
		R(NH)	1.190	1.153	1.028
		$\angle$(HNN)	113.5	113.4	106.9
N_2H_4	C_2	R(NN)	1.563	1.535	1.449
		R(NH_{int})	1.132	1.127	1.021
		R(NH_{ext})	1.132	1.127	1.021
		$\angle$(NNH_{int})	103.4	103.3	106.0
		$\angle$(NNH_{ext})	103.4	103.3	112.0
		Θ($H_{int}NNH_{ext}$)	75.2	77.0	91.0
H_2O	C_{2v}	R(OH)	1.102	1.093	0.958
		$\angle$(HOH)	99.3	99.6	104.5
H_2O_2	C_2	R(OO)	1.617	1.512	1.452
		R(OH)	1.134	1.109	0.965
		$\angle$(OOH)	97.2	102.2	100.0
		Θ(HOOH)	139.0	116.7	119.1
CO	$C_{\infty h}$	R(CO)	1.320	1.254	1.128
H_2CO	C_{2v}	R(CO)	1.349	1.309	1.208
		R(CH)	1.199	1.188	1.116
		$\angle$(HCH)	114.9	114.5	116.5
CH_3OH	C_s	R(CO)	1.530	1.508	1.421
		R(CH_{tr})	1.176	1.166	1.094
		R(CH_g)	1.184	1.173	1.094
		R(OH)	1.108	1.096	0.963
		$\angle$(OCH_{tr})	107.1	106.8	107.2
		$\angle$($H_gCH'_g$)	107.5	107.4	108.5
		$\angle$(COH)	104.4	105.5	108.0

[a] Frozen core

These results cover the equilibrium geometries for some molecules (See Tables 1 and 2), and the reaction energies for hydrogenation reactions (See Table 3). For comparison, we also listed the results from the usual frozen-core DFT calculations and experiment. In the frozen-core DFT calculations, the same (BLYP) exchange-correlation functional and basis set are used with the effects of the nuclei and core electrons modeled by a compact effective-core potential of Stevens et.al.[13]. The frozen-core DFT calculations are carried out by GAUSSIAN98[14]. The experimental values are from Reference([15,16]).

From the tables, we can see that the AITB results are quite close to those from the usual frozen-core DFT calculations. Both types of theoretical calculations give similar trends. For the molecular equilibrium geometries, both the AITB and DFT calculations give, for most molecules, longer bond lengths and smaller

Table 3. Selected reaction energies (in kcal/mol) for hydrogenation reactions

Reaction	DFT[a]	This work	HF/STO-3G	Exp.
$C_2H_6 + H_2 \rightarrow 2CH_4$	18	17	19	19
$C_2H_4 + 2H_2 \rightarrow 2CH_4$	72	66	91	57
$C_2H_2 + 3H_2 \rightarrow 2CH_4$	137	119	154	105
$CH_3OH + H_2 \rightarrow CH_4 + H_2O$	17	18	16	30
$H_2O_2 + H_2 \rightarrow 2H_2O$	32	28	31	86
$H_2CO + 2H_2 \rightarrow CH_4 + H_2O$	47	45	65	59
$CO + 3H_2 \rightarrow CH_4 + H_2O$	74	57	72	63
$N_2 + 3H_2 \rightarrow 2NH_3$	25	19	36	37
$N_2H_2 + 2H_2 \rightarrow 2NH_3$	52	48	75	68
$N_2H_4 + H_2 \rightarrow 2NH_3$	31	26	28	48
$HCN + 3H_2 \rightarrow CH_4 + NH_3$	81	73	97	76
$CH_3NH_2 + H_2 \rightarrow CH_4 + NH_3$	17	21	20	26
$CH_2NH + 2H_2 \rightarrow CH_4 + NH_3$	57	55	78	64

[a] Frozen core

bond angles. For the reaction energies for hydrogenation reactions, the AITB results are in general closer to those from the frozen-core DFT than to those from the Hartree-Fock approach with STO-3G as basis set.

4 Conclusion

In this paper, we have presented a reliable, yet highly efficient tight-binding-like approximate density-functional electronic structure calculation method. It is shown that the calculated molecular equilibrium geometries and the reaction energies for hydrogenation reactions are very close to those from the accurate results based on the density functional theory (DFT).

Acknowledgements

This work has been supported by the Swedish Science Council (VR).

References

1. Hohenberg, P., Kohn, W.: Inhomogeneous electron gas. Phys. Rev. 136, B864–871 (1964)
2. Kohn, W., Sham, L.J.: Self-consistent equations including exchange and correlation effects. Phys. Rev. 140, A1133–A1138 (1965)
3. Parr, R.G., Yang, W.: Density-functional theory of atoms and molecules. Oxford University Press, Oxford (1989)
4. Sankey, O.F., Niklewski, D.J.: Ab initio multicenter tight-binding model for molecular dynamics simulations and other applications in covalent systems. Phys. Rev. B40, 3979–3995 (1989)

5. Horsfield, A.P.: Efficient ab initio tight binding. Phys. Rev. B 56, 6594–6602 (1997)
6. Lewis, J.P., Glaesemann, K.R., Voth, G.A., Fritsch, J., Demkov, A.A., Ortega, J., Sankey, O.F.: Further developments in the local-orbital density-functional-theory tight-binding method. Phys. Rev. B 64, 195103.1–195103.10 (2001)
7. Harris, J.: Simplified method for calculating the energy of weakly interacting fragments. Phys. Rev. B 31, 1770–1779 (1985)
8. Foulkes, W.M.C., Haydock, R.: Tight-binding models and density-functional theory. Phys. Rev. B 39, 12520–12536 (1989)
9. Goedecker, S., Teter, M., Hutter, J.: Separable dual-space Gaussian pseudopotentials. Phys. Rev. B 54, 1703–1710 (1996)
10. Demkov, A.A., Ortega, J., Sankey, O.F., Grumbach, M.P.: Electronic structure approach for complex silicas. Phys. Rev. B 52, 1618–1630 (1995)
11. Becke, A.D.: Density-functional exchange-energy approximation with correct asymptotic-behavior. Phys. Rev. A 38, 3098–3100 (1988)
12. Lee, C., Yang, W., Parr, R.G.: Development of the Colle-Salvetti correlation energy formula into a functional of the electron density. Phys. Rev. B 37, 785 (1988)
13. Stevens, W.J., Basch, H., Krauss, M.: Compact effective potentials and efficient shared-exponent basis sets for the first- and second row atoms. J. Chem. Phys. 81, 6026–6033 (1984)
14. Frisch, M.J., Trucks, G.W., Schlegel, H.B., Scuseria, G.E., Robb, M.A., Cheeseman, J.R., Zakrzewski, V.G., Montgomery Jr., J.A., Stratmann, R.E., Burant, J.C., Dapprich, S., Millam, J.M., Daniels, A.D., Kudin, K.N., Strain, M.C., Farkas, O., Tomasi, J., Barone, V., Cossi, M., Cammi, R., Mennucci, B., Pomelli, C., Adamo, C., Clifford, S., Ochterski, J., Petersson, G.A., Ayala, P.Y., Cui, Q., Morokuma, K., Malick, D.K., Rabuck, A.D., Raghavachari, K., Foresman, J.B., Cioslowski, J., Ortiz, J.V., Baboul, A.G., Stefanov, B.B., Liu, G., Liashenko, A., Piskorz, P., Komaromi, I., Gomperts, R., Martin, R.L., Fox, D.J., Keith, T., Al-Laham, M.A., Peng, C.Y., Nanayakkara, A., Challacombe, M., Gill, P.M.W., Johnson, B., Chen, W., Wong, M.W., Andres, J.L., Gonzalez, C., Head-Gordon, M., Replogle, E.S., Pople, J.A.: Gaussian98. Technical Report Rev. A9, Gaussian Inc., Pittsburgh PA (1998)
15. Lide, D.R. (ed.): Handbook of Chemistry and Physics, 85th edn. CRC, Boca Raton (2004)
16. Hehre, W.J., Radom, L., Schleyer, P.V.R., Pople, J.A.: Ab initio molecular orbital theory. John Wiley and sons, New York (1986)

Protein Folding Properties from Molecular Dynamics Simulations

David van der Spoel, Alexandra Patriksson, and M. Marvin Seibert

Department of Cell and Molecular Biology, University of Uppsala,
Box 596, SE-75124 Uppsala, Sweden
spoel@xray.bmc.uu.se
http://folding.bmc.uu.se

Abstract. Protein folding simulations have contributed significantly to our understanding of the problem, since it is difficult to study individual molecules during the folding process. We have recently performed folding simulations of Chignolin, a decapeptide (Seibert *et al.*, J. Mol. Biol. **354** (2006) p. 173) and introduced a new algorithm for deriving kinetics information as well as thermodynamics from the trajectories (Van der Spoel & Seibert, Phys. Rev. Lett. **96** (2006), p. 238102). Here we investigate the algorithm further and show that the folding reaction for Chignolin is a two-state folding reaction, in accord with experimental data.

1 Introduction

The replica exchange molecular dynamics (REMD) method [1,2,3] allows for coupling multiple simulations run in parallel. In this manner simulations can travel between different temperatures (Fig. 1). By using a Metropolis criterion based on the potential energy, the conformations that have low energy are effectively sorted down, such that one obtains a larger population of low-energy structures at low temperature than at high temperature. In the limit of very long simulations one will obtain an equilibrium population at each temperature, and hence temperature dependent properties can in principle be determined [4,5,6]. REMD has been used frequently for protein folding simulations [7,4,8,6,9,10,11], and this is also the context in which we apply it. There are other computational approaches to the protein folding problem which we will not discuss here since they have recently been reviewed extensively by Snow *et al.* [12].

Although REMD in itself is a very useful tool, it was hitherto not possible to use the time-dimension in the simulations. That is, it was problematic to interpret the causality in the simulation quantitatively due to the fluctuating temperatures (Fig. 2). We have recently introduced a new algorithm that does exactly that: follow the simulations, including the intrinsic temperature jump, and deduct kinetic and thermodynamic information from these trajectories [13]. In this paper we focus on the inner workings of the algorithm, and demonstrate the power of the method on new simulation data. Finally, we discuss applications outside the field of protein folding simulations.

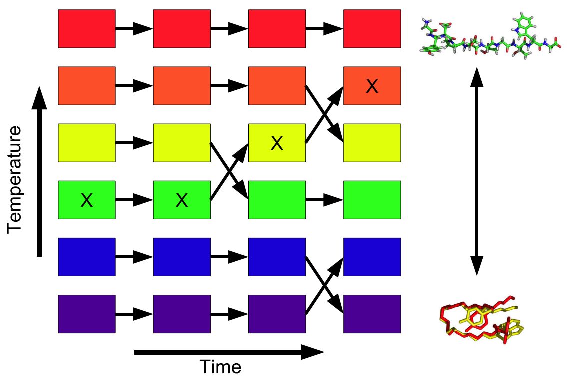


Fig. 1. Schematic representation of the replica-exchange algorithm [1]. The cross denotes the trajectory of a simulation in time and temperature. The peptide structures exemplify the "sorting" with native structures at low T and extended (or otherwise non-native) structures at high T.

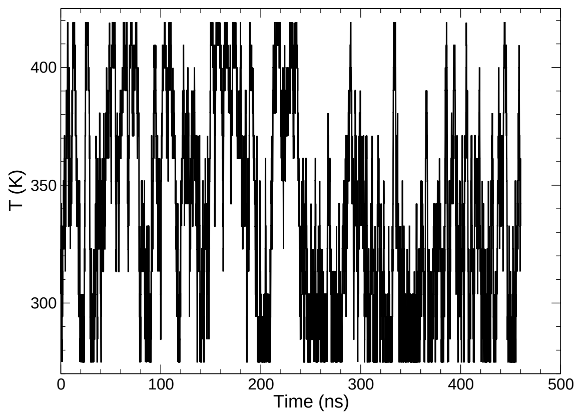


Fig. 2. Trajectory in temperature of one of the replicas in a 16-replica simulation of Chignolin [11]

2 Methods

2.1 Simulation Details

A linear peptide with sequence GYDPETGTWG, corresponding to the Chignolin peptide [14] was constructed using the PyMOL program and solvated with 888 water molecules and 2 Na+ ions. A rhombic dodecahedron box with periodic image distance of 3.4 nm was used. The energy of the system was minimized with the steepest descent algorithm, and a 200 ps simulation was performed during which the positions of the protein atoms were restrained. This system was subsequently used as a starting conformation for the replica exchange [1,2,3] MD simulations. The OPLS force field [15] was used with TIP4P water [16]. 16 replicas were used with temperatures of 275, 282, 289, 296, 305, 313, 322, 331, 340, 350, 360, 371, 382, 394, 406, 419 K respectively. Simulations were 510 ns. To maintain the temperatures at the chosen levels and the pressure at 1 bar, Berendsen weak coupling [17] was used, with coupling constants of 0.1 ps for the temperature and 1 ps for the pressure. A twin range cut-off of 0.9/1.4 nm for Van der Waals interactions was applied and the smooth particle mesh Ewald algorithm [18] was used for Coulomb interactions, with a switching distance of 0.9 nm. Neighbor lists were utilized and updated every fifth integration step. Constraints were used for bond lengths using the LINCS algorithm for the protein [19] and SETTLE [20] for the water. In addition two classical single temperature trajectories of 1.8 resp. 2.0 μs were generated at 300 K, and single trajectories of 510 ns at 277 K and 367 K respectively. All simulations and analysis were performed using the GROMACS software [21,22,23].

2.2 Kinetics Analysis

For Chignolin we have $M = 20$ simulation trajectories of the same protein under identical conditions except for temperature. Now, let $F_m(t)$ be a binary indicator of folding, i.e. $F_m(t) = 1$ corresponds to trajectory m being folded at time t, and $F_m(t) = 0$ corresponds to a not-folded state, and assume we can determine $F_m(t)$ from each conformation in our trajectories. An example $F_m(t)$ might be the criterion that the RMSD to the native state is less than a certain cut-off. From any state $F_m(t)$ we can determine the change using the reactive flux correlation:

$$\frac{dF_m(t)}{dt} = k_f U_m(t) - k_u F_m(t) \tag{1}$$

where k_f is the rate constant for folding ($u \rightarrow f$), k_u is the rate constant for unfolding ($f \rightarrow u$) and U is either the unfolded state [1-F] or another state close to the folded state. We adopt the notion that the rate constants are related to activation energies E_A and prefactors A:

$$k_u = A_u \mathrm{e}^{-\beta E_A^u}, \quad k_f = A_f \mathrm{e}^{-\beta E_A^f} \tag{2}$$

where $\beta = 1/k_B T$ with k_B Boltzmann's constant and T the temperature. If we rewrite Eqn. 1 with the explicit time dependence of the temperature, we have for each trajectory m:

$$\frac{dF_m(t)}{dt} = A_f \mathrm{e}^{-\beta_m(t) E_A^f} U_m(t) - A_u \mathrm{e}^{-\beta_m(t) E_A^u} F_m(t) \tag{3}$$

Since β changes stochastically with time, this equation can not be integrated analytically. We can however integrate it numerically, average over the trajectories, and define the integral $\Phi(t)$ as:

$$\Phi(t) = \frac{1}{M} \sum_{m=1}^{M} \int_0^t \frac{dF_m(\tau)}{dt} \mathrm{d}\tau \tag{4}$$

with which we can define a fitting parameter χ^2:

$$\chi^2 = \frac{1}{N}\sum_{t=1}^{N}\left[\Phi(t) - F(t)\right]^2 \tag{5}$$

where $F(t) = \langle F_m(t)\rangle$ i.e. the average fraction folded protein in all the simulations. χ^2 can be minimized numerically [24] with respect to E_A^f, E_A^u, A_f and A_u. In this manner the parameters that describe the kinetics are optimized to the change in the fraction of folded proteins with time, which implies that this analysis can be applied to any set of MD simulations.

The algorithm can be extended by defining an intermediate state $I(t)$. The root mean square deviation of the simulated structure with respect to the experimental structure can be used as a criterion, since there are two minima, one at 0.18 nm and one at 0.295 nm (see Fig. 1 in ref. [13]). In that case we define two coupled reactions:

$$\frac{dI_m(t)}{dt} = k_{ui}U_m(t) - k_{iu}I_m(t) \tag{6}$$

$$\frac{dF_m(t)}{dt} = k_{if}I_m(t) - k_{fi}F_m(t) \tag{7}$$

The four rate constants are now defined by four energies and four prefactors. If we equate the integral of Eqn. 6 to $\mathcal{I}(t)$ we can define a new fitting parameter θ^2:

$$\theta^2 = \frac{1}{N}\sum_{t=1}^{N}\left[\Phi(t) - F(t)\right]^2 + \left[\mathcal{I}(t) - I(t)\right]^2 \tag{8}$$

and these eight parameters can be optimized in order to minimize θ@2 by integrating both equations simultaneously.

3 Results and Discussion

We have recently reported simulations of the complete folding of the polypeptide Chignolin [14] from the extended state to the native conformation [11] along with detailed analysis of the energy landscape in three dimensions. Using the novel method described above and in ref. [13] it was possible to obtain kinetics information from 20 heterogeneous trajectories. This new algorithm, which was inspired by the hydrogen-bond kinetics analysis due to Luzar and Chandler [25,26,27], allows for the analysis of relaxation mechanisms in general, but here we restrict ourselves to protein folding.

In the algorithm, the reactive-flux correlation term is used to determine the change in a property. The present method introduces two new techniques, first the inclusion of the explicit time dependence in the analysis, and second the optimization technique in which the integral of the desired property is fitted to, rather than the property itself. In this manner we ensure that the kinetics of folding as it occurs in the simulation is reproduced at all intermediate times rather than just the equilibrium constant. For two-state reactions, like folding of Chignolin [14], we can extrapolate the results (activation energies) and constants to different temperatures and hence obtain faithful melting curves. However, there is considerable discussion in the literature as to whether two-state folding reactions really exist, or whether experimental methods are not sensitive enough to discriminate different rates of folding and/or discern intermediates that may be on the folding pathway even in case of the folding of very simple proteins like Chignolin. In order to test the proposition that folding follows a two-state reaction in our simulations of Chignolin, as can be inferred from the NMR experiments of Honda *et al.* [14], we re-analyzed the simulation results with an explicit intermediate state. The development of the intermediate state $I(t)$ and the folded state $F(t)$, as well as the function fitted to them $\mathcal{I}(t)$ and $\Phi(t)$ are plotted in Fig. 3. The energy terms and prefactors obtained in this analysis are given in Table 1 .

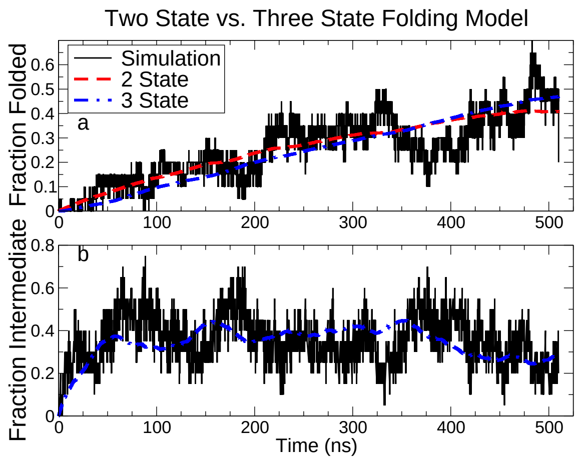


Fig. 3. Build up of **a** fraction folded and **b** fraction intermediate (average over the 20 Chignolin simulations). The fit to Eqn. 5 is given in **a**, and the fits to Eqn. 8 in **a** and **b** for folded and intermediate fractions respectively.

Both $\Phi(t)$ and $\mathcal{I}(t)$ are slightly out of phase with the direct simulation (Fig. 3), due to the very nature of the rate equations (Eqns. 1 and 6). When minimizing Eqn. 5 we obtain a final $\chi^2 = 0.1$ where with Eqn. 8 we obtain a final $\theta^2 = 0.38$. Since we are summing twice the number of deviations with Eqn. 8, we expect that $\theta^2 \leq 2\chi^2$, but in fact θ^2 is larger, indicating that the fit is not as good when using intermediates. Indeed, the activation energies between intermediate and folded states E_{fi} and E_{if} are both 2 kJ/mol (Table 1), i.e. less than $k_B T$ at room temperature, and hence the barrier between intermediate and folded is negligible in this model. Furthermore, we find that the melting temperature in the model with intermediates is 360 K vs. 340 K in the two-state model and 312 K experimentally [14]. The fact that the match with experiment is less good

Table 1. Values for activation energies (E_A) and prefactors A obtained by minimizing either χ^2 (Eqn. 5, top) or θ^2 (Eqn. 8, bottom). The time constants corresponding to room temperature kinetics are given as well (298.15 K).

	A (1/ps)	E (kJ/mole)	τ (μs)
U→F	9.3e-5(0.1)	11.2(1)	1.0(0.3)
F→U	0.094(0.01)	30.7(1)	2.6(0.4)
I→U	9.3e-5(0.2)	9.7(2)	0.5(0.1)
U→I	1.0e-4(0.1)	18.8(2)	19(2)
I→F	3.6e-5(0.3)	2.1(0.5)	0.06(0.02)
F→I	3.7e-5(0.3)	2.0(0.5)	0.06(0.02)

is no proof that the three-state model is not suitable to explain the simulation results. However, this observation, in combination with the comparison between θ^2 and χ^2 and the unreasonably low barriers between intermediate and folded state, indicates strongly that the kinetics in the simulation is best described by a two-state model. On the other hand, we have also shown how the kinetics model can be applied to processes governed by a three-state model.

The method could in principle be applied to many other fields of Science where processes can be followed in time directly using simulation. A particularly powerful feature of the method is that it operates on the time derivative of an observable that is, the change in a variable. This implies that a combination of starting conformations be used, like, for instance, the REMD simulations of Garcia and Onuchic [5] who used a mixture of unfolded and folded conformations. The fundamental observation that we should study the changes in "foldedness", rather than starting many simulation from the unfolded state, as is done by Pande *et al.* using distributed computing techniques [28,29,30], makes that it suffices to have a limited amount of heterogeneous simulation data. For protein folding, it is obvious that the trajectories have to be sufficiently long, in relation to the folding time. For Chignolin, we had a total of 10μs, an order of magnitude larger than the estimated folding time (1.0 μs [13]). This may serve as a useful guideline for determining how much trajectory is needed. In summary, we think that the proposed kinetics method will bring computational studies of protein folding with full atomic detail, and in explicit solvent within reach of a departmental computer cluster.

References

1. Hukushima, K., Nemoto, K.: J. Phys. Soc. Jpn. 65, 1604–1608 (1996)
2. Okabe, T., Kawata, M., Okamoto, Y., Mikami, M.: Chem. Phys. Lett. 335, 435–439 (2001)
3. Okamoto, Y.: J. Molec. Graph. Model, 22, 425–439 (2004)
4. Zhou, R., Berne, B.J., Germain, R.: Proc. Natl. Acad. Sci. U.S.A. 98, 14931–14936 (2001)
5. Garcia, A.E., Onuchic, J.N.: Proc. Natl. Acad. Sci. U.S.A. 100, 13898–13903 (2003)
6. Zhou, R.: Proc. Natl. Acad. Sci. U.S.A. 100, 13280–13285 (2003)

7. Gront, D., Kolinski, A., Skolnick, J.: J. Chem. Phys. 115, 1569–1574 (2001)
8. Pitera, J.W., Swope, W.: Proc. Natl. Acad. Sci. U.S.A. 100, 7587–7592 (2003)
9. Rao, F., Caflisch, A.: J. Chem. Phys. 119(7), 4035–4042 (2003)
10. Paschek, D., Garcia, A.E.: Phys. Rev. Lett. 93, 238105 (2004)
11. Seibert, M., Patriksson, A., Hess, B., van der Spoel, D.: J. Mol. Biol. 354, 173–183 (2005)
12. Snow, C.D., Sorin, E.J., Rhee, Y.M., Pande, V.S.: Annu. Rev. Biophys. Biomol. Struct. 34, 43–69 (2005)
13. van der Spoel, D., Seibert, M.M.: Phys. Rev. Lett. 96, 238102 (2006)
14. Honda, S., Yamasaki, K., Sawada, Y., Morii, H.: Structure Fold. Des. 12, 1507–1518 (2004)
15. Jorgensen, W.L.: Encyclopedia of Computational Chemistry, vol. 3, pp. 1986–1989. Wiley, New York (1998)
16. Jorgensen, W.L., Chandrasekhar, J., Madura, J.D., Impey, R.W., Klein, M.L.: J. Chem. Phys. 79, 926–935 (1983)
17. Berendsen, H.J.C., Postma, J.P.M., DiNola, A., Haak, J.R.: J. Chem. Phys. 81, 3684–3690 (1984)
18. Essmann, U., Perera, L., Berkowitz, M.L., Darden, T., Lee, H., Pedersen, L.G.: J. Chem. Phys. 103, 8577–8592 (1995)
19. Hess, B., Bekker, H., Berendsen, H.J.C., Fraaije, J.G.E.M.: J. Comp. Chem. 18, 1463–1472 (1997)
20. Miyamoto, S., Kollman, P.A.: J. Comp. Chem. 13, 952–962 (1992)
21. Berendsen, H.J.C., van der Spoel, D., van Drunen, R.: Comp. Phys. Comm. 91, 43–56 (1995)
22. Lindahl, E., Hess, B.A., van der Spoel, D.: J. Mol. Mod. 7, 306–317 (2001)
23. van der Spoel, D., Lindahl, E., Hess, B., Groenhof, G., Mark, A.E., Berendsen, H.J.C.: J. Comp. Chem. 26, 1701–1718 (2005)
24. Nelder, J.A., Mead, R.: Computer J. 7, 308–315 (1965)
25. Luzar, A., Chandler, D.: Nature 379, 55–57 (1996)
26. Luzar, A.: J. Chem. Phys. 113, 10663–10675 (2000)
27. van der Spoel, D., van Maaren, P.J., Larsson, P., Timneanu, N.: J. Phys. Chem. B 110, 4393–4398 (2006)
28. Shirts, M.R., Pande, V.S.: Phys. Rev. Lett. 86, 4983–4986 (2001)
29. Snow, C.D., Nguyen, H., Pande, V.S., Gruebele, M.: Nature 420, 102–106 (2002)
30. Rhee, Y.M., Sorin, E.J., Jayachandran, G., Lindahl, E., Pande, V.S.: Proc. Natl. Acad. Sci. U.S.A. 101, 6456–6461 (2004)

Recent Advances in Dense Linear Algebra: Minisymposium Abstract

Daniel Kressner[1] and Julien Langou[2]

[1] University of Zagreb, Croatia and Umeå University, Sweden
[2] The University of Tennessee, Knoxville, USA

These last past years have seen a tremendous amount of new results in computational dense linear algebra. New algorithms have been developed to increase the speed of convergence of eigensolvers, to improve the final accuracy of solvers, to improve the parallel efficiency of applications, and to harness even better the capability of our computing platforms. Of particular interest for this minisymposium are new algorithms that outperform the algorithms used in Sca/LAPACK's current routines or match expected new functionality of the library.

Parallel Variants of the Multishift QZ Algorithm with Advanced Deflation Techniques*

Björn Adlerborn, Bo Kågström, and Daniel Kressner

Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden
{adler,bokg,kressner}@cs.umu.se

Abstract. The QZ algorithm reduces a regular matrix pair to generalized Schur form, which can be used to address the generalized eigenvalue problem. This paper summarizes recent work on improving the performance of the QZ algorithm on serial machines and work in progress on a novel parallel implementation. In both cases, the QZ iterations are based on chasing chains of tiny bulges. This allows to formulate the majority of the computation in terms of matrix-matrix multiplications, resulting in natural parallelism and better performance on modern computing systems with memory hierarchies. In addition, advanced deflation strategies are used, specifically the so called aggressive early deflation, leading to a considerable convergence acceleration and consequently to a reduction of floating point operations and computing time.

1 Introduction

The QZ algorithm is the most widely used method for computing all n eigenvalues λ of a regular matrix pair (A, B) with $A, B \in \mathbb{R}^{n\times n}$, which satisfy

$$\det(A - \lambda B) = 0.$$

The QZ algorithm was developed by Moler and Stewart in [19] and relies on computing orthogonal matrices Q and Z such that $(S, T) = (Q^T AZ, Q^T BZ)$ is in real generalized Schur form, i.e., S is quasi-upper triangular with 1×1 and 2×2 blocks on the diagonal, while T is upper triangular. This equivalence transformation preserves the eigenvalues of (A, B), which then can be easily extracted from the block diagonals of S and T. The LAPACK [2] implementation of the QZ algorithm is mainly based on [19], with some improvements proposed in [13,22,24]. It consists of the following subroutines:

`DGGBAL` performs an optional preliminary balancing step [23] aiming to improve the accuracy of subsequently computed eigenvalues.

* This work was supported by the DFG Emmy Noether fellowship KR 2950/1-1 and by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

DGGHRD reduces a general matrix pair (A, B) to Hessenberg-triangular form, i.e., it computes in a finite number of steps orthogonal matrices Q_1 and Z_1 such that $H = Q_1^T A Z_1$ is upper Hessenberg while $T = Q_1^T B Z_1$ is upper triangular.

DHGEQZ reduces (H, T) further, by applying single- and double-shift QZ iterations combined with deflations, to real generalized Schur form.

DTGSEN and DTGEVC post-process the output of DHGEQZ to compute selected eigenvectors and deflating subspaces [12] of (A, B).

Additionally, there are a number of support and driver routines for solving generalized eigenvalue problems. The focus of the improvements described in the following are the QZ iterations and deflations implemented in DHGEQZ, see also [1,11]. Improvements to DGGBAL, DGGHRD and DTGSEN, which are also considered for inclusion in the next LAPACK release, can be found in [8,15,16,18,21].

The rest of this paper is organized as follows. Section 2 is concerned with techniques intented to decrease the execution time of the QZ algorithm on serial machines: chains of tightly coupled tiny bulges [5,11,17] and aggressive early deflation [6,11]. In Section 3, it is shown how these techniques can be employed to derive a parallel variant of the QZ algorithm. Numerical experiments, reported in Section 4, illustrate the performance of Fortran implementations based on the ideas presented in this paper.

2 Improvements to the Serial QZ Algorithm

Let us consider a regular matrix pair (H, T) in Hessenberg-triangular form. By a preliminary deflation of all infinite eigenvalues [19], we may assume without loss of generality that T is nonsingular. In the following, we only describe the two improvements of the QZ algorithm proposed in [11] that make our implementation perform so well in comparison with existing implementations. However, it should be emphasized that for a careful re-implementation of LAPACK's QZ algorithm one also needs to reinvestigate several somewhat detailed but nevertheless important issues, such as the use of ad hoc shifts to avoid convergence failures and the optimal use of the pipelined QZ iterations described in [8] for addressing medium-sized subproblems.

2.1 Multishift QZ Iterations

The traditional implicit double-shift QZ iteration [19] starts with computing the vector

$$v = (HT^{-1} - \sigma_1 I)(HT^{-1} - \sigma_2 I), \tag{1}$$

where I denotes the $n \times n$ identity matrix and $\sigma_1, \sigma_2 \in \mathbb{C}$ are suitably chosen shifts. Next an orthogonal matrix Q (e.g., a Householder reflector [9]) is computed such that $Q^T v$ is mapped to a scalar multiple of the first unit vector e_1. This transformation is applied from the left to H and T:

$$H \leftarrow Q^T H, \qquad T \leftarrow Q^T T.$$

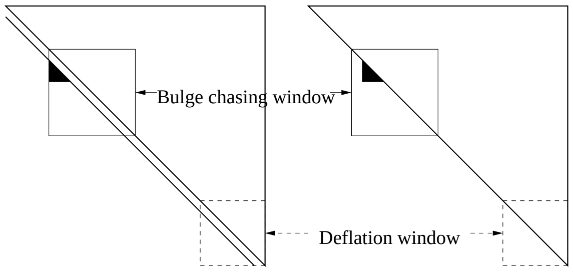


Fig. 1. Illustration of a multishift QZ step with aggressive early deflation

The Hessenberg-triangular structure of the updated matrix pair is destroyed in the first three rows and the rest of the implicit QZ iteration consists of reducing it back to Hessenberg-triangular form without touching the first row of H or T. Due to the special structure of (H, T), this process requires $O(n^2)$ flops and can be seen as chasing a pair of 3×3 bulges along the subdiagonals of H and T down to the bottom right corner, see [19,24]. If the shifts are chosen to be the eigenvalues of the 2×2 lower bottom submatrix pair of (H, T) then typically the $(n-1, n-2)$ subdiagonal entry of H converges to zero. Such a subdiagonal entry is explicitly set to zero if it satisfies

$$|h_{j+1,j}| \leq \mathbf{u}(|h_{jj}| + |h_{j+1,j+1}|), \tag{2}$$

where $\mathbf{u}$ denotes the unit roundoff. This criterion not only ensures numerical backward stability but may also yield high relative accuracy in the eigenvalues for graded matrix pairs, see [11] for more details. Afterwards, the QZ iterations are continued on the deflated lower-dimensional generalized eigenvalue problems.

The described QZ iteration performs $O(n^2)$ flops while accessing $O(n^2)$ memory. This poor computation/communication ratio limits the effectiveness of the QZ algorithm for larger matrices. An idea which increases the ratio without affecting the convergence of QZ iterations, has been extrapolated in [11] from existing techniques for the QR algorithm, see, e.g., [5]. Instead of only one bulge pair corresponding to one double shift, a tightly coupled chain of bulge pairs corresponding to several double shifts is introduced and simultaneously chased. This allows the use of level 3 BLAS without a significant increase of flops in the overall QZ algorithm.

The implementation of such a multishift QZ iteration is illustrated in Figure 1. In the beginning of a chasing step, the bulge chain resides in the top left corner of the bulge chasing window. Each bulge is subsequently chased downwards until the complete bulge chain arrives at the bottom right corner of the window. During this process only the window parts of H and T are updated. All resulting orthogonal transformations are accumulated and applied in terms of matrix-matrix multiplications (GEMM) to the rest of the matrix pair.

2.2 Aggressive Early Deflation

Another ingredient, which may drastically lower the number of iterations needed by the QZ algorithm, is aggressive early deflation introduced in [6] and extended in [1,11]. Additionally to the classic deflation criterion (2), the following strategy is implemented. First, H and T are partitioned

$$(H,T) = \left(\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{bmatrix}, \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ 0 & T_{22} & T_{23} \\ 0 & 0 & T_{33} \end{bmatrix} \right),$$

such that $H_{32} \in \mathbb{R}^{m\times 1}$ and $H_{33}, T_{33} \in \mathbb{R}^{m\times m}$ (typical choices of m are between 40 and 240). Then the matrix pair (H_{33}, T_{33}), which corresponds to the deflation window illustrated in Figure 1, is reduced to real generalized Schur form. By applying the corresponding left orthogonal transformation to H_{32}, a spike is introduced in H. If the trailing $d \leq m$ spike elements can be safely set to zero (see [6] for various criteria) then the bottom right $d \times d$ submatrix pair can be deflated. Otherwise, the Schur form of (H_{33}, T_{33}) is reordered to move other, untested eigenvalues to its bottom right corner, see [11] for more implementation details.

3 A Parallel QZ Algorithm

Parallel distributed memory (DM) algorithms and implementations for reducing a matrix pair to Hessenberg-triangular form have been presented in [1,7]. In this contribution, we consider the remaining part of the QZ algorithm, QZ iterations. Our new parallel variants are based on the LAPACK [2] implementation of the QZ algorithm as well as the blocked serial variants described in [8,11]. In order to gain better performance and scalability, we employ the following extensions:

– Use of several small bulges introduced and chased down the diagonals of H and T in a blocked manner.
– Accumulation of orthogonal transformations in order to apply them in a blocked manner, leading to delayed updates.
– Use of the aggressive early deflation technique described in Section 2.2.

Given a Hessenberg-triangular matrix pair (H, T), a parallel QZ iteration is divided into three major operations, which are implemented in separate routines: (1) deflation check; (2) bulge introduction; (3) bulge chasing. In the following, we give a brief description of these operations.

3.1 Parallel QZ Step – Deflation Check

The deflation check routine searches and tests for deflated eigenvalues at the bottom right corners of H and T using the aggressive early deflation technique, see [6,11]. This routine also returns the shifts, calculated eigenvalues from a bottom right submatrix pair of H and T within the current deflation window, see Figure 1, needed to start up a new bulge introduction and chase iteration.

The deflation check is performed by all processors and therefore communication is required before all processors have the required data. The output of the deflation check is, beside the deflated window, two orthogonal matrices which contain the accumulated equivalence transformations. If deflation was successful, these transformations are applied to the right and above the deflation window. The update is performed in parallel using GEMM operations. Some nearest neighbor communication is required to be able to perform the multiplications. The subsequent QZ iterations are restricted to the deflated submatrix pair, also called the *active submatrix pair*.

3.2 Parallel QZ Step – Bulge Introduction

The introduction of bulges takes place at the top left corner of the active submatrix pair. The number of bulges that can be created depends on how many shifts were returned from the last deflation check. At most #shifts/2 bulges are created using information from the top left corner of (H, T) to compute the first column of the shift polynomial.

After a bulge has been introduced it has to be chased down some steps in order to give room for a new bulge. If $N < n$ bugles are to be introduced the first bulge is chased $N \cdot (n_H + 1)$ positions, where n_H is the size of the Householder transformation, the second $(N - 1) \cdot (n_H + 1)$ positions and so forth ($n_H = 3$ in general). The chasing consists of applying Householder transformations to (H, T) from the left and right. We limit the update of (H, T) to a window of size $NB \times NB$. The orthogonal updates are also applied to two matrices U and V, initially set to the identity matrix. This way we can introduce all bulges and after that update the remaining parts of (H, T) by using GEMM operations with U and V to complete the calculation of the corresponding equivalence transformation $(Q^T HZ, Q^T TZ)$.

The window of size $NB \times NB$ is held by all processors. Communication is therefore required to send the data to all the processors. The subsequent update is performed in parallel where every processor updates its corresponding portion of (H, T). The communication in the update part is limited to nearest neighbor processors, interchanging matrix border elements (row and column data) to be able to perform the GEMM operations independently in parallel.

3.3 Parallel QZ Step – Bulge Chasing

The introduced bulges are repeatedly moved together within a bulge chasing window, see Figure 1, of size $NB \times NB$. The movement begins by moving the first introduced bulge until the bottom of the bulge chasing window. This is then repeated by moving each bulge the same number of steps. As in the introduction phase the bulge movement arises from applying pairs of (left and right) Householder transformations. Moreover, the update of (H, T) is again limited to the window of size $NB \times NB$ and the update of the remaining parts is performed afterwards in parallel as described in Section 3.2.

4 Numerical Experiments

In the following, we briefly report on numerical experiments performed on a Linux cluster Sarek at HPC2N, consisting of 190 HP DL145 nodes, with dual AMD Opteron 248 (2.2GHz) and 8 GB memory per node, connected in a Myrinet 2000 high speed interconnect. The AMD Opteron 248 has a 64 kB instruction and 64 kB data L1 Cache (2-way associative) and a 1024 kB unified L2 Cache (16-way associative).

4.1 Serial Results

First, we tested a random 2000×2000 matrix pair reduced to Hessenberg-triangular form. LAPACK's `DHGEQZ` requires 270 seconds, while the multishift QZ algorithm described in Section 2.1 with 60 simultaneous shifts requires 180 seconds (on machines with smaller L2 cache this reduction was observed to be even more significant). Applying aggressive early deflation with deflation window size 200 reduced the execution time further, to remarkable 28 seconds. This significant reduction of execution time carries over to other, practically more relevant examples. Table 1 contains a list of benchmark examples from [11] with order $n \geq 1900$.

Table 1. Selected set of matrix pairs from the Matrix Market collection [3], the Oberwolfach model reduction benchmark collection [14], and corner singularities computations [20]

#	Name	n	Brief description	Source
1	**HEAT**	1900	Heat conduction through a beam	[14]
2	**BCSST26**	1922	Seismic analysis, nuclear power station	[3]
3	**BEAM**	1992	Linear beam with damping	[14]
4	**BCSST13**	2003	Fluid flow	[3]
5	**CIRC90**	2166	Circular cone, opening angle 90 degrees	[20]
6	**FICH1227**	2454	Fichera corner, Dirichlet boundary conditions	[20]
7	**BCSST23**	3134	Part of a 3D globally triangularized building	[3]
8	**MHD3200**	3200	Alfven spectra in magnetohydrodynamics	[3]
9	**BCSST24**	3562	Calgary Olympic Saddledome arena	[3]
10	**BCSST21**	3600	Clamped square plate	[3]

We compared the performance of three Fortran implementations of the QZ algorithm: `DHGEQZ` (LAPACK), `KDHGEQZ` (pipelined QZ iterations [8]), `MULTIQZ` (multishift QZ iterations + aggressive early deflation). From Figure 2, which shows the execution time ratios `DHGEQZ`/`KDHGEQZ` and `DHGEQZ`/`MULTIQZ`, it can be observed that `MULTIQZ` is 2–12 times faster than LAPACK. It should be noted that aggressive early deflation can also be used to decrease the execution times of `DHGEQZ` and `KDHGEQZ`, see [11] for more details.

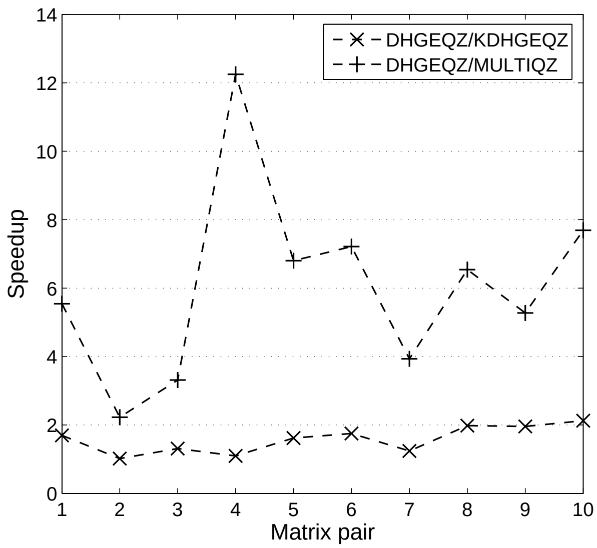


Fig. 2. Performance comparison on Sarek for of three serial implementations of the QZ algorithm. The test matrices used are from Table 1.

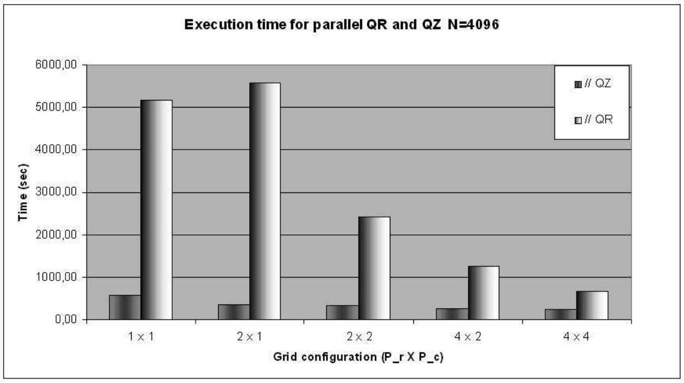


Fig. 3. Execution times for ScaLAPACK's QR for a 4096 × 4096 random Hessenberg matrix and parallel QZ for a 4096 × 4096 random Hessenberg-triangular matrix pair

4.2 Parallel Results

A preliminary Fortran implementation of the described parallel variant of the QZ algorithm has been developed based on BLACS and ScaLAPACK [4].

Figure 3 gives a brief but representative impression of the obtained timings. It can be seen from Figure 3 that the new parallel variant of the QZ algorithm is significantly faster than the ScaLAPACK implementation of the QR algorithm [10].

(Note that the traditional QZ algorithm takes roughly *twice* the computational effort of the QR.) This effect can be contributed to the use of blocking techniques and aggressive early deflation. In Table 2, we display performance results on Sarek for 1 up to 16 processors of the three stages in reducing a regular matrix pair to generalized Schur form. Stage 1 reduces a regular (A, B) to block upper Hessenberg-triangular form (H_r, T) using mainly level 3 (matrix-matrix) operations [1,8]. In Stage 2, all but one of the r subdiagonals of H_r are set to zero using Givens rotations, leading to (H, T) in Hessenberg-triangular form [1,8]. Finally, Stage 3 computes the generalized Schur form (S, T) by applying our parallel QZ implementation. The overall speedup for the complete reduction to generalized Schur form is over $\sqrt{p}$, where p is the number of processors used. From the performance results presented it can be seen that the scalability of the parallel QZ (Stage 3) is improvable; this issue will be subject to further investigation.

Table 2. Sample performance results on Sarek for 1 up to 16 processors of the three stages in reducing a regular matrix pair to generalized Schur form

Configuration				Stage 1	Stage 2	Stage 3	Total	
N	P_r	P_c	NB	*Time*	*Time*	*Time*	*Time*	S_P
1024	1	1	160	5.8	19.5	13.3	38.5	1.0
1024	2	1	160	3.3	12.4	11.5	27.2	1.4
1024	2	2	160	2.8	7.8	11.8	22.3	1.7
2048	1	1	160	55.1	188.3	67.8	311.1	1.0
2048	2	2	160	21.2	64.6	62.5	148.3	2.1
2048	4	2	160	13.8	37.1	49.9	100.8	3.1
2048	4	4	160	14.0	28.6	43.5	86.1	3.6
2048	8	2	160	11.9	28.5	43.3	83.7	3.7
4096	1	1	160	551.3	1735.4	494.1	2780.7	1.0
4096	2	2	160	166.9	475.5	365.9	1008.3	2.8
4096	4	2	160	99.3	294.7	303.2	679.2	4.1
4096	4	4	160	97.6	245.6	248.2	591.4	*4.7*
4096	8	2	160	78.9	204.9	231.4	515.2	*5.4*

References

1. Adlerborn, B., Dackland, K., Kågström, B.: Parallel and blocked algorithms for reduction of a regular matrix pair to Hessenberg-triangular and generalized Schur forms. In: Fagerholm, J., Haataja, J., Järvinen, J., Lyly, M., Råback, P., Savolainen, V. (eds.) PARA 2002. LNCS, vol. 2367, pp. 319–328. Springer, Heidelberg (2002)
2. Anderson, E., Bai, Z., Bischof, C.H., Blackford, S., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.C.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
3. Bai, Z., Day, D., Demmel, J.W., Dongarra, J.J.: A test matrix collection for non-Hermitian eigenvalue problems (release 1.0). Technical Report CS-97-355, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, (March 1997), Also available online from `http://math.nist.gov/MatrixMarket`

4. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J.W., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
5. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. I. Maintaining well-focused shifts and level 3 performance. SIAM J. Matrix Anal. Appl. 23(4), 929–947 (2002)
6. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. II. Aggressive early deflation. SIAM J. Matrix Anal. Appl. 23(4), 948–973 (2002)
7. Dackland, K., Kågström, B.: A ScaLAPACK-style algorithm for reducing a regular matrix pair to block Hessenberg-triangular form. In: Kagström, B., Elmroth, E., Waśniewski, J., Dongarra, J.J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 95–103. Springer, Heidelberg (1998)
8. Dackland, K., Kågström, B.: Blocked algorithms and software for reduction of a regular matrix pair to generalized Schur form. ACM Trans. Math. Software 25(4), 425–454 (1999)
9. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
10. Henry, G., Watkins, D.S., Dongarra, J.J.: A parallel implementation of the nonsymmetric QR algorithm for distributed memory architectures. SIAM J. Sci. Comput. 24(1), 284–311 (2002)
11. Kågström, B., Kressner, D.: Multishift variants of the QZ algorithm with aggressive early deflation. Report UMINF-05.11, Department of Computing Science, Umeå University, Umeå, Sweden, 2005, SIAM J. Matrix Anal. Appl. (to appear)
12. Kågström, B., Poromaa, P.: Computing eigenspaces with specified eigenvalues of a regular matrix pair (A, B) and condition estimation: theory, algorithms and software. Numer. Algorithms 12(3-4), 369–407 (1996)
13. Kaufman, L.: Some thoughts on the QZ algorithm for solving the generalized eigenvalue problem. ACM Trans. Math. Software 3(1), 65–75 (1977)
14. Korvink, J.G., Evgenii, B.R.: Oberwolfach benchmark collection. In: Benner, P., Mehrmann, V., Sorensen, D.C. (eds.) Dimension Reduction of Large-Scale Systems. Lecture Notes in Computational Science and Engineering, vol. 45, pp. 311–316. Springer, Heidelberg (2005)
15. Kressner, D.: Numerical Methods and Software for General and Structured Eigenvalue Problems. PhD thesis, TU Berlin, Institut für Mathematik, Berlin, Germany (2004)
16. Kressner, D.: Block algorithms for reordering standard and generalized Schur forms, 2005, ACM Trans. Math. Software (to appear)
17. Lang, B.: Effiziente Orthogonaltransformationen bei der Eigen- und Singulärwertzerlegung. Habilitationsschrift (1997)
18. Lemonnier, D., Van Dooren, P.: Balancing regular matrix pencils, 2004 SIAM J. Matrix Anal. Appl. (to appear)
19. Moler, C.B., Stewart, G.W.: An algorithm for generalized matrix eigenvalue problems. SIAM J. Numer. Anal. 10, 241–256 (1973)
20. Pester, C.: CoCoS – computation of corner singularities. Preprint SFB393/05-03, Technische Universität Chemnitz (2005), See `http://www.tu-chemnitz.de/sfb393/`
21. Quintana-Ortí, G., Quintana-Ortí, E.S.: An efficient algorithm for computing the Hessenberg-triangular form. Technical report ICC 2006-05-01, Universidad Jaime I, Castellón, Spain (2006)

22. Ward, R.C.: The combination shift QZ algorithm. SIAM J. Numer. Anal. 12(6), 835–853 (1975)
23. Ward, R.C.: Balancing the generalized eigenvalue problem. SIAM J. Sci. Statist. Comput. 2(2), 141–152 (1981)
24. Watkins, D.S., Elsner, L.: Theory of decomposition and bulge-chasing algorithms for the generalized eigenvalue problem. SIAM J. Matrix Anal. Appl. 15, 943–967 (1994)

Parallel Algorithms and Condition Estimators for Standard and Generalized Triangular Sylvester-Type Matrix Equations

Robert Granat and Bo Kågström

Department of Computing Science and HPC2N, Umeå University,
SE-901 87 Umeå, Sweden
{granat,bokg}@cs.umu.se

Abstract. We discuss parallel algorithms for solving eight common standard and generalized triangular Sylvester-type matrix equation. Our parallel algorithms are based on explicit blocking, 2D block-cyclic data distribution of the matrices and wavefront-like traversal of the right hand side matrices while solving small-sized matrix equations at different nodes and updating the rest of the right hand side using level 3 operations. We apply the triangular solvers in condition estimation, developing parallel sep^{-1}-estimators. Some experimental results are presented.

Keywords: Sylvester-type matrix equations, triangular matrix equations, Bartels–Stewart's method, explicit blocking, condition estimation, level 3 BLAS, ScaLAPACK, 2D block-cyclic data distribution.

1 Introduction

We consider the following standard Sylvester-type matrix equations: the *continuous-time Sylvester equation* (SYCT)

$$AX - XB = C, \tag{1}$$

the discrete-time Sylvester equation (SYDT)

$$AXB^T - X = C, \tag{2}$$

the continuous-time Lyapunov equation (LYCT)

$$AX + XA^T = C, \tag{3}$$

and *the discrete-time Lyapunov equation* (LYDT)

$$AXA^T - X = C, \tag{4}$$

where A of size $m \times m$, B of size $n \times n$ and C of size $m \times n$ or $m \times m$ are general matrices with real entries.

We also consider the following generalized Sylvester-type matrix equations: *the generalized coupled Sylvester equation* (GCSY)

$$(AX - YB,\ DX - YE) = (C,\ F), \tag{5}$$

where A and D of size $m \times m$, B and E of size $n \times n$ and C and F of size $m \times n$ are general matrices with real entries, *the generalized Sylvester equation* (GSYL)

$$AXB^T - CXD^T = E, \tag{6}$$

where A and C of size $m \times m$, B and D of size $n \times n$ and E of size $m \times n$ are general matrices with real entries, *the continuous-time generalized Lyapunov equation* (GLYCT)

$$AXE^T + EXA^T = C, \tag{7}$$

where A, E and C of size $m \times m$ are general matrices with real entries, and *the discrete-time generalized Lyapunov equation* (GLYDT)

$$AXA^T - EXE^T = C, \tag{8}$$

where A, E and C of size $m \times m$ are general matrices with real entries.

Solvability conditions for equations (1)-(8) can be formulated in terms of the standard or generalized eigenvalues of the involved matrices or regular matrix pairs, see, e.g., [15,16]. For (G)LYCT/(G)LYDT a symmetric right hand side C implies a symmetric solution X.

SYCT, LYCT and GCSY are called *one-sided* because the undetermined X (or X and Y) is multiplied by another matrix from one side only. SYDT, LYDT, GSYL, GLYCT and GLYDT are called *two-sided* [15,16].

In this contribution, we assume that all left hand side coefficient matrices or matrix pairs are (quasi-)triangular, i.e, in real or generalized Schur form (see, e.g., [5]). If this is not the case, we utilize Bartels–Stewart's method [2] for reducing the matrix equation to triangular form by orthogonal transformations:

1. Reduce the known left hand side matrices (or matrix pairs) of equations (1)-(8) to real (generalized) Schur form.
2. Update the right hand side matrix (or matrix pair) with respect to the Schur decompositions.
3. Solve the resulting triangular matrix equation.
4. Transform the computed solution matrix (or matrix pair) back to the original coordinate system.

Based on our previous work with parallel solvers for triangular matrix equations (see, e.g., [8,9,10]), we now focus on developing a complete set of parallel algorithms and library routines for solving the reduced matrix equations corresponding to equations (1)-(8). Besides being an integral part in solving general large scale matrix equations, these solvers are applied to condition estimation of the matrix equations themselves as well as for different subspace problems with applications in control theory. In this contribution, we present some of this work in progress. The final goal is a complete library of ScaLAPACK-style routines called SCASY for solving general as well as reduced (triangular) standard and generalized Sylvester-type equations (1)-(8).

2 Blocked Methods for Solving Reduced Matrix Equations

We focus on step 3 above for solving the reduced matrix equations. Assuming $m = n$, this is an $O(n^3)$ operation. We apply explicit blocking (see below) to reformulate each matrix equation problem into as much level 3 BLAS operations as possible. In the following, the (i, j)th block of a partitioned matrix, say X, is denoted X_{ij}.

Let mb and nb be block sizes used in an explicit block partitioning of the matrices A and B in SYCT, respectively. In turn, this imposes a similar block partitioning of C and X (which overwrites C). Then $D_a = \lceil m/mb \rceil$ and $D_b = \lceil n/nb \rceil$ are the number of diagonal blocks in A and B, respectively. Now, SYCT can be rewritten in block-partitioned form as

$$A_{ii}X_{ij} - X_{ij}B_{jj} = C_{ij} - (\sum_{k=i+1}^{D_a} A_{ik}X_{kj} - \sum_{k=1}^{j-1} X_{ik}B_{kj}), \tag{9}$$

for $i = 1, 2, \ldots, D_a$ and $j = 1, 2, \ldots, D_b$ [10].

For LYCT we use a similar approach: Partition A and C by rows and columns using a single block size mb and rewrite LYCT as

$$A_{ii}X_{ij} + X_{ij}A_{jj}^T = C_{ij} - (\sum_{k=i+1}^{D_a} A_{ik}X_{kj} + \sum_{k=j+1}^{D_a} X_{ik}A_{jk}^T), \tag{10}$$

reformulating our single LYCT problem into smaller SYCT ($i \neq j$) and LYCT ($i = j$) problems and level 3 updates in the right hand side C. Moreover, if C is symmetric, we rewrite (10) for the main diagonal blocks C_{ii} as

$$A_{ii}X_{ii} + X_{ii}A_{ii}^T = C_{ii} - (\sum_{k=i+1}^{D_a} A_{ik}X_{ik}^T + X_{ik}A_{ik}^T), \tag{11}$$

which defines a sum of SYR2K-operations, which are as fast as regular GEMM-operations when implemented as a GEMM-based level 3 BLAS [18,19].

We block the two-sided standard equations SYDT and LYDT similarly:

$$A_{ii}X_{ij}B_{jj}^T - X_{ij} = C_{ij} - \sum_{(k,l)=(i,j)}^{(D_a,D_b)} A_{ik}X_{kl}B_{jl}^T,\ (k,l) \neq (i,j) \tag{12}$$

and

$$A_{ii}X_{ij}A_{jj}^T - X_{ij} = C_{ij} - \sum_{(k,l)=(i,j)}^{(D_a,D_a)} A_{ik}X_{kl}A_{jl}^T,\ (k,l) \neq (i,j). \tag{13}$$

Notice that the blocking of LYDT decomposes the problem into several smaller SYDT ($i \neq j$) and LYDT ($i = j$) equations.

The same method of explicit blocking is applied to the generalized matrix equations (5)-(8).

All linear matrix equations considered can be rewritten as an equivalent large linear system of equations $Zx = y$, where Z is the Kronecker product representation of the corresponding Sylvester-type operator. For example, SYCT (1) corresponds to $Z_{\mathrm{SYCT}} = I_n \otimes A - B^T \otimes I_m$, $x = \mathrm{vec}(X)$, $y = \mathrm{vec}(C)$ (see also Section 4). These formulations are only efficient to use explicitly when solving small-sized problems in kernel solvers, see, e.g., LAPACK's `DLASY2` and `DTGSY2` for solving SYCT and GCSY and the kernels of the RECSY library [15,16,17].

3 Parallel Algorithms for Triangular Matrix Equations

The parallel algorithms for SYCT presented in [24,10,8,9] were based on the following basic ideas: Utilize explicit blocking and 2D block cyclic distribution of the matrices over a rectangular $P_r \times P_c$ process grid, following the ScaLAPACK conventions [3], and compute the solution by a wavefront-like traversal of the block diagonals of the right hand side matrix where several solutions of diagonal subsystems are computed in parallel, broadcasted along the corresponding block rows and columns, and used in level 3 updates of the rest of the right hand side. This is illustrated for SYCT in Figure 1. Notice that the solution X overwrites the right hand side C blockwise.

The algorithms are adapted to the symmetric LYCT by wavefront-like traversal of the *anti-diagonals* of the right hand side matrix while solving for the lower (or upper) triangular part of the solution. The situation is described in Figure 2. However, our solvers must be able to solve non-symmetric LYCT problems as well since symmetry cannot be assumed in condition estimation algorithms (see Section 4).

For two-sided standard matrix equations SYDT/LYDT the main difference from the SYCT/LYCT cases are the need for an extra buffer for storing *intermediate sums of matrix products* caused by a more complex data dependency (see equations (12)-(13)) which will, assuming $m = n$, cause any trivially blocked solver to use $O(n^4)$ flops. We illustrate with the following explicitly blocked SYDT system:

$$\begin{cases} A_{11}X_{11}B_{11}^T - X_{11} = C_{11} - A_{11}X_{12}B_{12}^T - A_{12}(X_{21}B_{11}^T + X_{22}B_{12}^T) \\ A_{11}X_{12}B_{22}^T - X_{12} = C_{12} - A_{12}X_{22}B_{22}^T \\ A_{22}X_{21}B_{11}^T - X_{21} = C_{21} - A_{22}X_{22}B_{12}^T \\ A_{22}X_{22}B_{22}^T - X_{22} = C_{22}. \end{cases} \tag{14}$$

From (14) we observe that by computing $X_{21}B_{11}^T + X_{22}B_{12}^T$ before multiplying with A_{12} and by computing $X_{22}B_{12}^T$ only once we avoid redundant computations. Consequently, for SYDT/LYDT we broadcast each subsolution X_{ij} in the process row corresponding to block row i and a sum of matrix products in the process column corresponding to block column j.

The generalized matrix equations are solved as follows: for GCSY the SYCT methodology is used except for the fact that we are now working with two

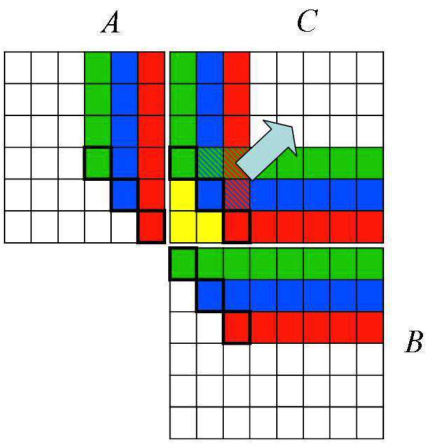


Fig. 1. The SYCT wavefront: standard, one-sided, non-symmetric. Yellow blocks correspond to already solved blocks, the blocks with bold borders correspond to the current position of the wavefront, blocks with the same color are used together in subsystems solves or GEMM-updates, stripe-colored blocks are involved in several rounds of GEMM-updates corresponding to the same block diagonal. The wavefront direction is indicated by the arrow. Each subsolution is broadcasted in the corresponding block row and column.

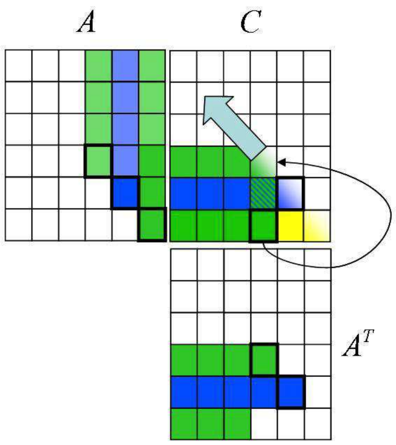


Fig. 2. The symmetric LYCT wavefront: standard, one-sided, symmetric. Each subsolution (i.e., a block of the solution matrix X) located outside the main block anti-diagonal is broadcasted in the corresponding block row and column and the block row corresponding to its transposed position.

equations at the same time. The methods of SYDT and LYDT are generalized for GSYL and GLYCT/GLYDT, respectively, in a similar fashion by using two extra buffers for storing intermediate sums of matrix products.

We remark that in a trivially blocked solver for the two-sided Lyapunov equations, we may reformulate the updates of the main block diagonal of C in terms of SYR2K-operation, as in the LYCT case. However, this is not possible when we use the intermediate sums of matrix products to reduce the complexity.

4 Condition Estimators for Triangular Matrix Equations

We utilize a general method [11,12,20] for estimating $\|A^{-1}\|_1$ for a square matrix A using reverse communication of $A^{-1}x$ and $A^{-T}x$, where $\|x\|_2 = 1$. In particular, for SYCT this approach is based on linear system $Z_{\text{SYCT}}x = y$ (see Section 1 and Table 1) which is used to compute a lower bound of the inverse of the *separation between the matrices A and B* [27]:

$$\text{sep}(A, B) = \inf_{\|X\|_F=1} \|AX - XB\|_F = \sigma_{min}(Z_{\text{SYCT}}) = \|Z_{\text{SYCT}}^{-1}\|_2^{-1}. \qquad (15)$$

The quantity (15) is used frequently in perturbation theory and error bounds (see, e.g., [13]). The exact value can be computed at the cost $O(m^3n^3)$ flops by the SVD of Z_{SYCT} but its inverse can be estimated much cheaper by solving a few (normally around five) triangular SYCT equations to the cost $O(m^2n+mn^2)$ flops [20].

Table 1. The Kronecker product representations of $Z_\star$ and $Z_\star^T$ considered in condition estimation of the standard and generalized matrix equations (1)-(8)

Acronym (ACRO)	$Z_\star$	$Z_\star^T$
SYCT	$I_n \otimes A - B^T \otimes I_m$	$I_n \otimes A^T - B \otimes I_m$
SYDT	$B \otimes A - I_{m \cdot n}$	$B^T \otimes A^T - I_{m \cdot n}$
LYCT	$I_m \otimes A + A \otimes I_m$	$I_m \otimes A^T - A^T \otimes I_m$
LYDT	$A \otimes A - I_{m^2}$	$A^T \otimes A^T - I_{m^2}$
GCSY	$\begin{bmatrix} I_n \otimes A & -B^T \otimes I_m \\ I_n \otimes D & -E^T \otimes I_m \end{bmatrix}$	$\begin{bmatrix} I_n \otimes A^T & I_n \otimes D^T \\ -B \otimes I_m & -E \otimes I_m \end{bmatrix}$
GSYL	$B \otimes A - D \otimes C$	$B^T \otimes A^T - D^T \otimes C^T$
GLYCT	$A \otimes A - E \otimes E$	$A^T \otimes A^T - E^T \otimes E^T$
GLYDT	$E \otimes A + A \otimes E$	$E^T \otimes A^T + A^T \otimes E^T$

This estimation method is applied to all matrix equations by considering the corresponding Kronecker product representation of the associated Sylvester-type operator (see Table 1). However, notice that condition estimation of GCSY is not as straightforward as for the uncoupled equations, since transposing Z_{GCSY} is not just a matter of transposing all involved left hand side matrices (excluding the solution), but requires a different algorithm (see, e.g., [21]).

The condition estimator in [20] was based on the serial LAPACK-routine `DLACON` [1]. The parallel version we use is implemented in ScaLAPACK [3,26] as the auxiliary routine `PDLACON`.

In our parallel estimators, we compute P_c different estimates independently and concurrently, one for each process column by taking advantage of the fact that `PDLACON` requires a column vector distributed over a single process column as right hand side, and we form the global maximum by a scalar *all-to-all reduction* [7] in each process row (which is negligible in terms of execution time). The column vector y in each process column is constructed by performing an *all-to-all broadcast* [7] of the local pieces of the right hand side matrix or matrices in each process row, forming P_c different right hand side vectors. Altogether, we compute P_c different estimates (lower bounds of the associated sep^{-1}-function) and choose the largest value at the same cost in time as computing only one estimate.

5 Experimental Results

Our target machine is the 64-bit Opteron Linux Cluster *sarek* with 192 dual AMD Opteron nodes (2.2 GHz), 8Gb RAM per node and a Myrinet-2000 high-performance interconnect with 250 MB/sec bandwidth. All experiments where conducted using the Portland Group's `pgf77 1.2.5 64-bit` compiler, the compiler flag `-fast` and the following software: MPICH-GM 1.5.2 [23], LAPACK 3.0 [22], GOTO-BLAS r0.94 [6], ScaLAPACK 1.7.0 [26], BLACS 1.1patch3 [4] and RECSY 0.01alpha [25] (used as node solvers). All experiments are conducted in double precision arithmetic.

Table 2. Condition estimation of GSYL invoking `PGSYLCON` on *sarek* using the blocksize 64. All timings are in seconds. For this table, (A, C) and (B, D) are chosen as random upper triangular matrices with specified eigenvalues as $\lambda^{(i)}_{(A,C)} = i$ and $\lambda^{(i)}_{(B,D)} = -i$, respectively. The known solution X is a random matrix with uniform distribution in the interval $[-1, 1]$.

$m = n$	$P_r \times P_c$	Time	*iter*	*est*	R_a	R_r	E_a	E_r
1024	1×1	12.7	5	0.6E-03	0.2E-06	0.1E+01	0.1E-11	0.2E-14
1024	2×2	7.4	5	0.6E-03	0.1E-06	0.1E+01	0.1E-11	0.2E-14
1024	4×4	3.9	5	0.6E-03	0.1E-06	0.1E+01	0.1E-11	0.2E-14
1024	8×8	2.1	5	0.6E-03	0.1E-06	0.1E+01	0.1E-11	0.2E-14
2048	1×1	86.6	5	0.3E-03	0.1E-05	0.1E+01	0.3E-11	0.2E-14
2048	2×2	48.3	5	0.3E-03	0.1E-05	0.1E+01	0.2E-11	0.2E-14
2048	4×4	21.9	5	0.3E-03	0.1E-05	0.1E+01	0.3E-11	0.2E-14
2048	8×8	9.7	5	0.3E-03	0.1E-05	0.1E+01	0.2E-11	0.2E-14
4096	1×1	923.9	7	0.2E-03	0.1E-04	0.1E+01	0.7E-11	0.3E-14
4096	2×2	503.3	7	0.2E-03	0.1E-04	0.1E+01	0.6E-11	0.2E-14
4096	4×4	193.8	7	0.2E-03	0.1E-04	0.1E+01	0.6E-11	0.2E-14
4096	8×8	77.5	7	0.2E-03	0.1E-04	0.1E+01	0.8E-11	0.3E-14
8192	1×1	5302.4	5	0.8E-04	0.1E-03	0.1E+01	0.1E-10	0.3E-14
8192	2×2	2625.9	5	0.8E-04	0.1E-03	0.1E+01	0.1E-10	0.3E-14
8192	4×4	904.5	5	0.8E-04	0.1E-03	0.1E+01	0.1E-10	0.3E-14
8192	8×8	331.4	5	0.8E-04	0.1E-03	0.1E+01	0.1E-10	0.3E-14

In Table 2, we present performance results for the parallel GSYL condition estimator `PGSYLCON` solving well-conditioned problems using the corresponding parallel triangular GSYL solver `PTRGSYLD`. For this table, *iter* is the number of iterations and calls to the triangular solver `PTRGSYLD`, *est* is the lower bound estimate of $\text{sep}^{-1}[\text{GSYL}]$, R_a, R_r, E_a and E_r correspond to the absolute and relative residual and error norms and are computed as follows:

$$R_a = \|E - A\tilde{X}B + C\tilde{X}D\|, \tag{16}$$

$$R_r = (\epsilon_{\text{mach}}^{-1} R_a)/((\|A\|\|B\| + \|C\|\|D\|)\|\tilde{X}\| + \|E\|), \tag{17}$$

$$E_a = \|X - \tilde{X}\|, \tag{18}$$

$$E_r = E_a/\|X\|. \tag{19}$$

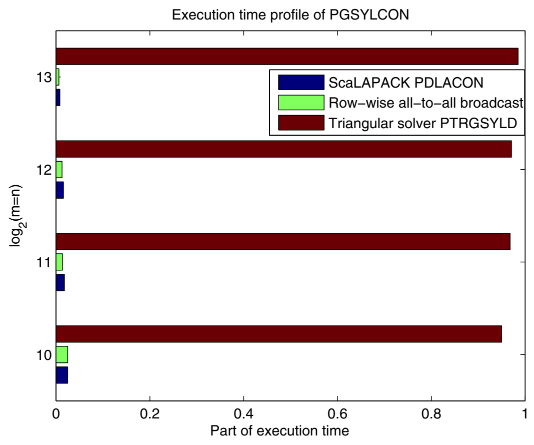


Fig. 3. Execution time profile of `PGSYLCON` on *sarek* using the blocksize 64. The results are typical for what we found using multiple (4×4) processors to solve the problem.

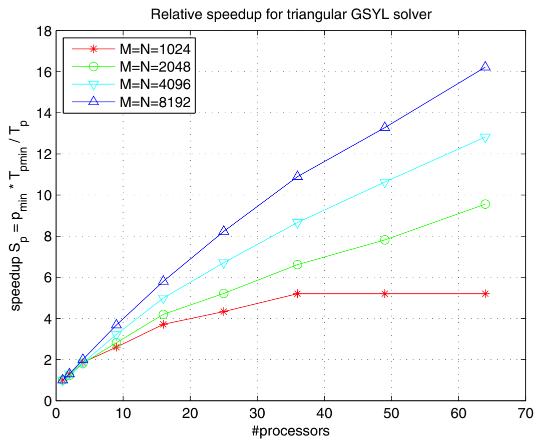


Fig. 4. Parallel speedup of `PTRGSYLD` on *sarek* using the blocksize 64

Here, $\epsilon_{\mathrm{mach}}(\approx 2.2 \times 10^{-16})$ is the relative machine precision and X and $\tilde{X}$ are the known and the computed solutions, respectively. The relative residual norm is computed in the 1-norm and the absolute residual and the error norms are computed in the Frobenius norm, respectively. Ideally, the relative residual norm should be of $O(1)$ [21], which is fulfilled remarkably well for this set of test problems. The high absolute residual norm results emerge from the large norms $(O(n^{3/2}))$ of the known left hand side matrices. The stable value of *est* depends on that exactly the same problem is generated and solved for every value of $m = n$.

An execution time profile of PGSYLCON is presented in Figure 3. The major part of the work is spent in the triangular solver, which is called around five (5) times (see Table 2). The influences of PDLACON and the all-to-all broadcast of the right hand side in each process row on the total execution time are diminished as the problem size grows. This implies that any effort on improving the condition estimator should concentrate on the triangular solver.

A representative selection of parallel speedup results for the triangular GSYL solver is presented in Figure 4. The algorithms for the other equations (see Section 1) have similar good qualitative behavior.

Acknowledgements

This research was conducted using the resources of the High Performance Computing Center North (HPC2N) [14]. Financial support has been provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.C.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
2. Bartels, R.H., Stewart, G.W.: Algorithm 432: The Solution of the Matrix Equation $AX - BX = C$. Communications of the ACM 8, 820–826 (1972)
3. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J.W., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
4. BLACS - Basic Linear Algebra Communication Subprograms. See http://www.netlib.org/blacs/index.html
5. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
6. GOTO-BLAS - High-Performance BLAS by Kazushige Goto. See http://www.cs.utexas.edu/users/flame/goto/
7. Grama, A., Gupta, A., Karypsis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Reading (2003)
8. Granat, R., Jonsson, I., Kågström, B.: Combining Explicit and Recursive Blocking for Solving Triangular Sylvester-Type Matrix Equations in Distributed Memory Platforms. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 742–750. Springer, Heidelberg (2004)
9. Granat, R., Kågström, B.: Evaluating Parallel Algorithms for Solving Sylvester-Type Matrix Equations: Direct Transformation-Based versus Iterative Matrix-Sign-Function-Based Methods. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 719–729. Springer, Heidelberg (2006)
10. Granat, R., Kågström, B., Poromaa, P.: Parallel ScaLAPACK-style Algorithms for Solving Continuous-Time Sylvester Equations. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 800–809. Springer, Heidelberg (2003)

11. Hager, W.W.: Condition estimates. SIAM J. Sci. Statist. Comput. (3), 311–316 (1984)
12. Higham, N.J.: Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. ACM Trans. of Math. Software 14(4), 381–396 (1988)
13. Higham, N.J.: Perturbation theory and backward error for $AX - XB = C$. BIT 33(1), 124–136 (1993)
14. HPC2N - High Performance Computing Center North. See `http://www.hpc2n.umu.se`
15. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems. I. One-sided and coupled Sylvester-type matrix equations. ACM Trans. Math. Software 28(4), 392–415 (2002)
16. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems. II. Two-sided and generalized Sylvester and Lyapunov matrix equations. ACM Trans. Math. Software 28(4), 416–435 (2002)
17. Jonsson, I., Kågström, B.: RECSY - A High Performance Library for Solving Sylvester-Type Matrix Equations. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 810–819. Springer, Heidelberg (2003)
18. Kågström, B., Ling, P., Van Loan, C.: GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. ACM Trans. Math. Software 24(3), 268–302 (1998)
19. Kågström, B., Ling, P., Van Loan, C.: Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. ACM Trans. Math. Software 24(3), 303–316 (1998)
20. Kågström, B., Poromaa, P.: Distributed and shared memory block algorithms for the triangular Sylvester equation with sep^{-1} estimators. SIAM J. Matrix Anal. Appl. 13(1), 90–101 (1992)
21. Kågström, B., Poromaa, P.: Computing eigenspaces with specified eigenvalues of a regular matrix pair (A, B) and condition estimation: theory, algorithms and software. Numer. Algorithms 12(3-4), 369–407 (1996)
22. LAPACK - Linear Algebra Package. See `http://www.netlib.org/lapack/`
23. MPI - Message Passing Interface. See `http://www-unix.mcs.anl.gov/mpi/`
24. Poromaa, P.: Parallel Algorithms for Triangular Sylvester Equations: Design, Scheduling and Scalability Issues. In: Kagström, B., Elmroth, E., Waśniewski, J., Dongarra, J.J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 438–446. Springer, Heidelberg (1998)
25. RECSY - High Performance library for Sylvester-type matrix equations, See `http://www.cs.umu.se/research/parallel/recsy`
26. ScaLAPACK Users' Guide, see `http://www.netlib.org/scalapack/slug/`
27. Stewart, G.W., Sun, J.-G.: Matrix Perturbation Theory. Academic Press, New York (1990)

LAPACK-Style Codes for Pivoted Cholesky and QR Updating

Sven Hammarling[1], Nicholas J. Higham[2], and Craig Lucas[3]

[1] NAG Ltd.,Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, England and School of Mathematics, University of Manchester, M13 9PL, England
`sven@nag.co.uk`
`http://www.nag.co.uk/about/shammarling.asp`
[2] School of Mathematics, University of Manchester, M13 9PL, England
`higham@ma.man.ac.uk`
`http://www.ma.man.ac.uk/~higham`
[3] Manchester Computing, University of Manchester, M13 9PL, England
`craig.lucas@manchester.ac.uk`
`http://www.ma.man.ac.uk/~clucas`.*

Abstract. Routines exist in LAPACK for computing the Cholesky factorization of a symmetric positive definite matrix and in LINPACK there is a pivoted routine for positive *semi*definite matrices. We present new higher level BLAS LAPACK-style codes for computing this pivoted factorization. We show that these can be many times faster than the LINPACK code. Also, with a new stopping criterion, there is more reliable rank detection and smaller normwise backward error. We also present algorithms that update the QR factorization of a matrix after it has had a block of rows or columns added or a block of columns deleted. This is achieved by updating the factors Q and R of the original matrix. We present some LAPACK-style codes and show these can be much faster than computing the factorization from scratch.

1 Pivoted Cholesky Factorization

1.1 Introduction

The Cholesky factorization of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ has the form $A = LL^T$, where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix with positive diagonal elements. If A is positive *semi*definite, of rank r, there exists a Cholesky factorization with *complete pivoting* ([7, Thm. 10.9], for example). That is, there exists a permutation matrix $P \in \mathbb{R}^{n\times n}$ such that P^TAP has a unique Cholesky factorization

$$P^TAP = LL^T, \quad L = \begin{bmatrix} L_{11} & 0 \\ L_{12} & 0 \end{bmatrix},$$

where $L_{11} \in \mathbb{R}^{r\times r}$ is lower triangular with positive diagonal elements.

* This work was supported by an EPSRC Research Studentship.

1.2 Algorithms

In LAPACK [1] there are Level 2 BLAS and Level 3 BLAS routines for computing the Cholesky factorization in the full rank case and without pivoting. In LINPACK [3] the routine `xCHDC` performs the Cholesky factorization with complete pivoting, but effectively uses only Level 1 BLAS. For computational efficiency we would like a pivoted routine that exploits the Level 2 or Level 3 BLAS. The LAPACK Level 3 algorithm cannot be pivoted, so we instead start with the Level 2 algorithm. The LAPACK 'Gaxpy' Level 2 BLAS algorithm is:

Algorithm 1. *This algorithm computes the Cholesky factorization $A = LL^T$ of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$, overwriting A with L.*

```
    Set $L$ = lower triangular part of $A$
    for $j = 1{:}n$
(*)     $L(j,j) = L(j,j) - L(j,1{:}j-1)L(j,1{:}j-1)^T$
(#)     if $L(j,j) \le 0$, return, end % Quit if $A$ not positive definite.
        $L(j,j) = \sqrt{L(j,j)}$
        % Update $j$th column
        if $1 < j < n$
            $L(j+1{:}n,j) = L(j+1{:}n,j) - L(j+1{:}n,1{:}j-1)L(j,1{:}j-1)^T$
        end
        if $j < n$
            $L(j+1{:}n,j) = L(j+1{:}n,j)/L(j,j)$
        end
    end
```

This algorithm requires $n^3/3$ flops. We can introduce pivoting into Algorithm 1, for $L = (\ell_{ij})$, by finding the largest possible ℓ_{jj} at $(*)$ from the remaining $n-j+1$ diagonal elements and using it as the pivot. We find

$$q = \min\Big\{p : L(p,p) - d(p) = \max_{j\le i\le n}\{L(i,i) - d(i)\}\Big\}, \tag{1.1}$$

where d is a vector of dot products with

$$d(i) = L(i,1{:}j-1)L(i,1{:}j-1)^T, \quad i = j{:}n, \tag{1.2}$$

and swap rows and columns q and j, putting the pivot ℓ_{qq} into the lead position. This is *complete pivoting.*

For computational efficiency we can store the inner products in (1.2) and update them on each iteration. This approach gives a pivoted gaxpy algorithm. The pivoting overhead is $3(r+1)n - 3/2(r+1)^2$ flops and $(r+1)n - (r+1)^2/2$ comparisons, where $r = \operatorname{rank}(A)$.

The numerical estimate of the rank of A, $\hat{r}$, can be determined by a stopping criterion at (#) in Algorithm 1. At the jth iteration if the pivot, which we will denote by $\chi_{jj}^{(j)}$, satisfies an appropriate condition then we set the trailing matrix $L(j{:}n,j{:}n)$ to zero and the computed rank is $j-1$. Three possible stopping criteria are discussed in [7, Sec. 10.3.2]. The first is used in LINPACK's code for the Cholesky factorization with complete pivoting, `xCHDC`. Here the algorithm is stopped on the kth step if

$$\chi_{ii}^{(k)} \le 0, \quad i = k{:}\, n. \tag{1.3}$$

In practice $\hat{r}$ may be greater than r due to rounding errors. In [7] the other two criteria are shown to work more effectively. The first is

$$\|\widetilde{S}_k\| \le \epsilon \|A\| \quad \text{or} \quad \chi_{ii}^{(k)} \le 0, \quad i = k{:}\, n, \tag{1.4}$$

where $\widetilde{S}_k = A_{22} - A_{12}^T A_{11}^{-1} A_{12}$, with $A_{11} \in \mathbb{R}^{k \times k}$ the leading submatrix of A, is the Schur complement of A_{11} in A, while the second related criterion is

$$\max_{k \le i \le n} \chi_{ii}^{(k)} \le \epsilon \chi_{11}^{(1)}, \tag{1.5}$$

where in both cases $\epsilon = nu$, and u is the unit roundoff. We have used the latter criterion, preferred for its lower computational cost. We do not attempt to detect indefiniteness, the stopping criteria is derived for semidefinite matrices only. See [8] for a discussion on this.

We derive a blocked algorithm by using the fact that we can write, for the semidefinite matrix $A^{(k-1)} \in \mathbb{R}^{n \times n}$ and $n_b \in \mathbb{N}$ [4],

$$A^{(k-1)} = \begin{bmatrix} A_{11}^{(k-1)} & A_{12}^{(k-1)} \\ A_{12}^{T(k-1)} & A_{22}^{(k-1)} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-n_b} \end{bmatrix} \begin{bmatrix} I_{n_b} & 0 \\ 0 & A^{(k)} \end{bmatrix} \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-n_b} \end{bmatrix}^T,$$

where $L_{11} \in \mathbb{R}^{n_b \times n_b}$ and $L_{21} \in \mathbb{R}^{(n-n_b) \times n_b}$ form the first n_b columns of the Cholesky factor L of $A^{(k-1)}$. Now to complete our factorization of $A^{(k-1)}$ we need to factor the reduced matrix

$$A^{(k)} = A_{22}^{(k-1)} - L_{21} L_{21}^T, \tag{1.6}$$

which we can explicitly form, taking advantage of symmetry.

From this representation we can derive a block algorithm. At the kth step we factor n_b columns, by applying a pivoted Algorithm 1 to the leading principal $n_b \times n_b$ submatrix of $A^{(k)}$ and then update the trailing matrix according to (1.6) and continue.

At each step the Level 2 part of the algorithm requires $(n - (k-1)n_b)n_b^2$ flops and the Level 3 update requires $(n - kn_b)^3/3$ flops. The Level 3 fraction is approximately $1 - 3n_b/2n$.

1.3 Numerical Experiments

Our test machine was a 1400MHz AMD Athlon. Using ATLAS [2] BLAS and the GNU77 compiler version 3.2 with no optimization. We tested and compared four Fortran subroutines: LINPACK's `DCHDC`, `DCHDC` altered to use our stopping criterion, and LAPACK-style implementations of a level 2 pivoted Gaxpy algorithm (`LEV2PCHOL`) and level 3 pivoted Gaxpy algorithm (`LEV3PCHOL`) .

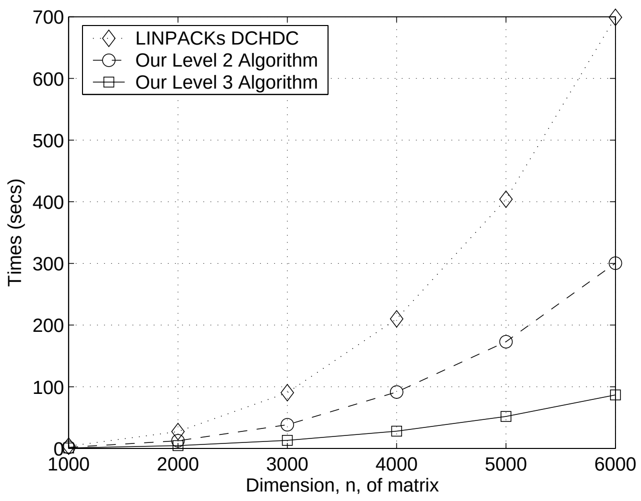


Fig. 1. Comparison of speed for different n

We first compared the speed of the factorization of the LINPACK code and our Level 2 and 3 routines for different sizes of $A \in \mathbb{R}^{n\times n}$. We generated random symmetric positive semidefinite matrices of order n and rank $r = 0.7n$. For each value of n the codes were run four times and the mean times are shown in Figure 1. We achieve a good speedup, with the Level 3 code as much as 8 times faster than the LINPACK code. Our level three code achieves 830 MFlops/sec compared to the LINPACK code with 100 MFlops/sec.

We also compared the speed of the unpivoted LAPACK subroutines against our Level 3 pivoted code, using full rank matrices, to demonstrate the pivoting overhead. The ratio of speed of the pivoted codes to the unpivoted codes varies smoothly from 1.6 for $n = 1000$ to 1.01 for $n = 6000$, so the pivoting overhead is negligible in practice for large n (recall that the pivoting overhead is about $3rn - 3/2r^2$ flops within the $O(n^3)$ algorithm). The use of the pivoted codes instead of the unpivoted ones could be warranted if there is any doubt over whether a matrix is positive definite.

We tested all four subroutines on a further set of random positive semidefinite matrices, this time with pre-determined eigenvalues, similarly to the tests in [6]. For matrices of rank r we chose the nonzero eigenvalues in three ways:

- Case 1: $\lambda_1 = \lambda_2 = \cdots = \lambda_{r-1} = 1, \quad \lambda_r = \alpha \leq 1$
- Case 2: $\lambda_1 = 1, \quad \lambda_2 = \lambda_3 = \cdots = \lambda_r = \alpha \leq 1$
- Case 3: $\lambda_i = \alpha^{i-1}, \quad 1 \leq i \leq r, \quad \alpha \leq 1$

Here, α was chosen to vary $\kappa_2(A) = \lambda_1/\lambda_r$. For each case we constructed a set of 100 matrices by using every combination of:

$$n = \{70,\ 100,\ 200,\ 500,\ 1000\},$$

$$\kappa_2(A) = \{1,\ 1\text{e}{+}3,\ 1\text{e}{+}6,\ 1\text{e}{+}9,\ 1\text{e}{+}12\},$$
$$r = \{0.2n,\ 0.3n,\ 0.5n,\ 0.9n\},$$

where $r = \text{rank}(A)$. We computed the relative normwise backward error

$$\frac{\|A - \widehat{P}\widehat{L}\widehat{L}^T\widehat{P}^T\|_2}{\|A\|_2},$$

for the computed Cholesky factor $\widehat{L}$ and permutation matrix $\widehat{P}$.

Table 1. Maximum normwise backward errors

n	70	100	200	500	1000
`DCHDC`	3.172e-13	1.498e-13	1.031e-12	2.823e-12	4.737e-11
`DCHDC` with (1.5)	7.778e-15	9.014e-15	1.810e-14	7.746e-14	1.991e-13
`LEV2PCHOL`	4.633e-15	9.283e-15	1.458e-14	7.290e-14	1.983e-13
`LEV3PCHOL`	4.633e-15	9.283e-15	1.710e-14	8.247e-14	2.049e-13

There was little difference between the normwise backward errors in the three test cases; Table 1 shows the maximum values over all cases for different n. The codes with the new stopping criterion give smaller errors than the original LINPACK code. In fact, for all the codes with our stopping criterion $\hat{r} = r$, and so the rank was detected exactly. This was not the case for the unmodified `DCHDC`, and the error, $\hat{r} - r$, is shown in Table 2.

Table 2. Errors in computed rank for `DCHDC`

n	70	100	200	500	1000
min	0	0	1	4	4
max	10	12	16	16	19

The larger backward error for the original `DCHDC` is due to the stopping criterion. As Table 2 shows, the routine is often terminated after more steps than our codes, adding more nonzero columns to $\widehat{L}$.

1.4 Conclusions

Our codes for the Cholesky factorization with complete pivoting are much faster than the existing LINPACK code. Furthermore, with a new stopping criterion the rank is revealed much more reliably, and this leads to a smaller normwise backward error. For more detailed information on the material in this section see [8]. For details of a parallel implementation see [9].

2 Updating the QR Factorization

2.1 Introduction

We wish to update efficiently the QR factorization

$$A = QR \in \mathbb{R}^{m\times n},$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is upper trapezoidal. That is we wish to find $\widetilde{A} = \widetilde{Q}\widetilde{R}$, where $\widetilde{A}$ is A with rows or columns added or deleted. We seek to do this without recomputing the factorization from scratch. We will assume that A and $\widetilde{A}$ have full rank.

We consider the cases of adding blocks of rows and columns and deleting blocks of columns. This has application to updating the least squares problem where observations or variables are added or deleted. Where possible we derive blocked algorithms.

2.2 Adding a Block of Rows

If we add a block of p rows, $U \in \mathbb{R}^{p\times n}$, just before the kth row of A we can write

$$\widetilde{A} = \begin{bmatrix} A(1\colon k-1, 1\colon n) \\ U \\ A(k\colon m, 1\colon n) \end{bmatrix}$$

and we can define a permutation matrix, P, such that

$$P\widetilde{A} = \begin{bmatrix} A \\ U \end{bmatrix},$$

and

$$\begin{bmatrix} Q^T & 0 \\ 0 & I_p \end{bmatrix} P\widetilde{A} = \begin{bmatrix} R \\ U \end{bmatrix}. \tag{2.1}$$

Thus to find $\widetilde{A} = \widetilde{Q}\widetilde{R}$, we can define n Householder matrices to eliminate U to give

$$H_n \dots H_1 \begin{bmatrix} R \\ U \end{bmatrix} = \widetilde{R},$$

so we have

$$\widetilde{A} = \left(P^T \begin{bmatrix} Q & 0 \\ 0 & I_p \end{bmatrix} H_1 \dots H_n \right) \widetilde{R} = \widetilde{Q}\widetilde{R}.$$

The Householder matrix, $H_j \in \mathbb{R}^{(m+p)\times(m+p)}$, will zero the jth column of U. Its associated Householder vector, $v_j \in \mathbb{R}^{(m+p)}$, is such that

$$\begin{aligned} v_j(1\colon j-1) &= 0, \quad v_j(j) = 1, \\ v_j(j+1\colon m) &= 0, \\ v_j(m+1\colon m+p) &= x/(r_{jj} - \|\,[\,r_{jj} \quad x^T\,]\,\|_2), \text{ where } x = U(1\colon p, j). \end{aligned}$$

We can derive a blocked algorithm by using the representation of the product of Householder matrices in [10].

2.3 Deleting a Block of Columns

If we delete a block of p columns, from the kth column onwards, from A, we can write

$$\widetilde{A} = [\, A(1{:}\, m, 1{:}\, k-1) \quad A(1{:}\, m, k+p{:}\, n) \,]$$

and then

$$Q^T \widetilde{A} = [\, R(1{:}\, m, 1{:}\, k-1) \quad R(1{:}\, m, k+p{:}\, n) \,]\,. \tag{2.2}$$

Thus we can define $n-p-k+1$ Householder matrices, $H_j \in \mathbb{R}^{m\times m}$, with associated Householder vectors, $v_j \in \mathbb{R}^{(p+1)}$ such that

$$\begin{aligned} v_j(1{:}\, j-1) &= 0, \quad v_j(j) = 1, \\ v_j(j+1{:}\, j+p) &= x/((\widetilde{Q}^T\widetilde{A})_{jj} - \|\, [\, (\widetilde{Q}^T\widetilde{A})_{jj} \quad x^T \,] \,\|_2), \\ &\qquad \text{where } x = Q^T\widetilde{A}(j+1{:}\, j+p, j), \\ v_j(j+p+1{:}\, m) &= 0. \end{aligned}$$

The H_j can be used to eliminate the subdiagonal of $Q^T\widetilde{A}$ to give

$$(H_{n-p} \ldots H_k Q^T)\widetilde{A} = \widetilde{Q}^T\widetilde{A} = \widetilde{R},$$

where $\widetilde{R} \in \mathbb{R}^{m\times(n-p)}$ is upper trapezoidal and $\widetilde{Q} \in \mathbb{R}^{m\times m}$ is orthogonal.

2.4 Adding a Block of Columns

If we add a block of p columns, $U \in \mathbb{R}^{m\times p}$, in the kth to $(k+p-1)$st positions of A, we can write

$$\widetilde{A} = [\, A(1{:}\, m, 1{:}\, k-1) \quad U \quad A(1{:}\, m, k{:}\, n) \,]$$

and

$$Q^T\widetilde{A} = \begin{bmatrix} R_{11} & V_{12} & R_{12} \\ 0 & V_{22} & R_{23} \\ 0 & V_{32} & 0 \end{bmatrix},$$

where $R_{11} \in \mathbb{R}^{(k-1)\times(k-1)}$ and $R_{23} \in \mathbb{R}^{(n-k+1)\times(n-k+1)}$ are upper triangular. Then if V_{32} has the (blocked) QR factorization $V_{32} = Q_V R_V \in \mathbb{R}^{(m-n)\times p}$ we have

$$\begin{bmatrix} I_n & 0 \\ 0 & Q_V^T \end{bmatrix} Q^T\widetilde{A} = \begin{bmatrix} R_{11} & V_{12} & R_{12} \\ 0 & V_{22} & R_{23} \\ 0 & R_V & 0 \end{bmatrix}.$$

We then eliminate the upper triangular part of R_V and the lower triangular part of V_{22} with Givens rotations, which makes R_{23} full and the bottom right block upper trapezoidal. So we have finally

$$\begin{aligned} G(k+2p-2, k+2p-1)^T \ldots G(k+p, k+p+1)^T G(k, k+1)^T \\ \ldots G(k+p-1, k+p)^T \begin{bmatrix} I_n & 0 \\ 0 & Q_V^T \end{bmatrix} Q^T\widetilde{A} = \widetilde{R}, \end{aligned}$$

where $G(i,j)$ are Givens rotations acting on the ith and jth rows.

2.5 Numerical Experiments

We tested the speed of LAPACK-style implementations of our algorithms for updating after adding (DELCOLS) and deleting (ADDCOLS) columns, against LAPACK's DGEQRF, for computing the QR factorization of a matrix.

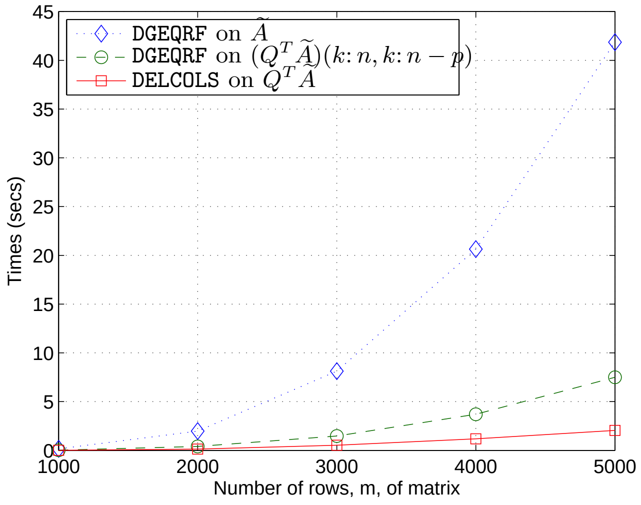


Fig. 2. Comparison of speed for DELCOLS with $k = 1$ for different m

We tested the codes with $m = \{1000,\ 2000,\ 3000,\ 4000,\ 5000\}$ and $n = 0.3m$, and the number of columns added or deleted was $p = 100$. We timed our codes acting on $Q^T\widetilde{A}$, the starting point for computing $\widetilde{R}$, and in the case of adding columns we included in our timings the computation of Q^TU, which we formed with the BLAS routine DGEMM. We also timed DGEQRF acting on only the part of $Q^T\widetilde{A}$ that needs to be updated, the nonzero part from row and column k onwards. Here we can construct $\widetilde{R}$ with this computation and the original R. Finally, we timed DGEQRF acting on $\widetilde{A}$. We aim to show our codes are faster than these alternatives. In all cases an average of three timings is given.

To test our code DELCOLS we chose $k = 1$, the position of the first column deleted, where the maximum amount of work is required to update the factorization. We timed DGEQRF on $\widetilde{A}$, DGEQRF on $(Q^T\widetilde{A})(k\colon n, k\colon n-p)$ which computes the nonzero entries of $\widetilde{R}(k\colon m, p+1\colon n)$ and DELCOLS on $Q^T\widetilde{A}$. The results are given in Figure 2. Our code is much faster than recomputing the factorization from scratch with DGEQRF, and for $n = 5000$ there is a speedup of 20. Our code is also faster than using DGEQRF on $(Q^T\widetilde{A})(k\colon n, k\colon n-p)$, where there is a maximum speedup of over 3. LAPACK's QR code achieves around 420 MFlops/sec, our's gets 190MFlops/sec, but can't use a blocked algorithm for better data reuse.

We then considered the effect of varying p with DELCOLS for fixed $m = 3000$, $n = 1000$ and $k = 1$. We chose $p = \{100,\ 200,\ 300,\ 400,\ 500,\ 600\ 700,\ 800\}$. As

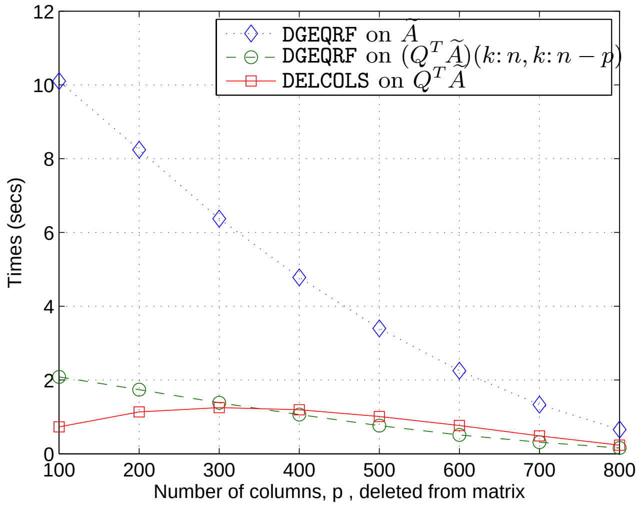


Fig. 3. Comparison of speed for `DELCOLS` for different p

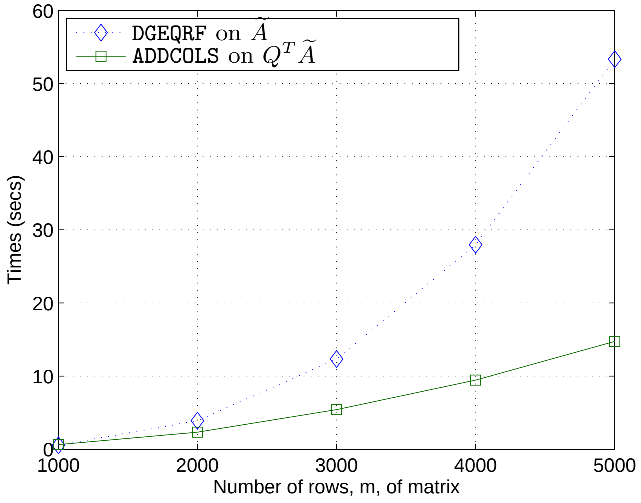


Fig. 4. Comparison of speed for `ADDCOLS` with $k = 1$ for different m

we delete more columns from A there are fewer columns to update, but more work is required for each one. We timed `DGEQRF` on $\widetilde{A}$, `DGEQRF` on $(Q^T\widetilde{A})(k{:}\,n, k{:}\,n-p)$ which computes the nonzero entries of $\widetilde{R}(k{:}\,m, k{:}\,n-p)$ and `DELCOLS` on $Q^T\widetilde{A}$. The results are given in Figure 3. The timings for `DELCOLS` are relatively level and peak at $p = 300$, whereas the timings for the other codes obviously decrease with p. The speedup of our code decreases with p, and from $p = 300$ there is little difference between our code and DGEQRF on $(Q^T\widetilde{A})(k\colon n, k\colon n-p)$.

To test ADDCOLS we generated random matrices $A \in \mathbb{R}^{m\times n}$ and $U \in \mathbb{R}^{m\times p}$. We set $k = 1$ where maximum updating is required. We timed DGEQRF on $\widetilde{A}$ and ADDCOLS on $Q^T\widetilde{A}$, including the computation of Q^TU with DGEMM. The results are given in Figure 4. Here our code achieves a speedup of over 3 for $m = 5000$ over the complete factorization of $\widetilde{A}$. Our code achieves 55 MFlops/sec compared to LAPACK's 420 MFlops/sec, but we use Givens rotations and only have a small fraction that uses a blocked algorithm. We do not vary p as this increases the work for our code and DGEQRF on $(Q^T\widetilde{A})(k\colon m, k\colon n+p)$ roughly equally.

2.6 Conclusions

The speed tests show that our updating algorithms are faster than computing the QR factorization from scratch or using the factorization to update columns k onward, the only columns needing updating. For more detailed information on the material in this section see [5]. Further work planned is the parallelization of the algorithms.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
2. Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/
3. Dongarra, J.J., Moler, C.B., Bunch, J.R., Stewart, G.W.: LINPACK Users' Guide. SIAM, Philadelphia (1979)
4. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore (1996)
5. Hammarling, S., Lucas, C.: Updating the QR factorization and the least squares problem, MIMS EPrint, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, 2006 (to appear)
6. Higham, N.J.: Analysis of the Cholesky decomposition of a semidefinite matrix. In: Cox, M.G., Hammarling, S.J. (eds.) Reliable Numerical Computation, pp. 161–185. Oxford University Press, Oxford (1990)
7. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. SIAM, Philadelphia (2002)
8. Lucas, C.: LAPACK-style codes for level 2 and 3 pivoted Cholesky factorizations. LAPACK Working Note 161 (February 2004)
9. Lucas, C.: Symmetric pivoting in scalapack. Cray User Group, Lugano, Switzerland (May 2006)
10. Schreiber, R., Van Loan, C.F.: A storage-efficient WY representation for products of householder transformations. SIAM J. Sci. Stat. Comput. 10(1), 53–57 (1989)

Implementing Linear Algebra Routines on Multi-core Processors with Pipelining and a Look Ahead

Jakub Kurzak[1] and Jack Dongarra[2]

[1] University of Tennessee, Knoxville TN 37996, USA
kurzak@cs.utk.edu
http://www.cs.utk.edu/~kurzak
[2] University of Tennessee, Knoxville TN 37996, USA
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
dongarra@cs.utk.edu
http://www.netlib.org/utk/people/JackDongarra/

Abstract. Linear algebra algorithms commonly encapsulate parallelism in Basic Linear Algebra Subroutines (BLAS). This solution relies on the fork-join model of parallel execution, which may result in suboptimal performance on current and future generations of multi-core processors. To overcome the shortcomings of this approach a pipelined model of parallel execution is presented, and the idea of look ahead is utilized in order to suppress the negative effects of sequential formulation of the algorithms. Application to one-sided matrix factorizations, LU, Cholesky and QR, is described. Shared memory implementation using POSIX threads is presented.

1 Introduction

The standard approach to parallelization of numerical linear algebra algorithms for shared memory systems, utilized by the LAPACK library [1,2], is to rely on parallel implementation of BLAS [3]. It is a proven method to extract parallelism from Level 3 BLAS routines. However, as the number of processors or cores grows, practice shows that parallelization of Level 1 and 2 routines is unlikely to yield speedups and can even result in slowdowns. As a result, Level 1 and 2 BLAS portions of the algorithms decrease the benefits of parallelization and limit achievable performance. A more flexible approach is required to address new generations of multi-core processors, which are expected to have tens, and possibly hundreds, of cores in near future.

The technique of *look ahead* can be used to remedy the problem by overlapping the execution of less efficient operations with the more efficient ones. Also, the use of different levels of look ahead is investigated and the idea of dynamic look ahead is discussed, where the algorithm execution path is decided at runtime.

Many of the ideas presented in this paper are already well known. Notably, an early application of dynamic operation scheduling in matrix factorization was introduced by Agarwal and Gustavson in parallel implementation of LU [4] and Cholesky factorization [5] for IBM 3090. Extensive discussion of the concept of look ahead with static scheduling was presented by Strazdins [6]. Similar techniques are also utilized in a recent work by Gustavson, Karlsson and Kågström on Cholesky factorization using packed storage [7]. Although the concept has been known for a long time, it never received a close look on its own. In particular the authors are not aware of any work, which would stress the impact of the depth of look ahead on performance and compare static and dynamic scheduling techniques, as well as show clearly how the idea generalizes to different matrix factorizations. This work attempts to fill the gap.

2 Factorizations

The LU factorization with partial row pivoting of an $m \times n$ real matrix A has the form

$$A = PLU,$$

where L is an $m \times n$ real unit lower triangular matrix and U is an $n \times n$ real upper triangular matrix for $m >= n$, or L is an $n \times n$ real unit lower triangular matrix and U is an $m \times n$ real upper triangular matrix for $m <= n$. P is a permutation matrix. The description of the block algorithm can be found in [8,9]. In LAPACK the double precision algorithm is implemented by the DGETRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK and BLAS routines: DGETF2, DLASWP, DTRSM, DGEMM, where DGETF2 factorizes a block of columns of the matrix (the panel) and DLASWP, DTRSM, DGEMM apply appropriate transformations to the submatrix to the right from the panel, which is the right-looking algorithm.

The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix A has the form

$$A = LL^T,$$

where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. The formulation of the block algorithm is analogous to the one for LU factorization. In LAPACK the double precision algorithm is implemented by the DPOTRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK and BLAS routines: DSYRK, DPOTF2, DGEMM, DTRSM. Here, for simplicity, only the case of lower triangular coefficient matrix is considered. In this case the routines DSYRK and DGEMM apply a pending update and DPOTF2 and DTRSM perform the factorization of a block of columns of the matrix (the panel), which is the left-looking algorithm.

The QR factorization of an $m \times n$ real matrix A has the form

$$A = QR,$$

where Q is an $m \times m$ real orthogonal matrix and R is an order n real upper triangular matrix. The traditional algorithm for QR factorization applies a series of elementary Householder matrices of the general form

$$H = I - \tau v v^T,$$

where v is a column vector and τ is a scalar. In the block form of the algorithm a product of nb elementary Householder matrices is represented in the form [10,11]

$$H_1 H_2 \ldots H_{nb} = I - VTV^T,$$

where V is an $m \times nb$ real matrix (block of columns) whose columns are the individual vectors v and T is an $nb \times nb$ real upper triangular matrix. For the derivation of the blocked algorithm the reader is referred to the original papers mentioned above. In LAPACK the double precision algorithm is implemented by the DGEQRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK routines: DGEQR2, DLARFT, DLARFB, where DGEQR2 and DLARFT operate on a block of columns of the matrix (the panel) and DLARFB operates on the submatrix to the right from the panel, which is the right-looking algorithm.

3 Parallelization

Block formulations of the three factorizations discussed, as well as many other one-sided factorizations, follow a common scheme. In a single step of each algorithm, first operations are applied to a single block of rows or columns, referred to as the panel, then the result is applied to the remaining portion of the matrix. The panel operations are usually implemented with Level 1 and 2 BLAS, and, in most cases, achieve the best performance when executed on a single processor. As a result, it is most straightforward to use one dimensional partitioning of work for parallel implementation, by cyclic assignment of blocks of rows or blocks of columns to processors, depending on the orientation of the panel; this is the approach used here.

It is a well known fact that matrix factorizations have left-looking and right-looking formulations [4,8,12]. It has even been observed that transition between the two can be done by automatic code transformations [13], although more powerful methods than simple dependency analysis may be necessary. Another well known fact is that the technique of look ahead can be used to significantly improve the performance of matrix factorizations, a method based on performing panel factorizations in parallel with the update to the remaining submatrix from previous step of the algorithm [6]. Also, look ahead can be of arbitrary depth and an example of software utilizing this idea is the high performance LINPACK benchmark (HPL) [14,15]. Look ahead is nothing else, but altering the order of operations in the factorization. A great number of permutations are legal, as long as algorithmic dependencies are not violated (Figure 1). It can be observed that the right-looking and left-looking formulation of a matrix factorization are on two

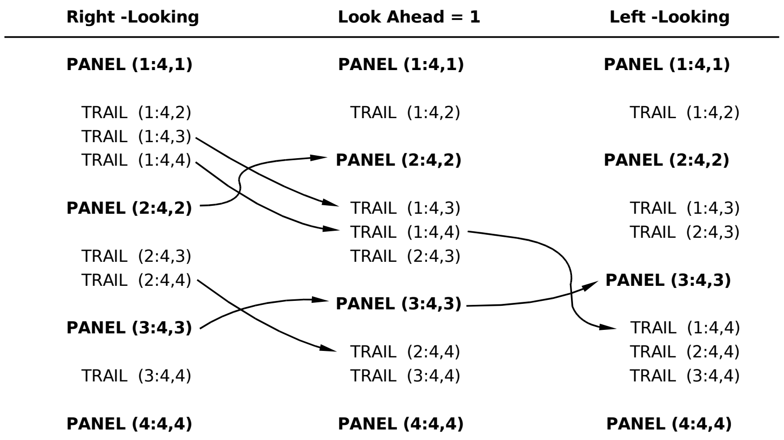


Fig. 1. Different variants of block LU factorization with 1D block cyclic work partitioning for a problem with the coefficient matrix of size 4 by 4 blocks. (PANEL - operations involved in panel factorization, TRAIL - operations involved in updating the remaining portion of the matrix).

opposite ends of a wide spectrum of possible execution paths, with look ahead providing a transition between them. If the straight right-looking formulation is regarded as one with look ahead of zero, then the left-looking formulation is equivalent to the right looking formulation with the maximum possible look ahead for a given problem.

4 Arbitrary (Static) Look Ahead

Classic implementation of a one-sided matrix factorization follows the execution pattern where the panel factorizations and the updates to the submatrix to the right from the panel are sequentially ordered. Panel factorization is performed by a single processor, while other processors are idle. Alternatively, when the first block of columns (block of rows) from step J of the factorization has been updated, one processor can perform panel factorization from step $J + 1$ on that block, when the remaining processors continue applying the update from step J, which decreases the idle time.

Panels can be factorized up to an arbitrary depth. When this depth is reached an update has to be finished before another panel can be factorized. The order of execution of operations is determined by the depth of look ahead and is static throughout the execution of the factorization. Figure 2 shows the simplified code implementing arbitrary (static) look ahead. At each step the cycle is followed by checking dependencies and stalling if necessary, executing the operation, updating the progress, and making a transition to the next operation. The transition is always known a priori based on the current stage and the depth of look ahead, since the algorithm is static in nature.

```
while (1) {
    switch (task.type) {

        case PANEL:
            check_dependencies();
            dgetf2();      dsyrk();       dgeqr2();
                           dpotf2();      dlarft();
            update_progress();
            make_transition();
            break;

        case COLUMN:
            check_dependencies();
            dlaswp();      dgemm();      dlarfb();
            dtrsm();       dtrsm();
            dgemm();
            update_progress();
            make_transition();
            break;

        case END:
            check_dependencies();
            for ()
                dlaswp();
            return;
        }
}
```

Fig. 2. Simplified code for one-sided factorizations with an arbitrary (static) look ahead

The deeper the depth, the less the processors stall at the end of the factorization. This is because the necessary panel is readily available, when the time comes to apply the update. At the same time, with deeper look ahead, stalls occur at the beginning of execution, where processors wait for a block of columns (or rows) to be updated in order to apply a forth coming panel factorization.

5 Dynamic Look Ahead

Idea of dynamic look ahead comes from the observation that the benefits of deep look ahead are obliterated by the stalls, or bubbles, at the beginning of the factorization. Basic idea behind dynamic look ahead is to implement the left-looking variant of the algorithm, where the panel factorizations are performed as soon as possible, with a modification occuring when a panel factorization introduces a stall. In such case an update to a block of columns of the right submatrix is performed instead. The updating continues only until next panel factorization is possible. As a result, dynamic look ahead is implemented by dynamic scheduling of work at runtime. Figure 3 shows the simplified code implementing dynamic look ahead. Here the steps of checking dependencies and making a transition are merged into the step of fetching next task, where the choice of transition is made dynamically at run-time depending on the progress of the execution.

```
while(1) {
    fetch_task();

    switch(task.type) {

        case PANEL:
            dgetf2();      dsyrk();      dgeqr2();
                           dpotf2();     dlarft();
            update_progress();
            break;

        case COLUMN :
            dlaswp();      dgemm();      dlarfb();
            dtrsm();       dtrsm();
            dgemm();
            update_progress();
            break;

        case END:
            for ()
                dlaswp();
            return;
        }
    }
```

Fig. 3. Simplified code for one-sided factorizations with dynamic look ahead

6 Results

Results presented here were collected on a shared memory system with two 1.8 GHz dual-core AMD Opteron^{TM} 265 processors. GOTO BLAS [16] version 1.05 was used, the most recent one at the time of writing this paper. Block size of 64 was used in all cases.

Figure 4 shows Gantt chart of execution of LU factorization using different approaches ordered from top to bottom according to their relative performance. On top is the right-looking version, where all but one processor are stalled by the panel factorization. Below is the left-looking version, which can also be considered the right-looking version with maximum possible amount of look ahead. Here stalls are eliminated at the end of factorization, but multiple bubbles are introduced at the beginning. Next are right-looking factorizations with look ahead of depth one and two. Lastly, at the bottom is the factorization with dynamic look ahead.

Figure 5, 6 and 7 show performance results for LU factorization, Cholesky factorization and QR factorization respectively.

The figures show clearly that the explicitly parallelized code benefits greatly from look ahead. Small, but noticeable performance gains can be achieved by choosing the proper depth of the static look ahead. Alternatively dynamic look ahead can be used to yield the best performance in most cases. Also, for this particular experiment, the code with look ahead outperforms the BLAS-parallel code in all investigated cases.

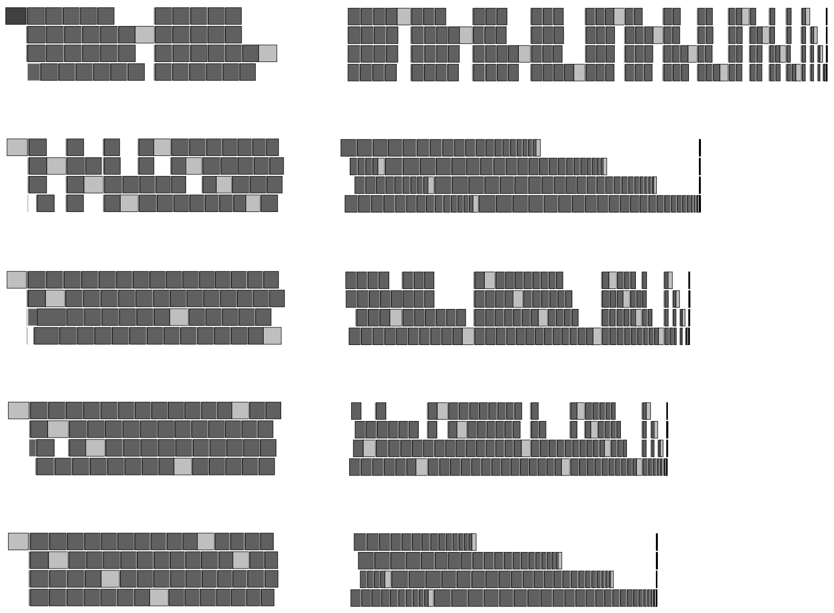


Fig. 4. Gantt chart of execution of different variants of LU factorization. From top to bottom: right-looking (look ahead of depth zero), right-looking with maximum look ahead (left-looking), right-looking with look ahead of depth one, right-looking with look ahead of depth two, dynamic look ahead. Problem size $n = 1536$, block size $nb = 64$. The graphs shows actual measurements (not a simulation). The white space in the middle represents a region clipped for clarity due to the length of the full chart.

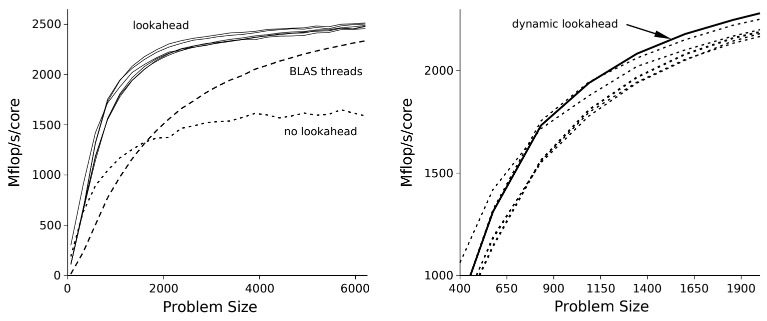


Fig. 5. Performance of different variants of LU factorization on two 1.8 GHz dual-core AMD Opteron^{TM} 265 processors (four cores total). The right graph shows magnification of the region between $n = 400$ and $n = 2000$.

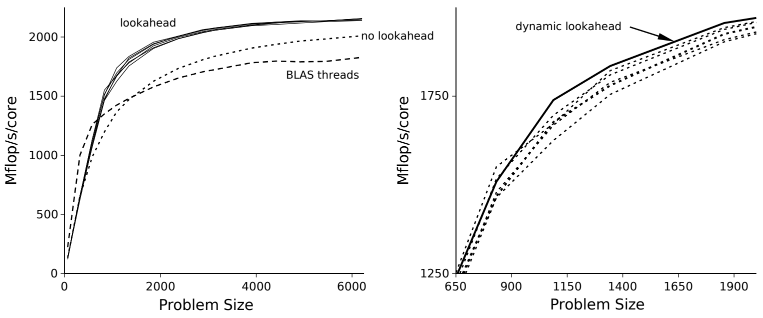


Fig. 6. Performance of different variants of Cholesky factorization. on two 1.8 GHz dual-core AMD Opteron^{TM} 265 processors (four cores total). The right graph shows magnification of the region between $n = 650$ and $n = 2000$.

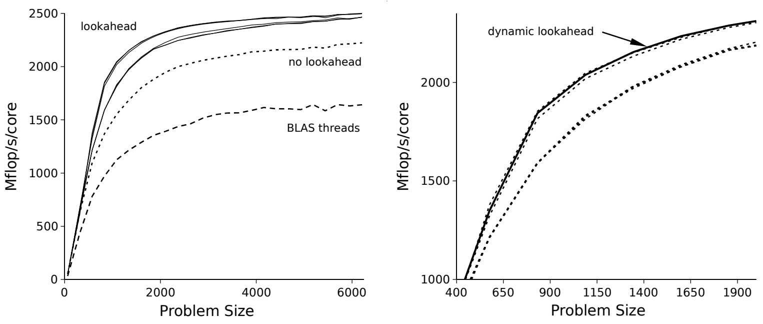


Fig. 7. Performance of different variants of QR factorization on two 1.8 GHz dual-core AMD Opteron^{TM} 265 processors (four cores total). The right graph shows magnification of the region between $n = 400$ and $n = 2000$.

Interesting observations can be made when comparing the explicitly parallelized code without look ahead with the BLAS-parallel code, which in principle should deliver similar performance. The main drawbacks of the explicitly parallel code stem from the 1D partitioning used here. The first disadvantage comes from the load imbalance introduced by such partitioning, which can be clearly visible on the right portions of graphs on Figure 4. The second disadvantage comes from the fact that in this case BLAS operations are invoked on narrow portions of the coefficient matrices, which is likely to cause performance loss, since typically bigger input translates to better BLAS performance. The better asymptotic performance of the BLAS-parallel code for LU factorization can be attributed to the two reasons mentioned above. It would be very desirable to closely investigate the efficiency of BLAS depending on the shape of the input arrays for different operation. This would, however, exceed the scope and space of this publication.

On the other hand, the main advantage of the explicitly parallel code is a reduced number of synchronization points in the algorithm. In particular in the update phase, each processor can apply all BLAS invocations to a column without synchronization with other processors. Also, applying a set of BLAS calls to a single column, potentially translates to data reuse between BLAS operations. At the same time, in the case of the BLAS-parallel code, each call to a BLAS operation requires forking and joining of threads, which may have a costly overhead and diminishes the potential for data reuse between BLAS operations. In particular, in the case of QR factorization, the update implemented by the DLARFB routine consists of the biggest number of BLAS calls plus a portion of sequential code, to which facts we attribute the worst performance of BLAS-parallel code compared to the explicitly parallel code without look ahead.

7 Conclusions

The application of the technique of look ahead was discussed in two algorithmic variants, with arbitrary (static) look ahead, where the algorithmic path is predetermined, and with dynamic look ahead, where the path is decided at runtime. The results collected on a modern multi-core system showed strong advantages of the idea of the look ahead in general.

For small number of cores the performance of dynamic look ahead can be matched by static look ahead. Nevertheless, dynamic look ahead eliminates the guesswork of setting the optimal depth of static look ahead. At the same time, im most cases it achieves the best performance.

Also, since it may be slightly counterintuitive, it should be pointed out that all of the techniques presented here deliver bit-wise identical results. Although the order of operations change from the processor perspective, it does not change from the perspective of each element of the input matrix.

8 Future Plans

The most important work envisioned in the future is application of the ideas presented here to the case of 2D partitioning, which will become necessary for load balancing with rapidly growing number of cores in multi-core processors.

Acknowledgements

The authors would like to thank Dr. Julien Langou for many interesting conversations and helpful comments about the work presented here.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J.W., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM (1992)

2. Barker, V.A., Blackford, L.S., Dongarra, J., Du Croz, J., Hammarling, S., Marinova, M., Wasniewski, J., Yalamov, P.: LAPACK95 Users' Guide. SIAM (1992)
3. Basic Linear Algebra Technical Forum: Basic Linear Algebra Technical Forum Standard (2001)
4. Agarwal, R.C., Gustavson, F.G.: A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In: Wright, M.H. (ed.) Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors, North-Holland, pp. 217–221. Elsevier, Amsterdam (1988)
5. Agarwal, R.C., Gustavson, F.G.: Vector and parallel algorithm for Cholesky factorization on IBM 3090. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (1989)
6. Strazdins, P.E.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Int. J. Parallel Distrib. Systems Networks 4(1), 26–35 (2001)
7. Gustavson, F.G., Karlsson, L., Kågström, B.: Three algorithms for Cholesky factorization on distributed memory using packed storage. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 550–559. Springer, Heidelberg (2007)
8. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM (1998)
9. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM (1997)
10. Bischof, C., van Loan, C.: The WY representation for products of householder matrices. J. Sci. Stat. Comput. 8, 2–13 (1987)
11. Schreiber, R., van Loan, C.: A storage-efficient WY representation for products of householder transformations. J. Sci. Stat. Comput. 10, 53–57 (1991)
12. Dongarra, J.J., Gustavson, F.G., Karp, A.: Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. SIAM Review 26(1), 91–112 (1984)
13. Menon, V., Pingali, K.: Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring. Int. J. Parallel Comput. 32(6), 501–523 (2004)
14. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency Computat.: Pract. Exper. 15, 803–820 (2003)
15. Petitet, A., Whaley, R.C., Dongarra, J.J., Cleary, A.: HPL - A portable implementation of the high-performance linpack benchmark for distributed-memory computers (2006), `http://www.netlib.org/benchmark/hpl/`
16. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical Report TR-02-55, Department of Computer Sciences, University of Texas at Austin (2002)

Specialized Spectral Division Algorithms for Generalized Eigenproblems Via the Inverse-Free Iteration[⋆]

Mercedes Marqués, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí

Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I,
12.071–Castellón, Spain
{mmarques,quintana,gquintan}@icc.uji.es

Abstract. We present two implementations of the inverse-free iteration for spectral division that reduce the computational cost of the traditional algorithm. One of the implementations is mainly composed of efficient BLAS-3 operations, and can be employed for spectral division of large-scale generalized eigenproblems on current computer architectures.

Keywords: Spectral division, generalized eigenproblem, matrix pencil, orthogonal transformations.

1 Introduction

Consider the matrix pencil $A - \lambda E$, where A, $E \in \mathbb{R}^{n \times n}$, and let $\Lambda(A, E)$ denote the set of generalized eigenvalues of the pencil, defined as

$$\Lambda(A, E) = \{z \in \mathbb{C} : \det(A - zE) = 0\}. \tag{1}$$

In the spectral division problem we are interested in finding a pair of (orthogonal) matrices U, $V \in \mathbb{R}^{n \times n}$ such that

$$U^T AV = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}, \quad U^T EV = \begin{bmatrix} E_{11} & E_{12} \\ 0 & E_{22} \end{bmatrix}, \tag{2}$$

and $\Lambda(A_{11}, E_{11})$, $\Lambda(A_{22}, E_{22})$ are disjoint sets containing certain parts of $\Lambda(A, E)$. The spectral division has important applications in matrix diagonalization, computation of deflating subspaces, and related problems such as linear-quadratic optimal control and model reduction of dynamic linear systems [1,10].

There are several methods to compute the decomposition in (2). The traditional approach proceeds by first obtaining the generalized (real) Schur form of the matrix pencil $A - \lambda E$ via the QZ algorithm [8], and then reordering the diagonal blocks of the matrices. This method, however, presents a major drawback

⋆ This research was partially supported by the DAAD programme Acciones Integradas HA2005-0081, the CICYT project TIN2005-09037-C02-02 and FEDER, and project No. P1B-2004-6 of the *Fundación Caixa-Castellón/Bancaixa and UJI.*

in that the QZ algorithm is composed of fine-grain computations which usually do not attain high performance on current computers. Recent advances in [4,5,6] aim at avoiding the poor performance of the QR and QZ algorithms.

A different alternative is to use spectral projectors related with the matrix sign and disk functions [3,7,9]. Iterative schemes resulting from this mainly consist of matrix-matrix products, solution of linear systems, and orthogonal factorizations. These operations deliver high performance on modern computer architectures.

In this paper we propose two variants of the inverse-free iteration for spectral division [7] that significatively reduce its computational cost. The modifications are based on a reordering of the orthogonal transformations that are applied during the first stage of the iteration. As a result, one of the matrices in the iteration is kept in (block) upper triangular form and the computational cost is reduced by 32%. The first variant employs Givens rotations and results in an algorithm composed of BLAS-1 computations. As in general this type of operations attains low performance on current computers, we also propose a block generalization of this scheme that employs efficient (BLAS-3) Householder transformations. Both variants rely solely on the use of orthogonal transformations; therefore, their numerical accuracy can be expected to be similar to that of the original algorithm. For a deep study on the practical numerical performance of the inverse-free iteration as a spectral division tool, see [2,12].

The rest of the paper is structured as follows. In Section 2 we review the inverse-free iteration for spectral division. Next, in Sections 3 and 4, we expose the two variants of the iterative scheme that reduce the (theoretical) computational cost. The experimental results in Section 5 show that one of these variants also reduces the execution time of the iteration. We conclude the paper with some final remarks in Section 6.

2 The Inverse-Free Iteration for Spectral Division

In [7] Malyshev proposed the following "inverse-free" iteration to separate the spectrum of a matrix pencil $A - \lambda E$:

$$\begin{array}{ll} A_0 \leftarrow A, & A_{k+1} \leftarrow Q_{12}^T A_k, \\ E_0 \leftarrow E, & E_{k+1} \leftarrow Q_{22}^T E_k, \quad k = 0, 1, \ldots, \end{array} \tag{3}$$

where, at each iteration, Q_{12} and Q_{22} are obtained from the QR factorization of the $2n \times n$ matrix M_k

$$M_k := \begin{bmatrix} E_k \\ -A_k \end{bmatrix} = Q_k \bar{R}_k = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} R_k \\ 0 \end{bmatrix}. \tag{4}$$

The convergence of this iteration is ultimately quadratic and the matrix pair at convergence allows the computation of a spectral projector inside the open unit disc [7]. Malyshev's proposal included a stopping criterion that required the computation of a matrix inverse and a subspace extraction technique which also involved this type of operation. Practical details on the inverse-free iteration can be found in [2], where the iteration and the subspace extraction scheme were modified to make them truly inverse-free. A new strategy for subspace extraction was formulated in [12] which provides both U and V in (2) from a single iteration, thus halving the cost of the algorithm proposed in [2].

The QR factorization in (4) is usually computed by applying n Householder reflectors, $H_1,\ H_2,\dots,H_n$, to M_k, so that $Q_k = H_1 \cdot H_2 \cdots H_n$. Let I_n and 0_n denote, respectively, the (square) identity matrix of order n and an $n \times n$ matrix with all its entries equal to zero. The appropriate part of Q_k can then be formed by "accumulating" the reflectors in reverse order as

$$H_1 \cdot H_2 \cdots H_n \begin{bmatrix} 0_n \\ I_n \end{bmatrix} = Q_k \begin{bmatrix} 0_n \\ I_n \end{bmatrix} = \begin{bmatrix} Q_{12} \\ Q_{22} \end{bmatrix}.$$

The computational cost in flops (floating-point arithmetic operations) of the iteration is reported in the column labeled as "Traditional" in Table 1. (The entries in other columns of the table will be introduced later.) In the table and also in later expressions for the computational costs we neglect lower-order terms. Compared with other spectral division tools, the cost of the inverse-free iteration is high. The computation of the generalized real Schur form via the QZ algorithm roughly requires $81n^3$ flops (plus the cost of the reordering procedure). Thus, in theory 6 inverse-free iterations are about as expensive as the QZ algorithm. The difference with the generalized Newton iteration for the sign function is more dramatic. By initially transforming the matrix pencil into $B_E - \lambda(U_E^T A V_E)$, where $B_E = U_E^T E V_E$ is (bi)diagonal, the cost of the sign function-based scheme is as low as $2n^3$ flops per iteration [11].

The inverse-free iteration requires storage for about 6 matrices of dimension $n \times n$ each. This represents a larger storage cost than the QZ algorithm, which employs only 4 matrices of the same dimension.

Table 1. Computational costs (in flops) of the algorithms for the inverse-free iteration

Step	Traditional	Givens-based	Blocked Householder
QR fact.	$3n^3 + n^3/3$	$3n^3$	$3n^3$
Accumulate Q_k	$6n^3$	$3n^3$	$3n^3$
$A_{k+1} \leftarrow Q_{12}^T A_k$	$2n^3$	n^3	n^3
$E_{k+1} \leftarrow Q_{22}^T E_k$	$2n^3$	$2n^3$	$2n^3$
Total	$13n^3 + n^3/3$	$9n^3$	$9n^3$

3 Reducing the Cost

In this section we illustrate how to reduce the cost of the inverse-free iteration. Consider the QR factorization $A = U_A R_A$, where U_A is orthogonal and R_A is upper triangular, and the matrix pencil $A_0 - \lambda E_0 = R_A - \lambda(U_A^T E)$, which shares its spectrum with $A - \lambda E$.

The key to reducing the cost of the iteration lies in exploiting and preserving the upper triangular structure of the sequence of matrices $\{A_k\}_{k=0,1,\ldots}$ during the iteration. We next show how to do so by using Givens rotations for a simple example involving 3×3 matrices. During the exposition, we will use $G_l^{i,j}$ to denote the l-th Givens rotation, which is applied at step l to rows $i-1$ and i in order to annihilate the (i,j) entry of $\left[E_k^T, \ -A_k^T\right]^T$.

Consider that before the computations in iteration k commence, the augmented matrix $M_k = [E_k^T, A_k^T]^T$ presents the following structure:

$$M_k = \left[\begin{array}{c} E_k \\ \hline -A_k \end{array}\right] = \left[\begin{array}{ccc} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \hline \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{array}\right],$$

where the symbol '$\times$' is used to denote a nonzero entry. Although using Givens rotations we can reduce M_k to upper triangular form in many different orders, we are particularly interested in Q_{12}^T being upper triangular, so that $A_{k+1} \leftarrow Q_{12}^T A_k$ remains upper triangular. Therefore, we first introduce a zero in the $(4,1)$ entry of M_k by applying a Givens rotation that involves the 3rd and 4th rows :

$$G_1^{4,1} \left[\begin{array}{ccc} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \hline \otimes & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{array}\right] = \left[\begin{array}{ccc} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \hline 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{array}\right];$$

here, the symbol '$\otimes$' is used to mark the entry that is annihilated by the application of $G_1^{4,1}$. We then continue by introducing zeros in the following order:

$$G_2^{3,1} \left[\begin{array}{ccc} \times & \times & \times \\ \times & \times & \times \\ \otimes & \times & \times \\ \hline 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{array}\right] \Rightarrow G_3^{2,1} \left[\begin{array}{ccc} \times & \times & \times \\ \otimes & \times & \times \\ 0 & \times & \times \\ \hline 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{array}\right] \Rightarrow G_4^{5,2} \left[\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ \hline 0 & \times & \times \\ 0 & \otimes & \times \\ 0 & 0 & \times \end{array}\right] \Rightarrow G_5^{4,2} \left[\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ \hline 0 & \otimes & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{array}\right] \Rightarrow$$

$$G_6^{3,2} \left[\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & \otimes & \times \\ \hline 0 & 0 & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{array}\right] \Rightarrow G_7^{6,3} \left[\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ \hline 0 & 0 & \times \\ 0 & 0 & \times \\ 0 & 0 & \otimes \end{array}\right] \Rightarrow G_8^{5,3} \left[\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ \hline 0 & 0 & \times \\ 0 & 0 & \otimes \\ 0 & 0 & 0 \end{array}\right] \Rightarrow G_9^{4,3} \left[\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ \hline 0 & 0 & \otimes \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right].$$

The application of a Givens rotation to a pair of vectors with q elements requires $6q$ flops. Therefore, computing the QR factorization of M_k as described above requires $\sum_{j=1}^{n}\sum_{i=1}^{n} 6(n-j) \approx 3n^3$ flops.

In order to construct the appropriate blocks of Q, we next proceed to apply the rotations in reverse order to the matrix $[0_n,\ I_n]^T$. If we now use the symbol '$\oplus$' to denote those entries which become nonzero by the application of the *last* Givens rotation, we have that the structure of this matrix is modified by the successive accumulation of Givens rotations as follows:

$$G_9^{4,3}\left[\begin{array}{ccc}0&0&0\\0&0&0\\0&0&0\\\hline \times&0&0\\0&\times&0\\0&0&\times\end{array}\right] \Rightarrow G_8^{5,3}\left[\begin{array}{ccc}0&0&0\\0&0&0\\\oplus&0&0\\\hline \times&0&0\\0&\times&0\\0&0&\times\end{array}\right] \Rightarrow G_7^{6,3}\left[\begin{array}{ccc}0&0&0\\0&0&0\\\times&0&0\\\hline \times&\oplus&0\\\oplus&\times&0\\0&0&\times\end{array}\right] \Rightarrow G_6^{3,2}\left[\begin{array}{ccc}0&0&0\\0&0&0\\\times&0&0\\\hline \times&\times&0\\\times&\times&\oplus\\\oplus&\oplus&\times\end{array}\right] \Rightarrow$$

$$G_5^{4,2}\left[\begin{array}{ccc}0&0&0\\\oplus&0&0\\\times&0&0\\\hline \times&\times&0\\\times&\times&\times\\\times&\times&\times\end{array}\right] \Rightarrow G_4^{5,2}\left[\begin{array}{ccc}0&0&0\\\times&0&0\\\times&\oplus&0\\\hline \times&\times&0\\\times&\times&\times\\\times&\times&\times\end{array}\right] \Rightarrow G_3^{2,1}\left[\begin{array}{ccc}0&0&0\\\times&0&0\\\times&\times&0\\\hline \times&\times&\oplus\\\times&\times&\times\\\times&\times&\times\end{array}\right] \Rightarrow G_2^{3,1}\left[\begin{array}{ccc}\oplus&0&0\\\times&0&0\\\times&\times&0\\\hline \times&\times&\times\\\times&\times&\times\\\times&\times&\times\end{array}\right] \Rightarrow$$

$$G_1^{4,1}\left[\begin{array}{ccc}\times&0&0\\\times&\oplus&0\\\times&\times&0\\\hline \times&\times&\times\\\times&\times&\times\\\times&\times&\times\end{array}\right] \Rightarrow \left[\begin{array}{ccc}\times&0&0\\\times&\times&0\\\times&\times&\oplus\\\hline \times&\times&\times\\\times&\times&\times\\\times&\times&\times\end{array}\right].$$

As the top 3×3 block of M_k contains Q_{12}, $A_{k+1} \leftarrow Q_{12}^T A_k$ remains upper triangular.

The cost of accumulating Q in this manner is $\sum_{i=1}^{n}\sum_{j=1}^{n} 6j \approx 3n^3$ flops and the total cost of one iteration is given in the column labeled as "Givens-based" in Table 1 and represents a rough reduction of 32% with respect to the traditional inverse-free iteration (via Householder reflectors). This cost could be further reduced by using fast Givens rotations.

In this approach the lower triangular part of A_k holds the Householder vectors from the initial factorization $A = Q_A R_A$. Given that a single rotation requires the storage of 2 parameters (the sine and cosine of the rotation), and the rotations need to be applied in reverse order, we also need storage space for $2n^2$ numbers. Half of this workspace can be obtained by reusing the entries of M_k which are annihilated, while storage for approximately $n^2/2$ numbers more is available in the strictly upper triangular part of Q_{12}^T. This represents an increase in $n^2/2$ numbers with respect to the traditional implementation of the inverse-free method.

Even with a reduction of 32% in the computational cost, we do not expect the Givens-based algorithm to outperform the traditional implementation as the application of Givens rotations is inherently a Level-1 BLAS operation, while the traditional algorithm employs much faster Level-3 BLAS kernels. Therefore, the effect of the decrease in theoretical cost is blurred by the use of a type of operation which attains a much lower performance on current architectures.

4 A High-Performance Algorithm

In this section we introduce a second algorithm that presents a theoretical cost similar to that of the Givens-based algorithm, but which performs most of its computations in terms of Level-3 BLAS operations.

As in the Givens-based algorithm, we start by computing the QR factorization $A = U_A R_A$, with U_A orthogonal and R_A upper triangular, so that the iteration is initialized with the matrix pencil $A_0 - \lambda E_0 = R_A - \lambda(U_A^T E)$. The key to this algorithm will be now to exploit and preserve a block upper triangular structure in the sequence of matrices $\{A_k\}_{k=0,1,\ldots}$ during the iteration.

For simplicity, we illustrate the algorithm for an example with $n = 3 \cdot b$. Thus, M_k consists of 6×3 blocks of dimension $b \times b$ each; that is,

$$M_k = \left[\begin{array}{c} E_k \\ \hline -A_k \end{array}\right] = \left[\begin{array}{ccc} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \\ \hline M_{41} & M_{42} & M_{43} \\ 0 & M_{52} & M_{53} \\ 0 & 0 & M_{63} \end{array}\right],$$

where all $M_{i,j} \in \mathbb{R}^{b \times b}$ and we will not assume any special structure for the diagonal blocks of A_k.

In our algorithm, we proceed to triangularize M_k by first computing the QR factorization of the $2b \times b$ matrix

$$\begin{bmatrix} M_{31} \\ M_{41} \end{bmatrix} = U_1^{4,1} \begin{bmatrix} R_{31} \\ 0 \end{bmatrix}.$$

By embedding the orthogonal matrix $U_1^{4,1}$ into the appropriate coordinates of a block diagonal matrix $Q_1 = \text{diag}\left(I_b, I_b, U_1^{4,1}, I_b, I_b\right)$, we have that

$$Q_1^T \left[\begin{array}{ccc} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \\ \hline M_{41} & M_{42} & M_{43} \\ 0 & M_{52} & M_{53} \\ 0 & 0 & M_{63} \end{array}\right] = \left[\begin{array}{ccc} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ R_{31} & \bar{M}_{32} & \bar{M}_{33} \\ \hline 0 & \bar{M}_{42} & \bar{M}_{43} \\ 0 & M_{52} & M_{53} \\ 0 & 0 & M_{63} \end{array}\right],$$

where R_{31} is upper triangular and all entries of M_{41} have been annihilated. Hereafter, we assume that M_k is overwritten with the application of the successive orthogonal factors that are computed to reduce M_k to upper triangular form. Thus,

$$\begin{bmatrix} M_{32} & M_{33} \\ M_{42} & M_{43} \end{bmatrix} \leftarrow U_1^{4,1} \begin{bmatrix} M_{32} & M_{33} \\ M_{42} & M_{43} \end{bmatrix}.$$

Let us now use $U_l^{i,j}$ to denote the orthogonal factor from the QR factorization of $\left[M_{i-1,j}^T,\ M_{i,j}^T\right]^T$. We next compute and apply a sequence of orthogonal matrices $U_2^{3,1}$, $U_3^{2,1}$, $U_4^{5,2}$, $U_5^{4,2}$, $U_6^{3,2}$, $U_7^{6,3}$, $U_8^{5,3}$, and $U_9^{4,3}$, so that zeros are introduced in M_k in the order specified in Figure 1.

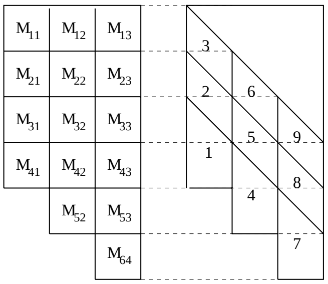


Fig. 1. Block partitioning of M_k (left) and the order in which blocks are annihilated by the application of orthogonal matrices (right). Each orthogonal matrix $U_l^{i,j}$ annihilates the elements in the lower triangle of $M_{i-1,j}$ and the elements in the upper triangle of $M_{i,j}$. For the diagonal blocks of A_k the elements in the lower triangle are annihilated together with those in the upper triangle.

If we assume $b \ll n$, the cost of computing the QR factorization using this blocked procedure is $\sum_{j=1}^{n/b}\sum_{i=1}^{n/b}(3b^3 + b^3/3 + 6(nb^2 - jb^3)) \approx 3n^3$ flops.

Consider Q_l to be the $6b \times 6b$ block diagonal matrix that embeds $U_l^{i,j}$ into its $(i-1)$st and ith diagonal blocks. By accumulating the orthogonal transformations required for the QR factorization of M_k in reverse order to $[0_n, I_n]^T$, we obtain a matrix of the form

$$Q_1Q_2\cdots Q_9\left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline I_b & 0 & 0 \\ 0 & I_b & 0 \\ 0 & 0 & I_b \end{array}\right] = \left[\begin{array}{ccc} V_{11} & 0 & 0 \\ V_{21} & V_{22} & 0 \\ V_{31} & V_{32} & V_{33} \\ \hline V_{41} & V_{42} & V_{43} \\ V_{51} & V_{52} & V_{53} \\ V_{61} & V_{62} & V_{63} \end{array}\right] = \begin{bmatrix} Q_{12} \\ Q_{22} \end{bmatrix}.$$

Thus, the product $A_{k+1} \leftarrow Q_{12}^T A_k$ involves two block upper triangular matrices with $b \times b$ diagonal blocks, Q_{12}^T and A_k, so that the result is also block upper triangular, with $b \times b$ diagonal blocks.

Provided $b \ll n$, the cost of the accumulation procedure is $\sum_{j=1}^{n/b}\sum_{i=1}^{n/b} 6(jb^3)$ $\approx 3n^3$ flops, and the (rough) overall cost of this algorithm is reported in the column labeled as "Blocked Householder" in Table 1. Thus, the cost is (asymptotically) as low as that of the Givens-based algorithm, but we now enable the use of Level-3 BLAS (at least) in the application of the orthogonal transformations.

The QR factorization of a $2b \times b$ matrix requires space to keep approximately $b^2/2 + b^2$ real numbers. As b^2 real numbers can be stored overwriting the entries that are annihilated, we still need a workspace of dimension $b^2/2$ for each tiny factorization, and a total space of dimension $\sum_{j=1}^{n/b}\sum_{i=1}^{n/b}(b^2/2) \approx n^2/2$. However, this space is available in the upper triangular part of the matrix that holds Q_{12} in the end.

In order to obtain a block triangular matrix Q_{12}, the matrix partitioning needs to be carefully done when n is not an exact multiple of the block size b. In particular, any row partitioning that is performed in the first column block of the matrix must be reproduced in the remaining blocks, starting from the diagonal. Thus, if the first column block is partitioned into blocks with row sizes $b_1, b_2, b_3, \ldots$, then any other column block of the matrix must be partitioned, starting at the diagonal block, into blocks with the same row sizes. For simplicity, we select $b_1 = b_2 = b_3 = \ldots = b$ except, possibly, for the last block column.

5 Experimental Results

All the experiments in this section were performed using IEEE double-precision (real) arithmetic on random matrices. The following routines are included in the evaluation of spectral division methods via the inverse-free iteration:

- DGGDFSP. The traditional iterative scheme as described in Section 2.
- DGGDFSG. The iteration with Givens rotations in order to introduce zeros in the matrix pair as described in Section 3.
- DGGDFSH. The iteration with Householder reflectors in order to introduce zeros in the matrix pair as described in Section 4.

For completion, we also include a routine DGGDFSX that performs the reduction of the matrix pair to generalized real Schur form via the QZ algorithm followed by a reordering of the generalized eigenvalues. The routine employs kernels available in LAPACK 3.0 for the QZ algorithm and related computations.

We report the performance of the spectral division codes on two architectures: the first platform, PENTIUM, consists of an Intel Pentium4 processor@3.2 GHz with 2048 KB of L2 cache; the second platform, ITANIUM, is composed of an Intel Itanium-2 processor@1.5 GHz with 256 KB/4 MB of L2/L3 cache. The BLAS implementation in Goto BLAS 1.0 and MKL 8.0 were used, respectively, on PENTIUM and ITANIUM.

Table 2 reports the execution time of all spectral division algorithms, including DGGDFSX, on random matrices with entries uniformly distributed in $[-1, 1]$. The column labeled as "#Iter." reports the number of iterations required by the algorithms based on the inverse-free iterative scheme. The results show that DGGDFSP and DGGDFSH are competitive with the QZ-based algorithm. However, no major conclusions should be extracted by comparing DGGDFSX with the remaining spectral division algorithms as the number of iterations of the inverse-free iterative schemes is quite dependent on the specific problem data. Besides, recent advances in the QR and QZ algorithms will surely change this ratio [4,5,6].

Figure 2 reports the MFLOPs (millions of flops per second) rate of the spectral division algorithms based on the inverse-free iterative scheme using a normalized

Table 2. Execution times of the spectral division algorithms on PENTIUM (top) and ITANIUM (bottom)

n	DGGDFSX	#Iter.	DGGDFSP	DGGDFSG	DGGDFSH
100	1.44E+00	13	7.13E−01	1.11E+00	7.21E−01
500	1.01E+02	15	6.56E+01	1.49E+02	5.57E+01
1000	6.80E+02	14	4.87E+02	1.11E+03	3.90E+02
1500	2.11E+03	16	1.80E+03	4.23E+03	1.41E+03
2000	4.92E+03	17	4.43E+03	1.06E+04	3.45E+03
2500	9.58E+03	16	8.20E+03	1.96E+04	6.31E+03

n	DGGDFSX	#Iter.	DGGDFSP	DGGDFSG	DGGDFSH
100	1.23E−01	13	5.24E−02	3.22E−01	5.34E−02
500	1.01E+01	15	5.98E+00	3.36E+01	4.70E+00
1000	6.70E+01	14	4.61E+01	2.58E+02	3.35E+01
1500	2.21E+02	16	1.76E+02	1.06E+03	1.25E+02
2000	4.97E+02	17	4.34E+02	2.70E+03	3.18E+02
2500	9.56E+02	16	8.13E+02	5.08E+03	5.88E+02

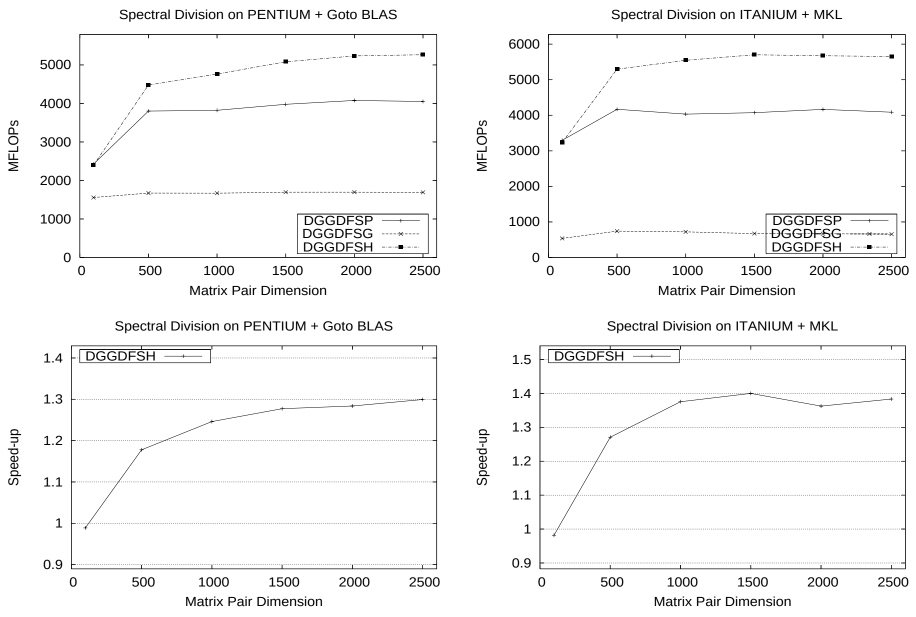


Fig. 2. Top: Normalized MFLOPs rate attained by the spectral division algorithms via the inverse-free iteration for spectral division on PENTIUM (left) and ITANIUM (right). Bottom: Speed-up attained by routine DGGDFSH over the traditional iterative scheme in DGGDFSP on PENTIUM (left) and ITANIUM (right).

flop count of $13.3n^3$ flops per iteration. The results show that, starting from $n = 500$, routine DGGDFSH consistently outperforms DGGDFSP by 20–40%, depending on the architecture and the problem size.

6 Concluding Remarks

We have described two variants of the inverse-free iteration for spectral division that reduce the theoretical cost of the traditional iterative scheme by 32%. The second variant carries out most of its computation in terms of Level-3 BLAS operation and clearly outperforms the traditional algorithm.

The reduction in the cost of the new algorithms narrows the gap between the computational costs of the inverse-free iteration and other, much cheaper, numerical tools for spectral division such as the sign function (via the Newton iteration) or the QZ algorithm.

Acknowledgments

We thank Prof. Xiaobai Sun for fruitful discussions that lead us to the joint derivation of the Givens-based algorithm.

References

1. Antoulas, A.C.: Approximation of Large-Scale Dynamical Systems. SIAM Publications, Philadelphia (2005)
2. Bai, Z., Demmel, J., Gu, M.: An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems. Numer. Math. 76(3), 279–308 (1997)
3. Benner, P.: Contributions to the Numerical Solution of Algebraic Riccati Equations and Related Eigenvalue Problems. Logos–Verlag, Germany (1997)
4. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part I: Maintaining well-focused shifts and level 3 performance. SIAM J. Matrix Anal. Appl. 23(4), 929–947 (2002)
5. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. SIAM J. Matrix Anal. Appl. 23(4), 948–973 (2002)
6. Kågström, B., Kressner, D.: Multishift variants of the QZ algorithm with aggressive early deflation. SIAM J. Matrix Anal. Appl. 29(1), 199–227 (2006)
7. Malyshev, A.N.: Parallel algorithm for solving some spectral problems of linear algebra. Linear Algebra Appl. 188/189, 489–520 (1993)
8. Moler, C.B., Stewart, G.W.: An algorithm for generalized matrix eigenvalue problems. SIAM J. Numer. Anal. 10, 241–256 (1973)
9. Roberts, J.D.: Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. Internat. J. Control 32, 677–687 (1980)
10. Sima, V.: Algorithms for Linear-Quadratic Optimization, Pure and Applied Mathematics, vol. 200. Marcel Dekker, Inc., New York (1996)
11. Sun, X., Quintana-Ortí, E.S.: The generalized Newton iteration for the matrix sign function. SIAM J. Sci. Comput. 24(2), 669–683 (2002)
12. Sun, X., Quintana-Ortí, E.S.: Spectral division methods for block generalized Schur decompositions. Math. Comp. 73, 1827–1847 (2004)

CFD Applications for High Performance Computing: Minisymposium Abstract

Siniša Krajnović

Chalmers, Sweden

Thanks to time-dependent simulations such as Large Eddy Simulations but also modelling more complex industrial flows, demands on HPC resources in CFD have increased dramatically in recent years. Typical time-dependent simulation uses large number of CPUs during long time (from weeks to months). Engineers in industry are simulating more complex flows and more care is put on details (such as representing all details in the geometry). All this would not be possible without cost efficient HPC resources in form of Linux clusters. This minisymposium will give an overview of HPC-intensive CFD applications covering computation of flows in water turbines, flows around cars, trains and airplanes and radiation heat transfer.

Some Experiences on the Accuracy and Parallel Performance of OpenFOAM for CFD in Water Turbines

Håkan Nilsson

Department of Applied Mechanics, Div. of Fluid Dynamics,
Chalmers, SE-412 96 Gothenburg, Sweden
`hani@chalmers.se`
`http://www.tfd.chalmers.se/~hani`

Abstract. 50% of the electric power in Sweden is generated by water power. Many of the power plants in Sweden are getting old and some major refurbishments are coming up. Due to the development of numerical methods and computer power the last decades Computational Fluid Dynamics (CFD) is to a large extent used as a design tool for this purpose. The general features of the flow in water turbines can be resolved with todays methods and computational power, but in order to study the flow in detail enormous HPC (High Performance Computing) facilities and new methods are required.

The present work presents the water turbine field with its HPC requirements, shows some state-of-the-art results from OpenFOAM CFD analysis, and presents a parallel performance analysis on a Linux cluster using an ordinary gigabit interconnect v.s. an Infiniband interconnect.

1 Water Turbines

Water turbines are designed to extract energy from the water flowing through the water turbine runner. The availabe power is determined by the difference in the elevation of the tail water and head water multiplied with the water mass flow and gravity. In reaction turbines the flow enters the runner with a swirl and the runner is designed to remove that swirl before the water enters the draft tube (see Figure 1). The draft tube is a diffuser which recovers the static pressure and leads the water towards the tail water.

1.1 HPC Requirements

Water turbines have complicated geometries where the flow in different parts of the turbine influences the flow in other parts of the turbine. To be able to set valid boundary conditions and to model the flow correctly it would thus be ideal to include all the geometry from head water to tail water. The Reynolds numbers in water turbine applications are always high, so the resolution of the computational mesh must be fine where large gradients in the flow occur. These

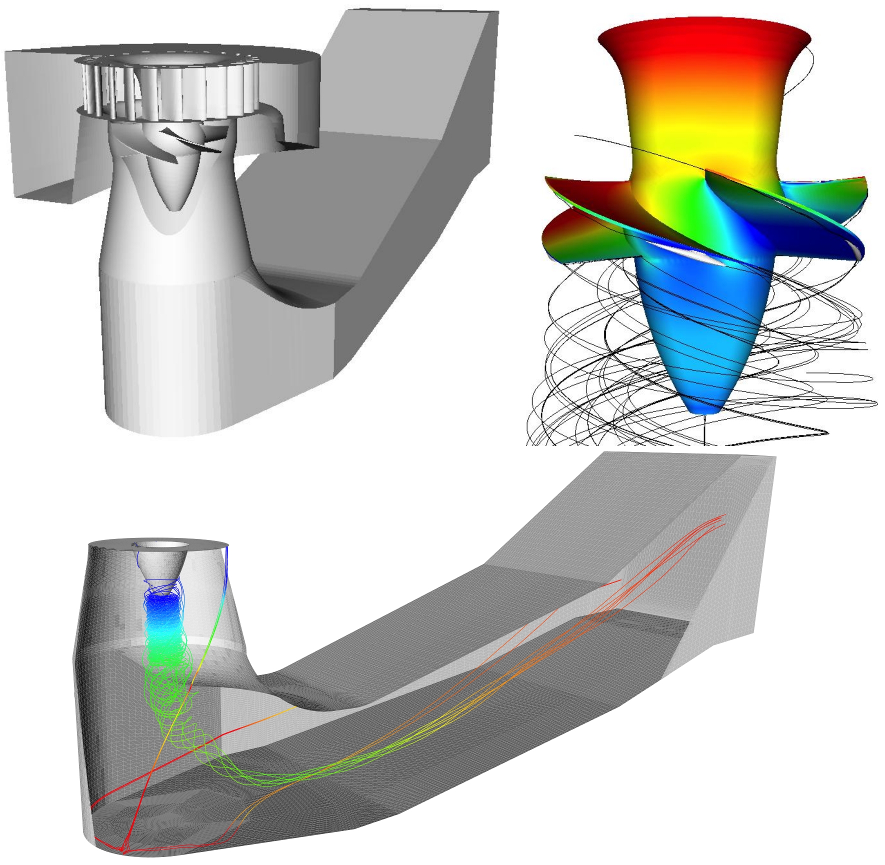


Fig. 1. CAD model of the Hölleforsen Kaplan turbine model and visualizations of the flow in the runner and draft tube

requirements together yields an enourmously large mesh. In addition to this there are both stationary and rotating parts in water turbines, which require rotor-stator interaction and unsteady computations. The thin wakes after stayvanes, guide vanes and runner blades affect the details of the flow and should also be resolved. Close to the runner blades cavitation often occurs, and the numerical methods to resolve those effects require even more cells and shorter time steps. There is probably no limit on the HPC requirements for flow in water turbines if all the important flow features should be appropriately predicted using CFD.

2 The OpenFOAM CFD Tool

The newly released OpenFOAM (Open Field Operation and Manipulation, www.openfoam.org) tool has been used in the present study. OpenFOAM is an Open-Source object oriented C++ tool for solving various partial differential equations

(PDEs). It includes preprocessing (grid generator, converters, manipulators, case setup), postprocessing (using OpenSource Paraview) and many specialized CFD solvers are implemented. Some of the more specialized features that are included in OpenFOAM, and which are important to the flow in water turbines are: sliding grid, moving meshes, two-phase flow (Langrange, VOF, Euler-Euler) and fluid-structure interaction. OpenFOAM runs in parallel using automatic/manual domain decomposition. In addition to the source code, OpenFOAM gives access to an international community of OpenFOAM researchers through the discussion board at the OpenFOAM home page.

3 The Studied Cases

In the present work two parts of the Hölleforsen turbine model have been modeled, the runner and the draft tube (see Figure 1). The two parts have been modeled separately, but the aim is to simultaneously model all parts including the spiral casing, the runner and the draft tube. The present runner computations have an inlet boundary condition from a previous computation of the flow through the guide vanes. Steady computations have been made for both a periodic part of the runner (1/5) and the whole runner. The present draft tube computations have inlet boundary conditions from detailed Laser-Doppler Anemometry (LDA) measurements. Both steady [1] and unsteady computations [2] have been made for the draft tube. In all cases the standard $k-\varepsilon$ turbulence model is used. The sizes of the meshes used are, 450 000 cells for the periodic runner computation, 2 240 000 cells for the full runner computation, and 1 000 000 cells for the draft tube computations. These are all wall-function grids that do not resolve the boundary layers in detail.

4 Results and Validation

The results from the present computations have been presented and validated in detail in [2]. Some of these results and validations against measurements and CFX-5 results are presented here also. The locations of the available measurements [3] of the flow in the present cases are shown in Figure 2, and the names of the measurement sections are used when presenting the results.

The dimensionless coefficients used in the validation are the velocity coefficient (C_v), the pressure coefficient (C_p), the mean pressure recovery (C_{prm}) and the energy loss coefficient (ζ).

The velocity coefficients, C_v, are the velocities normalized by Q/A_i, where $Q = 0.522m^3/s$ is the volume flow and A_i is the area of each cross-section ($A_{Ia} = 0.15m^2, A_{Ib} = 0.23m^2$). The tangential velocity component is positive when the flow is co-rotating with the runner, and the axial velocity component is positive in the main flow direction.

The pressure coefficient, C_p, is the local static pressure divided by the dynamic pressure at section Ia, $P_{dyn,Ia} = \rho Q^2/(2A_{Ia}^2)$.

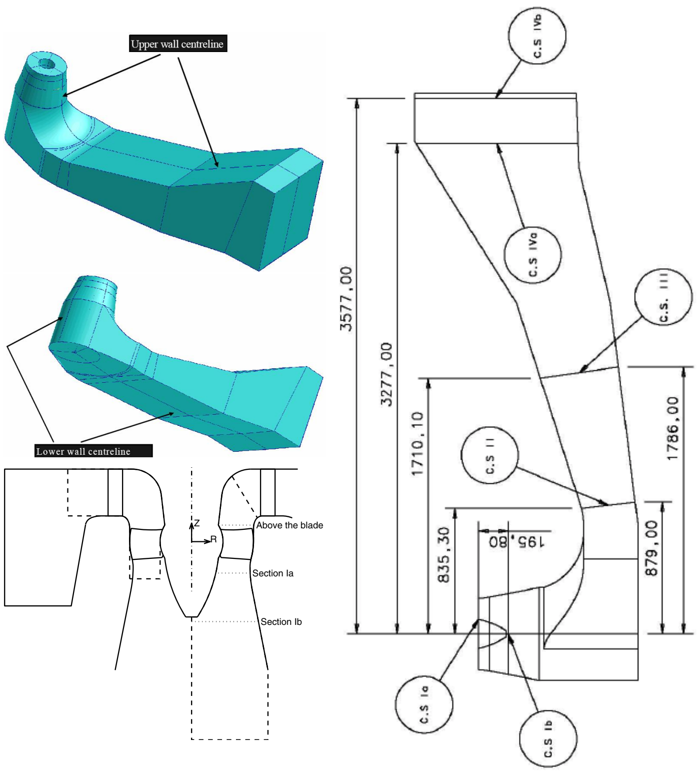


Fig. 2. Geometry and measurement sections. (Pictures from the Turbine-99 workshop guidelines and from previous work).

The mean pressure recovery is defined as

$$C_{prm} = \frac{\frac{1}{A_{CS}} \int\int_{A_{CS}} PdA - \frac{1}{A_{Ia}} \int\int_{A_{Ia}} PdA}{P_{dyn,Ia}}, \tag{1}$$

where $P_{dyn,Ia} = \rho Q^2/(2A_{Ia}^2)$ is the dynamic pressure at cross-section Ia.

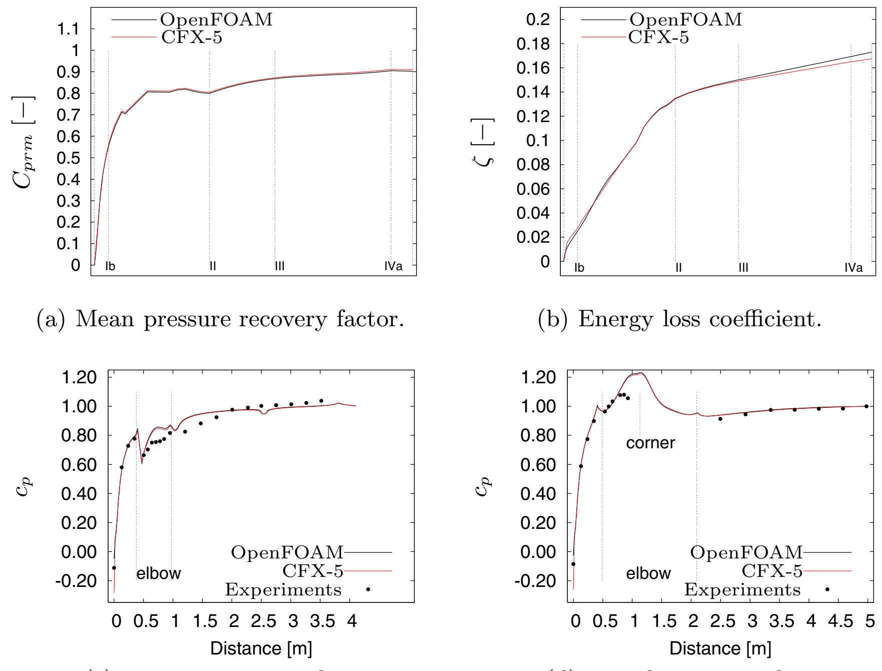
The energy loss coefficient is defined as

$$\zeta = \frac{\int\int_{A_{Ia}} \left(P + \rho\frac{|\boldsymbol{U}|^2}{2}\right) \boldsymbol{U} \cdot \hat{n} dA - \int\int_{A_{CS}} \left(P + \rho\frac{|\boldsymbol{U}|^2}{2}\right) \boldsymbol{U} \cdot \hat{n} dA}{\left|\int\int_{A_{Ia}} \rho\frac{|\boldsymbol{U}|^2}{2} \boldsymbol{U} \cdot \hat{n} dA\right|}. \tag{2}$$

4.1 Integral Quantities

The integral quantities described in eqs. (1) and (2) are here used to validate the steady draft tube computation. Figure 3 shows through-flow developments of the mean pressure recovery factor and energy loss coefficient. The cross-section integrals are computed for a number of cross-sections perpendicular to the flow, while keeping the reference integral at section Ia constant [1] (see Figure 2). The figure shows that the integrated result from the present computation is almost identical to the result from a CFX-5 computation [4].

The pressure coefficient at the upper and lower center lines (see Figure 2) are compared with both the experimental data and the CFX-5 computation in Figure 3. The two computations are almost identical, and they are close to the experimental results.



(a) Mean pressure recovery factor.

(b) Energy loss coefficient.

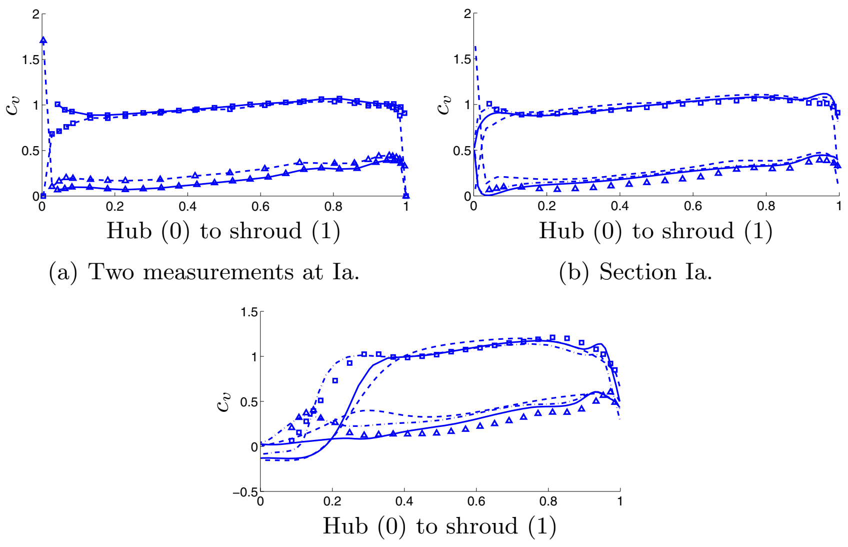
(c) c_p at upper centerline

(d) c_p at lower centerline

Fig. 3. Computed integral quantities between section Ia and the end of the draft tube (a & b), and pressure coefficient distributions along the upper and lower centerlines (c & d). The only difference that can be seen is in (b), where OpenFOAM gives a slightly higher value at the end of the draft tube.

4.2 Velocity Distributions at Ia and Ib

Figure 4 compares the velocity coefficient distributions at cross-sections Ia and Ib (see Figure 2) with the experimental results. Figure 4(a) shows two different results from the measurements at cross-section Ia. This gives an indication of the accuracy of the measurements. For the draft tube computations it was recommended (Turbine-99) to use the results corresponding to the dashed lines as inlet boundary condition. Thus the dashed lines in Figures 4(a) and (b) are



(a) Two measurements at Ia.

(b) Section Ia.

(c) Section Ib.

Fig. 4. Velocity coefficient distributions. Markers: Measurements of axial (squares) and tangential (triangles) velocity components. In (a) two different measurements are shown, and the markers of each mesasurement are connected with solid lines and dashed lines respectively. In (b) and (c): Dashed lines: steady draft tube, Solid lines: runner without hub clearance, Dash-dotted lines: runner with hub clearance.

equivalent, and the measurement markers in Figures 4(b) and (c) correspond to the solid line measurements in Figure 4(a). The choice of those measurements as inlet boundary condition however doesn't seem to give the correct behaviour of the flow in the boundary layer at the hub. The solid lines in Figure 4(a) show that the axial velocity has a local maximum close to the hub, which is important for the development of the flow down to section Ib and throughout the draft tube.

Figures 4(b) and (c) compare the computational results with the experimental results. The dashed line corresponds to the steady draft tube computation while the dash-dotted and solid lines correspond to the runner computations with and without a runner blade hub clearance respectively. It is shown that the runner blade hub clearance produces an increase in the axial velocity close to the hub. This effect is very important for how well the separation at the hub is modelled. In Figure 4(c) the runner computation with the runner blade hub clearance resembles the experimental results much better than the other results, which is an effect of the small increase in axial velocity at section Ia. The velocity distribution at section Ib is very important for the flow development in the draft tube, and it is thus likely that the boundary condition for the draft tube computation is inadequate.

For both runner computations the effect of the tip clearance flow can be seen as a local maximum in the axial and tangential velocity profiles close to the shroud.

5 Parallel Performance

A parallel performance test has been made using the draft tube case described above, with about 10^6 cells. The decompositions of the domain into 2, 4, 8 and 16 subdomains were made using the automatic load-balanced decomposition (Metis) in OpenFOAM. The Linux cluster was a 4 node Dual socket AMD Opteron 280 (2.4 GHz, dual core) with 4GB DDR400 RAM, i.e. 4 cores (CPUs) per node and a total of 16 cores (CPUs). Two different interconnects were tested, a Gigabit Ethernet through an HP ProCurve 2824 Switch, and an Infiniband (PCI-X) through a Silverstorm 9024 Switch. The SuSE Linux Enterprise Server, Service pack 3 operating system was used. The analysis has been made together with Peter Rundberg at Gridcore (www.gridcore.se).

The wall clock times were measured for three iterations, and the tests were repeated several times. No I/O was included in the performance test. The wall clock times are presented in Table 1. The table also shows the speed-up normalized by the single CPU run for each configuration, and the speed-up when using the Infiniband interconnect instead of the Gigabit interconnect (based on the speed-up columns in the table).

Table 1. Parallel performance using 1Gbit Ethernet (ETH) and Infiniband (IBA) interconnects. Packed vs. spread CPU distribution (the distribution of the processes on the nodes)

# CPU	# nodes	ETH (s)	IBA (s)	ETH (speed-up)	IBA (speed-up)	$\frac{IBA\ (speed-up)}{ETH\ (speed-up)}$ (based on speed-up)
1	1	165	163	1.0	1.0	1.0
2	1	86	78	1.9	2.1	1.1
2	2	85	81	1.9	2.0	1.0
4	1	76	72	2.2	2.3	1.0
4	2	64	62	2.6	2.6	1.0
4	4	53	56	3.1	2.9	0.9
8	2	43	41	3.8	4.0	1.0
8	4	41	35	4.0	4.7	1.2
16	4	23	20	7.2	8.2	1.1

Table 1 shows that the execution time of the computations decreases as more CPUs are used. When going from one to two CPUs the parallel speed-up is linear, but when using more CPUs the parallel efficiency is significantly reduced. When using 16 CPUs the parallel efficiency is 45% for the gigabit interconnect and 51% for the Infiniband interconnect. Table 1 shows no significant difference between the parallel runs using the different interconnects except when the computations are distributed on as many nodes as possible. This can most clearly be seen when using 8 CPUs on 2 and 4 nodes, where there is a 20% gain in using the Infiniband interconnect when distributing the computation on 4 nodes instead of

2 nodes. When distributing the processes on different nodes the computational requirement for each node is reduced and the requirement of the communication speed between the nodes is increased. When the computations are packed as much as possible on each node, the effect of the interconnect between the nodes is reduced. This effect could unfortunately not be further investigated for more nodes since there was only four nodes available during the present work.

Here follows some additional remarks on the results shown in Table 1. There is a small difference between the single CPU runs. Such differences are expected. Many things can influence computational speeds to this order of magnitude ($\sim$1%). There is also a difference between the runs with 2 CPUs on 1 node. The interconnect should not be important in this case. It has however been observed that sometimes when running two processes on the same node they end up at the same socket, which influences the computational speed to this order of magnitude ($\sim$10%). Later versions of the Linux kernel should have fixed this problem. An effect related to this can be seen in Table 1, where the speed-up is increased if the computations are spread over as many nodes as possible. Table 1 suggests that IBA has a worse tendency than ETH when distributing the 4 CPU case on different numbers of nodes. The more nodes involved, the more important is the interconnect. The table however suggests that the Gigabit interconnect is the best in the 4 CPU case.

6 Conclusions

This work has presented the water turbine field and its huge HPC requirements when the flow in a complete water turbine is to be predicted in detail. With complex geometries, high Reynolds numbers, turbulence, cavitation, and interaction between rotating and stationary parts, the computer power of today is still far from sufficient. By modeling parts of the physics and focusing the computations on parts of the water turbine it is however feasible to get reasonably accurate and useful solutions.

The OpenFOAM OpenSource tool was introduced and applied to CFD problems in the water turbine field. The results proved to be of the same accuracy as those of commercial CFD solvers. OpenFOAM is a true competitor to both commercial tools and in-house research codes. The major benefits of the OpenFOAM tool are that it is free of charge, and that the complete source code is available.

The parallel performance of OpenFOAM was analysed on a Linux cluster using an ordinary gigabit Ethernet interconnect and an Infiniband interconnect. The problem used for the analysis was the flow in a water turbine draft tube, and the grid consisted of approximately 10^6 control volumes. The Infiniband interconnect did not significantly improve the speed-up for this problem. The largest improvement is observed when the problem is distributed on as many CPUs as possible. When distributing the problem on 16 CPUs the Infiniband interconnnect has a $\sim$10% speed-up compared with the corresponding gigabit Ethernet interconnect computation. The parallel performance of OpenFOAM for the present problem was modest, with a parallel efficiency of approximately 50% when run on 16 CPUs.

References

1. Nilsson, H., Page, M.: OpenFOAM Simulation of the flow in the Hölleforsen draft tube model. In: Proceedings of Turbine-99 III (2005)
2. Nilsson, H.: Evaluation of OpenFOAM for CFD of turbulent flow in water turbines. In: Proceedings of the 23rd IAHR Symposium in Yokohama (2006)
3. Andersson, U.: Turbine 99 - Experiments on draft tube flow (test case T). In: Proceedings of Turbine 99 - Workshop on Draft Tube Flow (2000) ISSN: 1402 - 1536
4. Page, M., Giroux, A.-M., Nicolle, J.: Steady and unsteady computations of Turbine99 draft tube. In: Proceedings of Turbine-99 III (2005)

HPC Environments – Visualization and Parallelization Tools: Minisymposium Abstract

Anne C. Elster[1] and Otto J. Anshus[2]

[1] Norwegian University of Science and Technology (NTNU), Norway
[2] Department of Computer Science, University of Tromsø, Norway

As high-performance computing systems become more complex crunching on and generating terabytes or petabytes of data, it becomes more and more important to be able to visualize the results and utilize state-of-the-art tools to develop and optimize applications on modern HPC systems, including large parallel clusters and grids.

This minisymposium hightlights some of the recent developments in this area. The symposium is dived into two parts where the first part focuses on visualization environments and the second on HPC tools.

Trusting Floating Point Benchmarks – Are Your Benchmarks Really Data Independent?

John Markus Bjørndalen and Otto J. Anshus

Department of Computer Science, University of Tromsø, NO-9037 Tromsø, Norway
{johnm, otto}@cs.uit.no

Abstract. Benchmarks are important tools for studying increasingly complex hardware architectures and software systems.

Two seemingly common assumptions are that the execution time of floating point operations do not change much with different input values, and that the execution time of a benchmark does not vary much if the input and computed values do not influence any branches. These assumption do not always hold.

There is significant overhead in handling *denormalized* floating point values (a representation automatically used by the CPU to represent values close to zero) on-chip on modern Intel hardware, even if the program can continue uninterrupted. We have observed that even a small fraction of denormal numbers in a textbook benchmark significantly increases the execution time of the benchmark, leading to the wrong conclusions about the relative efficiency of different hardware architectures and about scalability problems of a cluster benchmark.

1 Introduction

Benchmarks are important tools for studying increasingly complex systems[1]. Understanding the characteristics of a benchmark is important to avoid drawing the wrong conclusions from the results.

Due to changing hardware architectures, however, assumptions that were made when the benchmark was created may no longer hold when moving the benchmark to new architectures or even new implementations of the same architecture.

We report on a problem where the execution time of a textbook benchmark varied significantly depending on the computed floating point values, even when the computed values were not used to influence any branches. We also show how the performance characteristics of the benchmark could be misinterpreted, wrongly indicating a scalability problem in the application or communication subsystem, and how the problem could lead to the wrong conclusions about the relative performance of a PowerPC and an Intel P4 computer.

[1] Examples are increasing difference in latency between different levels of caches and memory, and the introduction of Simultaneous Multithreading and Single-Chip Multiprocessors.

This paper continues as follows: section 2 describes normalized and denormalized representation of floating point, section 3 describes our benchmark and section 4 our experiment result. Section 5 discusses the results, lessons learned and provides some guidelines. Finally we conclude in section 6.

2 Floating Point Representation, Normalized and Denormalized Form

The IEEE 754 standard for floating point[1] specify the representation and operations for single precision (32-bit) and double precision (64-bit) floating point values. This provides a finite amount of space to represent real numbers that mathematically have infinite precision. Thus, computers only provide an approximation of floating point numbers, and a limited range of values that can be represented. For single precision floating point, the range is about $1.18 \cdot 10^{-38}$ to $3.40 \cdot 10^{38}$ positive and negative.

One of the goals for the designers of the IEEE floating point standard was to balance the resolution and magnitude of numbers that could be represented. In addition, a problem at the time was handling applications that *underflowed*, which is where the result of a floating point operation is too close to zero to represent using the normal representation. In the 70s, it was common for such results to silently be converted to 0, making it difficult for the programmer to detect and handle underflow[2].

As a result, a provision for *gradual underflow* was added to the standard, which adds a second representation for small floating numbers close to zero. Whenever the result of a computation is too small to represent using normalized form, the processor automatically converts the number to a *denormal* form and signals the user by setting a status register flag and optionally trapping to a floating point exception handler. This allows the programmer to catch the underflow, and use the denormalized values as results for further computations that ideally would bring the values back into normalized floating point ranges.

The processor automatically switches between the two representations, so computer users and programmers rarely see any side effects of this, and operations for normal values are also valid for any combination of normal and denormal values. Many programmers may be unaware that their programs are using denormalized values.

Goldberg gives a detailed description of denormalized numbers and many other floating point issues in [3].

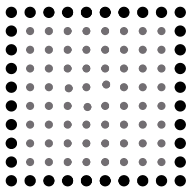
Instructions involving denormalized numbers trigger Floating Point Exceptions. Intel processors handle underflows using on-chip logic such that reasonable results are produced and the execution may proceed without interruption [4]. The programmer can unmask a control flag in the CPU to enable a software exception handler to be called.

The Intel documentation [5], and documents such as Goldbergs [3], warn about significant overhead when handling exceptions in software, but programmers may not expect the overhead to be very high when the floating point exceptions are handled in hardware and essentially ignored by the programmer.

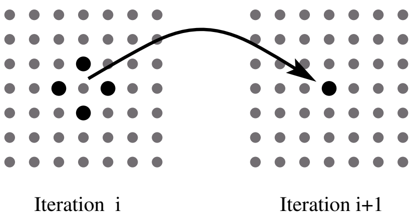
3 Experiments

3.1 Jacobi Iteration Benchmark

We first found the problem described in this paper in an implementation of a well known algorithm for solving partial differential equations, Successive Over-relaxation (SOR) using a Red-Black scheme. To simplify the analysis in this paper, we use the Jacobi Iteration method, which has similar behavior to the SOR benchmark.



(a) Square sheet of metal represented as a matrix of points.



(b) For each point, a new value is computed based on the average of the cells left, right, above, and below from the previous iteration (colored black in the figure).

Fig. 1. Heat distribution problem with the Jacobi iterative method. In a square sheet of metal, the temperature along the edges is known (black points), and the temperature for internal points (gray points) are computed based on the known values for the edges.

Figure 1 shows an example problem solved by Jacobi, where a sheet of metal is represented as a matrix of points. The temperature along the edges is known, while the temperature for the internal points is computed from the known values of the boundary (black) points.

The Jacobi iteration method gradually refines an approximation by iterating over the array a number of times, computing each internal point by averaging the neighbors (see Figure 1(b)). Two arrays are used: one is the source array, and the other is the target array where the results are stored. When the next iteration starts, the roles of the arrays are swapped (the target is now the source and vice versa).

A solution is found when the system stabilizes. The algorithm terminates when the difference between the computed values in two iterations of the algorithm is considered small enough to represent a solution.

The implementation used in this paper is a simplified version of Jacobi, running for a fixed number of iterations instead of checking whether the systems has stabilized. This simplifies benchmarking and analysis.

3.2 Methodology

The experiments were run on the following machines:

– Dell Precision Workstation 370, 3.2 GHz Intel Pentium 4 (Prescott) EMT64, 2GB RAM, running Rocks Linux 3.3.0 with Linux kernel 2.4.21 in 64-bit mode.
– A cluster of 40 Dell Precision Workstation 370, configured as above, interconnected using gigabit Ethernet over a 48-port HP Procurve 2848 switch.
– Apple PowerMac G5, Dual 2.5 GHz PowerPC G5, 4GB DDR SDRAM, 512KB L2 cache per CPU, 1.25GHz bus speed.
– AMD Athlon 64 X2 4400+ 2.2GHz, 2GB DDR SDRAM, running Ubuntu Linux 5.10 with kernel 2.6.12-10-k7-smp.

The results the Dell Precision machine have been verified using other Intel-based computers showing similar overheads.

The benchmark was compiled with GCC version 3.3.5 with the flags "-Wall -O3" on the Intel architecture. On the PowerMac, GCC 4.0.0 was used with the same flags. Intels C-compiler, version 8.0, was also used with the same flags to verify the results with a different compiler.

The execution time is measured by reading the wall-clock time before and after the call to the Jacobi solver (using *gettimeofday*), and does not include startup overhead such as application loading, memory allocation and array initialization. All benchmarks are executed with floating point exceptions masked, so no software handlers are triggered.

The benchmarks were run with the following sets of input values:

– One that produce denormal numbers: the border points are set to 10.000, and the interior points are set to 0.
– One that does not produce denormal numbers: the border points are set to 10.000 and the interior points are set to 1.
– One of the benchmarks use an additional input set with *all* values in the matrix initialized to a denormal number: 10^{-39}.

In all cases, we run Jacobi for 1500 iterations with a 750x750 matrix.

4 Results

4.1 Impact of Denormal Numbers on Intel Prescott

The experiment was run on a Dell Precision WS 370. The benchmark was instrumented using the Pentium time stamp counters to measure the execution time of each iteration of Jacobi (0-1500), and of the execution time of each row in each iteration.

Experiments Without Denormalized Numbers. Figure 2 shows the execution time of each iteration of Jacobi when no denormalized numbers occur in the computation. Apart from a spike, the execution time is very stable, as one would expect since the branches in the algorithm are not influenced by the computed values.

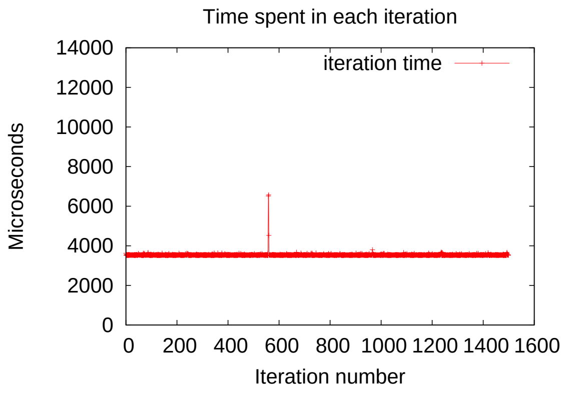


Fig. 2. Computation time for each iteration of Jacobi when no denormalized numbers occur

Experiments with Denormalized Numbers. Figure 3 shows snapshots of the matrix for four of the iterations of the algorithm. In the beginning, there is only a narrow band of non-zero (blue) values. For each iteration, non-zero values gradually spread further inwards in the matrix, but they do so by being averaged with the zero-values in the array.

After a number of iterations, the values that make up the front edge of the non-zero values are too small to represent using normalized floating point values, and the CPU automatically switches over to denormalized form (shown as red in the snapshots). The band of denormalized numbers grow in width and sweep towards the center until they eventually disappear.

Figure 4 shows the execution time per iteration of Jacobi when denormalized numbers occur in the computation (left). The figure also shows the number of denormalized numbers in the result array for each iteration (right).

The number of denormalized numbers correspond directly with the execution time per iteration. When denormalized numbers occur, the execution time rapidly increases, and after a while gradually decreases while the number of denormalized numbers decrease. After a while, the last denormalized number disappears and the execution time is back to normal. The jagged pattern on the graph showing the number of denormalized numbers is also reflected in the jagged pattern of the graph showing the execution time.

Figure 5 shows the execution time of each row of the matrix for the four iterations that are shown in figure 3. The execution time for each row is clearly correlated to the number of denormalized values in that row.

For the iteration with the highest number of denormalized results, iteration 338, about 3.88% of the internal numbers computed are in denormalized form. This is also the iteration that takes the longest to execute, taking about 3.83 times longer than the average of iterations without denormalized numbers.

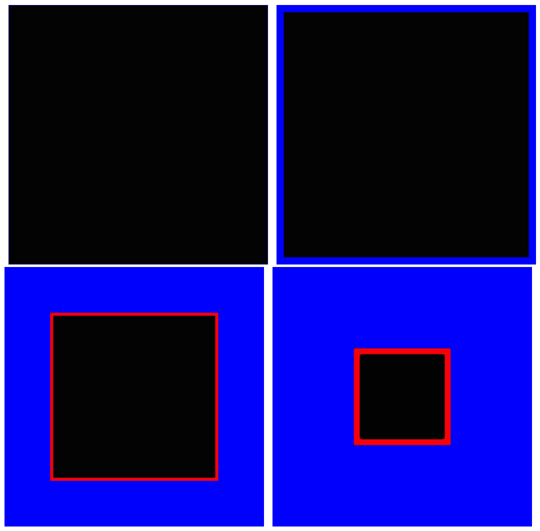


Fig. 3. Graphical representation of the result matrices for four iterations of the JACOBI benchmark (iterations 0, 20, 200 and 600). Black represents 0-values, blue represents non-zero values, and red represents nonzero values in denormal form.

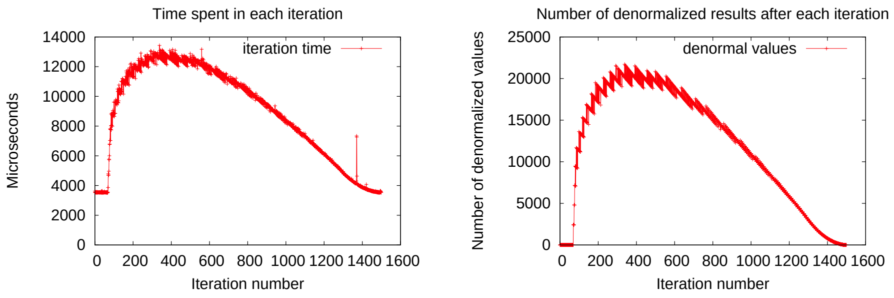


Fig. 4. Left: execution time of each of 1500 iterations. Right: number of floating point values in denormalized form stored in the result matrix on each iteration. Stored values are counted rather than the number of operations due to the lack of performance counters to count the number of denormal operations (see section 5.2).

4.2 Impact on Comparisons of PowerPC, AMD and Intel P4 Prescott

Table 1 shows that the Intel processor has higher overheads when handling denormalized numbers than the PowerPC machine, with a factor 70 between *normal* and *all denormal* compared to a factor 3.65 on the PowerPC machine. This may influence comparisons of architectures. As an example, consider the results of the PowerPC *denormal* (5.40) and Intel *denormal* (13.44), which indicate that

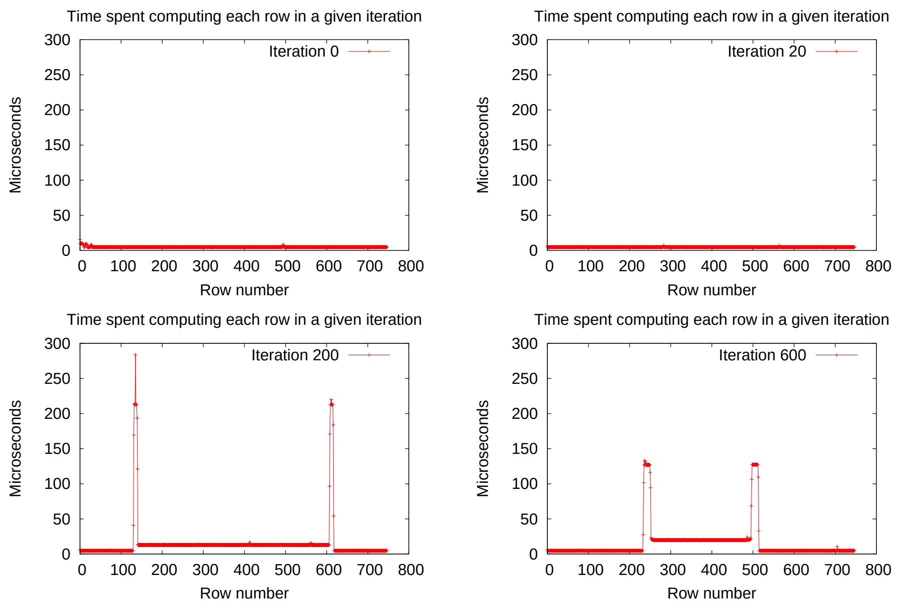


Fig. 5. Execution time for each row in the matrix for four iterations of the algorithm (iterations 0, 20, 200 and 600). The effect of denormalized numbers is clearly visible as rows with denormalized numbers have a higher execution time than rows without denormalized numbers. The execution time depends on the number of denormalized numbers (see lower left, which corresponds to the lower left picture in graph 3).

the PowerPC architecture is more efficient for the Jacobi algorithm. With the dataset with normalized numbers, the difference is much smaller (5.04 vs. 5.33).

The problem is also significant for the comparison of AMD vs. Intel, where for the *denormal* case, the AMD processor is faster than the Intel processor, while for the *normal* case, the Intel processor is faster.

This indicates that when moving benchmarks across architectures, corner cases that previously did not influence the benchmarks significantly may start to have an impact, and the benchmarks may not work as originally planned even if the computed results are identical.

4.3 Parallel Jacobi on a Cluster

Figure 6 shows the results of running parallelized implementation of Jacobi using LAM-MPI 7.1.1 [6] using 1 to 8 nodes of the cluster. The implementation divides the matrix into N bands, and each process exchange the edges of its band with the neighbor processes bands.

The graph shows that the dataset with denormalized numbers influence the scalability of the application, resulting in a speedup for 8 processes of 4.05 compared to a speedup of 6.19 when the computation has no denormalized numbers.

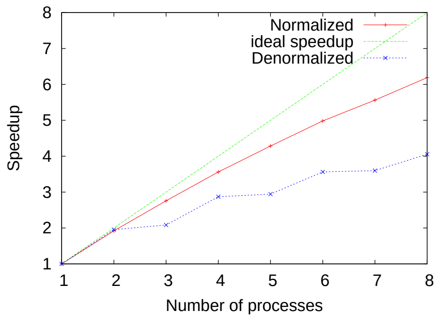


Fig. 6. Speedup of Jacobi on a cluster using 1-8 nodes

Table 1. Minimum and maximum execution times of 5 runs of Jacobi

Machine + Dataset	min	max
Intel, *normal*	5.33	5.34
Intel, *denormal*	13.44	13.47
Intel, *all denormal*	371.73	373.05
AMD, *normal*	6.65	6.71
AMD, *denormal*	8.46	8.61
AMD, *all denormal*	88.38	88.99
PPC, *normal*	5.04	5.05
PPC, *denormal*	5.40	5.41
PPC, *all denormal*	18.42	18.43

The step like shape of the "Denormalized" graph is a result of the denormalized band of numbers that move through the rows (as shown in figure 5), influencing at least one of the processes in every iteration. Since the processes are globally synchronized, one process experiencing a tight band of denormalized numbers will slow down the other processes.

This experiment shows that performance anomalies of floating point operations may cause an unaware programmer to attribute too much performance overhead to the parallel algorithm or communication subsystem, as the main change between the sequential and parallel version is the parallelized algorithm and the added communication.

5 Discussion

Normally, users don't care about the computed results in benchmarks as the important part is to have a representative instruction stream, memory access and communication pattern. The assumption is that the execution time of floating point values do not change significantly when using different input values.

Textbooks don't warn against using zero as an initial value for the equation solving methods. One book (Andrews [7] page 535) even suggest that the interior values should be initialized to zero: *"The boundaries of both matrices are initialized to the appropriate boundary conditions, and the interior points are initialized to some starting value such as zero"*. A contributing factor is that students are taught to use zero values for variables to help catching pointer errors, unless they have a good reason for initializing otherwise.

5.1 Implication for Benchmarks and Automatic Application Tuning

Benchmarks may be vulnerable when varying execution time of floating point operations can mask other effects. Worse even, wrong results from a benchmark may make later analysis based on the data invalid.

An example of automatic application or library tuning is ATLAS [8], where performance measurements are used to automatically tune compilation of the math kernel to a target architecture. If measurements are thrown off, the tuning heuristics may select the wrong optimizations. We have not tested ATLAS for any problems.

5.2 Remedies

Denormal values are well-known for people concerned about the accuracy of applications. Some remedies may help or delay the problem: changing from single precision to double precision delays the introduction of denormal values in our Jacobi benchmark, though we have observed that the band of denormal values will be wider when denormal numbers *do* occur, so the benefit may be limited.

Other remedies are architecture or compiler specific, such as enabling flush-to-zero (FTZ) which converts denormal numbers to zero. Some current architectures support it on-chip, while others have to support it through software, impacting performance. FTZ may also influence the results of the application. On our Intel computers, FTZ is handled in software for x87 instructions, though enabling SSE instructions (`-march=pentium4 -mfpmath=sse` with GCC 4.1.2) allow us to enable on-chip FTZ, resulting in a performance for the *denormal* data set that is similar to the *normal* set. Initial experiments, however, indicate that handling denormal numbers on-chip for SSE is twice as expensive as for x87 instructions.

For this particular benchmark, using nonzero values as initial values for the internal points can remove the problem. The application would also benefit from using multi-grid methods, which could reduce the problem of denormalized numbers considerably.

The floating point status register can be checked to detect if exceptions occurred. If detected, the benchmark results should be considered suspect and investigated, or a different data set should be used. Care must be taken to mask exception types that should be expected, such as *precision*, which occurs whenever a number not representable with floating point is used and does not cause any detectable overhead, such as 1/3. The problem is that checking for floating point exceptions is a suggestion on the line of checking return values from function calls. Programmers, however, tend to forget checking the return values of functions. It also requires that benchmark designers are aware of the problem.

Granularity is another problem. We have experimented with High Performance Linpack [9], observing that it generates *precision*, *underflow* and *denorm op* exceptions, but due to the granularity of the current method, we do not know if there are enough of the denormal numbers to impact the benchmark.

A much better solution would be to use performance counter registers that could count the number of denormal operations in the application. The PowerPC processor has one such counter, but initial experiments with that have not provided us with any stable results. The Intel processors did not have such a performance counter. If we were to be granted a wish, it would be that hardware designers introducing corner cases that causes high overheads would add a corresponding performance counter for those corner cases.

Lacking that, we are we are currently experimenting with the development of tools that can check a benchmark for denormal numbers and other identified performance anomalies. Two promising approaches are based on instrumentation of the Bochs emulator [10], and asking the compiler to generate calls to an instrumented software floating point library rather than floating point instructions. The problem with the latter method is that it doesn't catch inline assembly code in benchmarks.

6 Conclusions

As architectures are getting increasingly complex and the cost of using different components in the systems change, benchmarks may be difficult to create and interpret correctly. Processor designers optimizing for the common case can introduce unoptimized cases that, even if they occur infrequently, cost enough to significantly impact applications, a point well made by Intel's chief IA32 architect [11].

Benchmarks are often simplified versions of existing applications where the results of the computations are not used since the goal of a benchmark is to measure a systems behavior under a specific use. In these cases a programmer may choose to ignore some of the guidelines for programming floating point applications since the accuracy of unused results are not important, and since the guidelines generally stress the problems around correctness and accuracy of floating point values and operations.

We have shown that one concern, namely floating point exceptions, can easily occur in a benchmark and cause significant overhead even when the exceptions are masked off and handled internally in the CPU. Textbooks and well-known guidelines for programmers contribute to setting the trap for programmers.

We have also shown how a benchmark influenced by floating point exceptions in only a small fraction of the calculations may lead to the wrong conclusions about the relative performance of two architectures, and how a benchmark may wrongly blame the parallel algorithm or communication library for a performance problem on the cluster when the problem is in the applications use of floating point numbers.

In reflection of some of the observed behavior of our simple benchmark, we suggest some guidelines for programmers designing and implementing benchmarks.

References

1. IEEE 754: Standard for Binary Floating-Point Arithmetic. `http://grouper.ieee.org/groups/754/`
2. Severance, C.: IEEE 754: An Interview with William Kahan. IEEE Computer, pp. 114–115 (March 1998), A longer version is available as An Interview with the Old Man of Floating Point `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`

3. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23(1), 5–48 (1991)
4. Intel: Intel Architecture Software Developer's Manual – vol. 1: Basic Architecture. Intel, Order Number 243190 (1999)
5. Intel: Intel C++ Compiler and Intel Fortran Compiler 8.X Performance Guide, version 1.0.3. Intel Corporation (2004)
6. LAM-MPI homepage, `http://www.lam-mpi.org/`
7. Andrews, G.R.: Foundations of multithreaded, parallel, and distributed programming. Addison-Wesley, Reading (2000)
8. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing 27(1-2), 3–35 (2001), Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448 (2000) `www.netlib.org/lapack/lawns/lawn147.ps`
9. The LINPACK Benchmark, `http://www.netlib.org/linpack/`
10. Bochs: `http://bochs.sourceforge.net/`
11. Colwell, B.: What's the Worst That Can Happen? IEEE Computer, pp. 12–15 (May 2005)

CoMPI – Configuration of Collective Operations in LAM/MPI Using the Scheme Programming Language

Espen Skjelnes Johnsen, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science, University of Tromsø, NO-9037, Tromsø, Norway
{esj, johnm, otto}@cs.uit.no

Abstract. This paper describes CoMPI, an extension to LAM/MPI, which enables the user to configure collective operations by using the high level programming language *Scheme*. A high level language is used to achieve flexibility. When using CoMPI, applications do not have to be modified or recompiled. We implemented the allreduce collective operation in Scheme, preserving the original LAM/MPI algorithm as implemented in C. We measured the performance and found that the overhead using Scheme was insignificant. Using CoMPI to reconfigure the allreduce communication pattern taking the network topology into account, significant improvement in performance on a multi-cluster was achieved.

1 Introduction

This paper describes an extension to LAM/MPI[5] that enables the user to configure collective operations by using Scheme[1], which is a high level general purpose programming language in the Lisp family.

Currently, the operations that can be configured are broadcast, reduce, allreduce, gather and barrier, but the system is general enough to be extended with other operations if that is required. Sophisticated reconfigurations can be done without modifying or recompiling the application that is to be run.

We will show that high level languages, and Scheme in particular, could be used in performance critical systems despite the general opinion that such languages are slow. This would be an improvement compared to the more conventional way of writing extensions in low level languages like C, due to the fact that many details can be hidden or abstracted away in languages like Scheme.

We will also use CoMPI to show that reconfiguration of collective operations in MPI could result in better scaling when running in multi-cluster[1] environments.

The goal of CoMPI is to provide an efficient framework for experimenting with collective operations in LAM/MPI. It was largely inspired by the PATHS[2] system which uses Python to set up a runtime environment for parallel applications.

[1] In this paper, multi-cluster is defined as a collection of clusters connected by high latency or low bandwidth links.

2 LAM/MPI Background Information

LAM/MPI is an open source implementation of the MPI specification[8] maintained by the LAM team at *Indiana University.* Extensions to LAM/MPI are written as plug-in modules[11] that follow a specification called *System Service Interface*[10]. Multiple classes of SSIs exist for interfacing to various parts of LAM. CoMPI is implemented as a *Collective Operations SSI* module[9], which may be selected by the user at runtime.

After this work was done, LAM/MPI has largely been superseded by a new MPI implementation called Open MPI[2], which has a similar architecture for plug-ins. We thus believe that CoMPI could be ported to Open MPI, but have not yet looked into the details of this.

3 Architecture of CoMPI

The idea behind CoMPI is that when a collective operation is invoked, a hash table lookup is done to check if a suitable routine, a coll-op closure,[3] exists to perform the given operation. If no such closure exists, a coll-op creator function is invoked to create one. The newly created coll-op closure will then be invoked and also cached for reuse later.

Each supported collective operation has at least one coll-op creator. By default, creators are supplied for each collective operation to mimic the behavior of native LAM.

The most primitive way to configure operations, would be for the user to attach additional coll-op creators to each of the collective operations. Coll-op creators are invoked in reverse order of installation (the latest installed is invoked first) until one of them returns a closure. They may also be chained by having one creator calling the next in the list. Chaining may be useful to create wrappers for profiling or debugging.

The interface between the MPI API layer and CoMPI consists of functions called *coll-op trampolines* which are invoked by LAM when the application does a collective operation. The trampolines are responsible for looking up and invoking coll-op closures.

4 Configuration of Communication

The default LAM/MPI collective operations module, `lam_basic`, uses two different schemes depending on the size of a group: for small groups (four or less), the communication is based on a linear scheme (all processes communicate directly with the root process). For larger groups, the communication is organized as a binomial spanning tree[12]. Figure 1 shows the broadcast tree used by `lam_basic` in a cluster with 32 nodes.

[2] `http://www.open-mpi.org`

[3] A *closure* in Lisp terminology is an object containing a function and the lexical environment (e.i. local variable bindings) in which the function was created.

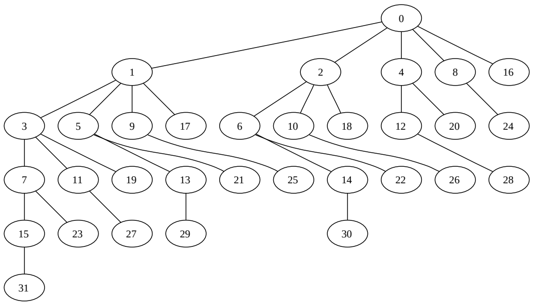


Fig. 1. Broadcast tree for a cluster with 32 nodes

If we assume a homogeneous cluster were the bandwidth and latency between nodes are more or less the same independent of which two nodes are communicating, a binomial tree is very efficient as it utilizes the parallelism in a switched network. But in a multi-cluster environment or in a cluster with SMP nodes, where there may be links between the clusters with lower bandwidth and higher latency than the intra-cluster links, this configuration may turn out to be very inefficient.

The problem is that when multiple messages are sent across the inter-cluster links simultaneously, these links could easily be saturated if the bandwidth is limited compared to the intra-cluster links. If messages are sent sequentially (i.e. at different levels in the tree), this will result in the total latency for sending all messages being the sum of the latency for each individual message. In figure 2 the nodes making up the different clusters have been grouped together, and we can easily see that a significant number of messages are sent across the inter-cluster links, which would greatly reduce the performance of the operation.

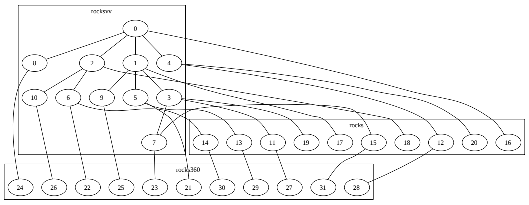


Fig. 2. Broadcast tree mapped to actual cluster layout

One way to improve this is to organize the communication between the nodes in a pattern that matches the underlying network topology in the most optimal way[7,4]. Figure 3 shows a broadcast tree where the nodes within each of the clusters are arranged as separate binomial trees. Communication between the clusters is now limited to only two messages.

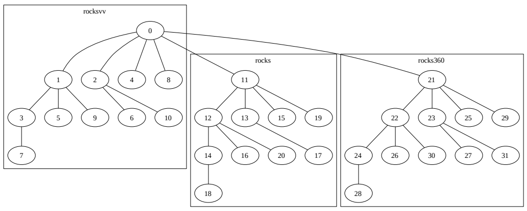


Fig. 3. Reconfigured broadcast tree

We can realize this by using a coll-op creator to CoMPI that considers the underlying network topology when creating coll-op closures. The network topology could potentially be automaticlly detected, but in the current version of our code the user has to manually specify it in a configuration file, by grouping nodes belonging to the same cluster.

5 Implementation Details

CoMPI is written almost entirely in *Scheme*, with only a small amount of glue code in C. The Scheme implementation used is *Bigloo*[4], which is particularly suited because of its efficient compiler and ease of integration. The compiler produces C code as its intermediate stage and is thus portable to all architectures which LAM/MPI is available on. It also contains useful extensions to standard *Scheme*[1] such as *Common Lisp* like macros, type annotations (static type declarations), foreign function interface and a module system.

Figure 4 shows the coll-op trampoline for the operation `MPI_Bcast`. The function `find-coll-op-closure` is called to locate or, if necessary, to create a coll-op closure. The parameters `comm` and `root` are used to identify a coll-op closure that implements the operation for the MPI communicator and operation root.

Finally, the returned coll-op closure is invoked with the remaining parameters, which would perform the actual operation.

```
(define (bcast::int buff::buffer count::int
         datatype::datatype root::int comm::comm)
  (with-coll-op (comm)
    (let ((bcast (intern-coll-op-closure BLKMPIBCAST comm root)))
      (bcast buff count datatype))))
```

Fig. 4. Coll-op trampoline for MPI_Bcast

Figure 5 shows an example of a simple coll-op creator[5] that implements linear broadcast communication. The name of the defined coll-op creator is

[4] `http://www-sop.inria.fr/mimosa/fp/Bigloo/`

[5] This is actually the default callback creator for broadcast, which would be invoked if none of the more specific creators return a coll-op closure.

`linear-bcast` and `BLKMPIBCAST` is a tag identifying the operation. The list `(comm root)` contains the parameters passed from the MPI layer.

The rank of the process is found from the communicator. If the rank matches the root, the helper function `create-psend-closure` (create parallel send closure) is called to create a closure that will send messages to each of the other processes in parallel. Otherwise a closure that calls `receive` first is created. A coll-op closure for broadcast takes three parameters; a buffer with the message to send, the size of the message and a data type. Coll-op closures for other operations may take different kind of parameters.

```
(define-coll-op-creator BLKMPIBCAST linear-bcast (comm root)
  "Coll-op creator for linear broadcast"
  (let ((rank (comm-rank comm)))
    (if (= rank root)
        ;; Root sends to all except to it selves
        (let ((dests (remq! root (range 0 (comm-size comm)))))
          (create-psend-closure dests BLKMPIBCAST comm))
      ;; All other receive from root
      (lambda (buff::buffer count::int datatype::datatype)
        (recv buff count datatype root BLKMPIBCAST comm
         MPI-STATUS-IGNORE)))))
```

Fig. 5. Coll-op creator for linear MPI_Bcast

A special kind of coll-op closures are *wrappers* which, as the name suggests, wraps around other more specific closures. The closure to be wrapped is created by invoking `call-next-creator` from within a wrapper creator. Wrappers may, among other things, be used to modify the behavior of the inner closures or to log information. Figure 6 shows an example of a wrapper which will measure the time spent in the inner coll-op closure, i.e. the time taken to execute the operation.

```
(define-coll-op-creator BLKMPIBCAST bcast-timer (comm root)
  (let ((bcast (call-next-creator))
        (rank (comm-rank comm)))
    (lambda ()
      (let ((rt (time (bcast root))))
        (printf "time spend in broadcast by rank ~a: ~a ms~%" rank rt)))))
```

Fig. 6. Coll-op creator wrapping MPI_Bcast with a timer

6 CoMPI Advantage

One of the advantages with CoMPI is the possibility to configure and experiment with MPI without having to recompile applications. The user does not even need to have access to the source code of the application he is running. LAM/MPI already has some options which may be used at runtime to configure and customize the environment, but the strength of CoMPI lies in the use of a high level programming language. By not having to think about low level details such as memory management, the task of trying out different configurations may become much simpler than if he were to use a language like C. Also by not having to stick with a limited customization language, but having access to a general purpose language gives the system great flexibility. For simpler configurations, CoMPI has a set of built in coll-op creators which let the user specify the network topology based on node ranks or IP numbers.

7 Experiments

A simple experiment was conducted to determine if the performance of CoMPI is comparable to that of native LAM when using the same algorithms, and that the choice of implementation language did not have a negative impact on performance. In addition we're running the same benchmark with a communication pattern configured to match the underlying network layout.

The hardware platform used for these experiments consists of three interconnected clusters, each having the following configuration:

– 28 Dell 370 P4 Prescott, 3.2 GHz, 2 GB RAM
– 38 Dell 370 P4 Prescott 64 EMT, 3.2 GHz, 2 GB RAM
– 44 Dell 360 P4 Prescott 3.2 GHz, 2 GB RAM

In addition, each cluster has a front-end computer identical to the cluster nodes, except that it has an extra network card for external communication. The nodes of the clusters and the front-ends are interconnected using TCP/IP over Ethernet through 1 Gb switches. All communication to and from a cluster goes through the front-end computer. The front-end computers are connected to each other and the outside world, through a dedicated switch. One of the switches is configured to operate as two logical switches through virtualization, but as far as we know this should have no impact on the performance of the benchmarks.

Each cluster uses IP addresses from the private 10.0.0.0 range and is not reachable from outside of the front-ends. To allow routing between nodes from different clusters, IP tunnels are set up between the front-ends. This adds to the inter-cluster latency, increasing it from 100 to 300–500 microseconds as measured with `ping`. The effective bandwidth through the front-ends is about 400 Mbps.

Figure 7 shows the code run in the experiment. The code measures the average execution time of 1000 `MPI_Allreduce` operations, with buffer sizes from 1 to 50000 elements. To make sure that the correct sum is computed, the code also checks the result of each iteration.

Figure 8 shows the result of running the global sum benchmark. The following experiments were done:

Native LAM/MPI (`lam_basic`). The experiment was run using LAM/MPI's native logarithmic broadcast and reduce trees.

```
t1 = get_usecs ();
for (i = 0; i < ITERS; i++) {
  MPI_Allreduce (&i, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  if (ghit != (i* size))
    printf ("oops at %d. %d != %d\n", i, ghit, (i * size));
}
t2 = get_usecs ();
```

Fig. 7. Global reduction benchmark

CoMPI using the `lam_basic` algorithm and tree used above. The experiment was run using CoMPI with the same algorithm as `lam_basic`.

CoMPI with hierarchical mapping. In this experiment we used a hierarchical mapping which minimizes the inter-cluster communication. This mimics the asymmetric algorithm used by MagPIe[7], which is basically a reduce followed by a broadcast.

8 Discussion

As Figure 8 shows, the overhead of CoMPI is insignificant compared to native LAM/MPI (`lam_basic`) when both systems use the same communication pattern. With a hierarchical mapping, the performance of CoMPI is significantly better than `lam_basic`. The main factor contributing to this, is probably the limited inter-cluster bandwidth. The network latency in our setup is relatively small (more than two orders of magnitude) compared to the total time per iteration, and should not have any significant impact on the result.

In addition to the experiment described in the paper, we have also done some informal testing with other collective operations, in particular broadcast, reduce and barrier. We have done this to get an indication if it would be possible

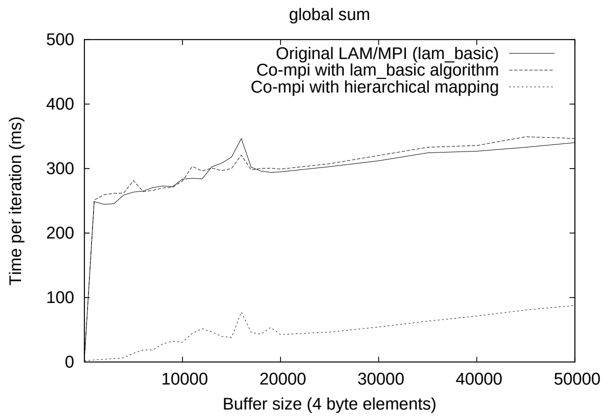


Fig. 8. Results of the global sum experiments

to achieve similar results with other operations. The results of this have been inspiring, which is not unexpected considering that the algorithms for these operations are similar to allreduce.

9 Related Work

CoMPI was originally inspired by PATHS[2] and the discoveries done in [4] and [3] about how performance in LAM/MPI could be improved in clusters with SMP nodes. PATHS uses *Python* as its runtime configuration language and a tuple space system for inter-process communication. The main difference between PATHS and CoMPI is that the former doesn't use MPI directly, but emulates its behavior in respect to collective operations.

MagPIe [7] is an implementation of MPI especially designed for multi-clusters with wide area network connections. The collective operations in MagPIe are optimized for long latency links using both asymmetric (tree based) and symmetric algorithms. The hierarchical mapping used in CoMPI mimics MagPIe's asymmetric algorithm. We have not compared the performance of CoMPI to MagPIe, since, in this paper, we only wanted to document the impact of our modifications of LAM.

Compiled Communication[6] (CC-MPI) describes a method to optimize cluster communication by utilizing information available at compile time using a modified C compiler. CC-MPI and CoMPI share some of the same basic ideas about doing as much static optimization as possible. In CC-MPI this happens when the application is compiled, while in CoMPI when an operation is invoked for the first time. With CoMPI, other configurations can be tested without recompiling the application.

10 Conclusions

We have shown that a high level scripting language can be well suited for use in the infrastructure of high performance clusters, by using Scheme to implement a collective operations module for LAM/MPI.

We have also created a flexible framework, CoMPI, which may be used for experimentation with, and optimization of, collective operations in LAM/MPI without having to modify existing applications.

When running experiments, the observed overhead by using CoMPI is negligible compared to the native collective operations module in LAM. By reconfiguring the communication pattern to match the actual network layout, substantial increase in performance was achieved.

In future work, we plan to extend CoMPI to include all collective operations available in MPI, and make improvements to the current code base to improve usability and stability. We are also planning to implement dynamic reconfiguration based on performance data gathered during program execution.

References

1. Abelson, H., et al.: Revised^5 report on the algorithmic language Scheme. ACM SIGPLAN Notices 33(9), 26–76 (1998)
2. Bjørndalen, J.M., Anshus, O., Larsen, T., Vinter, B.: Paths – integrating the principles of method-combination and remote procedure calls for run-time configuration and tuning of high-performance distributed application. In: Norsk Informatikk Konferanse, pp. 164–175 (November 2001)
3. Bjørndalen, J.M., Anshus, O., Vinter, B., Larsen, T.: Configurable collective communication in lam-mpi. In: Proceedings of Communicating Process Architectures 2002, Reading, UK (September 2002)
4. Bjørndalen, J.M., Anshus, O., Vinter, B., Larsen, T.: The Performance of Configurable Collective Communication for lam/mpi in clusters and multi-clusters.. NIK 2002, Norsk Informatikk Konferanse, November, 2002. Kongsberg, Norway (2002)
5. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium, pp. 379–386 (1994)
6. Karwande, A., Yuan, X., Lowenthal, D.K.: Cc–mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In: PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 95–106. ACM Press, New York (2003)
7. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. ACM SIGPLAN Notices 34(8), 131–140 (1999)
8. MPI Specification Documents, `http://www.mpi-forum.org/docs/docs.html`
9. Squyres, J.M., Barrett, B., Lumsdaine, A.: MPI collective operations system services interface (SSI) modules for LAM/MPI. Technical Report TR577, Indiana University, Computer Science Department (2003)
10. Squyres, J.M., Barrett, B., Lumsdaine, A.: The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department (2003)
11. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 379–387. Springer, Heidelberg (2003)
12. The LAM/MPI Team: LAM/MPI User's Guide. Technical report, Indiana University, Pervasive Technology Labs (2004)

A Load Balancing Strategy for Computations on Large, Read-Only Data Sets

Jan C. Meyer and Anne C. Elster

Norwegian University of Science and Technology,
Dept. of Computer and Information Science,
Sem Sælands v.7-9, NO-7491 Trondheim, Norway
{Jan.Christian.Meyer, elster}@idi.ntnu.no

Abstract. As data repositories grow larger, it becomes increasingly difficult to transmit a large volume of data and handle several simultaneous data requests. One solution is to use a cluster of workstations for data storage. The challenge, however, is to balance the system load, since these requests may appear and change continuously.

In this paper, a new method for load balancing requests on such large data sets is developed. The motivation for our method is systems where large geological data sets are rendered in real-time by a homogeneous computational cluster. The goal is to expand this system to accommodate multiple simultaneous clients. Our method assumes that the large input sets may be examined in advance, and uses simple, continuous functions to approximate the discrete costs associated with each data element. Finally, we show that partitioning a data set using our method involves very little overhead.

1 Background

1.1 Motivation

This work originates from the need to adapt the load balancing scheme of a server which renders images of geological data sets in real-time to suit a multi-user environment. The original system statically divided its input data in order to distribute the load on a computational cluster, but permitted only one user to access the data at a time. This approach would be ill-suited for a multi-user environment, since it spreads the load from each user across the entire array of machines without regard to their differing requirements.

With multiple simultaneous clients, each client may request a different part of the data sets to be rendered with a chosen analysis function. Thus, the computational load from each client may vary significantly. A real-time load balancing approach is required such that the total load is distributed evenly on the cluster, resulting in a similar framerate for all clients.

The objective of this project is to develop a fast balancing method, given that each client will demand rendering of large amounts of contiguous data. We also assume that the number of clients is small relative to the size of the data sets, since typically the original system is scaled such that the requirements of a single client represent a significant load.

1.2 Taxonomy of Load Balancing Approaches

We adopt the following classification of load balancing methods from Piersall and Elfayoumy [1]. In general, methods may be classified in terms of three broad categories: *static* methods, which embed a predetermined partitioning of the load in the description of the balanced task, *semi-static* methods, which estimate balance at program initialization, and *dynamic* methods, which continually reconfigure the load during run time.

Dynamic load balancing further breaks down into the following phases:

1. **Load evaluation** estimates whether to perform balancing.
2. **Profitability determination** compares the cost of rebalancing with the projected benefits.
3. **Work transfer calculation** determines which tasks may be transferred in order to balance the load.
4. **Task selection** selects which of the transfers from the previous phase to carry out.
5. **Task migration** carries out the work transfer finally decided upon.

Our approach is discussed as a dynamic method in these terms, but is partly semi-static due to its use of precomputed information about data sets.

1.3 Related Work

A significant body of work exists to examine the balancing of loads which consist of streams of small requests. This is conventionally modelled as a set of balls (requests) to be put into bins (available hardware). Mitzenmacher, Richa and Sitaraman [4] give an overview of this field, and describe proof techniques for establishing probabilistic bounds on the characteristics of different approaches to this problem.

The balls and bins model is turned around when the task to be performed outgrows the capacity of a single computer, so that the problem must be partitioned. Such partitioning is related to the algorithm solving the problem, which admits another diverse set of techniques.

Munasinghe and Wait [2] propose an approach which utilizes a discrete ideal load measure, and explore the adaptivity of a system where load changes are detected by the individual node, and adjustments are propagated throughout a heterogeneous cluster.

Watts and Taylor [3] attempt to develop a general dynamic load balancing application, and test its application to two different problems. Their approach proves to be restricted in application, but improvements are suggested, and the relevant class of systems is identified.

Piersall and Elfayoumy [1] present an application framework for dynamic load balancing of image processing applications. The framework is general in being platform-neutral, but it is application specific in the sense that it expresses load in terms of columns of image data.

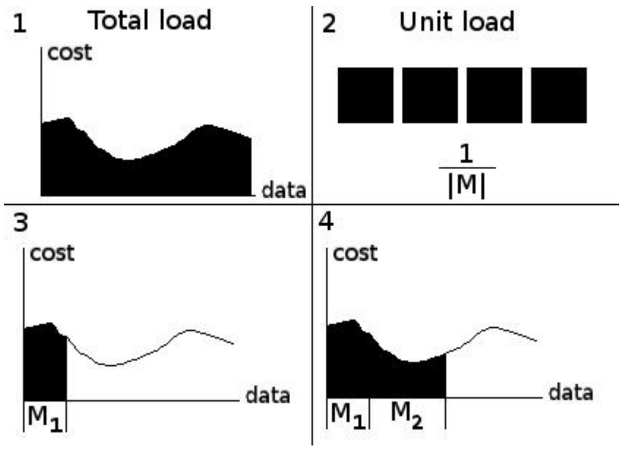


Fig. 1. Finding boundary points in the data set for unit loads on 2 out of 4 machines

Rosenvinge, Elster, and Banino [6] explore the scheduling of master/worker structured applications on heterogeneous systems with background load, and develop a monitor strategy in which worker nodes report measured runtimes to the master.

Overeinder, Sloot, Heederik and Hertzberger [5] propose to handle load balancing in middleware, with library support to partition problems.

2 An Overview of Our Approach

Our method is based on approximating a discrete, empirically measured cost of computation by a continuous, integrable function. The system load may then be idealized as the integral of this function, as opposed to the sum of its discrete values. The virtue of this approximation is that evaluating a definite integral can be computationally very cheap compared to evaluating a series.

Fig. 1 shows the following:

1. To derive a unit load measure, our approximation of the execution costs must be integrated across the entire data domain in use.
2. This computes the load modelled as the area under the cost approximation, which may be divided by the number of machines in the cluster ($|M|$).
3. By using the definite integral of the cost approximation and fixing the lower bound at the beginning of the domain, one can obtain an expression for load in terms of the upper bound. This can be used to compute the upper bound of a unit load, and this load can be assigned to the first machine in the cluster (M_1).
4. Having a domain interval for the first machine, we can now fix the lower bound of the integral at its top, and proceed to find an interval for the second machine to work on, etc.

3 Model

This section introduces a simplified theoretical model of a distributed-memory parallel computer, a data set, and a program to run on this set. The model is restricted to homogeneous and fully interconnected computational clusters.

The following notation is introduced to distinguish the *invocation* of a function f from its value. The invocation of f with the ordered parameter set $X' =< x_1, x_2, ..., x_n >$ will be denoted

$$\lambda f(X') = \lambda f(x_1, x_2, ..., x_n). \tag{1}$$

Alternatively, λf is used for brevity where parameters are obvious or irrelevant.

The system as a whole can now be modelled in terms of a set of machines $M = \{M_1, M_2, ..., M_k\}$, a set of data items $X = \{x_1, x_2, ..., x_n\}$, a program expressed as a set of functions $F = \{f_1, f_2, ..., f_m\}$, corresponding cost functions which map function invocations to the run time they require

$$c_i(\lambda f_i) : (F, X) \rightarrow \mathbb{R}^+, \tag{2}$$

and a transfer constant T which expresses the time required to move a data subset $X' \subset X$ from one machine to another, i.e. $(T \cdot |X'|) \in \mathbb{R}^+$.

With a model of the system in place, its behavior will be treated in terms of *queries* and *work distributions.*

A *query* q is a finite, unordered set of function invocations:

$$q = \{\lambda f_a(X_1), \lambda f_b(X_2), ..., \lambda f_j(X_n)\} \tag{3}$$

such that $X_1 \cup X_2 \cup ... \cup X_n \subseteq X$ and $1 \leq a, b, ..., j \leq m$.

A *work distribution* W is a finite, unordered set of ordered pairs, each associating a machine with a function invocation:

$$W = \{(M_\alpha, \lambda f_a), (M_\beta, \lambda f_b), ..., (M_\gamma, \lambda f_j)\} \tag{4}$$

such that $1 \leq \alpha, \beta, ..., \gamma \leq |M| = k$ and $1 \leq a, b, ..., j \leq m$.

Given a set of queries $Q = \{q_1, q_2, ..., q_n\}$ and a work distribution W, let W_Q denote the projection of W into a set of function applications. W will then *satisfy* Q if and only if $q_1 \cup q_2 \cup ... \cup q_n \subseteq W_Q$, that is, W satisfies Q when it assigns every function invocation in Q to a machine in M.

4 Semi-static Load Balancing Method

The basic observation which underlies this work is that the load arising from a query set Q can be written as the sum of costs of its function invocations, i.e.

$$\Sigma c_i'(\lambda f_i(X_j)) \ | \ \lambda f_i(X_j) \in Q. \tag{5}$$

Assuming that the discrete c_i' may be approximated by some continuous function c_i which shares its characteristics, we propose that the load due to f_i

may be approximated by $\int c_i(\lambda f_i(X))dX$, which can be split into definite integrals

$$\int_{x_j}^{x_{j+1}} c_i(\lambda f_i(X))dX + \int_{x_{j+1}}^{x_{j+2}} c_i(\lambda f_i(X))dX + \cdots + \int_{x_{k-1}}^{x_k} c_i(\lambda f_i(X))dX \quad (6)$$

covering the parts of the domain relevant to Q.

Deciding upon a balanced work distribution W which satisfies Q amounts to finding bounds such that the values of these definite integrals are equal, and assigning all points between bounds to the same machine. Note that this effectively imposes a strictly increasing order on the domain, and that we can therefore speak of "neighboring" machines as those working on adjacent subdomains in this order.

Our method exploits this representation of computational loads for efficiency through the empirical measurement of discrete cost functions and network transfer speeds, and off-line construction of appropriate functions for cost approximation. Furthermore, the following technique for determining a work distribution W from a query set Q is proposed:

1. Consider Q, and determine which $X' \subseteq X$ are involved.
2. Compute total cost of Q by integrating all relevant cost functions across X', i.e. evaluate $\Sigma_i \int c_i(\lambda f_i(X'_i))$.
3. Divide the total load by $|M|$ to find a unit load.
4. Begin at the lower bound of X', and determine successive upper bounds throughout the the domain, using the integral of the cost approximation.
5. Construct W by assigning each bounded subproblem to a machine in M.

Construction of Approximate Cost Functions

Simple continuous functions can be integrated analytically much faster than by numerical approaches. Estimating the cost of a function where run time depends strongly on the parameter values of each invocation, and these vary greatly throughout the set, may create a problem.

One way to address this problem follows from the fact that our model makes no assumption regarding the resolution of the measurements from which the cost function is synthesized. This affords flexibility in the construction, since a subdomain which displays great variability may be characterized by local behavior when it is dominant. While this will introduce some inaccuracy with load estimates, it can maintain a reasonable representation of the cost involved in assigning the local section to a single machine.

The negative impact of this technique is greater when a partition boundary falls in a region where the cost function is a coarser measure of actual cost. The cost of some inexpensive points will be overestimated, and the resulting skew in overall balance will reflect the chosen precision of the cost function in that region. The best tradeoff between the cost function's accuracy and complexity will be application-specific, but we note that as long as the size of the data set outscales the number of machines and queries by several orders of magnitude,

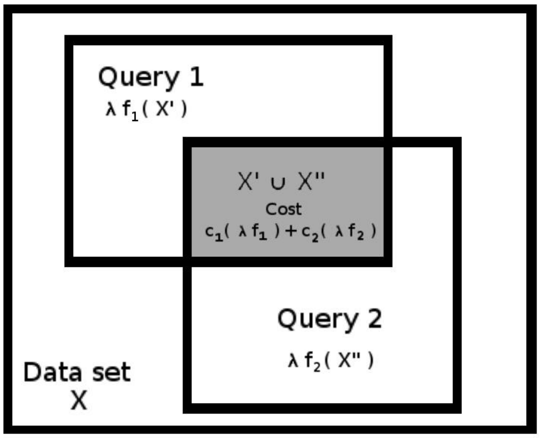


Fig. 2. Two overlapping queries and their intersection

there is considerable freedom to tailor cost approximations with limited impact on accuracy.

5 Dynamic Load Balancing Method

The semi-static technique just described limits dynamic behavior, both for better and worse - freedom of choice is restricted in terms of tuning dynamic operation, but the the problem of selecting tasks for transfer is greatly simplified.

The dynamic features of our approach are driven by data locality. This is to minimize communication, because of its cost compared to that of computation. Data for the function applications of different queries may overlap, as illustrated in Fig. 2. The total cost function will reflect this by an increase in the intersection, meaning that a unit load will span fewer elements in this region after integration. Correspondingly, a machine hosting data points in this region is responsible for computing the functions from all relevant queries on those elements, and the set of elements with lighter load will be proportionally smaller.

This consideration is the reason why data access must be read-only. The situation would be severely complicated by overlapping queries which could write different values to a point in their intersection.

Note that the presence of an intersection introduces a discontinuity in the total cost function, while it remains locally integrable on either side of this discontinuity. The location of the intersection is known from the query set, and the integrals of the separate functions are already known. Due to the linearity of integration, this suggests that boundaries for a partition spanning this discontinuity may be found by integrating up to the intersection, and using it as a lower bound for the sum-of-integrals inside the intersection. This can derive an upper bound corresponding to what remains of the unit load, with the overhead of computing two sums of functions at either end of an intersection. In the worst case, this overhead will be proportional to the number of queries.

5.1 Load Evaluation

Load evaluation must initially determine whether the query set has changed since the load was last balanced, which amounts to examining present query set, and comparing it to the last query set.

If a difference is found, load evaluation must facilitate profitability determination by finding a better partitioning of the present query set by using the integrals of the predetermined cost functions.

5.2 Profitability Determination

In case the queried domain has expanded, new data items must be brought into the computation irrespective of profitability, to keep the system functional.

Otherwise, comparing the previous partitioning with the new estimated partitioning of the altered load yields a set of differences in terms of data points. The cost of proceeding without rebalancing can be computed from integrating the cost functions of the present query set on the intervals indicated by the previous partitioning.

To determine whether it is profitable to rebalance the load, we consider the network transfer rate T in our model.

Rebalancing is profitable if and only if the time lost to transferring data is smaller than the estimated time won by rebalancing.

5.3 Work Transfer Calculation

The work involved in work transfer calculation has mostly been covered by the load evaluation phase. The remaining work breaks down into these points:

1. Review the results of the load evaluation, and determine if any machines have been deprived of their entire workload due to changes in the query set.
2. If any such machine exists, it should be assigned the domain segments which cause the greatest imbalance of its neighbors. As mentioned before, if a new query has arrived, that must take priority here.
3. If a total reassignment of machines has occurred, the boundaries of the $|M|$ subdomains in the data domain must be recomputed to adjust for the update.

5.4 Task Selection

Due to the order imposed on the data set, and the procedure described so far, we already know how far lower and upper bounds must be shifted for each machine in order to restore balance, which gives us the transfer set.

This simple selection of a transfer set is an effect of only considering domain neighbors for work exchange - Watts and Taylor [3] point out that the general problem of task selection is NP-complete. As we cannot guarantee the optimality of the selected transfer set without examining every possible transfer combination, the data set order is used (and thus, maintained) without proof of being the optimal choice. This saves effective computation time otherwise lost to finding an optimal set. At this stage, the balance already computed has been evaluated for profitability, and has been found to save time even if it may not be optimal.

5.5 Task Migration

As each participating computer has two neighbors in the ordered domain, task migration amounts to exchanging boundary areas on all machines in parallel, proceeding in two phases; one for the upper neighbor, one for the lower.

6 Experimental Results

As the dynamic features of the proposed method mostly follow from the semi-static features, it is essential to show that the cost approximation is an effective and efficient load measure. Two experiments are presented, indicating that the semi-static features require negligible runtime, and that effectiveness is related to the precision of the approximating function. Dynamic behavior is not examined empirically.

Both experiments attempt to partition the work load of deciding primality of natural numbers on the domain $[2, 10^7]$ by trial division. The purpose of the work load itself is purely illustrative. Primality by trial division was chosen for its nonuniform runtime across the domain: Best and worst cases of run time vary from 1 to $\sqrt{n}$ trial divisions, and cases are mixed throughout the domain in an undetermined pattern.

As discussed in subsection 4, we construct the cost function by measuring performance over intervals, and approximating cost by the locally dominating behavior. We therefore expect run time at a point n to grow approximately with $\sqrt{n}$, and with noise due to the variable density of primes.

6.1 Preparations

The preparations for the experiments consisted of timing executions of the trial division method. As indicated above, timings were collected from repeated calls across a small interval, to elicit some measure of predictability from each sample point. These can then be approximated by a simple integrable function.

6.2 Experiment 1

The first experiment attempted to approximate the run times purely by scaling the square root function to match the empirical values ($c(\lambda f(x)) = a \cdot \sqrt{x}$). This cost function was analytically integrated, and the resulting expression was solved directly with respect to upper bounds.

Dividing the load in 2 through 8 partitions resulted in the balance detailed in Table 1. The greatest difference column details the difference between the

Table 1. Results from experiments 1 and 2

Parts	E1 Run time	Greatest diff.	Imbalance	E2 Run time	Greatest diff.	Imbalance
2	24.438 s	2.008 s	8.21 %	23.508 s	0.141 s	0.60 %
3	16.828 s	2.039 s	12.12 %	15.672 s	0.117 s	0.75 %
4	12.836 s	1.625 s	12.66 %	11.820 s	0.195 s	1.65 %
5	10.430 s	1.524 s	14.61 %	9.695 s	0.492 s	5.07 %
6	8.875 s	1.453 s	16.37 %	8.141 s	0.453 s	5.56 %
7	7.688 s	1.407 s	18.30 %	7.211 s	0.609 s	8.45 %
8	6.820 s	1.406 s	20.62 %	6.267 s	0.556 s	8.87 %

queries which took the shortest and longest time, while imbalance is the greatest difference as a fraction of total run time.

The time required to compute the partitioning of the domain was recorded using the same system clock as the measurement of the run times, but did not register at all on the time scale of the computation. We hence claim that the performance impact of such a balancing method upon a real-time system working on this scale would be insignificant.

6.3 Experiment 2

The second experiment was motivated by the fact that inaccuracies in the first approximation resulted in an imbalance of $\frac{1}{5}$ of total run time already by an 8-way partitioning. While the test load was selected to be nonuniform, it was still possible to create a better approximation, as the deviations between theoretical and empirical values were known.

The second experiment increased precision by examining these deviations and introducing a linear correction term into the approximation $(c(\lambda f(x)) = a \cdot \sqrt{x} + b \cdot x + d)$.

As witnessed by the results in Table 1, this small improvement in approximation more than halved the level of imbalance in the resulting partitionings in the worst cases.

The resulting integral was not as easily solved with respect to boundaries as previously, so in place of a direct solution, a Newton/Raphson iterative scheme for finding roots was employed. Some performance penalty was expected for this, but in practice the timing of the balancing routine itself was still impossible to measure on the same scale as the rest of the computation.

7 Conclusions and Further Work

We have demonstrated that we can find a load balanced partitioning of a complex, static, nonuniform dataset with almost no overhead. Hence, our method is considered usable under soft real-time requirements. Preprocessing is necessary to find an efficiently computable expression for computational loads.

The developed method fits our motivating application, as the data sets it renders are immutable at run-time, and large relative to the number of clients.

Load balance alters in real time as clients change their selection of data or analysis function. Using predetermined set profiles with respect to analysis functions, the method can determine a balanced data distribution on a visualization cluster very inexpensively, so as not to delay rendering.

The results presented indicate that the precision of this technique can be increased in some proportion to the amount of resources spent on cost function construction, and that the technique thus affords some flexibility for making implementation-specific choices in the tradeoff between accuracy and speed.

Natural extensions to this work include measuring efficiency when dynamic behavior is incorporated, and quantifying the relationship between the accuracy of cost approximations and their complexity.

Acknowledgements. The authors wish to extend their gratitude to Dan Vasaru at Schlumberger, for motivating this research problem, and to our reviewers for providing insightful and very helpful comments.

References

1. Piersall, S., Elfayoumy, S.: DYLAPSI: A Dynamic Load-Balancing Architecture for Image Processing Applications. In: ISCA 15^{th} Intl. Conference on Parallel and Distributed Computing (PDCS 2002) (2002)
2. Munasinghe, K., Wait, R.: Study of Load Balancing Strategies for Finite Element Computations on Heterogeneous Clusters. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 1123–1130. Springer, Heidelberg (2006)
3. Watts, J., Taylor, S.: A Practical Approach to Dynamic Load Balancing. IEEE Transactions on Parallel and Distributed Systems 9(3), 235 (1998)
4. Mitzenmacher, M., Richa, A.W., Sitaraman, R.: The Power of Two Random Choices: A Survey of Techniques and Results. In: Handbook of Randomized Computing, vol. 1, pp. 255–305. Kluwer Press, Dordrecht (2001)
5. Overeinder, B.J., Sloot, P.M.A., Heederik, R.N., Hertzberger, L.O.: A Dynamic Load Balancing System for Parallel Cluster Computing. Future Generation Computer Systems 12(9) (1996)
6. Rosenvinge, E.M.R., Elster, A.C., Banino, C.: Online Task Scheduling on Heterogeneous Clusters: An Experimental Study. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 1141–1150. Springer, Heidelberg (2006)

Automatic and Transparent Optimizations of an Application's MPI Communication

Thorvald Natvig and Anne C. Elster

Norwegian University of Science and Technology(NTNU),
Dept. of Computer and Information Science, Sem Sælands vei 7-9,
NO-7491 Trondheim, Norway
{thorvan, elster}@idi.ntnu.no
http://www.idi.ntnu.no/~elster/hpc-group.html

Abstract. HPC users frequently develop and run their MPI programs without optimizing communication, leading to poor performance on clusters connected with regular Gigabit Ethernet. Unfortunately, optimizing communication patterns will often decrease the clarity and ease of modification of the code, and users desire to focus on the application problem and not the tool used to solve it.

In this paper, our new method for automatically optimizing any application's communication is presented. All MPI calls are intercepted by a library we inject into the application. Memory associated with MPI requests is protected using hardware supported memory protection. The request can then continue in the background as an asynchronous operation while the application is allowed to continue as if the request is finished. Once the data is accessed by the application, a page fault will occur, and our injected library will wait for the background transfer to finish before allowing the application to continue. Performance close to that of manual optimization are observed on our test-cases when run on Gigabit Ethernet clusters.

Keywords: MPI, Automatic Optimization, Memory Protection.

1 Introduction

Optimizing for parallel machines requires extensive knowledge of both general optimization techniques and specific hardware and software details of the machine to use. Any non-trivial parallelization will add some overhead for communication. For example, when parallelizing Game of Life, an iterative 2D partial differential equation solver (PDE) or any other kind of grid-based domain, each iteration requires the border data of its subdomain to be exchanged with the "neighbors".

The goal of our new method presented here was to achieve a solution that automatically optimizes the communication of MPI [1,2] programs. The solution should not require any user intervention at all to be enabled, and should ideally be fully usable on all MPI based parallel architectures.

It is important that the solution does not in any way alter the result of the MPI program, so we have to make sure any optimizations done do not alter the data flow of the program, only the communication. The only difference the user should notice, should be an improved wallclock running time.

A more detailed description of the implementation and more extensive benchmarking may be found in [3].

1.1 Previous Work

Much work has been done on optimizing the individual MPI functions. Some of these are inspired by the ATLAS (Automatically Tuned Linear Algebra Software [4]) idea of taking a basic routine and trying thousands of small variations until you find the specific set of parameters that is perfect for the architecture you are compiling on. Faraj and Yuan [5] have presented such a method for automatically optimizing the MPI Collective subroutines, and Østvold has presented numerous ways of timing collective communication [6].

Ogawa and Matsuoka [7] used compiler modifications to optimize MPI. The compiler would recognize the MPI calls in a program, do a static analysis to find out what arguments are static and then create specialized MPI functions for that program. However, they did not turn synchronous communication into asynchronous. With the introduction of interprocedural optimizations such as those available in the Intel C++ Compiler [8], such optimizations can be extended to all function calls and not just MPI Calls. Unfortunately, this type of optimization requires recompiling the program and for optimal results require several passes of profile-guided-compilation, which increases the amount of work a user has to do before running his program.

The inspiration for using the page protection mechanism and runtime analysis of program data comes in part from the work done by the DUMA[9] and Valgrind[10] projects.

A more user-involved approach to optimization has been presented by Jost [11], which build an expert system based on profiling data to aid the user in their optimizations. Their paper is based around OpenMP, but we believe the instrumentation provided by methods presented here could be used to extend such approaches to work with MPI.

Karwande et.al. have presented a method for compiled communication (CC-MPI), which applies more aggressive optimizations to communications whose information are known at compile time [12].

1.2 Motivation

To see how much the potential gain for this idea was, we implemented a Red-Black 2D SOR Solver [13] using three different methods of exchanging borders.

The first method uses MPI_Sendrecv 4 times. It first sends 'top' and receives 'bottom', then the same for the remaining 3 borders. This is the easiest to code, and allows easy and intuitive understanding of what happens.

The second method uses MPI_Isend and MPI_Irecv (asynchronous operations), and then waits for all 8 requests to finish. This is just a little bit more code, but requires the user to think carefully about what operations can be done concurrently with each other, and which operations can be done concurrently with computation. It is also less intuitive to read that the first method.

The third method computes the new value for the cells along the borders first, and then exchanges these with MPI_Isend and MPI_Irecv in the background while the interior cells are updated. Coding this way caused the communication code to become 5 times the number of codelines used in the first method, and the code was no longer intuitive and easy to understand. Furthermore, at least with MPICH 1.2.7, independent progress (fully asynchonous background transfer) only happens if the amount of data to send is small enough to fit in the OS network buffer (typically 64kb).

Benchmarking these methods on a Pentium 4 cluster using MPICH 1.2.7 showed that just using MPI_Isend gave a significant speedup for small problem sizes, and the full overlapping provided just marginal speedup on top of that (see Table 1 in the Results section for details).

Using blocking functions, the MPI_Sendrecv calls are executed in sequence, and this makes the application vulnerable to latency issues. The datasize in each transfer is so small that most of the transfer time is not spent physically transfering data on the wire, but in OS overhead. Additionally, the receiving machine will receive an interrupt which must be handled and the data must be copied to userspace before the transfer is done. The physcial network is idle during most of this time. By using MPI_Isend and MPI_Irecv we simply make sure we only have this very long wait once for each group of transfers instead of once per transfer.

2 Protected Asynchronous Transfers

2.1 Overview

Our goal is to show that, by using certain safeguards, it is safe to turn any synchronous request into an asynchronous one.

Our first method (paging) uses the hardware page protection mechanism to protect the memory associated with each request. While the transfer occurs in the background, the application is allowed to continue. If the protected memory is accessed, the application is paused until the background transfer is complete. This way, no change of logic in the application is necessary, we just postpone waiting for data until it's actually needed.

The second method (chaining) improves upon paging by recognizing chains of requests without any computation in-between them. The chains are recognized at run-time by our library, and when the start of such a chain is encountered, we allow the requests to proceed in the background. Once the last request is issued, we wait for all of them to complete. Hence, we use the paging to verify that the requests in a chain are isolated from other requests and computation, and after that we gain the benefits our paging method provides without the overhead of updating hardware paging tables.

2.2 Process Injection

Since it was a design goal that the users should not have to change their code at all when using our new method, the first challenge was how to place ourselves between the application and MPI. Our implementation therefore allows for the choice between run-time and static injection.

Run-time injection has the advantage that it can be used for precompiled programs, and may also be used for programs that call MPI through another library or is even written in another language (Python, Java etc). At the moment, run-time injection only works if MPI itself is a dynamic library. Our initial work recognized the specific parts of an executable that contain the MPI library. However, such recognition is impossible if the library is compiled with interprocedural optimizations, as that may make the library functions inlined in the user's code.

Static injection (compile-time) works by overriding `mpi.h` with our own header which wraps the calls we want to intercept. Therefore, it is a much cleaner solution which also works with highly optimizing compilers, but it is then no longer possible to update our injected library without recompiling.

2.3 Marking Memory

When an overridden synchronous MPI function is called and turned into an asynchronous one, we need to track the memory area in use.

For a send request, we perform a heuristic on the datasize. If it is small (contained within one memory page), we simply copy the data to a new buffer and start the send in the background. The application is now free to modify the original buffer without affecting the background transfer. If the request is large, we protect the memory pages the buffer occupies so that they are read-only.

For a receive request, we always protect the application memory and start the transfer in the background to a separate buffer. Once complete, we copy from this buffer to the application memory. If we had write-only memory protection, we could deny the application read access while still allowing MPI to write the incoming data to memory, but unfortunately no current memory protection hardware implements write-only protection [14].

The result of this is that the application will not actually stop until it touches an address that is still being transferred. At that point, a page fault will occur which calls our page fault handler, which simply waits for the request to finish before unprotecting the pages and allowing the application to continue. As such, the computation flow of the application is unchanged, and we avoid any deadlock issues. If the application calls MPI_Finalize before touching the data, we also wait for the communication to finish and restore the state of the page table before allowing the application to continue.

2.4 Using Memory Paging to Control Application Access

The foundation for allowing "safe" background transfer is denying the original application access to its own memory. We do this by using the standard page protection feature found on all modern CPUs. This is normally used to implement swapping and virtual memory. When the OS runs low on memory, it will mark a few pages in virtual address space as inaccessible, and copy the contents to disk, thereby freeing up physical memory. When the application tries to access the memory, a page fault occurs, and the OS copies the data back and marks the page accessible again.

We are only interested in the marking of memory (using standard UNIX `mprotect()` or Win32 `VirtualProtect()`), so when we are talking about page faults here, it is just a page fault in virtual memory. No disk I/O is performed.

As mentioned, most memory protection implementations today suffer from write-implies-read and read-implies-execute. Hence, it is impossible to get a write-only page. If you ask for write-only, what you get is a page with all permissions.

Furthermore, memory protection operates on a whole memory page (usually 4096 or 16384 bytes). You cannot protect just part of a page, so if you protect just a single byte, any other variable occupying the same memory page will be protected as well.

2.5 Overlapping Requests

When tracking memory, we need to be careful about overlapping elements and overlapping pages. Overlapping elements happens when two requests need to access the exact same memory location, while overlapping pages happens when two requests needing access to the same page.

For overlapping elements, the order of send operations is irrelevant, and any number of them can be done simultaneously. However, if a receive operation occurs, any other operation already started must complete, and no other operation can start until the receive operation has completed.

When two requests share the same page, we need to pay attention to what mode was used to start the previous operation on the same page. If we were to start a send operation, and a receive operation is in progress on the same page, it means the page is already protected. If we have determined that the actual elements don't overlap, we have to unprotect the page, copy out the data to send, and then reprotect the page. Similarly, if send operations are already enroute and the page is read-only, we can't change the protection of the page to no access as that will just make the MPI library cause a page fault, so we have to wait until they are done before we change protections. Shared pages also makes the pagefault handler quite complicated, since it has to wait for all requests that might share a page to complete before unprotecting it and allowing the application access.

It should be noted that since our method works on the process level, the method works equally well for single-threaded and multi-threaded applications as well as MPI implementations.

Figure 1 shows a theoretical example of all the requests for a classic Jacobi PDE solver with an 8x8 local grid, 1 layer of shadow cells, and a theoretical page size of 128 bytes. Each row represents one page in memory, and each color is a

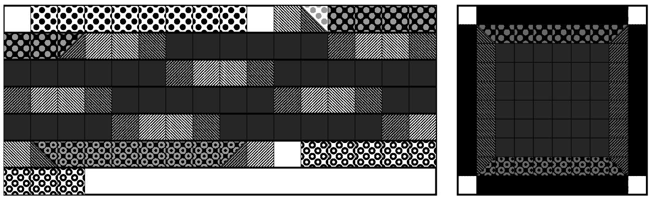


Fig. 1. Left: Memory cells used by border exchange in Jacobi PDE Solver. Right: Cells shown in structured form.

separate request, with bright colors being sends and dark colors being receives. No receive requests have overlapping elements, but the sends do for the corners. Additionally, the only requests that do not share pages are the ones for the top and bottom.

Detecting overlapping elements is computationally expensive due to the complete freedom one has to design MPI datatypes. We have chosen to simply look at the start and end addresses of the memory the request touches, but for the example shown here, that will clearly prevent concurrent transfers in cases where it should be possible (such as the east/west exchanges).

2.6 Chains of Requests

Protection of memory may generate some overhead when updating the pagetables. To avoid this, we have a refinement of our method – recognizing frequently performed operations that belong together and avoid protecting memory.

A chain of requests is any phase of the program that is pure communication and consists of more than a single transfer. For example, for most domain decomposited programs, the exchange of borders is a chain of requests as it consists of multiple MPI calls without any computation or other use of the transferred data in between the calls.

By recognizing such chains, and the knowledge that the majority of communication improvements are in communicating with multiple neighbors simultaneously, we can avoid the overhead of page protection by allowing the requests to start background transmission and wait for all outstanding requests at the end.

It is important that chains be remembered, since only once we have great confidence that a chain is identical for every iteration, can we perform this trick; otherwise the "end of chain" might never happen and the program might read or write unavailable data. If the application suddenly changes behavior, for example by switching solvers after a certain amount of iterations, this may cause data corruption as well. Currently, the best we can do is detect this and inform the user that data corruption might appear.

Our implementation generates a signature of all MPI calls, and once a series of specific signatures have been observed repeatedly, this is treated as part of an inner loop. No memory protection will be done, and we instead wait for all communication to finish at the last request in the chain.

Table 1. Results on Gigabit Ethernet cluster: Average iteration execution time in milliseconds of automatically optimized $n \times n$ 2D SOR PDE solver compared to manual optimization on a 16 node cluster

Method	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$
MPI_Sendrecv	1.14	3.01	3.01	3.49	6.05	16.98
MPI_Isend	0.36	0.99	1.00	1.97	4.48	14.09
Full overlap	0.35	0.99	1.00	1.15	3.87	13.84
Paging	0.92	1.13	1.32	2.94	6.34	22.01
Chaining	0.43	1.07	1.06	2.04	4.55	14.17

3 Experiments and Results

We implemented a Red-Black SOR 2D PDE Solver as it is a good representation of 2D communication patterns. Additionally, its alternating red and black cells stress our method by forcing it to operate under less-than-ideal conditions as it only operates on interleaved data with no contigous blocks, and has a very high degree of overlapping between the requests.

3.1 Results from Optimizing Library

The three versions (original naive implementation and two manually optimized ones) are compared with the automatically tuned program (which is done by injecting into the MPI_Sendrecv version). We ran these benchmarks on numerous machines. Table 1 shows the numbers from our cluster (3.4 GHz Pentium 4 with Gigabit Ethernet using MPICH 1.2.7).

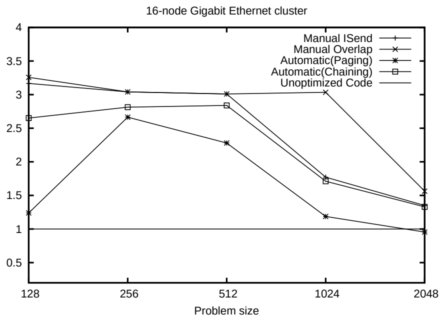
For comparison with dedicated high-speed interconnects, Table 2 shows the results when we tried the same method on a 16-way Altix 350 Itanium 2 machine.

As can be seen on Figure 2, Paging has considerable overhead, but even with this overhead, good results are achieved on small data sizes. Once the data size increases, multiple pages need to be protected and the latency part of the communication is no longer as dominating, so we end up with a slow-down.

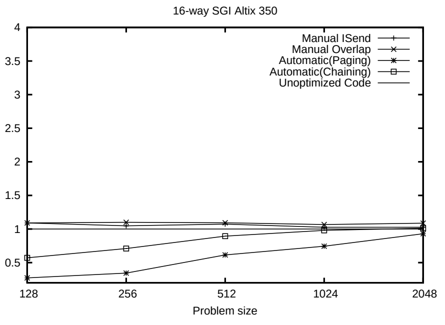
When we enable the recognition of chains, performance is close to that of manually optimizing the program. There is still a small overhead, as the first

Table 2. Results on Altix 350: Average iteration execution time in milliseconds of automatically optimized 2D SOR PDE solver compared to manual optimization on 16-way shared memory machine

Method	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
MPI_Sendrecv	0.12	0.22	0.59	1.92	8.00
MPI_Isend	0.11	0.21	0.55	1.87	7.80
Full overlap	0.11	0.20	0.54	1.80	7.36
Paging	0.44	0.64	0.96	2.58	8.60
Chaining	0.21	0.31	0.66	1.96	7.91



16 node cluster



Altix 350

Fig. 2. Speedup of automatic optimization and manual optimization measured relative to unoptimized code

few iterations will be performed with full memory protection to ensure the chains are valid and don't cause data corruption.

The overhead of memory protection and copying far outweighs the benefits on the Altix 350, which is to be expected. After all, an MPI_Sendrecv operation is simply a memory copy which might be done cache-to-cache. It is only for the really large datasizes ($n > 2048$) that any benefit was seen, and then only with the chaining versions.

4 Conclusion and Future Work

We have implemented, tested and verified a novel method for automatic run-time optimization of communication patterns. Our method requires little or no user intervention and, when paging is always enabled even for chained requests, cannot break data flow.

Our method is fully transparent, so a system administrator might install the static injection as part of the mpicc system, and users would not notice anything but a small speedup of their programs.

On clusters with regular Ethernet interconnects, the improvements make normal applications based on MPI_Sendrecv rival those written with MPI_Isend. This allows users to think and write using simple communication patterns which leads to greater productivity and faster application development for them, and it lets us focus on the optimization part at run-time.

4.1 Current and Future Work

It would be beneficial to add the call-return address to the signature of the request, to make sure the request originated from the exact same point in the source code as well. Currently, if two completely different places in the source code transfer with the exact same parameters, the signatures would be identical. Initial testing shows that if this happens, it will preclude chains from building up confidence as the "same" request has different requests around it.

At the moment, we use a very simple test for overlapping regions of data; if the upper and lower bounds intersect in any way, it's an overlap. However, this gives us a lot of false positives. Unfortunately, proper analysis of overlapping elements is computationally expensive and is likely to generate more overhead than we could gain. We are currently working on using only rudimentary analysis during the paging phase, but if enough confidence is achieved so that we move into chaining, we can do the expensive analysis only once. As long as the signatures stay identical, two requests that didn't overlap the first time will not overlap later on. By delaying until we reach the chaining phase, we also ensure that the added overhead will be spread over many following iterations.

We are evaluating different models for communication performance that can be evaluated and updated in real time. With such models, we can predict how long a transfer will take. If we have no choice but to wait for communication in our pagefault handler, we can then estimate how long the wait will be, and if sufficient time will pass we can start doing the expensive overlap analysis and more indepth chain analysis.

While it will be limited to just one OS (Linux kernel based ones), a kernel module that enables per-thread page manipulations would enable us to let the MPI library do its work from a separate thread without page protection, thereby reducing the amount of copying needed. Doing so would add some overhead in keeping the page tables consistent, and we would still need to flush the TLB when switching between the threads or updating the page tables.

We see a potential for allowing partial unlocking of a message. For the SOR case, on the left and right sides, unlocking half the pages touched once half the transfer is complete would allow half the computation to finish while we finish the last half of communication. While such a method would add overhead, it is better than simply waiting for communication to finish. Unfortunately, there is no good way to measure the progress of a MPI transfer, as it's either done or not done. Such a method would require some modification of the underlying MPI library.

Our method currently fails for applications that use messages for other things than data transfer. For example, if you use messages reception as a method of synchronization, our method will break the synchronization as it will allow the application to continue even before the message has arrived.

Acknowledgements. We would like to thank Dr. Henrik R. Nagel for his help with the structuring of this article, and the reviewers for their very helpful and constructive feedback. We would also like to thank NTNU and NOTUR for access to the supercomputing hardware we ran our benchmarks on.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Technical Report UT-CS-94-230 (1994)
2. Snir, M., Otto, S.: MPI-The Complete Reference: The MPI Core. MIT Press, Cambridge (1998)

3. Natvig, T.: Automatic Optimization of MPI Applications: Turning Synchronous Calls Into Asynchronous, Master's Thesis, NTNU (2006)
4. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing 27(1-2), 3–35 (2001)
5. Faraj, A., Yuan, X.: Automatic generation and tuning of MPI collective communication routines. In: ICS '05: Proceedings of the 19th annual international conference on Supercomputing, pp. 393–402. ACM Press, New York (2005)
6. Østvold, Å.: Timing and Measurement Techniques for MPI Collective Communication Operations, Master's Thesis, NTNU (2003)
7. Ogawa, H., Matsuoka, S.: OMPI: optimizing MPI programs using partial evaluation. In: Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing, p. 37. IEEE Computer Society, Washington (1996)
8. Dulong, C., Krishnaiyer, R., Kulkarni, D., Lavery, D., Li, W., Ng, J., Sehr, D.: An Overview of the Intel IA-64 Compiler. Intel Technology Journal Q4, p. 15 (1999)
9. D.U.M.A. - Detect Unintended Memory Access, `http://duma.sourceforge.net`
10. Nethercote, N., Seward, J.: Valgrind: A Program Supervision Framework, Electronic Notes in Theoretical Computer Science, 89 (2003)
11. Jost, G., Chun, R., Jin, H., Labarta, J., Gimenez, J.: An Expert Assistant for Computer Aided Parallelization. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 665–674. Springer, Heidelberg (2006)
12. Karwande, A., Yuan, X., Lowenthal, D.K.: An MPI prototype for compiled communication on Ethernet switched clusters. J. Parallel Distrib. Comput. 65(10), 1123–1133 (2005)
13. Young, D.M.: Iterative solution of large linear systems. Academic Press, New York (1971)
14. Intel®64 and IA-32 Architectures Software Developer's Manual, vol. 3A, System Programming Guide, Intel (2006)

Parallel Methods for Real-Time Visualization of Snow

Ingar Saltvik, Anne C. Elster, and Henrik R. Nagel

Norwegian University of Science and Technology,
Department of Computer and Information Science
Sem Sælands v.7-9, NO-7491 Trondheim, Norway
ingar@saltvik.org, {elster, hrn}@idi.ntnu.no

Abstract. Snow is a familiar scene in the Nordic countries during the winter months. This paper discusses some of the complex numerical algorithms behind snow simulations. Previous methods for snow simulation have either covered only a very limited aspect of snow, or have been unsuitable for real-time performance. Here, some of these methods are combined into a model for real-time snow simulation that handles snowflake motion through the air, wind simulation, as well as accumulation of snow on objects including the ground. With our goal towards achieving real-time performance with more than 25 frames per second, some new parallel methods for the snow model are introduced. The algorithms are first parallelized by dividing the data structures among threads. This scheme is then improved by overlapping inherently sequential algorithms with computations for the following frame, to eliminate processor idle time. SMP and multi-core systems are considered.

Keywords: Parallel Computing, Computer Graphics, Simulation, Snow.

1 Introduction

Realistic visualization of natural phenomena has been subject to intensive research for decades. Together with the tremendous increase in compute power, this has indeed resulted in beautiful imagery both in still images, animations, and computer games. Still, there is an ever present demand for further developments in the field with respect to both realism and speed, as well as utilization of the seemingly ever increasing power available in today's computer hardware.

Snow, a familial natural phenomenon in the Nordic countries, has the ability to completely change the mood and appearance of a scene. From a computer graphics point of view, snow can therefore be used to create evocative and realistic looking scenes. Despite this great visual significance, realistic and real-time snow modeling has been given little attention in computer graphics research.

The main characteristic of today's microprocessor design is the increased level of parallelism. As the dual- and multi-core processors are entering the consumer market with full momentum, parallelism traditionally found only on high-end servers and supercomputers is becoming available for the man on the street. This introduces new challenges with respect to exploiting these new architectures for maximum performance.

This paper presents new methods for utilizing parallel computer systems to achieve real-time performance of already known algorithms for snow visualization. These algorithms are analyzed with respect to the real-time requirement, visual quality and parallel efficiency. Finally, an optimized, parallel implementation of these algorithms is presented.

2 Previous Work

2.1 Falling Snow

Two main approaches can be found in previous work on rendering falling snow; billboards and particle systems. Billboard techniques creates the appearance of snow by rendering an image of snow on a distant object, or on a transparent plane in front of the camera. The effect of falling snow in real-time can also be obtained by rendering snowflakes to a texture that is composited onto the scene [1].

On the other hand, the particle system approach [2] is the most physically realistic, as it treats each individual snowflake as an object whose motion is determined by the influence of various forces in 3D. This requires that each snowflake is handled separately for updating and rendering, and is more expensive in terms of processing power.

Moeslund et al. [3] use a physically based model for the forces acting on falling snowflakes that is based on the particle system approach. The model identifies four main forces; drag, lift, gravity, and buoyancy. The drag force is the result of the difference in speed between the object and the surrounding air:

$$F_{drag} = \frac{V_{fluid}^2 \cdot m_{snow} \cdot g}{V_{max,z}^2} \tag{1}$$

The direction of the force is the same as $\boldsymbol{V}_{fluid}$, which is the velocity of the surrounding fluid. m_{snow} is the mass of the snowflake, and $V_{max,z}$ is the maximum velocity the snowflake achieves when falling freely in the z-direction (upwards). The lift force is the cause of the seemingly random side to side motion of heterogeneous falling objects, and is modeled as a spiral path. Gravity and buoyancy are constant forces. These forces are finally combined with Newton's second law of motion, to calculate new positions for the snowflakes for each frame.

2.2 Wind

Wind is of the utmost importance in order to simulate the motion of snowflakes. The basic idea is to simulate a wind field, or more specific, a velocity field, to simulate the force that the air exerts on the snowflakes. Notice that the wind velocity is needed in Equation (1). Two main methods have been identified for wind simulation; Computational Fluid Dynamics (CFD) and the Lattice Boltzmann Model (LBM).

CFD is based on the idea of simplifying the Navier-Stokes equations (NSE) to make them computationally feasible for fluid simulations. Much work has been performed in this area. A numerical finite-difference scheme for solving the NSE is introduced in [4]. This scheme is improved in [5] that presents an unconditionally stable method for the finite-difference term, and a method to compensate for numerical dissipation is presented in [6].

LBM takes a different path, by building on the concept of lattice gas automata. This is discussed in detail in [7] and a method that uses the GPU (graphics processing unit) to achieve real-time performance is discussed in [8].

2.3 Accumulating Snow

There are many different approaches to the visualization of a snow cover. Two important distinctions can made based on the methods found in previous work; the amount of necessary computing resources, and how much work is needed beforehand by the modeler to place the snow. The most common technique is to use textures with snow patterns to make objects snow-like. While being simple and cheap, it does not look very realistic, works only for small amounts of snow, and requires much manual work to place the textures beforehand.

One method uses hardware accelerated features of the GPU to generate a snow cover texture on objects exposed to the sky and requires no extra modeling effort [9]. Another method places height matrices on all surfaces that can receive snow, to store the snow depth [10]. When snowflakes hit a surface, the nearest height value is increased. To render the scene, triangulations are created from the height matrices, and rendered using Gouraud shading by means of OpenGL's functionality to efficiently do so. Both of these methods work in real-time.

Of the methods not intended for real-time rendering, the most successful one generates a thick snow blanket on the scene with minimum modeler interaction, taking advanced features such as avalanches into consideration [11]. Another method [12] uses the fluid simrrulation from [6] to calculate how the accumulating snow is affected by wind. Snow is accumulated by storing the snow depth on the horizontal surface of voxels marked as occupied.

2.4 Parallel Methods for Visualization of Snow

The literature is sparse with respect to the direct application of parallel computing to snow simulation. A CM-2 supercomputer has been used to simulate a particle system [2]. The unconditionally stable fluid simulation [5] has been parallelized [13]. Optimized, parallel algorithms for visual simulation of smoke, based on [6], are discussed in [14]. Smoke simulations are also relevant, because the parallelization techniques for the underlying fluid simulations can be used when simulating wind.

3 Snow Modeling

Our model for real-time visualization of snow consists of three main components based on known methods from previous work; falling snow, wind, and accumulating snow. These components are based on known theory from previous work, that is suitable for real-time application and parallelization. However, some visual quality may be sacrificed to reach the necessary performance.

3.1 Simulation Model

The method presented in [3] is used for falling snow simulation. This method has been chosen because we need to track the 3D positions of snowflakes to decide where to accumulate snow. In this model, the position and velocity of each individual snowflake is tracked and updated according to the forces described in Section 2.1, by using Newton's 2. law of motion.

For wind simulation, CFD is chosen, and is largely based on [5,6], with the vorticity confinement term left out. The main reason is that it is a well-tried theory, and some work on parallelizing the algorithms has been attempted. Refer to the original articles or [15] for a more thorough derivation of these algorithms. Basically, it consists in approximating the incompressible Euler equations:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nabla \boldsymbol{p} \tag{3}$$

The main steps in this algorithm are:

1. Solve for the first term in Equation 3 by self-advecting the velocities backwards in time using the unconditionally stable method of [5], creating a temporary velocity field $\boldsymbol{u}^*$.
2. Create right hand side vector of linear system resulting from the Poisson equation derived in [4,5,6].
3. Solve the Poisson equation using the SOR algorithm. SOR is chosen because of its parallel properties, as described later.
4. Project $\boldsymbol{u}^*$ into a new velocity field $\boldsymbol{u}$ for the next frame, using the result from step 3.

For snow accumulation, the height matrix method from [10] is used. For each frame, the objects are checked for intersections with snowflakes. If a hit occurs, the nearest snow height value is increased. In the original paper, it was suggested that the method should be extended with wind, as is realized in this paper.

3.2 Rendering

Realistic and efficient rendering of individual snowflakes is beyond the scope of this paper. The rendering has therefore been kept simple, by representing snowflakes with three rectangles that are perpendicular to each other. As most snowflakes are far away from the viewer, this approach works well.

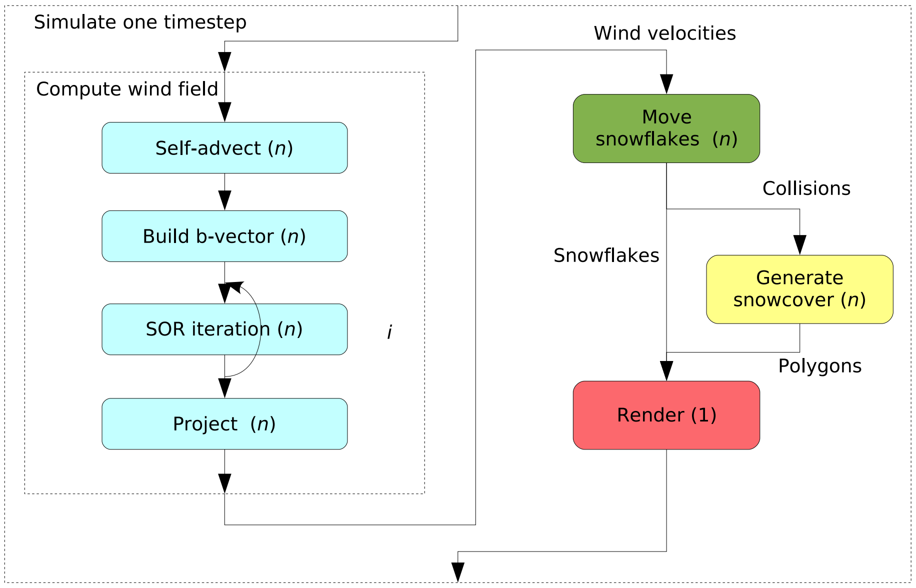


Fig. 1. Overview over units of work and the data flow between them. The numbers indicate the number of subtasks that can be executed in parallel, where n is any number.

Rendering of the snow cover is realized in the same way as in [10], which creates a triangulation based on the snow height values. Rendering is done using the shading capabilities of OpenGL, but to calculate normal vectors quickly, an approximation based on gradient vectors is used.

3.3 Complete Snow Model

Figure 1 shows how we have put together the components for the snow model. The pieces are not new, but, as far as we know, a real-time snow model including all of these features together has not been proposed before.

The first step for computing a new time-step, is to update the wind field. This is achieved with the four step fluid simulation already described. To update the snowflake positions, the velocity field of the wind simulation is used in the falling snow algorithm from [3]. When the snowflakes have been moved, they are checked for collisions with objects in the scene. If there has been a collision, the nearest height value is increased. After the intersection testing has completed, a triangle representation is created of the height values. Finally, the snowflakes and snow cover triangles are rendered using OpenGL.

3.4 Profiling

To see how the implementation behaves without regard to any parallelization, it was profiled in sequential mode. The GNU Profiler (gprof) was used to generate

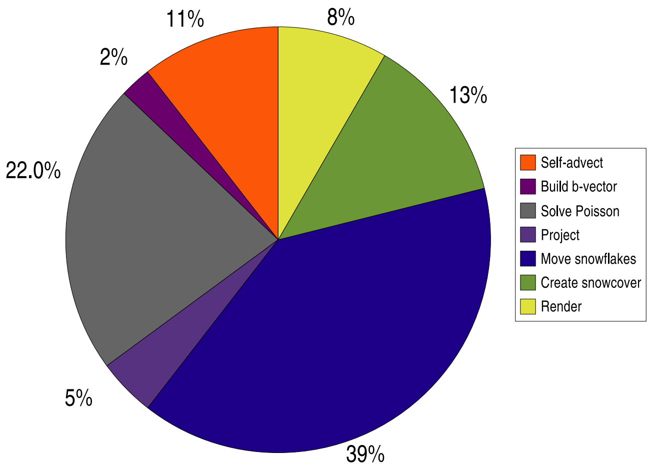


Fig. 2. Profiling results of sequential version with the default configuration

the profile, with one processor and our default size scene with a 40x40x10 wind field resolution, 20,000 snowflakes, and 12,050 triangles in the snow cover. The distribution of time spent is shown in Figure 2.

The profile shows that the simulation of snowflake movement and wind field simulation are the most resource hungry components of the model, together occupying 79% of the execution time.

3.5 Parallelization

To achieve better performance and allow real-time simulation of bigger scenes, the algorithms are parallelized. The computationally significant routines are loops over arrays of model data and because computations in each iteration are independent of the other iterations, they can easily be distributed among threads. However, it is not obvious how the successive over-relaxation (SOR) algorithm can be parallelized similarly. Our approach here is to disregard any data dependencies in the algorithm. Even though this may yield incorrect results, it has proved to be a sound optimization, without any visible degradation.

One great source of parallel inefficiency is rendering. Because only one thread is allowed to make calls to OpenGL, all but one thread must wait while this thread is doing the rendering. However, the wind computations for the following time-step do not depend on the completion of the rendering step. We therefore use the same idea as presented in [14]. By letting fluid simulation computations for the next time-step start before rendering has completed, we eliminate processor idle time. This is illustrated in Figure 3 and 4. Also, by using double buffering, we can overlap snowflake computations and rendering.

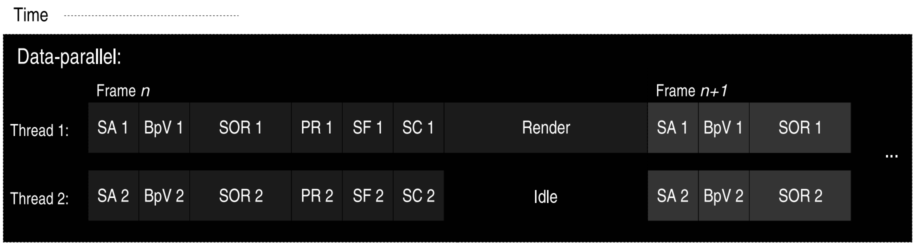


Fig. 3. Data-parallel implementation

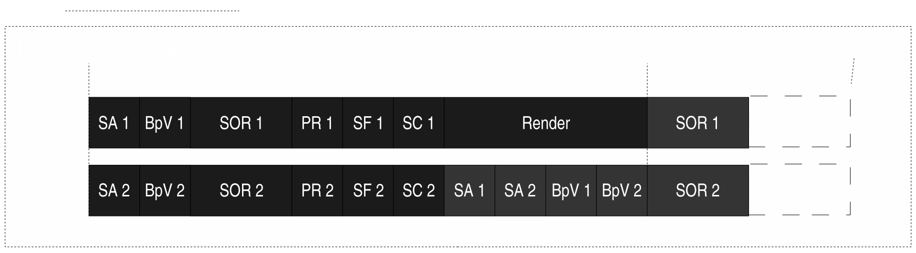


Fig. 4. Data-and-task-parallel implementation

4 Results

Figure 5 shows the visual results of our snow simulation. Performance results are given in Figure 6, which shows that, for the largest scene, we get a 1.79 speedup on a dual CPU computer. The performance gain of introducing the task-parallelism is actually 29%, which is much better than we would expect because the rendering takes only 8% of the time in sequential mode in Figure 2. Also, for the two largest scene sizes, the parallel simulations reach real-time frame-rates, while the sequential do not.

5 Discussion

This project has successfully demonstrated that it is possible to combine complex snow simulations with the requirements of real-time performance in order to obtain realistic, real-time calculated snow scenarios. The key point to achieving this is to keep both the simulation calculations as well as the rendering simple by , for instance, stopping the wind simulation after the wind field has reached a stable state and only using three triangles for each snow flake. Intersection testing was, however, not performed with vertical surfaces. Including this would

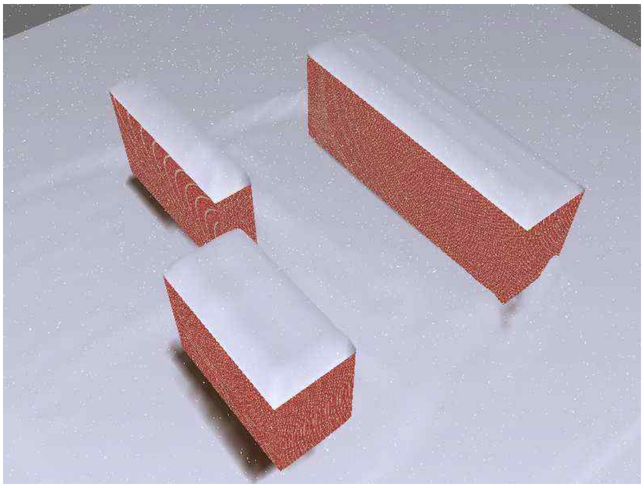
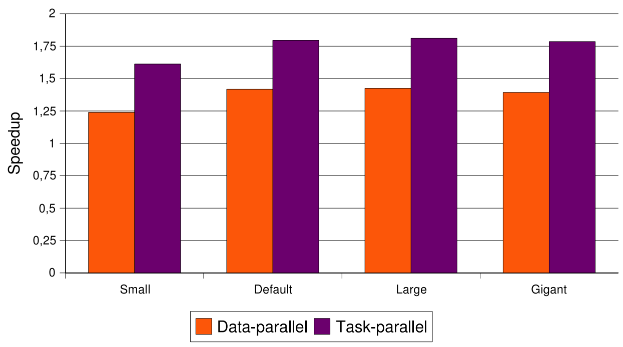


Fig. 5. Simulated snow scene, running at 37.5fps. The scene includes 30,000 snowflakes, 20,800 snow cover triangles, and the wind field resolution is 48x48x12 grid points, running on a 3.2GHz P4 dual CPU workstation. This is the large scene in Figure 6.



	Small	Default	Large	Gigant
Wind field resolution	20x20x5	40x40x10	48x48x12	54x54x9
Number of snowflakes	5 000	20 000	30 000	40 000
Triangles in snow cover	5 352	12 050	20 800	21 760
Data-parallel frame rate	133.2	45.8	29.5	23.4
Data-parallel speedup	1.24	1.42	1.43	1.39
Task-parallel frame rate	173.3	58.0	37.5	30.0
Task-parallel speedup	1.61	1.80	1.81	1.79

Fig. 6. Speedup comparison between data-parallel and task-parallel implementation on a Pentium Xeon 3.2GHz dual CPU workstation with NVidia Quadro FX3400 graphics

make snow scenes more realistic than the one depicted in Figure 5 by making snow stick on vertical surfaces as well.

Even though the snow simulation runs in real-time, it may be problematic to use it in other contexts. The reason for this is that it utilizes two CPUs to their maximum, leaving no resources for other tasks. Feasible solutions to this may be to use more CPUs or cores, as future processors are likely to have much more than two cores. The growth of multi-core systems was in fact one of the main motivations for this project, but the circumstances did not allow trying this new architecture. Concerning rendering, the limiting factor seems to be the data traffic to the GPU, so a possible solution here might be to use the GPU to take some load off the CPUs.

6 Conclusions and Future Work

The contributions of this paper are twofold. First, a snow model intended for real-time applications was presented. Our model combined falling snow, wind simulation, and snow accumulation techniques from previous work. The model is regarded as a contribution, because previous work either has focused one or two of these aspects or has not been intended for real-time use.

Second, a proof-of-concept implementation was created, that demonstrates that the snow model is feasible for real-time frame rates on today's modern workstations. This was achieved by parallelizing the algorithms involved in the simulation for shared-memory architectures.

Among candidates for future work, is analyzing multi-core CPU systems in more detail, as the circumstances only allowed explorations of dual CPU systems. Also, other higher level threading libraries such as OpenMP could be investigated further. Another highly relevant direction that is not considered here, is to use the GPU to accelerate the computations.

Related to the snow model, the wind computations could be terminated if the wind field converges to a stable state, to save computation time. A more advanced accumulation model would be highly beneficial, in particular the stability computations and arbitrary surface accumulation from [11] are interesting.

Acknowledgements. We would like to thank the referees for their very helpful comments.

References

1. Langer, M.S., Zhang, L., Klein, A.W., Bhatia, A., Pereira, J., Rekhi, D.: A spectral-particle hybrid method for rendering falling snow. In: Keller, A., Jensen, H.W. (eds.) Rendering Techniques, pp. 217–226. Eurographics Association (2004)
2. Sims, K.: Particle animation and rendering using data parallel computation. In: SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pp. 405–413. ACM Press, New York (1990)

3. Moeslund, T.B., Madsen, T.B., Aagaard, M., Lerche, D.: Modeling falling and accumulating snow. In: Second International Conference on Vision, Video and GraphicGraphics (2005)
4. Foster, N., Metaxas, D.: Modeling the motion of a hot, turbulent gas. In: Computer Graphics (Annual Conference Series), vol. 31, pp. 181–188 (1997)
5. Stam, J.: Stable fluids. In: SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pp. 121–128. ACM Press/Addison-Wesley Publishing Co., New York (1999)
6. Fedkiw, R., Stam, J., Jensen, H.W.: Visual simulation of smoke. In: SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 15–22. ACM Press, New York (2001)
7. Wei, X., Li, W., Mueller, K., Kaufman, A.E.: The lattice-boltzmann method for simulating gaseous phenomena. IEEE Transactions on Visualization and Computer Graphics 10(2), 164–176 (2004)
8. Wei, X., Zhao, Y., Fan, Z., Li, W., Yoakum-Stover, S., Kaufman, A.: Blowing in the wind. In: SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association, pp. 75–85 (2003)
9. Ohlsson, P., Seipel, S.: Real-time rendering of accumulated snow. In: SIGRAD 2004. The Annual SIGRAD Conference. Special Theme – Environmental Visualization, pp. 25–32 (2004)
10. Haglund, H., Andersson, M., Hast, A.: Snow accumulation in real-time. In: SIGRAD2002, The Annual SIGRAD Conference. Special Theme – Special Effects and Rendering, November 28–29, 2002, Norrköping, Sweden (2002)
11. Fearing, P.: Computer modelling of fallen snow. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp. 37–46. ACM Press/Addison-Wesley Publishing Co., New York (2000)
12. Feldman, B.E., O'Brien, J.F.: Modeling the accumulation of wind-driven snow, Technical Sketch. In: SIGGRAPH 2002, San Antonio, TX (2002)
13. Bryborn, M., Klein, R., May, T., Schneider, S., Weber, A.: A portable, parallel, real-time animation system for turbulent fluids. In: Guizani, M., Shen, X. (eds.) IASTED International Conference on Parallel and Distributed Computing and Systems '00, International Association of Science and Technology for Development, pp. 394–400 (2000)
14. Vik, T., Elster, A.C., Hallgren, T.: Real-time visualization of smoke through parallelization. In: PARCO, pp. 371–378 (2003)
15. Saltvik, I.: Parallel methods for real-time visualization of snow. Master's thesis, Norwegian University of Science and Technology (2006)

Support for Collaboration, Visualization and Monitoring of Parallel Applications Using Shared Windows

Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science, University of Tromsø, NO-9037 Tromsø, Norway
{daniels, jmb, otto}@cs.uit.no

Abstract. Results produced by a parallel application are typically collected and visualized on one display accessible to a single user. Collaboration between several researchers is usually achieved by sharing entire desktops. We have developed a system that shares windows, both from parallel applications and from desktop applications, with other users or to a wall-sized, high resolution display. Parallel applications can create several shared windows for each thread or process, enabling runtime visualization and monitoring. To aid collaboration, we provide multiple cursors for use on a display wall, allowing several researchers to interact simultaneously with windows shared by parallel and desktop applications. We measure the system's performance, and show that using shared windows for runtime visualization of the Mandelbrot computation increases the application's execution time by approximately 1.4%, while performance for sharing desktop application windows is halved as the number of users is doubled.

Keywords: Display wall, shared windows, multiple cursors.

1 Introduction

Current systems for runtime visualization of results from cluster applications are limited in their support for collaboration [1], as they rely on the X Window System [2] for window management and display. Visualization is typically done by a single user on a single display, making collaboration difficult. To collaborate, researchers have to share their entire display, which is often more than is necessary or desired. Finally, there are no standard desktop environments for wall-sized, high-resolution, tiled displays (display walls) that offer multiple cursors [3].

We have developed a system that can share windows from a parallel application and from desktop applications, and which provides support for multiple cursors on a display wall. Windows can be shared with other users and to a display wall, while the system's support for multiple cursors enables researchers to interact with shared parallel and desktop application windows on a display wall.

Figure 1 shows a user looking at a visualization of the Mandelbrot fractal, where each process of the parallel Mandelbrot computation draws into its own, shared window. On the display wall, the shared windows are displayed and placed next to each other to form a complete picture, whereas on the lower-resolution laptop display, there is only room for one window at a time.

Two scenarios further motivate the system presented in this paper. The first scenario concerns the use of shared windows as a means for run-time inspection of parallel applications. Each process of the parallel application creates a shared window, and uses it to visualize results or monitor the parallel application's performance. Figure 2 illustrates this scenario.

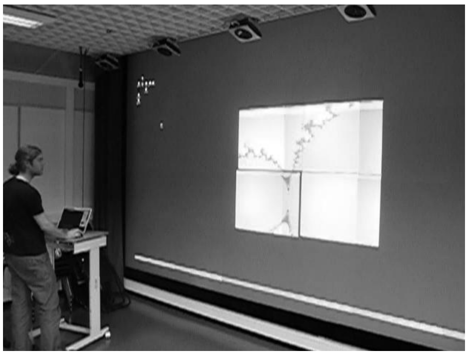


Fig. 1. A user looking at a visualization of the Mandelbrot fractal using shared windows on a display wall. The windows are placed next to each other, forming a complete picture.

In the second scenario, a group of researchers visualize a set of results on their desktop computers. To share data, they need to share desktop application windows with each other and a display wall. The other users can interact with the shared windows, modifying the shared view or change other settings as if the windows were local. On the display wall, several users can interact simultaneously using multiple cursors.

To meet the demands from these scenarios, our system should (i) support sharing of windows between different window systems and hardware platforms, and (ii) support the use of several cursors on a single, large desktop on a display wall.

We evaluate the performance of the parallel application window sharing subsystem by sharing windows containing the output from a parallel version of Mandelbrot, demonstrating that windows can be shared with less than 1.4% increase in the parallel application's execution time. This low impact on performance is due to a number of factors. First, the Mandelbrot application generates new content only about every five seconds, which means that the window sharing system only needs to provide updates to the shared windows at this rate. A higher rate of updates would likely increase the overhead from the window sharing system. Second, the benchmark was run without load balancing at either the application or system level, resulting in ample time for the window sharing system to run in on most nodes. Finally, since the window sharing system runs as a thread inside each process of the parallel application, it knows when the parallel application updates its shared windows and thus avoids sending unnecessary updates.

For the desktop application window sharing subsystem, the performance decreases by a factor of two when the number of window subscribers is doubled. This is caused by a combination of having to poll window contents in order to discover updates, the request-based protocol between publisher and subscribers, and the publisher's implementation. The publisher batches requests, before a best-effort timer fires and makes the publisher process the requests, polling the window for updates at the same time.

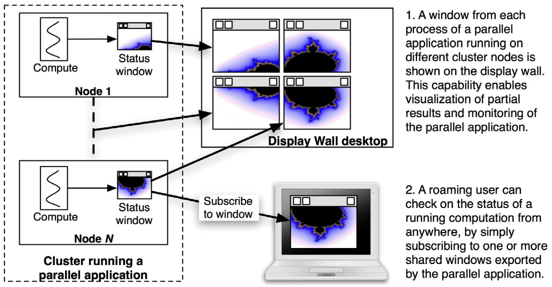

Compute
Status window
Node 1
Compute
Status window
Node N
Cluster running a parallel application
Display Wall desktop
Subscribe to window
1. A window from each process of a parallel application running on different cluster nodes is shown on the display wall. This capability enables visualization of partial results and monitoring of the parallel application.
2. A roaming user can check on the status of a running computation from anywhere, by simply subscribing to one or more shared windows exported by the parallel application.

Fig. 2. Windows shared by a parallel application are accessed both for collaboration on a display wall and monitoring on a laptop

Parallel application window sharing performs better, since the window sharing system is integrated with the parallel application at the source code level. In contrast, sharing desktop application windows does not require modifications to the desktop application's source code, at the cost of lower performance.

Our main contribution with this paper is the integration of (i) shared windows as a means for runtime visualization of results and state directly from processes of a parallel application, (ii) the ability to share desktop application windows rather than a user's entire desktop, (iii) displaying shared windows from both parallel and desktop applications on a wall-sized, high-resolution tiled display, and (iv) support for multiple cursors on a display wall.

2 Related Work

VNC [4] and other remote desktop solutions [5,6] allows one to share an entire desktop. Although some VNC implementations can restrict the shared area to regions of the desktop, this does not amount to true window sharing, as any window brought within the shared region will be visible to others, whether intentional or not. SharedAppVNC [7] enables sharing of independent windows over VNC's Remote Framebuffer protocol on Mac OS X, Windows and Linux. The technique and code we developed for sharing windows on Mac OS X was shared with the developers of SharedAppVNC.

VNC is based on sharing the pixel representation of a remote display. We use the same approach in our window sharing system. In THINC [6], the authors demonstrate a solution that achieves better performance, in part due to their use of lower-level drawing operations to reduce communication. Their techniques are more complex to integrate into the window sharing system, as they rely on installing drivers into the X Server and intercept drawing operations to the framebuffer. The operations are then encoded and transmitted to clients. Due to the low level at which this is implemented, THINC has no concept of individual windows. In our opinion, this makes a window sharing implementation utilizing ideas from THINC harder to realize. Other systems that use drawing operations to transfer display contents include the X Window System [2] and Microsoft Remote Desktop. We have chosen to share windows using their pixel representation, as parameters like coordinate systems, color spaces and line cap styles can be ignored.

Microsoft's Messenger and NetMeeting software [8] support application sharing under Microsoft Windows. Citrix' Presentation Server [9] supports application sharing across platforms. The drawback of application sharing is that it does not support sharing single windows. A shared application with multiple windows would make all those windows visible to other users, while window sharing would allow just a single window from the application to be shared. WinCuts [10] can support window sharing on Windows, but does not support interaction, and only updates windows once per second. For the X Window System, there are many application sharing solutions, including XTV [11] and Hewlett-Packard's commercial Shared X. Xmove [12] allows one to move applications between X servers, but does not support sharing the application with several users at the same time. MAST [13] is a tool that supports pixel-based application sharing for the Access Grid [14] on Microsoft Windows and Linux.

The MPI Parallel Environment, MPE [1], supports the creation of windows from each thread or process within a parallel application. Since MPE relies on the X Window System, the end-point for the visualization has to be fixed statically before starting the parallel application. That is, the X Server to use for display must be set prior to execution and can not be changed at runtime. Application sharing solutions like XTV or Xmove can alleviate this, but require that additional end-points also run the X Window System. Our window sharing system is more flexible, since the end-points are bound dynamically on-demand, and allows windows to be shared with computers running both Linux (X Window System) and Mac OS X. To visualize 3D data on display walls, software like Chromium [15] can be used. There is no concept of sharing visualizations in Chromium.

The first work on multiple cursors was Engelbart and English' paper from 1968 [16], where the mouse was introduced as an input device. One user had a controlling mouse, while the remaining users had mice that could only be used for pointing, and not interacting. Time-sharing the system cursor is used in [17], where a multi-cursor window manager similar to our own is presented. Their implementation adds a cursor ID to unused bits in the X event structure, which limits the number of cursors to seven. Multi-cursor events are then handled by removing the cursor ID and re-sending the cursor event to the X server as a regular system cursor event. Our implementation does not limit the number of cursors and does not require events to pass through the X server more than once. The Multi-Pointer X Server, MPX [18], integrates support at the hardware layer for several cursors, driven by mice connected to the computer running the X server. Presently, MPX only supports a single keyboard.

3 Model, Design and Implementation

In VNC, clients pull a single desktop from a VNC server. Our window sharing system is based on the publish-subscribe model. A publisher shares one or more windows through a publishing service. Subscribers access shared windows by connecting to the service, from which they can select the windows they are interested in and display them to the user. The service notifies subscribers when new windows are published, or old windows removed.

The service is realized using one or more servers, with publishers and subscribers acting as clients. The servers support network discovery using multicast, allowing clients to discover them on a LAN. Clients connect to servers over TCP for publishing or subscribing to shared windows. Subscribing clients are realized as separate processes, and receive updates to windows they subscribe to after requesting an update from the publishing client. For parallel applications, the publishing client is realized as a thread inside the parallel application. For desktop applications, the publishing client is realized as a separate process. Windows are shared using their pixel representation.

To illustrate how the window sharing system works in a parallel application, we added it to a parallel solver for the Mandelbrot fractal set. The solver on each node originally worked by displaying its part of the solution when all the nodes were done. In our modified version, the solver begins by creating a shared window. The shared window is maintained by a separate thread, and instead of displaying the solution when all nodes are done, the thread reads pixel data from memory, and sends an update to the shared window.

We have implemented window sharing for desktop applications on Mac OS X, allowing Mac OS X windows to be published to subscribers running on Mac OS X and Linux desktops, including the Linux-based display wall desktop. No changes to desktop applications are required in order to share their windows. On Mac OS X, each window is backed by a memory buffer that contains the most up-to-date window contents. Sharing such a window amounts to transmitting the contents of that buffer to subscribers. We use a polling approach on the buffer, as the OS does not notify the publisher when there are changes to other applications' windows. The user can configure the publisher to either send everything or detect changes in the buffer. The decision of which to use will impact the publisher's CPU and bandwidth usage. For static windows, change detection will reduce the bandwidth required for keeping subscribers updated, whereas for a window that is frequently updated, the bandwidth savings will be very small. Change detection is a very costly operation, as it requires calculating a diff between the last contents sent to subscribers, and the current version of the window. The OS X window sharing implementation is further detailed in [19].

The multiple cursor model is based on a service that handles cursor and keyboard input from a number of different users. Users push input events to the service, and the service is responsible for forwarding them from users to applications running on the desktop the service adds multi-cursor support to.

The service is realized as a server. The server runs in a thread, which in turn resides in the same process as a window manager for the X Window System.

Input from users is sent to the server via clients that run on the users' desktops. For each client, the server creates a cursor that is visible on the multi-cursor enabled desktop.

To emulate support for multiple cursors in the single-cursor X Window System environment, the system cursor is time-shared. For instance, when a user clicks his mouse button, the system cursor is moved to the position of that user's virtual cursor, and a mouse click event is posted. For applications, this creates the illusion that a single user is working on the desktop, when in reality there are several. For users, the illusion of several cursors supported by the window system is created. This approach is similar to the one taken in [17].

We implemented the design by incorporating the server thread in the Window Maker[1] window manager. Each client connects to the server over TCP, and is assigned a "virtual cursor." The virtual cursor maintains state associated with the client (such as current focus window and TCP socket information), and provides the actual cursor visible on the desktop to the user. The virtual cursor is drawn by creating an X Window, and modifying the window's appearance to match that of a cursor using the XShape extension. Different cursors are assigned different colors, and input events are posted using the XTestExtension.

4 Experiments

The hardware used for the experiments was (i) a 28-node cluster (Intel P4 EM64T, 3.2 GHz, 2GB RAM, hyperthreading enabled) running Rocks 3.3[2], (ii) a PowerMac Dual-G5 (2.5 GHz, 4GB RAM) running Mac OS X 10.4.2, (iii) a stand-alone PC (identical hardware configuration as the cluster nodes) running RedHat Enterprise Linux 4, (iv) a display wall (28 tiles, 1024x768 resolution per tile) with a combined resolution of 7168x3072, and (v) a GigaBit Ethernet. The cluster nodes are connected to a switch, and the remaining computers are connected to a second switch, with a single link joining the two switches. The cluster nodes also drive the individual tiles of the display wall.

We evaluated the impact of the window sharing system on parallel application performance by sharing windows from a parallel solver for the Mandelbrot fractal set. We measured the execution time of the parallel application both with and without window sharing running on the 28-node cluster. When window sharing was enabled, each node running the computation shared one window each, and each window had one subscriber. All the subscribers ran on the RedHat box. The experiment was repeated five times without window sharing, and five times with. The execution time for the Mandelbrot computation when running without window sharing was between 64.63 and 64.78 seconds, while the execution time when running with window sharing was between 65.40 and 66.19 seconds - an average increase of 1.38%.

We measured the performance of desktop application window sharing by sharing a window on the PowerMac G5 sized at 508x519 pixels in 32-bit color. The

[1] http://www.windowmaker.org/
[2] http://www.rocksclusters.org/

window contained an animation that updated at 30 frames per second (fps). The PowerMac G5 shared the window with subscribers running on the 28-node cluster. We conducted experiments varying the number of subscribers from 1 to 28, running each subscriber on a separate cluster node. We also conducted an experiment with 56 subscribers, where each node ran two subscribers. The Mac OS X publisher was configured to update the shared window at 30 fps without change detection, ideally reaching 30 fps at each subscriber. We measured the publisher's CPU load, the publisher's bandwidth usage and the number of frames received per second by each subscriber.

Figure 3 relates the publisher's CPU load with the publisher's bandwidth usage, with an increasing number of subscribers to the shared window. There is a clear correlation between CPU load and bandwidth usage, both steadily increasing until leveling out at about ten subscribers. At this level and beyond, the network is saturated, while the publisher still has available processing resources to handle additional subscribers.

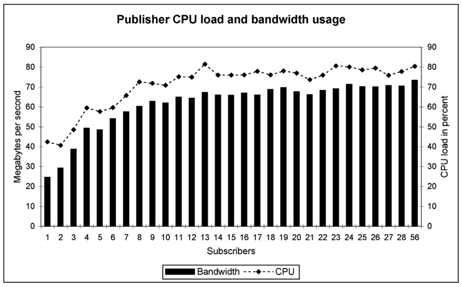


Fig. 3. The publisher's CPU load and bandwidth usage. For 1 to 28 subscribers, one subscriber runs on each node, while for 56 subscribers, two subscribers run on each node.

Figure 4 shows the average frame rate at the subscribers. With a single subscriber, about 28 fps is achieved, with 56 subscribers, the frame rate is 1.6. In general, doubling the number of subscribers cuts the frame rate approximately in half.

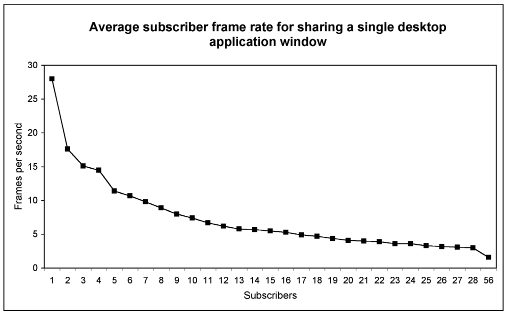


Fig. 4. Average frame rate as seen by each subscriber for sharing a single desktop application window

For the first few subscribers, Figure 3 shows that the publisher has available processing resources and network bandwidth but is unable to provide the subscribers with updates sufficiently fast to reach the target frame rate. There are two factors that contribute to this behaviour. First, subscribers must request updates from the publisher in order to receive them. If the subscribers do not do this sufficiently fast, the resulting frame rate will be lower. Second, the publisher accepts requests and processes them in batches. Each iteration is started by a timer that fires in a best-effort manner. If an iteration exceeds the timer interval, the next iteration starts late, resulting in a lower frame rate.

We also conducted an experiment comparing the publisher's two update modes. The publisher can either send everything for each iteration, or calculate a diff between the current window contents, and the window contents most recently sent to subscribers (change detection). We measured the publisher's CPU load, bandwidth usage and frame rate at each subscriber.

Performing change detection was very costly. With one subscriber, the frame rate was 7.13, and the publisher's CPU load at 76.2%, transmitting 5.8 MB/s. With 20 subscribers, the frame rate was only 3.7, and publisher CPU load was at 100.1%[3] with bandwidth usage at 50 MB/s. In comparison, sending everything to a single subscriber gave a frame rate of 28, with publisher CPU load at 42.5% and bandwidth usage at 24.5 MB/s. To 20 subscribers, the publisher CPU load was 77%, transmitting 67.6 MB/s, and the frame rate was 4.1. We have performed informal practical experiments with the multi-cursor implementation, testing it with up to 8 simultaneous cursors.

5 Discussion

The experiments indicate that the impact on performance from adding shared windows to a parallel application is low. The benchmark was run without load balancing, neither on the system level nor the application level. For the Mandelbrot computation, this results in a very uneven load distribution, which on many nodes result in ample time for the window sharing system to execute in. This may contribute to hiding additional overhead from the window sharing system.

The system shares windows by sharing their pixel representation. This is the simplest way of sharing windows between hardware platforms and different window systems, and is the same approach as that taken by VNC [4]. The alternative to sharing pixels is to share drawing operations, like "fill rectangle" or "draw line." This approach is more challenging to make platform independent, compared to the simple operation of copying a block of pixels and sending them across the wire. As an example, drawing operations can save bandwidth by transmitting the raw text rather than the pixels making up the text on a display.

The publisher running on Mac OS X can use two different strategies when sending updates to windows. Since it doesn't know when or which regions of a window is updated, it can decide to either always send everything to everyone, or compute a diff between what the subscriber already has, and the current contents of the window. The trade-off is between publisher CPU load and publisher bandwidth. Subscribers will also potentially have fewer updates to draw, resulting in lower subscriber CPU load. Sending everything consumes more bandwidth, but incurs a lower CPU load on the computer running the publisher, while performing change detection is very costly. In contrast, for the Mandelbrot application, window contents are only sent when there are actual updates to the window. This

[3] The publisher ran on the PowerMac, which has two CPUs.

is possible since the window sharing code has been built directly into the Mandelbrot application, allowing it to send updates only when the shared window is actually updated.

Integrating the window sharing system with parallel applications requires that the source code for the parallel application is available. This is not a major problem though, since the window sharing system only simplifies the task of publishing the windows. The application itself is responsible for filling that window with meaningful content - be it a runtime visualization or performance monitoring data. This will require further modifications to the application's code.

6 Conclusion

This paper presents a system that shares windows created by parallel and desktop applications between two or more users. To further enhance collaboration, a system supporting multiple cursors on a wall-sized, high-resolution display is used to allow many users to manipulate shared windows simultaneously.

Parallel applications require modifications to their source code in order to share windows and discover updates in them. For desktop applications, modifying their source code is usually neither practical nor possible. Because of this, sharing desktop application windows is more costly, both in terms of CPU and network load, as the system has to poll window contents in order to discover updates.

We have integrated the system with a parallel implementation of a solver for the Mandelbrot fractal, and measured its impact on the application's execution time. We measured the performance of desktop application window sharing by sharing a single window containing an animation with a varying number of users. The windows were displayed on the tiled display wall. The multi-cursor system was used with eight cursors.

We found that the addition of shared windows to the Mandelbrot application only added about 1.4% to the application's execution time. For this benchmark, no load balancing was used, resulting in an uneven distribution of work between the different processes. This gives the window sharing system CPU time to execute in that would otherwise remain unused, which combined with the Mandelbrot application's infrequent updates, explains the window sharing system's low impact on the application's performance. A higher update frequency would likely increase the window sharing system's impact on execution time.

For desktop application windows, the number of updates received per second by each subscriber went from 28 with one subscriber, to 1.6 with 56 subscribers. In general, when the number of subscribers is doubled, the update frequency seen by each subscriber is halved. For less than ten subscribers, CPU and network are not the limiting resources. Instead, the scaling behaviour is caused by the iteration- and timer-based approach used by the publisher. With many subscribers, the limiting resource is the network.

Sharing windows and support for multiple cursors are promising for increasing the flexibility of runtime visualization and monitoring of parallel applications, and for collaboration using desktop applications. More work remains to determine the window sharing system's impact on these issues, and to better characterize the system's performance.

Acknowledgements. Thanks to Lars Ailo Bongo, Vera Göbel, Espen Skjelnes Johnsen, Tore Larsen, Kai Li and Grant Wallace for their discussions, suggestions, help and support. This work has been supported by the NFR funded project No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices.

References

1. Chan, A., Gropp, W., Lusk, E.: MPE: MPI Parallel Environment, `http://www-unix.mcs.anl.gov/perfvis/software/MPE/index.htm`
2. Scheifler, R.W., Gettys, J.: The X Window System. ACM Trans. Graph. 5(2), 79–109 (1986)
3. Faith, R.E., Martin, K.E.: Xdmx: Distributed, multi-head X, `http://dmx.sourceforge.net/`
4. Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A.: Virtual Network Computing. IEEE Internet Computing 2(1) (January 1998)
5. NoMachine. NX server and client, `http://www.nomachine.com/`
6. Baratto, R.A., Kim, L.N., Nieh, J.: Thinc: a virtual display architecture for thin-client computing. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, pp. 277–290. ACM Press, New York (2005)
7. Wallace, G.: SharedAppVNC, `http://shared-app-vnc.sourceforge.net/`
8. Microsoft Corporation: Netmeeting, `http://www.microsoft.com/windows/netmeeting/`
9. Citrix: Citrix Presentation Server, `http://www.citrix.com/`
10. Tan, D.S., Meyers, B., Czerwinski, M.: WinCuts: Manipulating arbitrary window regions for more effective use of screen space. In: CHI '04 extended abstracts on Human factors in computing systems, pp. 1525–1528. ACM Press, New York (2004)
11. Abdel-Wahab, H., Feit, M.: XTV: A Framework for Sharing X Window Clients in Remote Synchrounous Collaboration. IEEE Tricomm (April 1991)
12. Solomita, E., Kempf, J., Duchamp, D.: XMove: A Pseudoserver for X Window Movement. The X Resource 11(1), 143–170 (1994)
13. Lewis, G.J., Hasan, S.M., Alexandrov, V.N., Dove, M.T., Calleja, M.: Multicast application sharing tool - Facilitating the eMinerals virtual organisation. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2005. LNCS, vol. 3516, pp. 359–366. Springer, Heidelberg (2005)
14. The AccessGrid Project website, `http://www.accessgrid.org/`
15. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: A stream-processing framework for interactive rendering on clusters. In: SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, pp. 693–702. ACM Press, New York (2002)
16. Engelbart, D.C., English, W.K.: A research center for augmenting human intellect. In: AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference, pp. 395–410 (December 1968)

17. Wallace, G., Bi, P., Li, K., Anshus, O.: A MultiCursor X Window Manager Supporting Control Room Collaboration. Technical Report TR-707-04, Princeton University, Computer Science (July 2004)
18. Hutterer, P.: The Multi-Pointer X Server, MPX, `http://wearables.unisa.edu.au/mpx/`
19. Stødle, D., Bjørndalen, J.M., Anshus, O.J.: Collaborative sharing of windows between Mac OS X, the X Window System and Windows. In: Norsk informatikkonferanse 2004, NIK 04 (2004)

Tools, Frameworks and Applications for High Performance Computing: Minisymposium Abstract

Osni Marques

Lawrence Berkeley National Laboratory, Berkeley, USA

The development of high performance simulation codes is often a demanding process, due to the complexity of the phenomena to be simulated but also to the proliferation and evolution of computer architectures. The success of such efforts is dictated by the time required to achieve a functional prototype code of the application, and then an optimized production version of the code. Simultaneously, achieving an optimal usage of the available, and frequently scarce, computational resources is of major importance to developers and users of simulation codes. To achieve these goals, it is often necessary to develop or integrate algorithms, models, and computational techniques from a group of collaborators with diverse expertise.

The purpose of this minisymposium is twofold. Firstly, it focuses on a set of cutting edge software tools and frameworks currently used to tackle scientific computing applications. The availability of advanced tools like the ones to be showcased has enabled more complex physical phenomena to be addressed and as a result contributed to the growth of the computational sciences community. This community includes scientists, designers and developers of high-end technology who require computerized modeling solutions, portable software libraries, but also convenient interfaces or friendly frameworks. Secondly, the minisymposium includes presentations on a set of challenge applications, ranging from the nano to the cosmic scale, that have important requirements for computing resources but that have also fostered the development of novel techniques and tools.

Multi-level μ-Finite Element Analysis for Human Bone Structures

Peter Arbenz[1], G. Harry van Lenthe[2], Uche Mennel[1], Ralph Müller[2], and Marzio Sala[1]

[1] Institute of Computational Science, ETH Zürich, CH-8092 Zürich`arbenz@inf.ethz.ch`

[2] Institute for Biomechanics, ETH Zürich, CH-8092 Zürich

Abstract. Using microarchitectural bone imaging, it is now possible to assess both the apparent density and the trabecular microstructure of intact bones in a single measurement. In combination with microstructural finite element (μFE) analysis this could provide a powerful tool to improve strength assessment and individual fracture risk prediction. However, the resulting μFE models are very large and require dedicated solution techniques. Therefore, in this paper we investigate the efficient solution of the resulting large systems of linear equations by the preconditioned conjugate gradient algorithm. We detail the implementation strategies that lead to a fully parallel finite element solver. Our numerical results show that a human bone model of about 5 million elements can be solved in about a minute. These short solution times will allow to assess the mechanical quality of bone *in vivo* on a routine basis. Furthermore, our highly scalable solution methods make it possible to analyze the very large models of whole bones measured *in vitro*, which can have up to 1 billion degrees of freedom.

1 Introduction

Osteoporosis is a disease characterized by low bone mass and deterioration of bone microarchitecture. It leads to increased bone fragility and risk of fracture, particularly of the hip, spine, and wrist. Worldwide, lifetime risk for osteoporotic fractures in women is estimated close to 40%; in men risk is 13% [11]. As reported by the World Health Organization, osteoporosis is second only to cardiovascular disease as a leading health care problem. Osteoporotic fractures are a major cause of severe long-term pain and physical disability, and have an enormous impact on the individual, society and health care systems. For the clinician, the prediction of bone quality for individual patients is, so far, more or less restricted to the quantitative analysis of bone density alone, although there is convincing evidence that bone microarchitecture plays a significant role as well.

With the advent of fast and powerful computers, simulation techniques are becoming popular for investigating the mechanical properties of bones and predicting the strength of a given patient's bones. Ideally, the development of a system with microstructural resolution better than 50 μm would allow *in vivo* measurement of patients at different instances in time and at different anatomical sites. Unfortunately, such systems are not yet available, but the resolution at peripheral sites has reached a level (90 micron) that allows elucidation of individual microstructural bone elements. Using this technique, two recent cross-sectional studies have shown sex- and age-related changes in trabecular structure and in cortical thickness of the distal radius [4, 9].

Using a direct voxel-conversion technique the three-dimensional computer reconstructions of bone can be converted to a finite element mesh, that can be used to perform a 'virtual experiment', i.e., to simulate a mechanical test in great detail and with high precision. The resulting procedure is called microstructural finite element (μFE) analysis.

μFE analyses are computationally demanding and require specially adapted solution schemes. For example, a linear analysis of one human vertebral body at a resolution of 30 microns requires over 130 million elements. Therefore, one is forced to use iterative methods, and in particular the preconditioned conjugate gradient algorithm. Element-by-element approaches have been proposed about 10 years ago and are now common [13, 18]. The memory requirements for solving such a problem with the element-by-element preconditioned conjugate gradient (EBE-PCG) algorithm is particularly small, and as such the ratio of element per processor is particularly high. However, a major drawback of this method is its slow convergence and its high sensitivity to discontinuities in the material properties. A better solution consists in using a scalable preconditioner like smoothed aggregation, as suggested by Adams et al. [2].

The aim of this study was to develop a fast multi-level, fully parallel finite element code for the μ-finite element analysis of human bone structures as developed in [12]. We refer to [5] for more details on how to obtain suitable models, and we focus exclusively on the *in silico* simulation. The paper is organized as follows. In Section 2 we describe the mathematical model and the system of linear equations it entails. In Section 3 we introduce smoothed aggregation preconditioners for bone modeling. In Section 4 we give some details on our implementation. Numerical results are reported in Section 5. Conclusions are drawn in Section 6.

2 Mathematical Model

The basic mathematical model of the problem is given by the Lamé equations of elasticity [5]. The weak formulation of the 3D problem is:

$$\begin{cases} \text{Find } \mathbf{u} \in V \text{ such that} \\ \int\limits_{\Omega} [2\mu\, \varepsilon(\mathbf{u}) : \varepsilon(\mathbf{v}) + \lambda\, \div \mathbf{u}\, \div \mathbf{v}]\, d\Omega = \int\limits_{\Omega} \mathbf{f}^T \mathbf{v}\, d\Omega + \int\limits_{\Gamma_N} \mathbf{g}_S^T \mathbf{v}\, d\Gamma, \quad \forall \mathbf{v} \in V, \end{cases} \quad (1)$$

where $V \subset (H^1_{\Gamma_D}(\Omega))^3$ is a Sobolev space, Γ_D and Γ_N are the Dirichlet and Neumann part of $\partial\Omega$, $\overline{\Gamma_D \cup \Gamma_N} = \overline{\partial\Omega}$, $\Gamma_D \cap \Gamma_N = \emptyset$, and $\mathbf{u}$ describes the nodal displacements, λ and μ are the Lamé constants, $\mathbf{f}$ the volume forces, and $\mathbf{g}$ the boundary tractions. The symmetric strains in (1) are given by

$$\varepsilon(\mathbf{u}) := \frac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^T).$$

The finite element discretization of problem (1) by means of piecewise trilinear polynomials [18] leads to a the linear algebraic system

$$A\mathbf{u} = \mathbf{f}, \tag{2}$$

where A is symmetric and positive definite as long as $\Gamma_D \neq \emptyset$.

Equation (2) is solved with a conjugate gradient method. It is well-known that $\kappa(A) \approx \mathcal{O}(h^{-2})$, where $\kappa(A)$ is the condition number of A and h is the mesh size [5]. Therefore, the linear system

$$A\,B^{-1}\mathbf{x} = \mathbf{f}, \qquad \mathbf{u} = B^{-1}\mathbf{x}, \tag{3}$$

is solved instead of (2), where B is the *preconditioning* matrix, chosen so that $\kappa(A\,B^{-1}) \ll \kappa(A)$. The conventional preconditioner adopted for bone modeling problems is the element-by-element preconditioner [13]. Here, instead, we consider scalable *multilevel* (or *multigrid*) preconditioners.

3 Scalable Preconditioners for Bone Modeling

Multigrid methods were introduced in the late 70's, and their success and development is testified by the vast literature and the many international conferences organized since then, see [19] for a survey.

A multilevel method tries to approximate the original PDE problem of interest on a hierarchy of levels and use 'solutions' from coarse levels to accelerate the convergence on the finest level. When used to define preconditioners for linear systems, multigrid methods are applied using the so-called V-cycle, which reads as follows:

```
 1. procedure MultiLevelSolve($A_\ell$, $\mathbf{b}_\ell$, $\mathbf{x}_\ell$, $\ell$)
 2.     if $\ell == L-1$ then
 3.         $\mathbf{x}_\ell = A_\ell^{-1}\mathbf{b}_\ell$
 4.     else
 5.         $\mathbf{x}_\ell = S_\ell(A_\ell, \mathbf{b}_\ell, \mathbf{0})$
 6.         $\mathbf{r}_\ell = \mathbf{b}_\ell - A_\ell\,\mathbf{x}_\ell$;    $\mathbf{b}_{\ell+1} = R_\ell\,\mathbf{r}_\ell$
 7.         $\mathbf{v}_{\ell+1} = \mathbf{0}$
 8.         MultiLevelSolve($A_{\ell+1}, \mathbf{b}_{\ell+1}, \mathbf{v}_{\ell+1}, \ell+1$)
 9.         $\mathbf{x}_\ell = \mathbf{x}_\ell + P_\ell\,\mathbf{v}_{\ell+1}$
10.         $\mathbf{x}_\ell = S_\ell(A_\ell, \mathbf{b}_\ell, \mathbf{x}_\ell)$
11.     endif
12. endprocedure
```

In the procedure, $\ell = 0$ defines the finest of the L levels, A_ℓ represents the discretization on level ℓ of the problem, P_ℓ and R_ℓ are prolongator and restriction operator from level $\ell + 1$ to ℓ, respectively, and the S_ℓ are approximate solvers (called smoothers).

The preconditioner for the solution of linear system (2) is applied by calling MultiLevelSolve$(A, \mathbf{b}, \mathbf{x}, 0)$. $\mathbf{x}_\ell$ is the starting solution, $\mathbf{b}_\ell$ the right-hand side, and the S_ℓ's are smoothers.

The key aspect in algebraic multigrid methods is the definition of the auxiliary operators P_ℓ, R_ℓ, and A_ℓ. At least two alternative approaches have been evolved in the literature: to algebraically coarsen on each level by identifying a set of coarser-level nodes (the so-called C-nodes) and finer-level nodes (F-nodes) [14], or to algebraically coarsen on each level by grouping the nodes into contiguous subsets, called aggregates, as done in smoothed aggregation (SA) [21].

In this paper we focus on SA preconditioners. Theoretical results support the usage of SA for linear elasticity problems [1]. Also, SA preconditioners have been proven scalable up to thousands of processors [10, 2]. A general setup procedure for smoothed aggregation multigrid is reported below.

```
 1. Define the maximum number of levels, $L$
 2. for each level $\ell$ do
 3.     if on coarsest level then
 4.         Define the coarse solver
 5.         Return
 6.     endif
 7.     Define $P_\ell$
 8.     $R_\ell = P_\ell^T$
 9.     $A_{\ell+1} = R_\ell A_\ell P_\ell$
10.     Define the smoother $S_\ell$
11. endfor
```

The construction of a SA preconditioner is now briefly outlined. First, a graph is built from the linear system. This graph contains an edge between the vertices i and j, if $A_{i,j} \neq 0$. For vector problems as the one considered in this paper, the graph is defined in a block fashion, meaning that one graph vertex is associated with all the unknowns at a grid vertex. An edge between two graph vertices i and j is added if there are any nonzeros in the block matrix defined by the i^{th} block row and j^{th} block column. Then, graph vertices are grouped into contiguous subsets, called *aggregates*. Each aggregate effectively represents a coarser grid vertex. Once the coarser grid is determined, a grid transfer must be defined. The simplest possible grid transfer is to use low-energy modes (in our case, the rigid body modes as obtained when no boundary conditions are applied), that are 'chopped' and inserted within the $(i,j)^{th}$ block if the i^{th} fine grid point has been assigned to the j^{th} aggregate.

There are many ways to define aggregates. In a standard algebraic multigrid method the most common way to create an aggregate is via some kind of greedy graph algorithm where an initial node is chosen along with all of its nearest neighbors. The net affect of this type of procedure is to produce aggregates which are 'ball-like' with an approximate diameter of three graph vertices. Unfortunately, this type of procedure leads to many aggregates and must be repeated several times in order to obtain a coarse matrix that is small enough to be efficiently solved by a direct solver. An alternative approach is to adopt a graph partitioner code like METIS or ParMETIS [8] to define the aggregates. The algorithms within these packages split the fine matrix so that each partition has no more nodes than a user supplied parameter. All the nodes in each partition are then combined to form a single aggregate. METIS is a serial code and so can only be used to partition the graph corresponding to the local matrix entries within a processor. This implies that no aggregate can span more than one processor. ParMETIS is a parallel code and so it can be applied to the entire distributed global graph to produce aggregates that span more than one processor.

The procedure we have outlined is sometimes referred to as *nonsmoothed aggregation*. For elliptic problems as the one considered in this paper, the so-called *smoothed aggregation* performs significantly better [21]. In this latter case, the (tentative) prolongator defined by nonsmoothed aggregation $P_{0,\ell}$ at a given level ℓ is smoothed with one step of a damped Jacobi iteration to obtain the final prolongator,

$$P_\ell = (I_\ell - \omega_\ell \, D_\ell^{-1} A_\ell) \, P_{0,\ell},$$

where $A_\ell \in \mathbb{R}^{n_\ell \times n_\ell}$, I_ℓ is the identity matrix of size n_ℓ, $D_\ell = \text{diag}(A_\ell)$, and ω_ℓ is a damping parameter, typically defined as

$$\omega_\ell = \frac{4/3}{\lambda_{\max}(D_\ell^{-1} A_\ell)}, \tag{4}$$

where $\lambda_{\max}(D_\ell^{-1} A_\ell)$ indicates the largest eigenvalue of $D_\ell^{-1} A_\ell$.

We still need to define the smoothers and the coarse solver S_ℓ. Common choices for the smoother are Chebyshev polynomial smoothers or processor-based symmetric Gauss-Seidel [3]. A direct solver is applied to the coarse problem.

4 Implementation Details

The code used to obtain the numerical results outlined in the next section has been developed for massively parallel, distributed memory architectures. The code is written in C++ with support for MPI, and interfaces with optimized BLAS and LAPACK for computationally intensive tasks on dense matrices and vectors. All sparse linear algebra objects, in contrast, are based on the Epetra library [7]. Epetra furnishes an efficient and flexible implementation of fundamental distributed linear algebra objects like vectors, multivectors and sparse matrices. Epetra objects are accepted by the Krylov accelerator solvers of the AztecOO library [20] and by the multilevel preconditioning package ML [17]. The aggregation scheme and construction of restriction and prolongation operators is implemented within the ML library. ML offers a variety of coarsening options, as well as several smoothers and coarse solvers, either developed within ML itself, or as part of the IFPACK [16] and Amesos [15] packages. Epetra, AztecOO, Amesos, IFPACK and ML are available through the Trilinos framework [7].

While Trilinos is used for most linear algebra operations, the I/O is performed by the HDF5 library [6]. HDF5 furnishes binary parallel I/O, meaning that all processors[1] can concurrently access the file by using so-called *hyperslabs*, which in our case are defined as contiguous chunks of data. The approach we follow is therefore as follows.

For a typical finite element code, the data structures to be read are the grid connectivity and the vertex coordinates, both specified as FORTRAN arrays, and a list of boundary vertices or faces. In order to obtain a fully parallel code, it is mandatory to parallelize the input and output phases. First, we read all the grid data by using the hyperslab technique and a linear distribution. Being highly unbalanced, this data distribution is not suitable for actual computations. Therefore, we first use the parallel graph partitioner ParMETIS [8] to compute a load-balanced partition, then we redistribute the data structures to match the balanced layout. This is accomplished by resorting to the Import/Export capabilities of the Epetra package. A rough ParMETIS space filling curve partioning is prepended to balance the required graph construction [8].

5 Numerical Experiments

The results have been obtained on the CRAY XT3, located at the Swiss National Supercomputing Centre (CSCS), containing 1100 AMD Opteron single-core processors that run at 2.6 GHz and are equipped with 2GB of memory. The processors are connected by the Cray SeaStar high-bandwidth (4 GB/s sustained), low-latency interconnect.

The linear system (2) has been solved up to a tolerance of 10^{-5} in the relative residual, with a zero initial vector. The multilevel preconditioner is built using METIS as the aggregation scheme. A target value of 50 vertices per aggregate was adopted. Smoothed aggregation is used, and ten iterations of the (non-preconditioned) conjugate gradient algorithm are adopted to estimate ω_ℓ in (4). The Chebyshev smoother MLS [3] was used on all levels but the coarsest, where the direct solver KLU from the Amesos package [15] is employed.

We now report on some numerical results to assess the scalability of the approach presented in this paper. First, we consider *weak scalability*, meaning that we keep the problem size per processor constant. Therefore, the overall problem size is proportional to the number of processor involved in the computation. A perfectly (weakly) scalable code executes in a time that is independent of the processors employed.

We performed a weak scalability test with an artificial problem. A small piece of trabecular bone was artificially taken from a piece of human trabecular bone,

[1] The actual parallel performances of HDF5 depends on the underlying I/O system, and on the number of installed I/O nodes, which is usually much smaller than that of the computing nodes.

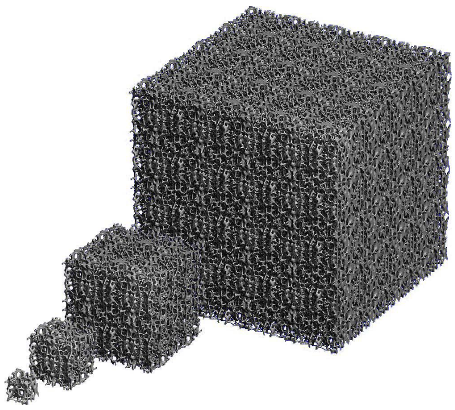


Fig. 1. Bone tissue models generated by 3D mirroring to test weak scalability. The displayed models are the ones run with 1, 8, 27, and 64 processors. The larger models are not shown.

scanned with a high resolution micro-CT system. This small piece has been mirrored in three dimensions to obtain arbitrary sized cubic models as illustrated in Fig. 1. The number of processors p is adjusted to the cubic problem sizes, i.e., $p = n^3$ where n ranges from 1 to 9.

For a problem consisting of $60,482 \times p$ elements the execution times and iteration count of the SA preconditioned conjugate gradient algorithm are given in Table 1 and Fig. 2. Note that the 200M degrees of freedom test is solved in less than 100 seconds. Also note that all phases but the I/O scale almost perfectly. The poor scaling of I/O is probably due to the limited number of I/O nodes.

The second set of tests aims at analyzing the *strong scalability*, in which the same problem is solved by a varying number of processors. With an ideally (strongly) scalable code the execution times are reduced according to the number

Table 1. Execution times (in seconds). efficiency, and number of PCG iterations of the weak scalability test on the Cray XT3.

CPUs	input	repart.	assembly	precond.	solution	output	total	efficiency	iters
1	1.25	2.28	6.25	8.58	28.86	0.10	47.32	100%	51
8	1.27	3.84	6.64	9.03	30.98	0.52	52.28	91%	53
27	2.00	4.18	7.03	9.67	34.23	0.78	57.88	82%	56
64	3.65	4.20	7.12	10.05	32.60	1.33	58.94	80%	53
125	5.03	4.78	7.26	15.86	32.71	2.33	67.97	70%	52
216	8.23	4.92	7.26	15.91	32.34	3.81	72.47	65%	51
343	9.58	5.27	7.38	16.09	31.64	5.25	75.21	62%	49
512	17.34	5.39	7.29	17.04	30.24	8.03	85.33	55%	47
729	20.98	6.18	7.36	23.98	30.24	11.05	99.78	47%	45

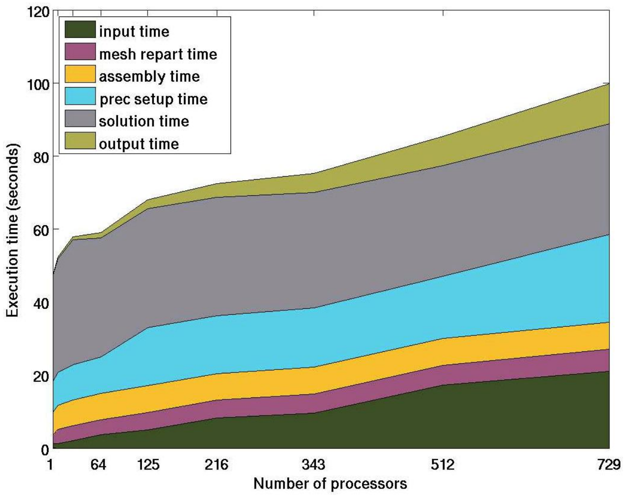


Fig. 2. Results of weak scalability test on the Cray XT3

Table 2. Execution times (in seconds) and number of PCG iterations of the strong scalability test on the Cray XT3

CPUs	input	repart.	assembly	precond.	solution	output	total	efficiency	iters
72	4.10	8.47	10.2	19.3	90.1	2.22	134.	100%	117
128	4.62	4.73	5.97	9.36	54.8	2.08	81.6	92%	120
256	8.16	2.47	3.09	6.75	28.6	3.11	52.2	72%	118
512	10.8	1.74	1.56	4.61	16.0	5.18	39.8	47%	118
768	15.5	2.13	1.04	5.02	12.0	7.58	43.3	29%	117

of processors employed, i.e., $T_p = T_{\bar{p}}\bar{p}/p$, where $\bar{p}$ is a reference number of processors and T_p is the execution time on p processors. Note that for a typical problem, the number of inter-processor communication increases with the number of processors, while the amount of local computation decreases. This makes strong scalability much harder to achieve than weak scalability. Nevertheless, a strong scalability test is useful to determine the optimal number of processors that should be used for a given problem size.

For the strong scalability test, we have considered a problem reflecting the distal part (of 20% of the length) of a human radius, that was scanned with a new generation high-resolution 3D peripheral quantitative computed tomography (pQCT) scanner (Scanco Medical, Bassersdorf, Switzerland) providing a resolution of 93 μm, cf. Fig 3. The model consists of 5.44 million elements which makes a minimal number of 72 processors necessary to solve it. Table 2 and Fig. 4 give execution times and iteration counts. The corresponding numbers show that the whole system is solved with high efficiency (>70%) on up to 256

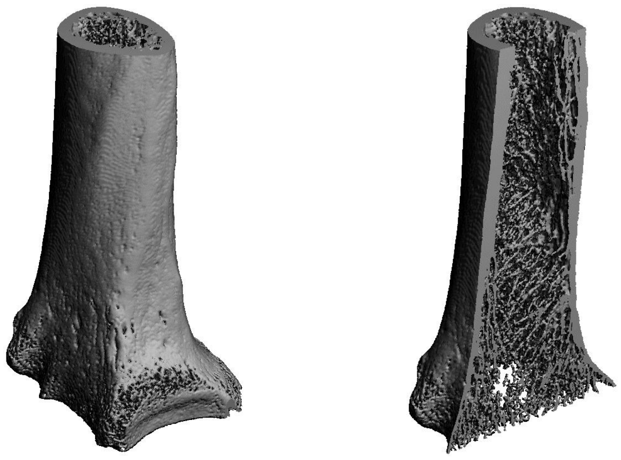


Fig. 3. Distal part (20% of the length) of the radius in a human forearm

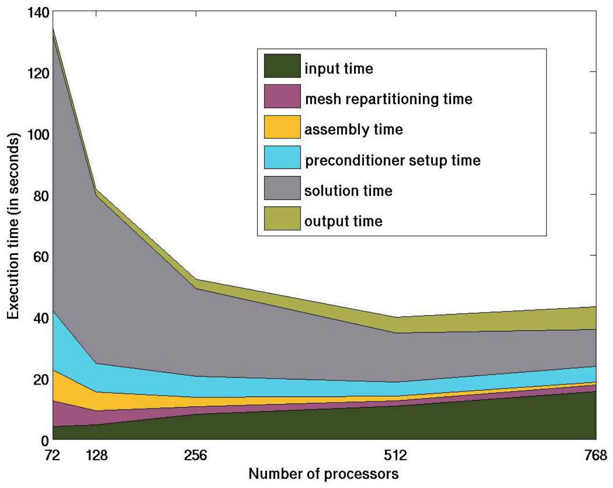


Fig. 4. Results of strong scalability test on the Cray XT3

processors. The efficiency is mostly reduced due to I/O. Note that less than 60 seconds are required for solving this problem with 256 CPUs, and only about 130 when using 72 CPUs.

6 Conclusions

We have described the model and the implementation details for multi-level μFE analysis of human bone structures. By using the presented techniques, a fully-parallel finite element code is obtained. Numerical results show excellent weak and strong scalability for both an artificial and a realistic human bone problem.

The short solution times which we have obtained will allow to assess the mechanical quality of bone *in vivo* on a routinely basis. Furthermore, the even larger models of whole bones measured *in vitro* has become possible. We expect these findings to improve our understanding of the influence of densitometric, morphological and loading factors in the etiology of spontaneous fractures of the hip and the spine.

Acknowledgments

The computations on the Cray XT3 have been performed in the framework of a Large User Project grant of the Swiss National Supercomputing Centre (CSCS), Manno, Switzerland. We acknowledge the help of the XT3 support team.

References

1. Adams, M.: Evaluation of three unstructured multigrid methods on 3D finite element problems in solid mechanics. Internat. J. Numer. Methods Engrg. 55(5), 519–534 (2002)
2. Adams, M.F., Bayraktar, H.H., Keaveny, T.M., Papadopoulos, P.: Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In: ACM/IEEE Proceedings of SC2004: High Performance Networking and Computing (2004)
3. Adams, M., Brezina, M., Hu, J., Tuminaro, R.: Parallel multigrid smoothing: polynomial versus Gauss–Seidel. J. Comput. Phys. 188(2), 593–610 (2003)
4. Boutroy, S., Bouxsein, M.L., Munoz, F., et al.: In vivo assessment of trabecular bone microarchitecture by high-resolution peripheral quantitative computed tomography. J. Clin. Endocrinol. Metab. 90(12), 6508–6515 (2005)
5. Ciarlet, P.G.: Three-dimensional elasticity. Studies in mathematics and its applications, vol. 20. North Holland, Amsterdam (1988)
6. HDF5: Hierarchical Data Format. Reference Manual and User's Guide are available from `http://hdf.ncsa.uiuc.edu/HDF5/doc/`
7. Heroux, M.A., et al.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31(3), 397–423 (2005)
8. Karypis, G., Kumar, V.: METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical Report, University of Minnesota, Department of Computer Science (1998)
9. Khosla, S., Riggs, B.L., Atkinson, E.J., et al.: Effects of sex and age on bone microstructure at the ultradistal radius: a population-based noninvasive in vivo assessment. J. Bone Miner. Res. 21(1), 124–131 (2006)
10. Lin, P.T., Sala, M., Shadid, J.N., Tuminaro, R.S.: Performance of fully coupled algebraic multilevel domain decomposition preconditioners for incompressible flow and transport. Internat. J. Numer. Meth. Engrg. 67, 208–225 (2006)
11. Melton III, L., Chrischilles, E., Cooper, C., Lane, A., Riggs, B.: Perspective. How many women have osteoporosis? J. Bone Miner. Res. 7, 1005–1010 (1992)
12. Mennel, U.: A multilevel PCG algorithm for the μ-FE analysis of human bone structures. Master thesis, ETH Zürich, Institute of Computational Science (2006)

13. van Rietbergen, B., Weinans, H., Huiskes, R., Polman, B.J.W.: Computational strategies for iterative solutions of large FEM applications employing voxel data. Internat. J. Numer. Methods Engrg. 39(16), 2743–2767 (1996)
14. Ruge, J., Stüben, K.: Algebraic multigrid (AMG). In: McCormick, S. (ed.) Multigrid Methods, Frontiers in Applied Mathematics, vol. 3, SIAM, Philadelphia (1987)
15. Sala, M.: Amesos 2.0 reference guide. Technical Report SAND-4820, Sandia National Laboratories (2004)
16. Sala, M., Heroux, M.: Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories (2005)
17. Sala, M., Hu, J., Tuminaro, R.: ML 3.1 smoothed aggregation user's guide. Technical Report SAND-4819, Sandia National Laboratories (2004)
18. Smith, I.M., Griffiths, D.V.: Programming the Finite Element Method. John Wiley, New York (1998)
19. Trottenberg, U., Oosterlee, C.W., Schüller, A.: Multigrid. Academic Press, London (2000)
20. Tuminaro, R., Heroux, M., Hutchinson, S., Shadid, J.: Official Aztec user's guide: Version 2.1. Technical Report Sand99-8801J, Sandia National Laboratories (1999)
21. Vaněk, P., Brezina, M., Mandel, J.: Convergence of algebraic multigrid based on smoothed aggregation. Numer. Math. 88(3), 559–579 (2001)

High-Level User Interfaces for the DOE ACTS Collection

L. Anthony Drummond[1], Vicente Galiano[2], Violeta Migallón[3], and José Penadés[3]

[1] Lawrence Berkeley National Laboratory,
One Cyclotron Road, Berkeley CA 94703, United States of America
LADrummond@lbl.gov
[2] Departamento de Física y Arquitectura de Computadores,
Universidad Miguel Hernández, ES-03202 Elche, Alicante, Spain
vgaliano@umh.es
[3] Departamento de Ciencia de la Computación e Inteligencia Artificial,
Universidad de Alicante, ES-03071 Alicante, Spain
{violeta, jpenades}@dccia.ua.es

Abstract. The ACTS collection project comprises a set of state-of-the-art software tools to speed up the development of High-Performance Computing Applications in science and engineering. We look at the development of High Level user interfaces using scripting languages like Python, to facilitate the access to ACTS technology to a wide community of computational scientists. PyACTS is our main project here, but we also visit other efforts within the community of developers of ACTS tools.

1 Introduction

The Advanced CompuTational Software (ACTS) [1] Collection comprises a set of computational tools developed primarily at DOE laboratories, sometimes in collaboration with universities and other funding agencies (NSF, DARPA), aimed at simplifying the solution of common and important computational problems. A number of important scientific problems have been successfully studied and solved by means of computer simulations built on top of tools available in the ACTS Collection [2]. The ACTS Collection brings robust and high-end software tools to the hands of application developers to accelerate the development of computational science codes and consequent results. However, this transfer of technology is not always successful due in part to the intricacy in understanding the interfaces associated with the software tools and the time an application scientists spends installing and learning the use of a given tool. Here we present a set of Python based interfaces to some of the tools in the ACTS Collection, PyACTS. We also present some examples of it applications and future development directions.

2 Some of the Tools in the ACTS Collection

In Table 1 we briefly list some of the numerical functionality available in the ACTS Collection. The tools in Table 1 have development projects that include interfaces in Python. A closer look at the functionality offered by these tools [1], we see that there are some tools that compliment others (i.e., the use of a direct solver inside a preconditioner used by an iterative scheme, or the use of a preconditioner from *Tool A* inside *Tool B*, etc.) and tools with functionality that overlap. Selecting the appropriate tool is not trivial and problem specific, it may require in some cases not only expertise in numerical linear algebra but also extensive testing and tuning. To be able to explore the full functional plethora in the ACTS Collection, a user may spend months learning the different interfaces and parameterizations of a giving tool. Our goal with PyACTS is to build a high level user interface that directly reduces the amount of work a user spends simply learning to use a tool, facilitates a faster development of her or his application.

PyACTS [15,16,17] provides a didactical user interface to assist with their first application prototype and following production code development. Here we look at the PyACTS development project and existing functionalities.
The reader is referred to the ACTS Information Center [18] for more details on these tools and others available in the collection. Our initial work has been focused on the development of PyScaLAPACK which is introduced in the next section.

3 PyACTS: A Python Interface to the ACTS Collection

Python [19] is an interpreted, interactive, object-oriented programming language. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface. Python is designed to make integration with other software components in a system as simple as possible. Programs written in Python can be easily blended with other languages. For instance, Python scripts can call out existing C and C++ libraries, Java classes, and much more. Actually, it is this feature of Python that is employed in our current work.

Additionally, Python is portable: it runs on many brands of UNIX, on Windows, Mac, and many other platforms. Python is copyrighted but freely usable and distributable, even for commercial use. Python is an ideal language for prototype development and other ad–hoc programming tasks, without compromising maintainability and it uses an elegant syntax for readable programs. All of the ACTS tools listed in the previous section use MPI as one of the methods for supporting message passing. In the PyACTS, we use pyMPI [20], which enables us to use the same Python modules and rich functionality. We have also tested other Python implementations of MPI and these can be replaced without any portability issues because of the MPI functionality used inside PyACTS is available in all flavor implementations and the observed performance is quite similar.

Table 1. A subset of the Numerical Tools in the ACTS Collection with their Python based Interfaces. At this time, all these third party Python based projects are independent of PyACTS and not a part of the ACTS collection.

TOOL	**Short Description**
ScaLAPACK [3] and PyScaLAPACK from PyACTS	Library of high-performance linear algebra routines for distributed-memory message-passing Multiple Instruction Multiple Data (MIMD) computers and networks of workstations. The library contains routines for solving systems of linear equations, least squares, eigenvalue problems and singular value problems. It also contains routines that handle matrix factorizations or estimation of condition numbers.
SuperLU [4] and PySuperLU from PyACTS	General purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high performance machines. The library is written in C and is callable from either C or Fortran. The library routines perform an LU decomposition with numerical pivoting and triangular system solves through forward and back substitution.
PETSc [5] and PyPETSc [6]	The **P**ortable, **E**xtensible **T**oolkit for **S**cientific **c**omputation [7], provides sets of tools for the parallel, as well as serial, numerical solution of PDEs that require solving large-scale, sparse linear and nonlinear systems of equations. PETSc includes nonlinear and linear equation solvers that employ a variety of Newton techniques and Krylov subspace methods.
SUNDIALS and some available Python bindings	**SU**ite of **N**onlinear and **DI**fferential/**AL**gebraic equation **S**olvers, and it refers to a family of four closely related solvers; CVODE [8,9], for systems of ordinary differential equations; CVODES [10], variant of CVODE for sensitivity analysis; KINSOL [11], for systems of nonlinear algebraic equations; and IDA [12], for systems of differential-algebraic equations.
Trilinos [13] and PyTrilinos [14]	A framework for the development of parallel solvers and libraries within an object-oriented environment. AztecOO is one of the libraries available in Trilinos and it is part of the ACTS Collection. The Trilinos framework offers a variety of mechanisms for a software package to interact with other software packages.

PyACTS is a collection of carefully designed and written *software wrappers* to the ACTS tools, it also includes other routines written in Python to provide high level users interfaces. Therefore, wrappers written for PyPBLAS, PyBLACS and PyScaLAPACK were generated first with the help of F2PY [21], and then we loaded these wrappers with functionality that will automatically validate the arguments passed to the actual PyACTS routines, and check for consistency between the types of objects expected by the ACTS tools. These wrappers also provide us with the ability to transparently convert data types between PyACTS modules to support interoperability. More details on these wrappers are giving later in this section. In additional, PyACTS interfaces contain fewer arguments in their calls but generate automatically other parameters that are later passed to the actual ACTS tool interfaces. An example of this abstraction is shown with a PBLAS 3 example in Figure 1. In the panel (a) of this figure, we notice that there are parameters like the PBLAS descriptors that do not contribute directly with the operation, $C = \alpha AB + \beta C$, but are there to support the parallelism and optimize the algorithmic implementation. Many users in computational science and engineering do not care for these levels of details.

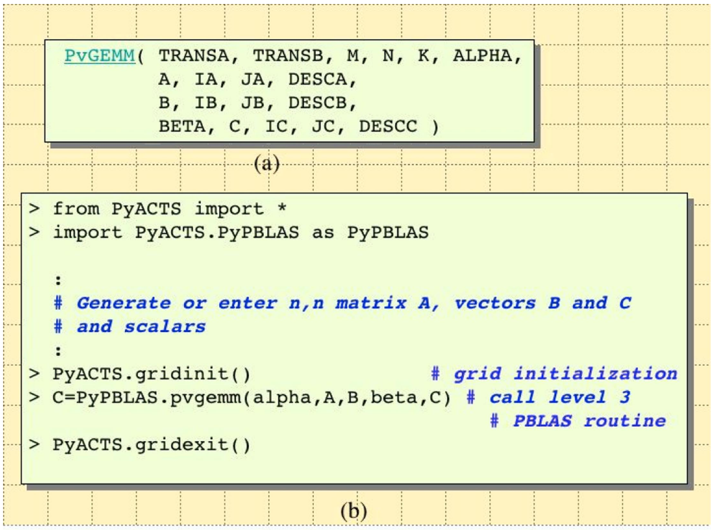
In example illustration (Figure 1), PyACTS removes these extra arguments from the PyPBLAS user interface. And internally PyPBLAS will automatically generate the missing parameters for the user and execute the proper call to the corresponding PBLAS routine. Furthermore, one more complex part of the ScaLAPACK and SuperLU interfaces is the handling of the two dimensional cyclic distribution. PyACTS provides an automatic mechanism to create the data layout and manage the resulting data distributions for the user.

While our PyACTS implementations automatically generates many of the parameters for the user, and provides support functionality, the user can still modify this parametric behavior by calling directly a routine at a lower lever of the PyACTS structure. Thus, PyACTS has resulted in a modular tool that support users with different levels of expertise with ACTS tools.

Figure 2 illustrates the internal structure of the PyACTS software. As illustrated in this graph we use components of pyMPI and Numerical Python, NumPy [22], to provide array management and parallelism. Additionally we have created a set of utilities to facilitate I/O of different formats (e.g., NETCDF, ascii), and general purpose processing routines.

The utilities module is shared by all the components of PyACTS and they are not particular to a tool in ACTS. The individual tool modules (e.g., the PyScaLAPACK module, PySuperLU module, etc.) contain the Python bindings to the ACTS tools.

The Python wrappers provide not only a level of transparency to some tool arguments but also a set of well designed validation procedures and generation of extra arguments to call the Fortran or C language libraries. Validation procedures verify that the correct variables are passed as parameters to a given routine. For instance when calling a PyScaLAPACK routines that takes a matrix as argument, the verification will consist of a checking of the ScaLAPACK type matrix and that the matrix has indeed values before it calls to the actual ScaLAPACK matrix. Internally, it also checks whether the corresponding ScaLAPACK contexts for the distributed arrays have been created.



```
PvGEMM( TRANSA, TRANSB, M, N, K, ALPHA,
        A, IA, JA, DESCA,
        B, IB, JB, DESCB,
        BETA, C, IC, JC, DESCC )
```

(a)

```
> from PyACTS import *
> import PyACTS.PyPBLAS as PyPBLAS

  :
  # Generate or enter n,n matrix A, vectors B and C
  # and scalars
  :
> PyACTS.gridinit()                 # grid initialization
> C=PyPBLAS.pvgemm(alpha,A,B,beta,C) # call level 3
                                     # PBLAS routine
> PyACTS.gridexit()
```

(b)

Fig. 1. The generic BLAS 3 Fortran and C call is shown in panel (a), and panel (b) shows the simplified PyACTS, specifically PyPBLAS, version of the same call

In PyACTS, the interoperability between ACTS tools is managed via a collection of routines for conversions of data representations between the different ACTS tools. For instance, this allows a user to convert a matrix from SuperLU into a PETSc matrix in an easier manner.

Currently, we have developed an interface to ScaLAPACK and SuperLU, PyScaLAPACK [15] and PySuperLU, respectively. In addition, we have designed a modular implementation of PyACTS that is shown in Figure 2. This design allows for easily handling of different versions of the same package and also the interoperability with other Python interfaces from other ACTS tool developers. For instance, PETSc and SUNDIALS provide their own Python extensions. Trilinos provides PyTrilinos [23], with Python extensions to provide access to most of the Trilinos functionality. TAU [24], a performance profiling and tuning tool in the ACTS Collection, can also profile programs written in Python. Thus, the PyACTS modular structure still allows for integration of existing PyACTS functionality with the ones being developed by other ACTS tool developers. As shown also in Figure 2, a user wanting to use PyACTS needs to have installed: MPI, BLAS, BLACS, ScaLAPACK, Python 2.1 or later, and NumPy

4 Some Examples of Applications Using PyACTS Modules

In this section we briefly present results from two parallel implementations of the Conjugate Gradient (CG) algorithm. The first implementation written in

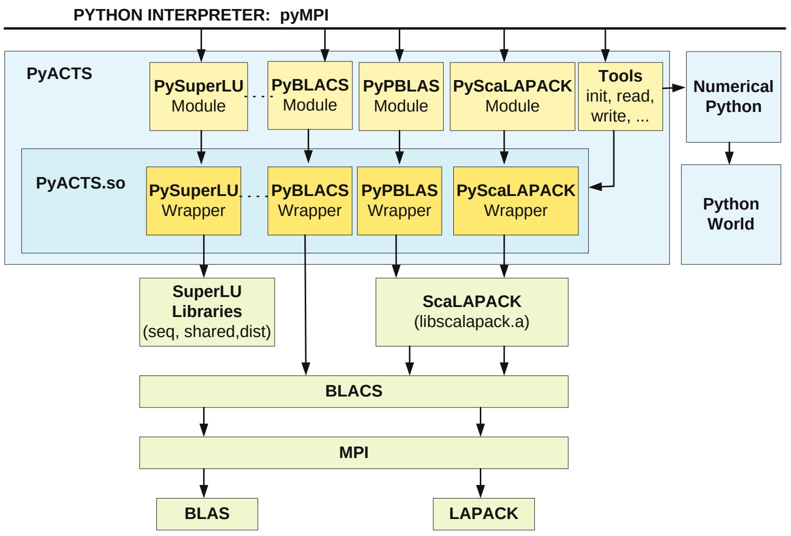
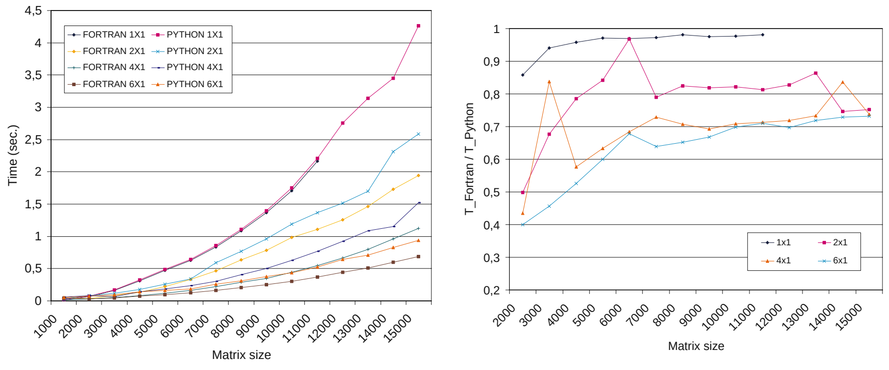


Fig. 2. Modular Structure of the PyACTS Project

pure Fortran and the second one in Python using PyACTS modules. There are many implementations of the CG for high performance computing, including robust and scalable implementations that are offered in the ACTS Collection. Furthermore, there is extensive literature on the subject that discusses in detail its derivation, applications and performance issues of the CG Algorithm. Thus, our goal here is not to provide a better version of the CG or extend the literature on the subject, but rather to illustrate with an example and performance results the low overhead of PyACTS. We have chosen this example because of the multiple calls to PBLAS routines and parallel manipulation of vectors and matrices.

In the PyACTS code, we have used the PyPBLAS module, and in the Fortran code we call the counterpart PBLAS routines directly. The experiments were perform in a Linux cluster with 6 2.0GHz Intel processors and 512Mbytes memory per processor. The BLAS level routines were previously optimized with ATLAS [25].

The different curves correspond to different processor grids for the two code implementations. First, we observe a marginal difference in the timings between the PyACTS and Fortran versions. Because Python automatically provides a more efficient memory management mechanism than Fortran (i.e., a Fortran code without any special memory management nor memory optimization schemes) in some cases we were able to run the parallel PyACTS implementation with matrix sizes and processes grids that were not possible with Fortran due to memory limitations. For instance see 1×1 configuration for matrix sizes over 11,000 in Figure 3(a). Figure 3(b) shows the relationship between the execution time of the Fortran code and Python based one ($Time\_FortranCode \div Time\_PyACTSCode$)



(a) Actual Run Times (b) Relative Run Times

Fig. 3. Comparing a Fortran vs a Python implementation of the Conjugate Gradient algorithm. We use PyACTS modules for the Python implementation.

A number of other examples have been performed with both PyScaLAPACK and PyPBLAS routines [26], and the results have consistently displayed a marginal difference between the PyACTS and Fortran code implementations in different computer platforms.

Additionally, PyScaLAPACK has already been used inside scientific applications [26]. In particular, it has been used to provided parallel functionality to a sequential Python-based package called PyClimate. PyClimate [27] provides support to common tasks during the analysis of climate variability data. It provides functions that range from simple IO operations and operations with COARDS-compliant netCDF files to Empirical Orthogonal Function (EOF) analysis, Canonical Correlation Analysis (CCA) and Singular Value Decomposition (SVD) analysis of coupled data sets, some linear digital filters, kernel based probability-density function estimation and access to DCDFLIB.C library from Python. PyClimate uses functionality available in LAPACK.

5 Conclusions and Future Work

Although, early experiments with PyACTS have shown a low overhead induced by the Python-based interface, PyACTS is not yet intended for large production runs in high-end system, rather it is a didactical tool for generating a first prototype of the application code. It helps the user to become familiar with a particular interface and also access in an interoperable manner other ACTS tools interface without having to learn it in great detail. We envision that the popularity of high-level programing languages, and their portability to many computer system will in the future enable technology that will make this high-level programming languages to scale. Thus, PyACTS may also scale to thousands of processors.
We are currently working on a PyACTS *scribe* that will allow to write out the Fortran and C language equivalent functions of the High-Level PyACTS routines.

Therefore, a user that prototypes an application using PyACTS will be able to get the exact Fortran or C calling interface sequence in order to produce a code that can be compiled and used for production runs in a large number of system.
In the future, we will be working closely with other ACTS tool developers and integrating more functionality to PyACTS.

Acknowledgments. This work was partially supported by the Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract No. DE-AC03-76SF00098", the Spanish Ministry of Science and Education under grant number TIN2005-093070-C02-02, and by Universidad de Alicante under grant number VIGROB-020.

References

1. Drummond, L.A., Marques, O.A: An Overview of the Advanced CompuTational Software (ACTS) Collection. ACM Transactions on Mathematical Software 31(3), 282–301 (2005), `http://doi.acm.org/10.1145/1089014.1089016`
2. Drummond, L.A., Hernández, V., Marques, O., Román, J.E., Vidal, V.: A Survey of High-Quality Computational Libraries and Their Impact in Science and Engineering Applications. In: Ganter, B., Godin, R. (eds.) ICFCA 2005. LNCS (LNAI), vol. 3403, pp. 37–50. Springer, Heidelberg (2005)
3. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J.W., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK User's Guide. SIAM, Philadelphia (1997)
4. Demmel, J.W., Gilbert, J.R., Li, X.: SuperLU User's Guide. University of California, Berkeley (2003)
5. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M., Curfman McInnes, L., Smith, B.F., Zhang, H.: PETSc's home page (2007), `http://www.mcs.anl.gov/petsc`
6. Korvola, T.: PyPETSc (2005), `http://www.elisanet.fi/tkorvola/hacks/''`
7. Balay, S.K., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. Modern Software Tools in Scientific Computing. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools in Scientific Computing, pp. 163–202. Birkhauser Press (1997)
8. Cohen, S.D., Hindmarsh, A.C.: CVODE User Guide. Technical Report UCRL-MA-118618, Lawrence Livermore National Laboratory (1994)
9. Byrne, G.D., Hindmarsh, A.C.: User documentation for PVODE, an ODE solver for parallel computers. Technical Report UCRL-ID-130884, Lawrence Livermore National Laboratory (1998)
10. Hindmarsh, A.C., Serban, R.: User Documentation for CVODES, An ODE Solver with Sensitivity Analysis Capabilities. Technical Report UCRL-MA-148813, Lawrence Livermore National Laboratory (2002)
11. Taylor, A.G., Hindmarsh, A.C.: User Documentation for KINSOL, A nonlinear solver for sequential and parallel computers. Technical Report UCRL-ID-131185, Lawrence Livermore National Laboratory (1998)
12. Hindmarsh, A.C., Taylor, A.G.: User Documentation for IDA, a Differential-Algebraic Equation Solver for Sequential and Parallel Computers. Technical Report UCRL-MA-136910, Lawrence Livermore National Laboratory (1999)

13. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An Overview of the Trilinos Project. ACM TOMS Vol 31:3 (2004) 127
14. Sala, M., Spotz, W., Heroux, M.: PyTrilinos: High-performance distributed-memory solvers for Python. ACM TOMS (2006) (submitted)
15. Galiano, V., Drummond, L.A., Migallón, V., Penadés, J.: High Level User Interfaces for High Performance Libraries in Linear Algebra: PyBLACS and PyPBLAS. In: Proceedings from 12th International Linear Algebra Society Conference, University of Regina, Regina, Saskatchewan, Canada (2005)
16. Drummond, L.A., Galiano, V., Migallón, V., Penadés, J.: Improving ease of use in BLACS and PBLAS with Python. In: Joubert, G., Nagel, W., Peters, F., Plata, O., Tirado, P., Zapata, E. (eds.) Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, vol. 33, NIC series (2006) ISBN 3-00-017352-8
17. Kang, N., Drummond, L.A.: A first prototype of PyACTS. Technical Report LBNL-53849, Lawrence Berkeley National Laboratory (2003)
18. Marques, O.A., Drummond, L.A.: The ACTS Information Center (2007), `http://acts.nersc.gov`
19. van Rossum, F.D.J.G.: An Introduction to Python. Network Theory Ltd (2003)
20. Miller, P.: PyMPI - An introduction to parallel Python using MPI (2002), `http://www.llnl.gov/computing/develop/python/pyMPI.pdf`
21. Peterson, P.: F2py users guide and reference manual (2005), `http://cens.ioc.ee/projects/f2py2e/`
22. Ascher, D., Dubois, P.F., Hinsen, K., Hugunin, J., Oliphant, T.: Numerical Python. Lawrence Livermore National Laboratory, Livermore, CA 94566, UCRL- MA-128569 (2001), `http://numpy.sourceforge.net`
23. Sala, M.: Distributed Sparse Linear Algebra with PyTrilinos. Technical Report SAND2005-3835, Sandia National Laboratories (2005)
24. Shende, S., Malony, A.D.: The tau parallel performance system. International Journal of High Performance Computing Applications 20, 287–311 (2006)
25. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the atlas project. Parallel Computing 27, 3–25 (2001)
26. Drummond, L.A., Galiano, V., Marques, O.A., Migallón, V., Penadés, J.: PyACTS: A High-Level Framework for Fast Development of High Performance Applications. In: Lecture Notes in Computer Science. vol. 4395, pp. 417–425 (2007)
27. Saenz, J., Zubillaga, J., Fernández, J.: Geophysical data analysis using Python. Computers and Geosciences 28/4, 457–465 (2002)

High-Performance Graph Algorithms from Parallel Sparse Matrices

John R. Gilbert[1,⋆], Steve Reinhardt[2], and Viral B. Shah[1]

[1] University of California, Dept. of Computer Science, Harold Frank Hall, Santa Barbara, CA 93106, USA
{gilbert, viral}@cs.ucsb.edu[⋆⋆]
[2] Silicon Graphics Inc.
spr@sgi.com

Abstract. Large-scale computation on graphs and other discrete structures is becoming increasingly important in many applications, including computational biology, web search, and knowledge discovery. High-performance combinatorial computing is an infant field, in sharp contrast with numerical scientific computing.

We argue that many of the tools of high-performance numerical computing – in particular, parallel algorithms and data structures for computation with sparse matrices – can form the nucleus of a robust infrastructure for parallel computing on graphs. We demonstrate this with an implementation of a graph analysis benchmark using the sparse matrix infrastructure in Star-P, our parallel dialect of the Matlab programming language.

1 Introduction

High performance applications increasingly combine numerical and combinatorial algorithms. Past research on high performance computation has focused mainly on numerical algorithms, and we have a rich variety of tools for high performance numerical computing. On the other hand, few tools exist for large-scale combinatorial computing.

Our goal is to build a general set of tools to allow scientists and engineers develop applications using modern numerical and combinatorial tools with as little effort as possible. Sparse matrix computations allow structured representation of irregular data structures, decompositions, and irregular access patterns in parallel applications.

Sparse matrices are a convenient way to represent graphs. Since sparse matrices are first class citizens in Matlab and many of its parallel dialects, it is natural to use the duality between sparse matrices and graphs to develop a rich infrastructure for numerical and combinatorial computing.

⋆ This author's work was partially supported by Silicon Graphics Inc.

⋆⋆ These authors' work was partially supported by the Air Force Research Laboratories under agreement number AFRL F30602-02-1-0181 and by the Department of Energy under contract number DE-FG02-04ER25632.

Table 1. Correspondence between some sparse matrix and graph operations

Sparse matrix operation	Graph operation
`G = sparse (U, V, W)`	Construct a graph from an edge list
`[U, V, W] = find (G)`	Obtain the edge list from a graph
`vtxdeg = sum (spones(G))`	Vertex degrees for an undirected graph
`indeg = sum (spones(G))`	Indegrees for a directed graph
`outdeg = sum (spones(G), 2)`	Outdegrees for a directed graph
`N = G(i, :)`	Find all neighbors of vertex i
`Gsub = G(subset, subset)`	Extract a subgraph of G
`G(i, j) = W`	Add or modify a graph edge
`G(i, j) = 0`	Delete a graph edges
`G(I, I) = []`	Remove vertices from a graph
`G = G(perm, perm)`	Permute vertices of a graph
`reach = G * start`	Breadth first search step

2 Sparse Matrices and Graphs

Every sparse matrix problem is a graph problem and every graph problem is a sparse matrix problem. We discuss some of the basic design principles to be aware of when designing a comprehensive infrastructure for sparse matrix data structures and algorithms in our earlier work [5,10]. The same principles apply to efficient operations on large sparse graphs.

1. Storage for a sparse matrix should be $\theta(max(n, nnz))$
2. An operation on sparse matrices should take time approximately proportional to the size of the data accessed and the number of nonzero arithmetic operations on it.

A graph consists of a set of vertices V, connected by edges E. A graph can be specified by tuples (u, v, w) – this means that there exists a directed edge of weight w from vertex u to vertex v. This is the same as a nonzero w at location (u, v) in a sparse matrix. According to Principle 1, the storage required is $\theta(|V|+|E|)$. An undirected graph is represented by a corresponding symmetric sparse matrix.

A correspondence between sparse matrix operations and graph operations is listed in Table 1. The basic design principles silently come into play in all cases. Consider breadth first search (BFS). A BFS can be performed by multiplying a sparse matrix G with a sparse vector x. The simplest case is doing a BFS starting from vertex i. In this case, we set $x(i) = 1$, all other elements being zeros. $y = G * x$ simply picks out column i of G which contains the neighbors of vertex i. If we repeat this step again, the multiplication will result in a vector which is a linear combination of all columns of G corresponding to the nonzero elements in vector x, or all vertices that are up to 2 hops away from vertex i. We can also do several independent BFS searches simultaneously by using sparse matrix sparse matrix multiplication [9]. Instead of starting with a vector, we

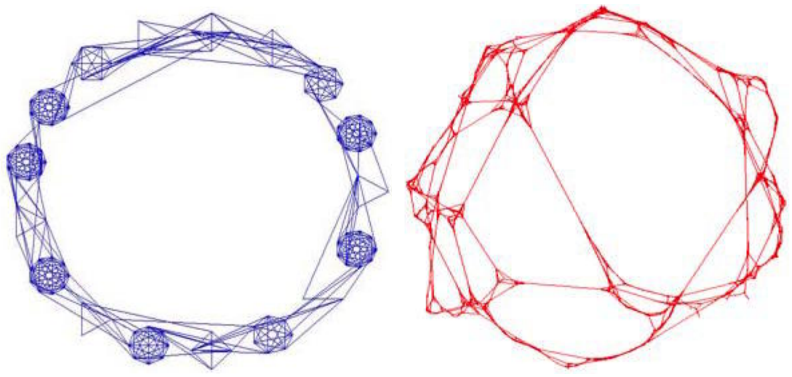


Fig. 1. SSCA #2 graph (a) Conceptual (b) Plotted with Fiedler co-ordinates

start with a matrix, with one nonzero in each column at some row j, where j is the starting vertex. So, we have $Y = G * X$, where each column of J contains the results of performing an independent BFS. Sparse matrix multiplication can be thought of as simply combining columns of G, and Principle 2 assures us that each of these indexing operations take time proportional to the number of nonzeros in that column. As a result, the time complexity of performing BFS using operations on sparse matrices is the same as that obtained by performing operations on other efficient graph data structures.

3 An Example: SSCA #2 Graph Analysis Benchmark

The SSCAs (Scalable Synthetic Compact Applications) are a set of benchmarks designed to complement existing benchmarks such as the HPL [4] and the NAS parallel benchmarks [2]. Specifically, SSCA #2 [1] is a compact application that has multiple kernels accessing a single data structure (a directed multigraph with weighted edges). The data generator generates an edge list in random order for a multigraph of sparsely connected cliques as shown in Figure 1. The four kernels are as follows:

1. Kernel 1: Create a data structure for further kernels.
2. Kernel 2: Search graph for a maximum weight edge.
3. Kernel 3: Perform breadth first searches from a set of start vertices.
4. Kernel 4: Recover the underlying clique structure from the undirected graph.

The benchmark spec is still not finalized. We describe our implementation of version 1.1 (integer only) of the spec in this paper.

3.1 Scalable Data Generator

The data generator is the most complex part of our implementation. It generates edge tuples for subsequent kernels. No graph operations are performed at this

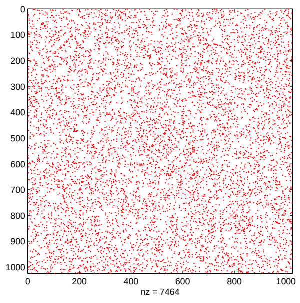


Fig. 2. MATLAB spy plot of the input graph

stage. The input to the data generator is a `scale` parameter, which indicates the size of the graph being generated. The resulting graph has 2^{scale} vertices, with a maximum clique size of $\lfloor 2^{scale/3} \rfloor$, a maximum of 3 edges with the same endpoints, and a probability of 0.2 that an edge is directional. The vertex numbers are randomized, and a randomized ordering of the edge tuples is presented to the subsequent kernels. Our implementation of the data generator closely follows the pseudocode published in the spec.

3.2 Kernel 1

Kernel 1 creates a read-only data structure that is used by subsequent kernels. We create a sparse matrix corresponding to each layer of the multigraph. The multigraph has 3 layers, since there is a maximum of 3 parallel edges between any two vertices in the graph. MATLAB provides several ways of constructing sparse matrices, `sparse()` being one of them. It takes as its input a list of 3-tuples - (i, j, w_{ij}). Its output is a sparse matrix with a nonzero w_{ij} in every location (i, j) specified in the input. Figure 2 shows a spy plot of one layer of the input graph.

3.3 Kernel 2

In kernel 2, we search the graph for edges with maximum weight. `find()` is the inverse of `sparse()`. It returns all nonzeros from a sparse matrix as a list of 3-tuples. We then use `max()` to find the maximum weight edge.

3.4 Kernel 3

In kernel 3, we perform breadth first searches from a given set of starting points. We use sparse matrix-matrix multiplication to perform all breadth first searches simultaneously from the given starting points. Let G be the adjacency matrix representing the graph and S be a matrix corresponding to the starting points. S has one non-zero in each column for every starting point. Breadth first search is performed by repeatedly multiplying G with S: $Y = G * X$. We perform several

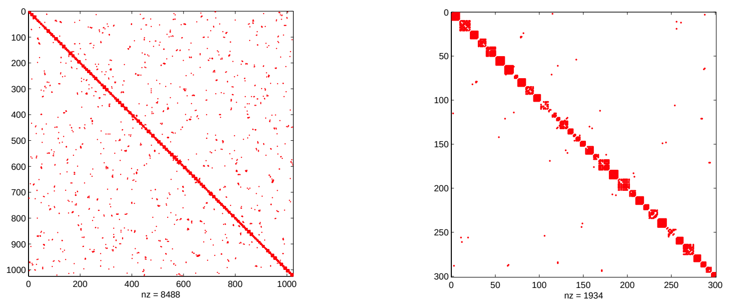


Fig. 3. (a) Clusters in full graph (b) Magnification around the diagonal

breadth first searches simultaneously by using sparse matrix-matrix multiplication. Star-P stores sparse matrices by rows, and parallelism is achieved by each processor computing some rows in the product [10,9].

3.5 Kernel 4

Kernel 4 is the most interesting part of the benchmark. It can be considered to be a partitioning problem or a clustering problem. We have several implementations of kernel 4 based on spectral partitioning (Figure 1), "seed growing" (Figure 3), and "peer pressure" algorithms. The peer pressure and seed growing implementations scale better than the spectral methods, as expected. We now demonstrate how we use the infrastructure described above to implement kernel 4 in a few lines of Matlab. Figure 3 shows a spy plot of the undirected graph after clustering. The clusters show up as dense blocks along the diagonal.

Our seed growing algorithm (Figure 4) starts with picking a small set of seeds (about 2% of the total number of vertices) randomly. The seeds are then grown so that each seed claims all vertices reachable by at least k paths of length 1 or 2. This may cause some ambiguity, since some vertices might be claimed by multiple seeds. We tried picking an independent set of vertices from the graph by performing one round of Luby's algorithm [7] to keep the number of such ambiguities as low as possible. However, the quality of clustering remained unchanged when we use random sampling. We used a simple approach for disambiguation – the lowest numbered cluster claiming a vertex got it. We also experimented with attaching singleton vertices to nearby clusters to improve the quality of clustering.

Our peer pressure algorithm (Figure 5) starts with a subset of vertices designated as leaders. There has to be at least one leader neighboring every vertex in the graph. This is followed with a round of voting where every vertex in the graph elects a leader, selecting a cluster to join. This does not yet yield good clustering. Each vertex now looks at its neighbors and switches its vote to the most popular leader in its neighborhood. This last step is crucial, and in this case, it recovers more than 95% of the original clique structure of the graph.

We experimented with different approaches to select leaders. At first, it seemed that a maximal independent set of vertices from the graph was a natural way to

```
% J is a sparse matrix with one seed per column.
J = sparse (seeds, 1:nseeds, 1, n, nseeds);

J = G * J;    % Vertices reachable with 1 hop.
J = J + G*J;  % Vertices reachable with 1 or 2 hops.
J = J > k;    % Vertices reachable with at least k paths of 1 or 2 hops.
```

Fig. 4. Breadth first parallel clustering by seed growing

pick leaders. In practice, it turned out that simple heuristics (such as the highest numbered neighbor) gave equally good clustering. We also experimented with different numbers of voting rounds. The marginal improvement in the quality of clustering was not worth the additional computation required.

We used STAR-P [6] for our implementation. STAR-P is a parallel implementation of the MATLAB language with global array semantics. We expect it to be straightforward to port to any other global-array parallel dialect of MATLAB, such as pMATLAB [11] or Mathworks Parallel MATLAB [8]. We present a basic performance analysis of our implementation in Section 5. We will include a detailed performance analysis of our implementation in a forthcoming journal version of our paper.

4 Visualization of Large Graphs

Graphics and visualization are a key part of an interactive system such as MATLAB. The question of how to effectively visualize large datasets in general, especially large graphs, is still unsolved. We successfully applied methods from numerical computing to come up with meaningful visualizations of the SSCA #2 graph.

One way to compute geometric co-ordinates for the vertices of a connected graph is to use Fiedler co-ordinates for the graph. Figure 1 shows the Fiedler embedding

```
% IS is the independent set. Find all neighbors in the IS.
neighbors = G * sparse(IS, IS, 1, n, n);

% Each vertex chooses a random neighbor in the independent set.
R = sprand (neighbors);
[ignore vote] = max (R, [], 2);

% Collect neighbor votes and join the most popular cluster.
[I, J] = find (G);
S = sparse (I, vote(J), 1, n, n);
[ignore cluster] = max (S, [], 2);
```

Fig. 5. Parallel clustering by peer pressure

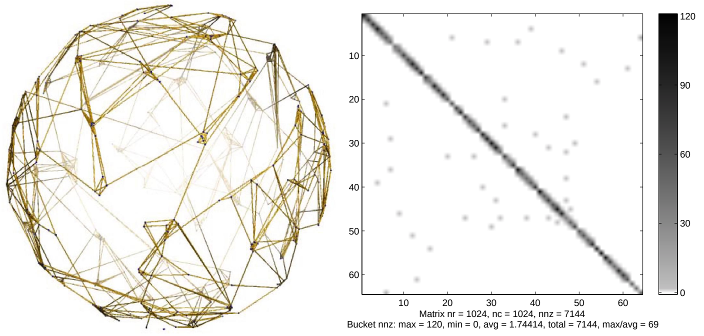


Fig. 6. SSCA #2 graph: (a) 3D visualization (b) density plot (spyy)

of the SSCA #2 graph. In the 2D case, we use the eigenvectors (Fiedler vectors) corresponding to the first two non-zero eigenvalues as co-ordinates for the graph vertices in a plane.

For 3D visualization of the SSCA #2 graph, we start with 3D Fiedler co-ordinates. We model the graph as particles on the surface of a sphere. There is a repulsive force between all particles, inversely proportional to the distance between them. If r is the distance between two particles, the forcing function we use is $1/r$. Since these particles repel each other on the surface of a sphere, we expect them to spread around and occupy the entire surface of the sphere. Since there are cliques in the original graph, we expect clusters of particles to form on the surface of the sphere. At each timestep, we compute a force vector between all pairs of particles. Each particle is then displaced some distance based on its force vector. All displaced particles are projected back onto the sphere at the end of each timestep.

This algorithm was used to generate Figure 6. In this case, we simulated 256 particles and the system was evolved for 20 timesteps. It is important to first calculate the Fiedler co-ordinates. Our effort to use random co-ordinates resulted in a meaningless picture. We used PyMOL [3] to render the graph.

We also developed a version of `spy()` suitable for visualization of large graphs - `spyy()`. Large sparse graphs are often stored remotely (for example, on a Star-P server). It is impracatical to transfer the entire graph to the frontend for display. We create a density plot of the sparse matrix, and only transfer the image to the frontend. We implemented `spyy()` completely in the Matlab language. It uses parallel sparse matrix multiplication to build the density plot on the backend. A spyy plot can also be thought of as a two dimensional histogram. Figure 6 shows a spyy plot of the SSCA #2 graph after clustering.

5 Experimental Results

We ran our implementation of SSCA #2 (ver 1.1, integer only) in Star-P. The Matlab client was run on a generic PC. The Star-P server was run on an SGI Altix with 128 Itanium II processors with 128G RAM (total, non-uniform mem-

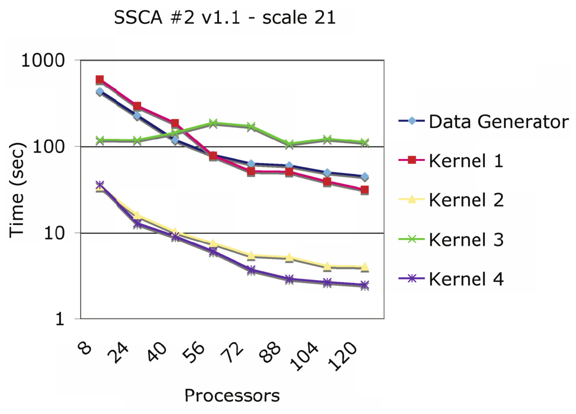


Fig. 7. SSCA #2 v1.1 execution times (Star-P, Scale=21)

ory access). We used a graph generated with scale 21. This graph has 2, 097, 152 vertices. The multigraph has 320, 935, 185 directed edges, whereas the undirected graph corresponding to the multigraph has 89, 145, 367 edges. There are 32, 342 cliques in the graph, the largest of them having 128 vertices. There are 88, 933, 116 undirected edges within cliques, and 212, 251 undirected edges between cliques in the input graph for kernel 4. The results are presented in Fig. 7.

Although it is not required by the spec, our data generator also scales very well. A lot of time is spent in kernel 1, where data structures for the subsequent kernels are created. The majority of the time is spent in searching the input triples for duplicates, since the input graph is a multigraph. Kernel 1 creates several sparse matrices using `sparse()`, each corresponding to a layer in the multigraph. Time spent in kernel 1 also scales very well with the number of processors. Time spent in Kernel 2 also scales as expected.

Kernel 3 does not show speedups at all. Although, all the breadth first searches are performed in parallel, the process of subgraph extraction for each starting point creates a lot of traffic between the Star-P client and the Star-P server - which were physically in different states. This client server communication time ends up dominating over the computation time. We will minimize this overhead by vectorizing all of kernel 3 in a future release.

Kernel 4, the non-trivial part of the benchmark, actually scales very well. We present results for our best performing implementation of kernel 4, which uses the seed growing algorithm.

The evaluation criteria for the SSCAs also include software engineering metrics such as code size, readability, maintainability etc. Our implementation is extremely concise. We provide the source lines of code (SLOC) for our implementation in Table 2. We also provide absolute line counts which include blank lines and comments, as we believe these to be crucial for code readability and maintainability. Our implementation runs without modification in sequential

Table 2. Line counts for Star-P implementation of SSCA#2. The "Source LOC" column counts only executable lines of code, whereas the "Line counts" column counts the total number of lines including comments and whitespace.

Operation	Source LOC	Total line counts
Data generator	176	348
Kernel 1	25	63
Kernel 2	11	34
Kernel 3	23	48
Kernel 4 (spectral)	22	70
Kernel 4 (seed growing)	55	108
Kernel 4 (peer pressure)	6	29

Matlab, making it easy to develop and debug on the desktop before deploying on a parallel platform.

6 Concluding Remarks

We have run the full SSCA #2 benchmark (spec v0.9, integer only) on graphs with 2^{27} = 134 million vertices on the SGI Altix. We have also manipulated extremely large graphs (1 billion vertices and 8 billion edges) on an SGI Altix with 256 processors using Star-P.

We demonstrate a robust, scalable way to manipulate large graphs by representing them with sparse matrices. Although it may be possible to achieve higher performance with different data structures and distributions, it is extremely hard to design a general purpose system which can support such a variety of representations, and the resulting combinatorial explosion of interactions between them. This is why Star-P has only one representation for sparse matrices [10]. This allows for a robust, scalable, well-tuned implementation of sparse matrix algorithms, and hence, operations on graphs.

Note that the codes in Figure 4 and Figure 5 are not pseudocodes, but actual code excerpts from our implementation. Although the code fragments look very simple and structured, the computation is anything but. All operations are on sparse matrices, resulting in highly irregular communication patterns on irregular data structures. We conclude that sparse matrix operations provide a convenient language to efficiently manipulate large graphs.

References

1. Bader, D.A., Madduri, K., Gilbert, J.R., Shah, V., Kepner, J., Meuse, T., Krishnamurthy, A.: Designing scalable synthetic compact applications for benchmarking high productivity computing systems. Cyberinfrastructure Technology Watch, 2(4B) (November 2006)

2. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. The International Journal of Supercomputer Applications 5(3), 63–73 (1991)
3. DeLano, W.L.: The PyMOL molecular graphics system, DeLano Scientific LLC, San Carlos, CA, USA (2006), http://www.pymol.org/
4. Dongarra, J.J.: Performance of various computers using standard linear equations software in a Fortran environment. In: Karplus, W.J. (ed.) Multiprocessors and array processors: proceedings of the Third Conference on Multiprocessors and Array Processors, San Diego, CA, USA, January 14–16, 1987. pp. 15–32, Society for Computer Simulation (1987)
5. Gilbert, J.R., Moler, C., Schreiber, R.: Sparse matrices in MATLAB: Design and implementation. SIAM J. on Matrix Anal. Appl. 13(1), 333–356 (1992)
6. Husbands, P., Isbell, C.: MATLAB*P: A tool for interactive supercomputing. In: SIAM Conference on Parallel Processing for Scientific Computing (1999)
7. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. 15(4), 1036–1053 (1986)
8. Moler, C.B.: Parallel matlab. In: Householder Symposium on Numerical Algebra (2005)
9. Robertson, C.: Sparse parallel matrix multiplication. M.S. Project, Department of Computer Science, UCSB (2005)
10. Shah, V., Gilbert, J.R.: Sparse matrices in Matlab*P: Design and implementation. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296, pp. 144–155. Springer, Heidelberg (2004)
11. Travinin, N., Kepner, J.: pMatlab parallel matlab library. International Journal of High Performance Computing Applications 2006 (submitted)

A Python Module for PDE-Based Numerical Modelling

Lutz Gross, Ben Cumming, Ken Steube, and Dion Weatherley

Earth Systems Science Computational Centre (ESSCC),
The University of Queensland, St Lucia, QLD 4072, Australia
l.gross@uq.edu.au

Abstract. The *escript* package is an extension of *python*. It provides an easy-to-use programming environment for numerical simulations based on the solution of partial differential equations (PDEs), while at the same time providing for fast solution of large models by performing time-intensive calculations in C++ and C. The *escript* functionality allows the user to implement high-level numerical schemes to reduce coupled, non-linear, time-dependent PDEs to linear, steady PDEs that have to be solved in each time and/or iteration step. The PDEs are then solved by our *finley* PDE solver library. The layer of abstraction provided by *escript* allows an implementation which is independent from particular discretization schemes, PDE solver libraries, their data structures, and the computing platform itself. In the paper we will briefly outline the basic concepts of *escript*, illustrate its usage for modelling seismic wave propagation and discuss some parallelization issues with OpenMP and MPI.

1 Introduction

The basic idea of the *python* [4] module *escript* [3,1,2] is to provide an abstraction of the mathematical formulation of a model from the discretization techniques that are used to run these models. Firstly, this approach creates an environment that is very natural for scientists to work with, as it resembles the functions and equations used in formulating the model. The scientist does not have to deal explicitly with the numerical objects, such as sparse matrices and their data structures. We see this as an important contribution to handle the complexity of models as can be found in some area of science, such as rheology.

Moreover, *escript* facilitates a high degree of reusability and portability. Models can be run with various discretization techniques. For instance, for a simple geometry a finite difference method may be most efficient while for a more complex geometry finite elements have to be used. As *escript* is not bound to a particular numerical library it is possible to run simulations on different platforms with different numerical libraries where there is an advantage in performance.

Several projects have developed environments for defining and solving PDEs. Some of these environments are using their own programming languages, for instance FASTFLO [12], ELLPACK [11]. One drawback of this approach is that users have to learn a new programming language. Moreover, as not designed as proper programming languages they are often missing important programming language features and functionalities as requested by new application areas that have not been considered in the original design. Therefore, the approach of embedding the PDE solving environment into an existing language such as C++, MATLAB or *python* is more successful in terms of acceptance in the user community as well as richness of functionality that can be provided with minimum investment costs. For instance, *Sundance* [13], which follows an approach very similar to *escript* but is less open, is an extension to C++. As the efficient usage of a C++–based environment requires good software engineering skills, most users prefer interactive and script–based programming environments such as MATLAB and *python* even if this comes at the costs of less efficient codes. As an object oriented approach is very appropriate for an abstraction layer to define PDEs and spatial function *python* has been chosen as the user environment for *escript*.

In the next section, we will present a mathematical model for seismic wave propagation in the earth crust. In the third section we will briefly outline the basic concept of *escript* and show how *escript* used to implement the seismic wave propagation model. Section four will discuss the parallelization of *escript* for OpenMP [9] and MPI [7].

2 Example: Seismic Wave Propagation

To illustrate the usage of *escript* we present the implementation of a simple wave propagation model after a seismic event [6]. The domain of interest is a three-dimensional block in the outer crust of the earth. Typical dimensions are 30km in depth and 100-500km in length. The block is meant to model a coastal region and contains various materials such as rock, sediment and water.

The model for the propagation of waves after a seismic event is based on the solution of the wave equation given below: For any given time $t > 0$, the displacement field $u = (u_i)$ on the domain Ω is given by

$$\rho u_{i,tt} = \sigma_{ij,j} + F_i \tag{1}$$

where ρ is the density and F_i is an internal load. In this equation, the notation $Z_{,j}$ denotes the derivative of function Z with respect to the j-th spatial direction, and we are using standard tensor summation notation. The function σ_{ij} is the stress field, which in case of an isotropic, linear elastic material is given by

$$\sigma_{ij} = \lambda \, u_{k,k}\delta_{ij} + \mu \left(u_{i,j} + u_{j,i}\right) \, , \tag{2}$$

The coefficients λ and μ are the Lame coefficients and δ_{ij} denotes the Kronecker symbol. The density as well as the Lame coefficients depend on their location in the domain. Typically, they are represented through piecewise constant functions representing bed rock, sand and water in the domain.

At time $t = 0$, the displacement u and the velocity $u_{,t}$ are assumed to be zero. On the boundary Γ of Ω the normal stress is assumed to be zero for all time $t > 0$:

$$\sigma_{ij} n_j = 0 \tag{3}$$

where n_i is the outer normal field of Ω. This boundary condition will reflect waves on the boundary surface. On some parts of the domain surface it could be appropriate to use non-reflecting boundary conditions. As it is always possible to choose a sufficiently large domain to eliminate the effects of reflected waves and for the sake of a simple presentation, non-reflecting boundary conditions are not discussed here.

The load induced by a seismic event at time t_q is modeled in the form

$$F_i(x,t) = Q_i e^{-\left(\frac{\|x-x_q\|^2}{s^2} + \frac{\|t-t_q\|^2}{l^2}\right)} . \tag{4}$$

where x_q is the location of the event, s its spatial size and l its temporal length. The constant Q_i is representing the acting force at the time of the event.

We employ the explicit Verlet scheme [6] with constant time step size dt to solve the wave propagation equation (1). If $u^{(n)}$ gives the displacement at time $t^{(n)} = n \cdot dt$ one sets

$$u^{(n)} = 2 \cdot u^{(n-1)} - u^{(n-2)} + dt^2 \cdot a^{(n)} . \tag{5}$$

where $a^{(n)}$ is the acceleration at time $t^{(n)}$ given by the solution of

$$\rho a^{(n)} = \sigma_{ij,j}^{(n-1)} + F_i^{(n)} \tag{6}$$

together with the natural boundary condition (3) for $\sigma_{ij,j}^{(n-1)}$.

To ensure that the Verlet scheme (5) is stable the following Courant condition has to be fulfilled everywhere in the domain:

$$dt \le \theta \frac{h}{v_p} \text{ with } v_p = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{7}$$

where h denotes the local discretization length, for instance the diameter of the elements in the finite element mesh, and θ is a safety factor, typically $\theta = \frac{1}{5}$.

3 The *escript* Environment

We will use the example presented in section 2 to outline the basic ideas of *escript* [3,1] and illustrate how *escript* is used to implement a practical simulation. We will assume that the simulation is implemented using the finite element method (FEM) [5] but these scripts can be used with other discretization techniques after only minor changes.

3.1 Spatial Functions

Functions of spatial coordinates play a key role when implementing mathematical models. The following *python* [4] script defines a function over the domain dom as it would be used to define the displacement field $u = 0$ at the initial time step:

```
from finley import Brick
from escript import *
dom=Brick(...) # Arguments define the 3-D domain
u=Vector(0.,Solution(dom))
```

This defines the domain that will be used by our FEM solver C library *finley* [1] to run the simulation. The Domain class object returned by the function Brick holds pointers to the key *finley* data structures defining the finite element mesh. The function represented by the *escript* Data object u is a vector-valued function. The initial value is 0.0. The argument Solution(dom) declares the FunctionSpace of u (function spaces will be described below). In this case the function represented by u is declared as a solution of a PDE on the domain defined by the object dom. The definition of u does not require to specify how the function is actually represented. The representation being used is depending on the Domain. In the context of the FEM, PDE solutions are typically represented by their values on the nodes of the mesh. It is pointed out that here as well as in the following we never explicitly refer to the actual representation of a function. This makes it possible to run a mathematical model with various discretization techniques as well as code portability to across compute platforms.

The following function returns the stress as calculated by equation (2). It takes the current displacement u and the Lame coefficients lmbd and mu as argument:

```
def getStress(u,lmbd,mu):
   dom=u.getDomain()
   eps=symmetric(grad(u))
   k=kronecker(Function(dom))
   stress=lmbd*trace(eps)*k+2*mu*eps
   return stress
```

The function grad calculates the gradient of its argument u. In the FEM context, the input has to be represented by values on the mesh nodes while the gradient is calculated on the element centers or integration points within each element. So in comparison to its argument the gradient is defined on the same domain but is represented differently. This is reflected by the fact that FunctionSpace of the returned Data object is now Function(dom) rather than Solution(dom).

On a given domain dom we will need functions of different types, which we distinguish by what we call a FunctionSpace. For example, some functions are defined on the nodes, and thus are continuous across the elements. These belong to the function space called Solution(dom). Other functions are defined on integration points off the nodes. These belong to the function spaced called

Function(dom). If functions on different function spaces are added together, then *escript* chooses the best way to use interpolation to effect the addition, and the result is returned with an appropriate function space. Binary operations such as addition and multiplication and unary operations such as taking the matrix trace do not otherwise depend on the type of function space.

Also, when we run *escript* as a parallel application your Data objects may be distributed across multiple processors. All binary and unary operations work seamlessly on distributed data without requiring modification of the python script.

escript itself is completely independent of the concept of FunctionSpace. Any time knowledge is required concerning the placement of data on the nodes or distribution of data across processors, the request is passed to a deeper layer associated with the domain. This allows us to link with third-party software libraries for dealing with spatial data.

The getStress function does not make any assumptions about the type for the Lame coefficients lmbd and mu except that they have to be scalars. In fact, because of the way *python* is handling variables and *escript* resolves argument types in operations, they can be simple floating point numbers, *numarray* objects [8] or Data objects. In the latter case, the FunctionSpace of an argument does not need to be equal to Function(dom) which is the FunctionSpace of the other variables in the stress calculation. Any mismatch is resolved automatically by interpolation in *escript*.

In many applications, material coefficients such as the Lame coefficients λ and μ are represented by piecewise constant functions. *escript* provides the mechanism of assigning values to data points using tags as an easy and efficient way to define piecewise constant functions. Generators for FEM meshes typically allow us to assign a single tag, typically an integer, to all elements in a subregion of the domain. This tag can then be used as an index to assign the same material parameters to all elements in that subregion.

The following script shows how this is done in *escript*:

```
lmbd=Scalar(lmbd_rockbed,Function(domain))
lmbd.setTaggedValue(sand,lmbd_sand)
lmbd.setTaggedValue(water,lmbd_water)
```

Here we assume that the mesh generator has assigned the tags sand and water to the corresponding regions filled with sand and water. The Data object lmbd representing λ is initialized as a scalar function with constant value lmbd_rockbed which is the value of λ for the rock bed. Then new values for λ for the tags sand and water are set. The initial value lmbd_rockbed is the default value used by all other tags different from sand and water.

The tagged data set used above is just one of *escript*'s three different storage schemes: in the general case for each data point a different value (or array of values) is stored. Alternatively, if all data point uses the same value, only a single value is stored. The third scheme is the tagged data set, which uses a dictionary to define the values to be used for a given tag. The first storage scheme is the most computationally expensive to process and requires the most memory. If `Data` object with different storage schemes are combined in arithmetic expressions, *escript* intelligently resolves the different storage schemes and chooses the most appropriate storage scheme for the result.

3.2 Partial Differential Equations

Now that we have the domain for our model we are ready to define our PDEs. In *escript* the `LinearPDE` class objects define general linear, steady, second order PDEs for an unknown function u on the PDE domain. In tensor notation, the PDEs have to take the form

$$-\left(A_{ijkl}u_{k,l} + B_{ijk}u_k\right)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k = -X_{ij,j} + Y_i \ , \tag{8}$$

with natural boundary conditions

$$n_j(A_{ijkl}u_{k,l} + B_{ijk}u_k) + d_{ik}u_k = n_j X_{ij,j} + y_i \tag{9}$$

A general form of constraints can be used but it is not presented here. The functions A, B, C, D, X, Y, d and y are the coefficients of the PDE. Note that A, B and X in equations (8) and (9) are identical. When dealing with non-linear and time-dependent problems, you can use a suitable high-level scheme, for instance Newton-Raphson or Verlet, to reduce the problem to solutions of linear PDEs. The coefficients are defined by `Data` objects with function space `Function` for A, B, C, D, X, Y and `FunctionOnBoundary` for d and y. If coefficients are not defined with expected function space they will be interpolated.

For the seismic wave propagation problem introduced in section 2 we have to solve the (degenerated) PDE (6) in each time step to get $a^{(n)}$. The values for the PDE coefficients can be easily identified by comparison with equations (8) and (9):

$$D_{ij} = \rho\delta_{ij} \ , X_{ij} = -\sigma_{ij}^{(n-1)} \ , Y_i = F_i^{(n)} \tag{10}$$

The following script shows how to use the `LinearPDE` class:

```
from escript import LinearPDE
pde=LinearPDE(dom)
k=kronecker(Function(dom))
pde.setValue(D=k*rho, X=-stress, Y=F)
a=pde.getSolution()
```

We assume that `dom` is a `Domain` class object and `rho`, `stress` and `F` are `Data` class objects with function space `Function`, or compatible objects. When the solution of the PDE is requested, *escript* passes the PDE coefficient to the solver library defined by `Domain` of the `LinearPDE`. The return value is a `Data` object with the function space `Solution`.

4 Implementation

The following script shows the implementation of the seismic wave propagation model using *escript*:

```
from escript import *
def wave(dom,rho,mu,lmbd):
   x=Function(dom).getX()
   pde=LinearPDE(dom)
   pde.setSolverMethod(LinearPDE.LUMPING)
   k=kronecker(Function(dom))
   pde.setValue(D=k*rho)
   v_p=sqrt((2*mu+lmbd)/rho)
   dt=(1./5.)*inf(dom.getSize()/v_p)
   t=0
   u=Vector(0.,Solution(dom))
   u_last=Vector(0.,Solution(dom))
   while t<t_end:
     eps=symmetric(grad(u))
     stress=lmbd*trace(eps)*k+2*mu*eps
     F=Q*exp(-((length(x-xq)/s)**2+((t-tq)/l)**2))
     pde.setValue(X=-stress,Y=F)
     a=pde.getSolution()
     u_new=2*u-u_last+dt**2*a
     u_last,u=u,u_new
     t+=dt
```

The script puts together the components that have been discussed in the previous section 3. The argument `dom` defines the `Domain` to be used for the simulation. The arguments `rho`, `mu`, and `lmbd` may be just floating point numbers or `Data` class objects defined using tagged vales to represent piecewise constant functions.

As is commonly used in explicit time integration schemes, lumping of the stiffness matrix is applied to solve the (degenerated) PDE. An instance of the `LinearPDE` class is created outside the time iteration loop. Within the loop only coefficients that are changing over time are updated. As the left-hand side coefficients are not altered over time, the `LinearPDE` class will reuse the lumped stiffness matrix formed in the first time step and the assemble a new right-hand side for each time step only.

5 Parallelization

The *escript* library has been parallelized for OpenMP [9]. A version parallelized using MPI [7] is currently under construction. The OpenMP version is optimized for distributed shared memory architectures. The program runs as a single *python* thread, often calling C++ and C methods for speed in which a parallel region will be encountered. At that time parallel threads are spawned. In MPI *python* is running on all processors.

Table 1. Wall clock time for 3D seismic wave propagation using *escript* and *finley* with OpenMP on SGI Altix 3700. N_p is the number of threads, N_elem is the total number of elements, T_step is the wall clock time per time step in seconds and T_pes is the wall clock time per time step and per element per thread in milliseconds.

N_p	N_elem	T_step	T_pes
16	492032	12.1	0.396
8	221184	10.2	0.369
4	118125	5.61	0.190
2	63916	5.52	0.172
1	37553	6.50	0.173

Simulations developed in *escript* can be run serial, with OpenMP or MPI without modifications. In order to guarantee portability *python* variables are assumed to have the same value on all processors. The data distribution of `Data` class objects is not visible at the *python* level, but rather to lower-level numerical library using it. In practice the user do not need to know anything about parallel data distribution. On the implementation level of the *escript* library knowledge of data distributions which are inherited from the numerical library using them is required to perform synchronization and reduction operations. The exchange of data between processors is required only for calculating gradients, interpolation and solving PDEs and is therefore left to the numerical library.

Currently, *escript* is linked with the finite element library *finley*, which is designed for handling unstructured meshes and is parallelized for ccNUMA architecture and OpenMP [1] using multi–coloring. The MPI version is based on the non-overlapping distribution of the rows of the stiffness matrix and overlapping distribution for elements is used. These distributions are inherited by *escript* `Data` class objects with the `FunctionSpace` `Solution` and `Function`, respectively.

Table 1 shows the timings of 3D seismic wave propagation simulations using *escript* and *finley* on an SGI Altix 3700 [10] using OpenMP. The problem sizes has been chosen such that about 30000 elements per processors are used. The mesh is a rectangular grid with hexahedron elements of order one. The column T_step shows the processing time per element and time step, i.e. $T_\mathrm{step} = \frac{T_\mathrm{step} \cdot N_\mathrm{p}}{N_\mathrm{elem}}$. In case of perfect scalability this value is constant. For our experiment the value is constant for up to four threads and but jumps by a factor 2 when increasing the number of threads to 8. This is caused by the job scheduler used on the test platform. It allocates processors in blocks of four such that processor proximity is guaranteed for up to four processors only. Using more then four processors can lead to a drop of available communication bandwidth for logically adjoining processors by more than a factor of three. The measurements suggest that T_step stays constant if more than four processors are used. A more detailed discussion of the performance of *escript* and *finley* for OpenMP can be found in [1]. The MPI version of *finley* has not been fully tested yet and timings will be published at a later stage.

6 Summary

With *escript* we have developed an environment that separates the layer of mathematical model description from the numerical techniques and data structures used when running the simulation. This ensures an high degree of reusability models and code portability. We have illustrated this for a seismic wave propagation code which can be run with different parallelization paradigms without changes to the code.

Another module of *escript* which has not been discussed in this paper is addressing the problem of representing mathematical models as *python* objects and describing their interface through XML. This infrastructure, called *modelframe*, allows coupling models on the *python* level through sharing *escript* objects and to build entire simulations from independent models. The XML description of a simulation opens the door for building user interfaces, including GUIs and grid services, automatically. Appropriate software components are currently under construction.

Acknowledgment

Project work is supported by Australian Commonwealth Government through the Australian Computational Earth Systems Simulator Major National Research Facility and the Australian Partnership for Advanced Computing, Queensland State Government through the Smart State Research Facility Fund, The University of Queensland, the Queensland Cyberinfrastructure Foundation and Silicon Graphics Inc.

References

1. Davies, M., Gross, L., Mühlhaus, H.-B.: Scripting high performance Earth systems simulations on the SGI Altix 3700. In: Proceedings of the 7th international conference on high performance computing and grid in the Asia Pacific region (2004)
2. `http://access.edu.au/content/view/78/` (September 2006)
3. Gross, L., Cochrane, P., Davies, M., Mühlhaus, H., Smillie, J.: Escript: numerical modelling in python. In: Proceedings of the Third APAC Conference on Advanced Computing, Grid Applications and e-Research (APAC05) (2005)
4. `http://www.python.org` (September 2006)
5. Zienkiewicz, O.C., Taylor, R.L.: The Finite Element Method. 5th edn., vol. 3, Butterworth Heinemann (2000)
6. Mora, P., Place, D.: Simulation of the Frictional Stick-slip Instability. Pure Appl. Geophys. 143, 61–87 (1994)
7. `http://www-unix.mcs.anl.gov/mpi/` (September 2006)
8. `http://www.stsci.edu/resources/software_hardware/numarray` (September 2006)
9. `http://www.openmp.org/` (September 2006)

10. Brownell, D.: SGI Altix 3000 Users Guide (2003)
11. Rice, J.R., Boisvert, R.F.: Solving Elliptic Problems Using ELLPACK. Springer Series in Computational Software (1985)
12. Luo, X.–L., Stokes, A.N., Barton, N.G.: Turbulent flow around a car body - report of Fastflo solutions. In: Proc. WUA-CFD Conference, Freiburg (1996)
13. Long, K.: Sundance 2.0 Tutorial. Sandia National Laboratories Technical Report SAND2004-4793 (2004)

COMODI: Architecture for a Component-Based Scientific Computing System

Zsolt I. Lázár[1,2], Lehel I. Kovács[1], and Zoltán Máthé[1]

[1] Babeş-Bolyai University,
400084 Cluj-Napoca, Romania
zlazar@phys.ubbcluj.ro
[2] University College Dublin, Belfield, Dublin 4, Ireland

Abstract. The COmputational MODule Integrator (COMODI) [1] is an initiative aiming at a component-based framework, component developer tool and component repository for scientific computing. We identify the main ingredients of a solution that would be sufficiently appealing to scientists and engineers to consider alternatives to their deeply rooted programming traditions. The overall structure of the complete solution is sketched with special emphasis on the Component Developer Tool forming the basis of COMODI. Prototypes for a framework and an automatic interface description generator are presented.

1 Introduction

As signaled already in the late sixties, when the term "software crisis" got coined [2], brute computing force growing at exponential rate is not the answer to the problem of complexity. Recently, scientists find similar obstacles standing in the way of large scale scientific software projects [3,4], and argue for a change of paradigm. These ideas can be further extended in the context of small and medium size projects that make up the bulk of the activity in the community. In [5] and [6] it is pointed out that *reuse oriented programming (ROP)* would dramatically improve the efficiency and quality of computational research. In the last decade, there has been significant progress in overcoming the thorny technical problems of component based software engineering (CBSE) in high-performance computing [7]. A number of notable efforts have been started under the umbrella of the Common Component Architecture (CCA) Forum (http://www.cca-forum.org), yielding a specification for the behavior and interaction of components and frameworks. Several complying solutions have been implemented ([8,9,10]) based on Babel, the language interoperability tool, and the associated Scientific Interface Definition Language (SIDL) [11].

While accounting for all technical questions is a necessary condition for reaching the proposed goals of CBSE, this might not be sufficient for a real impact on the community's software development practices. The COmputational MODule Integrator (COMODI) is another initiative aiming at a component based framework, component developer tools and component repository for scientific computing. The emphasis is, however, shifted toward issues pertaining to the human-computer interaction. It is based on a set of requirements formulated in the light of the conclusions of a preliminary survey made with a mixed group of computational scientists on the occasions of conferences, workshops and via an on-line form on the COMODI website [1]. Present paper briefly reviews the key concepts behind COMODI and lays out the large scale architecture of the complete suite of tools that is up to the challenges of an ROP paradigm. The formulated requirements and design strategies focus on the needs of component developers rather than those of users. In section 4 we report on the experience gathered while working with a first prototype endowed with some of the functionalities prescribed in the previous sections.

2 Requirements

For most scientists the computer is merely a tool. Therefore, whatever little time is spent on making the computer do its job, is usually a loss from the point of view of the objectives of the project. In more and more fields, "survival" without a decent level of programming and system management skills is not possible. At a global scale, the time scientists spend waiting for computing jobs to finish is a fraction of what computers spend idle waiting for the scientists to finalize the development of code. In view of these facts it is only fair to admit that "high-performance programming" should get equal attention to high-performance computing. To this end, future computing technologies should strive to make the usage of computers significantly easier. Another factor that, if ignored, can prevent a valuable software project from getting the deserved attention is the level of support for existing technologies. The new solution should be able to accommodate old-style data while newly produced data should be usable within the old framework. And finally, the new technology can only penetrate the scientific community if it is free of charge.

In order to make the above general requirements more specific we must first make a distinction between *component developers* and *end-users*: the former design and implement new components while the latter are primarily involved in assembling these into executable applications.

Since the two activities require different skills and work methods, the requirements set for the employed tools in each case also differ. For user satisfaction the solution has to be endowed with the features listed below:

– user friendly graphical interfaces;
– intuitive, high-level representation of data and processes such that the elements of low-level programming can also be clearly identified;
– possibility for low-level control;
– support for most popular hardware and software platforms;
– comprehensive component repository;
– high-performance;
– open source.

In order to fully support developers it is imperative that no compliance criteria are set for the computational code, neither in terms of structure nor used data types. In other words, any valid code written in the supported programming languages should automatically be ready for COMODI. Therefore several restrictions apply to the process of adapting existing code to COMODI:

– no change in the source code, the interface or the implementation;
– no extra coding - connectivity is achieved by supplementing author provided source-code with automatically generated glue-code;
– no need for the author to know other languages/standards than the ones used for implementing the code;
– no platform dependence - the capabilities of the system the development is carried out on is extended by on-line servers providing compilation as web service;
– no language dependence - all present and future languages should be able to communicate seamlessly;
– low performance overhead;
– support for both open source and commercial components.

3 Architecture

The complete solution should include the following elements:

– a visual programming environment for computational projects;
– component developer tools for adapting regular code to the framework;
– a distributed component repository;
– a compilation web service.

The above software elements will come with several standards, including one for global naming of components and a language for documenting their services. Figure 1 shows the COMODI architecture. The responsibilities of each part are summarized in Table 1.

The developer layer contains a user friendly *Graphical User Interface* (GUI) and a *Component Developer Tool* (CDT) with a *Parser*.

The CDT, after semi-automatically collecting information pertaining to the content, the behavior, and the representation of the component, generates a *component descriptor file* (CDF) in the XML based *Component Descriptor Language* (CDL) and the source of the *glue-code* that will intermediate the communication of the component within the COMODI framework (see figure 2). At this stage the CDF will contain all communication related information, such as exported functions and data types. It describes both syntactically and semantically the component, supports the programming style of computational scientists as far as data structures, and it is extensible. Its complexity is expected to grow together with the user community and the number of application areas. By semi-automatic we mean that the *Parser*, which stands at the basis of the tool, analyzes lexically and syntactically the source file, extracts interface information *only*, and generates a primary CDF. Using the GUI, the developer only has to confirm the exported ports, provide human readable documentation for the

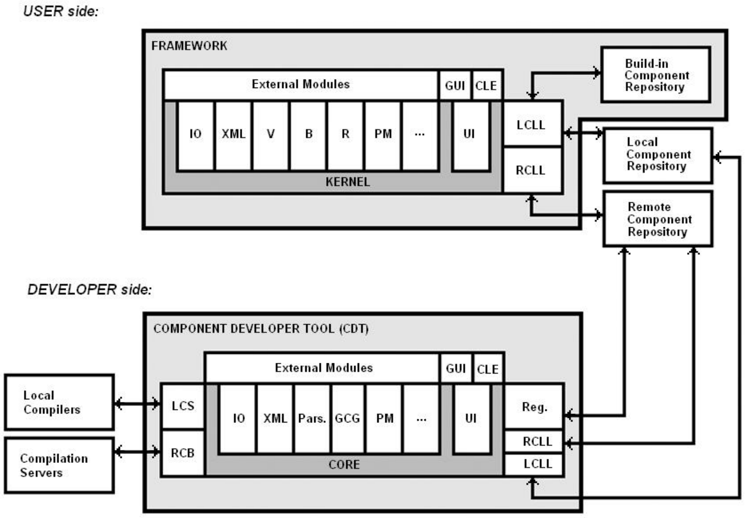


Fig. 1. Architecture of COMODI. On the user side: IO: *Input/Output System*, XML: *Extended Markup Language Parser*, V: *Validator*, B: *Binding System*, R: *Running System*, PM: *Project Manager*, UI: *User Interface*, GUI: *Graphical User Interface*, CLE: *Command Line Editor*, LCLL: *Local Component Locater and Loader*, RCLL: *Remote Component Locater and Loader*. On the developer side: LCS: *Local Compilation Service*, RCB: *Remote Compilation Broker*, Pars.: *Parser*, GCG: *Glue-Code Generator*, Reg.: *Registrar*.

Table 1. Responsibilities of the two major parts of COMODI

Role of the framework	**Role of the component developer tool**
– component assembling – project verification and validation – project execution – runtime user interaction	– assist the developer in documenting the code – generate glue-code – assist the developer in compiling the component – register the component with the global repository

component, set default values and visual representation related preferences. The CDT then contacts on-line *compilation servers* and returns ready-made binaries for the platforms of the developer's choice. The compiled library together with the descriptor file are packed into a standard format such as `tar.gz`, are uploaded by the developer to a place where it can be accessed publicly while the CDT registers the component in the *Remote Component Repository*. The deployed

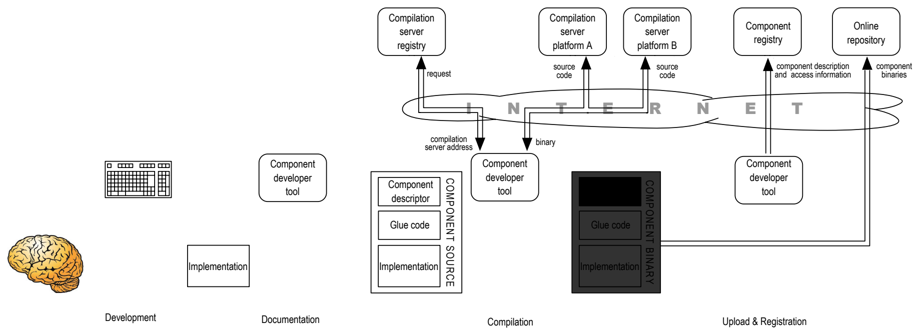


Fig. 2. Component development process

component is a package containing the component's source code - if the developer chooses to make the source open - the component descriptor file, the binaries for both the computational- and the generated glue-code, and further resources. Upon use within the COMODI framework, the component is downloaded and stored in the *Local Component Repository.*

The sources provided by the component developer suffer no changes during the component creation process. All glue-code comes as additional functions in a separate file. Not touching the source of the developer has the benefit of the compiled component being usable both within and outside the COMODI framework making COMODI components fully compatible with traditional programming environments.

4 Prototype

4.1 Framework

Language interoperability is one of the most important challenges for CBSE in high-performance computing. Fortunately, Babel can successfully handle many of the technical difficulties. Therefore, together with the CCA standard it served as a good starting point for COMODI. The present version of the COMODI framework is a partial Java implementation of the CCA specification. Only those parts that are essential for testing the ideas of interest have been covered.

We find that the CCA specification is overly comprehensive for the average user primarily interested in elementary tasks like instantiating components, discovering their properties, linking their ports, and running the so formed diagrams. Thus, a simplified add-on Java package has been created, targeting the user-framework interaction. Even though the reduced interface is mainly a subset of CCA, it adds one concept to those encountered in the CCA specification, namely *diagrams.* There is an 1:N type of association relation between diagrams and components. Diagrams are independent of each other but managed by a single framework instance. The user can create components, components, connections and diagrams. Components and connections can be added and removed from diagrams. These are executed starting from a GoPort. In this version the only pursued goal was to keep this interface minimal and free of concepts that are not immediately obvious to the user.

Since enhanced interactivity is one of the main requirements of the ultimate solution we were testing yet another layer on top of the reduced interface. The motivation behind the interactive layer not directly interfacing to the framework is that this layer is independent of both the CCA and the underlying framework, both at the interface and the implementation level. Instead of creating a completely new interaction console like in CCAFE, we chose Python for a number of reasons. To name a few: *i.* implementation and maintenance is considerably easier while the interaction is much richer. History, code completion and documentation are readily available; *ii.* the user has at hand a full-fledged and easy-to-use programming language for pre- and post-processing data and for performing auxiliary tasks such as interacting with the operating system, working with strings, etc.; *iii.* no COMODI specific syntax has to be learned by the user. A typical session would contain most of the following elements:

```
>>> from comodipython import *
>>> framework = Framework()
>>> server = framework.createcomponent("HelloWorldServer")
>>> client = framework.createcomponent("HelloWorldClient")
>>> print server.ports()
['HelloPort']
>>> print client.ports()
['GoPort', 'HelloPort']
>>> clientport = client.port('HelloPort') # uses port
>>> serverport = server.port('HelloPort') # provides port
>>> connection = framework.createconnection(clientport, serverport)
>>> diagram = framework.creatediagram('HelloWorld')
>>> diagram.add(server)
>>> diagram.add(client)
>>> diagram.add(connection)
>>> diagram.run(client.port('GoPort'))
```

4.2 Parser

The parser is implemented in such a way that it can recognize the majority of imperative programming languages, once their EBNF definitions are available. In addition to the complete interface information available in the source file, it also extracts dependencies such as function calls made from inside other functions. This feature helps in selectively including dependences from header files. The level of flexibility with respect to the source's implementation language at the input is replicated at the output. The description of the source code's interface is generated based on a user provided table of prescribed structure. Creating SIDL, Babel XML or any other format, requires almost straightforward customization of the table. Naturally, not all information for the output of a given format is available in all languages o,r there is no unique translation. These cases require human intervention. However, the appropriate output table can make the parser generate data that can directly be fed to the tool in charge for collecting the remaining information from the user. Alternatively, the raw XML output can be transformed into other formats and a higher-level description by a separate piece of software. A parser prototype can be found on the COMODI website.

Presently, the parser has been tested for C and Pascal while Java and Fortran 77 are expected to follow shortly, as the only challenges they pose are strictly implementational. Complex hybrid-languages like C++ or Fortran 95 will most probably pose more difficult challenges [12], however, as the parser is only targeting interface information, the vast majority of problems that could be encountered in case of a complete parser are avoided.

5 Future Work

As our primary concern is the component developer tool, most effort will go into creating a prototype that strikes a balance between the requirements of minimial human intervention during the component creation process and that of a large enough scope so that it would be useful.

Presently, connecting two components requires a very tight match between the corresponding uses and provides ports. Babel based frameworks cannot directly link a client component using an interface defined as `A {f:F}` to a server component providing an essentially equivalent interface `AA {ff:F}`, where `A` and `AA` are the interface names, `f` and `ff` are method names, and `F` is the type of the two methods. Even though some of the emerging issues are not merely technical, several solutions are available. With this obstacle moved out of the way, one still needs more flexible subtyping rules that allow the connection of a uses port with a method such as `f(p1:T1, p2:T2)` to provides ports like `ff(p2:T2, p1:T1)` or `ff(p1:T1, p2:T2, p3:T3 = v3)`, where `p1, p2, p3` are formal parameters, `T1, T2, T3` denote data types and `v3` stands for a default value. Higher-level programming will require the framework and the component developer tools to fill in automatically or semi-automatically small incompatibility gaps like those above by generating appropriate glue-code and by hidden or user controlled connector components. We also consider that Babel's SIDL, if it is to serve as a component IDL, should be extended to cover the declaration of interfaces for uses ports.

Pushing the automation level of the interface definition via the parser will raise thorny issues such as how to disentangle environment dependences from component interdependences, and where is the borderline between the framework and the rest of the system. The ideal solution would require a consistent component based construction of the whole operating system and all third party software.

On the user side, the prototype for a GUI should ideally be preceded by a study on how the representation of component networks and their interaction with the user is preferred by the different groups in scientific computing. The interface of Ccaffeine provides a high-level view on the component diagram.

Alternatively, the GUI features promoted in [6] reflect the actual low-level structure of component interfaces, allowing the user to have the control that is typical for languages like C and Fortran. On the other hand, this low-level control requires extra effort from the user that on the long run might not be acceptable.

6 Conclusions

The transition of computational research toward a reuse oriented programming paradigm is conditioned not only by the existence of sophisticated tools capable of integrating several technologies but also by the extent these can bring about efficiency in the work of scientists already overwhelmed by the technicalities plaguing today's software solutions. We attribute the lack of impact of present solutions to code reuse to the high threshold of effort required, in many cases made worse by a closed source and restrictive copyrights. The paper presents the general requirements and a few design guidelines for a complete reuse oriented solution for computational scientists. We argue that the community needs a solution that allows a smooth, effortless transition to the new paradigm. The COMODI project is an attempt to identify these requirements, design complying solutions and create proof-of-concept prototypes following these principles. Our prototype framework was built on Babel and the CCA specification which proved to be powerful tools. On the other hand, we found that the tasks are not purely technical and several principle questions have to be addressed on the way toward an ubiquitous CBSE.

Acknowledgments. This work is supported by the National University Research Council of Romania with grant no. 27687/14.03.2005 and assisted by the Marie-Curie Grant no. MTKD-CT-2004-003134. One of the authors (Zs.I.L) thanks Ronan McNulty for his valuable comments.

References

1. COMODI homepage: `http://comodi.phys.ubbcluj.ro`
2. Naur, P., Randell, B.: Software Engineering: Report on 1968 NATO Conference, NATO (1969)
3. Post, D.E.: The Coming Crisis in Computational Science. In: Proceedings of the IEEE International Conference on High Performance Computer Architecture: Workshop on Productivity and Performance in High-End Computing, Madrid, Spain, February 14, 2004 (2004)
4. Post, D.E., Votta, L.G.: Computational Science Demands a New Paradigm. Phys. Today, 35 (January 2005)
5. Lázár, Zs.I., Pârv, B., Fanea, A., Heringa, J.R., de Leeuw, S.W.: COMODI: Guidelines for a Component Based Framework for Scientific Computing. Studia Babeş-Bolyai, Series Informatica, vol. XLIX(2), p. 91 (2004)
6. Lázár, Z.I., Fanea, A., Ciobotariu-Boer, V., Petraşcu, D., Pârv, B.: COMODI: On the Graphical User Interface. In: Proceedings of the 7th IEEE Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timişoara, IEEE Computer Society Press, Los Alamitos (2005)

7. Allan, B.A., et al.: A Component Architecture for High-Performance Scientific Computing. The International Journal of High Performance Computing Applications 20(2), 163 (2006)
8. Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., Bramley, R.: Merging the CCA component model with the OGSI framework. In: 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (2003)
9. Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., Wolfe, P.: Ccaffeine - a CCA component framework for parallel computing (2003), `http://www.cca-forum.org/ccafe/`
10. Zhang, K., Damevski, K., Venkatachalapathy, V., Parker, S.: SciRun2: A CCA framework for high performance computing. In: Proceedings of the 9th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE Computer Society Press, Los Alamitos (2004)
11. Lawrence Livermore National Laboratory: Babel homepage (2004), `http://www.llnl.gov/CASC/components/babel.html`
12. Quinlan, D., Yi, Q., Kumfert, G., Epperly, T., Dahlgren, T., Schordan, M., White, B.: Toward the Automated Generation of Components from Existing Source Code. In: The Second Workshop on Productivity and Performance in High-end Computing, San Francisco (February 2005)

Workload Characterization Using the TAU Performance System

Sameer Shende, Allen D. Malony, and Alan Morris

Performance Research Laboratory,
Department of Computer and Information Science,
University of Oregon, Eugene, OR 97403, USA
{sameer,malony,amorris}@cs.uoregon.edu

Abstract. Workload characterization is an important technique that helps us understand the performance of parallel applications and the demands they place on the system. It can be used to describe performance effects due to application parameters, compiler options, and platform configurations. In this paper, workload characterization features in the TAU parallel performance system are demonstrated for elucidating the performance of the MPI library based on the sizes of messages. Such characterization partitions the time spent in the MPI routines used by an application based on the type of MPI operation and the message size involved. It requires a two-level mapping of performance data, a unique feature implemented in TAU. Results from the NPB LU benchmark are presented. We also discuss the use of mapping for memory consumption characterization.

Keywords: Performance mapping, measurement, instrumentation, performance evaluation, workload characterization.

1 Introduction

Technology for empirical performance evaluation of parallel programs is driven by the increasing complexity of high performance computing environments and programming methodologies. To keep pace with the growing complexity of large scale parallel supercomputers, performance tools must provide for the effective instrumentation of complex software and the correlation of runtime performance data with system characteristics. Workload characterization is an important tool for understanding the the nature and performance of the workload submitted to a parallel system. Understanding the workload characteristics helps in correlating the effects of architectural features on workload behavior. It helps us in planning system capacity based on an assessment of the demands placed on the system, and in identifying which components in a system may need to be upgraded. This is a *systems perspective* on workload characterization. There is also an *application perspective* that characterizes application-specific performance behavior in the context of workload and platform aspects. For instance, in this paper, we use workload characterization techniques recently implemented in the TAU performance system [1] to study message communication performance.

Workload characterization methods collect performance data for each application in the workload set. For instance, performance profiles can contain statistics on performance in application regions (e.g., routines) and with respect to specific behaviors, such as message communication based on the message size. Profiling tools that focus their attention on capturing aggregate performance data over all invocations of message communication and I/O routines ignore the performance variation for small and large buffer sizes. It is this ability to expose application features and observe their performance effects that we are interested in supporting as part of a workload characterization methodology.

In this paper, we describe the techniques for measuring the performance of a parallel application's message communication based on message buffer sizes. When this information is gathered from several applications and stored in a performance database, we can classify the performance of the entire system using histograms that show the time spent in inter-process communication and I/O routines based on buffer sizes. We discuss the improvements that we made to the TAU performance system [1] in the areas of instrumentation, measurement and analysis to support workload characterization. Section §2 describes the related work in this area, Section §3 describes the TAU performance system, and describes how performance mapping is applied to characterize the performance of MPI routines based on the message sizes. Section §5 reports on our experience with message communication characterization of the NPB LU benchmark. We have also applied performance mapping to memory usage characterization. Brief discussion is given to workload characterization of memory consumption. Section §6 concludes the paper and we discuss future work in this section.

2 Related Work

Workload characterization is a rich area in performance evaluation research. Our specific interest is in workload characterization for high-performance computing. There are two projects of related interest to our work.

The OpenWLC [2] system is a scalable, integrated environment for systematically collecting the monitored data and applying workload characterization techniques to raw data produced by monitoring application programs. OpenWLC's framework employs a component-based, multi-tier, architecture to cope with large amounts of monitored data during collection, storage, visualization and analysis stages.

IPM [3] is an integrated performance monitoring system developed at the Lawrence Berkeley Laboratory (LBL) for use at the National Energy Research Supercomputing Center (NERSC). IPM is in active use for application performance analysis and workload characterization. Specific to our interests, IPM can characterize the application performance based on message sizes. It uses library preloading mechanisms for instrumenting an application under Linux and on other platforms where preloading of shared libraries is available. The performance data is stored in a performance data repository which can be queried for application characteristics based on a number of parameters such as execution

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
  int rank, size, i, j;
  int buffer[16*1024*1024];
  MPI_Init(&argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  for (i=0;i<1000;i++)
    for (j=1;j<16*1024*1024;j*=2) {
      if (rank == 0) {
        MPI_Send(buffer,j,MPI_INT,1,42,MPI_COMM_WORLD);
      } else {
        MPI_Status status;
        MPI_Recv(buffer,j,MPI_INT,0,42,MPI_COMM_WORLD,&status);
      }
    }
  MPI_Finalize();
}
```

Fig. 1. Message Size Characterization Instrumentation

date and MPI performance data. LBL has implemented a web-based interface for this purpose.

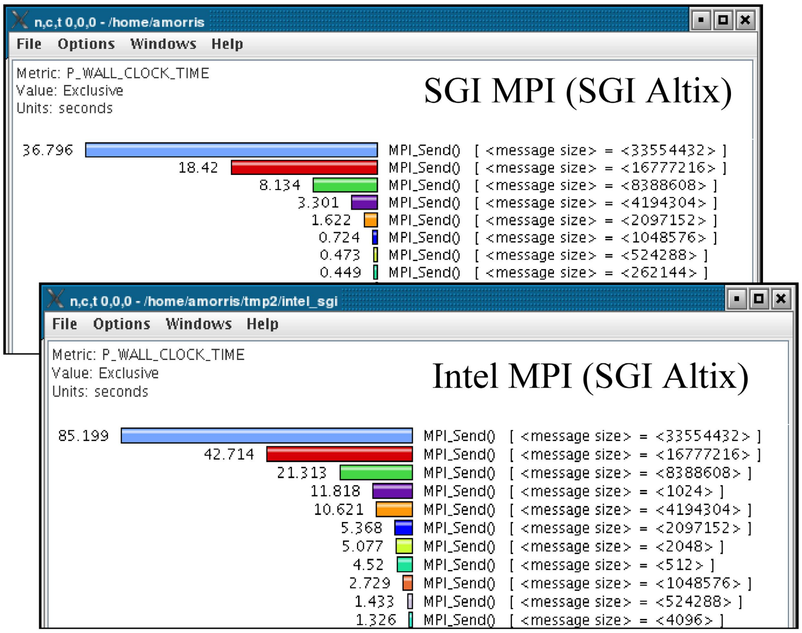
Certainly, other application performance measurement tools can be applied to workload characterization. However, the ability to store multi-experiment performance data, including metadata about compiler and system parameters, is important criteria for workload characterization support. PerfSuite [4] is a performance toolkit that builds a performance data repository based on execution time and hardware performance counters [5] to characterize the performance of an application and the system. TAU can work with PerfSuite and other tools to integrate performance results across applications and platforms.

3 Workload Characterization and Performance Mapping

Workload characterization analyzes the effects of application execution in a system context. Application measurements could be made of total performance, such as total execution time, but finer granularity measurements can better identify workload effects specific to program properties. However, certain properties require a measurement system that can observe execution parameters and characterize application performance based on unique parameters instances. The general concept is one of *peformance mapping*, wherein an association can be established between low-level performance data and high-level measurement abstractions, specialized by program semantics. The TAU performance system is able to support performance mapping for workload characterization.

TAU[1] is an integrated, configurable, and portable profiling and tracing toolkit. It provides support for portable instrumentation, measurement, and analysis. In-

```
% icc mpi.c -lmpi
% mpirun -np 2 tau_load.sh -XrunTAU-icpc-mpi-pdt a.out
```


n,c,t 0,0,0 - /home/amorris
File Options Windows Help
SGI MPI (SGI Altix)
n,c,t 0,0,0 - /home/amorris/tmp2/intel_sgi
File Options Windows Help
Intel MPI (SGI Altix)

```
Metric: P_WALL_CLOCK_TIME
Value: Exclusive
Units: seconds

36.796  MPI_Send()  [ <message size> = <33554432> ]
 18.42  MPI_Send()  [ <message size> = <16777216> ]
 8.134  MPI_Send()  [ <message size> = <8388608> ]
 3.301  MPI_Send()  [ <message size> = <4194304> ]
 1.622  MPI_Send()  [ <message size> = <2097152> ]
 0.724  MPI_Send()  [ <message size> = <1048576> ]
 0.473  MPI_Send()  [ <message size> = <524288> ]
 0.449  MPI_Send()  [ <message size> = <262144> ]
```

```
Metric: P_WALL_CLOCK_TIME
Value: Exclusive
Units: seconds

85.199  MPI_Send()  [ <message size> = <33554432> ]
42.714  MPI_Send()  [ <message size> = <16777216> ]
21.313  MPI_Send()  [ <message size> = <8388608> ]
11.818  MPI_Send()  [ <message size> = <1024> ]
10.621  MPI_Send()  [ <message size> = <4194304> ]
 5.368  MPI_Send()  [ <message size> = <2097152> ]
 5.077  MPI_Send()  [ <message size> = <2048> ]
  4.52  MPI_Send()  [ <message size> = <512> ]
 2.729  MPI_Send()  [ <message size> = <1048576> ]
 1.433  MPI_Send()  [ <message size> = <524288> ]
 1.326  MPI_Send()  [ <message size> = <4096> ]
```

Fig. 2. MPI (SGI vs. Intel) Message Characterization

strumentation calls can be inserted in TAU at the source level, the library level, the binary code level, and even in a virtual machine. Unique in the TAU performance system is an instrumentation API for performance mapping. It uses the SEAA model [6] of mapping that provides support for both embedded and external associations. External associations use an external map (implemented as a hash table) to access performance data using a user specified key. The performance data is collected for interval events or atomic events that are triggered at a certain place in the program code. Performance mapping is a powerful concept and technology. It has been used in TAU for callpath profiling [1] and phase profiling [7]. Context events that map atomic events to the currently executing application callstack, are also implemented using TAU's mapping capabilities. Here we apply performance mapping to MPI communication characterization.

TAU's MPI wrapper interposition library helps us track the time spent in each MPI call. It defines a separate name-shifted MPI interface for each MPI routine that can be used to invoke timer calls at routine entry and exit. This mechanism can also be used to access arguments that flow through the MPI routines. Hence, measurement code could be created to track the sizes of messages for each MPI call. We have followed this approach using TAU's mapping technology to implement a two-level map of the MPI call ID and the size of the message buffer used in the call. With this data, we can determine if a given message buffer size and call have occured in the past. If not, a new performance structure is created with a name that embeds the MPI call ID and buffer size. At the end of the application, we obtain the performance in each invoked MPI call for each message size used.

In general, TAU can take any routine parameter and create a performance mapping. The measurement library implements routines for different parameter types, such as `TAU_PROFILE_PARAM1L(value, "name")`. The following code segment shows how this is used:

```
void foo(int input) {
  TAU_PROFILE("foo", "", TAU_DEFAULT);
  TAU_PROFILE_PARAM1L(input, "input");
  ...
}
```

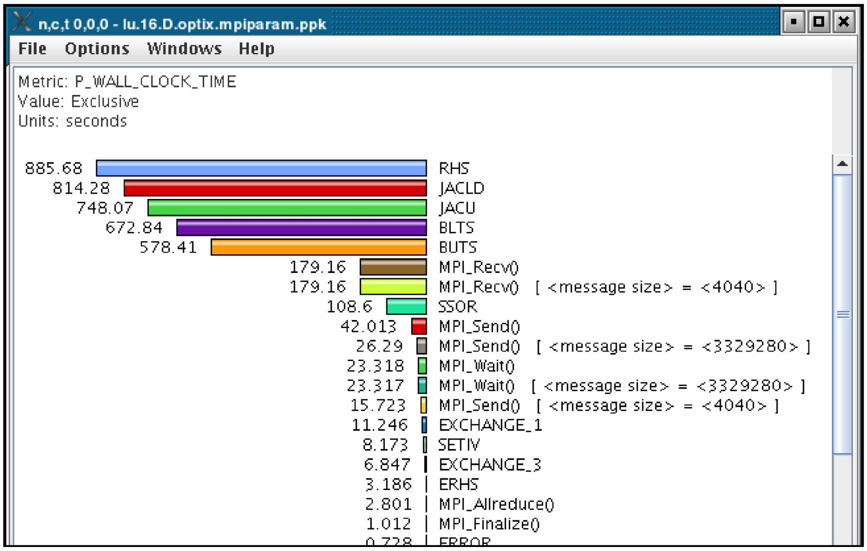
When the measurement library is configured with `-PROFILEPARAM`, the parameter mapping API is enabled.

Figure 1 shows a simple program for message communication of different sizes. Figure 2 shows profile output characterizing communication performance for different MPI libraries, SGI and Intel. With such information, we can obtain a better understanding of workload effects. Also shown is the experiment compilation and run commands.

4 Performance Experimentation

Performance experimentation and results management are important components for any workload characterization system. The use of the TAU performance system involves the coordination of several steps: instrumentation selection, measurement configuration, compilation and linking with the application, application execution and generation of performance data on the target platform, and performance data storage for analysis. We describe the sequence of these steps as a *performance experiment*. We use the term experiment generally to denote a specific choice of instrumentation and measurement for a specific application code, but what this means exactly should be left to the user. We define the term *trial* to mean an instance of an experiment. A trial might either repeat an experiment run (e.g., to determine performance variation) or modify an experiment run parameter (e.g., number of processors), which would not represent such a significant change as to constitute a new experiment.

The performance data gathered from executing the application is stored in TAU's performance database, PerfDMF [8] which is then queried by the ParaProf profile browser and other analysis tools such as PerfExplorer [9] for performance data mining operations. The performance data stored in PerDMF is multi-variate and multi-dimensional, both within single trials and experiments as well as across experiments, applications, and platforms. PerfExplorer is a framework for parallel performance data mining and knowledge discovery – finding


n,c,t 0,0,0 - lu.16.D.optix.mpiparam.ppk
File Options Windows Help

```
Metric: P_WALL_CLOCK_TIME
Value: Exclusive
Units: seconds

885.68   RHS
814.28   JACLD
748.07   JACU
672.84   BLTS
578.41   BUTS
179.16   MPI_Recv()
179.16   MPI_Recv()  [ <message size> = <4040> ]
108.6    SSOR
42.013   MPI_Send()
26.29    MPI_Send()  [ <message size> = <3329280> ]
23.318   MPI_Wait()
23.317   MPI_Wait()  [ <message size> = <3329280> ]
15.723   MPI_Send()  [ <message size> = <4040> ]
11.246   EXCHANGE_1
8.173    SETIV
6.847    EXCHANGE_3
3.186    ERHS
2.801    MPI_Allreduce()
1.012    MPI_Finalize()
0.728    ERROR
```

Fig. 3. Profile of LU Benchmark on SGI Altix

out new performance facts and relationships as the outcome of searching and analyzing the stored performance data.

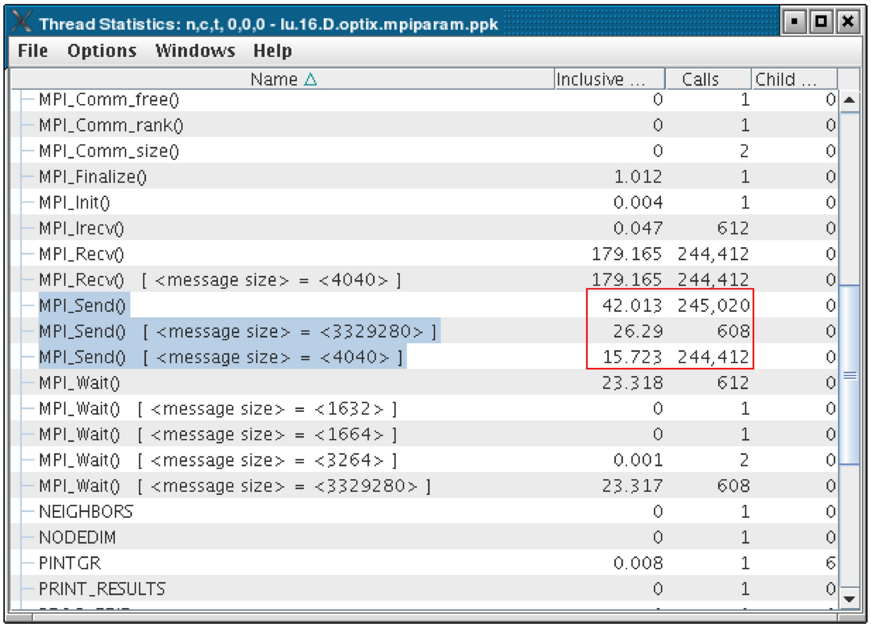
5 Workload Characterization Experiments

To demonstrate TAU's mapping support for workload characterization, the NAS parallel benchmark LU is used as a testcase. Specifically, we are interested in understanding how this MPI benchmark behaves respective of its message communication. TAU's message size mapping was enabled and experiments were run on a SGI Altix platform. We also can capture memory usage statistics using mapping technologies.

5.1 MPI Message Size Characterization

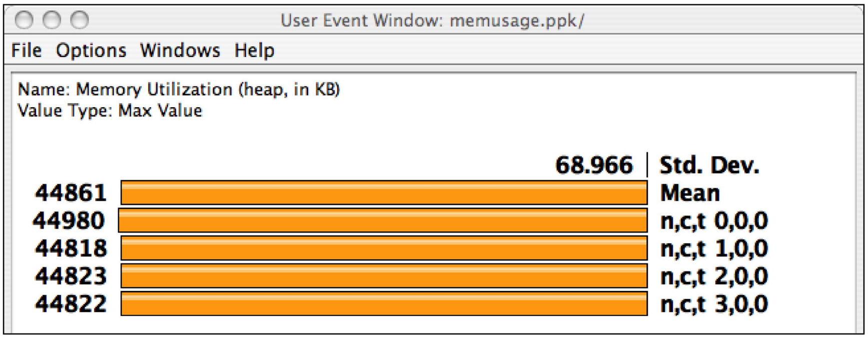
Each performance experiment ran captured execution time performance for the LU routines. For the MPI routines, execution time performance was broken down based on message size. Figure 3 shows an example (flat) parallel profile for one process of a 16-process LU exectuion. Seen are the times spent in routines in decreasing order. Most of the time is spent in computation, but message communication is also significant. The communication event IDs encode the size of the message in the names. The majority of the `MPI_Recv` time was spent receiving messages with 4040 bytes.

Further analysis of the message characterization shows the distribution of each MPI operation across the message size used for that operation. Figure 4 highlights the inclusive time of `MPI_Send` and the number of calls for one LU


Thread Statistics: n,c,t, 0,0,0 - lu.16.D.optix.mpiparam.ppk
File Options Windows Help

Name △	Inclusive ...	Calls	Child ...
MPI_Comm_free()	0	1	0
MPI_Comm_rank()	0	1	0
MPI_Comm_size()	0	2	0
MPI_Finalize()	1.012	1	0
MPI_Init()	0.004	1	0
MPI_Irecv()	0.047	612	0
MPI_Recv()	179.165	244,412	0
MPI_Recv() [<message size> = <4040>]	179.165	244,412	0
MPI_Send()	42.013	245,020	0
MPI_Send() [<message size> = <3329280>]	26.29	608	0
MPI_Send() [<message size> = <4040>]	15.723	244,412	0
MPI_Wait()	23.318	612	0
MPI_Wait() [<message size> = <1632>]	0	1	0
MPI_Wait() [<message size> = <1664>]	0	1	0
MPI_Wait() [<message size> = <3264>]	0.001	2	0
MPI_Wait() [<message size> = <3329280>]	23.317	608	0
NEIGHBORS	0	1	0
NODEDIM	0	1	0
PINTGR	0.008	1	6
PRINT_RESULTS	0	1	0

Fig. 4. Message Size Characterization for LU Benchmark



Name: Memory Utilization (heap, in KB)
Value Type: Max Value

68.966	Std. Dev.
44861	Mean
44980	n,c,t 0,0,0
44818	n,c,t 1,0,0
44823	n,c,t 2,0,0
44822	n,c,t 3,0,0

Fig. 5. Memory Consumption Tracking for LU Benchmark

process. Here is it seen that relatively large number of small messages were sent, accounting for approximately 37% of `MPI_Send`'s total time.

5.2 LU Memory Usage Characterization

TAU performance mapping can also be used to characterize memory usage. This can show how memory is allocated, in what size chunks, and the amount of free space available. Figures 5 displays the heap memory utlization for LU on four processes.

6 Conclusion

In the process of workload characterization for high performance parallel systems, it is important to have portable and configurable tools that can target the different performance features and experiments of interest. Presently, the TAU performance system has such capabilities for steps in this process, from common event instrumentation, profile and trace measurements, and data analysis to meet workload characterization objectives. A novel feature of TAU is its performance mapping technology. The presentation above demonstrates how mapping can be used to characterize message communication and memory usage.

Our objectives in the future include better support for experiment automation and knowledge discovery for workload characterization. We are also working to integrate our tools with IPM.

Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-05ER25663, DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science.

References

1. Shende, S., Malony, A.D.: The TAU Parallel Performance System. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
2. Ong, H., Subramaniyan, R., Leangsuksun, C., Studham, S.: OpenWLC: A Scalable Workload Characterization System. In: High Availability and Performance Workshop, in conjunction with Sixth LACSI Symposium (October 11-13, 2005), `http://xcr.cenit.latech.edu/wlc/index.php?title=PUBLICATIONS`
3. Borrill, J., Carter, J., Oliker, L., Skinner, D., Biswas, R.: Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms. In: Proc. of International Conference on Parallel Processing (ICPP 2005), pp. 119–128. IEEE, Los Alamitos (2005)
4. Kufrin, R.: PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux. In: Proceedings of the 6th International Conference on Linux Clusters: The HPC Revolution 2005 (LCI-05) (2005)
5. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications 14(3), 189–204 (2000)
6. Shende, S.: The Role of Instrumentation and Mapping in Performance Measurement. Ph.D. Dissertation, University of Oregon (August 2001)
7. Malony, A.D., Shende, S., Morris, A.: Phase-Based Parallel Performance Profiling. In: Proceedings of the PARCO 2005 conference (2005)
8. Huck, K.A., Malony, A.D., Bell, R., Morris, A.: Design and Implementation of a Parallel Performance Data Management Framework. In: Proceedings of International Conference on Parallel Processing (ICPP 2005), IEEE Computer Society, Los Alamitos (2005)
9. Huck, K.A., Malony, A.D.: PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing. In: SC 2005, ACM, New York (2005)

Grid Data Management: Minisymposium Abstract

Siegfried Benkner[1] and Heinz Stockinger[2]

[1] University of Vienna, Austria
[2] Swiss Institute of Bioinformatics, Lausanne, Switzerland

Data management is one of the most fundamental aspects in a Data Grid where potentially Tera- or Petabytes are stored in distributed sites. Although there are many solutions to data management, mass storage systems and file system features are among the most dominant ones. This minisymposium will take out a few of these aspects and discuss them in more detail.

Supporting SLA Negotiation for QoS-Enabled Simulation Services in a Medical Grid Environment[⋆]

Siegfried Benkner[1], Gerhard Engelbrecht[1],
Stuart E. Middleton[2], and Mike Surridge[2]

[1] Institute of Scientific Computing, University of Vienna,
Nordbergstrasse 15/C3, 1090 Vienna, Austria
sigi@par.univie.ac.at
[2] IT Innovation Centre, University of Southampton, UK

Abstract. Many advanced medical simulation applications are based on compute intensive numerical methods often exceeding the computational capacity available in hospitals and clinical centers. Addressing this issue, the EU Project GEMSS has developed a Grid infrastructure that supports the on-demand provision of medical simulation services running on remote parallel computers over the Internet. A flexible QoS infrastructure supporting dynamic negotiation of service level agreements and advance reservation of compute resources facilitates using Grid services for supporting time-critical clinical procedures. In this paper we present an overview of the GEMSS Grid infrastructure and outline the main issues involved in the provision of QoS-enabled Grid services and the negotiation of service level agreements for ensuring response time and price guarantees.

1 Introduction

Many advanced medical simulation applications are based on computationally demanding methods such as Finite Element Modeling, Computational Fluid Dynamics and Monte Carlo simulation, usually requiring the availability of a high performance computing infrastructure. In order to make such applications available to hospitals and clinical centers without the need of acquiring HPC systems, the EU project GEMSS [9] has developed a service-oriented Grid infrastructure, based on standard Web services technologies, for the provision of advanced simulation services over the Internet. Besides security and legal issues involved in the processing of patient related data [17], a key challenge was the development of mechanisms for ensuring the timeliness of results for time-critical simulation services. To address this issue, GEMSS has developed a flexible Quality of Service (QoS) infrastructure supporting dynamic negotiation of Service Level

[⋆] The work was supported by by the European Union's GEMSS (Grid-Enabled Medical Simulation Services) Project under contract IST 2001-37153 and by the Austrian Science Fund as part of the AURORA project under contract SFB F011-02.

Agreements (SLAs) for ensuring response time and price guarantees. Service providers may expose parallel simulation applications running on clusters as QoS-enabled Grid services capable of automatically negotiating with clients response time and price guarantees. Grid clients are able to choose from several service providers before agreeing to book a specific service. Once a client has found a suitable service provider, a corresponding SLA is signed and exchanged to commit both parties before a service is accessed.

GEMSS adopts a reservation-based approach to QoS coupled with application-specific performance models, advance reservation mechanisms [13], and client-driven negotiation of service level agreements subject to price constraints. Besides explicitly negotiable QoS guarantees like response time and price, the GEMSS infrastructure provides implicit QoS by realizing highest security levels and providing support for error recovery.

In the course of the project, six Grid-enabled medical simulation applications [12] have been developed and tested at sites in several EU countries. The GEMSS applications can be characterized by a relatively small number of time consuming jobs requiring powerful parallel computers in order to meet the tight time-constraints usually required during clinical procedures.

The remainder of this paper is structured as follows. Section 2 provides an overview of the GEMSS Grid infrastructure and the QoS configuration of application services. Section 3 describes the service-side QoS management infrastructure, the generation of QoS offers, and the negotiation of service level agreements. Section 4 presents experimental results with a medical image reconstruction service, followed by related work and conclusions in sections 5 and 6.

2 GEMSS Grid Infrastructure

The GEMSS Grid infrastructure relies on a service-oriented architecture based on standard Web service technologies. A generic service provision environment enables service providers to expose parallel simulation applications as services that can be accessed on-demand by clients subject to dynamically negotiated QoS constraints in the form of SLAs following the Web Service Level Agreement specification (WSLA) [24].

GEMSS services are defined via WSDL and securely accessed using SOAP messages. Supported security mechanisms comprise PKI, HTTPS, WS Security and an end-to-end security protocol for separate encryption of sensitive portions of the transferred data. In order to support the development of client applications that interact with Grid services, a component-based client framework and a high-level application programming interface (API) is provided. Furthermore, GEMSS provides a registry environment for setting up service registries and a certificate authority for issuing X.509 compliant certificates. The GEMSS Grid infrastructure has been implemented in Java and relies on the open-source frameworks Apache/Tomcat and Axis for service hosting and deployment.

2.1 Grid Application Services

The GEMSS Grid environment provides a generic service provision framework and an intuitive deployment tool for automating the task of providing parallel applications installed on clusters or other HPC platforms as Grid services, hiding the details of Grid and Web Service technologies from service providers.

Service provision is based on the concept of generic application services. A generic application service is composed of configurable components with common operations for job handling, error recovery, and QoS management. Configuration of a Grid application service usually comprises the specification of input/output file names and of scripts for starting job execution and for gathering status information. This information is stored internally in an XML application descriptor. Upon deployment, a generic application service is transformed into a Web service with a corresponding WSDL interface and deployed in the GEMSS hosting environment.

2.2 QoS Configuration

In order to enable the provision of response time and price guarantees, GEMSS services have to be QoS-enabled. For this purpose an application-specific performance model for estimating the runtime of potential service requests, and a pricing model for performing price calculations have to be provided. GEMSS prescribes for these models only an abstract interface with XML-based input and output descriptors. Each descriptor comprises one ore more parameters which have to be specified during service configuration using the deployment tool.

The performance model receives as input a request descriptor with a set of request parameters representing application-specific input meta data supplied by the client during QoS negotiation, and generates a performance descriptor typically comprising estimates for the required runtime, memory, and disk space. The pricing model takes as input a resource descriptor and calculates the price for the required resources. Finally, a machine descriptor has to be specified containing details about the compute resources (e.g. maximum number of processors) that may be made available for an application service (see Section 3 for more details).

2.3 GEMSS Client Applications

For constructing client-side applications that interact with remote Grid services, GEMSS provides a light-weight component framework and a high-level client application programming interface (API). Ready-to-use components are provided for session management, service discovery, QoS negotiation, job handling, and logging. Different versions of components may be dynamically loaded as required. The API hides the complexity of dealing with remote services from the client application developer by providing appropriate service proxies with a simple interface for accessing remote application services.

GEMSS relies on a purely client-driven model for accessing Grid services. First, there is usually an offline business workflow to open an account with potential service providers, agree on a payment mechanism and legal issues. The quality of service negotiation is then run to request offers from service providers who could run the clients job; this usually results in a QoS contract being agreed with a single service provider. The client then uploads the job input data to the service provider, starts the job, monitors its progress, and finally downloads the results. The reason for this client driven service access model is the requirement to operate with firewalls and not tunnel holes through them.

3 QoS Infrastructure and SLA Negotiation

The GEMSS Grid infrastructure enables a client to negotiate with one or more service providers the required QoS constraints for individual jobs in the form of service level agreements following the Web Service Level Agreement specification (WSLA) [24]. In order to support clinical procedures, simulation services are usually configured to support WSLA parameters for specifying the required begin and end time. To enable the use of GEMSS services in a commercial context the price of a service request may be negotiated as well.

The GEMSS service provider infrastructure employs a flexible QoS manager that can be configured by the service provider with an application specific performance model and a pricing model in order to determine the best possible QoS offer for a service request. In order to ensure the availability of appropriate computing resources for a service request, a backend scheduling system with support for advance reservation is utilized.

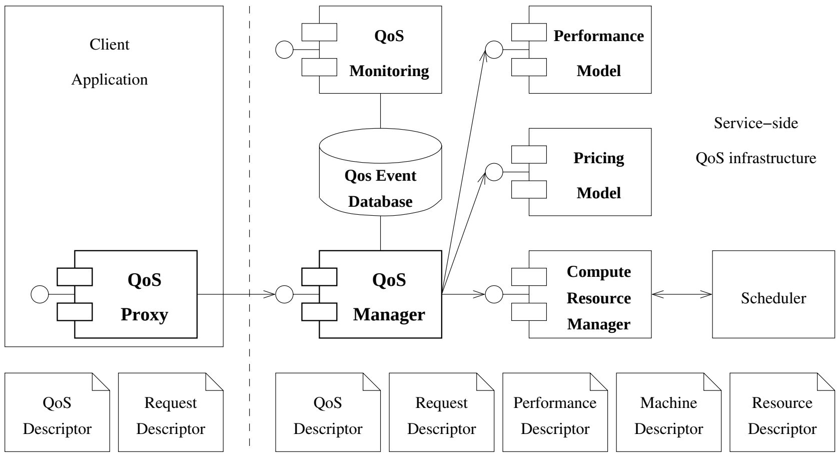


Fig. 1. QoS infrastructure

3.1 QoS Manager

The service-side QoS infrastructure is centered around the QoS manager which provides a high level interface for QoS negotiation to clients. The QoS manager receives a QoS request from a client, checks whether the client's QoS constraints can be met, and generates a corresponding QoS offer which is returned to the client. If the client decides to accept an offer a corresponding service level agreement (SLA), which defines the agreed constraints for individual jobs in the form of SLA parameters, is established and signed by both parties. As illustrated in Figure 1, the QoS manager interacts with the application performance model, the compute resource manager, and the pricing model using XML-based descriptors as explained below. QoS descriptors are based on a subset of WSLA, and represent, depending on the state of a negotiation, QoS requests, QoS offers, or, once an agreement has been achieved, QoS contracts. A QoS event data base and an associated monitoring component provide support for diagnosis and auditing.

3.2 Performance Model and Pricing Model

The performance model is used to estimate the run time and other performance relevant data for a service request. It takes as input a request descriptor and returns a performance descriptor. The request descriptor, supplied by the client during QoS negotiation, contains application specific meta-data about a specific service request. For example, in the case of an image reconstruction service, request parameters typically include image size and required accuracy. The returned performance descriptor usually contains the estimates for the execution time, the required memory, and the required disk space. In the case of a MPI applications, the performance model is usually parameterized by the number of processors. It may be executed repeatedly with a varying number of processors until the time constraints set by the client are met, or the range of feasible processors as specified in the machine descriptor is exceeded.

GEMSS supports a flexible pricing policy that can be customized for each service a service provider supports. Two pricing models have been realized, a fixed price telephone pricing model where users are charged at a prearranged CPU hour rate, and a dynamic pricing model where the CPU hour rate is dependent on the current load levels the service provider is experiencing.

GEMSS does not prescribe the actual nature of performance and pricing models, only an abstract interface is prescribed. The choice of model implementation is left to the service provider, with each model implemented as a Java library that can be plugged in and selected dynamically. For example, a performance model could be implemented based on an analytical model, or where this is not feasible, a neural network or a database could be used to relate typical problem parameters to resource needs like main memory, disk space and execution time.

3.3 Compute Resource Manager

The compute resource manager provides an interface to the scheduler for obtaining information about the actual availability of computing resources. It is utilized by the QoS manager for creating temporary reservations during QoS negotiation. The compute resource manager takes as input the performance descriptor generated by the performance model, and generates a resource descriptor containing details about temporarily reserved resources. The resource descriptor is then used as input to the pricing model to determine the price for a service request. Currently a compute resource manager is available for two scheduling systems which provide support for advance reservation, the Maui scheduler [14] and COSY [6].

3.4 SLA Negotiation and Generation of QoS Offers

The basic QoS negotiation in GEMSS is based on a request-offer model where the client requests offers from service providers. If the client agrees to an offer, it is confirmed by the client and signed by both parties resulting in a QoS contract in the form of a WSLA. Figure 2 shows the basic negotiation process and the current QoS manager strategy for generation of QoS offers.

In an initial task the client may access a GEMSS registry service to obtain a list of candidate services. The client then invokes for each candidate service the operation `requestQosOffer`, passing along a request descriptor with input meta data and a QoS request document with the required QoS constraints.

On the service side, the QoS manager executes the performance model with the request descriptor as input and compares the estimated execution time in the resulting performance descriptor with the time constraints specified in the

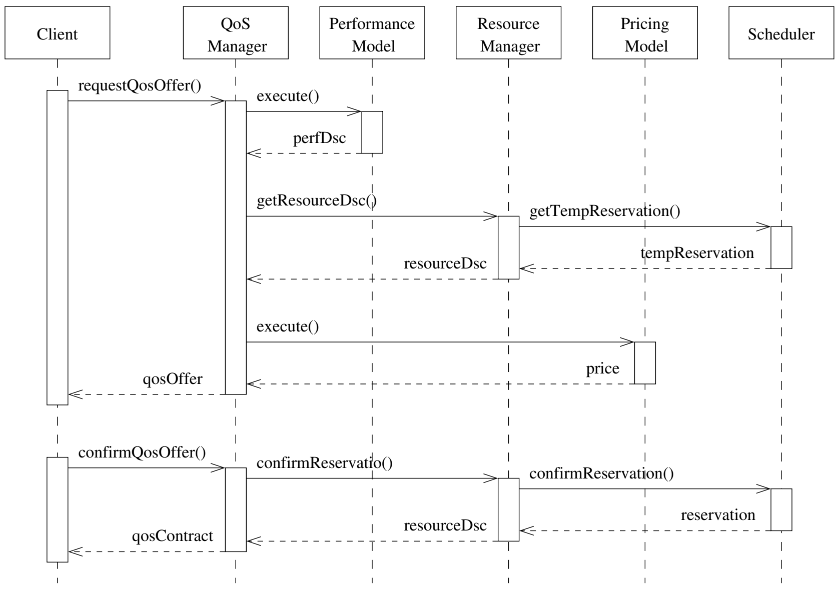


Fig. 2. QoS negotiation and offer generation

QoS request. If the client's execution time constraints can be met, the QoS manager instructs the resource manager to check whether the required resources can be made available. If this is the case, the QoS managers invokes the operation `getResourceDsc` of the resource manager passing along the performance descriptor and the QoS request document. The resource manager contacts the scheduler to check whether the required resources as specified in the performance descriptor (number of processors for the estimated runtime) can be made available within the time frame (begin time, end time) specified in the client's QoS request. If this is possible, a temporary reservation is made with the scheduler and a corresponding resource descriptor is returned. The QoS manager then executes the pricing model, passing as argument the resource descriptor, to determine if the price for the required resources is within the client's price constraints. If the price constraints can be met, the QoS manager generates a corresponding QoS offer, and returns it to the client.

If the time or price constraints can not be met, the QoS manager may execute the performance model with a different number of processors (as specified in the machine descriptor). If the clients QoS constraints cannot be met at all, no offer is generated.

On the client side, the QoS offers from different service providers are received and analyzed. The client confirms the best offer, or, if it is not satisfied with the offered QoS constraints, may set up a new QoS request with different constraints and start a new negotiation. If the client confirms an offer, the QoS manager confirms the temporary resource reservation made for the offer, signs the QoS contract and returns it to the client.

Clients and service providers employ a relatively low level of trust in the negotiation. A service provider only makes a temporary reservation that will expire if the client takes too long to make a decision. Likewise service providers will be dropped from the negotiation if they fail to provide an offer in time. Within the GEMSS project also more sophisticated negotiation strategies based on a closed-bid reverse English auction model have been realized, a description of which is beyond the scope of this paper.

3.5 Service Level Agreements

The GEMSS QoS infrastructure utilizes a subset of the WSLA specification for representing QoS requests, QoS offers and QoS contracts (see Figure 3). Being machine readable, WSLA documents allow processing the QoS negotiation without the need of human intervention until the final confirmation step. In the context of GEMSS, SLA parameters are QoS parameters and include the begin time of the job execution, the end time of the job execution, and the price of the job execution. The service definition section specifies the overall contract duration and a metric for each parameter. The obligations section contains a list of objectives. Each objective is linked to an obliged party and defines the acceptable values of a specific SLA parameter.

```
<SLA name="SpectSLA" xmlns="http://www.ibm.com/wsla">
 <Parties>
  <ServiceProvider name="ISC"/>
  <ServiceConsumer name="gerry"/>
 </Parties>
 <ServiceDefinition name="SPECTService">
  ...
  <Operation ... name="start" ...>
   <SLAParameter unit="GMT" type="time" name="beginTime">
   <SLAParameter unit="GMT" type="time" name="endTime">
   <SLAParameter unit="euro" type="double" name="price">
    ...
   <SOAPOperationName>start</SOAPOperationName>
  </Operation>
  ...
 </ServiceDefinition>
 <Obligations>
  <ServiceLevelObjective name="priceObjective">
   <Obliged>provider</Obliged>
   <Validity>
    <Start>2006-09-13T15:23:06.000+02:00</Start>
    <End>2006-09-13T18:10:06.000+02:00</End>
   </Validity>
   ...
```

```
   ...
   <Expression>
    <Predicate xsi:type="Equal" ... >
     <SLAParameter>price</SLAParameter>
     <Value>33.4</Value>
    </Predicate>
   </Expression>
   ...
  </ServiceLevelObjective>
  ...
  <ServiceLevelObjective name="endTimeObjective">
   ...
   <Expression>
    <Predicate xsi:type="LessEqual" ... >
     <SLAParameter>endTime</SLAParameter>
     <Value>211479006000</Value> <!-- ms since 01/01/2000 -->
    </Predicate>
   </Expression>
   ...
  </ServiceLevelObjective>
 </Obligations>
</SLA>
```

Fig. 3. Excerpt of WSLA for GEMSS SPECT service

4 Experimental Results

We present experimental results with a SPECT application for fully 3D iterative medical image reconstruction based on a Maximum Likelihood Expectation Maximization algorithm, which has been parallelized using MPI. For QoS support we used a performance model, parameterized with a set of request parameters (image resolution, number of slices, number of iterations, etc.) and the number of processors, with an accuracy of more than 95%.

In order to demonstrate the basic QoS negotiation we have devised a test with a single service provider running the SPECT service on a 16 CPU cluster with the Maui scheduler starting with a cluster utilization of zero. Clients issued consecutive offers for equal SPECT jobs, where the duration of the jobs just depended on the number of CPUs as shown in Figure 4. The QoS constraints for all jobs were defined within a 65 minutes timeslot with no price constraints. The graph in Figure 4 shows the number of CPUs used for each consecutive job,

CPUs	speedup	minutes
2	1.9	33:59
4	3.5	18:27
8	6.5	09:56
16	10.5	06:09

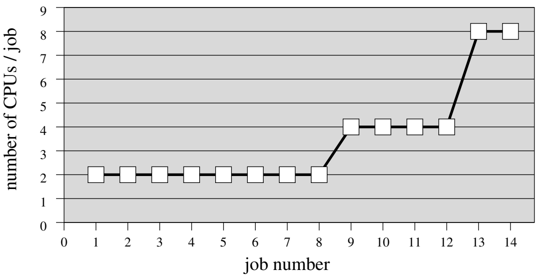


Fig. 4. Experimental results

where the initial 8 jobs were scheduled with 2 CPUs each, the next 4 jobs were scheduled on 4 CPUs and the last two jobs on 8 CPUs.

We can see from this test that the strategy of the QoS manager follows rational behavior. Initial requests are scheduled with fewer CPUs and if required the jobs are scheduled with more CPUs to met the given time constraints. After scheduling 14 jobs, the machine is almost fully utilized and no more offers will be generated for this specific timeframe.

5 Related Work

Standard Web Service technologies have now been adopted as the base middleware technology by many Grid computing environments including Globus GT4 [10], gLite [8], OMII [21] and Unicore [23]. The Open Grid Service Architecture (OGSA [7]) outlines the vision for a service level management and attainment model based on a generic control loop pattern, however, this vision has not yet been realized in available Grid environments. The OGSA specification mainly discusses macro QoS assurance (i.e. system-level QoS of the overall Grid infrastructure) while our work presented here focuses on micro QoS support for individual Grid application services.

The Web Services Agreement Specification (WS-Agreement) [2] of the GGF GRAAP working group proposes a language and protocol for service level agreements. A state-of-the art survey of advance reservation for grid applications can be found in [13]. In [1] an OGSA-based QoS model is presented which supports service discovery based on QoS attributes and Grid service execution subject to QoS constraints. The work in [16] outlines the challenges involved in mapping application-level SLAs to resource-level SLAs. In [18] a model for a QoS-aware component architecture for Grid computing is proposed.

Several Grid projects deal with Quality of Service aspects such as the GRASP project [11], which was concerned with extending the concept of Application Service Provision (ASP) to Grids, the GRACE project [5] which addressed deadline and budget based scheduling of Grid resources, and GRIA [22], which relies on workload and capacity estimation models to estimate the execution time of a submitted job using resource specific parameters [22].

There are an increasing number of Grid projects in the bio-medical and life science including the EU projects OpenMolGRID [20], MammoGrid [15], and myGrid [19], and the US BIRN initiative [4], to name a few. While most of these projects focus on data management aspects, GEMSS focused on the computational aspects of the Grid.

6 Conclusions

The GEMSS project developed a service-oriented Grid infrastructure for the on-demand provision of compute-intensive medical simulation services across wide area networks. The GEMSS QoS infrastructure presented in this paper relies on a reservation-based approach to QoS coupled with application-specific performance models, advance mechanisms, and client-driven negotiation of service level agreements. The GEMSS Grid provides guarantees to clients regarding quality of service within an economic model, and the legal and security framework needed to provide a platform for future exploitation. The Grid technology developed in the course of the GEMSS project is being utilized and further developed within the EU Project Aneurist [3], which aims to create an IT infrastructure for the management of all processes linked to research, diagnosis and treatment development for complex and multi-factorial diseases.

References

1. Al-Ali, R., Amin, K., von Laszeswski, G., Rana, O., Walker, D.: An OGSA Based Quality of Service Framework. In: Proceedings of the Second International Workshop on Grid and Cooperative Computing (GCC2003), Shanghai, China (2003)
2. Andrieux, A., et al.: Web Services Agreement Specification, Draft, Global Grid Forum, GRAAP Working Group (September 2005), `http://forge.gridforum.org/projects/graap-wg`
3. Aneurist EU IST Integrated Project IST-2004-027703, `http://www.aneurist.org/`
4. The Biomedical Informatics Research Network, `http://www.nbirn.net`
5. Buyya, R.: Economic-based Distributed Resource Management and Scheduling for Grid Computing, Ph.D Thesis, Monash University, Melbourne, Australia (2002)
6. Cao, J., Zimmermann, F.: Queue Scheduling and Advance Reservations with COSY. In: Proceedings of the International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico (2004)
7. Foster, I., et al.: The Open Grid Services Architecture, Version 1.5., Open Grid Forum, GFD-I.080 (July 2006), `http://forge.gridforum.org/projects/ogsa-wg`
8. Gagliardi, F., Jones, B., Laure, E.: The EU DataGrid Project: Building and Operating a large scale Grid Infrastructure. In: Di Martino, B., Dongarra, J., Hoisie, A., Yang, L.Y., Zima, H. (eds.) Engineering the Grid: Status and Perspective, American Scientific Publishers (January 2006)
9. The GEMSS Project: Grid-Enabled Medical Simulation Services. EU IST Project, IST-2001-37153, `http://www.gemss.de/`
10. The Globus Toolkit, `http://www.globus.org/toolkit/`
11. GRASP, The GRASP Project, `http://eu-grasp.net/`
12. Jones, D.M., Fenner, J.W., Berti, G., Kruggel, F., Mehrem, R.A., Backfrieder, W., Moore, R., Geltmeier, A.: The GEMSS Grid: An evolving HPC Environment for Medical Applications. In: HealthGrid 2004, Clermont-Ferrand, France (2004)
13. MacLaren, J.: Advance reservations: State of the Art. GGF GRAAP-WG (June 2003), `http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html`
14. Maui Cluster Scheduler, `http://www.clusterresources.com/products/maui/`
15. The MammoGrid project, `http://mammogrid.vitamib.com/`
16. Menasce, D.A., Casalicchio, E.: QoS in Grid Computing. EEE Internet Computing 8(4), 85–87 (2004)
17. Middleton, S., Herveg, J.A.M., Crazzolara, F., Marvin, D., Poullet, Y.: Privacy and security for a Medical Grid. Methods of Information in Medicine 44/2, 182–185 (2005)

18. Musunoori, S.B., Eliassen, F., Staehli, R.: QoS-aware component architecture support for grid. In: 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004, pp. 277–282 (2004)
19. The myGrid Project, `http://mygrid.man.ac.uk/`
20. Open Computing GRID for Molecular Science and Engineering, `http://www.openmolgrid.org/`
21. The Open Middleware Infrastructure Institute. OMII 2.0 User Guide, `http://www.omii.ac.uk/docs/2.3.3/omii_2_user_guide.htm`
22. Panagakis, A., Litke, A., Doulamis, A., Doulamis, N., Varvarigou, T., Varvarigos, E.: An Advanced Grid Architecture for a Commercial Grid Infrastructure. In: The 2nd European Across Grids Conference, Nicosia, Cyprus, Springer, Heidelberg (2004)
23. Unicore, `http://www.unicore.org`
24. Web Service Level Agreement (WSLA) Language Specification, IBM (2003), `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf`

A Transparent Grid Filesystem

Brian Coghlan, Geoff Quigley, Soha Maad, Gabriele Pierantoni, John Ryan, Eamonn Kenny, and David O'Callaghan

Trinity College Dublin,
Computer Science Department, Rm-005, CALab1, INS building, TCD,
Dublin 2, Ireland
coghlan@cs.tcd.ie

Abstract. Existing data management solutions fail to adequately support data management needs at the inter-grid (interoperability) level. We describe a possible solution, a transparent grid filesystem, and consider in detail a challenging use case.

1 Introduction

State of the art grid data management solutions on offer include: replica management (Globus RLS[1], LFC[2], Globus RC[3]); secure file transfer (Globus RFT[4], ELFI[5]); the GFARM special purpose middleware[6]; the SlashGrid credential-based grid filesystem[7]; Resource Namespace Service[8]; the GGF proposal of a service oriented architecture for a grid filesystem[9]; and middleware-specific grid filesystems such as in TeraGrid[10] and DEISA[11].

While meeting to some extent data management needs at the single middleware level, existing solutions fail to adequately support data management needs at the inter-grid level. This is due to the following reasons:

– Existing data management solutions are middleware specific;
– Middlewares have differing job life cycle data movement patterns; and
– There are no existing inter-grid information systems.

Some data management tools transfer data by value (for example Webcom [12,13], LCG2 sandboxes[14]) while others allow data to be pointed to by location independent identifiers, i.e. they transfer by reference (e.g. RC, RLS, LFC, RFT, ELFI, Slashgrid). The former is inefficient for large data sets. The latter overcomes this but complicates interoperability as the identifiers are generally middleware specific. Most tools are invoked via middleware-specific APIs (RC, RLS, LFC, RFT), but a few use native filesystem calls (ELFI, GFARM, Slashgrid). The latter definitely eases interoperability between different grids; data is transfered by reference and the transfers are invoked by standard file operations.

The paper is divided into two main sections. The first section briefly overviews existing grid data management solutions and discusses their limitations in meeting the needs of grid interoperability, then describes a possible solution, a grid filesystem. The second section considers in detail a challenging use case. The paper concludes with a summary.

2 The Grid Filesystem

The above considerations motivated the development of our grid filesystem, *gridFS*[15]. GridFS has basic but relatively complete functionality provided by four abstract engines for directory handling, discovery, data movement, and consistency, as shown in Figure 1. This grid filesystem can be deployed on every Linux node using EGEE, Globus, or any other middleware and even on user's workstations. The grid filesystem is specifically intended to support interoperability between arbitrary middlewares.

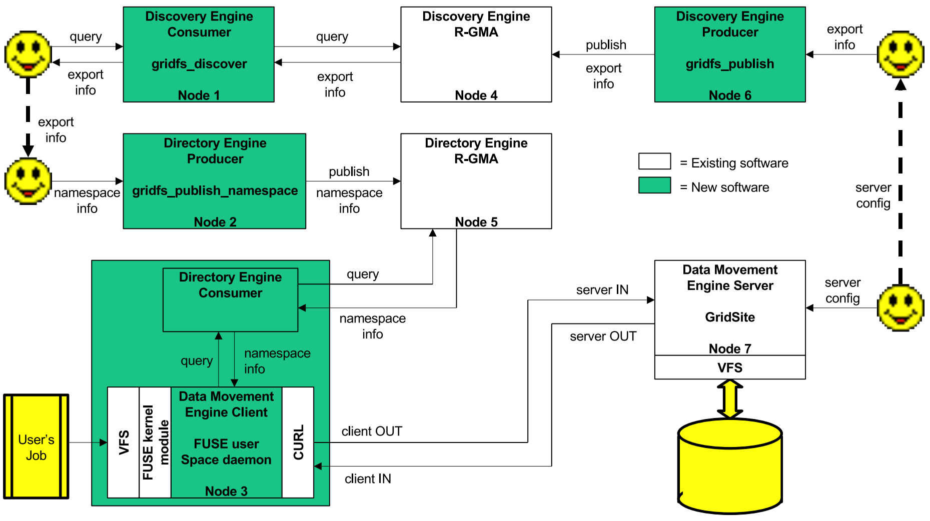


Fig. 1. The grid filesystem architecture

The data movement engine consists of the basic client- and server-sides of a filesystem. Its purpose is to allow I/O operations such as open, read, write, seek, close, etc. to be invoked using the normal libraries for C, Java, etc. but on remote files, i.e. it performs remote file access. In the initial version the consistency engine enforces consistency only via write-through and write-back coherency policies. The directory engine allows the creation and validation of a user view of the grid filesystem namespace. That is, a user can create their own logical view of grid data as a tree of filesystems. The discovery engine allows metadata to be published and queried as needed to support grid and inter-grid activity. The discovery engine assumes that every machine on the grid is able to export some directories according to given permissions. The filesystem has been designed according to two explicit policies; *reuse* (do not reinvent the wheel) and *reduce* (eliminate what is not necessary).

2.1 Data Movement Engine

The data movement engine is intended to overcome two major limitations in most existing data management solutions. Firstly, they do not operate at the block level. Secondly, in the main they require the programmer to use software-specific APIs to access, for example, file catalogues.

With the above concerns in mind, the data movement engine focusses on accessing various types of data storage via the Linux Virtual Filesystem layer (VFS)[16], and operation at block rather than at file level. The server side of the data movement engine exploits the GridSite [17] module (`mod_gridsite`) for the Apache webserver [18]. GridSite accesses files using the HTTP 1.1 protocol. It includes very desirable features including authentication and GACL [19] authorisation at each directory level, and supports convenient editing of these permissions. Each directory contains an XML `.gacl` file that defines the permissions, conditioned by host, VO or person.

GridSite supported byte-level access for get, but only file-level access for put, move and delete. For the data movement engine we modified `mod_gridsite` to provide byte-level access for both read and write. Gridsite now includes this.

The client side of the data movement engine uses block-level caching to minimise network traffic and support consistency. Like ELFI, GFARM and more recently Slashgrid, it derives from the FUSE user-space daemon. This approach has several notable attributes:

1. By using a user-space daemon invoked by the user, it can support a user-specific view of the grid filesystem.
2. By using HTTPS to communicate with the server-side, it can:
   - traverse firewalls as easily as any browser can;
   - authenticate connections;
   - leverage web protocol optimizations; and
   - leverage the proxy functionality of web servers.
3. By using a user-space daemon and HTTPS, it supports complete user-level privacy:
   - at the client even the root user cannot access the user's filespace, and all data movement beyond the client daemon is secured by SSL.
4. By operating at block level:
   - it can fractionally read or write remote files, only transferring the blocks required; and
   - it has the potential to provide block-level locking to facilitate future mutually-exclusive multiple-writers to the same file.

At a high level then, these components provide a filesystem that communicates over HTTPS with a web server and allows secure file operations on byte-ranges within files. Use of a user-space daemon addresses the transparency requirement, by providing access to remote files through the VFS layer using standard I/O libraries that are universally available.

2.2 Consistency Engine

The consistency engine maintains file consistency. Consistency semantics define the outcome of multiple accesses to a single file. An example case where inconsistency may arise is when several grid processes access the same filesystem and attempt to write simultaneously to a single file. To avoid problems several consistency models can be adopted. In the initial version the engine enforces cache consistency via a simple single-reader single-writer write-back coherency policy.

2.3 Directory Engine

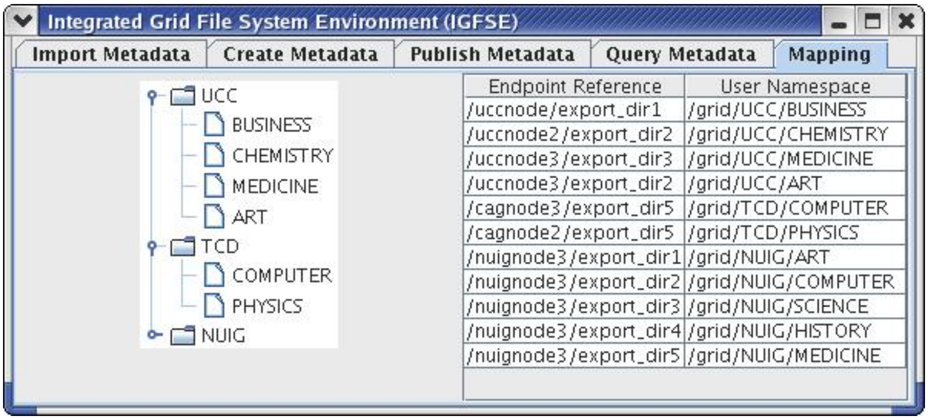
Although the data movement engine provides transparency, the directory engine is needed to fully address the issue. Only by allowing views that hide data location and complexity can a truly location-transparent and user-friendly system be constructed. A user creates a personal hierarchy of virtual directories, where leaves in this tree correspond to subdirectories of personal interest that exist on physical filesystems. It is assumed these subdirectories have been exported using the grid filesystem by the same or another user. The user runs their own instance of the FUSE daemon that accesses this personalised view of the grid filesystem namespace, using their own grid credentials. In essence the directory engine publishes and retrieves filesystem user–namespace mappings.

The directory engine implements many of the concepts of the Resource Namespace Service (RNS) provided by the Global Grid Forum's Grid File System working group. RNS identifies resources within a Grid by a universal name that ultimately resolves to a meaningful address, with a particular emphasis on hierarchically managed names that may be used in human interface applications. RNS embodies a three-tier naming architecture, which consists of human interface names, logical reference names, and endpoint references. Name-to-resource mapping in RNS features an optional arrangement of two levels of indirection. The first level of indirection is realized by mapping human interface names directly to endpoint references ($Name \rightarrow ER$).

A second level of indirection may be useful when mapping human interface names to logical references (identified by logical names), which in turn map to endpoint references ($Name \rightarrow LR \rightarrow ER$). The advantage of using a logical name to represent a logical reference is that they may be referenced and resolved independently of the namespace.

Figure 2 shows an example of the first level of indirection. The table on the right-hand side shows the list of mappings of the full physical locations to a user-specified namespace that is easily recognised and makes sense to the user.

The directory engine uses the Relational Grid Monitoring Architecture (R-GMA) [20,21] to publish namespace information. An R-GMA system is composed of three types of components: producers, which insert data into a virtual database; consumers, which retrieve information from the virtual database; and registries, which match consumers to producers which publish information that the consumers are interested in. Information can be inserted into and retrieved from the R-GMA virtual database using a subset of the SQL92 Entry Level



Endpoint Reference	User Namespace
/uccnode/export_dir1	/grid/UCC/BUSINESS
/uccnode2/export_dir2	/grid/UCC/CHEMISTRY
/uccnode3/export_dir3	/grid/UCC/MEDICINE
/uccnode3/export_dir2	/grid/UCC/ART
/cagnode3/export_dir5	/grid/TCD/COMPUTER
/cagnode2/export_dir5	/grid/TCD/PHYSICS
/nuignode3/export_dir1	/grid/NUIG/ART
/nuignode3/export_dir2	/grid/NUIG/COMPUTER
/nuignode3/export_dir3	/grid/NUIG/SCIENCE
/nuignode3/export_dir4	/grid/NUIG/HISTORY
/nuignode3/export_dir5	/grid/NUIG/MEDICINE

Fig. 2. GUI showing mapping of Endpoint References to User Namespaces. The Endpoint Reference information is expressed in the form */node/export_dir*, where *node* is a server and *export_dir* is a directory.

standard. Directory engine producers publish namespace information into the database using SQL INSERT statements, and their consumers retrieve that information from the database using SQL SELECT statements.

R-GMA currently offers rather limited security based on host authentication. Whereas the directory engine functions are subject to credential checks and permissions, data published in R-GMA is not yet subject to user-based authorisation and can be queried by any authenticated user regardless of permissions. This is expected to be remedied by the end of 2006.

2.4 Discovery Engine

Whereas existing grid middleware focusses on delivering partial data management solutions, there is very little existing metadata that may be used to describe a grid filesystem. The discovery engine allows metadata to be published and discovered or queried as needed to support grid and inter-grid activity, using two sets of tools:

1. *gridfspublish:* command-line and graphical tools to publish metadata to an inter-grid discovery service.
2. *gridfsdiscover:* command-line and graphical tools to discover filesystems that match a search string by querying published metadata.

The discovery engine assumes that any machine on the grid is able to export some directories according to given permissions (at grid and inter-grid levels) of *read*, *write*, *open*, *execute*, *admin*, etc. Like the directory engine, the discovery engine depends on the R-GMA to publish metadata. The same security constraints are imposed by the fact that currently R-GMA only offers host-based authentication.

Individual files may not be discovered, only filesystems (directories), as grid-wide discovery down to the file level would not scale with this architecture. However, an activity-specific (perhaps VO-wide) discovery engine may be established that publishes richer metadata, e.g. to file level.

3 A Grid Interoperability Use Case for the *gridFS*

This use case demonstrates the use of the grid filesystem engine in supporting interoperability between three middlewares: WebCom-G [22], Globus 4 (GT4) [23,24] and the Large Hadron Collider Computing Grid (LCG2)[14].

WebCom-G can be considered a fledgling grid-enabled workflow engine based on WebCom [25], although it is much more. It offers a non-von-Newmann programming model that automatically handles task synchronization (load balancing, fault tolerance, and task allocation at the system level) without burdening the application writer [22]. In WebCom, applications are specified as Condensed Graphs (CGs), in a manner which is independent of the execution architecture, thus separating the application and execution environments. Condensed Graphs are a mathematical abstraction capable of representing different computational models in a unified way. They consist of nodes with arcs connecting them. The nodes can be atomic, consisting of a single operation, or can be condensed graphs themselves.

Various patterns of data movement are involved during the life cycle of a WebCom job. Jobs are submitted through a web service, specifying the following:

– Condensed Graph representation in XML.
– Files used by the WebCom job, including executable files representing atomic operations.
– Arguments to the WebCom graph.

To demonstrate the importance of gridFS for grid interoperability we describe the data movements of the use case of Figure 3 with and without the gridFS. This use case describes a complex job involving three different Grid middlewares (WebCom, LCG2 and GT4). This job is described by the CG $= \{\mathrm{E}, \mathrm{W}_1, \mathrm{L}, \mathrm{G}, \mathrm{W}_2, \mathrm{X}\}$. It encompasses at least three compute nodes (one for $\mathrm{W}_1, \mathrm{W}_2$ and one each for L and G) and at least one filestore (for A and E and possibly for B, C and D too).

– E represents the entry node of the CG.
– W_1 represents an atomic operation that must be be executed on WebCom. It reads file A and writes file B.
– L represents an atomic operation that must be executed on the LCG2 middleware; this operation is described (in JDL language) in file J. It reads files J, P, A and B and writes file C.
– G represents an atomic operation that must be executed on GT4 middleware; this operation is described (in the RSL language) in file R. It reads files R, Q, A and B and writes file D.

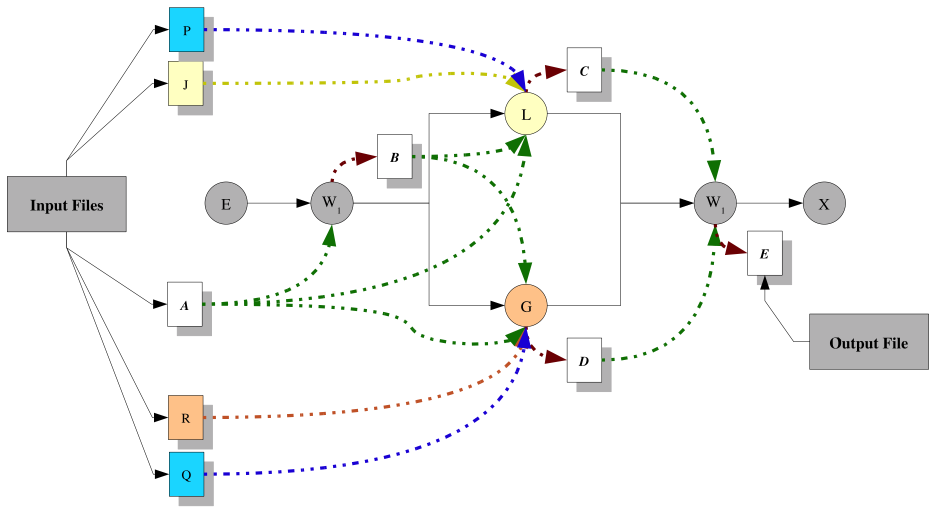


Fig. 3. Data Movement for the Use Case

- W_2 represents an atomic operation that must be executed on WebCom. It reads files C and D, and writes result file E.
- X represents the exit node of the CG.

In the absence of a grid filesystem such as gridFS the submission of this job implies a significant number of file transfers among the participating hosts. Typically, the following sequence of operations would happen:

- File A would be copied by value from a filestore to the WebCom middleware host (using a WebCom mechanism) for the execution of W_1.
- Operation W_1 would be executed. It would read a local copy of file A and write a local copy of file B.
- File B would be copied to a filestore, using a WebCom mechanism.
- Files J, P, A and B would be copied by value to the LCG2 middleware for the execution of job L, either by prestaging, use of sandboxes or file catalogs, or by explicit file transfer; all of these methods are intrusive and some will require job L to be modified.
- Job L would be executed. It would read local copies of files J, P, A, B and write a local copy of file C.
- File C would be copied to a filestore, using an LCG2 mechanism.
- Files R, Q, A and B would be copied by value to the GT4 middleware for the execution of job G, either by prestaging, use of file catalogs, or by explicit file transfers; again these methods are intrusive and job G may need to be modified.
- Job G would be executed. It would read local copies of files R, Q, A, B and write a local copy of file D.
- File D would be copied to a filestore, using a GT4 mechanism

– Files C and D must be copied by value to the WebCom middleware for the execution of operation W_2, using a WebCom mechanism.
– Operation W_2 would be executed. It would read local copies of files C and D, and write a local copy of file E.
– Result file E would be copied to the filestore using a WebCom mechanism, where it can be accessed by the user.

As can be seen, there are multiple transfer mechanisms. It is assumed that full (not partial) file transfers occur before and after executions. The operations or applications to be executed can be modified to use special grid file I/O libraries but there are multiple libraries, each middleware-specific and none transparent. Thus, in the absence of a grid filesystem, such as gridFS, the submission of this job involves a significant amount of data transfer that intrudes on the development of the workflow. The above is a simplified scenario as multiple transfers are often necessary to pass files between the different grids and a filestore residing in a 'border' region (grid $\rightleftharpoons$ border $\rightleftharpoons$ filestore). In addition, unless WebCom executes all its executable graph nodes on the same physical machine (an unintended restriction) any intermediate files will have to be expensively copied by value between the nodes. A similar workflow executed using Condor DagMan [26] would use the Condor sandbox mechanism to move files to where they were required, again with significant data transfer.

The presence of a grid filesystem allows access to all files as if they were present on a local file system, greatly simplifying the data movement pattern. In all cases, input files can be very efficiently copied by reference between the grid middlewares, i.e. the filenames can be passed to and fro as text arguments. There is a single transfer mechanism, applications need not be modified; grid file I/O libraries are not used, only standard I/O libraries that are middleware-independent and transparent. For example, the LCG2 job L can be very simply specified using the JDL:

```
Executable="script_L.sh";
Arguments="gridfs/mydir/B gridfs/mydir/C";
StdOutput="std.out";
StdError="std.err";
InputSandbox={"script_L.sh"};
OutputSandbox={"std.out","std.err"};
```

The script `script_L.sh` needs to start up gridFS, which then fetches the user's namespace mappings from the directory engine and thereby automatically points to the remote directory. The script can then run the user's LCG2 application code P with the arguments specified in the JDL. Any data transfers during execution take place as normal file reads and writes (but to a remote filestore using gridFS) and full file transfers to local disk do not take place, only partial transfers to an in-memory cache. When the job completes the script needs to terminate gridFS by shutting down the user's FUSE daemon; this final step also ensures the cache has been fully flushed. Thus `script_L.sh` is as follows:

```
gridfs
bash gridfs/mydir/P $1 $2
fusermount -u gridfs
```

Alternatively, the grid filesystem can either be continuously executing as a system daemon, or automatically started in the users context on all relevant machines. The latter is preferable as it guarantees per-user privacy, even from system administrators. Notice that there is an opportunity for even greater transparency then: if gridFS is automatically started and terminated like this, then the outer two commands above are not needed. Grid-Ireland plans to install gridFS in this configuration. In summary, large numbers of explicit file transfer operations (some multistage) are necessary when executing jobs across multiple grids if a grid filesystem is not available. When a grid filesystem is used, these file transfers are replaced by fewer direct remote file accesses by reference.

4 Conclusions

We have described a transparent grid filesystem that provides access to remote files. It does not require proprietary APIs, instead using standard file I/O libraries that are universally available. Hence applications do not need to be altered to allow remote file access. The transport protocols (https) are those normally allowed through firewalls. The gridFS supports authentication using certificates. The authorisation mechanism supports higher level constructs such as virtual organisations. The gridFS supports partial file access and relatively efficient caching. It allows per-user views of the file namespaces. It supports publication and subsequent discovery of filesystems. It can be deployed on any Linux workstation or server and is currently deployed on three sites within the Grid-Ireland infrastructure. It is specifically intended to support interoperability between arbitrary middlewares and we have described one challenging use case.

References

1. Chervenak, A.L., Palavalli, N., Bharathi, S., Kesselman, C., Schwartzkopf, R.: Performance and scalability of a replica location service. In: Proceedings of the International IEEE Symposium on High Performance Distributed Computing (HPDC-13), IEEE Computer Society Press, Los Alamitos (2004)
2. Baud, J.P., Casey, J., Lemaitre, S., Nicholson, C.: Performance analysis of a file catalog for the LHC Computing Grid. In: Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, pp. 91–99. IEEE Computer Society Press, Los Alamitos (2005)
3. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., Tuecke, S.: The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets (based on conference publication from proceedings of netstore conference 1999). Journal of Network and Computer Applications 23, 187–200 (2001)
4. Globus Alliance: GT 4.0 Reliable File Transfer (RFT) service (2006), `http://www.globus.org/toolkit/docs/4.0/data/rft/`

5. EGRID: ELFI file system (2006), http://www.egrid.it/sw/elfi/index_html
6. Tatebe, O., Soda, N., Morita, Y., Matsuoka, S., Sekiguchi, S.: Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In: Proceedings of the 2004 Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland (2004)
7. McNab, A.: SlashGrid – a framework for grid-aware filesystems. In: Storage Element Workshop, CERN (2002)
8. Pereira, M., Tatebe, O., Luan, L., Anderson, T., Xu, J.: Resource namespace service specification (2005)
9. Global Grid Forum: The GGF Grid File System architecture workbook, version 0.54 (2005)
10. Reed, D.: Grids, the TeraGrid and beyond. Computer 36(1), 62–68 (2003)
11. DEISA: Distributed European Infrastructure for Supercomputing Applications (2004), http://www.deisa.org
12. Morrison, J.P., Kennedy, J.J., Power, D.A.: WebCom: A web based volunteer computer. The Journal of Supercomputing 18, 47–61 (2001)
13. Morrison, J.P., Power, D.A., Kennedy, J.J.: An evolution of the WebCom metacomputer. J. Math. Model. Algorithms 2, 263–276 (2003)
14. CERN: The Large Hadron Collider (LHC) Computing Grid Project for high energy physics data analysis (2003), http://lcg.web.cern.ch/LCG/
15. Maad, S., Coghlan, B., Quigley, G., Ryan, J., Kenny, E., Callaghan, D.O.: Towards a complete grid filesystem functionality. Future Generation Computer Systems, Special Issue on Data Analysis, Access and Management on Grids (2006)
16. Gooch, R.: Overview of the virtual file system. Linux Documentation project (1999), http://www.atnf.csiro.au/~rgooch/linux/docs/vfs.txt
17. McNab, A.: The GridSite web/grid security system. Software: Practice and Experience 35(9), 827–834 (2005)
18. Fielding, R.T., Kaiser, G.: The Apache HTTP server project. IEEE Internet Computing 1(4), 88–90 (1997)
19. McNab, A.: Grid-based access control and user management for Unix environments, filesystems, web sites and virtual organisations. In: Proceedings of Computing in High Energy Physics, La Jolla, CA (2003)
20. Fisher, S.: Relational model for information and monitoring (2001)
21. Coghlan, B., Djaoui, A., Fisher, S., Magowan, J., Oevers, M.: Time, information services and the grid. In: Read, B. (ed.) Advances in Databases. LNCS, vol. 2097, Springer, Heidelberg (2001)
22. Morrison, J.P., Coghlan, B., Shearer, A., Foley, S., Power, D., Perrot, R.: WebCom-G: A candidate middleware for Grid-Ireland. High Performance Computing Applications and Parallel Processing (2005)
23. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. International Journal of Supercomputer Applications 11(2), 115–128 (1997)
24. Foster, I.: Globus toolkit version 4: Software for service-oriented systems. In: Jin, H., Reed, D., Jiang, W. (eds.) NPC 2005. LNCS, vol. 3779, pp. 2–13. Springer, Heidelberg (2005)
25. Morrison, J., Power, D., Kennedy, J.: Load balancing and fault tolerance in a condensed graphs based metacomputer. The Journal of Internet Technologies, Special Issue on Web based Programming 3(4), 221–234 (2002)
26. Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. In: Grid Computing: Making The Global Infrastructure a Reality (2003)

Grid Data Integration Based on Schema Mapping

Carmela Comito and Domenico Talia

DEIS, University of Calabria,
Via P. Bucci 41 c,
87036 Rende, Italy
{ccomito, talia}@deis.unical.it
http://www.deis.unical.it/

Abstract. Data integration is the flexible and managed federation, analysis, and processing of data from different distributed sources. Data integration is a key issue for exploiting the availability of large, heterogeneous, distributed and highly dynamic data volumes on Grids. This paper presents a framework for integrating heterogeneous XML data sources distributed among the nodes of a Grid. We present a query reformulation algorithm to combine and query XML documents through a decentralized point-to-point mediation process among the different data sources based on schema mappings. The above cited XML integration formalism is exposed as a Grid Service within an OGSA-based Grid architecture.

1 Introduction

The goal of a data integration system is to combine heterogeneous data residing at different sites by providing a unified view of this data. Data integration on Grids has to deal with unpredictable, highly dynamic data volumes. So, traditional approaches to data integration, such as FDBMS [1] and the use of mediator/wrapper middleware [2], are not suitable in Grid settings. The federation approach is a rather rigid configuration where resources allocation is static and optimization cannot take advantage of evolving circumstances in the execution environment. The design of mediator/wrapper integration systems must be done globally and the coordination of mediators has to be done centrally, which is an obstacle to the exploitation of evolving characteristics of dynamic environments. As a consequence, data sources cannot change often and significantly, otherwise they may violate the mappings to the mediated schema. Recently, several works on data management in peer-to-peer (P2P) systems are moving along this direction [3, 4]. All these systems focus on an integration approach not based on a global schema: each peer represents an autonomous information system, and data integration is achieved by establishing mappings among the various peers.

The Grid community is devoting great attention toward the management of structured and semi-structured data such as databases and XML data. The most significant examples of such efforts are the *OGSA Data Access and Integration* (OGSA-DAI) [5] and the *OGSA Distributed Query Processor* (OGSA-DQP) [6] projects. However, till today only few of those projects [7,8] actually meet schema-integration issues necessary for establishing semantic connections among heterogeneous data sources. For these reasons, we designed the XMAP framework [9] for integrating heterogeneous XML data sources distributed over a Grid. By designing this framework, we aim at developing a decentralized network of semantically related schemas that enables the formulation of distributed queries over heterogeneous data sources. We designed a method to combine and query XML documents through a decentralized point-to-point mediation process among the different data sources based on schema mappings. We offer a decentralized service-based architecture that exposes this XML integration formalism as a Grid Service [10]. We refer to this architecture as the *Grid Data Integration System* (GDIS) [11]. The GDIS infrastructure exploits the middleware provided by OGSA-DAI building on top of it schema-integration services. As said before, among the few works designed to provide schema-integration in Grids, the most notable ones are *Hyper* [7] and *GDMS* [8]. Both systems are based on the same approach that we have used ourselves: building data integration services by extending the reference implementation of OGSA-DAI. The *Grid Data Mediation Service* (GDMS) uses a wrapper/mediator approach based on a global schema. GDMS presents heterogeneous, distributed data sources as one logical virtual data source in the form of an OGSA-DAI service. This work is essentially different from ours as it uses a global schema. For its part, *Hyper* is a framework that integrates relational data in P2P systems built on Grid infrastructures. As in other P2P integration systems, the integration is achieved without using any hierarchical structure for establishing mappings among the autonomous peers. In that framework, the authors use a simple relational language for expressing both the schemas and the mappings. By comparison, our integration model follows as Hyper an approach not based on a hierarchical structure, however differently from Hyper it focuses on XML data sources and is based on schema mappings that associate paths in different schemas.

The rest of the paper is organized as follows. Section 2 describes the XMAP framework. Section 3 shows as the XMAP algorithm is deployed as a Grid service within the GDIS architecture. Some performance figures are presented in Section 4 and Section 5 gives some concluding remarks, together with possible extensions of this work.

2 XMAP: A Decentralized XML Data Integration Framework

The XMAP framework [9], semantically relates XML schemas enabling the formulation of queries over heterogeneous, distributed XML data sources. The environment is modeled as a system composed of a number of Grid nodes, where each node can hold one or more XML databases. These nodes are connected to each other through declarative mappings rules.

2.1 Integration Model

As mentioned before, traditional centralized architecture of data integration systems is not suitable for highly dynamic and distributed environments such as the Grid. Thus, we propose an approach inspired from [4] where the mapping rules are established directly among source schemas without relying on a central mediator or a hierarchy of mediators. In consequence, in our integration model, there is no global schema representing all data sources in a unique data model but a collection of local schemas (the native schema of each data source). Regardless of the total number of nodes composing the system, each source schema is directly connected to only a small number of other schemas. However, it remains reachable from all other schemas that belong to its “transitive closure”. For any mapping M, its closure is defined as the set of rules that can be derived from M by repeated composition of schema paths. In other words, the system supports two different kinds of mapping to connect schemas semantically: *point-to-point* mappings and *transitive* mappings. In transitive mappings, data sources are related through one or more “mediator schemas”. For example, if we have a source A directly connected to a source B and B connected to C, A is connected to *both* B and C. Establishing the mappings this way creates a graph of semantically related sources where each of the sources knows its direct semantic neighbors (point-to-point mapping) and can learn about the mappings of its neighbors (transitive mapping). Therefore, in our integration model all nodes are equal: there is no distinction between data sources and mediators. Each node acts both as a data source contributing data and as a local mediator providing an uniform view over the data provided by other nodes.

Our integration model is based on schema mappings to translate queries between different schemas. The goal of a schema mapping is to capture structural as well as terminological correspondences between schemas. We address this goal by associating paths in different schemas. Mappings are specified as path expressions that relate a specific element or attribute (together with its path) in the source schema to related elements or attributes in the destination schema. The data integration model we propose is indeed based on path-to-path mappings expressed in the XPath [12] query language, assuming XML Schema as the data model for XML sources. Specifically, this means that a path in a source is described in terms of XPath expressions.

The mapping rules are specified in XML documents called XMAP documents (see Figures 2, 3). Each source schema in the framework is associated to an XMAP document containing all the mapping rules related to it.

2.2 The XMAP Reformulation Algorithm: A Case Study

Our query processing approach exploits the semantic connections established in the system by performing the *XPath query reformulation algorithm* before executing the input query, in order to gain further knowledge. This way, when a query is posed over the schema of a source, the system will be able to use data from any source that is transitively connected by semantic mappings. Indeed, it will reformulate the given query expanding and translating it into appropriate queries for each semantically related source. Thus, the user can retrieve data from all the related sources in the system by simply submitting a single XPath query.

The rationale of the algorithm is to perform single reformulation steps. A reformulation step corresponds to the reformulation of a given query only with respect to the schemas directly connected to it. So, the algorithm is composed of several reformulation steps, but each of such steps performs only direct reformulations by using the point-to-point mappings. Each time a reformulated query is obtained, the algorithm tries to rewrite it by recursively invoking the reformulation function and using its direct mappings.

The query reformulation algorithm uses as input an XPath query and the mappings, and it produces as output zero, one or more reformulated queries.

In the following we briefly describe an example of use of the XMAP algorithm (see Figure 1).

Let suppose a user wants to find the *title* of the paper published in the *year* 2000. To this aim the following query Q_{UW} is formulated over the schema UW:
Q_{UW}=*/uw/area/pubs/paper*[*year* = "2000"]*/title*

In the first step the algorithm identifies the paths in the query:

– P_1=/uw/area/pubs/paper/title
– P_2=/uw/area/pubs/paper/year

and produces as output the set $\mathcal{P}$. Next, exploiting the XMAP document associated to the schema UW (see Figure 2), the algorithm finds two mapping rules connecting UW to $DBLP$ through the paths P_1 and P_2.

More precisely, one of these rules relates P1 to two paths in $DBLP$, respectively `/dblp/article/title` and `/dblp/proceedings/title`. Similarly, the

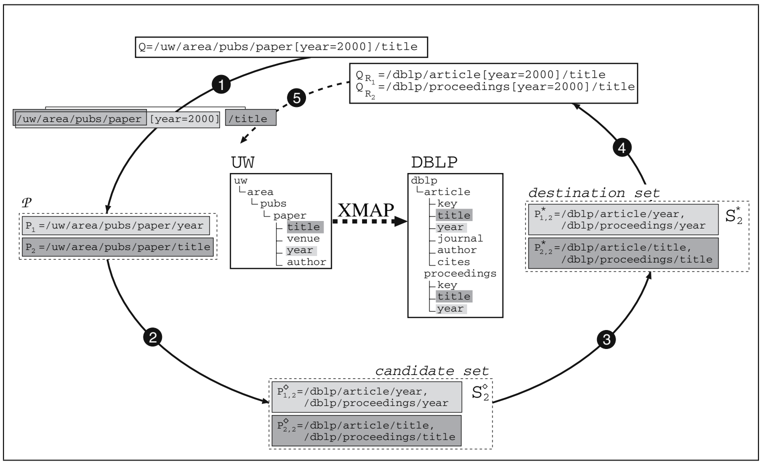

Q=/uw/area/pubs/paper[year=2000]/title
1
5
QR1 =/dblp/article[year=2000]/title
QR2 =/dblp/proceedings[year=2000]/title
/uw/area/pubs/paper [year=2000] /title
4
UW
uw area pubs paper title venue year author
XMAP
DBLP
dblp article key title year journal author cites proceedings key title year
𝒫
P1 =/uw/area/pubs/paper/year
P2 =/uw/area/pubs/paper/title
destination set
P*1,2=/dblp/article/year, /dblp/proceedings/year
P*2,2=/dblp/article/title, /dblp/proceedings/title
S*2
2
3
candidate set
P◇1,2=/dblp/article/year, /dblp/proceedings/year
P◇2,2=/dblp/article/title, /dblp/proceedings/title
S◇2

Fig. 1. Example of use of the XMAP algorithm

```
<sourceSchema>uw</sourceSchema>
<Rule cardinality="Mapping1-N">
 <destinationSchema>dblp</destinationSchema>
 <sourcePath>/uw/area/pubs/paper/title</sourcePath>
 <destinationPath>/dblp/article/title</destinationPath>
 <destinationPath>/dblp/proceedings/title</destinationPath>
</Rule>
<Rule cardinality="Mapping1-N">
 <destinationSchema>dblp</destinationSchema>
 <sourcePath>/uw/area/pubs/paper/year</sourcePath>
 <destinationPath>/dblp/article/year</destinationPath>
 <destinationPath>/dblp/proceedings/year</destinationPath>
</Rule>
```

Fig. 2. Fragment of the *UW* XMAP document

other mapping rule relates P_2 to the path `/dblp/article/year` and the path `/dblp/proceedings/year`. So, the second step of the algorithm produces as output a candidate set composed of the elements $P^{\diamond}_{i,j}$ and the (candidate) schema $DBLP^{\diamond}$. In the considered example as the schema $DBLP^{\diamond}$ has correspondences for both paths P_1 and P_2, it is identified as a destination schema (step 3), so it can be used to reformulate the query UW. In particular, the algorithm (step 4), produces two direct reformulations of the query Q_{UW} over the schema $DBLP$, respectively Q_{DBLP_1} and Q_{DBLP_2}.

Q_{DBLP_1}=/dblp/article[year="2000"]/title
Q_{DBLP_2}=/dblp/proceedings[year="2000"]/title

Since in the XMAP document associated to the schema UW there are no more mapping rules involving the paths in the query Q_{UW}, no further reformulations are produced. Then the algorithm is recursively invoked over the direct reformulations Q_{DBLP_1} and Q_{DBLP_2}, exploiting the XMAP document associated to the schema $DBLP$ (see Figure 3). In so doing, the algorithm finds two mapping rules connecting the schema $DBLP$ to the schema $IEEE$ through the paths composing the query Q_{DBLP_2}, whereas any mapping rules involving the paths of the query Q_{DBLP_1} have been found. As a consequence, the execution of the XMAP reformulation algorithm over the query Q_{DBLP_1} does not produce any reformulated query, instead the reformulation of the query Q_{DBLP_2} produces the reformulation Q_{IEEE}:

Q_{IEEE}=/ieee/proceedings[year="2000"]/title

This first recursive invocation of the algorithm ends here producing the query Q_{IEEE} as a direct reformulation of the query Q_{DBLP_2} over the schema $IEEE$ (that is a transitive reformulation of the original query Q_{UW}). So, now we can exploit the transitive mappings of the schema $DBLP$, by recursively invoking the reformulation algorithm over the query Q_{IEEE}. As the XMAP document

```
<Rule cardinality="Mapping1-1">
  <destinationSchema>ieee</destinationSchema>
  <sourcePath>/dblp/proceedings/title</sourcePath>
  <destinationPath>/ieee/proceedings/title</destinationPath>
 </Rule>
 <Rule cardinality="Mapping1-1">
  <destinationSchema>ieee</destinationSchema>
  <sourcePath>/dblp/proceedings/year</sourcePath>
  <destinationPath>/ieee/proceedings/year</destinationPath>
 </Rule>
```

Fig. 3. Fragment of the *DBLP* XMAP document

associated to the schema *IEEE* has no mappings for the paths composing the query Q_{IEEE}, the algorithm ends here.

3 The Grid Data Integration System (GDIS)

The XMAP reformulation algorithm has been deployed in the *Grid Data Integration System* (GDIS). GDIS is a decentralized service-based data integration architecture for Grid databases; it has been presented in a previous work [11].

The GDIS system offers a wrapper/mediator-based approach to integrate data sources: it adopts the XMAP decentralized mediator approach to handle semantic heterogeneity over data sources, whereas syntactic heterogeneity is hidden behind ad-hoc wrappers. More precisely, in the GDIS architecture (see [11]), the XMAP query reformulation engine is run by the *data integration nodes*. Specifically, these nodes offer: (i) a set of data integration utilities allowing to establish mappings, and (ii) the query reformulation algorithm introduced by the XMAP integration formalism. The user query is handled by the reformulator engine that through the XMAP query reformulation algorithm produces zero, one or more reformulations of the original query. All the obtained reformulations (included the original query) are then processed by the distributed query processor (DQP) that partitions each of such queries in several sub-queries. Then, the each produced sub-query execution plan is processed independently by the DQP that through the wrapper access data source and produces the partial query result. After that, the DQP collects the sub-query results into the final query result and return it to the querying node.

GDIS is designed as a service-based distributed architecture and introduces the OGSA-based *XML Data Integration* (*OGSA-XDI*) service that extends OGSA-DAI with additional functionality both to enable users to specify semantic mappings (in the form of XMAP documents) among a set of data sources, and to execute the XMAP query rewriting algorithm. The architecture of OGSA-DAI and the core engine are designed for new activities in order to provide new functionalities. Indeed, in order to implement the OGSA-XDI service we decided to extend the free available OGSA-DAI Grid Data Service (GDS) reference implementation. According to this, we introduced a new activity, the

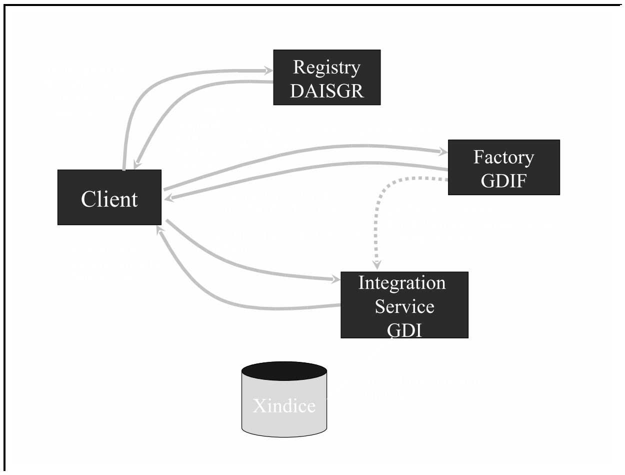


Fig. 4. OGSA-XDI component services

`XPathQueryReformulation` activity, that wraps the XPath query reformulation algorithm of the XMAP framework. In so doing, our implementation is regardless of the specific version of OGSA-DAI, and, consequently of the adopted standards (e.g., WSRF, WS-I, OGSI). The OGSA-XDI architecture is composed, analogously to OGSA-DAI, of a number of new services including (see Figure 4):

– Grid Data Integration service (GDI). This service provides XML schema mapping utilities for semantically connected XML data sources. To this aim the GDI extends the OGSA-DAI by introducing a new activity devoted to the reformulation of an XPath query by using the XMAP reformulation algorithm.
– Grid Data Integration Factory (GDIF) which creates a GDI on request. A GDIF extends a Grid Data Service Factory (GDSF) by introducing new metadata in the form of XMAP doucments concerning the mapping rules of the schema of the wrapped database.
– Database Access and Integration Service Group Registry (DAISGR) which allows clients to search for GDIFs and GDIs that meet their requirements. This registry service is the one provided by the OGSA-DAI system, as no modifications are necessary to manage the introduced integration services.

4 GDIS Evaluation

In this section we briefly present preliminary evaluations of the GDIS prototype. To experiment with GDIS, we have setup two GDIS systems, one located on LAN and the other one over the Internet. In these experiments, we focus on the performance of GDIS as a whole, not on the specificity of the reformulation process which has been detailed in a previous work. The results presented in this section were produced according to the following protocol: the client uses the factory service (GDIF) to create a GDI instance. It then uses this instance to submit the same request 100 times (it waits for the results of the previous request before sending the next one), and finally destroys the instance it has created.

Reformulating queries that are submitted to it is the real task of our GDI. Interactions taking place in this process are represented in Figure 5. First, the client requests to the GDIF the creation of the GDI instance. After some time spent to prepare the perform document, the client submits it to the instance it has just created. During the reformulation process, the GDI asks the registry for any mapping information it might need. It is visible on the figure that mapping information queries are performed lazily, only when it is first needed, which explains why we don't see a bunch of requests to the registry and then some quiet time to perform reformulation. Instead, reformulation CPU time is spread in between mapping information requests: as soon as a new reformulated query is found, the GDI asks the registry for the XMAP document of the schema over which the query is expressed (assuming it has not seen it before). We shall also notice that XMAP documents are all stored on the registry and that the GDI therefore does not interact with the other GDIFs to obtain mapping information.

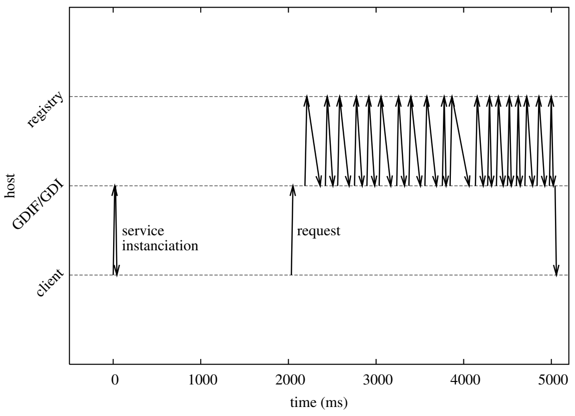
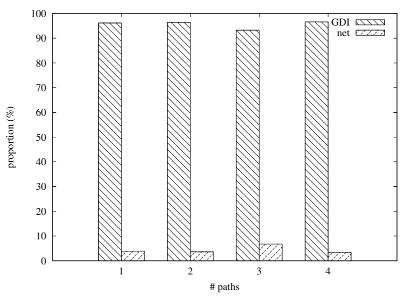


Fig. 5. Entity interactions during the reformulation of a specific query in the WAN configuration. Interactions are also in this case HTTP requests and responses.

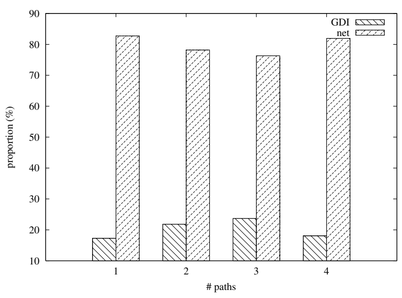
Understanding the dynamics of the query reformulation is of high importance to let us understand the results of the experimentations we have performed.

We evaluated the query reformulation time in both the LAN and WAN scenarios. As XMAP evaluation is out of the scope of these tests, we will focus on the cutoff of the time spent in the system.

LAN Case. We start considering the case when all entities involved in the system are located on the same LAN.



(a) LAN Setting.



(b) WAN Setting.

Fig. 6. Relative repartition of the time spent for query reformulation with mappings of an average rank equals to 2

The results (relative values) of the experiments for a mapping set with an average rank of 2 are shown in Figure 6(a). The GDI time here is really the time used by XMAP to perform the reformulation, while the "net" time is the communication time spent on the network asking and waiting for XMAP documents. The low-latency and high-bandwidth network connecting the different machines makes this time really low. At the maximum, it represents 6.7% of the total time spent for the reformulation. Reformulation times, for their part, are consistent with the results obtained in the XMAP evaluation, given the discrepancy of the hardware between the different series of test. Network times are almost negligible compared to those required to perform query reformulation. The results obtained confirm the intuition that network time only depends on the number and the size of the XMAP documents required to perform the full reformulation.

WAN Case. Results for the rank 2 mapping set are shown in Figure 6(b). They make it obvious that even for a moderate round-trip time (42 ms on average) between the GDIF and the registry, network cost becomes largely predominant. From our results, it appears that the total network time is roughly equivalent to two round-trip times between the GDI and the registry, plus the time necessary to transmit the document at the speed of the wire, multiplied by the number of XMAP documents to import. With our settings, this was always inferior in value to 3 round-trip times. This gives us useful hints to reduce the overall execution time of the reformulation.

5 Conclusions

The continuously growing availability of data sources on Grids requires a coordinated and integrated access of such data due to the heterogeneity of the involved data models. Based on these reasons we proposed the XMAP framework and the GDIS architecture combining a data integration approach with existing grid services for querying heterogeneous, distributed databases. This way we provide an enhanced, data integration-enabled service middleware supporting distributed query processing. As for future work, we plan to extend the XMAP framework in order to support more expressive query languages (e.g. XQuery). Further, declarative query approaches will be used to express the distributed and dynamic properties of the evolving network (it could change during query execution), following recent ideas on declarative networking.

Acknowledgements

This work has been partially supported by the FP6 Network of Excellence CoreGRID (Contract IST-2002-004265).

References

1. Sheth, A.P., Larson, J.A.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys 22(3), 183–236 (1990)
2. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: VLDB, pp. 251–262 (1996)
3. Calvanese, D., Damaggio, E., Giacomo, G.D., Lenzerini, M., Rosati, R.: Semantic data integration in P2P systems. In: DBISP2P, pp. 77–90 (2003)
4. Halevy, A.Y., Suciu, D., Tatarinov, I., Ives, Z.G.: Schema mediation in peer data management systems. In: ICDE, pp. 505–516 (2003)
5. Antonioletti, M., et al.: OGSA-DAI: Two years on. In: Global Grid Forum 10 — Data Area Workshop (2004), `http://www.ogsadai.org.uk/`
6. Alpdemir, M.N., Mukherjee, A., Gounaris, A., Paton, N.W., Watson, P., Fernandes, A.A.A., Fitzgerald, D.J.: OGSA-DQP: A service for distributed querying on the grid. In: EDBT, pp. 858–861 (2004)
7. Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R., Vetere, G.: Hyper: A framework for peer-to-peer data integration on grids. In: Bouzeghoub, M., Goble, C.A., Kashyap, V., Spaccapietra, S. (eds.) ICSNW 2004. LNCS, vol. 3226, pp. 144–157. Springer, Heidelberg (2004)
8. Brezany, P., Woehrer, A., Tjoa, A.M.: Novel mediator architectures for grid information systems. FGCS - Grid Computing: Theory, Methods and Applications 21(1), 107–114 (2005)
9. Comito, C., Talia, D.: XML data integration in OGSA grids. In: Pierson, J.-M. (ed.) Data Management in Grids. LNCS, vol. 3836, pp. 4–15. Springer, Heidelberg (2006)
10. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: The physiology of the grid. In: Global Grid Forum (2002), `http://www.globus.org/alliance/publications/papers/ogsa.pdf`
11. Comito, C., Talia, D.: GDIS: A service-based architecture for data integration on grids. In: GADA, pp. 88–98 (2004)
12. Clark, J., DeRose, S.: XML path language (XPath) version 1.0. W3C Recommendation (1999), `http://www.w3.org/TR/xpath`

Simulations in Geophysics and Space Physics: Minisymposium Abstract

Mats Holmström[1] and Kjell Rönnmark[2]

[1] Swedish Institute of Space Physics, Kiruna, Sweden
[2] Department of Physics, Umeå University, Sweden

High Performance Computing (HPC) enables increasingly complex computer models of physical processes to be built. Parallel computing, adaptive grids, and other computing tools are used to solve ever larger problems, but they also increase the complexity of the software significantly and can have an impact on the reliability of the software and the results of the computations.

These issues make the software and algorithms used an important part of the science produced by simulations in geophysics and space physics.

However, traditional conferences and journals in geophysics and space physics do not lend themselves for presentations and discussions of details of the models. The aim of this minisymposium is to provide such a forum.

Parallelization of a Public Image Restoration Algorithm*

Francisco Almeida[1], Carlos Delgado[2], Ramon García-López[3], and Francisco de Sande[1]

[1] Depto. de E.I.O. y Computación, Universidad de La Laguna, ES-38271–La Laguna, Spain
fsande@ull.es
[2] Instituto de Astrofísica de Canarias (IAC), ES-38205–La Laguna, Spain
[3] IAC and Astrophysics Dept., Universidad de la Laguna, La Laguna, Spain

Abstract. We present a deconvolution method intended for image restoration. Although the method is computationally heavy, it obtains a good restoration quality. We have successfully achieved the scientific aim of parallelizing the algorithm, obtaining acceptable scalability on shared and distributed memory machines.

1 Introduction

Any imaging instrument used to acquire data has a finite resolving power, or resolution. For example, the image of a point source seen through a telescope has an angular size related to the diameter of the optical elements. Most of the scientific information can be extracted from these resolution limited data, so further effort to improve the resolution using a post-processing mechanism is not necessary. However, for some studies it is required to obtain data as independent of the instrument properties as possible. Currently much effort is being devoted to developing numerical methods for this purpose, generally called deconvolution algorithms. In this paper we aim to show our implementation of a method, primarily intended for processing astronomical images, that has wider application.

2 The Problem

Let us assume that we are trying to determine the probabilistic distribution of the arrival position of some particles to our detector. The detector has a discrete set of windows or pixels, numbered from 1 to N. When a particle enters one of these pixels, the instrument records the pixel number. Hereafter we refer to this process as a measurement. After measuring a large number of particles, M, we are left with a sequence of numbers $(r_1 r_2 \ldots r_M)$, being r_k the pixel number

* This work has been partially supported by the EC (FEDER) and the Spanish MEC (Plan Nacional de I+D+I, TIN2005-09037-C02-01).

recorded after k measurements. If we are not interested in the ordering, the set of measurements can be summarized through the number of times pixel i appears in the sequence, denoted by n_i.

$$n_i \equiv freq(i, (r_1 r_2 \ldots r_M)) \tag{1}$$

If the detector is not perfect there is a non-null probability of measuring a wrong pixel. We can model this effect by means of the conditional probability of measuring j if the true entrance pixel was i, denoted as R_i^j, that we assume to be known. In that case we have a second unknown sequence of the true entrance pixels, $(t_1 t_2 \ldots t_M)$, associated to the measured ones. Similarly to eq. 1, we have

$$n_i^{true} \equiv freq(i, (t_1 t_2 \ldots t_M)), \tag{2}$$

and the sets of $\{n_i\}_{i=1}^N$ and $\{n_i^{true}\}_{i=1}^N$ are related according to

$$n_i = \sum_{j=1}^{N} R_j^i n_j^{true}, \tag{3}$$

We assume that R_j^i is a matrix with non-null determinant. In the special case when it depends only of some distance measure between pixel i and j, then eq. 3 is a convolution. Generally the measured value of the right hand side of eq. 3 is not exactly equal to the left side due to statistical fluctuations. With the adopted detector model, the distribution of these values follows a Poisson distribution [1].

The aim of the inversion algorithms is to compute an estimate of the set $\{n_i^{true}\}_{i=1}^N$ given the values $\{n_i\}_{i=1}^N$. This is known to be an ill-posed problem, therefore there is no unique and stable solution, and the algorithms have to choose a solution according to some criteria.

A good review of the plethora of algorithms devised to recover the "best" estimate in the case of astronomical images can be found in [2]. These methods can be roughly classified as linear regularization methods, and Bayesian framework methods. The Wiener filter and the Tikhonov regularization belong to the first class, which is described in detail in [3]. These methods are very fast, but the use of a priori information is difficult and they create Gibbs oscillations in the neighborhood of discontinuities. Methods in the second class rely on a model for the probability of obtaining the measured image given a candidate to the solution, or likelihood, and an a priori probability for any candidate solution. Then the maximum a posteriori solution of the product of the likelihood and the a priori probability is chosen as the "best" solution. The difference between the methods in this class is how the a priori probability is chosen, ranging from entropy based priors [4] to methods that make use of the spatial coherence of the solution like [5] and [6]. Moreover, in the case of choosing a uniform prior, an iterative maximization can be constructed and the solution is chosen by taking a finite number of steps in this iteration. The resulting method in the case of the detector model presented above is the Richardson-Lucy (RL) algorithm [7,8], which is nowadays widely used in the astronomy community. This method has been subsequently improved to include a priori information on the objects in the image and to add smoothness constraints [9] in order to avoid noise amplification in the solution, although at the cost of introducing free parameters. During the last years there has been a trend towards the use of multiresolution methods, especially wavelet transforms, together with these inversion algorithms, which cures some of their problems and fixes some of their free parameters [10]. Apart from these algorithms, there are others that do not fit within the above classes like the CLEAN [11] algorithm and its refinements, or the Markov Chain Monte Carlo (MCMC) ones, like Pixon in its first versions [12]. The MCMC methods present the peculiarity of increasing the number of degrees of freedom of the inversion problem, thus alleviating its ill-poseness and reducing the dependence on free parameters. Moreover they allow a better error estimation. The method we propose belongs to this class, although a further approximation allows us a deterministic computation.

3 Algorithm

In this section we first motivate our approach from a Bayesian point of view, and then we describe our algorithm.

3.1 A Bayesian Approach to the Problem

Let us introduce the compressed notation $\mathbf{S}_M(x) \equiv (x_1x_2 \ldots x_M)$ for a sequence. Given the sequence of measurements $\mathbf{S}_M(r) \equiv (r_1r_2 \ldots r_M)$, the problem stated in the previous section is equivalent to find a sequence $\mathbf{S}_M(t) \equiv (t_1t_2 \ldots t_M)$ such that for any position in it, k, the true pixel traversed by the particle which produced the measurement r_k, was t_k. Given the probability of a realization of such solution conditional to the measured sequence, $P_{sequ}(\mathbf{S}_M(t)|\mathbf{S}_M(r))$, we adopt as the estimate of $\{n^{true}\}_{i=1}^{N}$ the following average

$$n_i^{true*} = \sum_{(t'_1t'_2 \ldots t'_M)} freq(i, (t'_1t'_2 \ldots t'_M)) \times P_{sequ}(t'_1t'_2 \ldots t'_M|r_1r_2 \ldots r_M) \ , \tag{4}$$

where the asterisk denotes estimate, and the summation is for all sequences $(t'_1t'_2 \ldots t'_M)$. The problem is then to compute $P_{sequ}(\mathbf{S}_M(t)|\mathbf{S}_M(r))$. For this purpose we make use of the Bayes theorem to write

$$\begin{aligned} P_{sequ}(\mathbf{S}_M(t)|\mathbf{S}_M(r)) = \mathcal{K} P_R(r_M|t_M) \times P_{prior}(t_M|\mathbf{S}_{M-1}(t)) \\ \times P_{sequ}(\mathbf{S}_{M-1}(t)|\mathbf{S}_{M-1}(r)) \end{aligned} \tag{5}$$

where $\mathcal{K}$ is a normalization constant, the probability $P_R(r_M|t_M)$ coincides with $R_{t_M}^{r_M}$ defined in section 2, and $P_{prior}(t_M|\mathbf{S}_{M-1}(t))$ is the probability of next entrance pixel being t_M given the sequence of all previous entrance pixels. Equation 5 defines an iterative process which allows to compute the average of eq. 4 if $P_{prior}(t_M|\mathbf{S}_{M-1}(t))$ is given. Taking into account that in our detector model the sequence of true entrance pixels is generated by an stationary process, the number of times a given pixel number appears in the sequence $\mathbf{S}_{M-1}(t)$ follows a binomial distribution. It is well known that a good estimate of the probability of that distribution for pixel number t_M is given by [13]

$$P_{prior}(t_M|t_1 \dots t_{M-1}) = \frac{freq(t_M, (t_1 \dots t_{M-1})) + \epsilon}{M - 1 + N\epsilon} \ , \tag{6}$$

N being the number of pixels, and ϵ a small parameter which depends on the prior on the binomial distribution parameter[1]. In order to avoid a large dependency of the prior with the number of pixels for small M, we choose $\epsilon = 1/N$.

The iterative expression obtained inserting eq. 6 in eq. 5 is well suited to compute the average in eq. 4 using the Monte Carlo method . An example of the application of this method with synthetic data is given by Figure 1a, where we use a total of 100 sampled sequences to compute the average in eq. 4. The synthetic measured values of $\{n^i\}_{i=1}^N$, shown as a dotted histogram in the figure, have been obtained by sampling from the the distribution resulting from applying eq. 3, where $\{n_i^{true}\}_{i=1}^N$ is given by the continuous histogram labelled *True distribution*, applying subsequently the definition given by eq. 1. For comparison figure 1b shows the deconvolution of the same reconstruction obtained with the RL [7,8] method, with the regularization condition tuned to obtain the best fit to the true solution. The errors in this case are computed according to [14]. Clearly the MCMC method recovers the true distribution within the uncertainties due to the small size of the Monte Carlo sample. It also shows a good agreement with the RL solution, except for the quoted errors. In RL they are clearly underestimated, since for any pixel i they are very close or even smaller than $\sqrt{n_i^{true}}$, which is the lower bound assuming no correlation between pixels.

3.2 A Method Inspired by the Bayesian Approach for Large Samples

The method based on the Monte-Carlo computation of average in eq. 4, scales linearly with the size of the measured sequences, M. Since for many real applications, for example astronomical images, this number can be well above 10^6, sampling a single sequence can be very time consuming and even unaffordable in practice. We aim to obtain a deeper understanding of why the proposed method works in order to apply a faster approximation for these real cases. To this end we focus our discussion on the case in which the pixels can be arranged in a two-dimensional matrix, such that some distance measure, $d(i,j)$, between pixels is defined. Let us assume that the value of R_j^i for fixed j increases smoothly with decreasing $d(i,j)$, and that the absolute value of $R_{j_1}^i - R_{j_2}^i$ approaches zero if $d(j_1, j_2)$ does. Under these conditions, we justify below that the values of $freq(i, \mathbf{S}_M(t))$ for a single sequence $\mathbf{S}_M(t)$ sampled according to eq. 5, have large values for a small set of pixels, and small values for the rest. Given the sequence

[1] Here we assume that the size of all pixels is the same. If this is not the case, a similar argument can be built up upon the sizes of the windows.

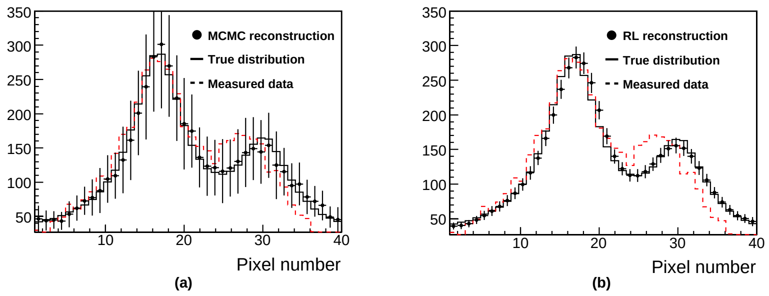


Fig. 1. Restoration of the synthetic data using the proposed Monte Carlo method (a) and the RL method (b). In the Y axis n_i^{true} is represented as a continuous histogram, n_i as a dotted histogram, and the circles represent the estimate n_i^{true*}, i being the pixel number of a total of $N = 40$. The error bars are the estimate fluctuation due to Poisson fluctuations of the dotted histogram.

of measurements, $(r_1 r_2 \dots r_M)$, we can proceed to sample one realization of the solution, $(t_1 t_2 \dots t_M)$, making use of eq. 5. According to this equation, the pixel number t_1 is sampled from the distribution

$$P(t_1) = \frac{R_{t_1}^{r_1}}{\sum_{j=1}^{N} R_{r_1}^{j}}, \tag{7}$$

where N is the number of pixels. Now for t_2, the distribution is

$$P(t_2) = \mathcal{K}' R_{t_2}^{r_2} \times \begin{cases} 1 & \text{if } t_2 = t_1 \\ N^{-1} & \text{if } t_2 \neq t_1 \end{cases} \tag{8}$$

where $\mathcal{K}'$ is a normalization constant. If the distance $d(r_2, t_1)$ is small enough to make $R_{t_1}^{r_2} >> N^{-1}$, then t_2 will with very high probability be the same pixel as t_1 if $N >> 1$.

If this iteration is continued in the same manner, it is clearly seen that the pixels which appear at the beginning of the sequence $(t_1 t_2 \dots t_M)$ are the ones that appear more frequently in the whole sequence, whereas pixels close in distance to these will rarely appear. Therefore if we represent the values of $freq(i, (t_1 t_2 \dots t_M))$ in the 2D pixels arrangement we expect to see a distribution of spikes, whereas for another sampled sequence $(t'_1 t'_2 \dots t'_M)$ the position of the spikes will be different. Taking into account this observation, we assume that the position of the spikes for a single sequence can be approximated by the nodes of a rectangular grid embedded in the 2D pixel arrangement, with a fixed distance between adjacent nodes, s given by an integer number of pixels, as illustrated in figure 2. Let us consider that the distance s, that we call step size, is fixed and one of the possible embedded grids is chosen, We can then try to estimate the sum in eq. 4 with the condition that all the pixel numbers in the sequence $(t'_1 t'_2 \dots t'_M)$ of that equation coincide with a pixel in coincidence with one of these nodes. If M is large, we expect the summation to be dominated by the most probable realizations of the sequence $\mathbf{S}_M(t')$. Since the prior given by eq. 6 is non-informative in absence of measurements, this sequence has to be very close to the ones which minimize the following quantity

$$\mathcal{L} = \sum_i \frac{(freq(i, r_1 r_2 \dots r_M) - n_i^{est})^2}{freq(i, r_1 r_2 \dots r_M)} = \sum_i \frac{(n_i - n_i^{est})^2}{n_i} \tag{9}$$

where the summation is for all the pixels where the denominator is non zero, $(r_1 \dots r_M)$ is the sequence of measurements, and n_i^{est} is given by

$$n_i^{est} = \sum_j R_j^i freq(j, (t_1' t_2' \dots t_M')) \tag{10}$$

where the summation in j is for all the pixels with a coincident position with a node of the grid.

Since there are only $n_{grid}(s) \simeq N/s^2$ possible embedded grids with step size s, this approximation allows the computation of eq. 4 by performing $n_{grid}(s)$ minimizations of eq. 9, one for each grid, without the need of sampling.

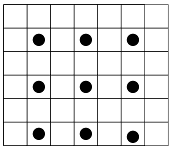


Fig. 2. Illustration of the arrangement of the pixels in a 2D matrix (squares) and the embedded grid of spikes (circles), as explained in the text. In this case the distance between adjacent nodes is two pixels.

Finally we are left with the problem of finding the optimal value of s. We do so by noticing that in case the grid approximation is a good one, then eq. 9 should be distributed like a χ^2 distribution with $N - n_{grid}(s)$ degrees of freedom, N being the number of pixels. Since in this case the values of $\mathcal{L}$ at the minimum peak around $N - n_{grid}(s)$, we look for the s such that the following quantity is minimum

$$V(s) = \sum_{i=1}^{n_{grid}(s)} (\mathcal{L}^{min(i)} - N + n_{grid}(s))^2 \ , \tag{11}$$

where the summation index runs over all the embedded grids of step size s and $\mathcal{L}^{min(i)}$ is the value of eq. 9 evaluated at the minimum for the grid number i.

The algorithm resulting from this discussion can be summarized as follows: Take s to be two pixels and V_{best} a large number. For all the embedded grids with step size s find the set of n_i^{est} which minimizes eq. 9 using any reasonable minimization algorithm. Compute $V(s)$. If $V(s) < V_{best}$ compute the estimated solution as

$$n_i^{true*} = \frac{(n_i^{est})_{node}}{n_{grid}(s)} \ , \tag{12}$$

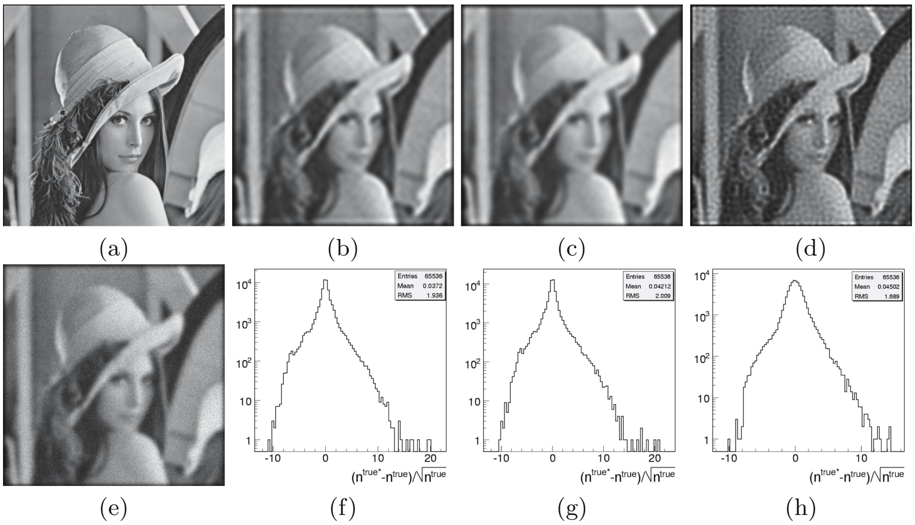


Fig. 3. (a) Original image used to show the performance of the algorithm. (e) Histogram resulting of drawing 10^7 samples distributed according to the convolution of the original image with the Gaussian distribution as explained in the text. (b) Reconstructed image using our algorithm, (c) the *RL-1* solution, and (d) the *RL-2* for the case of 10^7 samples. Captions (f), (g) and (h) show the histogrammed residuals, defined as $(n^{true*} - n^{true})/\sqrt{n^{true}}$ for the reconstructions of captions (b), (c) and (d) respectively.

where $(n_i^{est})_{node}$ is the value at the minimum for the grid with a node in pixel i, increase s by one, make $V_{best} = V(s)$ and go to the second step. Otherwise return the previously estimated solution and finish.

To show the performance of this algorithm, we have performed some tests with the Lena image in figure 3(a). This image is usually used in general image reconstruction test because it presents a rich variety of features like sharp edges and homogeneous regions. For the conditional probability R_j^i we have used an isotropic Gaussian distribution with $\sigma = 2.5$ pixels around the pixel j. To simulate the measurement process, we have convolved the original with this distribution, and we have drawn 10^6, 10^7 and 10^8 samples distributed according to the resulting image. The obtained measured image for the case of 10^7 samples is shown in figure 3(e).

We have restored the images using our method, the RL method with the regularization condition suggested in [8], that we call *RL-1*. We have also searched the regularization condition such that the RL method yields the solution which best fits the original, undistorted image, that we call *RL-2*. The obtained reconstructions for the case of 10^7 samples are shown in captions (b), (c) and (d) of figure 3.

In order to quantitatively compare the different deconvolutions, we use the metric

Table 1. Quantitative comparative table of the estimates obtained with the different methods as explained in the text

Samples	Method	$2\mathcal{L} \times 10^{-4}$	$D \times 10^3$
10^6	*RL-1*	11.2	5.7
	Our method	8.4	2.8
	RL-2	7.7	1.2
10^7	*RL-1*	10.0	2.8
	Our method	7.1	2.4
	RL-2	4.5	0.9
10^8	*RL-1*	11.4	0.2
	Our method	6.9	0.1
	RL-2	4.4	0.1

$$D = \frac{1}{M} \max_p | \sum_{j=1}^{p} n_j^{true} - n_j^{true*} | , \tag{13}$$

where M is the number of samples, the index j runs from pixel 1 to pixel p, and $max_a f(a,x)$ denotes the maximum value of the function $f(a,x)$ with respect to the variable a. This distance is closely related to the Kolmogorov test [1], and thus is a sensitive measure of agreement between the reconstructed and the original image. Indeed, a low value of D means that the reconstructed image globally resembles better the original image. Notice also that D is related to the residual. We also compute the value of $\mathcal{L}$, summing all the pixels in the image, as a metric of the distance of the reconstructed image convolved with the kernel to the measured one.

The obtained values of the metrics for each method are shown in Table 1. The *RL-2* estimate is the best reconstruction, as it is inferred from both metrics. Obviously this is because we have artificially forced it to be the best fit to the true solution. However, as it is clearly seen in the corresponding row of figure 3, the result is not satisfactory at small spatial scales, especially for low number of samples, since the appearance of artifacts greatly distorts the result. On the contrary, the *RL-1* estimate approaches the correct solution as the number of samples grows very slowly, resulting in worse values for both metrics and in a final image too smooth. In comparison, *Our method's* results stay between these two. The corresponding metric values approach the best ones as the number of samples grow, introducing less artifacts for small number of samples.

4 Parallelization and Computational Results

Our algorithm admits several parallelizations with different grain. In this work, we have focused our attention on the minimization step, which requires to compute eq. 3. This is done by decomposing the summation of partial sums with the same number of terms. Each partial sum is performed by a different processor, and a subsequent combination allows to perform the required operation.

We have developed our parallelization using both OpenMP [15], and MPI [16], the current standards for shared and distributed memory architectures. While the domain decomposition is straightforward in OpenMP, just introduce the appropriate pragmas into the code, the MPI implementation requires some minor changes in the sequential code, and involves a collective operation to combine the partial sums.

Three different target platforms were used in the experiments. An IBM RS-6000 SP with 8*16 Nighthawk Power3 @375Mhz (192 Gflops/s) with 64 Gb RAM. The nodes are connected through an SP Switch2 operating at 500MB/sec. The second platform is a 16 dual node cluster with Intel Pentium Xeon processors running at 2.8 GHz, with 2 GByte of RAM memory each, and connected through a Gigabit switch. The operating system of the cluster was Linux. The last platform is a Bull NovaScale 6320 server with 64 Intel-Itanium 1.5GHz processors running Linux Kernel 2.6.7 The nodes are connected through a Quadrics QsNetII network.

The IBM RS-6000 is a shared memory architecture while both clusters have distributed memory. The compiler we used in the IBM RS-6000 was the OpenMP native compiler, and the MPICH implementation of MPI was used in the clusters. In the case of the IBM RS-6000 and the Bull NovaScale server, we used only one 16 processors node.

Table 2 summarizes the results for all implementations of the code on different platforms. We observe that the sequential code takes about 2 hours on the PC cluster while it takes 3.5 hours on the Bull cluster. This is a consequence of the larger computational power of the processors in the PC cluster. Nevertheless, the speedup is much higher on the Novascale server due to the higher performance of the Quadrics network. With 16 processors, the performance of the Bull architecture overcomes that of the PC cluster, taking advantage of the network in the implementation of the collective operation: 17 minutes on the PC cluster vs. 15 on the Novascale.

The last four columns in Table 2 show the results on the shared memory architecture using OpenMP for PSF sizes 30×30 and 20×20 (here we call PSF

Table 2. Execution time (secs.) and speedup for the implementations. For the IBM RS-6000 implementation only speedups are shown.

	PC Cluster		BULL Novascale		IBM 20×20	IBM 30×30
#Proc.	Time	Speedup	Time	Speedup	Speedup	Speedup
SEQ	7499	1	12574	1	1	1
2	3801	1.97	6335	1.98	1.95	1.92
4	1973	3.8	3275	3.84	3.66	3.64
8	1518	4.94	1684	7.47	6.77	7.02
16	1034	7.25	885	14.21	12.19	13.52
32	600	12.5	-	-	-	.

to the R^i_j matrix). For the IBM RS-6000 we observe an almost linear increase of the speedup with the number of processors and we see that the results improve with larger PSF sizes.

5 Conclusions

In this work we have introduced a deconvolution method. Although the method is computationally heavier than classical methods, all regularization parameters are fixed and the quality of the restoration is good. We have successfully achieved the scientific aim of parallelizing our algorithm, obtaining acceptable scalability on shared and distributed memory architectures.

Work in progress in our project includes the development of a computational web service that will ease the access to this technology as a free service to the scientific community.

References

1. Eadie, W.T., Drijar, D., James, F., Roos, M., Sadoulet, B.: Statistical Methods in Experimental Physics. North-Holland (1971)
2. Starck, J.L., Pantin, E., Murtagh, F.: Pub. Astro. Soc. Pac. 114, 1051–1069 (2002)
3. Bertero, M., Boccacci, P.: Introduction to Inverse Problems in Imaging. Inst. Phys., London (1998)
4. Gull, S., Skilling, J.: MEMSYS5 Quantified Maximum Entropy User's Manual. Maximum Entropy Data Consultants, Suffolk (1991)
5. Molina, R., Nunez, J., Cortijo, F.: IEEE Signal Process. Magazine 18, 11 (2001)
6. Delgado, C.: Astron. and Astrophy. 454, 385–392 (2006)
7. Richardson, W.H.: J. Opt. Soc. Am. 62, 55 (1972)
8. Lucy, R.B.: Astron. J. 79, 745 (1974)
9. Pirzkal, N., Hook, R., Lucy, L.: Astronomical Data Analysis Software and Systems IX. In: ASP Conf. Ser. 216. ASP, San Francisco (2001)
10. Starck, J.L., Murtagh, F.: Astron. and Astrophy. 288, 342–348 (1994)
11. Högbom, J.A.: Astron. and Astrophys. Supp. 15, 417 (1974)
12. Dixon, D.D., Johnson, W.N., Kurfess, J.D., Pina, R.K., Puetter, R.C., Purcell, W.R., Tuemer, T.O., Wheaton, W.A., Zych, A.D.: Astron. and Astrophys. Supp. 120, C 683+ (1996)
13. Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B.: Bayesian Data Analysis. CRC Press, Boca Raton (2003)
14. D'Agostini, G.: Nucl. Instrum. Meth. A 362, 487–498 (1995)
15. OpenMP Architecture Review Board: OpenMP Application Program Interface 2.5 (2005)
16. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. University of Tennessee, Knoxville, TN (1995), http://www.mpi-forum.org/

Visualizing Katrina - Merging Computer Simulations with Observations

Werner Benger, Shalini Venkataraman, Amanda Long, Gabrielle Allen, Stephen David Beck, Maciej Brodowicz, Jon MacLaren, and Edward Seidel

Center for Computation and Technology, Louisiana State University, Baton Rouge, LA 70803, USA
{werner, shalini, along, gallen, sdbeck, maciek, maclaren, eseidel}@cct.lsu.edu
http://sciviz.cct.lsu.edu/katrina/

Abstract. Hurricane Katrina has had a devastating impact on the US Gulf Coast, and her effects will be felt for many years. Forecasts of such events, coupled with timely response, can greatly reduce casualties and save billions of dollars. We show how visualizations from storm surge and atmospheric simulations were used to understand the predictions of how strong, where, and when flooding would occur in the hours leading up to Katrina's landfall. Sophisticated surface, flow and volume visualization techniques show these simulation results interleaved with actual observations, including satellite cloud images, GIS aerial maps and LIDAR showing the 3D terrain of New Orleans. The sheer size and complexity of the data in this application also motivated research in efficient data access mechanisms and rendering algorithms. Our goals were to use the resulting animation as a vehicle for raising awareness in the general populace to the true impact of the event, to create a scientifically accurate representation of the storm and its effects, and to develop a workflow to create similar visualizations for future and simulated hurricanes. Screenings of the animation have been well received, both by the general public and by scientists in the field.

1 Motivation

The catastrophe of Hurricane Katrina has not only highlighted the need for timely and accurate measurements from instruments and forecasts from numerical simulations but also for meaningful visualizations that draw upon these diverse data sources. In this paper, we highlight one such effort to visualize the events leading to the flooding cased by Hurricane Katrina pulling together models of the hurricane's wind, temperature and pressure fields, the storm surge, 3D terrain views from LIDAR and GIS data, combined with comparisons to what actually happened using time-varying atmospheric imagery from the GOES-12 weather satellite and the actual hurricane tracks. The Center for Computation & Technology at the Louisiana State University (LSU) is a partner in the SURA Coastal Ocean observing and Prediction Program (SCOOP) [1], a interdisciplinary community engaging in distributed coastal modeling across the southeastern US with the goal of building an integrated virtual laboratory for coastal research. Advisories from the National Hurricane Center(NHC) about impending storms automatically trigger automated workflows that use different wind fields to initiate coastal models such as the ADvanced CIRCulation hydrodynamic model[1] (ADCIRC) that require significant parallel computing resources, available at CCT through the 1024 processor cluster *SuperMike*. The wind-fields are generated by the MM5[2] atmospheric model. The SCOOP data archive [2] developed and deployed at LSU aggregates the model outputs from multiple sources across the nation and is the source of data for our visualization efforts.

2 Previous Work

The geoscience community has concentrated on visualizing, generally in 2D, data from remote sensing and GIS mapping sources. In contrast, within the atmospheric sciences, much work is confined to scientific visualization methods such as isosurfaces or volume rendering for 3D atmospheric model outputs. Nativi et al. [3] clearly highlighted the differences between the data models in the GIS and atmospheric sciences. While GIS is concerned with 2D georeferenced spatial data in multiple layers, atmospheric science deals with hyperspatial (3D, 4D and beyond) where geo-referencing is not critical. Moreover, the temporal scales for GIS is orders of magnitude more than in the atmospheric sciences (years vs. minutes).

Examples of atmospheric visualization include work from National Oceanic and Atmospheric Administration (NOAA) Geophysical Fluid Dynamic Laboratory (GFDL)[3], and more recently that of Hurricane Isabel motivated from the IEEE Visualization 2004 contest. The above efforts deal with effectively rendering [4] or data mining and feature extraction [5] of time-varying multdimensional scalar and vector fields and so, do not incorporate other data like GIS or storm-surge models. Recent work in storm-surge and GIS visualization from Zhang et al[6] focuses on geo-information processing such as extracting buildings from LIDAR and interactive animation of flooding. The NOAA Satellite and Information service also have some height-field visualizations of weather satellite data including GOES-12[4]. What makes our contribution unique is the integrated visualization of all the above diverse data sources.

3 Data Management

3.1 Data Sources

ADCIRC accurately models a wind-driven storm surge - its formation, movement across the ocean and morphology as it impacts land. The underlying computational mesh is built upon topographic or bathymetric information given on an

[1] `http://www.nd.edu/~adcirc/`
[2] `http://www.mmm.ucar.edu/mm5/`
[3] `http://www.gfdl.noaa.gov/research/weather/hurricane.html`
[4] `http://www.nnvl.noaa.gov/`

adaptively refined unstructured grid ranging from the atlantic ocean into the canals of New Orleans, where the physical resolution approaches just 100m. The numerical simulation outputs water elevation, given as a scalar quantity on each surface vertex, plus wind and water flow directions, provided as a 2D vector field, on each vertex as well. In our dataset, this output responds to a physical time interval of every 30 mins during Aug 15^{th} until Sep 1^{st}, 2005, just after Katrina had made landfall.

To show the atmospheric conditions that lead to the hurricane formation and the resulting surge, we make use of wind, pressure and temperature fields from the MM5 atmospheric model simulation covering the same time period. Only the domain-2 from this hierarchical data set was used. Here, each time-step is a structured 3D grid with dimensions 150x140x48 storing the wind velocity, pressure and temperature values, output for every hour of simulation domain.

The satellite imagery is based on observational data captured every 15 min from the GOES-12 satellite. GOES[5] is an acronym for Geostationary Operational Environmental Satellite, they are the American counterparts to the European METEOSAT weather satellites. Here, we focus on the "longwave" infrared channel, $10.7\mu m$ measured at 4km resolution.

The dates for all data sets were available on the same temporal domain. In the MM5 case, the available atmospheric simulation results covered the time span of up to 72 hours starting at midnight each day. Since this yielded multiple datasets corresponding to the same time coordinates, we decided to restrict the analysis to data representing the first 24 hour period of each simulation time frame. While this might result in a slight discontinuity of animations when crossing the date boundary, it also ensures that the input datasets contain only the simulation results based on the most recently acquired meteorological measurements.

A 5m resolution elevation grid of the New Orleans area is provided by a LIDAR[6] data set, obtained from the State of Louisiana. In addition, we used satellite imagery of the terrain from the MODIS and LANDSAT instruments at 500m and 250m resolution respectively.

3.2 Data Management Challenges

Satellite images and GIS data are well representable in common image file formats such as GEOtiff [7]. In contrast, MM5 and ADCIRC data are more complex, and no standard format exists for these kind of data types. A huge number of file formats compete, each with particular features for each application. Mostly file formats are optimized for a certain data type, and consequently become mutually exclusive. For instance, a file format being able to cover MM5 data can not necessarily handle ADCIRC data as well and vice versa. Even for each specific, allegedly simple class of data, such as a triangular surface, there co-exist myriads of file formats. Supporting each of these is a tedious work wasting time

[5] `http://www.oso.noaa.gov/goes/`

[6] LIght Detection And Ranging `http://www.lidarmapping.com`

[7] `http://www.remotesensing.org/geotiff/geotiff.html`

of application developers. In a scenario where n various independent applications with no a-priori mutual knowledge need to interact, supporting each other application's file format becomes a major n^2 implementation effort.

Ideally, we would like to use a common file format which covers all cases of types of scientific data and thus achieves maximum synergy effects. To find such a unified description, a common denominator is essential, which, following D. Butler [7,8], is naturally provided by the language of mathematics for the domain of scientific data. D. Butler proposed to use the mathematical concept of vector and fiber bundles to layout data, a concept which is successfully implemented in the IBM DataExplorer, now available as OpenDX [8]. Within the classification scheme of the fiber bundle data model, MM5 outputs are dynamic scalar and vector data on three-dimensional regular domain, while ADCIRC data are described by a a dynamic scalar field given on a static triangular surface.

3.3 The "F5" Approach

We do not necessarily need to introduce a new file format from scratch. The Hierarchical Data Format V.5 [9] is a widely used I/O library developed at NCSA with a corresponding file format, known as HDF5. The HDF5 API provides many unique features, which are particularly valuable in the context of Grid computing [9]. However, while HDF5 provides a syntax for the efficient representation of scientific data, there still remains ambiguity in how to formulate a certain type of scientific data. The layout in the concept of a fiber bundle provides a direction toward narrowing down such ambiguities, and at the same time defining generic operations. Such a layout is still not unique per se; our version [10] has been shown to be able of covering a wide range of data types. Among other features, it intrinsically supports the notion of time and handles scalar, vector, tensor and other multivalued fields of arbitrary dimensions on regular and irregular mesh triangulation schemes.

The fiber bundle HDF5 formulation according to [10] ("F5") casts data into a non-cyclic graph of five levels, called the `Slice`, `Grid`, `Topology`, `Representation` and `Field` levels, with two additional invisible levels describing internal memory layout. This graph maps well to the hierarchical grouping scheme of HDF5 by identifying the nodes of the graph with HDF5 groups. Writing custom file converters to transform the time-varying surge surfaces, wind, pressure and temperature volumes into the `F5` format was a one-time effort. The conversion of MM5 data was done via translation into the intermediate NetCDF format[10] generated by the utilities available from the MM5 site[11]. Even though NetCDF shares some similarities with the HDF5 (self-description, platform independence), it is missing several of its crucial features, such as the capability of organization of datasets in named hierarchies as well as allowing their cross-referencing at metadata level. The second step of the translation focused on the

[8] `http://www.research.ibm.com/people/l/lloydt/dm/DM.htm`
[9] Hierarchical Data Format version 5 `http://hdf.ncsa.uiuc.edu/HDF5/`
[10] `http://www.unidata.ucar.edu/software/netcdf`
[11] `ftp://ftp.ucar.edu/mesouser/user-contrib/mm5tonetcdf_1.2.tar.gz`

extraction of the desired data volumes from the NetCDF files, reorganizing them in memory and storing the resulting buffers in correctly annotated F5 hierarchy. We simplified this approach by treating cell-related quantities as given on vertices and σ-level as height coordinates .

A file-system like listing of the 5-levels of the F5 structure of MM5 data (regular uniform grid with three fields) will then appear as

```
/T=1.0/MM5/Points/Cartesian Group
/T=1.0/MM5/Points/Cartesian/Positions Group
/T=1.0/MM5/Points/Cartesian/wind Dataset {43, 135, 174}
/T=1.0/MM5/Points/Cartesian/temperature Dataset {43, 135, 174}
/T=1.0/MM5/Points/Cartesian/pressure_perturbation Dataset {43, 135, 174}
```

whereby the fifth level contains the actual data, displayed here with their shared dimensionality.

The ADCIRC data set provides topological information about the connectivity of vertex points, explicit vertex coordinates and scalar values denoting surge elevation on each vertex. The F5 structure listing appears as:

```
/T=1.0/ADCIRC/Connectivity Group
/T=1.0/ADCIRC/Connectivity/Points Group
/T=1.0/ADCIRC/Connectivity/Points/Positions Dataset {1190404}
/T=1.0/ADCIRC/Points Group
/T=1.0/ADCIRC/Points/Cartesian Group
/T=1.0/ADCIRC/Points/Cartesian/Positions Dataset {598240}
/T=1.0/ADCIRC/Points/Cartesian/elevation Dataset {598240}
```

Going via F5 reduced the loading time of the large time-varying datasets provided originally as text files from several minutes to a fraction of a second. Moreover, the integrated caching algorithms in the HDF5 library itself eases loading of data on demand, both for ADCIRC as well as for MM5, as both can be accessed through the same interface. They may even be stored in the same file, thereby allowing to specify relationships among both simulations types and ensuring consistency (e.g., with respect to same timescale).

3.4 Data Import for Visualization

We used the Amira visualization tool [11] for rendering. It does not have an intrinsic notion of time-dependent objects and supports only static geometries well. Each `Grid` node in the fiber bundle hierarchy describes a geometry at a certain time step and can thus be mapped into a static geometry. The enveloping `Slice` level provides a sequence of `Grid` objects. Thus we extended the Amira class hierarchy by deriving dynamic objects from their static pendants. This recipe is straightforward to implement and scales well to the diverse data types.

As drawback, this approach does not allow to inspect more than one time step at once (e.g. in different viewers) except by copying the entire visualization network. Another implementation issue is that not all of the Amira base classes allow easy modification of their properties once created.

The mapping of `Grid` objects to static objects works fine for entirely time-varying objects (as long as the topological type, e.g. of being a triangular surface, remains the same). However, some components of the dynamic `Grid` may well remain constant. In particular, the connectivity and vertex location of the ADCIRC grid does not change through time, only the data values (surge elevation, wind velocity) evolve. We can address this issue by utilizing symbolic links among HDF5 datasets - a feature provided by HDF5 similar to a Unix filesystem. For instance, we make a symbolic link of the `Grid`'s connectivity information at time 1290.0 to the connectivity information of time 0.0 to indicate that it did not change, resulting a structure as follows:

```
/T=1290/ADCIRC/Connectivity Group
/T=1290/ADCIRC/Connectivity/Points Group
/T=1290/ADCIRC/Connectivity/Points/Positions Dataset,
⟶        same as /T=0/ADCIRC/Connectivity/Points/Positions
/T=1290/ADCIRC/Points Group
/T=1290/ADCIRC/Points/Cartesian Group
/T=1290/ADCIRC/Points/Cartesian/Positions Dataset,
⟶        same as /T=0/ADCIRC/Points/Cartesian/Positions
/T=1290/ADCIRC/Points/Cartesian/elevation Dataset {598240}
```

This way we can easily specify any property of an evolving `Grid` to remain constant, equally referring to the entire time range, just a time interval or even intermittent. This feature can well be utilized as certain ADCIRC runs are also performed on a mesh that is modified once during the simulation in order to cope with levee failure. We are not aware about any other file format which supports a comparable mechanism to express partial time-dependency.

4 Specific Visualization Algorithms

4.1 Atmospheric Data (MM5)

Amira provides many means of visualizing vector fields such as LIC, streamsurfaces and streamlines [12]. The non-commercial research version also includes advanced algorithms for extracting and displaying topological features [13,14] However, for our purposes of communicating the results of hurricane simulations to the public and scientists unfamiliar with vector field topologies, we found the the technique of illuminated stream lines [15] most intuitive beside simple vector arrows icons. While vector arrows are frequently used as a first step and easily convey the values of a vector field, they do not scale well to display its global structure. Streamlines are superior to depict features such as the vortex of a hurricane. The *seeding* of streamlines is a critical issue affecting the overall appearance. For a static view, we can manually seed the streamlines within region of interest such as the city of New Orleans (Fig. 1). This approach does not extend to a dynamic vector field, where streamlines are no longer appropriate at all due to their vastly changing character. However, we can reduce the length of the streamlines radically such they only depict local variations of the vector

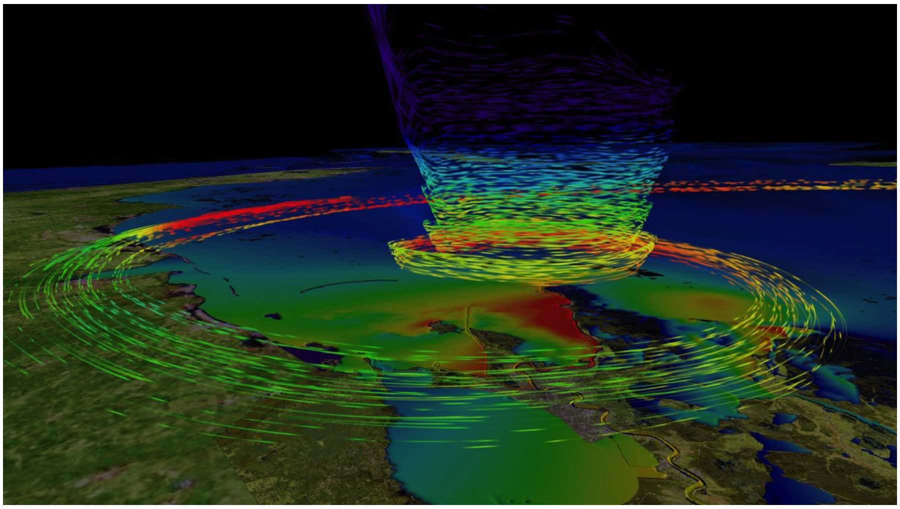


Fig. 1. Streamlines of the hurricane wind vector field at landfall. The streamlines are color-coded by temperature showing higher temperatures above sea surface than at land indicating loss of energy after landfall, while at the same time depicting the push onto the Lake Ponchartrain causing the flood in the city of New Orleans.

field, as these are more likely to be temporally smooth than global features. We need to compensate the smaller line fragments by increasing the density of lines. Consequently this increases the visual clutter again and requires suppressing of

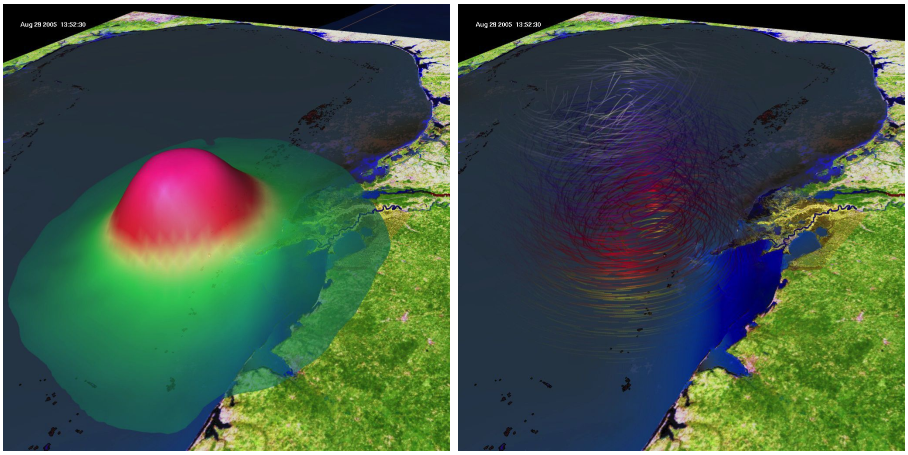


Fig. 2. The pressure scalar field indicates the location of the eye of the hurricane (left). We use it to set the transparency of the streamlines (right), thereby emphasizing the hurricane's eye in an automated way which is suitable for animation.

regions in the volume where the wind is of minor relevance. Such regions are indicated by the *pressure*, see Fig. 2. We therefore map this scalar field to the transparency of the stream lines and get wind field indicators limited to the vicinity of the eye of the hurricane.

4.2 Surge Data (ADCIRC)

The ADCIRC data set consists of more than one million triangles plus time-varying scalar for surge elevation and a vector field for wind information. In order to achieve good rendering performance we utilize OpenGL extensions such as Vertex Buffer Objects. Surge elevation is most intuitively represented by modifying the vertex locations like a height field. For a triangular surface given in 3D, this is not straightforward because there is a freedom of choice in which direction to extrude the surface at each vertex. For the special case here we may just concentrate on the z (height) direction of the surface, which denotes the bathymetry of the sea ground. We *blend* this bathymetric value and the surge elevation, as this provides a visually appealing mean to display structure within otherwise homogeneously watered regions.

4.3 Elevation Data (LIDAR)

A LIDAR data set is canonically visualized as a height-field. The extremely high-resolution of GIS elevation models, 11Kx7K in our case renders 154 million triangles using a brute-force triangulation method. Interactive rendering is virtually impossible with this approach. We have implemented Continuous Level-of-Detail (CLOD) techniques to dynamically simplify the mesh at run-time depending on the view point. Several algorithms already exist in this area. We chose the ROAM [16] algorithm due to its inherent simplicity and low memory overhead. At every frame, ROAM recursively tessellates the terrain generating triangles depending on the distance to the viewer criteria (or one could also use surface roughness). One nice feature of the recursive method is that we are not storing any per-vertex data but just generating them on the fly for the drawing, freeing up huge amounts of memory. We use the automatic texture coordinate generation functionality in OpenGL, mapping texture coordinates to the vertices. The drawback of this approach is heavy computation on the CPU and only using the GPU for drawing triangles.

4.4 Cloud Data (GOES)

The channel from the GOES satellites do not correspond to visual colors, so they cannot be used to create a true-color image. Five channels are beyond the capabilites of our trichromatic color perception anyway, so we face the challenge of appropriate representation. As we also require integrated display with atmospheric, surge, LIDAR and GIS data, the cloud representation needs to be minimalistic and we refrain from displaying all channels at once. The visible or IR

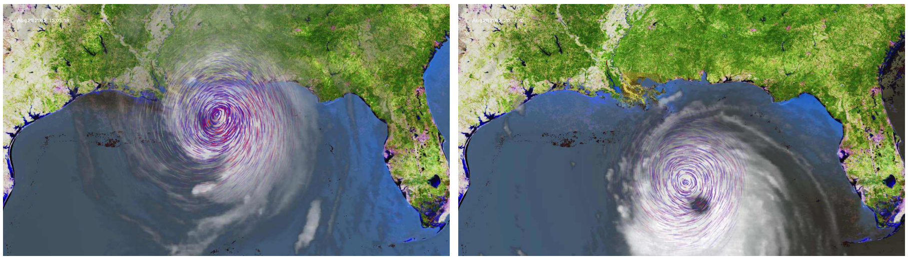


Fig. 3. Match (left) and mismatch (right) of atmospheric simulation and satellite data

channels are appropriately displayed as a transparent 2D gray-scale layer; after geospatial alignment their evolution allows depicting the match with atmospheric data (see Fig. 3), where the discontinuity in the MM5 model data mentioned above appears more or less prominently. Alternatively it is also reasonable to represent the long-wave IR channel as height field, because this channel which is directly related to the physical height of the cloud cover through their temperature. Thus we can employ again the height field rendering algorithm described earlier, using the GOES-12 visible channel as texture.

5 Conclusions

We have shown how various data sources ranging from computational models of storm-surge and wind fields to observational data from satellites and sensors can be integrated into a holistic, compelling and interactive visualization of Hurricane Katrina. The selected methods illustrate the interaction between topographical and surge data (LIDAR/ADCIRC), the development of the surge as predicted from the atmospheric model (ADCIRC/MM5) and allow to assess the deviation of the atmospheric model from observation (MM5/GOES), all within the geospatial context provided by GIS reference images. We have developed efficient data layout mechanisms to ensure fast and uniform access to the multiple time-varying datasets. Existing rendering techniques were also applied and extended to better understand the phenomenon and her effects. All these above efforts required new partnerships between coastal modelers, engineers and computer scientists.

Acknowledgements. We thank our colleagues Shreekanth Balasubramanian, Dewitt Braud, Chirag Dekate, Carola Jesch and Wei He and LSU collaborators - Earth Scan Laboratory, Coastal Studies Institute, CLEAR and Hurricane Center, SURA SCOOP, UNC Department of Marine Science and TAMU for invaluable assistance with the visualization and data access.

References

1. Allen, G., Bogden, P., Creager, G., Dekate, C., Jesch, C., Kaiser, H., MacLaren, J., Perrie, W., Stone, G., Zhang, X.: Gis and integrated coastal ocean forecasting. Concurrency and Computation: Practice and Experience (2006) (submitted)
2. MacLaren, J., Allen, G., Dekate, C., Huang, D., Hutanu, A., Zhang, C.: Shelter from the storm: Building a safe archive in a hostile world. In: Meersman, R., Tari, Z., Herrero, P. (eds.) On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops. LNCS, vol. 3762, pp. 294–303. Springer, Heidelberg (2005)
3. Nativi, S., Blumenthal, B., Habermann, T., Hertzmann, D., Raskin, R., Caron, J., Domenico, B., Ho, Y., Weber, J.: Differences among the data models used by the geographic information systems and atmospheric science communities. In: Proceedings American Meteorological Society - 20th Interactive Image Processing Systems Conference (2004)
4. Helgeland, A., Elboth, T.: High-quality and interactive animations of 3d time-varying vector fields. IEEE Transactions on Visualization and Computer Graphics (2006)
5. Doleisch, H., Gasser, M., Hauser, H.: Interactive feature specification for focus+context visualization of complex simulation data. In: VISSYM '03: Proceedings of the symposium on Data visualisation 2003, pp. 239–248. Eurographics Association, Aire-la-Ville, Switzerland (2003)
6. Zhang, K., Chen, S.C., Singh, P., Saleem, K., Zhao, N.: A 3d visualization system for hurricane storm-surge flooding. IEEE Computer Graphics and Applications 26(1), 18–25 (2006)
7. Butler, D.M., Pendley, M.H.: A visualization model based on the mathematics of fiber bundles. Computers in Physics 3(5), 45–51 (1989)
8. Butler, D.M., Bryson, S.: Vector bundle classes from powerful tool for scientific visualization. Computers in Physics 6, 576–584 (1992)
9. Benger, W., Hege, H.-C., Merzky, A., Radke, T., Seidel, E.: Efficient Distributed File I/O for Visualization in Grid Environments. In: Engquist, B., Johnsson, L., Hammill, M., Short, F. (eds.) Simulation and Visualization on the Grid. Lecture Notes in Comp. Science and Engineering, vol. 13, pp. 1–6. Springer, Heidelberg (2000)
10. Benger, W.: Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model. PhD thesis, Free University Berlin (2004)
11. Stalling, D., Westerhoff, M., Hege, H.C.: Amira - an object oriented system for visual data analysis. In: Johnson, C.R., Hansen, C.D. (eds.) Visualization Handbook, Academic Press, London (2005)
12. Stalling, D.: Fast Texture-Based Algorithms for Vector Field Visualization. PhD thesis, Free University Berlin (1998)
13. Theisel, H., Weinkauf, T., Hege, H.C., Seidel, H.P.: Saddle connectors - an approach to visualizing the topological skeleton of complex 3d vector fields. In: Turk, G., van Wijk, J.J., Moorhead, R. (eds.) Proc. IEEE Visualization 2003, Seattle, U.S.A., pp. 225–232. IEEE Computer Society Press, Los Alamitos (2003)
14. Weinkauf, T., Theisel, H., Shi, K., Hege, H.C., Seidel, H.P.: Extracting higher order critical points and topological simplification of 3D vector fields. In: Proc. IEEE Visualization 2005, Minneapolis, U.S.A., pp. 559–566. IEEE Computer Society Press, Los Alamitos (2005)

15. Zöckler, M., Stalling, D., Hege, H.C.: Interactive visualization of 3d-vector fields using illuminated streamlines. In: Visualization '96, pp. 107–113 (1996)
16. Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., Mineed-Weinstein, M.B.: Roaming terrain: Real-time optimally adapting meshes. In: Proceedings of IEEE Visualization '97, pp. 81–88. IEEE Computer Society Press, Los Alamitos (1997)

Generation of Microlensing Magnification Patterns with High Performance Computing Techniques*

Pablo López[2], Antonio J. Dorta[1], Evencio Mediavilla[2], and Francisco de Sande[1]

[1] Depto. de E.I.O. y Computación, Univ. de La Laguna, ES-38271–La Laguna, Spain
{ajdorta, fsande}@ull.es
[2] Instituto de Astrofísica de Canarias (IAC), ES-38205–La Laguna, Spain
{plopez, emg}@iac.es

Abstract. The use of high performance computing techniques has not reached the level of simplicity needed to allow their use by the average researcher. Interdisciplinary groups are still a necessity in many cases. We present the result of a collaboration between astrophysicists and computer science researchers as part of a project to develop a useful tool for astrophysics research. Three different parallelization tools, OpenMP, MPI and `llc`, are compared in order to determine which one is the most appropriate in terms of performance and ease-of-use.

1 Introduction

High Performance computers have became an essential part of modern research, but software tools used to harness their power have not reached the level of simplicity expected by the average researcher. They are familiar with the logic of sequential coding, but the time needed to learn these tools is seen as a distraction from their field of research. Furthermore, most of the issues related to parallel programming (such as concurrency, message passing, etc.) are foreign to them. Last, but not least, codes are usually developed on their workstations which are single processor systems. They can develop, test and run without having to deal with queues, shared resources, etc., using tools they are comfortable with. This generates situations where problems that can be solved with current technology are put aside because the execution times are too large to consider the programs useful. In many cases, only through collaboration with researchers in computer science can these codes be transformed into tools that allow for further advances in their fields.

In this work we expose a case of a simple sequential code, where parallel programming is needed to enable fast computing. A high performance computing background is needed to solve this parallelization problem. Indeed, astrophysics

* This work has been partially supported by the EC (FEDER) and the Spanish MEC (Plan Nacional de I+D+I, TIN2005-09037-C02-01).

researchers at the Instituto de Astrofísica de Canarias (IAC) developed a Fortran77 sequential code to generate magnification patterns induced by micro lensing. The problem was the execution time. A simple pattern took aproximately 10 hours to complete. It was clear that only through the use of high performance computing techniques could this code become useful. Our main objective was to reduce the execution times to the point that would allow the generation of large numbers of these patterns within a reasonable time. A second objective was to evaluate different parallelization tools that can be used with these types of problems.

The remainder of the paper is organized as follows: In order to understand the value of this research and its computational complexity, an introduction to the problem is given in Section 2. In Section 3 we describe the method to generate the magnification patterns. Section 4 describes the different parallelizations that have been implemented. Section 5 presents the results of the computational experience. Finally, we summarize a few concluding remarks and future work in Section 6.

2 The Problem

Deflection of light by gravity is one of the most outstanding predictions of general relativity. It was experimentally confirmed by an expedition to measure the displacement of the apparent positions of stars on the sky caused by the gravitational field of the Sun during the 1919 eclipse. In spite of its great importance as a test of general relativity, the deflection of star light by the Sun or any other star has a relatively weak effect. After the discovering of galaxies, astronomers realized that the huge gravitational field of these objects may induce strong optical effects like the formation of multiple images (in a way similar to terrestrial atmospheric mirages). But, it was only 60 years after the expedition to the 1919 eclipse, that a double image of a very far away object caused by the gravitational field of an intervening galaxy (the lens galaxy) was discovered. The distant source was a quasar, a galaxy that hosts in a tiny nuclear region a super-massive black hole surrounded by a shining disc of in-spiraling matter (active galactic nucleus). The brightness of the two images of the quasar is different and depends on the source and lens distances and on the degree of alignment among observer, lens and source. As far as this geometry is not changing, the ratio between the brightness of the two images should be constant. However, the distribution of matter in galaxies is not smooth but strongly discontinuous. It is granulated in stars and, perhaps, other compact objects. Thus, the movement of one star (or more realistically of a distribution of stars) crossing the light beam of one of the quasar images can produce a small scale but measurable gravitational lens effect (quasar microlensing). As long as the movements of the stars in the regions where the light-beams of the two images cross the lens galaxy are independent, microlensing will induce uncorrelated variability in the brightness of the two images. This variability will give us information about two important physical issues very difficult to study by other means: the distribution of stars in the lens galaxy and the unresolved structure of active galactic nuclei.

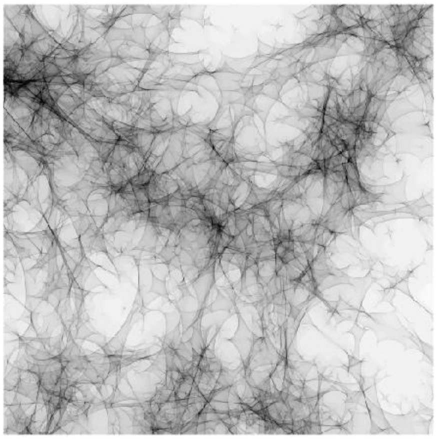


Fig. 1. A sample magnification pattern

The study of the experimental records of the brightness variability of lensed quasars with time (light curves) faces a statistical (formally stochastic) problem. We must find which configurations of stars in the lens galaxy and relative displacements of the source that can reproduce the observed variability. To obtain statistically acceptable estimates of the physical magnitudes involved, a large number of light curves based in random distributions of stars and relative displacements should be generated. Usually this is attempted by computing magnification patterns that give, as a function of position, the magnification induced by microlensing in the source plane. The displacements of the source across this pattern will generate the model light curves.

```
for (i = 0; i < end_y; ++i)
    for (j = 0; j < end_x; ++j)
        /* Calculate the deflection caused */
        /* by the star field               */
        x1 = i
        y1 = j
        for (index = 0 ; index < number_stars; ++index)
            x1,y1 = deflect(x1, y1, star[index])
        /* Translate the ray coordinates to the */
        /* resulting matrix coordinates         */
        x,y = transform(x1,y1, N, end_x, M, end_y)
        /* If the resulting coordinates are within the */
        /* matrix range, update the matrix element.    */
        if  x > 0 and x < N
        and y > 0 and y < M
            result[x,y] = result[x,y] + 1
```

Listing 1. Pseudo code

3 The Algorithm

Magnification patterns are usually computed using the inverse ray shooting technique that consists of back shooting a regular grid of rays from the image plane to the source, making the amplification in a given source-plane pixel proportional to the number of light rays collected by it. This technique implies a high computational cost. On one hand, around 2^8 rays per pixel should be shot to obtain magnification patterns with reduced noise. On the other hand, the ray equation includes contributions for thousands of stars that should be evaluated for each shot.

Listing 1 presents the pseudo code to generate the magnification pattern. The loops in lines 1 and 2 generate the regular grid of rays which represents the image plane. The loop in line 5 determines the effect of the gravitational field of each star on the ray path. Finally, in line 12, the resulting coordinates update the matrix that represents the source plane.

The original sequential code used for the generation of the magnification patterns is quite straightforward, but its execution time, even for cases with a small number of stars, was unacceptable. Considering the fact that the number of executions needed to obtain a statistically acceptable estimate is approximately one hundred, we can only conclude that the usefulness was, to say the least, limited. In order to make it a viable tool, execution time had to be reduced by at least one order of magnitude. Clearly this could only be achieved through the use of high performance computing.

```
        /* Main loop */
        for ( i = 0; i < end_y; ++i) {
            ...
#pragma omp parallel for private(j, ...)
            for ( j = 0; j < end_x; ++j )
            {
                  ...
                  for ( index = 0; index < number_stars; ++index ) {
                       ...
                  }
                  if ( x >= 0 && x < N &&
                       y >= 0 && y < M ) {
                       pos[j] = x + y * N
                  }
             }

             /* Update the resulting matrix */
             for (j = 0 ; end_x; ++j) {
                  if ( pos[j] != -1) {
                       result[pos[j]] += 1;
                       pos[j] = -1;
                  }
             }
        }
```

Listing 2. OpenMP code

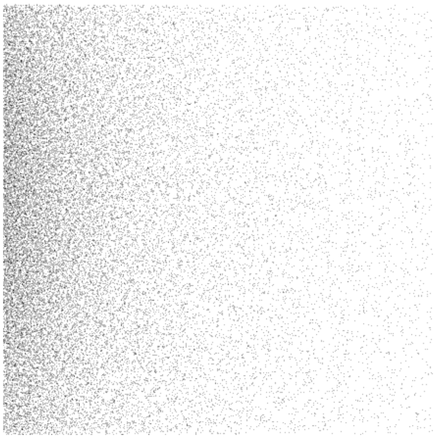


Fig. 2. Hits distribution after processing 100 lines

4 Parallelization

One should expect a high level of speed up in a parallel version of the sequential code, given that the processing of each light ray is independent from the others. While it may appear as a simple problem of parallelizing nested loops, a difficulty arises from the way that the memory is accessed. Figure 1 is a common example of a microlensing magnification pattern. Areas with a large number of ray hits are shown in dark, while white areas represent a low density of ray hits.

As described in Section 3, the rays are launched as a regular grid (lines 2 and 5 of Listing 2) and are then deflected in different directions by the star field causing a large amount of hits, that means updates of the same memory position in some areas of the resulting matrix, while other areas get very few. We observe that the distribution of the rays hits is irregular. Due to the random nature of the star field, there is no way to determine any access pattern beforehand. In Figure 2, we can observe the effect of the deflection after one hundred lines of rays have been processed. What was originally a regular grid has became a cloud of points spread throughout the resulting matrix. We are dealing with a sparse access pattern with, as explained before, a high level of concurrency. Therefore, apart from the concurrency problem, we also have to deal with minimizing the number of cache misses that are intrinsic to this disperse memory access. A final problem arises by the fact that the algorithm looses precision at the edges of the star field. To avoid this problem, the resulting matrix represents only the central area of the magnification pattern. A significant amount of rays fall outside the matrix and do not produce an update. This generates a load imbalance situation.

In order to parallelize the code, these issues had to be taken into account.

We decided to try three different parallelization tools: OpenMP, MPI and `llc`. Nowadays, OpenMP [1] and MPI [2] are universally accepted as the standard tools to develop parallel applications. `llc` ([3], [4]) is a C based parallel language

```
chunk_size = end_y / MPI_NUMPROCESSORS;
start = MPI_NAME * chunk_size;
if (MPI_NAME == (MPI_NUMPROCESSORS - 1)) {
   chunk_size += (end_y % MPI_NUMPROCESSORS);
   end = end_y;
} else {
   end = (MPI_NAME + 1) * chunk_size;
}

for ( i = start; i < end; i++ ) {
   ...
   for ( j = 0; j < end_x; ++j )
   {
      for ( index = 0; index < number_stars; ++index ) {
         ...
      }
      ...
      if ( x >= 0 && x < N &&
           y >= 0 && y < M ) {
           result[x + y * N] += 1;
      }
   }
}

if (MPI_NAME == 0) {
   for (i = 0; i < (MPI_NUMPROCESSORS - 1); i++) {
      MPI_Recv (s_aux, size, MPI_FLOAT, MPI_ANY_SOURCE,
                MSG_TAG, MPI_COMM_WORLD, &status);
      for (j = 0; j < size; j++) {
         result[j] += result_aux[j];
      }
   }
} else {
   MPI_Send (result, size, MPI_FLOAT, 0, MSG_TAG,
        MPI_COMM_WORLD);
}
```

Listing 3. MPI code

that uses compiler pragmas similar to OpenMP to express parallelism. `llCoMP`, the `llc` compiler, is a source to source compiler that generates MPI code. For the interested reader we refer to [4] for further details regarding the language and its implementation.

Listing 2 outlines the OpenMP code version. The main computational area of the code consists of two nested loops (lines 2 and 5). Prior to reaching the code in Listing 2, the access pattern to the resulting matrix had to be modified to avoid concurrent accesses when parallelizing the inner loop. This was achieved by using a temporary array of size (N * number of rays per pixel) to store the address of the pixel to be updated in the resulting matrix. Upon leaving the parallel loop, the resulting matrix gets updated by using the contents of the temporary array as the index (lines 18 – 23 in Listing 2).

The parallelization of the external loop (line 2) does not require modifications to the memory access pattern in the innermost loop. While being the most simple part, it is also very memory intensive. The resulting matrix for the average problem has a size of 64Mb. If this loop is parallelized, all the threads would replicate this matrix. While this is not a problem in clusters where each processor has its own memory, it may be an issue on shared memory systems. For the

```
#pragma llc for
#pragma llc reduce (result , result_aux , size , LLC_SUM)
     for ( i = 0; i < end_y ; ++i) {
         ...
         for ( j = 0; j < end_x ; ++j )
         {
               ...
               for ( index = 0; index < number_stars ; ++index ) {
                    ...
               }

               if ( x >= 0 && x < N &&
                    y >= 0 && y < M ) {
                    result[x + y * N] += 1;
               }
          }
     }
```

Listing 4. `llc` code

same problem, the size of the temporary array is 32Kb. Therefore, parallelizing the internal loop is the best choice for the OpenMP version. On the contrary, for MPI and `llc` it was clear that the parallelization of the external loop was the best choice due to the larger grain size. Still, in order to make a more complete comparison between MPI and `llc` we also developed internal loop parallelizations using both tools. Finally, a master-slave implementation, both in MPI and `llc`, has also been included as part of the computational experience in order to evaluate the impact of the load imbalance pointed out at the beginning of this section. For code complexity comparison purposes Listing 3 and 4 schematically present the MPI and `llc` implementations respectively for the external loop parallelization.

In order to parallelize the external loop in `llc` we only have to add two pragmas before the external loop. The first pragma declares a parallel for as it is done in OpenMP and the second pragma defines a reduce operation to be performed on the resulting matrix.

5 Computational Results

We present results for three different architectures: a shared memory system Bull NovaScale 6320 server with 32 1.5 GHz Intel Itanium 2 processors with a Quadrics QsNetII (Figure 3), a distributed shared memory system IBM RS-6000 with 8*16 Nighthawk Power3 375Mhz processors with a SP Switch2 at 500MB/sec (Figure 4) and a distributed system Dell PC cluster with 32 Intel Xeon 2.8 GHz processors using a gigabit Ethernet (Figure 5).

Figure 3 shows no difference between the OpenMP version, the external loop parallelizations of MPI and `llc` and the internal loop MPI parallelization. On one hand we can observe a fall in scalability of the `llc` internal loop parallelization. This behavior is also present in Figure 4 and most noticeable in Figure 5, the system with the lower interconnecting bandwidth. The `llc` internal loop parallelization seems to suffer a significant degradation which is probably caused by

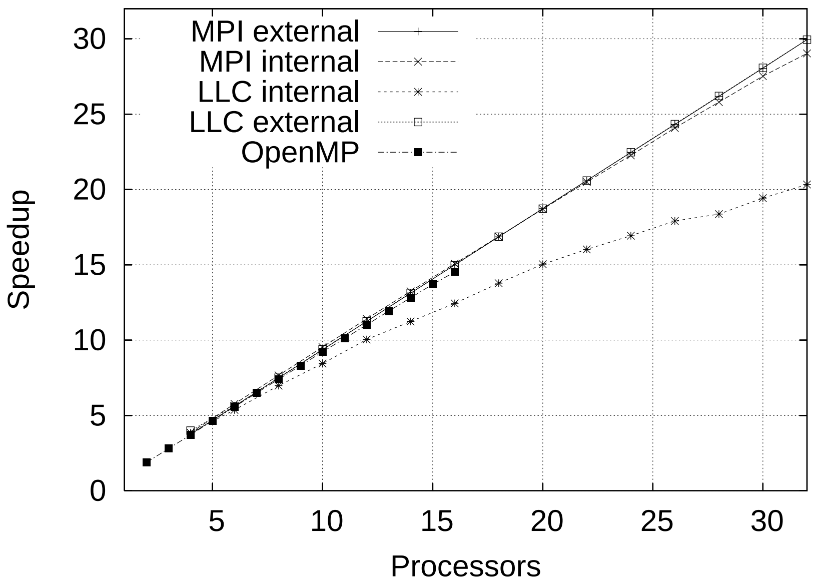


Fig. 3. BULL NovaScale

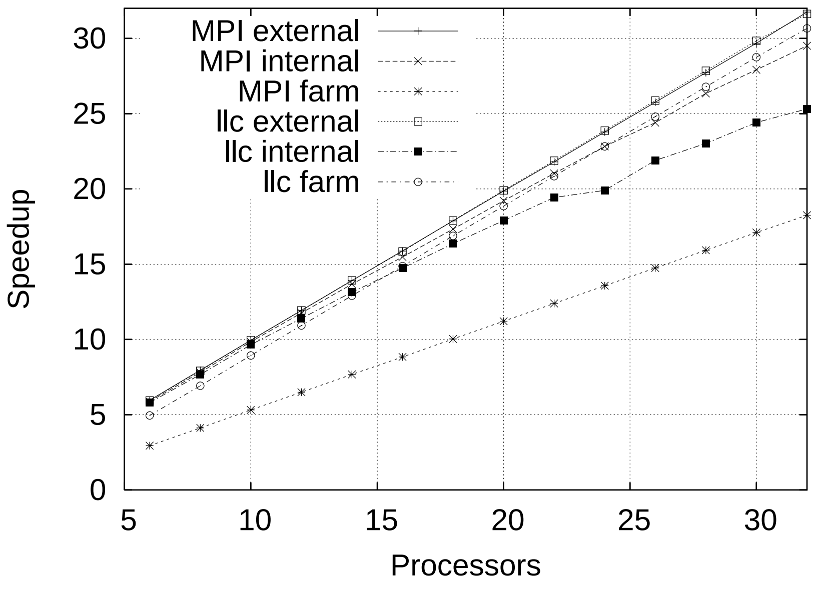


Fig. 4. IBM RS-6000

an inefficient handling of the excess communication. On the other hand the external loop parallelization of MPI and `llc` show no performance difference in any of the systems. In Figures 4 and 5, the master-slave versions of the code

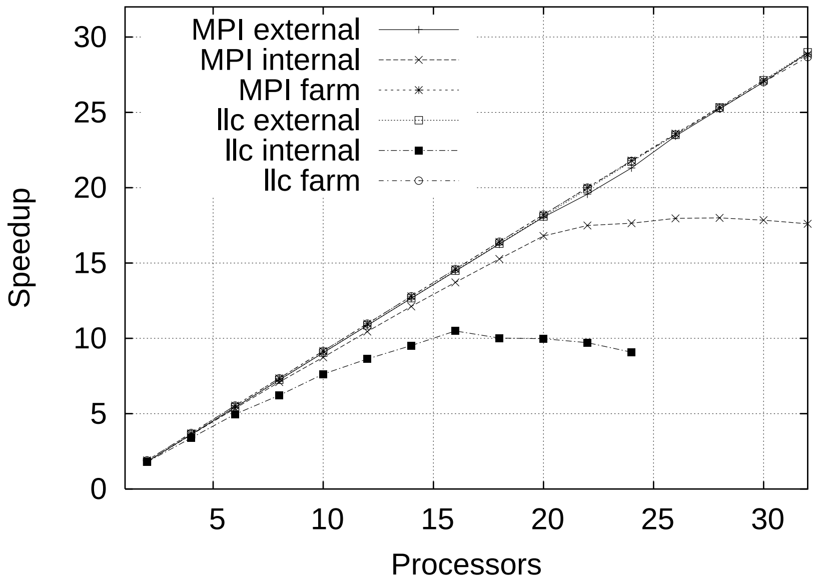


Fig. 5. Dell PC Cluster

also seem to scale equally well. The lower performance of the master-slave MPI version on the IBM RS-6000 system was caused by a high system load during the experiment.

It is clear from the graphs that the best performance across all architectures are the external loop parallelization and the master-slave for MPI and `llc`. We can also determine that the load imbalance situation described in section 4 does not seem to have any significant impact on performance. Due to the added complexity of the master-slave, we can reduce our choice to the external loop parallelization. Finally, the choice is only between MPI and `llc`. Comparing Listings 3 and 4, the choice is clear; `llc` is simpler than MPI while offering the same performance.

The overhead in communications in the internal loop `llc` parallelization that we have pointed before, implies that as the number of processors increase, MPI will be able to maintain a better scalability than `llc`. Given that the execution time on the Bull system using 32 processors was under 10 minutes, it is clear that the possible gain in performance provided by the higher scalability of MPI does not compensate for the added complexity.

6 Conclusions

We can conclude that between the three tools that we have used, OpenMP, MPI and `llc`, the best choice for distributed systems is `llc`. The simplicity of the OpenMP-like syntax, combined with its high performance makes it the ideal tool for parallelizing this type of problems. For shared memory systems, OpenMP should be the choice if memory usage is a problem.

We have also shown that, the cooperation between researchers in science and high performance computing can overcome problems that for the scientist may seem too computing intensive to be viable. In this particular case, through code optimization and parallelization, we were able to reduce the execution times by a factor of 56.

Work in progress consists of the development of methods for extraction and analysis of the light curves from the magnification patterns using high performance techniques. Once this step is completed, we will be ready to offer a complete tool to the astrophysics community.

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface, vol. 2.5 (2005)
2. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. University of Tennessee, Knoxville, TN (1995), `http://www.mpi-forum.org/`
3. Dorta, A.J., González, J.A., Rodríguez, C., de Sande, F.: llc: A parallel skeletal language. Parallel Processing Letters 13(3), 437–448 (2003)
4. Dorta, A.J., Lopez, P., de Sande, F.: Basic skeletons in llc. Parallel Computing 32(7-8), 491–506 (2006)

Phase Space Modulations in Magnetised Plasmas by a Mildly Relativistic Two-Stream Instability

Madelene Jeanette Parviainen, Mark Eric Dieckmann, and Padma Kant Shukla

Theoretische Physik IV, Ruhr-Universität Bochum, D-44780 Bochum, Germany
{madpa, markd, ps}@tp4.rub.de

Abstract. A kinetic particle-in-cell simulation and 3D point-rendering visualisation are used to investigate a two-stream plasma instability, possibly found in the accretion disc of black holes. A plasma in an oblique external magnetic field is considered. The instability gives rise to a quasi-electrostatic wave able to trap electrons and accelerate them by cross-field transport. The results of the simulation show an acceleration of the electrons to speeds similar to the bulk speed of microquasar jets.

Keywords: collisionless plasma, oblique magnetic field, astrophysical jets, electron surfing acceleration, PIC simulation, visualisation.

1 Introduction

Relativistic plasma flow in astrophysical environments such as microquasars[1], active galactic nuclei or quasars[2] and gamma ray bursts[3], implies the existence of efficient particle acceleration mechanisms. The transient jets of microquasars can reach Lorentz factors[1] of a few. The collimated jets of active galactic nuclei may reach Lorentz factors of 10-100 and gamma ray burst jets probably reach Lorentz factors of 100-1000. Since in-situ measurements through satellite experiments at these remote environments cannot be performed as is done in solar system plasmas[4], the underlying mechanisms have to be deduced from limited observational data. Key similarities between microquasars, active galactic nuclei and probably the long gamma ray bursts is the presence or formation of a compact object (a black hole or a neutron star) surrounded by an accretion disc and a jet coupled with the disc[5].

The accretion disc is formed by material that is accelerated towards the compact object[6] but prevented from falling directly into it by the material angular momentum. Thermal processes within the disc allow for the transport of angular momentum across the disc. The energies involved in mass accretion give that these thermal processes may be the sources of jets, electromagnetic emissions and cosmic ray particles. The collective mechanisms in the accretion disk dynamics are intrinsically multi-scale and range from global thermo-viscous instabilities[1]

[1] The Lorentz factor $\gamma = \left(1 - (v/c)^2\right)^{-1/2}$ and v/c is the velocity magnitude to the speed of light ratio.

down to small scale kinetic instabilities, driven by charged particle beam dissipation. Charged particle beams are known from satellite observations in solar system plasmas and laboratory plasma experiments and may arise from internal accretion disc shocks[7][8] or magnetic field reconnection events[9][10]. Macroscopic instabilities can be understood by hydrodynamic or magnetohydrodynamic models[11] whereas the energy dissipation of charged particle beams are kinetic in nature. Our research addresses the latter and is based on particle-in-cell (PIC) simulations[12]. The PIC code solves Maxwell's equations together with the relativistic Newton-Lorentz equation as the particle equation of motion.

The purpose of our work is to examine the thermalisation of relativistic particle beams in case studies relevant for astrophysical plasma environments. In particular we assess the efficiency of beam thermalisation in generating relativistic flow, energetic particles and, within the limits of the kinetic approximation and finite code resolution, electromagnetic radiation. The present work focuses on the two-stream (Buneman) instability[13] in combination with a weak magnetic field not aligned with the beam flow direction. This instability results in quasi-electrostatic upper hybrid waves which saturate by trapping electrons in the wave potential[14]. The trapped electrons move on average with the phase speed of the wave. In the presence of a magnetic field, with a component perpendicular to the wave propagation direction, the trapped electrons undergo acceleration due to the cross-field transport. The special case of a perpendicular magnetic field is known as electron surfing acceleration(ESA)[15][16]. ESA has been shown to produce highly energetic electrons if the ion beam driving the two-stream instability propagates with a mildly relativistic speed. The peak energy gain is however constrained by the stability of the saturated wave[17][18]. Previous work has shown that an increase in the beam speed from $0.6c$ to $0.9c$ to $0.99c$ yields peak Lorentz factors $\gamma(0.6) = 10$, $\gamma(0.9) = 200$ and $\gamma(0.99) > 1000$ [19][20] respectively. Such electrons will emit synchtrotron radiation through their gyration in the external magnetic field. As a consequence of the perpendicular magnetic field geometry, no magnetic field aligned relativistic plasma flow is generated by the ESA. The electrons remain confined at the acceleration site and are eventually cooled down by their synchrotron radio emission. Tilting the magnetic field can introduce field-aligned flow that may feed relativistic astrophysical jets. In Refs.[21][22] the relativistic boost of the magnetic field, however, implied initial conditions similar to ESA[2]. In this paper the weaker Lorentz boost gives a different scenario to that in Refs.[21][22] and our aim is to investigate whether relativistic bulk flow speeds can be achieved for the considered plasma parameters.

The outline for the paper is: Section 2 discusses the underlying equations to the numerical code. Section 3 gives the simulation model and setup and discusses the point-rendering visualisation application.The PIC-simulation results are treated in section 4.

[2] A Lorentz transformation to the frame of reference of the wave affects only the perpendicular magnetic field component.

2 Particle-in-Cell Simulation

The `TwoDEM` PIC code is based on the virtual particle scheme[23] and approximates the phase-space distribution of a space plasma by a collisionless fluid. The fluid is represented by a set of computational particles evolved in time by the relativistic Newton-Lorentz equation,

$$\frac{d\mathbf{p}}{dt} = \frac{d(m\mathbf{v})}{dt} = q\left(\mathbf{E} + \mathbf{v} \times \mathbf{B}\right), \tag{1}$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}, \tag{2}$$

where m is the relativistic mass of the particle and q the particle charge, and Maxwell's equations,

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{j} + \epsilon_0 \mu_0 \frac{\partial \mathbf{E}}{\partial t}, \tag{3}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \tag{4}$$

using standard notation. The code automatically ensures $\nabla \cdot \mathbf{B} = 0$ to round-off precision. The current assignment scheme, known as the virtual particle scheme[23], fulfills exactly a discretised form of $\nabla \cdot E = \rho/\epsilon_0$ and no separate correction term is necessary. However, it results in higher noise levels. The simulation results are made more general with normalised units and the following transformations from physical (unprimed) to normalised (primed) electric and magnetic fields are used; $\mathbf{E}' = e\mathbf{E}/cm_e\omega_{p,e}$ and $\mathbf{B}' = e\mathbf{B}/m_e\omega_{p,e}$. The normalised current density $\mathbf{j}' = \mathbf{j}/n_e ce$. We substitute time and space variables $t' = t\omega_{p,e}$ and $x' = x\omega_{p,e}/c$. The velocity is $\mathbf{v}' = \mathbf{v}/c$ and the charge and mass of particle i are $q_i' = q_i/e$ and $m_i' = m_i/m_e$. Here m_e and e are the electron rest mass and the fundamental charge magnitude respectively. The electron plasma frequency $\omega_{p,e} = \sqrt{\frac{n_e e^2}{\epsilon_0 m_e}}$, where n_e is the electron number density. Under appropriate initial conditions, waves will self-consistently grow from noise level to non-linear saturation. We have chosen to focus on the dominant quasi-electrostatic two-stream modes. Thus we are able to use a simple geometry of the simulation box and resolve only one spatial dimension. The remaining spatial components are represented by one cell only. Periodic boundary conditions are set in all directions. The code is implemented in Fortran 90 and parallelised with the MPI library. Each computational node has the full electromagnetic field data available and the simulation particles are distributed evenly, where a node always keeps the same set of particles. At each time step, all nodes compute the partial current that arise from its subset of computational particles. The partial currents are then communicated to every node and the total current is summed up through a `MPI_Allreduce` command. The total current is used to update the electromagnetic fields in Maxwell's equations and the fields then update the particle velocities. Perfect load balancing is achieved since each node has the same number of particles. The drawback of the parallelisation scheme is the rapid increase of data communication with increasing simulation grid size. However, the scaling is almost linear up to 20 processors for typical research problems.

3 Simulation Model

Free energy sources are abundant in astrophysical plasmas and fundamental in the process of plasma heating. Since the plasma is collisionless the question is what kinds of dissipation mechanisms allow for the thermalisation. Velocity gradients in accretion disc and jet plasmas are likely to form shocks. Plasma particles gyrate in the shock front magnetic field and are reflected into the upstream plasma, here the particle momentum is conserved in the shock frame of reference. Such specularly reflected ions will form fast beams in the upstream plasma frame of reference. The energy of an external disturbance will, in the absence of collisions, become redistributed through the generation of instabilities. Assuming the cold beam model in Ref.[13] with a positive beam speed, v_b, and solving for the most unstable electrostatic wave we obtain the frequency $\omega_u \approx \omega_{p,e}$ and wave number $k_u \approx \omega_{p,e}/v_b$. Thus the phase speed of the wave is approximately equal to the speed of the beam feeding it. In the simulation

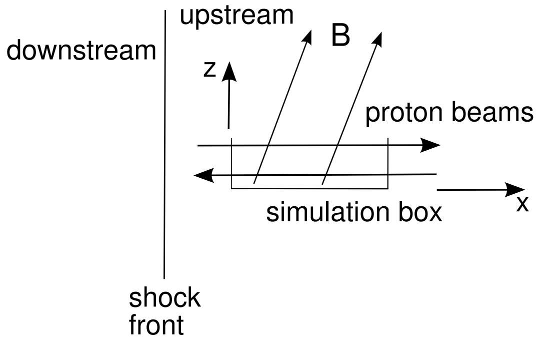


Fig. 1. The simulation box is placed in the upstream plasma in the foreshock region. Two counter-propagating proton beams are present at time zero and cancel the initial net current. The beams encounter the magnetic field at an oblique angle.

model, see Fig.1, we assume that the simulation box is located far from the shock front. The shock generation itself and the reflection of protons are not modelled here since this requires a full kinetic shock simulation, see Fig.2. In Fig.2 the structure 1 is upstream protons rotated in the shock-compressed downstream magnetic field and reflected back into the upstream region[8]. This proton beam constitutes the energy source considered in the present work.

The electron plasma frequency is set to $\omega_{p,e} = 2\pi \times 10^5$. The number densities of the proton beams $n_{b1} = n_{b2} = 0.1n_e$ and the number density of the background plasma protons $n_p = 0.8n_e$. The initial thermal speed of the electrons is $v_{th,e} = \sqrt{\frac{k_B T_e}{m_e}} = 10^{-2}c$, where k_B is Boltzmann's constant and T_e the

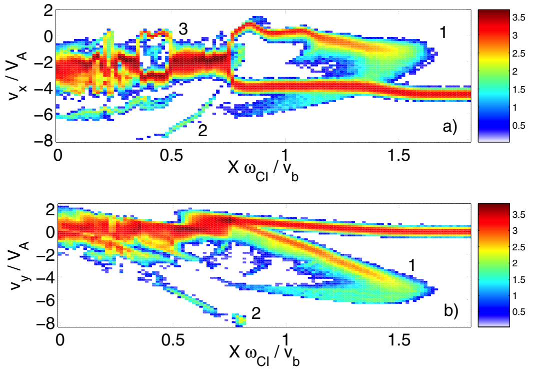


Fig. 2. Proton distribution of a one-dimensional perpendicular shock developing when an electron-proton plasma slab, coming from the right, impacts on an electron-proton plasma slab at rest. The collision speed is $4.5V_A$, where $V_A = B/\sqrt{\mu_0 n_i m_i}$ is the Alfven speed and B, n_i, μ_0 and m_i are the magnetic field strength, the ion number density, the magnetic permeability and ion mass respectively. Plot a) shows the (x, v_x) phase-space and plot b) the (x, v_y) phase-space. Structure 2 is protons trapped in the shock potential and carried across the upstream magnetic field. These protons undergo shock surfing acceleration[24]. Structure 3 consists of islands of trapped protons[25]. The colour scale is the log_{10} of the number of simulation protons.

electron temperature. The thermal speeds of the background protons and beam 1 are set to $v_{th,p} = v_{th,b1} = \sqrt{\frac{m_e}{m_p}} v_{th,e}$ whereas for beam 2 $v_{th,b2} = 10 v_{th,b1}$. The asymmetry in the initial beam conditions is used in order to examine the robustness of the physical model against geometry considerations. The proton mass to electron mass ratio in the simulation is $m_p/m_e = 1836$. The average velocities along the simulation box x-axis for the background plasma particle species are initially zero and the beam velocities are $\mathbf{v_{b1}} = -\mathbf{v_{b2}}$ with $\mathbf{v_{b2}} = v_b \hat{x} = 0.6c\hat{x}$, with $\hat{x}$ the unitvector. The chosen thermal speeds give the same temperature for the background plasma species and beam 1. At the beginning of the simulation there is no electric field present. The external magnetic field is directed in the xz-plane, $\mathbf{B} = B_0(\cos\theta, 0, \sin\theta)$, with $B_0 = 5 \times 10^{-7}$ T and $\theta = 45°$. The total simulation time $T_{tot} = 500{,}000\Delta_t$, with time step $\Delta_t = 5 \times 10^{-9}$ s. The simulation box length $L = 1200\Delta_x$, where the cell size $\Delta_x = 3$ m. The particle species are represented with computational particles that are initially distributed spatially homogeneously in the simulation box. The particles per cell (ppc) count for the electrons is 1250 ppc and 450 ppc for each proton species.

Data Visualisation. The capacity of high-performing computer clusters is continuously improving but analysing ever growing amounts of data is turning into a complex task. Especially when dealing with time-evolving, multi-dimensional data. A common work situation is to assign the heavy computational work load to a cluster and then analyse the data on desktop systems. Recently it has

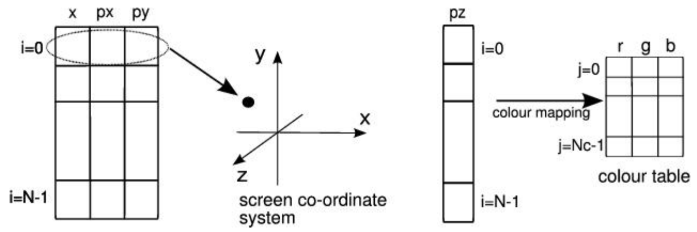


Fig. 3. The (x, p_x, p_y)-coordinates of the particles are stored in a matrix structure with three columns and N rows for a single time step. The phase-space co-ordinates are then treated as screen position co-ordinates. The p_z-coordinate index a colour look-up table for setting the particle colour. The index i is the identity number of a particle, N the total number of particles and N_c is the number of colours in the colour table.

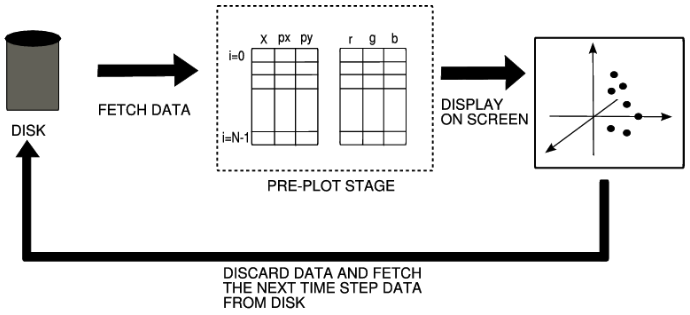


Fig. 4. The animation of data is done by fetching data and displaying it on screen in a serial fashion. The pre-plot stage could be done outside the visualisation application but would require extra storage. The main bottleneck is fetching the data from disk and recent dual-core systems may offer a solution in parallelisation schemes.

become possible to implement and run customised visualisation applications on common PCs, with the advantage of application design flexibility and user accessibility. The Open Graphics Library (OpenGL)[26] provides an interface to the graphics hardware and is integrated into most platforms. A straightforward technique of visualising the PIC-simulation data is to map the phase-space of the computational particles to a three-dimensional coordinate system and render them on screen. In order to use built-in structures in the OpenGL, before any call to plot the data is issued, particle coordinate and colour data are arranged in matrices with dimension $3 \times N$, where N is the total number of particles. Fig.3 explains the internal data structures in the visualisation application and how the four-dimensional phase-space data of each particle is mapped to screen coordinates and colour. Dealing with time-evolving structures, we would like to animate a sequence of time steps. The Fig.4 displays the animation sequence where time step data is fetched from disk, arranged in the pre-plot stage, displayed on screen and finally discarded to make room for the next data.

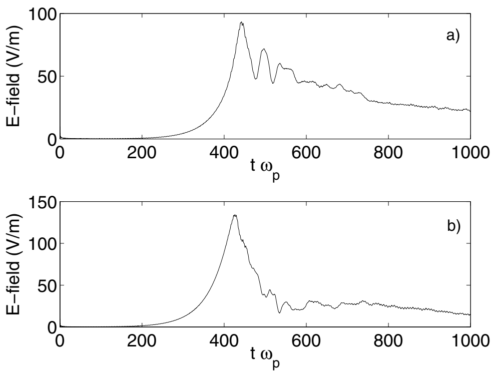


Fig. 5. The amplitudes of the most unstable quasi-electrostatic waves. The wave with negative phase speed is shown in a) and the wave with positive phase speed in b).

4 Results and Conclusions

We have used a PIC-simulation to follow the growth and saturation of quasi-electrostatic waves due to mildy relativistic proton beams in an obliquely magnetised plasma. The beam driven two-stream instabilities saturate by electron trapping. Energy is transfered from the protons to the trapped electrons through the exchange with the wave and by the particle transport across the magnetic field. The initial conditions in the simulation corresponds to a mildly relativistic shock in an accretion disc system and may be representative for a microquasar foreshock region. The electric field amplitudes of the two most unstable quasi-electrostatic waves driven by the two-stream instabilities are shown in Fig.5.

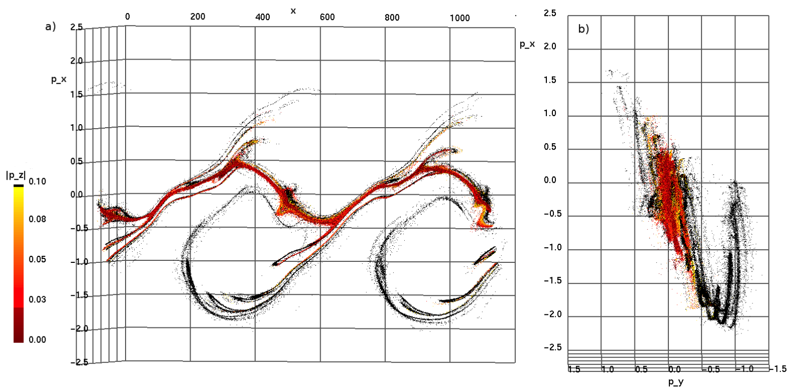


Fig. 6. Electron phase-space trajectories for (x, p_x) in a) and (p_y, p_x) in b). All momentum components have normalised values in the plot and the colour scale is based on the absolute value of p_z. The units of the x-axis in a) is simulation box cell numbers.

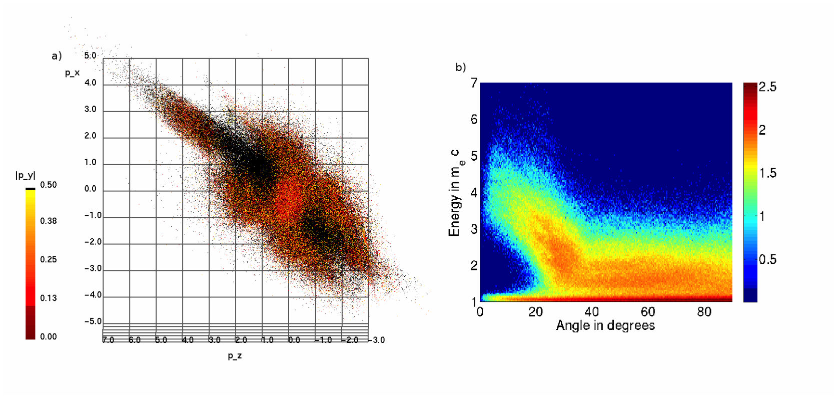


Fig. 7. Plotting the (p_z, p_x) phase-space in a) discloses a beam of electrons gaining field-aligned momentum. The momentum in a) is in normalised units and the colour is based on the absolute value of p_y. A magnetic field aligned energy gain is found in the pitch angle plot b). The colour scale is the log_{10} of the number of simulation electrons and 0 degrees is field-aligned.

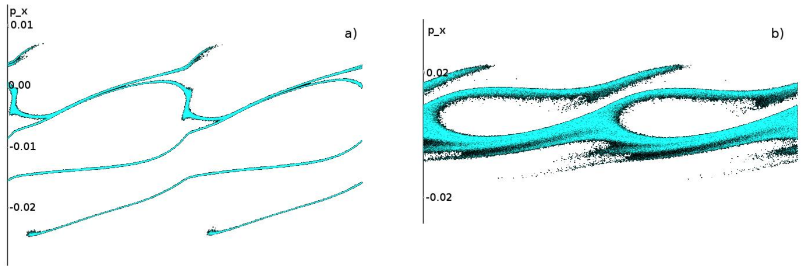


Fig. 8. Proton beam populations after the collapse of the initial two-stream waves. The (x, p_x) phase-space of the colder beam with positive speed is depicted in a) and the hotter beam with negative speed in b). Zero indicates initial beam p_x-momentum and the x-direction of the plots covers the entire simulation box length.

Both wave amplitudes grow exponentially and saturate at approximately the same time. The differences in the curves are due to the initial thermal spread of the beams. In the PIC-simulation the initial wave amplitudes are set by the temperature dependent noise levels. The Fig.6a is a plot of the (x, p_x) phase-space at a time when electrons are trapped in the potential of the quasi-electrostatic wave propagating with negative phase speed in the simulation box. The closed particle trajectories are formed by resonant electrons oscillating in the wave potential. Electron orbits can also be seen starting to form for the positive wave in the upper half of the plot. The Fig.6b displays the (p_y, p_x) phase-space of the electrons and shows how the trapped particles have gained p_y-momentum. Since the plasma is immersed in an oblique magnetic field we also expect to find a gain in p_z. Animating the data over a subperiod of the simulation time a 'beam' of electrons grow in the (p_z, p_x) phase-space. The beam electrons reach a peak-value and then detrap when the electric field weakens, see Fig.5. The final particle distribution is spatially isotropic. The screen capture in Fig.7a is taken from the growth phase of the electron beam in (p_z, p_x) phase-space. The pitch angle plot in Fig.7b indicates a peak energy along the magnetic field direction corresponding to a Lorentz factor of 4-6. This would allow the electrons to escape the gravitational pull from the black hole with speeds close to that observed for microquasar jets. Dilute proton beams are susceptible to non-linear modulations similar to the electron distribution and can produce secondary instabilities, e.g. the frying-pan distribution in Ref.[19]. The proton beams have lost about 1-2% of their initial energy and display well-defined phase-space holes at the end of the simulation, see Fig.8.

Acknowledgements. This work was supported by the German DFG through grant SH 21/1-1. The authors would like to thank the Swedish NSC and HPC2N for providing the computer time.

References

1. Fender, R., Belloni, T.: GRS 1915+105 and the Disc-Jet Coupling in Accreting Black Hole Systems. Annu. Rev. Astrophys. 42, 317–364 (2004)
2. Zensus, J.A.: Parsec-scale jets in extragalactic radio sources. Ann. Rev. Astron. Astrophys. 35, 607–636 (1997)
3. Piran, T.: The physics of gamma-ray bursts. Rev. Mod. Phys. 76, 1143–1210 (2004)
4. Balogh, A., Schwartz, S.J., Bale, S.D., et al.: Cluster at the bow shock: Introduction. Space Sci. Rev. 118, 155–160 (2005)
5. Wang, J.M., Luo, B., Ho, L.C.: The connection between jets, accretion disks, and black hole mass in blazars. Astrophys. J. 615, L9–L12 (2004)
6. Shakura, N.I., Sunyaev, R.A.: Astron. Astrophys. 24, 337–355 (1973)
7. Aoki, S.I., Koide, S., Kudoh, T., Nakayama, K., Shibata, K.: Quasi-Periodic Shock Formations in the System of a Black Hole and an Accretion Disk and Application to Quasi-Periodic Oscillations in Galactic Black Hole Candidates. The Astrophysical Journal 610, 897–912 (2004)
8. McClements, K.G., Dendy, R.O., Bingham, R., Kirk, J.G., Drury, L.O.: Acceleration of cosmic ray electrons by ion-excited waves at quasi-perpendicular shocks MNRAS 291, 241–249 (1997)
9. Woolsey, N.C., Courtois, C., Dendy, R.O.: Laboratory plasma astrophysics simulation experiments using lasers Plasma. Phys. Control. Fusion 46, B397–B405 (2004)
10. Hoshino, M., Mukai, T., Nishida, A., Saito, Y., Yamamoto, T., Kokubun, S.: J. Geomag. Geoelec. 48, 515–523 (1996)
11. Kigure, H., Shibata, K.: Three-dimensional magnetohydrodynamic simulations of jets from accretion disks. Astrophys. J. 634, 879–900 (2005)
12. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. IoP Publishing, Bristol
13. Buneman, O.: Instability, Turbulence, and Conductivity in Current-Carrying Plasma. Phys. Rev. Lett. 1, 8 (1958)

14. O'Neil, T.M., Winfrey, J.H., Malmberg, J.H.: Nonlinear Interaction of a Small Cold Beam and a Plasma. Phys. Fluids 14, 1204 (1971)
15. Sagdeev, R.Z., Shapiro, V.D.: JETP Lett. 17, 279 (1973)
16. Katsouleas, T., Dawson, J.M.: Phys. Rev. Lett. 51, 392 (1983)
17. Dieckmann, M.E., Eliasson, B., Stathopoulos, A., Ynnerman, A.: Connecting shock velocities to electron-injection mechanisms. Phys. Rev. Lett. 92, 65006 (2004)
18. Eliasson, B., Dieckmann, M.E., Shukla, P.K.: Simulation study of surfing acceleration in magnetized space plasmas. New J. Phys. 7, 136 (2005)
19. Dieckmann, M.E., Sircombe, N.J., Parviainen, M., Shukla, P.K., Dendy, R.O.: Phase speed of electrostatic waves: The critical parameter for efficient electron surfing acceleration. Plasma Phys. Control. Fusion 48, 489–508 (2006)
20. Dieckmann, M.E., Eliasson, B., Shukla, P.K.: Self-consistent studies of electron acceleration to ultrarelativistic energies by upper hybrid waves. Astrophys. J. 617, 1361–1370 (2004)
21. Dieckmann, M.E., Shukla, P.K., Parviainen, M., Ynnerman, A.: Numerical simulation and visualization of stochastic and ordered electron motion forced by electrostatic waves in a magnetized plasma. Physics of Plasmas 12, 92902 (2005)
22. Dieckmann, M.E., Eliasson, B., Parviainen, M., Shukla, P.K., Ynnerman, A.: Electron surfing acceleration in oblique magnetic fields. Mon. Not. R. Astron. 367, 865–872 (2006)
23. Eastwood, J.W.: The virtual particle electromagnetic particle-mesh method. Computer Physics Communication 64, 252–255 (1991)
24. Shapiro, V.D.: Ucer, D. Planet. Space Sci. 51, 665–680 (2003)
25. Eliasson, B., Shukla, P.K., Dieckmann, M.E.: Theoretical and simulation studies of relativistic ion holes in astrophysical plasmas New J. Phys. 8, 55 (2006)
26. Woo M. et al.: OpenGL Programming Guide: the official guide to learning OpenGL, version 1.2 3rd ed., Addison Wesley, Reading

Implementing a Particle-Fluid Model of Auroral Electrons

Jörgen Vedin and Kjell Rönnmark

Department of Physics, Umeå University, SE-901 87 Umeå, Sweden
{jorgen.vedin, kjell.ronnmark}@space.umu.se

Abstract. The particle-fluid model of auroral electrons that is presented in [1] is a major step forward within the field of dynamic models of the auroral generation mechanisms. The model is, however, also an example where the implementation of a physical model requires a lot of knowledge from the field of computer science. Therefore, this paper contains a detailed description of the implementation behind the particle-fluid model. We present how the particles are implemented in doubly linked lists, how the fluid equations are solved in a time-efficient algorithm, and how these two parts are coupled into a single framework. We also describe how the code is parallelized with an efficiency of nearly 100%.

1 Introduction

The aurora is created by electrons that are accelerated along the Earth's magnetic field lines before they impinge on the ionosphere and create light through excitation processes. To model the large-scale processes involved in the generation of auroras it is common to describe the electrons as a charged fluid, where the fluid equations are solved self-consistently together with Maxwell's equations. However, as the electrons are accelerated they are also heated in a process that is not properly described by a fluid model. Therefore, we have in [1] introduced a particle-fluid model of the auroral electrons, where the field solver is complemented by a particle pusher. By letting the electric field from the field solver accelerate the particles, and then using the temperature of the particle distribution as a feed-back to the field solver, we obtain a self-consistent description of the auroral electrons.

The physical description of the model is given in [1], and in this paper we concentrate on the details of the implementation. We will describe how the set of equations in the field solver are solved using an implicit algorithm in which the discretized equations are rewritten on a block-tridiagonal form to achieve good performance. We will also discuss how the particle data is implemented in linked lists to obtain a time-efficient algorithm, and how the particles and the field solver are coupled in a parallelized code. First, however, we will in the next section describe the basics of the model.

2 Model

We use a two-dimensional model where the coordinates are z along the Earth's magnetic field and x spanning different latitudes, as can be seen in Fig. 1. The dynamics of auroral electrons is in real life controlled by a generator mechanism located in the tail of the Earth's magnetosphere. To mimic this process we prescribe a generator force in our model. The generator creates an ion current perpendicular to the magnetic field lines. At the flanks of the generator region the perpendicular current is diverted into field-aligned currents connecting the generator to the ionosphere. At the ionosphere these field-aligned currents are closed by a perpendicular ion current. Thus, we have a current circuit where the upward field-aligned current is carried by downgoing electrons that create auroras.

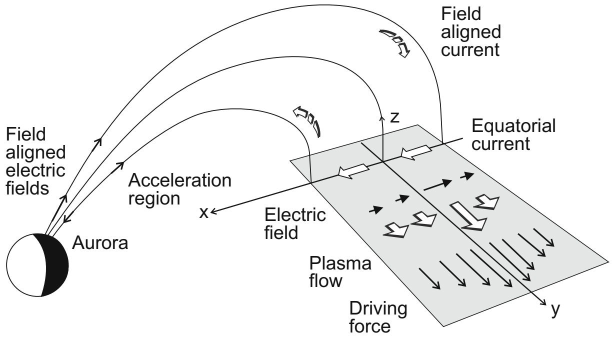


Fig. 1. The geometry of the auroral current circuit and the generator region in the equatorial magnetosphere. The curvature of the magnetic field lines is neglected in our model equations, but the convergence of the magnetic field lines is retained.

2.1 Field Solver

The electron fluid is in our model described by the equations

$$\partial_t E_x = -A^2 \partial_z B_y - (1 - A^2) F \tag{1a}$$

$$\partial_t E_z = \frac{B_z}{B_0} \left(\partial_x B_y + j_z \right) \tag{1b}$$

$$\partial_t B_y = \partial_x E_z - \partial_z E_x \tag{1c}$$

$$\partial_t n = -\partial_z j_z \tag{1d}$$

$$\partial_t j_z = -\frac{m_i}{m} n E_z - \partial_z \left(n T_z + \frac{j_z^2}{n} \right) - n T_\perp \frac{\partial_z B_z}{B_z} \tag{1e}$$

which are the non-zero components of Maxwell's equations together with the equation of continuity and the momentum equation. Here we use simulation variables, where $\mathsf{E_x}$ and $\mathsf{E_z}$ are the electric fields, $\mathsf{B_y}$ is the perpendicular magnetic field, A is the Alfvén velocity, F is the generator force, n is the electron density, $\mathsf{j_z}$ is the field-aligned electron current, while $\mathsf{T_z}$ and $\mathsf{T_\perp}$ are the field-aligned and perpendicular temperatures. $B_z(z)$ is the geomagnetic field and B_0 is the field strength at $z = 0$ in the equatorial plane. The mass ratio in the last equation is the ion mass m_i divided by the electron mass m. For an extensive derivation of these equations and a detailed description of the physics they describe, the reader is referred to [1]. Notice that the ion dynamics is neglected in the present version of the model. This introduces constraints on the generator, and we choose its length and time scales to be larger than the ion gyro radius and the ion gyro period respectively.

The equation system in (1) is underdetermined since the time evolution of the temperatures $\mathsf{T_z}$ and $\mathsf{T_\perp}$ is not included. To close the equation system, we therefore introduce a particle pusher from which we can obtain temperatures that are consistent with the field solver in each time step.

2.2 Particle Pusher

The field solver that solves equations (1) is coupled to a particle pusher according to the cartoon in Fig. 2. This cycle is performed for each time step in the simulation. The particles are accelerated by the electric field from the field

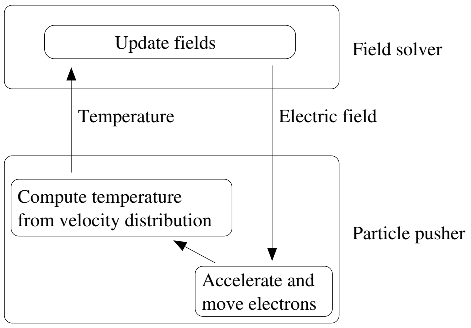

Update fields
Field solver
Temperature
Electric field
Compute temperature from velocity distribution
Particle pusher
Accelerate and move electrons

Fig. 2. Cycle performed for each time step to couple the field solver and the particle pusher

solver, which implies that the particles move consistently with the fluid's evolution. From the velocity distribution of the electrons we can then determine temperatures that can be used in the momentum equation (1e).

The particles move along z with velocity v_z and gyrate the field line with velocity $v_\perp$. Their position and velocity are updated according to

$$d_t z = v_z \ , \tag{2}$$

$$d_t v_z = \frac{-e}{m} E_p - \mu \partial_z B_z \ , \tag{3}$$

where $-e$ is the electron charge, $\mu = m v_\perp^2 / 2B_z$ is the conserved magnetic moment of the particle, and E_p is the electric field that accelerates the particle. The electric field E_p is equal to the field-aligned electric field in the field solver, but with a correction described in [1] to ensure that the density and current of the particles are equal to the density and current of the fluid.

3 Implementation

3.1 Grid

The model is implemented on a two-dimensional grid of mesh size $n_x \times n_z$ illustrated in Fig. 3, where the generator boundary is at $z = 0$ and the ionosphere is on the opposite side of the simulation region. Since the z-coordinate is aligned with the magnetic field and the magnetic field lines are converging as they approach the ionosphere, the grid point separation Δx must decrease towards the ionosphere. The size of the simulation region is $L_z = 55{,}000$ km in the z-direction and $L_x = 4{,}600$ km in the x-direction at the generator boundary. The system length in the x-direction at the ionospheric boundary is merely 200 km. The grid is also chosen to be inhomogeneous for resolving the interesting features of auroral acceleration that take place mainly at altitudes below 10,000 km ($z > 45{,}000$ km) and at the field lines close to $x = 0$. The typical mesh size in our simulations is $n_x \times n_z = 27 \times 100$. The value of n_x is chosen to resolve current filaments with a width of a few kilometers at the ionospheric boundary, while the value of n_z is large enough to get a good resolution of the acceleration

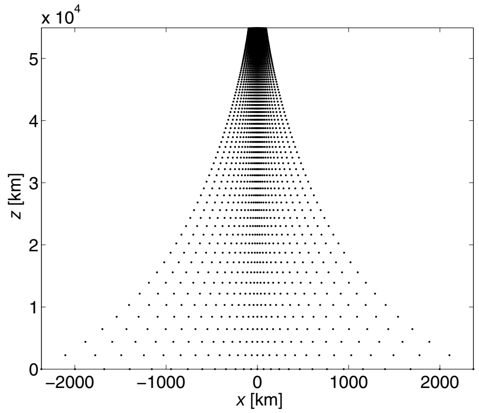


Fig. 3. The inhomogeneous grid on which the model is implemented with the generator boundary at $z = 0$ and the ionosphere on the opposite side of the simulation region

region which has a length of a few thousand kilometers along z at an altitude centered about roughly 6,000 km.

3.2 Field Solver

The set of equations in (1) is solved by a time-centered implicit method, based on factorization of the two-dimensional spatial differential operators. Algorithms of this type are discussed by, for example, [3]. If we for notational convenience collect all the fields in a vector $\mathbf{U} = (\mathsf{E_x}, \mathsf{E_z}, \mathsf{B_y}, \mathsf{n}, \mathsf{j_z})^T$, we can after time discretization sum up (1) in the form

$$\mathbf{U}(t + \Delta t) = \mathbf{U}(t) + \Delta t \left\{ \mathbf{Q}(\mathbf{U}(t + \Delta t/2)) + \mathbf{G} \right\} \ , \tag{4}$$

where $\mathbf{Q}(\mathbf{U})$ represents the $\mathbf{U}$ dependence of the right hand side of (1) and $\mathbf{G}$ represents the inhomogeneous term involving the generator force F. We now linearize $\mathbf{Q}$ in $\mathbf{U}(t + \Delta t)$ by a Taylor expansion:

$$\begin{aligned} &\mathbf{Q}(\mathbf{U}(t + \Delta t/2)) \\ &\approx \mathbf{Q}(\mathbf{U}(t)) + \frac{1}{2} \left[\mathbf{U}(t + \Delta t) - \mathbf{U}(t) \right] \cdot \partial_{\mathbf{U}} \mathbf{Q}(\mathbf{U}(t)) \\ &= \frac{1}{2} \left[\mathbf{U}(t + \Delta t) + \mathbf{U}(t) \right] \cdot \partial_{\mathbf{U}} \mathbf{Q}(\mathbf{U}(t)) \ , \end{aligned} \tag{5}$$

where the second step follows from the homogeneous properties of $\mathbf{Q}$. This results in a linear set of equations:

$$\left[\mathbf{I} - \frac{\Delta t}{2} \partial_{\mathbf{U}} \mathbf{Q} \right] \cdot \mathbf{U}(t + \Delta t) = \left[\mathbf{I} + \frac{\Delta t}{2} \partial_{\mathbf{U}} \mathbf{Q} \right] \cdot \mathbf{U}(t) + \Delta t \, \mathbf{G} \ . \tag{6}$$

The operator $\partial_{\mathbf{U}} \mathbf{Q}$, which contains both ∂_x and ∂_z, is now split into two parts as $\partial_{\mathbf{U}} \mathbf{Q} = \mathbf{X} + \mathbf{Z}$, where $\mathbf{X}$ contains only ∂_x and $\mathbf{Z}$ contains only ∂_z. This decomposition is not unique, and it should be chosen in a way that makes the product $\mathbf{X} \cdot \mathbf{Z}$ as small and simple as possible. Neglecting the term $\Delta t^2/4 \, \mathbf{X} \cdot \mathbf{Z} \cdot [\mathbf{U}(t + \Delta t) - \mathbf{U}(t)]$, which is of third order in Δt, we can factorize the terms in equation (6) as in an alternating direction implicit (ADI) method to find

$$\begin{aligned} &\left[\mathbf{I} - \frac{\Delta t}{2} \mathbf{X} \right] \cdot \left[\mathbf{I} - \frac{\Delta t}{2} \mathbf{Z} \right] \cdot \mathbf{U}(t + \Delta t) = \\ &\left[\mathbf{I} + \frac{\Delta t}{2} \mathbf{X} \right] \cdot \left[\mathbf{I} + \frac{\Delta t}{2} \mathbf{Z} \right] \cdot \mathbf{U}(t) + \Delta t \, \mathbf{G} \ . \end{aligned} \tag{7}$$

Introducing $\mathbf{U}^* = [\mathbf{I} - \Delta t/2 \, \mathbf{Z}] \cdot \mathbf{U}(t + \Delta t)$ as a new variable, we can solve (7) in two steps. First we solve

$$\begin{aligned} &\left[\mathbf{I} - \frac{\Delta t}{2} \mathbf{X} \right] \cdot \mathbf{U}^* = \\ &\left[\mathbf{I} + \frac{\Delta t}{2} \mathbf{X} \right] \cdot \left[\mathbf{I} + \frac{\Delta t}{2} \mathbf{Z} \right] \cdot \mathbf{U}(t) + \Delta t \, \mathbf{G} \end{aligned} \tag{8}$$

for $\mathbf{U}^*$. Then we use

$$\left[\mathbf{I} - \frac{\Delta t}{2}\mathbf{Z}\right] \cdot \mathbf{U}(t + \Delta t) = \mathbf{U}^* \tag{9}$$

to solve for the fields at $t + \Delta t$. When the operators $\mathbf{X}$ and $\mathbf{Z}$ are expressed as centered finite differences, each of these two steps consists of solving a block-tridiagonal set of equations. The number of operations needed for this scales linearly with the mesh size $(n_x \times n_z)$ and hardly requires any extra memory, which makes this algorithm very efficient. The number of operations needed for a direct integration of (6) would typically be proportional to $(n_x \times n_z)^2$.

3.3 Particle Pusher

In this model the particles are used roughly as in a regular Particle-In-Cell (PIC) code, see for example [2]. The input to the particle pusher is the electric field computed in the field solver, and the output from the particle pusher is the temperatures T_z and $\mathsf{T}_\perp$. The electric field and the temperatures are given on the grid points, but the particles can of course be located also in-between two grid points. To handle this, a PIC code utilizes a weight scheme, and in this model we use linear weights. The electric field (and also the magnetic field) used in (3) is determined from the fields at the two grid points that are closest to the particle's position, as can be seen in Fig. 4. At the particle's position z_i, the electric field is given by

$$E(z_i) = \left(\frac{Z_{j+1} - z_i}{\Delta z}\right) E_j + \left(\frac{z_i - Z_j}{\Delta z}\right) E_{j+1} \ , \tag{10}$$

where E_j and E_{j+1} are the electric field values at the grid points Z_j and Z_{j+1}, while Δz is the grid point separation. When the temperatures are computed, the weighting is inverted, and the temperatures at a grid point get a contribution from all particles located in the two grid cells surrounding the grid point, as can be seen in Fig. 5. If T_i is the contribution to the temperature from a particle located at z_i, then

$$T_j = \left(\frac{Z_{j+1} - z_i}{\Delta z}\right) T_i \tag{11}$$

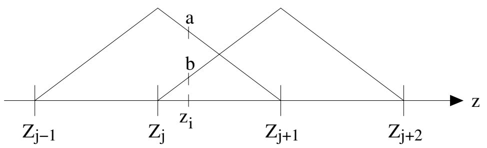


Fig. 4. To determine the electric field at the particle position z_i we use triangular shape functions. The electric field at grid point Z_j is weighted by the value of the shape function in point a and the electric at grid point Z_{j+1} is weighted by the value of the shape function in point b. These two values are then assigned to the electric field $E(z_i)$.

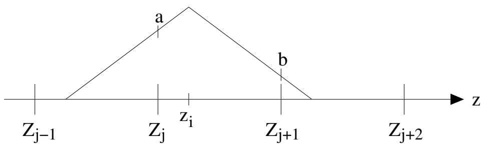


Fig. 5. A particle at position z_i will contribute to the temperatures at grid points Z_j and Z_{j+1}. If T_i is the contribution to the temperature from a particle located at z_i, then T_i will be weighted by the shape function's value at a before it is added to the temperature at Z_j and it will be weighted by the shape function's value at b before it is added to the temperature at Z_{j+1}.

is the part of T_i that is added to the temperature at grid point Z_j. Correspondingly the temperature at Z_{j+1} gets the contribution

$$T_{j+1} = \left(\frac{z_i - Z_j}{\Delta z} \right) T_i \tag{12}$$

from a particle at z_i.

During the simulation, particles that move outside the system boundaries are removed and new particles are injected at a rate determined to maintain a constant density and temperature at the boundaries. To handle this adding and removing of particles in a sufficiently fast manner the particles are implemented as a linked list. Actually, we implement the particles as six linked lists. There are two types of particles, particles of magnetospheric origin that are injected at the generator boundary and particles of ionospheric origin that are injected at the ionospheric boundary. Each type of particles are stored in three separate lists. The particles that are active in the simulation are stored in an inside-list, while the particles that have been removed from the system are stored in a used-list. Furthermore, we have an unused-list that is used as a reservoir for particles that are to be injected. In this way we need not free and allocate memory for each particle that is removed or injected. Used particles are simply transferred from the inside-list to the used-list and particles that are to be injected are transferred from the unused-list to the inside-list. When the unused-list becomes empty, we rehash the particles in the used-list to give them the desired velocity distribution and place them in the unused-list. By letting the initial unused-list hold some extra particles, we can through this procedure avoid the need for any memory allocation during the simulation. As a fallback, the program can allocate more particles if the initial unused-list turned out to be too small.

The implementation with several lists is fast, and the memory overhead is not significant. However, we need to implement the lists as doubly linked lists in order to enable particles to switch lists. This implies some overhead in the memory per particle since each particle need a pointer to both the next particle and the previous particle in the list. The total memory per particle is still only 40 bytes since each particle, apart from the two pointers, only holds three keys: z, v_z, and μ.

3.4 Particle-Fluid Coupling

The electrons are strongly magnetized which means that they are guided by the magnetic field lines and therefore only move along z, although in a gyrating motion. Hence, there are n_x independent particle pushers in the total simulation. In each particle pusher we need at least about 20 million particles to obtain reasonably good statistics when computing the temperatures from the particle distribution. For each x the particles use at least about 0.8 GB of memory, and the entire simulation uses over 20 GB, which indicates that a parallelization is needed. A huge number of particles, of course, leads to long computer times. The ratio between the time spent in a single particle pusher and the time spent in the field solver is roughly 1500 to 1.

Since almost all time is spent in the particle pushers and there are n_x independent pushers, this application is perfectly suited for parallelization on n_x processors. The code, which is written in C, is therefore parallelized using the Message Passing Interface (MPI) [4]. For each time step in the parallelized code, every processor solves (8), then each processor solves (9) and calls the particle pusher with its own value of x. Finally, before time is incremented, the fields in the vector $\mathbf{U}$, the temperatures and some help variables are synchronized on all processors using `MPI_Allgather`. This implies a message passing of about 0.2 MB for each time step.

The simulations are performed on the Sarek Linux Cluster at the High Performance Computing Center North (HPC2N) [5]. This cluster has 384 64-bit AMD Opteron 2.2 GHz CPUs in 192 dual nodes. The network, which is switched to give similar performance between each pair of nodes, has a bandwidth of about 250 MB/s and a latency of a few microseconds. With this bandwidth, the message passing will for each time step last only 10^{-3} s, while the entire time step lasts for roughly 10 s. Thus, the time for communication is negligible. If the load balancing between processors is good this would imply a parallelization efficiency of nearly 100%, and since each processor handles roughly equal amounts of particles we have no reason to expect otherwise. The Sarek cluster also has the advantage that each processor has access to 4 GB of RAM, which implies that we can use the huge number of particles that is needed for good statistics in the temperature calculations.

4 Discussion and Conclusions

According to the physical results presented in [1], the particle-fluid model produces results that are consistent with observations, and the model is a major step forward compared to previous dynamic models of auroral electron acceleration, for example [6], [7], and [8], where the variation in the electron temperatures has been neglected. The implementation is, however, much more complicated compared to implementing a pure fluid model.

The model proves to be very efficient to parallelize on the same number of processors as the number of grid points in the direction perpendicular to the magnetic field. A further parallelization of the code can be accomplished by letting the particles at a certain field line be distributed in several linked lists and update each list on different processors. If the lengths of these lists are approximately equal, which is easily accomplished by always injecting particles into the shortest list, the load balancing of this parallelization would be perfect. Furthermore, the only communication needed in this parallelization is summing the temperatures of all lists into a single temperature along the field line, which makes this parallelization very efficient.

Acknowledgments

This research was conducted using the resources of High Performance Computing Center North (HPC2N), and was supported by the Swedish National Graduate School of Space Technology.

References

1. Vedin, J., Rönnmark, K.: Particle-fluid simulation of the auroral current circuit. J. Geophys. Res. 111, A12201 (2006)
2. Birdsall, C.K., Langdon, A.B.: Plasma physics via computer simulation. McGraw-Hill Book Company, New York (1985)
3. Degrez, G.: Implicit time-dependent methods for inviscid and viscous compressible flows. In: Wendt, J.F. (ed.) Computational Fluid Dynamics, pp. 180–222. Springer, New York (1992)
4. The Message Passing Interface (MPI), `http://www-unix.mcs.anl.gov/mpi/`
5. High Performance Computing Center North, Umeå, Sweden, `http://www.hpc2n.umu.se`
6. Goertz, C.K., Boswell, R.W.: Magnetosphere-ionosphere coupling. J. Geophys. Res. 84, 7239–7246 (1979)
7. Streltsov, A.V., Lotko, W., Johnson, J.R., Cheng, C.Z.: Small-scale, dispersive field line resonances in the hot magnetospheric plasma. J. Geophys. Res. 103, 26559–26572 (1998)
8. Rönnmark, K., Hamrin, M.: Auroral electron acceleration by Alfvén waves and electrostatic fields. J. Geophys. Res. 105, 25333–25344 (2000)

Tools for Parallel Performance Analysis: Minisymposium Abstract

Felix Wolf

Forschungszentrum Jülich GmbH, Germany

Size and complexity of advanced high-end computing systems present difficult challenges to the optimization of application codes running on them. To improve the efficiency and expand the potential of these applications, powerful tools are needed that collect relevant data on code performance and identify the causes of performance problems. This minisymposium will focus on new methods and tools to collect, analyze, and present performance data from executions of high-end computing applications.

Automatic Tuning in Computational Grids[*]

Genaro Costa, Anna Morajko, Tomás Margalef, and Emilio Luque

Computer Architecture and Operating Systems Department,
Universitat Autònoma de Barcelona,
ES-08193 Bellaterra, Spain
genaro.costa@aomail.uab.es,
{anna.morajko, tomas.margalef, emilio.luque}@uab.es

Abstract. The possibility of having available massive computer resources to users opens ideas for the future of interoperability between multiple infrastructure systems. This wide system should be composed of multiple high performance resource clusters and their users should share them to solve big scientific problems. These resources have a dynamic behavior and to reach the expected performance indexes it is necessary to tune the application in an automatic and dynamic way. The MATE environment was designed to tune parallel applications running on a cluster. This paper presents the key ideas for tracking down application process in a wide distributed environment like Computational Grids. We explain how to enable the use of MATE for dynamic application optimizations in such systems.

1 Introduction

With the advent of Internet there was an intensification of sharing information and services availability and nowadays the commodity computers have the computational power of the old mainframes. New systems like computer clusters became more common and accessible to everyone to solve new classes of scientific problems which have high performance requirements. Interconnection of clusters with different administrative domains builds a new classes of systems called Computational Grids to address available resource share [1]. In this context, novel complex applications can be executed in this wide environment spawning processes over more than a single cluster. The control of the application can be done by using a high level scheduler like Condor-G [2] or Community Scheduler Framework (CSF) [1]. The composing processes could be submitted using the Globus Toolkit [1] to fulfill the requirements of security and resource localization. They can be executed in a cluster using certain scheduler environments like Condor [2] or OpenPBS [3].

One common use of Grid resources like processing nodes is to decrease application execution time through parallelization. A user can describe the characteristics of resources required by the application and the Grid system layers drive the application execution to those nodes that can fulfill the expectations.

[*] This work has been supported by the MCyT (Spain) under contract TIN 2004-03388.

Grid Information Services [4] present at these layers indexes different Grid resource properties. Meta-schedulers use these properties to select and provide resources required by the application. The application submission is controlled by the Grid system layers. The user may interact with a meta-scheduler which uses job requirements and the information services to find the appropriate set of available resources required for the execution. These resources may represent complex parallel machines composed by many processors or single machines. Resource availability is dynamic at Grid environment and the meta-scheduler may find different set of resources which fulfill application requirements.

At the machine layer, or Fabric Layer according to the terminology presented by Foster in [5], the application execution is controlled by a batch queue system. The application can use a single cluster to avoid communication overheads or can be distributed over more than one cluster, possibly dealing with high latency message passing communication problems. The application may get different resources for different executions. Therefore, to enable the application tuning process in such an environment, it is necessary to provide some mechanisms that determine which resources are in use by the application.

We developed a Monitoring, Analysis and Tuning Environment (MATE) that runs at clusters under PVM platform [6]. In our environment, an application execution can be guided based on performance models to obtain better execution time or less resource utilization. MATE uses the concept of tuning components called *tunlets* which encapsulate the monitoring, analysis and tuning logic for specific problems. By using MATE, application programs are dynamically instrumented to generate the monitoring data required by the performance models. Based on generated trace data, the *tunlets* may request a tuning action (modification) to be performed on the application. This is a continuous process and help applications to deal with dynamic system behavior.

This paper discusses how to attach our Monitoring, Analysis and Tuning Environment in a Computational Grid in order to overcome the problem of process location by providing two different approaches: enable MATE as a fabric layer service or integrate it as a user application plug-in. Section 2 describes the target Grid environment and characterizes the process tracking in such a system. Section 3 presents related work. Section 4 describes the changes that we have to apply in MATE to adapt it to the Grid environments. Finally, Section 5 presents the conclusions of this work.

2 Computational Grids

According to Foster in [1]. A simplification of Computational Grid could be simply described as a wide collection of interconnected resources distributed under different administrative domains [7]. The Grid is a distributed system architecture generally composed by different levels of interconnect network between its resources. Some resources are accessible by others, while some others are not. Data communication between different resources may have different throughput and latencies. These characteristics make it difficult to locate and even more difficult to solve typical problems of distributed systems like load imbalance, synchronization bottlenecks that are difficult to locate and more difficult to solve.

A common way of use of the computational power of a Grid is to spawn the processes of a massive parallel application within the available node resource [1]. Following the Grid Protocol Stack, the Grid Fabric Layer is where the resources are managed, acquired and controlled. In a typical environment, each processing resource or Compute Element (CE) would be single host or a cluster controlled by a local scheduler [1]. In case of a CE composed of multiple processors like clusters or parallel machines, each machine is called a Compute Host (CH) [1].

An important problem that an application execution may deal in a Grid environment is that such systems generally have a heterogeneous infrastructure and in most cases, a dynamic system configuration. That behavior could lead to performance drawbacks such as load imbalance problems, high latencies caused by improper message fragmentation or inadequate buffer sizes. If the target platform is changed (number of processors, processor speed, network bandwidth, etc.) the required optimizations may be different. Grid brings new challenges in performance analysis and tuning making classical approaches less useful. For example, post-mortem analysis that presumes repeatability may not be used in the grid platform as it is highly dynamic and rarely repeatable. In this sense, a dynamic tuning tool seems to be relevant approach that could adapt applications to runtime changes in the system configuration.

To instrument an application in a Grid environment, a monitoring tool should track down the application process components to gather instrumentation data. This starts to be a problem as CE allocation is indirectly selected through the Grid Protocol Stack.

2.1 Process Tracking

Due to dynamic characteristic of the Grid, application processes may be executed on different resources for different executions. The tracking process should be able to detect application process startup as soon as possible. In that sense, this detection can be done by the application program itself or by the CH where the application program starts running. Following these ideas, to track down process in such an environment, two approaches can be used: (i) bind the application together with a tracking process in a single binary or (ii) have a tracking process running on the CH selected for application running and waiting for the application process.

In the first approach, the idea is to change the application program binary in order to take control over the execution. This is done in a preparation stage before execution submission. In this scenario, the execution of the application process is controlled by the bound tracking process, which may be responsible for gathering instrumentation or doing the dynamic tuning.

In the second approach, the tool is installed previously as CE infrastructure. In that sense, the tracking process waits for application process startup. This can be performed by looking for application startup in a pooling method or monitoring the scheduler runtime events which represent the application startup. The first approach can be done by application developers and users, targeting better application execution time. The second one can be done by site administrators interested in efficiency, since installation requires administrative privileges.

3 Related Work

There are many tools available for performance analysis. A good reference survey covering the available tools for performance analysis is presented on [8] and a more deep Grid integration analysis is done on [9]. Most tools presented on [8] and [9] are system monitoring tools. Most instrumentation tools are bound at compile time such as TAU [10] and SCALEA-G [11]. Both TAU and SCALEA-G also have dynamic instrumentation functionality. TAU redirects its trace data to files and in case of dynamic instrumentation the startup is driven by the user. The application is started by TAU tools through command line. SCALEA-G has Grid integration and export instrumentation facility as middleware services. The Paradyn [12] tool controls the application startup through a daemon running on target machines that are configured to connect to the tool front-end. Monitoring tools that write trace data to local disc on the processing nodes do not have problems to execute on Grid environment because the result files can be scheduled for delivery at the end of execution. In case of online analysis on tools like Paradyn, the dynamic behavior of resource selection presented on Section 2 difficult its use on Grid systems.

There is an initiative to provide control of process startup on batch schedulers by providing a controllable shell with extensive configuration options called Tool Daemon Protocol (TDP), presented on [13]. TDP can be used, for example, to allow Paradyn to track process on different scheduler execution scenarios. The idea behind TDP is to cheat the batch scheduler to give the control to a shell which is responsible for environment configuration and execution control.

Another alternative to integrate automatic performance tuning is to provide a the tool as a program library. Active Harmony [14] uses that approach which provides an API that allows the running application to configure performance parameters, to trigger execution points within its execution and to report metrics by function calls. The Active Harmony philosophy is to treat the application as a 'black-box' controlled by the performance parameters. That approach differs from MATE which runs outside process space and requires some knowledge about the performance models that drives application execution in order to tune.

4 Changes in MATE

MATE consists of an execution environment that permits dynamic tuning of application without the need of code modification, compilation or linkage, based on Dyninst [15]. When an application runs under the control of MATE, its processes are dynamically instrumented in order to generate monitoring information required for automatic performance analysis and allows dynamic process changes as result for tuning actions. MATE structure is explained on [6] and the detailed architecture and implementation is presented on [16].

The two main components of MATE are the Application Controller (AC) and the Analyzer. The AC is the component which interacts with the application process, inserting instrumentation code and doing the dynamic tuning modifications [6,16]. The Analyzer consists of a container of tuning components called *tunlets*. Each *tunlet* can encapsulate the logic of what should be measured, how data can be interpreted by a performance model and what can be changed to achieve better execution time or better resource utilization.

Figure 1 presents the iteration between MATE components. The Analyzer interacts with the *tunlets* that request for application instrumentation. These requests are forwarded to the AC. The AC receives the requests, instruments the application and forwards the trace back to the Analyzer. This trace is dispatched then to the desired *tunlets*. The *tunlet* decides, based on its performance model, what should be changed to tune the application and requests application changes to the Analyzer. These requests are forwarded to the AC and the dynamic changes are made on the application.

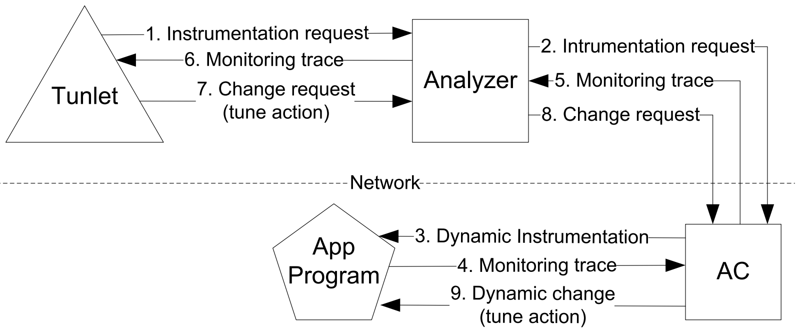

Tunlet
1. Instrumentation request
6. Monitoring trace
7. Change request (tune action)
Analyzer
2. Intrumentation request
5. Monitoring trace
8. Change request
Network
App Program
3. Dynamic Instrumentation
4. Monitoring trace
9. Dynamic change (tune action)
AC

Fig. 1. Sequence of message exchange between MATE components

In the current version of MATE, the application is started under the control of the Analyzer component. The AC binds itself as PVM Tasker, so, when the Analyzer spawns the application the AC is notified to spawn each process. By this sequence MATE has total control of the application and does steering execution of its processes [16]. To work in a Grid environment, the first required change is that, the Analyzer is started independently of the application and the AC is in charge to start the tuning process. The Analyzer needs to steer the execution of application processes to gather runtime information, to execute the analysis process and to perform tune modifications. In order to locate application processes components, MATE may use the application plug-in and system service approaches. Currently, we have developed a prototype implementation on MATE without PVM dependencies in order to test the proposed Grid integration approaches as a proof of concept.

4.1 Application Plug-In Approach

Considering the first approach on how to track down application processes presented on Section 2. To track down the application processes, our tool may be bound to application binary and delivered with it through execution as an Application Plug-in. In that strategy, we decided to generate the composed binary using three applications: a glue code, the current application and the AC binary. That preparation step can be done by the developer or even by the application users before the execution. By doing that composition, at runtime, the first executed code is the one that is loaded on memory. The key idea is to put the glue code as the first application in the composed binary. When the application program runs on the CH, the glue code locates the application and AC code within the composed binary, does some checkup initializations and executes the AC code. When the AC executes, it starts the application child process using the Dyninst library.

The information needed at runtime by the glue code is a fixed size information record appended at the end of the composed binary during the preparation phase. When the glue code runs, it uses this information record to un-pack the AC and application code. After that, the glue code verifies if the environment has Dyninst installed by checking its environment variables. If the environment permits dynamic instrumentation, the glue code executes the AC code; otherwise, it executes the application program without instrumentation. If the AC is started, it creates the application process under its control. The glue code acts as a wrapper process to AC or the application based on target system execution properties.

Figure 2 shows an execution of a composed binary by the Fabric Layer represented by a Batch Scheduler. When AC starts it uses Grid Information Services to locate the Analyzer in charge for the tuning session and establish a communication channel with it. The startup overhead of application plugin approach is small in comparison to a long application execution and that is directed related to the size of application binary and storage performance. In our experiments with executables from 500kb to 1Mb the startup overhead is less than 0.2s which is meaningless considering a long run application execution. The Analyzer location and registration execution times are not critical because they consist of simple sequence of web service calls.

4.2 System Service Approach

Another approach to process tracking is to wait process startup on Grid node resources. Tracking action can be performed by a System Service running on the

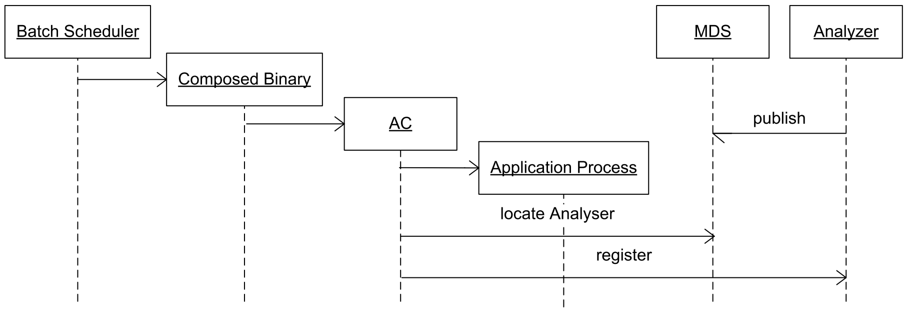


Fig. 2. Sequence of an instrumented application execution using the Application Plug-in Approach

machines selected for application execution. In our context, that corresponds to the AC process which is installed as an operation system daemon by the site administrator. The key idea is to enable the machine with dynamic tuning services that can be used by any application that executes on such machines. With the AC daemon running on each machine of a cluster, this cluster is capable of application tuning. The user should register the application on the Analyzer in order to start the tuning process. By doing this, the Analyzer broadcasts the location request containing application identification with the name and binary checksum hash to all registered AC process using the Grid Information Services. This is necessary due possible process name coincidences.

In response to the Analyzer action, the AC component starts the process of application startup detection for a time period. That can operate in pull mode or push mode. In pull mode, it monitors changes in *proc* file system to detect applications process startup. When an application process name is found, the AC ensures that the binary belongs to the application, attaches to the found process and starts the steering execution. In push mode, the AC instruments the batch scheduler with Dyninst and waits for the callback event generated by *exec* system call. That allows a precise application startup detection and control. We currently support OpenPBS [3] as proof of concept. In each of presented models of execution, the target execution machine should support Dyninst, without that, the dynamic tuning cannot be done.

Figure 3 presents the sequence of execution of an application process that is located using the System Service approach. In both modes of application process startup detection, the AC uses the Grid Information Services, represented in figure 3 as MDS, to locate the Analyzer and establish a communication channel with it.

The startup overhead using the System Service Approach depends of the information propagation within the MDS hierarchy. The resource information in different MDS sites is updated on configured recurrent intervals. The total time of the subscribe action, notification and register messages is fixed and less than 1s on our tests experiments using one MDS site.

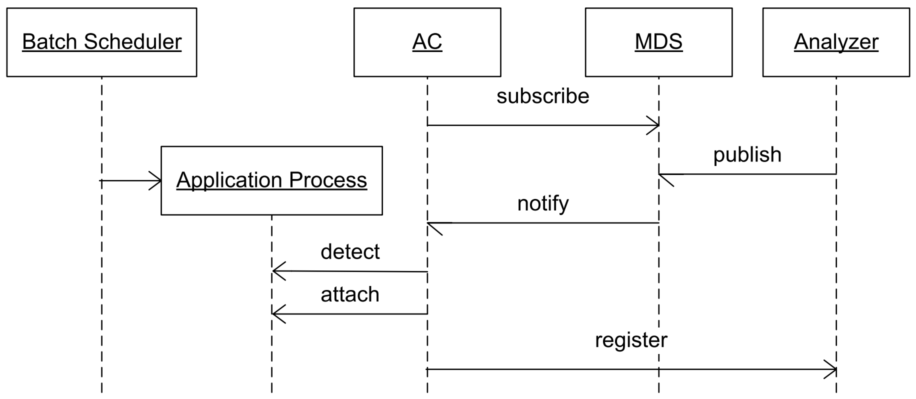


Fig. 3. Sequence of process location using the System Service Approach

5 Conclusions

Performance tuning of Grid application becomes very complicated due to unique Grid characteristics such as time varying resource demands, heterogeneous resources, geographic distribution and network sharing. The dynamic behavior of Grid environment reinforces the need of dynamic tuning tools since the user has less control of the application target execution hosts. MATE can help applications to adapt on different execution configurations. On such an environment process tracking is required as first step to tool integration. The proposed process tracking approaches presented enable MATE for use on Grid environments.

The literature is not clear about the requirements that make a tool enabled for Computational Grids, although, if this tool could be used transparently within the Grid Protocol Stack, it can be used to operate on a Computational Grid in conformance with Grid requirements. We presented two alternatives that enable MATE to be used transparently in such a system within the Fabric Layer of the Grid Protocol Stack.

In the presented Application Plug-in approach, MATE components bound to application binary provides transparent process location on Grid executions without need of infrastructure changes. In the pool and push operation modes of the System Service approach, MATE components are integrated in the execution machines as infrastructure components that require administrative privileges. Both alternatives allow MATE to be used on Computational Grids to perform automatic tuning.

References

1. Foster, I., Kesselman, C.: The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kauffman, San Francisco (2003)
2. Frey, J., Tannenbaum, T., Foster, I., Livny, M., Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids. In: Cluster Computing, vol. 5, pp. 237–246. Springer, Netherlands (2002)

3. Portable Batch System Administrator Guide. Veridian Information Solutions, Inc.: Veridian Systems PBS Products Dept. 2672 Bayshore Parkway, Suite 810 Mountain View, CA 94043 (2000)
4. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: Proceedings. 10th IEEE International Symposium on, High Performance Distributed Computing 2001, pp. 181–194 (2001)
5. Foster, I.T., Kesselman, C., Tuecke, S.: The Anatomy of the Grid - Enabling Scalable Virtual Organizations. International Journal of High Performance Computing 15, 200 (2001)
6. Morajko, A., Morajko, O., Margalef, T., Luque, E.: MATE: Dynamic Performance Tuning Environment. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 98–107. Springer, Heidelberg (2004)
7. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: The Physiology of the Grid, pp. 217–249 (2003)
8. Gerndt, M., Wismuuller, R., Balaton, Z., Gombás, G., Kacsuk, P., Námeth, Z., Podhorszki, N., Truong, H.-L., Fahringer, T., Bubak, M., Laure, E., Margalef, T.: Performance Tools for the Grid: State of the Art and Future. APART White Paper (2004)
9. Zanikolas, S., Sakellariou, R.: A taxonomy of grid monitoring systems. Future Generation Computer Systems 21, 163–188 (2005)
10. Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. International Journal of High Performance Computing Applications 20, 287–311 (2006)
11. Truong, H.-L.: Novel Techniques and Methods for Performance Measurement, Analysis and Monitoring of Cluster and Grid Applications. University of Innsbruck (2005)
12. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn parallel performance measurement tool. Computer 28, 37–46 (1995)
13. Miller, B., Cortes, A., Senar, M.A., Livny, M.: The Tool Dæmon Protocol (TDP). IEEE Computer Society, Washington (2003)
14. Hollingsworth, J.K., Keleher, P.J.: Prediction and adaptation in Active Harmony. Cluster Computing 2, 195–205 (1999)
15. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. Journal of High Performance Computing Applications (2000)
16. Morajko, A.: Dynamic Tuning of Parallel/Distributed Applications. vol. Phd. Universitat Autonoma de Barcelona (2004)

Automated Performance Analysis Using ASL Performance Properties

Karl Fürlinger and Michael Gerndt

Institut für Informatik,
Lehrstuhl für Rechnertechnik und Rechnerorganisation,
Technische Universität München, Boltzmannstr. 3, DE-85748 Garching, Germany
{Karl.Fuerlinger, Michael.Gerndt}@in.tum.de

Abstract. We present our approach for automating the performance analysis of parallel applications based on the idea of ASL performance properties. Our tool *Periscope* automatically searches for inefficiencies specified as ASL properties, leveraging a set of agents distributed over the target machine and arranged in a tree-like hierarchy. Decomposing the analysis using a set of agents allows the analysis process to be performed in a scalable way. If the machine or target application scales in number of nodes or processors used, Periscope similarly scales in number of agents employed.

1 Introduction

Performance analysis of applications can be a complicated and time-consuming task. Tools and methodologies have therefore been developed that try to automate the process of locating inefficiencies in applications and explaining their reasons. With growing size and complexity of applications and high-performance computing systems, automation becomes essential while manual analysis gets increasingly infeasible. In this work we present our approach, and our tool Periscope, for automated performance analysis based on the notion of ASL performance properties.

The rest of this paper is organized as follows: Sect. 2 introduces the concept of ASL performance properties and gives some examples. Sect. 3 then outlines our properties-based performance analysis tool Periscope. In Sect. 4 we present results from conducting a performance analysis with Periscope on several OpenMP benchmarks. We discuss related work in Sect. 5 and conclude in Sect. 6.

2 Performance Properties

Performance properties formalize what can be regarded as a situation of inefficient execution, given a set of performance observations (measurements) for an application. A property's specification is given in a formal language called ASL (Apart Specification Language) and it has three main constituents: *condition* checks the existence of the property, *confidence* quantifies the certainty

```
property ImbalanceInParallelLoop(ParPerf pd) {
  let
     imbal = pd.exitBarT[0]+...+pd.exitBarT[pd.threadC-1];

  condition  : (pd->reg.type==LOOP || pd->reg.type==PARALLEL_LOOP &&
               (imbal > 0);
  confidence : 1.0;
  severity   : imbal / RB(pd.exp);
}
```

Fig. 1. The ASL specification of the `ImbalanceInParallelLoop` property

that the property holds and *severity* denotes how large the negative impact on the performance is.

An example for a property describing imbalance in a parallel loop is shown in Fig. 1. The specification of severity, confidence, and condition can refer to elements of a data model (`ParPerf` in this case), that depend on the particular programming model and the instrumentation and monitoring system available. In Fig. 1, `ParPerf` contains summary data for OpenMP regions, `type` denotes the type of the construct that the region represents, `exitBarT` refers to the summed time spent in the exit barrier of the construct and `threadC` gives the number of threads executing the region. The `ParPerf` structure has a number of other entries and several other properties can be formulated using these entries. A more detailed discussion can be found in [2].

3 The Periscope Tool

Periscope is a tool which automatically searches for performance properties during the execution of a target application (online operation). Performance data is analyzed on the fly by distributed components of Periscope called agents. The agents are arranged in a tree-like hierarchy, as shown in Fig. 2.

On the lowest level, node-level agents (NLAs) are responsible for the detection of properties on a single node (assuming a system that is composed of several SMP nodes). A node-level agent tries to instantiate each property for each possible property context (e.g., source code region) according to the data model as far as it is available up to the point of invocation of the search command. Evaluated and positively identified properties are passed up the agent hierarchy, processed by potentially grouping together similar properties (see below), and reported to the user by the front-end.

The high-level agents aggregate the results of the lower-level agents and pass the detected properties on towards the root of the agent tree (the tool's front-end). The periscope agents use a custom sockets-based binary protocol to communicate with each other, leveraging the ACE framework [3]. The protocol is designed to support platform-independent messaging. That is, the front-end can reside on a different hardware platform as the target application and the agents. The front-end can even run on the users desk-computer or, in fact, anywhere on the internet as long as the required sockets-based communication links can be established. The protocol is based on synchronous messages (the sender awaits the receivers acknowledgement) and can carry data.

User commands are distributed from the front-end to the node-level agents to control (start, stop) the search process and from the node-level agents to the front-end (the detected performance properties). Properties with the same property context and with similar severity and confidence values are combined. For example, the set of properties

```
(1) WaitAtBarrier "opt33" 0.34 1.0 "ssor.f 186"
(2) WaitAtBarrier "opt33" 0.33 1.0 "ssor.f 207"
(3) WaitAtBarrier "opt34" 0.30 1.0 "ssor.f 186"
(4) WaitAtBarrier "opt34" 0.01 1.0 "ssor.f 207"
(5) LockContention "opt34" 0.01 1.0 "0xBF232343"
```

is passed on as

```
WaitAtBarrier "opt33,opt34" 0.32 1.0 "ssor.f 186"
WaitAtBarrier "opt33" 0.33 1.0 "ssor.f 207"
WaitAtBarrier "opt34" 0.01 1.0 "ssor.f 207"
LockContention "opt34" 0.01 1.0 "0xBF232343"
```

That is, properties (1) and (3) are combined into a compound property with averaged severity and confidence values, while (1) and (4) are not combined, because they have different property contexts. Properties (2) and (4) are not combined, because their difference in severity exceeds the predefined threshold of ten percent.

The front-end displays the search results to the user and takes a user's commands in order to display the agent hierarchy graph and to control the search process for performance properties. Currently, a command line interface (CLI) implementation of the Periscope front-end exists, but a front-end with a graphical user interface (GUI) could easily be developed as well.

The Periscope front-end is started on a node with interactive access, and the user has to specify the name of the target application and a set of nodes that can be used to instantiate the agent hierarchy. After invoking the agent hierarchy, the CLI front-end interactively responds to user commands much like an operating system shell.

3.1 Periscope Startup and Usage

The startup process of Periscope proceeds in a hierarchical and distributed way, beginning at the front-end, over the high-level agents, and ending at the node-level agents. The Periscope startup procedure is pictured in Fig. 3. The slanted numbers shown in the figure indicate the sequence of steps that happen during the Periscope tool startup. The startup sequence is:

1. First, the target application is started by relying on external mechanisms (i.e., the application is started without involvement of Periscope). Depending

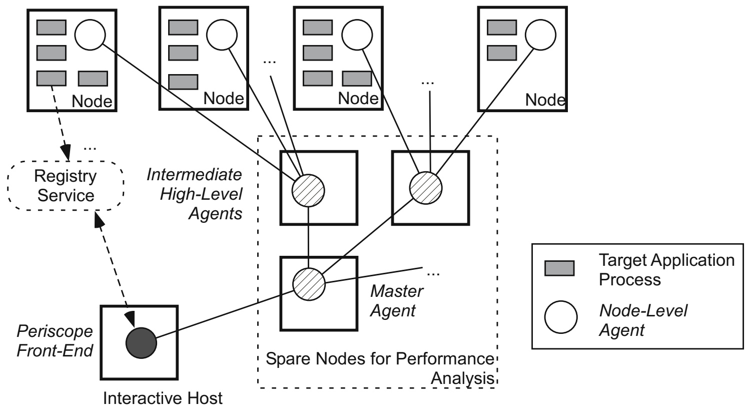


Fig. 2. Periscope agents are arranged in a hierarchy. At the lowest level, node-level agents detect performance properties. Intermediate agents integrate the results of the node-level agents. A single master agent forms the connection to the tool's front-end. A registry service is used by the components to discover each other and to establish communication links.

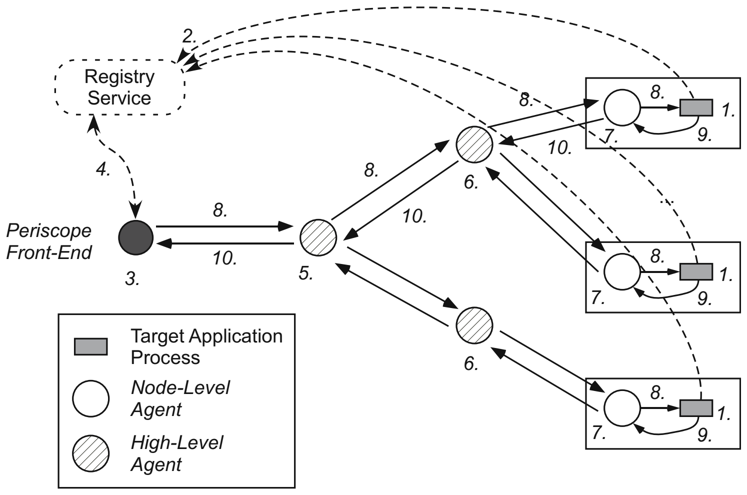


Fig. 3. The startup process of the Periscope tool

on the programming language used and the computing platform on which the program is executed, special launcher commands have to be used in this step. For example, MPI programs are typically started with the help of a utility program that invokes the MPI processes and establishes their communication environment. However, this startup process is not fully standardized and hence a completely portable mechanism to start MPI programs cannot be integrated in a performance tool.

2. When the target application begins its execution, at some point the first call to the Periscope monitoring library occurs by virtue of the (static) instrumentation added to the application's code. This initial call can either be caused by the first encounter of an OpenMP construct or MPI call or by an explicit initialization call added by the programmer.
3. The Periscope performance tool is now instantiated using a multi-level hierarchical startup process. The tool user starts the front-end specifying the name of the target application. Since several applications can be analyzed at the same time (using different instances of the Periscope tool, but sharing the same registry server), the application's name must be given to identify a particular target application. The user must also specify a list of nodes that should be used to host the Periscope agent hierarchy. Typically, this is a subset of the nodes comprising the supercomputer system, which are set aside for performance analysis.
4. Upon start, the front-end registers itself and queries the registry service for entries matching the given target application name. The location of the registry server is either inferred from the command line arguments, from environment variables, or from the Periscope configuration file.
5. The master agent is started by the front-end using a remote program invocation mechanism such as *ssh* or *rsh*. At least one of those programs is typically available on high performance computing systems, since the startup process for MPI application often also relies on such services.

   All arguments for the invocation of the master agent are supplied by command line options. Since the master agent also receives the list of application IDs and hosts from the front-end via command line options, it only needs to contact the registry service to register itself.
6. Each high-level agent that is started is responsible for the startup of the sub-tree for which it is the root. Similar to the master-agent, it takes the application IDs and hosts and separates them into n partitions and invokes sub-agents that are responsible for these partitions. When a partition consists of only one node, a node-level agent is started.
7. Finally, at the lowest level, a node-level agent is invoked on each node on which application processes execute. Each node-level agent (NLA) is provided with the list of IDs specifying the application processes for which the NLA is responsible on the local node using a command line option.

   When the node-level agent is started and ready to process performance data it sends a message to its parent. Upon reception of this message form all its child agents, a high-level agent similarly passes this message on to its parent, until finally the message arrives at the front-end, which hence infers that the entire Periscope agent hierarchy is up and running.
8. The user can now resume the execution of the target application by issuing the `start` command from the front-end. This command is passed down the agent hierarchy until it arrives at the node-level agents, where it causes the resumption of the application processes.
9. Performance data is generated by the Periscope monitoring library and analyzed periodically by the node-level agents. The user can start the search

process for properties at any time by issuing the `check` command from the front-end. This command is propagated through the agent network and causes the node-level agents to instantiate and check for performance properties.

10. Detected performance properties are propagated up the agent hierarchy. Each high-level agent aggregates the property set it receives before passing it on to its parent. This minimizes the amount of data that has to be transferred and increases the comprehensibility of the performance analysis results for the user by combining similar performance properties on different nodes.

    For example, this is useful for master-worker applications where typically two different classes of processes (one or more master processes and several workers) can be distinguished. If the behavior within each class of processors is sufficiently similar (i.e., the same properties are detected for the same program regions with similar severity and confidence values), Periscope can combine the reported properties and thus raise the level of abstraction reported to the user.

4 Evaluation

To evaluate the usability of the Periscope approach we have analyzed inefficiencies of the OpenMP version of the NAS parallel benchmarks (size "C"). The NAS benchmark suite consists of five kernels (EP, MG, CG, FT, and IS) and three simulated CFD applications (LU, BT, and SP). The applications were executed on a 32-CPU SGI Altix system based on Itanium-2 processors with 1.6 GHz and 6MB L3 Cache using a batch system. The number of OpenMP threads was set to eight and the Periscope node-level agent was executed on a separate CPU (i.e., nine CPUs were requested for the batch runs).

The following table shows which performance properties were discovered in each of the NAS benchmarks, the numbers shown in this table count the different instances a particular property was detected.

Property	BT	CG	EP	FT	IS	LU	MG	SP
`ImbalanceAtBarrier`					1	3		
`ImbalanceInParallelLoop`	12	13	1	8	2	9	12	16
`ImbalanceInParallelRegion`	6	9	1		2	8	2	5
`UnparallelizedInSingleRegion`						3		
`UnparallelizedInMasterRegion`	4					13	2	5
`CriticalSectionContention`								1

The `Imbalance-` properties refer to the fact that threads perform a different amount of work (prior to a barrier, in a parallel loop or in a parallel region, respectively). The `Unparallelized-` properties refer to the usage of `single` and `master` constructs resulting in serialization of the execution. `CriticalSection-Contention` captures the situation that several threads contend to enter a critical section, resulting in waiting time for some threads. A number of other properties were tested but not detected by Periscope in the NAS benchmarks.

The following table shows the five most severe inefficiencies discovered in the NAS benchmarks by Periscope. Severity refers to the percentage of total execution time lost due to the inefficiency.

Benchmark	Property	Region	Severity (%)
MG	`ImbalanceInParallelLoop`	`mg.f 608--631`	8.31
FT	`ImbalanceInParallelLoop`	`ft.f 606--625`	6.76
BT	`ImbalanceInParallelLoop`	`rhs.f 177--290`	4.46
BT	`ImbalanceInParallelLoop`	`y_solve.f 40--394`	3.53
BT	`ImbalanceInParallelLoop`	`rhs.f 299--351`	3.47

5 Related Work

Expert [8] is a tool for automated post-mortem performance analysis of C/C++ and Fortran applications. The execution of an instrumented application generates a trace file, which is scanned for patterns of inefficient execution by Expert. The detected inefficiencies are displayed using a viewer with three panes, the first giving the kind of inefficiency and the other two detailing its location (with respect to program resources and machine organization). In contrast to Expert, Periscope is an online tool and performance analysis can be conducted during the execution of the application. The set of bottlenecks covered by the two tools is somewhat similar, with Expert having the advantage of having full trace information available while Periscope's properties currently rely on profiling data only.

Paradyn [4] is an automated online performance analysis tool leveraging dynamic instrumentation techniques. Paradyn looks for performance problems starting with a root hypothesis. In each step of the search process the currently tested hypothesis is then refined along one of the dimensions of the W^3 (why, where, when) search model. In comparison to Periscope, Paradyn has the advantage of using dynamic instrumentation and is thus able to tailor instrumentation overhead to the current hypothesis. Until recently [7] data analysis and the search for bottlenecks was performed centrally at the tool's front-end, a limiting factor for scalability. In contrast, in Periscope the analysis process is performed inherently distributed by the node-level agents.

MATE [5,6] (Monitoring, Analysis and Tuning Environment) is a project that tries to expand beyond automated performance analysis by incorporating methods for automated program *tuning*. MATE relies on dynamic run-time code patching using the Dyninst API [1] to gather performance data and to dynamically adapt the running application to improve its performance.

6 Conclusion

We have presented Periscope, our tool for automated performance analysis based on the idea of capturing situations of inefficient execution in the form of ASL properties. Using Periscope, developers can quickly locate the most severe inefficiencies and discover their reasons. We have tested Periscope on several OpenMP benchmarks and have discovered bottlenecks of up to 8% in overall execution time.

References

1. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. The International Journal of High Performance Computing Applications 14(4), 317–329 (2000)
2. Fürlinger, K., Gerndt, M.: Performance analysis of shared-memory parallel applications using performance properties. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J.J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 595–604. Springer, Heidelberg (2005)
3. Huston, S.D., Johnson, J.C.E, Syyid, U.: The ACE Programmer's Guide. Pearson Education (2003)
4. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn parallel performance measurement tool. IEEE Computer 28(11), 37–46 (1995)
5. Morajko, A., Cèsar, E., Margalef, T., Sorribes, J., Luque, E.: Dynamic performance tuning environment. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, pp. 36–45. Springer, Heidelberg (2001)
6. Morajko, A., Morajko, O., Jorba, J., Margalef, T.: Automatic performance analysis and dynamic tuning of distributed applications. Parallel Processing Letters 13(2), 169–187 (2003)
7. Roth, P.C., Miller, B.P.: The distributed performance consultant and the sub-graph folding algorithm: On-line automated performance diagnosis on thousands of processes. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06), March 2005 (Accepted for Publication)
8. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. In: Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003), pp. 13–22. IEEE Computer Society Press, Los Alamitos (2003)

A Parallel Trace-Data Interface for Scalable Performance Analysis

Markus Geimer[1], Felix Wolf[1,2], Andreas Knüpfer[3],
Bernd Mohr[1], and Brian J. N. Wylie[1]

[1] John von Neumann Institute for Computing (NIC),
Forschungszentrum Jülich, 52425 Jülich, Germany
{m.geimer, f.wolf, b.mohr, b.wylie}@fz-juelich.de
[2] Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany
[3] Center for Information Services and High Performance Computing (ZIH),
Dresden University of Technology, 01062 Dresden, Germany
andreas.knuepfer@tu-dresden.de

Abstract. Automatic trace analysis is an effective method of identifying complex performance phenomena in parallel applications. To simplify the development of complex trace-analysis algorithms, the EARL library interface offers high-level access to individual events contained in a global trace file. However, as the size of parallel systems grows further and the number of processors used by individual applications is continuously raised, the traditional approach of analyzing a single global trace file becomes increasingly constrained by the large number of events. To enable scalable trace analysis, we present a new design of the aforementioned EARL interface that accesses multiple local trace files in parallel while offering means to conveniently exchange events between processes. This article describes the modified view of the trace data as well as related programming abstractions provided by the new PEARL library interface and discusses its application in performance analysis.

1 Introduction

Event tracing is a well-accepted technique for post-mortem performance analysis of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and analyzed afterwards with the help of software tools. For example, graphical trace browsers, such as VAMPIR [1] or PARAVER [2], provide a zoomable time-line display, allowing a manual, fine-grained investigation of parallel performance behavior.

However, regarding the large amounts of data usually generated, automatic off-line trace analyzers, such as EXPERT from the KOJAK toolset [3,4], can provide the user with relevant information more quickly by automatically searching traces for complex patterns of inefficient behavior and quantifying their significance. In addition to usually being faster than a manual analysis performed using the aforementioned trace browsers, this approach is also guaranteed to cover the entire event trace and not to miss any pattern instances.

To simplify the analysis logic incorporated in EXPERT, it has been designed on top of EARL [5], a high-level interface to access individual events from a single global trace file. As opposed to a low-level interface that allows reading individual event records only in a sequential manner, EARL offers random access to individual events. Not to restrict trace-file size, EARL assumes access locality allowing it to buffer the context of recent accesses in main memory while reading events outside this context from file. In addition, to support the identification of pattern constituents, EARL provides a set of abstractions representing execution state information at the time of a given event as well as links between related events, such as corresponding enter and exit events for function instances.

Unfortunately, sequentially analyzing a single and potentially large global trace file does not scale well to applications running on thousands of processors. Even if access locality is exploited as described above, the amount of main memory might not be sufficient to store the current working set of events. In addition, the preceeding step of merging local event trace data generated by individual processes into a global trace file is very time-consuming. Moreover, the amount of trace data might not even fit into a single file, which already suggests to perform the analysis in a more distributed fashion.

To enable scalable trace analysis for modern large-scale systems and applications running on them, we have designed a parallel trace-data interface PEARL as a building block for parallel trace analysis algorithms and tools. In this article, we describe the modified view of the trace data in combination with programming abstractions representing this view. We start our discussion with a review of related work in Section 2, followed by a description of the serial interface in Section 3. In Section 4, we detail the programming abstractions offered by the new parallel interface, before presenting the intended usage as a framework for implementing automatic parallel trace analysis in Section 5. Finally, we conclude the paper and outline some further improvements in Section 6.

2 Related Work

In [6], Wolf et al. review a number of approaches addressing scalable trace analysis. The frame-based SLOG trace-data format [7] supports scalable visualization, whereas dynamic periodicity detection in OpenMP applications [8] prevents redundant performance behavior from being recorded in the first place. Important to our approach has been the distributed trace analysis and visualization tool VAMPIR Server [9], which already provides parallel trace access mechanisms, albeit targeting a "serial" human client in front of a graphical trace browser as opposed to fully automatic and parallel trace analysis. Miller et al. have used a distributed algorithm on multiple local trace data sets [10] to calculate the critical path, which identifies parts of the program responsible for its length of execution.

Unlike common linear storage schemes for event trace data, the tree-based main memory data structure called cCCG [11] allows potentially lossy compression of trace data while observing previously specified deviation bounds. Since we are considering to use cCCGs as an alternate base data structure for our trace-data interface, the parallel programming abstractions described in this paper are designed in such a way that the underlying data structure can be easily changed when the library is compiled.

3 Serial Programming Interface: Earl

EARL (Event Analysis and Recognition Library) is a C++ class library that offers a high-level interface to access event traces of MPI, OpenMP, or SHMEM applications. In the context of EARL, an event trace is stored in a single global trace file that includes events from all processes or threads in chronological order. The user is given random access to individual events allowing the retrieval of distinct events by their index within the chronologically sorted sequence. Loops iterating over the entire trace can be easily implemented by querying the total number of events beforehand.

In addition, EARL provides execution state information at the time of a given event in the form of event sets describing a particular aspect of this state. The state being calculated is either local or global. Local state always refers to a single process or thread, whereas global state may encompass multiple processes or threads. Local state information provided by EARL includes the call stack in form of the enter events of currently active region instances; global state information includes the set of messages currently in transit represented by their respective send events, completed collective operations represented by their respective exit events, and the global call tree derived from the different call stacks, as it evolves over time. Based on this state information, EARL also provides links between related events, which are called pointer attributes. Pointer attributes can also be divided into local and global attributes. There is one local attribute pointing to the enter event of the currently active region instance and allowing traversal of the call stack. Several global attributes support functions, such as locating the send event corresponding to a given receive event, uniquely identifying call paths, traversing the global call tree, or following the ownership history of OpenMP locks between threads.

The intended trace analysis process supported by EARL is a sequential traversal of the event trace from beginning to end. As the analysis progresses, EARL updates the execution-state information and calculates pointer attributes for the most recent event being read, which always point backwards to avoid a costly look-ahead. To make the trace analysis process more efficient, EARL buffers the context of the current event so that events within this context can be directly accessed from main memory. This context includes the last n events (i.e., the history), including the entire related execution-state information.

To avoid re-reading the trace file from the very beginning in cases where an event outside the context is requested, EARL additionally stores the complete execution state information at regular intervals in so-called bookmarks. The history size as well as the bookmark distance can be flexibly configured, however, since these parameters have a significant performance impact with respect to memory consumption and the number of required file accesses and, in addition, these effects are highly application dependent, finding an optimal set of parameters is a non-trivial task. For more details about EARL, the interested reader may refer to the user manual [12].

4 Parallel Access to Trace Data: Pearl

In this section, our new parallel programming interface for accessing event trace data is presented. Before going into the details of the programming abstractions provided, we start with an outline of the overall design and show how it differs from the previous approach described above.

4.1 Design Overview

Similar to EARL, our new parallel trace data access interface, which we call PEARL, is also implemented as a C++ class library. However, we no longer assume a single global trace file, as was the case for the serial interface. Instead, PEARL operates on multiple process-local trace files.

For simplicity, our initial implementation of the PEARL library focuses only on single-threaded MPI-1 applications. However, during the entire design it was taken into account that we plan to extend this approach to alternate parallel programming models, such as OpenMP or MPI-2. Therefore, the PEARL library interface is subdivided into a generic part, which is independent of the programming model used, and an MPI-specific part, implemented by deriving specialized subclasses from the generic interface. Once support for multi-threaded applications has been added, PEARL will access one local trace file per thread.

The benefit behind the approach of using multiple local trace files instead of a single global file is twofold. First, it avoids the time-consuming step of merging the process-local trace files generated by the measurement system into a single global file. And second, it allows us to effectively exploit the distributed memory and parallel processing capabilities available on modern supercomputer systems. As a consequence, the analysis algorithms and tools based on PEARL will usually be parallel applications in their own right with expected scalability improvements compared to serial versions.

The originally envisaged usage model of PEARL assumes a one-to-one mapping between analysis and target-application processes. That is, for every process of the target application, one analysis process responsible for the trace data of this application process is created. However, the PEARL library itself imposes no restrictions on how many traces can be handled by a single analysis process, as long as sufficient system resources (especially memory) are available. It is even possible to implement sequential tools based on PEARL, processing the local traces one after another.

The trace data is exposed to the user through two fundamental classes, namely `GlobalDefs` and `LocalTrace`. Before describing both classes in more detail, the required organization of trace files and how it is established are explained.

4.2 Trace File Organization

To generate trace data suitable for PEARL, we have modified the original KOJAK measurement system. An essential change has been omitting the merge step and storing event data and the definitions of entities referenced by events in separate types of files, which correspond to the classes `GlobalDefs` and `LocalTrace` mentioned above.

Event data stored in trace files refer to static program entities, such as the code regions entered or left. However, to avoid redundancy and save storage space, event records contain only identifiers referencing these entities and the identifiers are defined separately. During measurement each process assigns local identifiers to these entities, and subsequently uses these local identifiers in event records whenever the corresponding entities are referenced.

Immediately after program execution, the measurement system unifies the local definitions and generates a single global definitions file, where each entity is assigned a global identifier. In addition, to allow the conversion of local into global identifiers, the measurement system creates one mapping table per process. In this way, the actual event files, which still contain local identifiers, need not be rewritten and costly I/O can be avoided. This unification step was previously performed using a separate executable, but has recently been fully integrated it into the measurement system itself. A more comprehensive description of these mechanisms can be found in [13].

4.3 Accessing Global Definitions

In the context of the PEARL library interface, information about static program entities that can be shared between all processes or threads of a parallel trace analysis tool is represented by the class `GlobalDefs`. That is, every process has to create a single instance of this class for each experiment it analyzes, which on instantiation reads the corresponding global definitions file generated by the measurement system. All processes read the same file, because they share the same set of global definitions.

The `GlobalDefs` instance provides the user with details regarding the hierarchical structure of the computer system used during trace file generation, consisting of machine, node, process, and thread descriptions as well as their relationships, such as the topological distribution (either from a logical point of view or with respect to the hardware). In addition, it offers ways to query information on groups of locations in the system hierarchy (i.e., threads or processes), which are, for instance, used to specify MPI communicators.

Moreover, the `GlobalDefs` object stores the details of instrumented code regions and call sites, as well as the global call tree of the application that is currently analyzed. This global call tree will be generated by the measurement system at the end of execution of the target application and stored in the global definitions file. Alternatively, the PEARL library provides functionality to reconstruct the global call tree during the trace analysis as an optional preprocessing step. However, this reconstruction can not be performed before the entire event-trace data has been loaded into memory. Note that different from EARL, the call tree is no longer defined in terms of links between individual events (i.e., pointer attributes), but in a separate data structure which simplifies the handling of call-path information.

4.4 Accessing Event Data

In addition to the `GlobalDefs` object, each analysis process (or thread in case of multi-threaded applications) has access to one local event trace represented by an instance of the class `LocalTrace`. Since we assume that the internal in-memory representation of a local trace is smaller than the memory available to a single process on a parallel machine, the entire local trace can be kept in main memory, relaxing the aforementioned limitations resulting from strict forward analysis. In other words, the PEARL library can provide performance-transparent access to individual events plus local execution state information.

To make sure that our assumption of being able to keep the entire trace data in memory is not too restrictive for the future, the `LocalTrace` interface provided by PEARL is designed in such a way that it allows to select between different underlying trace data structures, which can be chosen when the library is compiled. At present, PEARL offers (i) a linear list with full functionality and (ii) rudimental support for cCCG graphs. The latter option will be extended and further investigated. To support very long traces, it would even be possible to add an EARL-like backend using a sliding-window approach and sophisticated buffering mechanisms.

While reading the event trace into memory, the `LocalTrace` object automatically performs two important operations: First, it corrects the timestamps of the individual events using linear interpolation to – at least partially – compensate for unsynchronized clocks. And second, it "globalizes" all identifiers used in the trace file by creating references to the corresponding static program entities provided by the single `GlobalDefs` instance, using the per-process mapping tables mentioned in Section 4.2. That is, the event objects created from the event records provide pointers into the same set of objects. After these on-the-fly transformations, which are completely transparent to the user, the instances of class `LocalTrace` provide a unified view of the event data with respect to timestamps and references to global definition objects. This is especially useful when exchanging event data between processes, as described in Section 4.5.

Individual events of local traces can be accessed through the `Event` class which provides access to all possible event attributes, following the *Composite* design pattern [14]. For navigating through the local trace, this class also exposes *Iterator* semantics available via simple `operator--()` and `operator++()` methods. With respect to the iterator functionality, the two classes `LocalTrace` and `Event` provide an interface that is very similar to that of the C++ Standard Template Library (STL) container classes and their corresponding iterators. For example, the `LocalTrace` class provides `begin()` and `end()` methods returning reasonable `Event` instances. In fact, it is even possible to apply STL algorithms, such as `for_each()` or `count_if()`, to local traces. Not to impose too many restrictions on the underlying data structure, we have refrained from providing event access by index. However, our experience suggests that iterator functionality in combination with traversal of pointer attributes is sufficient to implement complex applications such as a parallel trace analyzer.

In addition to the iterator functionality, instances of the `Event` class also provide pointer attributes for more sophisticated navigation tasks. As a result of the parallel in-memory event storage, pointer attributes can now also point forward, but no longer to remote events. Currently, there are pointer attributes to identify the enter and exit events of the enclosing region instance. These can be used to determine the duration of the communication operation (i.e., region instance) belonging to a given communication event. The return values of these pointer attribute methods are always new `Event` (i.e., iterator) objects that can be subject to further navigation operations. Local call stacks are easily calculated on the fly by traversing the chain of pointer attributes. Another special attribute identifies the call path of an event by providing a pointer into the global call tree. In this way, PEARL applications can easily identify events with equivalent call paths, a feature used to automatically associate bottlenecks with the call paths causing them. However, in contrast to the serial version, all other global states and pointers now have to be established on the application level using the event exchange operations discussed below.

4.5 Exchanging Event Data Between Processes

To facilitate inter-process analysis of communication patterns, PEARL provides means to conveniently exchange one or more events between processes. Remote events received from other processes are represented by a class `RemoteEvent`, which provides a public interface very similar to the class `Event`, but without iterator semantics and pointer attributes, since we do not have full access to the remote event trace.

There are generally two modes of exchanging events: point-to-point and collective. Point-to-point exchange allows a `RemoteEvent` instance to be created with arguments specifying the source process, a communicator, and a message tag. In addition, the corresponding source process has to invoke a send method on the local `Event` object to be transferred.

Moreover, the exchange of multiple events can be accomplished in one batch by first collecting local events in an object of the class `EventSet` on the sender's side and instantiating an object of class `RemoteEventSet` on the receiver's side by supplying message parameters to the constructor. Each event stored in these sets is identified by a numeric identifier which can be used to assign a role to it, for example, to distinguish a particular constituent of a pattern. However, both set classes are able to transparently handle multiple role identifiers for one and the same event to avoid sending its data twice.

Unlike point-to-point communication, the collective event exchange provided by PEARL has the form of a reduction operation that identifies the earliest or latest event (i.e., minimum or maximum operation based on the timestamps) in participating processes' local `EventSets`, and creates a corresponding instance of class `RemoteEvent` for those processes.

5 Example: Scalable Parallel Trace Analysis

The strength of our sequential interface EARL has been the provision of truly parallel abstractions that allows access to higher-level structures, such as messages and collective operations, which requires the ability to match corresponding events across several processes. In the case of our new parallel interface, this is more difficult, since matching those events incurs costly communication. To minimize this communication overhead, the intended usage of PEARL is that of a *replay-based* analysis. This approach has been successfully utilized to implement a parallel trace-based performance analyzer functionally almost equivalent to the MPI-1 part of the aforementioned serial trace analyzer EXPERT.

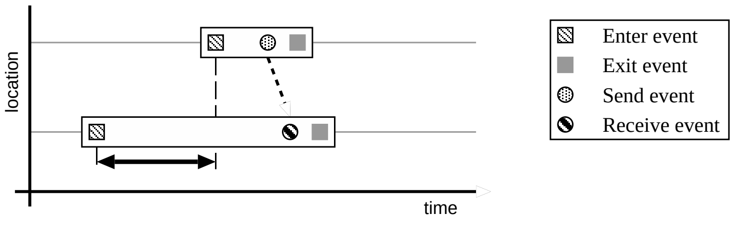
The central idea behind a replay-based analysis is to analyze each communication operation using an operation of similar type, that is, by reconstructing the original communication behavior of the target application currently under investigation. For example, to analyze a message transfer in point-to-point mode, the related event data is also exchanged using a single point-to-point operation.

To accomplish the analysis, the new analyzer is executed on as many CPUs as used by the target application – assuming a one-to-one mapping between analysis and application processes. That is, for every process of the target application, one analysis process responsible for the trace data of this application process is created. Using a one-to-one mapping, the analysis can be efficiently carried out immediately after trace generation as part of the same job. In the future, however, we plan to relax this model and allow a smaller number of analysis processes which might be useful if the analysis should be performed on a different system.

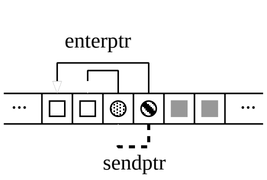
As a first step, each process of the analyzer instantiates a `GlobalDefs` as well as a `LocalTrace` object, thereby loading the corresponding trace data into main memory. Next, they traverse their local traces in parallel using the iterator functionality provided by the `LocalTrace` and `Event` classes and meet at the synchronization points of the target application by replaying the original communication. For this purpose we use the event data exchange abstractions described in Section 4.5. See Figure 1 for an exemplary illustration of how this principle works for the *Late Sender* pattern.

Since the PEARL library provides performance-transparent access to all events of a local trace, the analysis is no longer restricted to a pure forward analysis. That is, PEARL offers the possibility to not only exchange the data of a communication event (and potentially also the enter event of the surrounding function call accessible via a pointer attribute), but also the data of the corresponding exit event or any other event occurring in the "future". Our current prototype implementation of the parallel analyzer does not yet take advantage of this fact, but this is likely to change in subsequent versions.

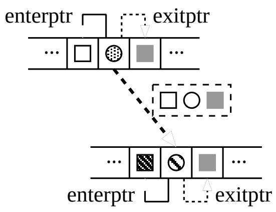
Using the replay-based analysis approach implemented with PEARL, we were able to analyze execution traces of a parallel-tree application called PEPC-B running on 1,024 CPUs and the ASCI benchmark SMG2000 running on up to 16,384 CPUs. Thereby, the largest data set consisted of more than 40 billion events, which amounted to approximately 230 GBytes of disk space. By contrast,



(a) Illustration of the *Late Sender* pattern



(b) Sequential trace



(c) Distributed trace

Fig. 1. Comparison of old and new approaches: (a) Illustration of the *Late Sender* pattern. (b) In a sequential EARL-based analyzer, the detection of the *Late Sender* pattern is triggered by a receive event. All other relevant events are found through pointer attributes. (c) In the parallel, i.e., PEARL-based approach, an `EventSet` is created whenever a send event is found by an analysis process. This send event and the associated enter and exit events are added to the set. This set is then sent to the corresponding receiver, which in turn instantiates a `RemoteEventSet` when the corresponding receive event is reached. Now the receiving process has access to all relevant constituents and can therefore verify the existence of the *Late Sender* pattern.

sequentially analyzing such a huge amount of data using an EARL-based analyzer was impractical. A more elaborate discussion of the parallel analyzer and the experimental results can be found in [15].

6 Conclusion and Future Work

This paper presented the design of a new parallel trace data access library called PEARL. Instead of using a single and potentially large global trace file, as was the case for our previous serial approach, the new library operates on multiple process-local trace files. This allows effective exploitation of the distributed memory and processing capabilities of modern supercomputing systems for parallel trace-analysis algorithms and tools.

The library offers basic functionality to easily access event-trace data, following well-known design principles. Because of the distributed data storage scheme, the entire event trace is held in main memory, thus yielding performance-transparent access to individual events. In addition, the interface provides a global view of static program entities referenced by the events, such as code regions or communicators, and of the call tree. Compared to its serial predecessor EARL, PEARL's storage scheme and usage model allows pointer attributes to point forward in time, which gives tool builders more flexibility in recognizing event patterns. On the other hand, PEARL no longer offers global abstractions, such as pointer attributes pointing to other processes or execution state. These have been replaced by mechanisms to conveniently exchange events between processes that, when used in conjunction with the concept of parallel reply, can provide almost equivalent but significantly more scalable trace-analysis functionality. Moreover, the basic design of the library offers the option of selecting between different trace data structures when the library is compiled. In this context, we are planning to fully implement support for the tree-based cCCG data structure and to explore its use for automatic performance analysis.

As an example of a parallel trace analysis application based on the PEARL library, we have outlined some details of our current prototype implementation of a scalable performance analyzer. The analyzer and the underlying PEARL library at present focus only on MPI-1 applications, however, we intend to extend them to support other parallel programming paradigms, such as MPI-2 and OpenMP. Finally, we plan to exploit the advantages of efficient event access to implement more sophisticated patterns that are impractical to recognize within the serial EARL framework.

References

1. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12, 69–80 (1996)
2. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP: A parallel program development environment. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 665–674. Springer, Heidelberg (1996)
3. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49, 421–439 (2003)
4. Wolf, F., Mohr, B., Dongarra, J., Moore, S.: Efficient pattern search in large traces through successive refinement. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 47–54. Springer, Heidelberg (2004)
5. Wolf, F., Mohr, B.: EARL - A programmable and extensible toolkit for analyzing event traces of message passing programs. In: Sloot, P.M.A., Hoekstra, A.G., Bubak, M., Hertzberger, B. (eds.) High-Performance Computing and Networking. LNCS, vol. 1593, pp. 503–512. Springer, Heidelberg (1999)
6. Wolf, F., Freitag, F., Mohr, B., Moore, S., Wylie, B.J.N.: Large event traces in parallel performance analysis. In: Proc. 8th Workshop on Parallel Systems and Algorithms. LNI, Gesellschaft für Informatik, pp. 264–273 (2006)
7. Wu, C.E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., Gropp, W.: From trace generation to visualization: A performance framework for distributed parallel systems. In: SC2000, IEEE Computer Society Press, Los Alamitos (2000)
8. Freitag, F., Caubet, J., Labarta, J.: On the scalability of tracing mechanisms. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 97–104. Springer, Heidelberg (2002)

9. Brunst, H., Nagel, W.E.: Scalable performance analysis of parallel systems: Concepts and experiences. In: Parallel Computing: Software Technology, Algorithms, Architectures and Applications, vol. 13, pp. 737–744. Elsevier, Amsterdam (2004)
10. Miller, B.P., Clark, M., Hollingsworth, J.K., Kierstead, S., Lim, S.S., Torzewski, T.: IPS-2: The second generation of a parallel program measurement system. IEEE Transactions on Parallel and Distributed Systems 1, 206–217 (1990)
11. Knüpfer, A., Nagel, W.E.: Construction and compression of complete call graphs for post-mortem program trace analysis. In: Proc. Int'l Conf. on Parallel Processing, pp. 165–172. IEEE Computer Society Press, Los Alamitos (2005)
12. Wolf, F.: EARL – API documentation. Technical Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory (2004)
13. Wylie, B.J.N., Wolf, F., Mohr, B., Geimer, M.: Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. In: Proc. Workshop on State-of-the-Art in Scientific and Parallel Computing (2006)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, London (1995)
15. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)

Search of Performance Inefficiencies in Message Passing Applications with KappaPI 2 Tool

Josep Jorba[1], Tomás Margalef[2], and Emilio Luque[2]

[1] Universitat Oberta de Catalunya (UOC),
Estudis d'Informatica, Multimedia i Telecomunicacio, Rambla del Poblenou 156,
ES-08018 Barcelona, Spain
jjorbae@uoc.edu
[2] Universitat Autonoma de Barcelona (UAB),
Computer Architecture and Operating Systems Department,
ES-08193 Bellaterra, Spain
{tomas.margalef, emilio.luque}@uab.es

Abstract. Performance is a crucial issue of parallel/distributed applications. One kind of useful tools, in this context, are the automatic performance analysis tools, that help developers in some of the phases of the performance tuning process. KappaPI 2 is an automatic performance tool, with an open extensible knowledge base about typical inefficiencies in message passing applications, and it is able to detect and analyze these inefficiencies, and then make suggestions to the developer about how to improve their application behavior.

1 Introduction

Designers and developers of parallel/distributed applications, expect that their applications reach high performance indexes to meet the expectations of HPC. In this context, performance analysis is a crucial issue.

However, there is a lack of useful tools, and the most popular approach to carry out the performance analysis is the use of visualization tools [1,2] to show several indexes obtained from the execution of the application. The analysis of these views is a difficult and time consuming task that requires a high degree of expertise from the developer.

To overcome this situation, more user-friendly tools are needed. Such tools should provide a step ahead from the visualization techniques. To fulfill these requirements, some automatic performance analysis tools have been developed, like Scalea [4] and Expert [3]. These tools take data from the execution of the application, in form of profiling or tracing data, and try to detect performance bottlenecks in the application. To identify these bottlenecks, these tools uses certain performance property specification, for example, derived from the APART specification Language (ASL) [5].

KappaPI 2 is an automatic performance analysis tool that extends these ideas, enhancing the use of current knowledge of performance inefficiencies in the message passing environments, and providing support in the performance analysis process in form of recommendations directed to the developer. The application developer can use the recommendations provided by the tool to improve the performance of the applications.

The rest of the paper is organised as follows. Section 2 shows basic concepts and goals inside the general architecture in the KappaPI 2 approach. Section 3 surveys related work on automatic performance tools. Section 4 summarizes typical KappaPI 2 tool operation, showing some internals (as we devoloped in KappaPI 2 tool prototype) of the main mechanisms about detection, classification and suggestions, related to the performance bottlenecks. Section 5 presents some experimental results obtained by the tool prototype. Finally, section 6 summarizes our conclusions.

2 KappaPI 2

In KappaPI 2 [9] the main goal is to provide useful hints to the application developer, to enhance the performance of their applications. The tool uses a series of specifications or performance knowledge as input data. This knowledge represents a set of parallel performance bottlenecks that can be found in the execution of parallel/distributed applications on message passing environments (MPI and PVM in our case).

The main goal of the tool architecture is to provide the mechanisms to support an automatic performance analysis tool that has the following features:

– Performance knowledge specification: Independent specification mechanisms to introduce new performance inefficiencies. Facilitate the introduction of new performance bottlenecks in the tool's knowledge base, without hard-coding this knowledge in the tool.
– Independence from background message passing system: The tool builds abstract entities that are independent from the particular trace file format or the message passing primitives.
– The performance inefficiency detection engine must read the performance knowledge specification and classify the inefficiencies found in the trace.
– Relate inefficiencies to the source code of the application: the source code must be examined to determine why the inefficiency appears.

In KappaPI 2 (figure 1), the first step is to execute the application under the control of a tracing tool (for PVM or MPI environments) that captures all the events related to the message passing primitives that occur while running the application. Our tool takes this trace (obtained from the post-mortem application execution) and the performance inefficiency knowledge base as inputs, to detect the performance bottleneck patterns defined from a structural point of view. In our case, a performance problem pattern is defined as a well know inefficiency structure on a message passing application.

After this pattern detection process, the tool sorts the found performance bottlenecks according to certain indexes of importance, to classify the bottlenecks in a rank. The most important found are taken to analyse their root causes of appearance. After it is carried out a bottleneck cause analysis, based on well known

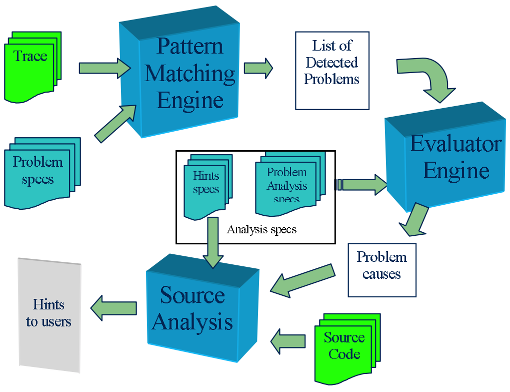


Fig. 1. Module architecture of the KappaPI 2

bottleneck use cases, and the application source code analysis. And finally the tool provides a set of recommendations (hints) to the user, indicating how to modify the source code of his application to overcome the detected bottlenecks.

3 Related Work

Several automatic performance analysis tools can be related to KappaPI 2, including the first version of KappaPI [7], Scalea [4], Expert [3].

In the first version of KappaPI, detection of performance bottlenecks focused on idle intervals affecting the largest number of processes. Processor efficiency was used to measure execution quality, and idle processor intervals represented performance inefficiencies. The knowledge about the performance inefficiencies was a closed, hard coded set of bottlenecks, and no mechanisms were provided to add new bottleneck specifications. Similar limitation also affects root cause analysis. In these terms, respect to original KappaPI, one of the main important point is the introduction of one open database of knowledge about performance problems, without recoding each problem, in a hard coded way, by means of using independent specification languages (based in standards like XML) defining each problem in form of structural pattern of events involved in the problem, and the constraints related to where and when the problem appears.

Scalea [4] uses an interface called JavaPSL API to specify the performance properties [5], using a Java syntax in a form of classes for each problem. The user can specify new properties (by adding new Java classes) without changing the implementation of the tool's search phase.

In Expert [3,8], the specification of the performance properties are realised using script languages based on internal APIs (for trace manipulation, and information retrieval related to the events). Expert tries to answer the question of where the application spends time. It summarizes the indexes of each problem found and accumulate their times to compare its impact to the total application execution time. Main differences between KappaPI 2 and Expert are: a) Bottleneck specification from a structural point of view, meanwhile Expert uses bottleneck detection/matching programming (in different base programming languages). b) Expert specification is based on some trace API, that the user needs to know in order to specify the bottleneck property; in Kappa PI 2 the use of the trace is only internal. KappaPI 2 also offers abstraction mechanisms from trace formats and environments: we can use different tracers in different environments (PVM, MPI). c) Expert does not offer direct techniques for source code analysis, or any kind of recommendations to help the developer to improve the application. d) In Kappa PI 2 an additional level of specification is added for bottleneck analysis causes. The user can provide knowledge about bottlenecks and their analysis process.

4 KappaPI 2 Operation

First of all, it is necessary to execute the application, with a tracer tool (adapted to MPI or PVM), that collects all the events related to the message passing primitives that occur during the application execution. The stored trace is used as input for the detection phase, based on the matching of the bottleneck structural specification between the specified events and their presence in the application trace.

At starting the detection phase, KappaPI 2 needs to read the knowledge about performance inefficiencies, by means of structural bottleneck specification. Each bottleneck in the knowledge base is specified as a structural pattern, defined with:

- Root event: The main event as root of the bottleneck.
- Event instances: Which other events appear.
- Constraints: When and where, constraints of relating events.
- Computations: Small computations required to evaluate some rank indexes.

For example, we can see an example of one classical bottleneck, the Blocked Sender [9], in the figure 2:

We can observe different events from the trace, each line in this Gantt chart, has the events related to one of the tasks, three tasks are involved in this bottleneck. In a task the execution of each primitive generates two events: one for the entry point (in time) for the primitive call, and other exit point of the primitive call.

And in the resulting specification, an XML based specification, with structure as commented previously, we need to consider event instances involved, temporal order constraints, and calculations to realize.

We need to include the constraints related to time, which event appear after or before another, and place constraints relating events to their tasks, and also computations needed to evaluate the ranking of the bottleneck, in this case an idle time waiting for the initiation of a second point to point operation. In the specification, we see these items:

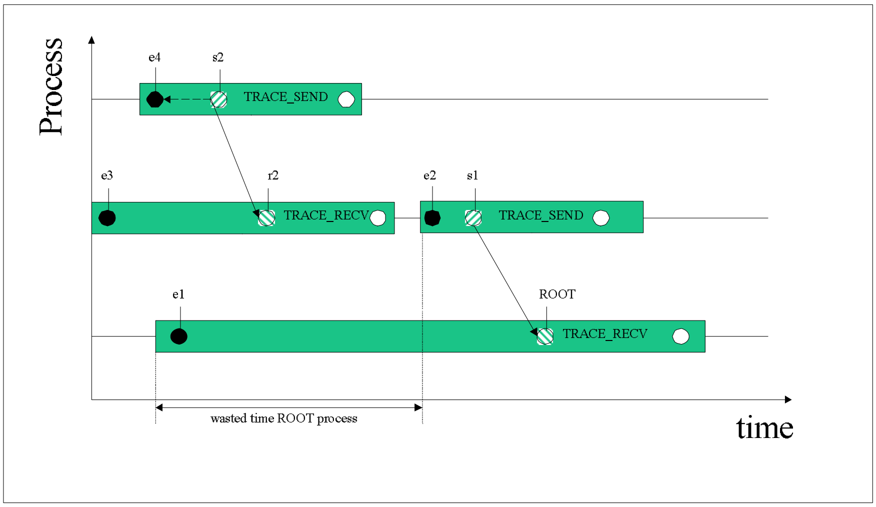


Fig. 2. Blocked Sender: A third task is blocked waiting for a communication, not started

```
Part of Blocked Sender specification
<PATTERN Name="Blocked Sender">
 <ROOTTYPE>RECV</ROOTTYPE>
 <INSTANCES>
  <EVENT NAME="S1" TYPE="SEND" TO="ROOT"></EVENT>
  <EVENT NAME="S2" TYPE="SEND" TO="R2"></EVENT>
  <EVENT NAME="R2" TYPE="RECV" FROM="S2"></EVENT>
  ...
 </INSTANCES>
 <CONSTRAINT>
  ...
  <COND TYPE=">" OP1="E2.stamp" OP2="E1.stamp"></COND>
  <COND TYPE=">" OP1="E2.stamp" OP2="E3.stamp"></COND>
  <COND TYPE="=" OP1="E3.taskId" OP2="E2.taskId"></COND>
 </CONSTRAINT>
 <EXPORT>
  <COMPUTE NAME="idle_time" AS="-"
    OP1="E2.stamp" OP2="E1.stamp"></COMPUTE>
 </EXPORT>
</PATTERN>
```

In our tool knowledge catalog we have specifications about different classical performance bottlenecks, some related to point to point communications, collective communications and synchronisation. The performance problems in this specification catalog include:

– Late Sender (LS)
– Late Receiver (LR)

- Blocked Sender (BS): A third task receive operation cannot be initiated because second task has a previous not initiated communication from a first task.
- Multiple Output (MO): Multiple messages from one task to the others.
- Wrong Order (WO): Of the messages between two tasks.
- Early root in 1 to N collectives (E1N).
- Delayed root in N to 1 collectives (DN1).
- Early tasks in N to N collectives (ENN).
- Block at Barrier (BB).

And also the catalog provides specifications that help the detection and optimisation of particular structures of parallel applications, like master-worker [6], and pipelines. Some of these performance problems are described in previous works on KappaPI [6,7]. Others can be found in related work in performance analysis tools [5,8], as for example the works about ASL language in the APART project [5], and the concept of compound event, as form to express a structural specification of a performance bottleneck with related constraints.

In the next internal process in the tool, we do the knowledge reading process (obtaining the bottleneck catalogue), which builds a tree with the common roots and paths between the bottlenecks, making a decision tree, which classifies the knowledge. During detection phase the decision tree was walked for problem matching. As we can see in the figure 3:

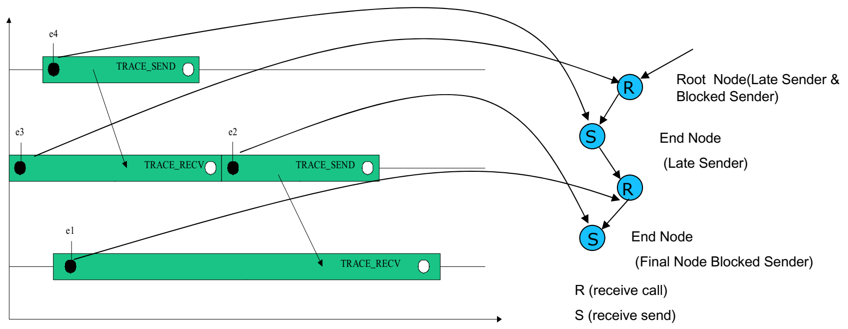


Fig. 3. Detection phase using a walk in the decision tree

In this example of a decision tree, it only represents two bottlenecks: later sender and blocked sender. The blocked sender is that there is a receiver that is blocked by a sender. And his sender is blocked by another receiver which is blocked by a sender. And a later sender is a receiver blocked by a sender. As we can see, two different bottlenecks can have common roots.

In this case in the trace showed, the two first nodes detect a Late sender, if appears a receive event connected with the sender, as in the example trace, a Blocked sender problem is found.

One walked path in the tree, define a bottleneck from the root to end node, but two problems can share steps in a path, if the bottlenecks can have common partial paths.

Considering this, one end node of a problem isnt necessary a final bottleneck detected, we need to maintain it as a possible path to extend to another bottleneck, if more related events appears to continue with the bottleneck which shares path. In some terms, the bottleneck can be seen has a specialization (or composition) of the first.

Once a bottleneck is detected, its information is captured and stored as a match in a table of inefficiencies, which is used as classification scheme based on indexes of presence and importance of the bottleneck. When the detection phase has finished, a table of main problems is provided.

These bottlenecks, with information related to source code (where the events are produced, in form: which calls, in which file and line number in source code), are analysed to determine the causes of their occurrence.

In this point a second level of specification (based also in XML standard) is used. Each bottleneck has a set of cases to test (like a series of use cases of the bottleneck), referred to a different kinds (or instances) about how the bottleneck can be found. It provides information about the possible cases of the bottlenecks, based on a description of preconditions to test for determining which exact case of the match of the problem have been found.

These condition tests require to analyse some source code, for example to determine data dependences, or information about parameters in use, or if it is feasible to make some code transformations. And in that case these evaluated tests can be used to determine what cause (or what initial conditions) provoke the bottleneck, and suggest after, in a hint form, how the user can actuate to solve (or minimize) the bottleneck.

For example, in a simple case: In a isolated Late Sender bottleneck, you can test if it's possible to do the primitive message passing call for the send operation before in time in the sender task, or test if the receiver can do the receive operation after, obtaining a better synchronization scheme (minimizing the time delay). The tests, in this case, are about the source code points of send and receive primitive calls, and the data dependence around the code blocks involved, and test possible code reorganizations, to improve a better point to point communication.

One simple test for Late Sender (simplified)

```
<ANALYSIS name="Late Sender">
  <points>
        <place name="recv" event_info="E1">
        <place name="send" event_info="E2">
  </points>
  <test type="CODE_UP" place_name="send">
    <message>Hint: Receive delays</message>
    <hint_true>
      Code in send call can be located before
      this place, no data dependences
    </hint_true>
    <hint_false>
      Try to refactor this code, can be located
      before this place
    </hint_false>
  </test>
  ...
</ANALYSIS>
```

The source analysis process is done by means of structural source code representation, based in a XML like representation (developed in APART project works [10]). We can analyze some blocks of code, and determine their structure, or see information about symbols, or detect the context of a message passing call (in a loop, conditional...). This representation is used to get information of a particular point of source code (point of interest in case analysis). Afterwards, some quick parsers (for small dedicated tasks) detect some of the conditions of interest for the use case evaluated.

In that sense, we can see that KappaPI 2 works with source code between two visions:

– Uses a representation of source code (based in APART SIR) in the XML mark-up language, for block code structure inspection, and to locate caller-calls positions.
– Uses a small API (quick parsers API) to test some basic conditions around lines of source code, like: data dependences, reorganizations of code (e.g. if a portion of code is moveable up or down a number of lines), test parameters of a primitive call, etc...

Finally, once a case is found the tool provides a recommended action (hint) to be done on the source code to overcome or eliminate the found bottlenecks. One hint template is associated with each use case, and the hint template is filled by the information recollected in all previous tool phases about the bottleneck (related events, recollected metrics, indexes calculated, etc...). When one use case is validated, the user finally obtains the hints, in a form of recommendations about how he can improve his source code to avoid or minimize the main performance bottlenecks detected.

We can see an example of recommendation for a simple case of a detected of one Later sender bottleneck:

```
Hint template for a Late Sender
[Late sender] situation found in tasks [0] and [1],
the sender primitive call can be initiated before
(no data dependences)
the current call is on line [35] of the file [ats_ls.c].
```

Where info in brackets is filled by the tool, starting with the hint template, and using the data recollected in the use case of analysis of the Late Sender.

5 Experimentation

We have carried out a series of study cases, with different kind of tests, for the validation of the architectural phases of the tool: detection, classification, cause analysis (with extra source code analysis), and final recommendations for the developer.

Various tests are made using standard synthetic code [9], some performance benchmarks, and some real scientific applications, in terms of validation of analysis phases of the tool, and final results of improvement of the application performance.

With real applications, we try to show a complete cycle of analysis, and some performance improvements obtained by suggestions emitted by the performance tool.

For example with the Xfire PVM application, which is a physical simulation system, for study the forest fire propagation [11]. Basically this is an intensive computing application (with a master-worker model), which models the propagation of a forest fire considering the weather and vegetation parameters, and proceeds with a time discrete simulation; where in each time step it evaluates the next position of the forest fire. Running this application within a cluster of

Workers	Bottlenecks	Main
1	6 LS	1 LS
2	2 BS, 7 LS	3 LS
4	4 BS, 10 LS	3 LS
8	9 BS, 20 LS	2 LS
16	74 BS, 45 LS	10 LS

Fig. 4. Main bottlenecks found in Xfire

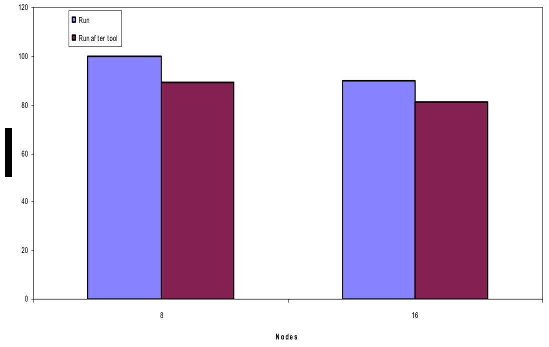


Fig. 5. Performance improvement in Xfire

16 nodes, with different configurations, we have obtained by KappaPI 2 some data about bottlenecks found:

We observe an important bottleneck increase in the 8 and 16 nodes cases. The tool in the hint mentions the detection of the main bottlenecks, LS and one BS, with two tasks involved, which are in the same node (from the detection of structures [6], the tool knows that one is a master and the other one slave). The tool suggest as a hint to exam the code of tasks creation, pointing out the line of code associated, in this case one pvm_spawn primitive call. The proposed modification is to make explicit the mapping of the tasks into physical nodes, touching the dynamic creation of tasks (in pvm_spawn).

Running the new modified application, in the problematic cases (8 and 16 nodes) makes an improvement of performance of 10% on the global application execution time.

6 Conclusions

We discussed an automatic performance analysis architecture oriented toward the end user to avoid the high degree of expertise needed to improve message-passing applications. The performance analysis knowledge involved in the current tool prototype, is open to the introduction of new models for performance bottlenecks. It is able to make useful suggestions about the source code, to improve the application execution time and therefore avoid the performance problems.

Past and present experimentation corroborates that the detection of parallel performance problems based on structural patterns, the use of an open tool to incorporate knowledge in a highly flexible form, and the generation of suggestions directed to the developer is a feasible approach that helps the performance tuning process significantly.

The architecture implementation in the KappaPI 2 tool prototype, has been tested on a range of applications, including some benchmarks (to test bottleneck detection coverage), and real applications, to see real improvements in application performance. And shows the current prototype as a useful tool for enhance application performance avoiding main bottlenecks effects.

References

1. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 63 XII(1) (1996)
2. De Rose, L., Zhang, Y., Reed, D.A.: SvPablo: A Multilanguage Performance Analysis system. In: Puigjaner, R., Savino, N.N., Serra, B. (eds.) Computer Performance Evaluation. LNCS, vol. 1469, pp. 352–399. Springer, Heidelberg (1998)
3. Wolf, F., Mohr, B., Dongarra, J., Moore, S.: Efficient Pattern Search in Large Traces Through Succesive Refinement. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, Springer, Heidelberg (2004)
4. Truong, H., Fahringer, T., Madsen, G., Malony, A., Moritsch, H., Shende, S.: On using SCALEA for Performance Analysis of Distributed and Parall el Programs. In: Supercomputing 2001 Conference (SC2001), November 10-16, 2001, Denver, Colorado, USA (2001)

5. Fahringer, T., Gerndt, M., Riley, G., Larsson, J.: Specification of Performance problems in MPI Programs with ASL. In: Proceedings of ICPP, pp. 51–58 (2000)
6. Espinosa, A., Margalef, T., Luque, E.: Automatic Performance Evaluation of Parallel Programs. In: IEEE proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing (January 1998)
7. Espinosa, A., Margalef, T., Luque, E.: Automatic Performance Analysis of PVM Applications. In: Dongarra, J.J., Kacsuk, P., Podhorszki, N. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1908, pp. 47–55. Springer, Heidelberg (2000)
8. Wolf, F., Mohr, B.: Automatic Performance Analysis of MPI Applications Based on Event Traces. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 123–132. Springer, Heidelberg (2000)
9. Jorba, J., Margalef, T., Luque, E.: Automatic Performance Analysis of Message Passing Applications using the KappaPI 2 tool. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, Springer, Heidelberg (2005)
10. Fahringer, T.: Towards a Standardized Intermediate Program Representation (SIR) for Performance Analysis. In: 5th International APART workshop, SC2003 (November 2003)
11. Jorba, J., Margalef, T., Luque, E., Andre, J., Viegas, D.: Application of Parallel Computing to the Simulation of Forest Fire Propagation. In: Proceedings of International Conference in Forest Fire Propagation, Portugal, November, 1998. vol. 1, pp. 891–900 (1998)

Automatic Monitoring of Memory Hierarchies in Threaded Applications with AMEBA[⋆]

Edmond Kereku and Michael Gerndt

Technische Universität München,
Fakultät für Informatik I10, Boltzmannstr. 3, 85748 Garching, Germany
{kereku, gerndt}@in.tum.de

Abstract. In this paper we present an approach to online automatic monitoring of memory hierarchies in threaded applications. Our environment consists of a monitoring system and an automatic performance analysis tool. The CMM monitoring system uses static instrumentation of the source code and information from the hardware counters to generate performance data for selected code regions and data structures. The monitor supports threaded applications by providing per-thread performance data or by aggregating it. It also provides a monitoring request API for the performance tools. Our tool AMEBA performs an online automatic search for cache and thread-related ASL properties in the code.

1 Introduction

With SMP architectures being more and more a commodity computing resource and with paradigms such as OpenMP, the programming of threaded applications became a task that almost any average programmer can master. A great number of HPC applications originated this way. But this is where the easy part ends. While debugging of such an application is not as easy as writing it, obtaining performance and scalability from it could really be a daunting task.

Memory bottleneck problems already present in serial programs obtain a greater significance in threaded applications. In addition, new problems are added to the traditional problems, e.g., false sharing. Usual profiling and tracing tools cannot find the new problems, what is needed is more extensive and complex monitoring and analysis.

Traditional performance tools assist in analyzing the raw data but do not guide the developer in the iterative performance optimization process. He decides which performance data to collect and which graphical display is best to identify a performance problem. This requires knowledge of performance analysis which most average application developers do not have. By building automatic analysis tools we can make performance analysis accessible to every developer.

Automatic Monitoring Environment for Bottleneck Analysis (AMEBA) [1] is our approach to automatic analysis of threaded applications in SMPs. By

[⋆] This work is part of the EP-CACHE project funded by the German Federal Ministry of Education and Research (BMBF) and the Periscope project funded by the German Science Foundation (DFG).

using the Configurable Monitor for Memory Hierarchies (CMM) [2] monitoring environment, our tool is able to automatically search for complex cache problems in serial and OpenMP regions of an application.

The performance problems are expressed as APART Specification Language (ASL) [3] properties. A property evaluated and holding for a region means that there is a performance problem in that region. The search for bottlenecks can even be refined to single data structures in specific regions of the application.

In order to provide the performance data needed for the evaluation of properties, CMM uses either simulation or the available hardware counters in the processors. Particularly interesting for the monitoring of threads and data structures is the port of CMM to Itanium-based SMPs and especially to ccNUMA architectures, such as the SGI Altix 3700.

The rest of this document is organized as follows: Section 2 shows the support of CMM for threaded applications. Section 3 is an overview of our automatic tool, AMEBA. Section 4 introduces new ASL properties related to threads and cache problems. We evaluate the CMM monitoring system and the automatic search with AMEBA in Section 5. Sections 6 and 7 present related and future work respectively.

2 The CMM Monitoring Environment

The goal of CMM is to provide detailed information about the memory access behavior of Fortran OpenMP programs. Programs written in other programming languages can also be monitored with CMM, provided that language-specific instrumenters for program regions and data structures are available.

The CMM environment, shown in Figure 1, is structured in several layers. Each layer represents a different abstraction for the addresses of data structures. While the hardware monitor is configured through a low level API using physical addresses to restrict the monitoring to an address range, the second level, ePAPI, achieves a higher level of abstraction by working with virtual addresses. At the top of the architecture, the Monitoring Request Interface (MRI) provides performance tools with a symbol-based interface. Tools can for example request the execution time of a specific loop or the number of cache misses within a region of the source code.

CMM can be used by a broad range of performance analysis tools such as GUIs, trace analyzers, or automatic performance tools. In Figure 1 three tools are shown which already use CMM. One of the tools is a trace archiver which configures the monitor to produce trace data and then saves this data in a format readable by VAMPIR [4]. Another tool which uses CMM is Performance Cockpit, an Eclipse-based GUI [5] which allows the user to explore the source code of his program and to interactively request performance data for single regions or data structures. The last tool shown in the figure is the AMEBA automatic analysis tool [1].

Mapping monitored information to code regions and data structures is done by source code instrumentation. That is, code regions such as functions, function

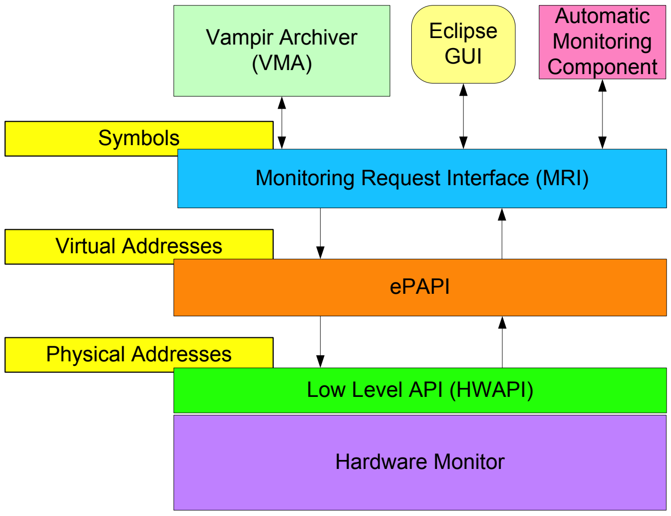


Fig. 1. The layers of the CMM monitoring architecture. Each layer represents a different abstraction for the address of data structures.

calls, loops, or I/O operations are marked in the source code by a code region instrumenter. Due to the overhead of instrumentation at runtime, the user can guide the instrumentation process and select those types of regions to be instrumented. For example, instrumentation can either be performed automatically for all types or regions or can be limited to subroutines and parallel regions thus ignoring sequential loop nests. Additional instrumentation may be necessary to restrict measurements to memory accesses of specific data structures. Such an instrumentation inserts calls to a library that gathers data structures' address information at runtime.

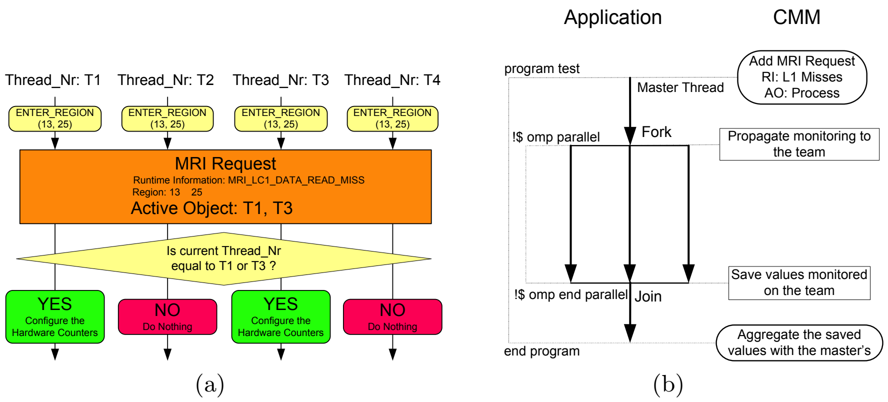
During the execution of the instrumented application, performance tools access the performance data via the MRI. In addition to the dynamic information, the performance analysis tools can also use the static information (information about the program's structure) generated by the instrumenter. This information is helpful for discovering problems dependent on the application's static information and is necessary for the generation of region-related performance measurement requests through the MRI.

2.1 Monitoring Threads with the CMM Environment

The CMM environment's support for threaded applications includes the instrumentation of OpenMP regions, the thread-specific configuration of monitoring resources in runtime, and the ability to deliver thread-related performance data.

In order to configure single threads upon a monitoring request, CMM has a thread management utility for storing thread-related information. The environment uses this utility to determine how many threads are executing the monitored application and which thread issued a call to the monitoring library.

Figure 2 (a) shows how the configuration for thread-related monitoring works. After a thread has entered the monitoring library and a monitoring request is pending for the current region, CMM determines which thread has entered the monitoring library. If the thread id matches one of threads specified in the monitoring request, the hardware resources are configured accordingly, otherwise the execution control is immediately returned to the application.



Thread_Nr: T1
Thread_Nr: T2
Thread_Nr: T3
Thread_Nr: T4
ENTER_REGION (13, 25)
MRI Request
Runtime Information: MRI_LC1_DATA_READ_MISS
Region: 13 25
Active Object: T1, T3
Is current Thread_Nr equal to T1 or T3 ?
YES Configure the Hardware Counters
NO Do Nothing
(a)
Application
CMM
program test
Master Thread
Add MRI Request RI: L1 Misses AO: Process
!$ omp parallel
Fork
Propagate monitoring to the team
Save values monitored on the team
!$ omp end parallel
Join
end program
Aggregate the saved values with the master's
(b)

Fig. 2. (a) Configuration of monitoring for multithreaded applications. (b) Propagation of monitoring configuration on threads.

CMM also implements a monitoring propagation mechanism which correctly handles thread-related MRI requests no matter where in the code they are specified. To understand how this mechanism functions, consider Figure 2 (b). Let us assume that a simple program containing the main program region and an OpenMP parallel region is being monitored. A monitoring request is specified for the main program region which requires the number of L1 cache misses for the whole program. The specified active object is: "process" which means "all the threads" for a multithreaded application.

As the program execution starts, CMM gets the execution control and determines that an MRI request is specified for the main program region. The master thread is configured accordingly. Additionally, CMM saves a copy of this configuration which contains the requested runtime information (kind of performance data), the requested active objects, and, if specified, the requested data structure. The monitoring is started for the master thread and the program continues executing.

The program runs with only the master thread until the OpenMP parallel directive is executed. As shown in the figure, according to the fork-join model of OpenMP, a team of threads is created. If the OpenMP parallel directive was instrumented, all the threads enter the monitoring library of CMM. At this point, CMM propagates the saved configuration to all the threads on the team. The monitoring is started on all the threads except the master which is already monitoring.

At the end of the parallel region, the team of threads enters again the monitoring library. CMM stops the monitoring on all the threads but the master and saves the monitored runtime information for the threads whose monitoring was stopped. At the end of the program, the master thread enters the monitoring library one last time and its monitoring is stopped. The runtime information collected by the master is aggregated with the saved values of the team according to the aggregation specified in the MRI request. Finally, the aggregated result is saved as the MRI request's result.

3 The AMEBA Automatic Performance Analysis Tool

Our analyzer, AMEBA, performs an automated iterative search for performance problems. AMEBA is an online performance tool, meaning that the automatic search happens while the application is running. The search process is iterative in the sense that AMEBA starts with a set of potential performance properties, performs an experiment, evaluates the hypotheses based on the measured data, and then refines the hypotheses. The refinement can either be towards more specific performance properties or towards subregions and data structures of the already tested region.

AMEBA is built over the model shown in Figure 3. This model includes a set of fixed base classes which are implemented in the tool and a set of derivative classes which are implemented separately for a specific monitoring environment, hardware system or application domain. Those classes are dynamically loaded at runtime, which greatly improves AMEBA's versatility and extensibility.

The base classes include the *Strategy* which specifies the search process, the *Property* which describes the performance problems, the *PropertyContext* which

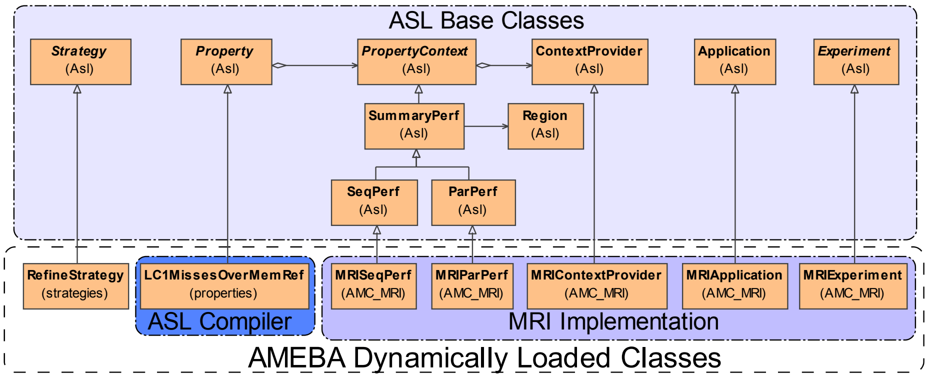


Fig. 3. AMEBA Classes derived from the ASL data model

holds the performance data needed to evaluate the property, and the *ContextProvider* which sets the connection with the monitoring system. The classes derived from the PropertyContext contain summaries of the performance data specific to sequential (*SeqPerf*) or parallel (*ParPerf*) regions, and specific to the monitoring environment (*MRI(Seq/Par)Perf*).

4 Threads and Cache-Related ASL Properties

As stated before, AMEBA uses the APART Specification Language (ASL) to describe the performance problems in terms of condition, confidence, and severity. Consider for example the following ASL Property:

```
property UnbalancedLC1DataMissRateInThreads(ParPerf pp, float t){
 let
  miss_rate(int tid)=pp.lc1_miss[tid]/pp.mem_ref[tid];
  mean=sum(miss_rate(0),...,miss_rate(pp.nrThreads))/pp.nrThreads;
  max = max(miss_rate(0),...,miss_rate(pp.nrThreads));
  min = min(miss_rate(0),...,miss_rate(pp.nrThreads));
 in
  condition: max(max - mean, mean - min) > t;
  confidence: 1;
  severity: max(max - mean, mean - min) * pp.parT[0]; }
```

where *pp* is the summary of performance data for a parallel region and *parT[0]* is the execution time of the master thread.

This property specifies that there is a problem in parallel regions if the L1 cache miss rate in a thread deviates from the mean value achieved over the threads beyond a given percentage *t*. The thresholds (percentage here) are predefined by the expert who defined the properties or can be set manually by the application expert. Furthermore it specifies that the problem is more *severe* if it is found in regions where most of the execution time is spent. We also are 100% *confident* about the existence of the problem in regions where the condition is true because our measurements are based in precise counter values and are not statistical values.

ASL performance properties can be hierarchically ordered and properties can be specified starting from existing predefined ASL templates and existing ASL properties (meta-properties). An ASL compiler is used to generate C++ classes, in the form of definition and partial implementation, from ASL specifications. Finally the tool developer includes the classes in his automatic analyzer.

[3] and [6] already defined ASL properties for serial regions as well as for MPI and OpenMP regions based on time measurements. We extended this set of properties by describing performance problems whose existence can be discovered with the help of performance data delivered by the CMM environment. The properties we defined determines cache problems in parallel regions of threaded applications running on SMP nodes and are divided into five groups. Each group is defined by an ASL template. With the help of this template ASL properties

are created for L1, L2, and L3 caches as well as refinements for write and read misses.

1. **PropPerThread.** The properties created from this template determine the existence of a performance problem in one specific thread. The most important use of these properties is not to be directly searched by the automatic performance tool (although this is possible), but to serve as parameters for the creation of other meta-properties.
2. **PropOnAllThreads.** This template is used to create properties which hold for all threads running a parallel code region.
3. **PropInSomeThreads.** The properties created from this template hold for a parallel region if the performance problem is exhibited in at least one of the threads.
4. **MeanMissRatePar** specifies that there is a problem in the parallel region if it exhibits a high mean value of cache misses among the threads.
5. **UnbalancedDMissRateInThreads** specifies that there is a problem in the parallel region if the differences of cache misses measured for different threads are too large.

If a performance problem expressed by an ASL property is the refinement of another performance problem, we organize them in an hierarchical order. In AMEBA we implemented search strategies which take this organization into account. For example `UnbalancedLC1DataMissRateInThreads` is actually a refinement of the problem specified by `LC1DataMissRateInSomeThreads`.

5 Evaluation

We evaluated our automatic performance analysis approach using several benchmark applications. The evaluation includes two main aspects. The first aspect is the evaluation of the CMM monitoring environment, the overhead introduced by the performance monitoring. The second aspect is the evaluation of our automatic tool, AMEBA.

The evaluation of the CMM was based on four applications. Three of them are benchmark applications taken from the SPEC OMP2001 and the NAS Parallel Benchmark (NPB) 3.2. The fourth evaluation application is a Gaussian Elimination code parallelized with OpenMP. The evaluation of the CMM environment was performed on 4X Itanium2 SMP nodes of the Infiniband Cluster at the LRR-TUM.

To evaluate the overhead introduced by the instrumentation and the monitoring, three versions were compiled for each of the applications. The first version is the original application. The second version is the instrumented application linked to an empty monitoring library. That is, each function in the monitoring library returns immediately. This way, we measure the instrumentation's overhead. The third version is the instrumented application linked to the CMM's monitoring library.

For the third version, we performed two sets of runs. Once, the application was executed but no monitoring requests were made. This way, we measured the overhead introduced by the monitoring library. The second set of runs included full monitoring during the whole application's execution. The monitoring configuration was done so that all the four performance counters of Itanium were occupied.

All experiments were executed with four threads and a different CPU was used by each thread. Table 1 shows the resulting execution times for the experiments compared to the execution times of the original applications. The deviations from the original execution times was calculated for experiment X as: $Deviation_{\mathrm{X}} = (Time_{\mathrm{X}} - Time_{\mathrm{original}}) \div Time_{\mathrm{original}}$. As shown by the table, the overhead introduced by CMM is very low to insignificant.

To evaluate the AMEBA automatic tool we used up to 32 CPUs on the SGI Altix 3700 Bx2 at the Leibniz Rechenzentrum (LRZ) in München. We searched for thread-related properties using a strategy which searches only on the OpenMP regions of the application. The strategy searches first for the properties LC2DMissRateInSomeThreads and LC3DMissRateInSomeThreads. For regions where one of these properties hold, the search refines by evaluating the properties UnbalancedLC2DMissRatePar and UnbalancedLC3DMissRatePar.

These are the results of the automatic search. For the sake of presentation we only show the results for FT from the NAS Parallel Benchmark.

```
                The results for FT running with 16 threads

      Region                         LC2DMissRate           UnbalancedLC2D
reg. type   file  line               InSomeThread             MissRatePar
DO_REGION, ft.f, 227                   0.0796055              0.34044
DO_REGION, ft.f, 567                   0.0553533
DO_REGION, ft.f, 612                   0.0546933

      Region                         LC3DMissRate           UnbalancedLC3D
reg. type   file  line               InSomeThread             MissRatePar
DO_REGION, ft.f, 227                   0.0364304              0.439014

                The results for FT running with 32 threads

      Region                         LC2DMissRate           UnbalancedLC2D
reg. type   file  line               InSomeThread             MissRatePar
```

Table 1. Execution times for different runs of the applications showing the overhead of instrumentation, monitoring library, and measurements

Applications	LU	FT	SWIM	PGAUSS
Instrumented regions	113	55	29	11
Entered regions	2E+08	8E+06	3604	537726
Original execution time (seconds)	1546.5	219.8	388.5	680.1
Instrumentation overhead (time)	0.42%	0.52%	0.03%	0.09%
Monitoring library overhead (time)	2.45%	1.75%	0.13%	0.17%
Measurements overhead (time)	2.63%	1.66%	1.84%	2.54%

```
DO_REGION, ft.f, 227               0.079456               0.791722
DO_REGION, ft.f, 567               0.0555033
DO_REGION, ft.f, 612               0.0545275

     Region                      LC3DMissRate           UnbalancedLC3D
reg. type  file  line            InSomeThread             MissRatePar
DO_REGION, ft.f, 227               0.0299765              0.537159
```

The LC2-related properties hold for a region if the miss rate is higher than 5% the LC3-related properties hold if the miss rate is higher than 2%. The threshold for UnbalancedLC(2,3)DMissRate is set to 30%. This means that we consider a region to have a problem if the miss rate in one of the threads deviates by at least 30% from the mean miss rate over all the threads.

The automatic search results show that FT seems to have the same cache problems when running with 16 and 32 threads. There are three regions with LC2 cache miss problems, and one of them also has LC3 miss problems. What we also observe is that the problem of unbalanced cache misses is aggravated when running with 32 threads.

6 Related Work

The Paradyn tool by Miller et al.[7] is the closest related work. The Performance Consultant module of Paradyn uses a W^3 (*Why* there is a problem, *Where* in application is the problem, and *When* the problem occurs) search model to automate the identification of performance problems by analyzing data provided by means of run-time dynamic instrumentation. Performance problems are simply expressed in terms of a threshold and a counter-based metric. The dynamic instrumentation of measurement probes in the running code is also guided by the Property Consultant.

EXPERT[8] developed at the Forschungszentrum Jülich, performs an offline hierarchical search for patterns of inefficient executions in trace files. EXPERT uses source code instrumentation for MPI, OpenMP, and hybrid applications and defines more complex bottlenecks than Paradyn by using Python for the specification of performance properties. The huge amount of raw performance data it produces and the long execution time of the post mortem analysis, pose the main limitations of this tool for use with large parallel programs.

JavaPSL[9] is a language for specifying performance properties based on the Apart Specification Language (ASL) developed in the European Working on Automatic Performance Analysis Tools (APART). Performance properties are formalized as Java abstract classes taking advantage of Java language mechanisms such as polymorphism, abstract classes, and reflection. JavaPSL is used in a tool called Aksum[10] which relies on source code instrumentation to generate raw performance data, stores it in a relational database and then automatically searches for the existence of the predefined performance properties.

7 Future Work

In the future we would like to support automatic reduction of the monitoring overhead. This could be done by automatically guiding the instrumentation process. The automatic tool may run the instrumented application for a certain period of time and discover the regions with minor impact on the overall application's performance, but high impact on the monitoring overhead. To eliminate these regions from the search, a reinstrumentation of the program may be requested by the tool.

CMM could also be extended to support additional programming languages. Actually, only Fortran 90 OpenMP is fully supported. We plan to support C and C++ by using existing instrumenters, or implementing new language-specific ones. Finally AMEBA could be extended from an automatic performance analyzer to an automatic performance optimizer.

References

1. Kereku, E., Gerndt, M.: The EP-Cache Automatic Monitoring System. In: Proceedings of Parallel and Distributed Computing and Systems, Phoenix AZ, pp. 39–44 (November 2005)
2. Kereku, E., Li, T., Gerndt, M., Weidendorfer, J.: A Data Structure Oriented Monitoring Environment for Fortran OpenMP Programs. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 133–140. Springer, Heidelberg (2004)
3. Fahringer, T., Gerndt, M., Mohr, B., Wolf, F., Riley, G., Träff, J.L.: Knowledge Specification for Automatic Performance Analysis. Technical report, APART Working Group (2001)
4. Barabas, L., Müller-Pfefferkorn, R., Nagel, W.E., Neumann, R.: Tracing the cache behavior of data structures in fortran applications. In: Proceedings of Parallel computing ParCo 2005, Malaga, Spain (2005)
5. Li, T., Gerndt, M.: Cockpit: An Extensible GUI Platform for Performance Tools. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, Springer, Heidelberg (2005)
6. Gerndt, M.: Specification of Performance Properties of Hybrid Programs on Hitachi SR8000. Technical report, Technische Universität München (2002)
7. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvine, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer 28(11), 37–46 (1995)
8. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture: the EUROMICRO Journal 49(10-11), 421–439 (2003)
9. Fahringer, T., Seragiotto, C.: Modelling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In: Proceedings of the Conference onSupercomputers (SC2001) (2001)
10. Fahringer, T., Seragiotto, C.: Aksum: a performance analysis tool for parallel and distributed applications. In: Performance analysis and grid computing, Norwell, MA, Kluwer Academic Publishers, Boston (2004)

Visualization of Repetitive Patterns in Event Traces

Andreas Knüpfer, Bernhard Voigt, Wolfgang E. Nagel, and Hartmut Mix

TU Dresden, Center for Information Services and
High Performance Computing (ZIH), 01062 Dresden, Germany
{andreas.knuepfer, wolfgang.nagel, hartmut.mix}@tu-dresden.de,
voigt@zhr.tu-dresden.de

Abstract. Performance Tracing has always been challenged by large amounts of trace data. Software tools for trace analysis and visualization successfully cope with ever growing trace sizes. Still, human perception is unable to "scale up" with the amounts of data.

With a new model of trace visualization, we try to provide *less data* but *additional information* or rather more convenient information to human users. By marking regular repetition patterns and hiding the inner details, less complex visualization can offer better insight. At the same time potentially interesting irregular sections are revealed.

The paper introduces the origin of repetition patterns and outlines the detection algorithm used. It demonstrates the new visualization scheme which has also been incorporated into Vampir NG as a prototype. Finally, it gives an outlook on further development and possible extensions.

1 Performance Tracing

Event tracing is a well-established method for performance analysis of computer programs, especially in parallel and High Performance Computing (HPC) [5,9]. It has a reputation for producing *large* amounts of trace data where "*large*" has always been defined by the time's standards.

Many advancements in HPC contributed to that. This includes faster processors and growing parallelism. Also, more detailed instrumentation and additional data sources increase trace data volume. Last but not least availability of larger memory and storage capacities allowed traces to grow.

Therefore, trace analysis and visualization has always been a challenging task on contemporary computers and always will be. The more so as one cannot require an actual super-computer for analyzing super-computer traces.

But what is the effect from the human users' perspective? Could screen resolution grow with an appropriate rate? Can human perception *scale* with the growing amounts of data as well? And is there really *more information* when there is simply *more data*[1]?

[1] Let "information" be what contributes to the users insight while "data" is just the byte sequence transporting information.

The next chapter tries to address those questions. Then, Section 3 will propose a new visualization methodology to provide *more convenient information* or to the user with *less data*. The following Section 4 will outline the algorithms for pattern detection. In the final Section 5 there is a conclusion as well as an outlook on improved analysis methods based on patterns.

2 Data vs. Information of Traces

There are several reasons for growing trace sizes. First, there may be larger software projects with bigger source codes. Also, it might grow larger due to optimization and specialization of code. Second, instrumentation and measurement evolve, providing additional data, for example performance counters in processors, in communication sub-systems and in I/O backends. Third, longer or faster running programs with more and more parallelism will increase the number of repeated executions of certain program parts.

The former two reasons have only secondary effect on growing trace data sizes. They will hardly increase it by orders of magnitude. Only the latter is responsible for traces of tens of gigabytes, today. On average the trace size will double when iteration count or run-time is doubled or when there are twice as many parallel processes. (This involves a certain redundancy which is exploited by approaches like introduced in Section 4.1.)

In terms of information to the user the same situation looks different. Assuming an interesting situation can be described as "sequence $A = (a_1, a_2, \dots)$ is repeated n times". That might have been inferred from a Vampir display showing distinguishable iterations like in Figure 2.

How would the statement "sequence A is repeated $4n$ times" relate to it? (Compare Figure 1) Would this fact be four times as useful to the human user? Or would it carry an "equal amount" of information? Or even less information because it might not be distinguishable anymore with limited screen resolution? Thus, information is difficult to quantify, although data size is evidently fourfold.

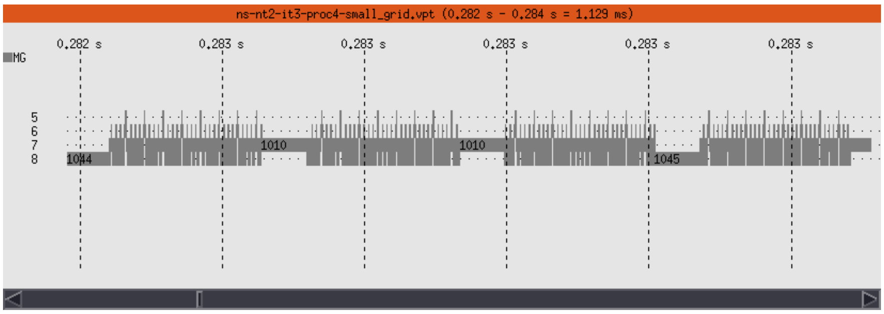


Fig. 1. Vampir screen-shot of 32 identical iterations hardly distinguishable with current horizontal resolution. Note that iterations are of strictly regular structure unlike runtime which shows some notable delays.

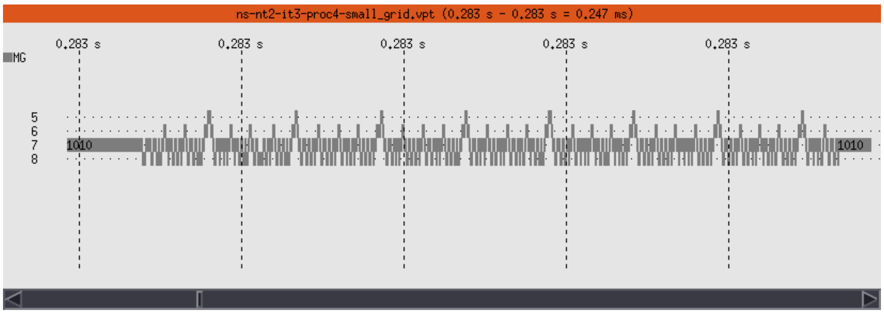

ns-nt2-it3-proc4-small_grid.vpt (0.283 s - 0.283 s = 0.247 ms)
0.283 s
MG
5
6
7
8
1010

Fig. 2. Vampir screen-shot showing a subset of 8 iterations from above (Figure 1). Only with this zoomed version iterations' structure becomes visible.

However, hidden in the data there are some additional information of interest. First, whether there is a (strict) regular standard behavior among iterations. Most likely this is independent of iteration count. Second, if there are outliers and abnormal cases differing from regular behavior.

This additional information is not accessible to the human user by pristine visualization. Rather sooner than later human perception will be overcharged by too much data. Could one tell for the iterations shown in Figures 1 and 2?

Therefore, we want to propose a new scheme of visualization that reduces the amount of data in favor of information perceivable by human users. Nevertheless, all familiar information *and* data will still be available on demand.

3 Visualization of Repetition Patterns

The new visualization approach focuses on Process Timeline Diagrams like found in the Vampir NG [1] tool, for example. For each and every function call a rectangle represents a state of execution, compare Figure 3a. Now, in a simplified Process Timeline Diagram all regular repetition patterns of arbitrary size and nesting depth are replaced with a single highlighted rectangle. See Figure 3b for an example. Those rectangles indicate a region with regular behavior, inner details are not shown.

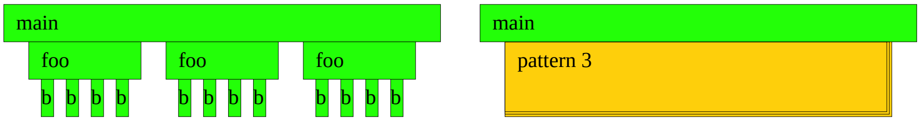


Fig. 3. Plain Process Timeline Diagram of some nested function calls (a), left, and simplified Pattern Diagram of the same situation (b), right. The highlighted rectangle **'pattern 3'** states that there is a repetition of regular call sequences.

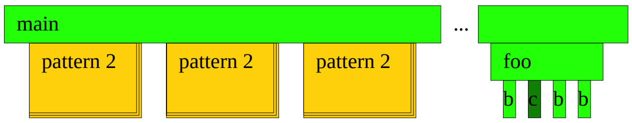


Fig. 4. Pattern Diagram after `pattern 3` in Figure 3b has been decomposed. The fourth call to `foo` is not covered by the pattern because there is one differing sub-call.

However, this may hide too much information. Therefore, patterns can be decomposed interactively. This replaces every instance of a pattern by its direct sub-patterns. See Figure 4 for an example. All sub-patterns can be decomposed as well until the fully decomposed view is identical to the traditional one (Figure 3a). In addition to decomposing patterns there is a context dialog available on demand to provide the pattern structure as well as statistics about all occurrences.

This kind of visual representation allows an easy distinction of regular and irregular parts of a trace. For example, the fourth iteration in Figure 4 is not covered by a pattern because there is a small structural difference. In general such important differences can hardly be perceived directly from a traditional timeline display. Either, because there are too many details to figure out, or because differences are not visible at all when there are more details than actual screen pixels.

4 Pattern Detection

Actual pattern detection is based on *Complete Call Graphs* (CCG) [8]. From this general purpose data structure it derives the so called *Pattern Graph* [10] which contains the pattern information.

4.1 Complete Call Graphs

Usually, the handling of event trace data is done in simple and straight forward linear data structures like arrays and lists. Such data structures can be quite efficient in terms of their inherent memory access performance. Some of them support efficient search algorithms like binary search. However, they do not allow compression of redundant event sequences as frequently found in HPC program traces [6,7,8].

Complete Call Graphs do allow in-memory compression that is fully transparent to read access. The compression scheme can be adjusted to work lossless or lossy with respect to event properties like time, duration, etc. Deviations due to lossy compression can be controlled by customizable bounds.

The *Complete Call Graph* data structure resembles the complete call tree. It is not limited to the simple *caller-callee* relation, which is commonly known as the Call Graph or Call Path [2,3,4]. Instead, it holds the full event sequence including time information in a tree.

An instance of such a tree structure is maintained for every process covered. The actual tree is formed by nodes representing the nested function calls. Non-function-call events like messages, hardware counter samples, etc. are attached as leaf nodes.

The data structure allows linear traversal with respect to event timestamps similar to arrays and linear lists. Furthermore, it can be searched with respect to time stamps with $O(\log N)$ complexity.

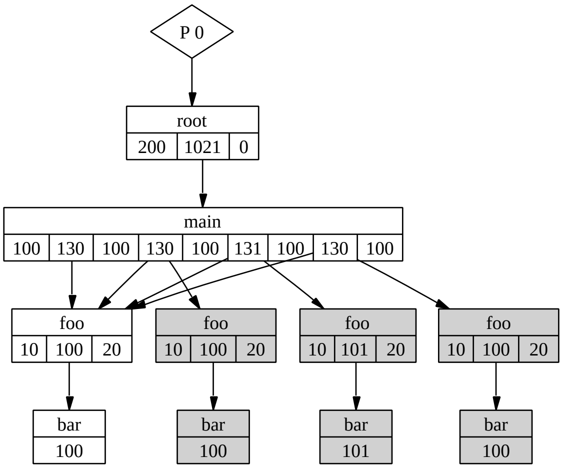


Fig. 5. Example of a CCG. All subtrees with `foo` as root node are mapped onto a single instance while the redundant copies are discarded. Please, note that the third instance of `foo` deviates from the other instances. It is nonetheless going to be compressed given that its deviation is below a specified bound.

HPC applications typically feature very many almost identical iterations which cause repetitive, i.e. potentially redundant, sub-sequences in the event stream. Those are mapped to identical or similar CCG subtrees[2]. The compression scheme replaces (many) redundant sub-trees with a reference to a (single) representing instance reducing the number of graph nodes. CCG compression has proven very suitable for HPC traces. For real-life examples compression ratios R_n in the order of 100 : 1 to 1000 : 1 have been observed [6,8].

Naturally, compressing (i.e. transforming the CCG in the described way) conflicts with the global tree property[3]. However, this is no limitation to the algorithms for construction, compression and querying since the local tree property is maintained[4] and because the nodes graph is acyclic.

[2] Where the meaning of *similar* needs to be defined.

[3] The global tree property requires that every node has exactly one parent node except for the root node which has none.

[4] According to the local tree property every node has $n \in \mathbb{N}_0$ child nodes.

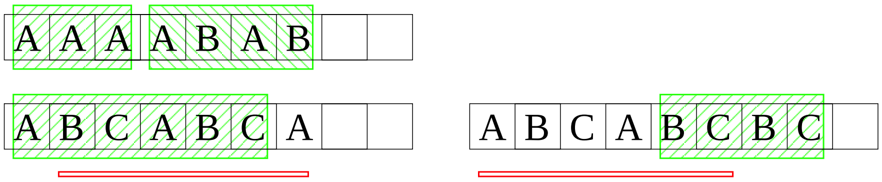


Fig. 6. Pattern examples: In the first example (top) a sequence with two patterns is shown. First, a pattern of length one appearing three times. And second, a pattern with two states appearing twice. The second example (bottom left) shows the pattern `ABC` appearing twice (marked) while the conflicting pattern `BCA` is ignored (underlined). The third example (bottom right) shows how the pattern `ABC` (underlined) is ignored in favor of the shorter pattern `BC` (marked).

To successfully map multiple sub-trees of uncompressed CCGs certain node transformations are required in order to reveal equalities or similarities. Most notably, this applies to the timestamp assigned to all events. According to their nature, timestamps are strictly increasing for every process. Thus, at first glance there is no exploitable repetition inside a single process trace. Therefore, the tree nodes are transformed to store time durations instead of absolute timestamps, which resolves this matching issue (see Figure 5).

More details about the CCG data structure as well as efficient algorithms for construction and querying, further optimizations and benchmark figures are presented in detail in [6,7,8].

4.2 Pattern Graph

Based on an existing CCG the *pattern graph* is computed. It contains only graph nodes' pattern affiliation but no run-time information or further details. A pattern is defined as follows:

Definition: Pattern

A pattern is a sequence of l states which is consecutively repeated k times. A state is either a function call or a hierarchy of calls. Patterns can be nested. Overlapping patterns are not allowed, instead smaller pattens are preferred against longer patterns. For overlapping pattern candidates of same length the earlier one is preferred.
(See Figure 6 for some examples of patterns and conflicting patterns.)

The pattern detection algorithm is applied to every function call node in the CCG, i.e. to every non-leaf node of the CCG. Assume the node has m child references $c_0, ..., c_{m-1}$. Then the detection algorithm checks for every suitable pattern length $l = 1, ..., {}^{m}\!/2$ if there is a pattern at position $i = 0, ..., m - l - 1$, i.e. $c_j \equiv c_{j+l} \forall j = i, ..., i + l - 1$. If there is a mismatch at position j the next position $i'$to check is $j + 1$ instead of $i + 1$. Compare Figure 7. The operation $\equiv$ is a simple pointer comparison.

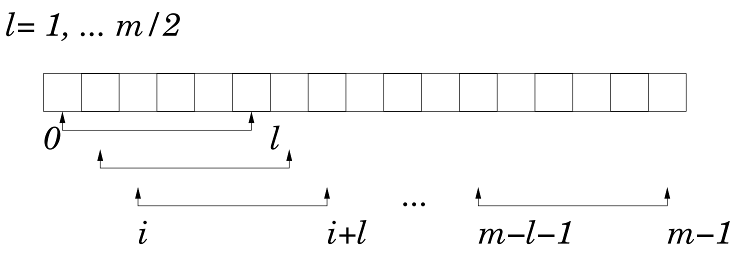


Fig. 7. Pattern detection algorithm: successively compare pointers at indices j and $j + l$ for all $j = i, ..., i + l - 1$ with $l = 1, ..., m/2$ and $i = 0, ..., m - l - 1$

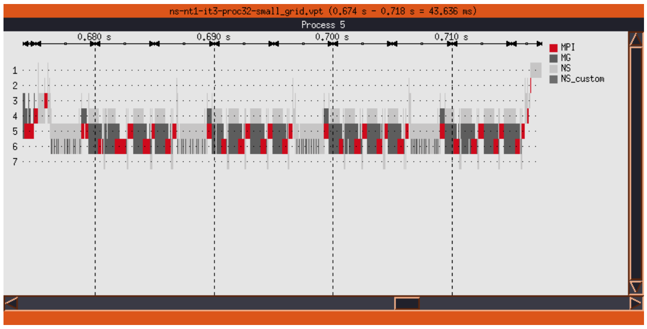

ns-nt1-it3-proc32-small_grid.vpt (0.674 s - 0.718 s = 43.636 ms)
Process 5
0.680 s
0.690 s
0.700 s
0.710 s
1
2
3
4
5
6
7
MPI
MG
NS
NS_custom

Fig. 8. Example screen-shot of Vampir NG's process timeline display [1] in traditional way, showing four iterations of a complex call hierarchy

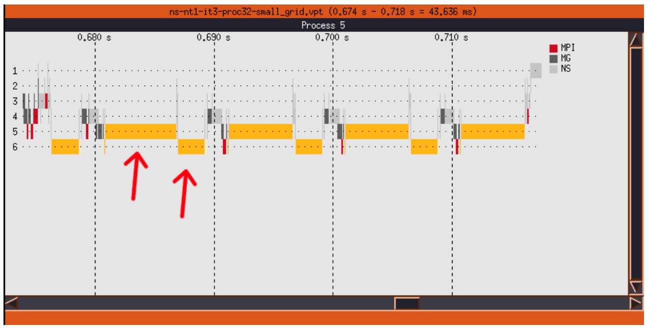

ns-nt1-it3-proc32-small_grid.vpt (0.674 s - 0.718 s = 43.636 ms)
Process 5
0.680 s
0.690 s
0.700 s
0.710 s
1
2
3
4
5
6
MPI
MG
NS

Fig. 9. Pattern display showing the same situation as above (Figure 8). The main pattern of the four iterations has already been decomposed into two different sub-patterns (see arrows).

The computational complexity for pattern detection at a single node with m child references results in $O(m^2)$ for this algorithm. Note that without the restriction to consecutive sub-sequences the complexity would increase to $O(m^3)$.

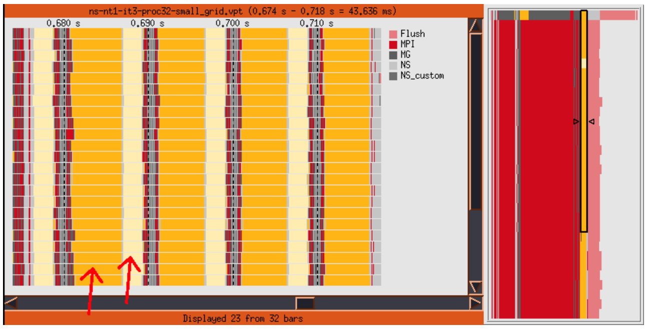

ns-nt1-it3-proc32-small_grid.vpt (0.674 s - 0.718 s = 43.636 ms)
0.680 s
0.690 s
0.700 s
0.710 s
Flush
MPI
MG
NS
NS_custom
Displayed 23 from 32 bars

Fig. 10. Global pattern display with the same zoom level as above (Figure 9). The latter sub-pattern is shown like before. A mouse-over event at one instance of the former sub-pattern highlights all occurrences of that pattern in all processes in a lighter color (second arrow). A thumbnail view at right hand side provides a global overview.

For a whole CCG with n nodes the pattern detection algorithm has a complexity of $O(n \cdot m)$. Here, m is the maximum number of child function calls per function. Unlike the branching factor b of a CCG, which can be limited by a given constant, m is unbounded. In the worst case of a completely flat call hierarchy m equals n.

The term n is the node count of the compressed CCG. Depending on compression ratio R_n it is much smaller than the uncompressed node count N:

$$n = \frac{N}{R_n}.$$

Since soft properties, i.e. properties subject to lossy compression, are unimportant for pattern detection, it is suitable to always use the maximum compressed CCG, and thus have smallest node count n.

The pattern detection algorithm can easily be parallelized by processing CCG nodes independently [10]. Only a simple synchronization operation for pattern tokens will be necessary in order to obtain a consistent global graph.

5 Conclusion and Outlook

The pattern detection and visualization outlined above have been incorporated into the Vampir NG tool as a prototype [10] see Figures 8, 9 and 10. Patterns are represented just like ordinary states but highlighted in orange. That is true for global as well as local timeline displays (Figures 9 and 10).

At first, highest level patterns are shown, hiding all inner details. Via context menu a pattern can be decomposed. That means all occurrences of that pattern are decomposed revealing contained sub-patterns or states. By this means there is always a consistent display – for every pattern either all instances are decomposed or none.

A mouse-over event on any pattern will highlight all occurrences of that pattern in light orange in all displays (see Figure 10). This allows an easy overview about local or global distribution of patterns.

Further improvement in user interface design might introduce alternative visual representation for patterns. For example, height of rectangles showing pattern occurrences could be adapted to the maximum call depth of the contained call hierarchy.

Another prospective development might be towards a generalization of pattern definition. So far, pattern detection relies on structurally identical patterns. Then, pattern matching could accept certain variation, e.g. regard loops as equal if they have identical inner structure but different iteration counts. Again, this might involve alternative visual representation.

Last but not least, pattern detection opens new opportunities for automatic performance analysis (like in [11]). Instead of analyzing performance properties of single function calls, now whole pattern occurrences could be addressed. This might reveal standard behavior for frequently appearing patterns, for example with respect to run-time, hardware performance counter values, communication speed etc. Based on this, outliers or performance flaws could be identified.

References

1. Brunst, H., Nagel, W.E., Malony, A.D.: A distributed performance analysis architecture for clusters. In: IEEE International Conference on Cluster Computing, Cluster 2003, Hong Kong, China, December 2003, pp. 73–81. IEEE Computer Society Press, Los Alamitos (2003)
2. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: A Call Graph Execution Profiler. In: SIGPLAN Symposium on Compiler Construction, Boston, Massachusetts, pp. 120–126 (1982)
3. Grove, D., Chambers, C.: An Assessment of Call Graph Construction Algorithms. Technical Report RC 2169, 9, IBM Research Report (March 2000), `citeseer.nj.nec.com/grove00assessment.html`
4. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call Graph Construction in Object-Oriented Languages. In: ACM Conference on Object-Oriented Programming, Atlanta, Georgia, pp. 108–124. ACM Press, New York (1997)
5. Knüpfer, A., Brunst, H., Nagel, W.E.: High Performance Trace Visualization. In: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing, Lugano, Switzerland, Febuary 2005, pp. 258–263 (2005) ISBN 0-7695-2280-7
6. Knüpfer, A., Nagel, W.E.: Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis. In: Proc. of ICPP 2005, June 14-17, 2005. Oslo/Norway (2005)
7. Knüpfer, A., Nagel, W.E.: New Algorithms for Performance Trace Analysis based on Compressed Complete Call Graphs. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2005. LNCS, vol. 3514, Springer, Heidelberg (2005)
8. Knüpfer, A., Nagel, W.E.: Compressible Memory Data Structures for Event-Based Trace Analysis. Future Generation Computer Systems 22(3), 359–368 (2006)

9. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. In: Dongarra, E.S.J.J., Meuer, H.-W. (eds.) TOP500 Supercomputer Sites (November 1995)
10. Voigt, B.: Effiziente Erkennungs- und Visualisierungsmethoden für hierarchische Trace-Informationen. Diploma thesis (german), TU Dresden (2006)
11. Wolf, F., Mohr, B.: Automatic Performance Analysis of MPI Applications Based on Event Traces. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 123–132. Springer, Heidelberg (2000)

Optimization of Instrumentation in Parallel Performance Evaluation Tools

Sameer Shende, Allen D. Malony, and Alan Morris

Performance Research Laboratory,
Department of Computer and Information Science,
University of Oregon, Eugene, OR 97403, USA
{sameer, malony, amorris}@cs.uoregon.edu

Abstract. Tools to observe the performance of parallel programs typically employ profiling and tracing as the two main forms of event-based measurement models. In both of these approaches, the volume of performance data generated and the corresponding perturbation encountered in the program depend upon the amount of instrumentation in the program. To produce accurate performance data, tools need to control the granularity of instrumentation. In this paper, we describe developments in the TAU performance system aimed at controlling the amount of instrumentation in performance experiments. A range of options are provided to optimize instrumentation based on the structure of the program, event generation rates, and historical performance data gathered from prior executions.

Keywords: Performance measurement and analysis, parallel computing, profiling, tracing, instrumentation optimization.

1 Introduction

The advent of large scale parallel supercomputers is challenging the ability of tools to observe application performance. As the complexity and size of these parallel systems continue to evolve, so must techniques for evaluating the performance of parallel programs. Profiling and tracing are two commonly used techniques for evaluating application performance. Tools based on profiling maintain summary statistics of performance metrics, such as inclusive and exclusive time or hardware performance monitor counts [1], for routines on each thread of execution. Tracing tools generate time-stamped events with performance data in a trace file. Any empirical measurement approach will introduce overheads in the program execution, the amount and type depending on the measurement method and the number of times it is invoked. However, the need for performance data must be balanced with the cost of obtaining the data and its accuracy. Too much data runs the risk of measurement intrusion and perturbation, yet too little data makes performance evaluation difficult.

Performance evaluation tools either employ sampling of program state based on periodic interrupts or direct instrumentation of measurement code. Sampling generally introduces a fixed overhead based on the inter-interrupt sampling interval. Thus, sampling is often considered to be generate less perturbation on performance. Unfortunately, sampling suffers from lack of event specificity and an inability to observe inter-event actions. For these reasons, sampling is less viable approach for robust parallel performance analysis. Here, we consider direct measurement-based techniques where instrumentation hooks are inserted in the code at locations of relevant events. During execution, events occur as program actions and the measurement code is activated to inspect performance behavior. Because event generation does not occur as the result of an interrupt, the number of events generated is instead tied to how often the program code executes. In some cases, this could be significant. Furthermore, the time between event measurements is not fixed, which can affect measurement accuracy. Whereas direct instrumentation and measurement can produce robust performance data, care must be taken to maintain overhead and accuracy. We use the term "instrumentation optimization" to describe this objective.

In this paper, we discuss our work in optimizing program instrumentation in the TAU performance system [2]. Section §2 gives further background and motivation for the problem. Sections §3 and Sections §4 describe how we can limit the instrumentation based on selective instrumentation and runtime measurement control. Section §5 discusses our future plans.

2 Motivation

Given a performance evaluation problem, certain performance data must be observed to address it. How should the program be instrumented and measurements made to capture the data? If the measurements cost nothing, the degree of instrumentation is of no consequence. However, measurements introduce overhead in execution time and overhead results in intrusion on the execution behavior. Intrusion can cause the performance of an application to change (i.e., to be *perturbed*). Performance perturbation directly affects the accuracy of the measurements. Accuracy is also affected by the resolution of the performance data source (e.g., real time clock) relative to the granularity of the metric being measured (e.g., execution time of a routine). Thus, it is not enough to know what performance data needs to be observed. One must understand the cost (overhead, intrusion, perturbation, accuracy) of obtaining the data.

Optimization of instrumentation is basically a trade-off of performance data detail and accuracy. If a measurement is being requested at a granularity too fine for the measurement system, the performance data will be faulty. Clearly, instrumentation should be configured to prevent the measurement from being made at all. At the other end of the spectrum, there can be situations where more data is gathered than necessary for a certain level of accuracy. Here, instrumentation optimization would be used to limit unnecessary overhead.

Most users are not so sophisticated in their performance measurement practices. Therefore, performance tools must provide mechanisms that enable users to understand instrumentation effects and accuracy trade-offs, and adjust their performance experiments accordingly. Balancing the volume of performance data produced and the accuracy of performance measurements is key to optimizing the instrumentation. Techniques for improving performance observability fall into three broad categories:

- *Instrumentation* – Techniques that *reduce* the number of instrumentation points inserted in the program.
- *Measurement* – Techniques that *limit* and *control* the amount of information emitted by the tool at the instrumentation points, and
- *Analysis* – Techniques that *scale* the number of processors involved in processing the performance data, and techniques that reduce and reclassify the performance information.

In this paper, we will limit our discussion to instrumentation and measurement based approaches.

3 Instrumentation

There are two fundamental aspects to optimizing instrumentation: deciding which events to instrument and deciding what performance data to measure. Event selection can occur prior to execution or at runtime. However, what is possible in practice depends entirely on the instrumentation tools available. Different types of events also determines the complexity of the instrumentation problem. The following discusses the possible instrumentation approaches and optimization issues that arise.

3.1 Event Types

Because direct instrumentation inserts measurement instructions in the program code, events are most often defined with respect to program flow of control. *Standard events* include the begin and end of routines and basic blocks. Other events may be defined at arbitrary code locations by the user. Events can also be defined with respect to program state. These events are still instrumented for with code insertion, but are 'enabled' depending on the value of state variables or parameters. We also distinguish between paired *entry/exit* events and events that are *atomic*.

The different types of events represent, in a sense, the range of possible event instrumentation scenarios. This could range from having all routines in a program code instrumented to having only a few specific routines instrumented, such as in a library. Clearly, the more events instrumented for, the more measurements will be active. Also, it is important to distinguish events that occur on individual threads (processes) of execution. Hence, the more threads executing, the more concurrent events are possible.

3.2 Instrumentation Mechanisms

Coinciding with the types of events to instrument are the mechanisms for instrumentation. There are five common approaches:

– *Compiler* – Instrumentation occurs as the program is being compiled. The choice of events is determined by the compiler, but options may be provided. Instrumentation for *gprof*-style profiling is generally done.
– *Library* – A library has been pre-instrumented and this instrumentation can be invoked by a program just by relinking. The MPI library is a special case of this since it also provides a "profiling interface" (PMPI) for tool developers to build their own instrumentation.
– *Source (automatic)* – This is an instrumentation approach based on source rewriting. The Opari [3] and TAU instrumentor [2] tools work in this manner.
– *Source (manual)* – A manual instrumentation API often accompanies measurement tools to allow users to create their own events anywhere in the program. For non-trivial applications, the effort to do so becomes quite cumbersome.
– *Binary* – Instead of working at the source level, some tools can instrument binary code, a form of binary rewriting. For the most part, the events are the same except lower-level code features may be targeted. Binary rewriting is hard and ISA specific.
– Dynamic – Some tools work at runtime to instrument executable code. For instance, DyninstAPI [5] can dynamically instrument running parallel executions based on code trampolining techniques.

The choice of which instrumentation mechanism to use is based on different factors. One factor is the accessibility (visibility) of events of interest. Another is the flexibility of event creation and instrumentation. These factors affect whether the mechanism can meet the event requirements. However, mechanisms are sometimes chosen based on their perceived overheads. For instance, source instrumentation has been criticized for its effects on code optimization, while binary and dynamic instrumentation purport to work with optimized code. On the other hand, source-instrumented measurement code also undergoes optimization, and can generate more efficient code than dynamically-instrumented measurements inserted under pessimistic assumptions of register usage and other factors.

3.3 Selective Instrumentation

Given the above discussion, instrumentation optimization with respect to events reduces essentially to a question of selective instrumentation. Put another way, we want to conduct performance experiments that capture performance data only for those events of interest, and nothing more. If a mechanism does not allow instrumentation of some event of interest, it is less useful than one that does. When there are many events that can be instrumented, a means to select events will allow only those events desired to be instrumented. Each of the mechanisms above can support event selection, but not all tools based on these mechanisms support it.

In the TAU project, a variety of instrumentation techniques are used: source pre-processing with PDT [4], MPI library interposition, binary re-writing and dynamic instrumentation with DyninstAPI [5], and manual. For each of these mechanisms, TAU allows an event selection file to be provided to control what events are to be 'included' in and 'excluded' from instrumentation. The specification format also allows for instrumentation to be enabled and disabled for entire source files.

Unfortunately, it is common for naive performance tool users to ignore such support and ask for all events to be instrumented. There are two downsides of this. First, it is probably true that not all of the events are really needed. Second, some events may be generated that are very small, resulting in poor measurement accuracy, or high-frequency, causing excessive buildup of overhead. Of course, a user could use TAU's selective instrumentation to disable such events, but they might not be aware of them.

TAU's selective instrumentation file also allows rules for instrumentation control to be specified. TAU provides a tool, *tau_reduce*, to analyze the profiles and apply the instrumentation rules. Effectively, the output is a list of routines that should be excluded from instrumentation. Naive instrumentation of parallel programs can easily include lightweight routines that perturb the application significantly when measured. What rules should the user then write?

If the user does not specify the rules for removing instrumentation using *tau_reduce*, TAU applies a default set (e.g., the number of calls must exceed one million and the inclusive time per call for a given routine must be less than 10 microseconds to exclude the routine). The program is then re-instrumented using the *exclude list* emitted by *tau_reduce*. To ensure that other routines that were above the threshold for exclusion before do not qualify for exclusion after re-instrumentation (due to removal of instrumentation in child routines), the user may re-generate the exclude list by re-running the program against the same set of rules. When any two instrumented executions generate no new exclusions, we say that the instrumentation *fixed-point* is reached for a given set of execution parameters (processor size, input, and so on) and instrumentation rules. The instrumentation is sufficiently coarse-grained to produce accurate measurements.

The selective, rule-based instrumentation approach implemented in TAU is a powerful methodology for performance experimentation. Users can create multiple event selection files and apply them depending on their experimentation purposes. However, there is still an issue of optimization with respect to the amount of performance data generated. This is discussed in the next section.

4 Measurement

Event instrumentation coupled with measurement code produces a "ready-to-run" performance experiment. Profile and trace measurements are the standard types used to generate performance data. Issues of instrumentation optimization regarding choice of measurement trade off detail for overhead. That is one part of the story. The other part has to do with the number of events generated versus overhead and measurement accuracy.

4.1 Measurement Choice

The overhead to generate the performance data during profiling and tracing is roughly comparable. However, because tracing produces more data, it runs the risk of additional overhead resulting from trace buffer management. Extremely high volume trace data can be produced. When this is unacceptable, one alternative is to switch the measurement method to profiling. The general point here is that the choice of measurement method is an effective means to control overhead effects, but with ramifications on the type of data acquired.

Once events have been specified for an experiment, TAU allows users to chose between profiling and tracing at runtime. The details of data produced for each event are decided both at link time and through environment settings.

4.2 Runtime Event Control

However, let us assume for the moment that only 'null' measurements are made, that is, no performance data is created and stored, but the 'instrumented' events are still detected. Since events are, in general, defined by their code location in direct instrumentation, the number of times an event occurs depends on how many times control passes through its code location. The event count is an important parameter in deciding on measurement optimization, regardless of whether profiling or tracing is used.

As the count for a particular event increases, the measurement overhead (from either profiling or tracing) for that event will accumulate. Since not all events will have the same count, the intrusions due to the overheads are distributed unevenly, in a sense, across the program and during execution. The intrusions may be manifested in different ways, and may lead to performance perturbations.

The only way to control the degree of measurement overhead is to control the event generation. That is, mechanisms must be used to *enable* and *disable* events at runtime. We call this technique *event throttling*. During program execution, instrumentation may be disabled in the program based on spatial, context, or location constraints. Spatial constraints deal with event count and frequency, context constraints consider program state, and location constraints involve event placement.

TAU allows the user to disable the instrumentation at runtime based on rules similar to the ones employed by the offline analysis of profiles using *tau_reduce*. For instance, the number of calls to each event can be examined at runtime. and when it exceeds a given user specified threshold (e.g., 100000 calls), it can be disabled [6]. This is an example of a count threshold. Disabling decisions can additionally consider measurement accuracy. For instance, if the per-call execution value for an event is below a certain threshold (e.g., 10 microseconds per call), the event is disabled. TAU disables events at runtime by adding them to the profile group (TAU_DISABLE). Subsequent calls to start or stop that event incur a minimal overhead of masking two bitmaps to determine enabled state.

Profilers based on measured profiling have timers that track routines of groups of one or more statements. A timer has a name and a profile group associated with it. A routine may belong to one or more profile groups. Performance analysis tools such as Vampir[7] and ParaProf[8] organize timer-based performance data by groups.

4.3 Group Based Control

During program execution, instrumentation may be disabled in the program based on spatial, context or location based constraints imposed. TAU provides an API for controlling instrumentation at runtime.

Logically related timers or phases may be grouped together by classifying these in a common profile group. Directory or file based association of routines is common. TAU provides a mechanism for enabling or disabling the program instrumentation based on groups. The top-level timer that is associated with main in C or C++, and the program unit in Fortran 90 belongs to a special group (TAU_DEFAULT) that is always enabled. The user may annotate the program at special points in the program based on certain conditions, to enable and/or disable instrumentation belonging to certain groups. These groups may be optionally specified on the command-line of the program as a set of groups that should be instrumented for the entire program. Limiting instrumentation based on groups, however, has the same disadvantages as knowing during program instrumentation which files or sets of routines to exclude from instrumentation.

4.4 Full Program Instrumentation Control

TAU allows the user to enable or disable all program instrumentation using the above instrumentation control API. This is useful for limiting the instrumentation (and generation of trace records) in parts of the program based on program dynamics. For regular iterative parallel applications where a program executes a sequence of iterations, it might be helpful to enable the instrumentation in a given subset. For instance, instead of enabling the instrumentation for tracing a million iterations of the program, it may be sufficient to trace the first and the last thousand iterations. The user may choose to disable the instrumentation based on the rank of a MPI process. For instance, it may be useful to limit the instrumentation for a large number of processors to only generating trace records for only one out of a hundred processors. This technique is similar to *sampling by space*[9]. TAU's trace merging and conversion utilities do not require all tasks to generate trace data. This technique cannot be applied effectively for MIMD applications where each task may have potentially different performance characteristics.

4.5 Context Based Control

TAU provides a unique depth limited instrumentation control option. A user can specify that a routines instrumentation be turned off when it executes beyond a given callpath depth. The limit may be specified as a runtime parameter. When this depth is specified as one, only the top level routine is active; at a depth of two, only the top level routine and the instrumented routine called directly by it are active and so on. When a routine executes below this threshold at some point in execution, and beyond this threshold at other points, only the former instance is recorded in the trace files. At the expense of truncating the performance information for those routines that execute beyond the given threshold, we can limit the performance data to the top few routines. The message communication events are not affected by this option.

4.6 Callpath Based Control

The KOJAK toolkit [3] includes the Expert tool that automates performance bottleneck diagnosis by examining communication events. In the analysis phase, it ignores routines that do not directly call MPI routines along a calling stack. To generate traces for Expert, it is useful to limit the instrumentation to only those routines that call an MPI routine. This is done by first configuring TAU to generate callpath profiles[2]. TAU allows a user to specify a callpath depth as a runtime parameter. All callpaths originating from a given instrumented routine, and extending to its parents are truncated when these exceed the threshold. So, the user sets a sufficiently high threshold of callpath depth so that every callpath reaches the top level routine. Then, a script parses the profile files and extracts the names of routines that directly or indirectly called an MPI routine. This list is then fed to the instrumentor as an include list, and it instruments only routines that had a calling path to the MPI routines. This technique can dramatically reduce the trace size for Expert. The drawback is that if a routine calls an MPI routine at some instances in its execution and does not invoke MPI calls at others, all of its instances are recorded, although Expert ignores those instances where it does not invoke MPI routines. This can potentially increase the trace file size and it requires a re-execution of the program with callpath profiling enabled.

4.7 Trace Based Control

It is possible to address the above problem by keeping track of all calls in an event buffer. When an MPI routine is executed, we need to examine the buffer and move trace records by eliminating those records that do not directly call the given MPI routine. This problem has a drawback that this scheme cannot work effectively with fixed size buffers that are commonly found in trace generation libraries. When a buffer overflow event takes place, all records are to be flushed to the trace file. However, if an MPI event has not taken place, it is unknown whether one will take place in the future or not. So, to preserve the trace information, we must increase the size of the trace buffer and keep processing the trace records. When it does encounter an MPI event, the trace buffer can be examined again and un-necessary instances of routines removed at runtime, and the buffers flushed to disk. This scheme does not sufficiently address the concerns as, the program could run out of memory in expanding the trace buffers and be forced to write the records to disk.

4.8 Callstack Based Control

To better address the previous requirement, TAU has introduced a callstack based runtime instrumentation control option for tracking only those instances of a routine that directly or indirectly invoke an MPI call. Trace records are generated for routines on the calling stack when an entry into an MPI routine (all MPI routines belong to a special group) is detected. When a routine entry takes place, we store the exact time it occurred on the callstack. Each routine on the callstack has a flag that indicates if it has been recorded in the trace file. When an MPI routine is started, we traverse the callstack recursively from the given routine and generate trace records if the routine has not been recorded. We stop when we encounter a routine that has been recorded. This limits the trace file to just those instances of events that are ancestors of an MPI call. By using elements of profiling and tracing together, we can better address efficient trace generation.

5 Conclusion

Parallel performance systems strive to build measurement systems as efficiently as possible. However, users can make poor instrumentation and measurement choices that lead to performance data proliferation and inaccuracies. Performance tools should support users in effective performance experimentation by providing mechanisms for optimizing instrumentation. This is true for specifying events and measurements to meet the objectives of the experiment, as well as controlling the degree of overhead and data accuracy.

The TAU performance system implements a robust set of instrumentation optimization methods. Some are discussed here. Other techniques implemented in TAU included compensation of instrumentation overhead, APIs for event grouping and control, context-based control based on callpath depths, and callstack-based control. It should be understood that all of the techniques work in parallel execution.

Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-05ER25663, DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science.

References

1. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications 14(3), 189–204 (2000)
2. Shende, S., Malony, A.D.: The TAU Parallel Performance System. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)

3. Mohr, B., Wolf, F.: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, Springer, Heidelberg (2003)
4. Lindlan, K., Cuny, J., Malony, A., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In: SC 2000 conference (2000)
5. Buck, B., Hollingsworth, J.: An API for Runtime Code Patching. Journal of High Performance Computing Applications 14(4), 317–329 (2000)
6. Malony, A.D., Shende, S., Bell, R., Li, K., Li, L., Trebon, N.: Advances in the TAU Performance System. In: Getov, V., Gerndt, M., Hoisie, A., Malony, A., Miller, B. (eds.) Performance Analysis and Grid Computing, pp. 129–144. Kluwer Academic Publishers, Dordrecht (2003)
7. Brunst, H., Kranzmüller, D., Nagel, W.: Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz. In: DAPSYS conference, pp. 93–102. Kluwer Academic Publishers, Dordrecht (2004)
8. Bell, R., Malony, A.D., Shende, S.: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 17–26. Springer, Heidelberg (2003)
9. Mendes, C., Reed, D.: Monitoring Large Systems via Statistical Sampling. In: LACSI Symposium (October 2002)

GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models

Hung-Hsun Su[1], Dan Bonachea[2], Adam Leko[1], Hans Sherburne[1], Max Billingsley III[1], and Alan D. George[1]

[1] HCS Research Lab, Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611-62001, USA
leko@hcs.ufl.edu
[2] Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720-1770, USA

Abstract. The global address space (GAS) programming model provides important potential productivity advantages over traditional parallel programming models. Languages using the GAS model currently have insufficient support from existing performance analysis tools, due in part to their implementation complexity. We have designed the Global Address Space Performance (GASP) tool interface that is flexible enough to support instrumentation of any GAS programming model implementation, while simultaneously allowing existing performance analysis tools to leverage their tool's infrastructure and quickly add support for programming languages and libraries using the GAS model. To evaluate the effectiveness of this interface, the tracing and profiling overhead of a preliminary Berkeley UPC GASP implementation is measured and found to be within the acceptable range.

1 Introduction

Parallel performance analysis tools (PATs) such as KOJAK [1] and TAU [2] have proven to be useful in tuning time-critical applications. By simplifying the instrumentation process, organizing performance data into informative visualizations, and providing performance bottleneck detection capabilities, these tools greatly reduce the time needed to analyze and optimize the parallel program under investigation. However, the majority of these tools support only a limited set of parallel programming models, focusing primarily on the message-passing model. As a result, programmers using newer parallel models are often forced to manually perform tedious and time-consuming ad-hoc analyses if they wish to optimize the performance of their parallel application.

While some work has been done in this area, the great majority of newer programming models remain unsupported by performance analysis tools due to the amount of effort that must be traditionally invested to fully support a new model. In particular, models providing a global address space (GAS) abstraction to the programmer have been gaining popularity, but are currently underrepresented in performance analysis tool support. These models include Unified Parallel C (UPC) [3], Titanium [4], SHMEM, and Co-Array Fortran (CAF) [5]. Due to the wide range of compilers and techniques used to support execution of parallel applications using the GAS model, performance analysis tool writers face many challenges when incorporating support for these models into their tools. Among these problems are the technical issues associated with instrumenting code that may be highly transformed during compilation (e.g. by parallel compiler optimizations), and the challenge of adequately instrumenting parallel applications without perturbing their performance characteristics. The latter is especially challenging in the context of GAS languages where communication is one-sided and locality of access might not be linguistically explicit, such that statically indistinguishable accesses may differ in runtime performance by orders of magnitude.

In this paper, we present a Global Address Space Performance (GASP) tool interface [6] that is flexible enough to be adapted into current GAS compilers and runtime infrastructures with minimal effort, while allowing performance analysis tools to efficiently and portably gather valuable information about the performance of GAS programs. The paper is organized as follows. Section 2 provides the background and motivation in specifying such an interface. Section 3 gives a high-level overview of the interface, and Sect. 4 presents the preliminary results of the first implementation of the GASP interface. Finally, in Sect. 5, conclusions and future directions are given.

2 Background and Motivation

The traditional message-passing model embodied by the Message Passing Interface (MPI) currently dominates the domain of large-scale production HPC applications, however its limitations have been widely recognized as a significant drain on programmer productivity and consequently GAS models are gaining acceptance [7]. By providing a shared address space abstraction across a wide variety of system architectures, these models allow programmers to express interprocess communication in a way that is similar to traditional shared-memory programming, but with an explicit semantic notion of locality that enables high-performance on distributed-memory hardware. GAS models tend to heavily emphasize the use of one-sided communication operations, whereby data communication does not have to be explicitly mapped into two-sided send and receive pairs, a tedious and error-prone process that can significantly impact programmer productivity. As a result, programs written under these models can be easier to understand than the message-passing version while delivering comparable or even superior parallel performance [8,9].

Most large-scale parallel systems employ communication hardware that requires explicit interaction with networking components in software, and consequently the compilers and libraries that support the execution of GAS programs often need to perform a non-trivial mapping to convert user-specified one-sided communication operations into hardware-level communication operations. As a result, it can be challenging to determine the appropriate location for insertion of instrumentation code to track performance data. Furthermore, the one-sided nature of GAS model communication inherently biases available information to the initiator, making it more complicated for PATs to infer the state of the communication system and observe communication bottlenecks that may be incurred on passive participants. Finally, the instrumentation process has the potential to interfere with compiler optimization that normally takes place, which may perform aggressive rearrangement and scheduling of communication.

Several instrumentation techniques are used by existing parallel PATs. Unfortunately, none of these techniques provide a fully effective approach for GAS programming models. Source instrumentation may prevent compiler optimization and reorganization and lacks the means to handle relaxed memory models, where some semantic details of communication are intentionally underspecified at source level to allow for aggressive optimization. Binary instrumentation is unavailable on some architectures of interest, and with this method it is often difficult to correlate the performance data back to the relevant source code, especially for systems employing source-to-source compilation. An intermediate library approach that interposes wrappers around functions implementing operations of interest does not work for compilers that generate code which directly targets hardware instructions or low-level proprietary interfaces.

Finally, different compilers and runtime systems may use wildly different implementation strategies (even for the same source language), which further complicates the data collection process. For example, existing UPC implementations include direct, monolithic compilation systems (GCC-UPC, Cray UPC) and source-to-source translation complemented with extensive runtime libraries (Berkeley UPC, HP UPC, and MuPC). These divergent approaches imply sufficient differences in compilation and execution such that no single existing instrumentation approach would be effective for all implementations. A naïve way to resolve this issue is to simply select an existing instrumentation technique that works for one particular implementation. Unfortunately, this approach forces the writers of performance analysis tools to be deeply versed in the internal and often fluid or proprietary details of the implementation, and can result in system-dependent tools that lack portability, to the detriment of the user experience.

It is clear that a new instrumentation approach must be found to handle these GAS models. The alternative we have pursued is to define a standardized performance interface between the compiler and the performance analysis tool. With this approach, the responsibility of adding appropriate instrumentation code is left to the compiler writers who have the best knowledge about the execution environment. By shifting this responsibility from tool writer to compiler writers, the chance of instrumentation altering the program behavior is minimized. The simplicity of the interface minimizes the effort required from the compiler writer to add performance analysis tool support to their system. Concomitantly, this simple interface makes it easy for performance analysis tool writers to add support for new GAS languages into existing tools with a minimum amount of effort.

3 GASP Interface Overview

The Global Address Space Performance interface is an event-based interface which specifies how GAS compilers and runtime systems communicate with performance analysis tools (Fig. 1). Readers are referred to the GASP specification [6] for complete details on the interface – this paper restricts attention to a high-level overview for space reasons.

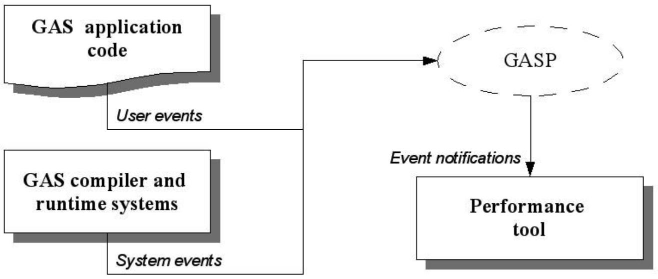


Fig. 1. High-level system organization of a GAS application executing in a GASP-enabled implementation

The most important entry point in the GASP interface is the event callback function named `gasp_event_notify` (Fig. 2), whereby GAS implementations notify the measurement tool when events of potential interest occur at runtime, providing an event ID, source code location, and event-related arguments to the performance analysis tool. The tool is then free to decide how to handle the information and what additional metrics to record. In addition, the tool is permitted to make calls to routines that are written in the source language or that use the source library to query model-specific information that may not otherwise be available. Tools may also consult alternative sources of performance information, such as CPU hardware counters exposed by PAPI [10] for monitoring serial aspects of computational and memory system performance in great detail.

The `gasp_event_notify` callback includes a per-thread, per-model context pointer to an opaque tool-provided object created at initialization time, where the tool can store thread-local performance data; the GASP specification is designed to be fully thread-safe, supporting model implementations where arbitrary subsets of GAS model threads may be implemented as threads within a single process and virtual address space.

3.1 GASP Events

The GASP event interface is designed to be highly extensible, allowing language- and implementation-specific events that capture performance-relevant information at varying levels of detail. Additionally, the interface allows tools to intercept just the subset of events relevant to the current analysis task.

```
typedef enum {
  GASP_START,
  GASP_END,
  GASP_ATOMIC,
} gasp_evttype_t;

void gasp_event_notify(gasp_context_t context,
                       unsigned int event_id,
                       gasp_evttype_t event_type,
                       const char *source_file,
                       int source_line, int source_col, ...);
```

Fig. 2. Structure of GASP event notification

A comprehensive set of events has been defined for capturing performance information from the UPC programming model and includes the following basic categories. Shared variable access events capture one-sided communication operations occurring implicitly (through shared variable manipulation) and explicitly (through bulk transfer and asynchronous communication library calls). Synchronization events, such as fences, barriers, and locks, serve to record synchronization operations between threads. Work-sharing events handle the explicitly parallel regions defined by the user. Start-up and shutdown events deal with initialization and termination of each thread. There are also collective events which capture broadcast, scatter, and similar operations, and events which capture memory management operations on the shared and private heaps.

The GASP interface provides a generic framework for the programming model implementation to interact with the performance analysis tool, and the GASP approach is extensible to new GAS models through the definition of model-appropriate sets of events. The GASP interface is also designed to support mixed-model applications whereby a single performance analysis tool can record and analyze performance information generated by each GAS model in use and present the results in a unified manner.

Finally, GASP provides facilities for user-defined, explicitly-triggered performance events to allow the user to give context to performance data. This user-defined context data facilitates phase profiling and customized instrumentation of specific code segments.

3.2 GASP Instrumentation and Measurement Control

Several user-tunable knobs are recommended by the GASP specification to provide finer control over instrumentation and measurement overheads. First, the `--inst` and `--inst-local` compilation flags are used to request instrumentation of operations excluding or including events generated by shared local accesses (i.e. one-sided accesses to local data which are not statically known to be local). Because shared local accesses are often as fast as normal local accesses, instrumenting these events can add a significant runtime overhead to the application.

By contrast, shared local access information is useful in some analyses, particularly those that deal with optimizing data locality and performing privatization optimizations, and thus may be worth the additional overhead. Instrumentation #pragma directives are provided, allowing the user to instruct the compiler to avoid instrumentation overheads for particular lexical regions of code at compile time. Finally, a programmatic control function is provided to toggle performance measurement for selected program phases at runtime.

4 Preliminary Results

A preliminary GASP implementation was added to Berkeley UPC [11] to test the effectiveness of the GASP interface. To test this implementation, we ran the UPC implementation of the NAS parallel benchmark suite version 2.4 ("class B" workload) under varying instrumentation and measurement conditions. For the CG, MG, FT, and IS benchmarks, we first compiled each benchmark using an installation of Berkeley UPC version 2.3.16 with all GASP code disabled. We used the best runtime for each benchmark as a baseline, and then recompiled each application against the same version of Berkeley UPC with GASP support enabled. We subsequently re-ran each benchmark under the following scenarios:

Instrumentation: A trivial GASP tool was linked to each benchmark that intercepted all `gasp_event_notify` calls and immediately returned. This scenario records the absolute minimum overhead that results as a consequence of GASP instrumentation being inserted into a program.
Instrumentation, local: The same trivial GASP tool used in the previous scenario was linked to each benchmark, and the `--inst-local` flag was also passed to the compiler. This scenario demonstrates the absolute minimum overhead imposed by GASP instrumentation that tracks both remote and local references in the global address space.
Measurement (profiling): We linked each benchmark against an actual performance analysis tool named Parallel Performance Wizard (PPW) [12] that records statistical information about each benchmark's runtime characteristics. In particular, the total amount of time spent executing one-sided memory operations is collected and stored relative to the source line in which the operation was initiated. This scenario does not include local data accesses in the instrumentation.
Measurement (profiling with PAPI): This scenario used the same tool as the profiling scenario, but in addition to raw temporal data the PAPI hardware counter library was used to collect the total number of cycles and number of floating-point instructions consumed by each profiled entity. This scenario does not include local data accesses in the instrumentation.
Measurement (tracing): In this scenario, the same performance analysis tool recorded a full trace of the program's activity, storing information about each UPC operation such as byte count and source/destination threads in one-sided memory operations. This scenario does not include local data accesses in the instrumentation. Additionally, this scenario includes time spent writing trace

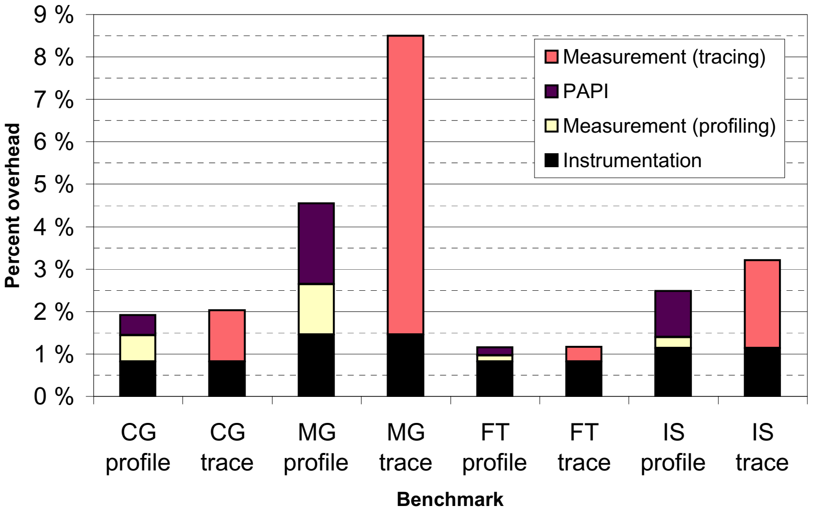


Fig. 3. Berkeley UPC GASP overhead for NAS benchmark 2.4 class B on a 32-node, 2-GHz Opteron/Linux cluster with a Quadrics QsNet^{II} interconnect

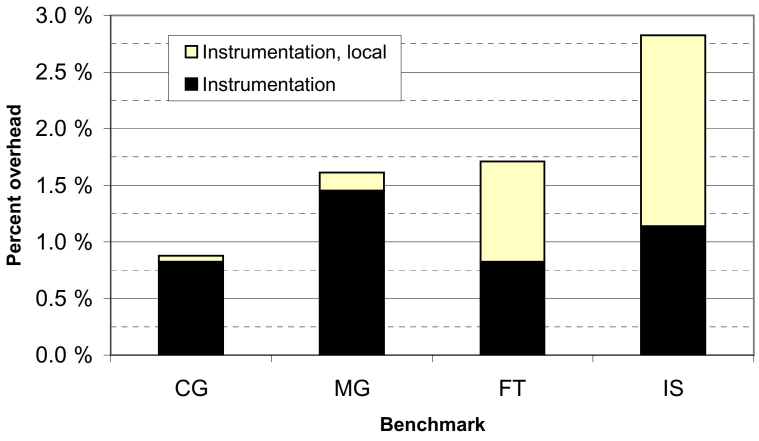
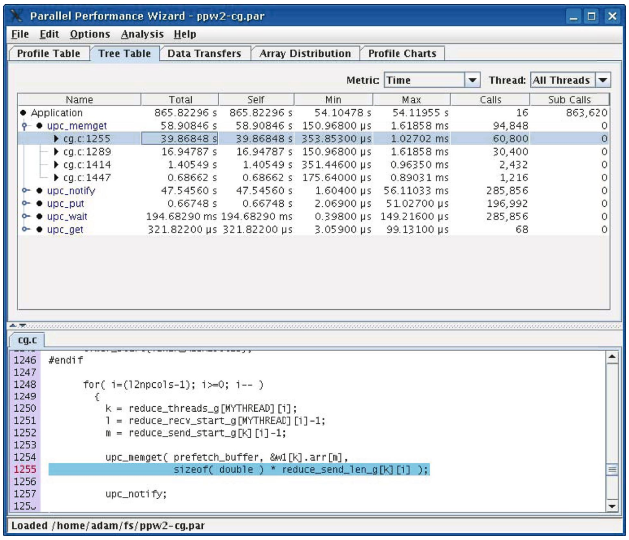


Fig. 4. Incremental raw instrumentation cost for profiling remote and local GAS accesses in the Berkeley UPC GASP implementation

data to disk using a small trace buffer periodically flushed at runtime, but not time spent during the post-execution merge phase.

Each benchmark was run under each scenario a total of ten times, and an average was used to determine the percentage increase in overall execution time against the baseline after any data outliers were discarded. The measured



Metric: Time Thread: All Threads

Name	Total	Self	Min	Max	Calls	Sub Calls
● Application	865.82296 s	865.82296 s	54.10478 s	54.11955 s	16	863,620
● upc_memget	58.90846 s	58.90846 s	150.96800 µs	1.61858 ms	94,848	0
cg.c:1255	39.86848 s	39.86848 s	353.85300 µs	1.02702 ms	60,800	0
cg.c:1289	16.94787 s	16.94787 s	150.96800 µs	1.61858 ms	30,400	0
cg.c:1414	1.40549 s	1.40549 s	351.44600 µs	0.96350 ms	2,432	0
cg.c:1447	0.68662 s	0.68662 s	175.64000 µs	0.89031 ms	1,216	0
● upc_notify	47.54560 s	47.54560 s	1.60400 µs	56.11033 ms	285,856	0
● upc_put	0.66748 s	0.66748 s	2.06900 µs	51.02700 µs	196,992	0
● upc_wait	194.68290 ms	194.68290 ms	0.39800 µs	149.21600 µs	285,856	0
● upc_get	321.82200 µs	321.82200 µs	3.05900 µs	99.13100 µs	68	0

```
1246  #endif
1247
1248        for( i=(l2npcols-1); i>=0; i-- )
1249          {
1250            k = reduce_threads_g[MYTHREAD][i];
1251            l = reduce_recv_start_g[MYTHREAD][i]-1;
1252            m = reduce_send_start_g[k][i]-1;
1253
1254            upc_memget( prefetch_buffer, &w1[k].arr[m],
1255                        sizeof( double ) * reduce_send_len_g[k][i] );
1256
1257            upc_notify;
```

Loaded /home/adam/fs/ppw2-cg.par

Fig. 5. Parallel Performance Wizard interface displaying performance data for the NPB MG benchmark

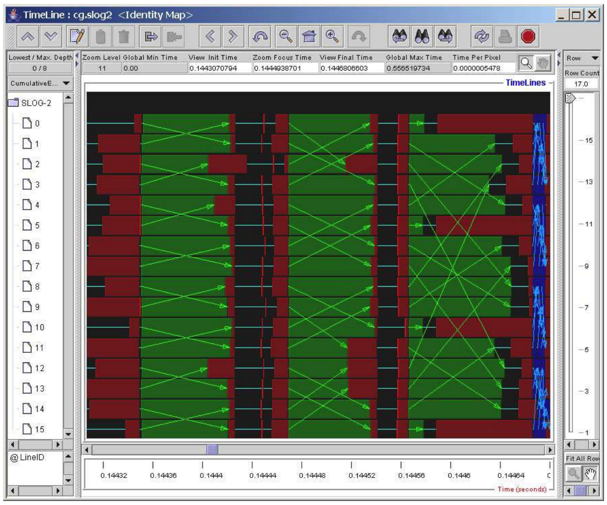


Fig. 6. Jumpshot timeline view of NPB CG benchmark

variance for the data set was low, with standard deviation peaking at less than a few percent of overall runtime for the MG benchmark.

Figure 3 presents the results from our profiling and tracing experiments, giving a breakdown of the overheads obtained from each scenario listed above. In all cases, the overhead imposed by profiling was less than 5%, and the worst overhead for tracing (which is typically more expensive than profiling due to the disk I/Os needed to capture a complete record of events) was less than 9%. These overheads are well within acceptable limits for obtaining representative performance data.

Figure 4 compares the overhead of the "Instrumentation" and "Instrumentation, local" scenarios showing the minimum incremental cost of profiling local memory accesses in addition to remote accesses. It is encouraging that the instrumentation overhead alone for each benchmark was under 1.5% and 3.0% for remote and remote+local instrumentation (respectively), even in this relatively untuned GASP implementation. This outcome shows that the overall design of GASP is sound enough to accurately capture the fine-grained performance data typically associated with GAS models.

5 Conclusions and Future Directions

This paper introduces the Global Address Space Performance (GASP) interface that specifies a standard event-based performance interface for global address space languages and libraries. This interface shifts the responsibility of where to add instrumentation code from the tool writer to the GAS compiler/library writer, which improves the accuracy of the data collection process and reduces measurement perturbation. In addition, any performance analysis tool can quickly add support for GAS models by simply defining the body of a single, generic `gasp_event_notify` function when corresponding GASP implementations are available. To evaluate the effectiveness of such an interface, a preliminary version of the GASP interface was implemented in Berkeley UPC and overhead was measured and was found to be within an acceptable range.

To further evaluate and improve the interface, we're currently working to define GASP event sets for additional GAS models and plan to integrate GASP instrumentation into several GAS implementations including the Titanium compiler and several UPC and SHMEM implementations. In doing so, we aim to encourage more compiler developers to adopt the GASP interface in their own implementations. Indeed, as of January 2007, three additional UPC compiler vendors have committed to adding GASP support in upcoming releases. In addition, we are currently developing a new comprehensive parallel performance analysis tool called Parallel Performance Wizard (PPW) that makes full use of GASP. While a full description of the tool is beyond the scope of this paper, we encourage readers to try the PPW tool for free at the PPW website [12]. Figure 5 shows a screenshot of the tool's user interface, and Fig. 6 shows our tool integrating with the Jumpshot timeline viewer via a SLOG-2 trace export to allow the user to browse a visualization of trace data. Finally, we hope to extend the GASP interface to support other parallel programming models such as MPI-2, OpenMP, Chapel, Fortress, and X10.

References

1. Mohr, B., Wolf, F.: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, Springer, Heidelberg (2003)
2. Shende, S., Malony, A.D.: TAU: The TAU Parallel Performance System. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
3. UPC Consortium: UPC Language Specifications v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208 (2005)
4. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect. Concurrency: Practice and Experience 10(11-13) (1998)
5. Numrich, B., Reid, J.: Co-Array Fortran for Parallel Programming. ACM Fortran Forum 17(2), 1–31 (1998)
6. Leko, A., Bonachea, D., Su, H., George, A.D.: GASP: A Performance Analysis Tool Interface for Global Address Space Programming Models, Specification Version 1.5. Lawrence Berkeley National Lab Tech Report LBNL-61606 (2006)
7. DARPA High Productivity Computing Systems (HPCS) Language Effort, `http://www.highproductivity.org/`
8. Bell, C., Bonachea, D., Nishtala, R., Yelick, K.: Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In: 20th International Parallel & Distributed Processing Symposium (IPDPS) (2006)
9. Datta, K., Bonachea, D., Yelick, K.: Titanium Performance and Potential: an NPB Experimental Study. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, Springer, Heidelberg (2006)
10. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications (IJHPCA) 14(3), 189–204 (2000)
11. Berkeley UPC Project: University of California at Berkeley and Lawrence Berkeley National Lab, `http://upc.lbl.gov/`
12. Parallel Performance Wizard Project: University of Florida, HCS Research Lab, `http://ppw.hcs.ufl.edu/`

Integrated Runtime Measurement Summarisation and Selective Event Tracing for Scalable Parallel Execution Performance Diagnosis

Brian J. N. Wylie[1], Felix Wolf[1,2], Bernd Mohr[1], and Markus Geimer[1]

[1] John von Neumann Institute for Computing (NIC), Forschungszentrum Jülich, 52425 Jülich, Germany
[2] Dept. Computer Science, RWTH Aachen University, 52056 Aachen, Germany
{b.wylie, f.wolf, b.mohr, m.geimer}@fz-juelich.de

Abstract. Straightforward trace collection and processing becomes increasingly challenging and ultimately impractical for more complex, long-running, highly parallel applications. Accordingly, the SCALASCA project is extending the KOJAK measurement system for MPI, OpenMP and partitioned global address space (PGAS) parallel applications to incorporate runtime management and summarisation capabilities. This offers a more scalable and effective profile of parallel execution performance for an initial overview and to direct instrumentation and event tracing to the key functions and callpaths for comprehensive analysis. The design and re-structuring of the revised measurement system are described, highlighting the synergies possible from integrated runtime callpath summarisation and event tracing for scalable parallel execution performance diagnosis. Early results from measurements of 16,384 MPI processes on IBM BlueGene/L already demonstrate considerably improved scalability.

1 KOJAK/SCALASCA Event Tracing and Analysis

The KOJAK toolset provides portable automated measurement and analysis of HPC applications which use explicit message-passing and/or implicit shared-memory parallelisations with MPI, OpenMP and PGAS [2,3]. Via interposition on library routines, preprocessing of source code directives/pragmas, or interfacing with compilers' function instrumentation, a comprehensive set of communication and synchronisation events pertinent to the execution of a parallel application can be acquired, augmented with timestamps and additional metric measurements, and logged in trace files. These time-ordered event traces from each application thread are subsequently merged into a global time-ordered trace for analysis, via automatic rating of performance property patterns or interactive time-line visualisation.

Despite the demonstrated value of the event tracing approach, using KOJAK, VAMPIR [4], DiP/Paraver [5] or commercial alternatives such as Intel Trace Collector and Analyzer, a key limitation is the trace volume which is directly proportional to granularity of instrumentation, duration of collection, and number of threads [6]. Multi-process and multi-thread profiling tools such as MPIP [13], OMPP [14] and proprietary equivalents, avoid this limitation by aggregating execution statistics during measurement and producing performance summaries at completion. A variety of tracing and profiling options have been incorporated by the TAU toolkit [7], including flat (function), specifiable-depth callpath and calldepth profiling. Measurement tools can generally exploit library interposition, via re-linking or dynamic library loading, to track MPI usage, however, implicit parallelisation with OpenMP and function tracking typically require dynamic instrumentation [8] or re-compilation to insert special instrumentation.

SCALASCA is a new project which is extending KOJAK to support scalable performance measurement and analysis of large-scale parallel applications, such as those consisting of thousands of processes and threads. Parallelisation of post-mortem trace analysis via replay of events and message transfers required reconstruction of the trace measurement and automated analysis foundation [9]. This is being complemented with runtime summarisation of event measurements into low-overhead execution callpath profiles, which will also be used to reconfigure instrumentation and measurements or direct selective event tracing. Reports from both runtime summarisation and trace analysis will use the same CUBE format, so they can also be readily combined, for presentation and investigation with the associated browser [10]. Ultimately, improved integration of instrumentation, measurement and analyses will allow each to be progressively refined for large-scale, long-running, complex application executions, with automation providing ease of use.

After describing the synergies possible from integrating runtime callpath profiling and event tracing, the design of the revised measurement system is introduced, and specific usability and scalability improvements that have been incorporated are detailed, followed by discussion of initial results with the prototype revised implementation of event tracing which demonstrate its effectiveness at a range of scales.

2 Runtime Measurement Summarisation

An approach without the scalability limitations of complete event tracing is runtime measurement summarisation. As each generated event is measured, it can be immediately analysed and incorporated in a summary for events of that type occurring on that program callpath (for that thread). Summary information is much more compact than event traces, with size independent of the total collection duration (or the frequency of occurrence of any function): it is equivalent to a local profile calculated from the complete event trace, and combining summaries produces a global callpath profile of the parallel execution.

For measurements which are essentially independent for each process, such as interval event counts from processor hardware counters, runtime summarisation can effectively capture the profile without the overhead of rendering a bulky vector of measurements. On the other hand, performance properties related to inter-process interaction, such as the time between when a message was sent and available to the receiver and its eventual receipt (i.e., “late receiver”), can only be determined in a portable way by combining disjoint measurements that is not practical at runtime. Fortunately, the inter-process performance properties are generally specialised refinements of the process-local ones available from runtime summarisation.

Doing the local analysis at runtime during measurement, and in parallel, reduces the need for large files and time-consuming post-processing and results in a timely initial overview of the execution performance.

Runtime measurement processing and summarisation also offers opportunities to decide how to deal with each event most effectively as it is generated. Frequently encountered events may be candidates to be simply ignored, due to the overhead of processing measurements for them. Other events may have a very variable cost which is typically small enough to be negligible but occasionally significant enough to warrant an explicit detailed record of their occurrence.

Alternatively, a profile summary from which it is possible to determine how frequently each function is executed, and thereby assess their importance with respect to the cost of measurement, can be used as a basis for selective instrumentation which avoids disruptive functions. Subsequent measurements can then benefit from reduced perturbation for more accurate profiling or become suitable for complete event tracing.

Runtime measurement summarisation therefore complements event tracing, providing an overview of parallel execution performance which can direct instrumentation, measurement and analysis for more comprehensive investigation.

3 Integration of Summarisation and Tracing

An integrated infrastructure for event measurement summarisation and tracing offers maximum convenience, flexibility and efficiency. Applications instrumented and linked with the measurement runtime library can be configured to summarise or trace events when measurement commences, and subsequent measurements made without rebuilding the application. It also becomes possible to simultaneously combine both approaches, with a general overview profile summary refined with analysis of selective event traces where they offer particular insight.

Along with the practical benefit of maintaining a single platform-specific measurement acquisition infrastructure, sharing measurements of times, hardware counters and other metrics avoids duplicating overheads and potential access/control conflicts. It also facilitates exact comparison of aggregate values obtained from both approaches.

Some form of runtime summarisation is probably always valuable, perhaps as a preview or compact overview. Metrics calculated from hardware counter measurements are generally most effectively captured in such summaries. Extended summaries with additional statistics calculated may be an option. Only in the rare case where the runtime overhead should be reduced to an absolute minimum is it expected that summarisation might be completely disabled.

Unless it can be readily ascertained that the application's execution characteristics are suitable for some form of event tracing, the default should be for tracing to initially be inactive. When activated, simply logging all events would provide the most complete execution trace where this was desired (and from which a summary profile could be calculated during postprocessing analysis). Alternatively, selective tracing may be based on event characteristics (such as the event type or a measurement duration longer than a specified threshold), or based on analysis from a prior execution (e.g., to filter uninteresting or overly voluminous and obtrusive measurements).

Furthermore, the availability of measurements for the entry of each new function/region frame on the current callstack, allows for late determination of whether to include them in an event trace. For example, it may be valuable to have a trace of all communication and synchronisation events, with only function/region entry and exits relevant to their callpaths (and all others discarded at runtime). The callstack and its entry measurements can be tracked without being logged until an event of interest is identified (e.g., by its type or duration), at which point the (as yet unlogged) frame entry measurements from the current callstack can be used to retroactively log its context (and mark the associated frames such that their exits will also be logged), while new frames subsequently encountered remain unlogged (unless later similarly identified for logging).

To have the most compact event traces, only message transfers need to be logged with their callpath identifier. Non-local performance properties subsequently calculated from post-mortem analysis of the traced message transfer events can then be associated with the local performance properties for the same callpaths already generated by runtime summarisation.

If desired, separate dedicated libraries for summarisation and tracing could also be provided and selected during application instrumentation preparation.

4 Implementation of Revised Measurement System

KOJAK's measurement runtime system formerly was based on an integrated event interfacing, processing and logging library known as EPILOG [11]. Files containing definitions and event records were produced in EPILOG format, and manipulated with associated utilities. Execution traces can have performance properties automatically analysed, or can be converted to other trace formats for visualisation with third-party tools.

The EPILOG name, file format and utilities are retained in the revised design of the measurement system, but only for the logging/tracing-specific component of a larger integrated summarisation and tracing library, EPIK, as shown in Figure 1. Event adapters for user-specified annotations, compiler-generated function instrumentation, OpenMP instrumentation, and the MPI library instrumentation are now generic, rather than EPILOG/tracing specific. Similarly, platform-specific timers and metric acquisition, along with runtime configuration and experiment archive management, are common EPIK utility modules. A new component, EPITOME, is dedicated to producing and manipulating totalised

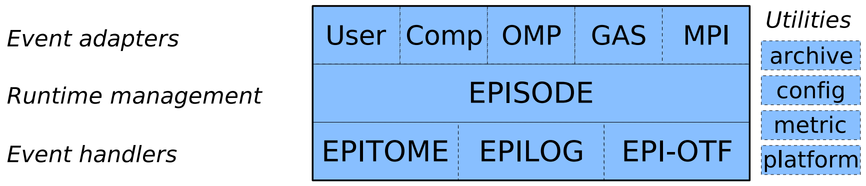


Fig. 1. EPIK runtime library architecture

measurement summaries. Both EPILOG and EPITOME share a common runtime management system, EPISODE, which manages measurement acquisition for processes and threads, attributes them to events, and determines which summarisation and/or logging subsystems they should be directed to (based on the runtime measurement configuration). Additional auxilliary event processing and output can also be incorporated as EPIK back-end event handler modules.

In addition to restructuring the measurement system, various usability and scalability improvements have been incorporated, to be able to manage measurements collected from thousands of processes.

4.1 Usability Improvements

To facilitate diverse measurement collections and analyses, and avoid the clutter of multiple (and perhaps thousands of) files appearing in the program's working directory, a new experiment archive directory structure has been introduced. A unique directory is created for each measurement execution to store all of the raw and processed files associated with that measurement and its analysis. Instead of applying each KOJAK tool to files of the appropriate type, the tools can now also accept the name of the experiment archive directory, from which the appropriate files are transparently located and into which new files are deposited. This new structure makes it easier for the tools to robustly determine and maintain the integrity of the experiment measurement/analyses, and should also be easier for users to manage (e.g., when they wish to move an experiment to a different system for storage or analysis).

On-the-fly file compression and decompression [12] reduces the size of experiment archives, with an additional bonus in the form of reduced file reading and writing times (despite additional processing overheads).

In addition to a library which can be used by external tools to read and write EPILOG traces, utilities are provided to convert traces to and from the formats used by other tools, such as VAMPIR and Paraver. Furthermore, as an alternative to post-mortem trace conversion, experimental support has been incorporated within EPIK to directly generate traces in OTF format [15].

4.2 Scalability Improvements

EPILOG traces were previously written from each thread's collection buffer into temporary files, which were merged according to event timestamp order into process rank traces and finally a global trace file at measurement finalisation. Re-reading and re-writing these large trace files was required to produce a sequential trace that the sequential analysis tool could handle. Furthermore, additional scans through each trace were required to locate the definition records interspersed within the files so that they could be globalised and written as a header in the merged file. Although this merging was often initiated automatically, and the thread stage was partially parallelised, it was a notable bottleneck in the combined measurement and analysis process which was extremely sensitive to filesystem performance. Fortunately, parallel trace analysis has no need for merged trace files, since the analysis processes read only the trace files that they require [9].

The traces written by each thread can therefore be written directly into the experiment archive, from where the subsequent analysis processes can access them. These event traces are written using process-local identifiers for regions, communicators, threads, etc., which will later need to be converted to a globally consistent set of identifiers for a unified analysis. Definitions which were previously interspersed with event records in the traces are now handled separately.

As an interim solution, these local definitions have been written to files to be unified into a global definitions file and associated local–global mapping files via postprocessing. These files are much smaller than the corresponding event traces, and can be unified quite efficiently by the separate unifier, however, creating large numbers of (small) intermediate files has been found to be inefficient.

Generation of the global definitions and associated local–global identifier mappings is required for unified analysis, and although it is a predominantly sequential operation, it is advantageous for it to be done on conclusion of the parallel measurement. Instead of each process rank writing local definitions to file(s), the buffers can be sent to rank 0 to produce a global set and associated identifier mappings. Post-mortem trace analysis requires this information to be filed along with the traces in the experiment archive, however, the runtime measurement summarisation can immediately exploit the returned identifier mappings to directly produce a unified measurement summary.

Although all of the definitions are required for the complete analysis report, identifiers do not require to be globalised when they are common to all processes or can be implied, such as the measurement metrics (time and hardware counters) and machine/node & process/thread identifiers. The remaining analysis is for callpaths, consisting of lists of region identifiers; unified analysis requires globalisation of these callpath identifiers.

Callpaths can be specified as node-region-id and parent-callpath-id records, so that only tail-segments need to be defined and callpaths are reconstructed by combining segments from each tail node via its parents' nodes to the root (which has a null parent-callpath-id). These can be added to the existing local definitions in the buffers to be unified by rank 0, or specified and unified separately.

Tracking the current callpath is part of the EPISODE functionality, whereas EPITOME maintains the set of local callpaths according to which measurements are summarised, therefore it is straightforward to provide a complete set of (local) callpaths on measurement conclusion. Provision of the global callpath set and local–global callpath identifier mappings also allows these to be used in the post-mortem trace analysis, avoiding a scan through the trace to determine the local callpaths and their subsequent unification. Even in the case where EPITOME measurement summarisation is not performed, there are still advantages from maintaining callpaths and associated visit counts.

The primary use is likely to be the use of callpath visit counts to threshold the number of times paths (events) are traced. In conjunction with the callstack of (region-entry) measurements, region enter/exit events can be tracked until an event of interest (such as a message transfer) indicates that the current buffer of events should be logged, such that uninteresting callpaths are effectively pruned from the trace. Alternatively, message transfers may be tagged with (local) callpath identifiers, allowing them to be efficiently traced in isolation from the region enter/exit events that otherwise determine callpath context. These approaches reduce the overhead of tracing frequently-executed, intrusive and/or uninteresting callpaths, making tracing and subsequent trace analysis more effective.

5 Results

To evaluate the effectiveness of some of the changes to the KOJAK measurement system, a number of tracing experiments have been performed at a range of scales with the current prototype implementation and the prior version.

Measurements were taken on the Jülicher BlueGene/L system (JUBL), which consists of 8,192 dual-core 700 MHz PowerPC 440 compute nodes (each with 512 MBytes of memory), 288 I/O nodes, and IBM p720 service and login nodes each with eight 1.6 GHz Power5 processors [16]. The parallel measurements were made on a dedicated compute partition, whereas the sequential post-processing steps ran on the lightly loaded login node.

ASC benchmark SMG2000 [17] is a parallel semi-coarsening multigrid solver, and the MPI version performs many non-nearest-neighbor point-to-point communication operations (and only a negligible number of collective communication operations). In an investigation of *weak scaling* behaviour, a fixed $64 \times 64 \times 32$ problem size per process with five solver iterations was configured, resulting in a nearly constant application run-time as additional CPUs were used: uninstrumented execution times are shown with open diamonds in Figure 2, along with a breakout of the wall time spent in the parallel solver as open triangles.

Instrumented versions of SMG2000 were prepared with the prior and current development versions of the KOJAK measurement system, and the times for running these are also shown in Figure 2: the lighter solid diamonds are the older version and the darker solid diamonds the latest version, which is more than an order of magnitude faster at the larger scales.

In each case, measurement was done using 100MByte trace buffers on each process to avoid intermediate buffer flushes to file which would otherwise seriously impact performance during measurement. The time taken by the parallel solver is the same for both versions when instrumented and measured (shown with solid right triangles) and found to be dilated less than 15% compared to the uninstrumented version (open left triangles), which is generally considered acceptable. Similarly, the number of events traced for each process is also identical in both versions, and slowly increases with the total number of processes: the crosses in Figure 2 are the mean number of events per process (in thousands), with the vertical extents corresponding to the range (which grows to more than ±50% of the mean). The aggregate number of events traced therefore increases somewhat faster than linearly, to over 40,000 million events in the 16,384-process configuration, and this manifests in the total sizes of the measurements when archived on disk. In its new compressed form, the corresponding experiment archive totals almost 230 GBytes, whereas the former uncompressed trace files are around 2.5 times larger. On-the-fly compression is a factor in the improved performance, however, the most significant gain is from avoiding trace re-writing and merging (which also makes trace writing times rather more deterministic).

The benefits of the new approach are especially evident when post-processing the traces. The prior version of the KOJAK measurement system performs a semi-parallel thread trace merging when experiments are archived, which is followed

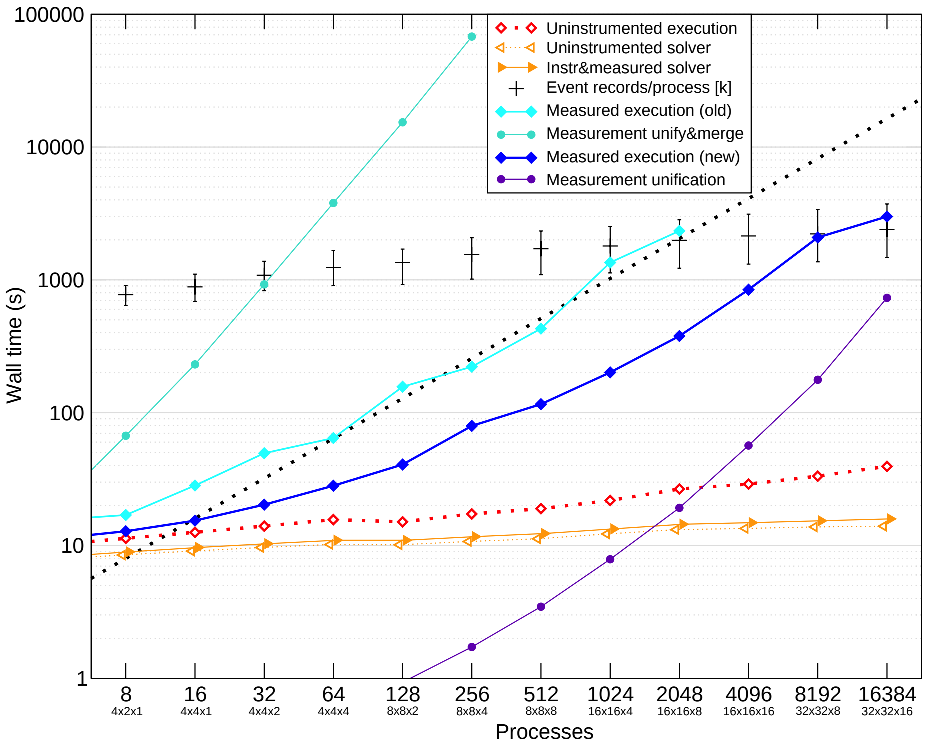


Fig. 2. SMG2000 trace collection and post-processing times with prior and current development versions of KOJAK at a range of scales on Blue Gene/L, compared with the uninstrumented execution. (Log-log scale with dotted line marking linear scaling.)

by a separate sequential process trace unification and merge: the light circles in Figure 2 show how this quickly becomes impractical for larger scales. By writing local definitions and thread event traces directly into the experiment archive, followed by a separate sequential unification of just the definition records and creation of local–global identifier mappings (the darker circles), the new version scales much more favourably.

These improvements combined demonstrate trace measurement now scaling to 16,384 processes, considerably beyond the prior practical limits. (The correspondingly more scalable analysis of the new traces is further motivation [18].)

6 Further and Future Work

Definition unification has subsequently been incorporated within the measurement finalisation, where gathering and unifying the local definitions both avoids the need to create definitions files for each process and the separate (sequential) unification step, thereby further improving measurement performance. Only when traces are produced for post-mortem analysis, is it necessary for the unified global definitions and identifier mappings to be written for each rank.

Runtime callpath summary data similarly needs to be gathered by the master process, so that it can be written as a unified report. The local callpath identifiers used by each process to store its measurements must therefore be mapped to those of the global call-tree, using the mappings returned to each process rank by the master process for this purpose. Finally, serial report writing functionality has already been refactored from the CUBE library, allowing callpath measurement data to be streamed to file as it is gathered, measurement by measurement, without it needing to be previously stored in memory in its entirety.

The runtime analysis summary reports are being formatted for presentation and investigation with the same CUBE analysis browser used for the reports produced by the former sequential and new parallel automatic event trace analysers. Direct comparison will thereby be possible using the CUBE algebra utilities [10], and will facilitate determination of instrumented functions which are problematic, due to their frequency of execution or measurement overheads. Selective instrumentation or measurement configuration can then be employed to circumvent those functions (or callpaths) in subsequent performance measurement executions, to obtain the highest quality analysis in a reliable, scalable manner.

The effectiveness of the new measurement and analysis capabilities are being evaluated on a range of HPC systems and applications, particularly at large scale, where they are also being compared with other tracing and profiling tools. Even simple operations, such as creating a separate file for each process in the same directory, can become prohibitively expensive at the largest scales, suggesting a need for exploiting the system-specific hierarchy in the structure of the measurement archive. Similarly, coordination of file writing may benefit from being adapted to the capabilities of the underlying I/O and file system.

References

1. Forschungszentrum Jüelich GmbH: SCALASCA: Scalable performance Analysis of Large-Scale parallel Applications, `http://www.scalasca.org/`
2. Forschungszentrum Jüelich GmbH (ZAM) and the University of Tennessee (ICL): KOJAK: Kit for Objective Judgement and Knowledge-based detection of performance bottlenecks, `http://www.fz-juelich.de/zam/kojak/`
3. Wolf, F., Mohr, B.: Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. J. Systems Architecture 49(10-11), 421–439 (2003)
4. Nagel, W., Arnold, A., Weber, M., Hoppe, H.-C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 63(1), 69–80 (1996)
5. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP: A Parallel Program Development Environment. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 665–674. Springer, Heidelberg (1996)
6. Wolf, F., Freitag, F., Mohr, B., Moore, S., Wylie, B.J.N.: Large Event Traces in Parallel Performance Analysis. In: Proc. 19th Int'l Conf. on Architecture of Computing Systems, Frankfurt am Main, Germany. Lecture Notes in Informatics, p. 81. Gesellschaft für Informatik, pp. 264–273 (2006)
7. Shende, S.S., Malony, A.D.: The TAU Parallel Performance System. Int'l J. High Performance Computing Applications 20(2), 287–331 (2006)
8. Cain, H.W., Miller, B.P., Wylie, B.J.N.: A Callgraph-based Search Strategy for Automated Performance Diagnosis. Concurrency and Computation: Practice and Experience 14(3), 203–217 (2002)
9. Geimer, M., Wolf, F., Knüpfer, A., Mohr, B., Wylie, B.J.N.: A Parallel Trace-Data Interface for Scalable Performance Analysis. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 398–408. Springer, Heidelberg (2007)
10. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An Algebra for Cross-Experiment Performance Analysis. In: Proc. 33rd Int'l Conf. on Parallel Processing (ICPP'04), Montreal, Quebec, Canada, pp. 63–72. IEEE Computer Society Press, Los Alamitos (2004)
11. Wolf, F., Mohr, B., Bhatia, N., Hermanns, M.-A.: EPILOG binary trace-data format, version 1.3 (2005), `http://www.fz-juelich.de/zam/kojak/doc/epilog.pdf`
12. Gailly, J., Adler, M.: zlib general-purpose compression library, version 1.2.3 (2005), `http://www.zlib.net/`
13. Vetter, J., Chambreau, C.: MPIP — lightweight, scalable MPI profiling (2005), `http://www.llnl.gov/CASC/mpip/`
14. Fürlinger, K., Gerndt, M.: OMPP — A Profiling Tool for OpenMP. In: Proc. 1st Int'l Work. on OpenMP (IWOMP) Eugene, OR, USA (2005)
15. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006)
16. The BlueGene/L Team at IBM and LLNL: An overview of the BlueGene/L supercomputer. In: Proc. SC2002, Baltimore, MD, USA. IEEE Computer Society (2002)
17. Advanced Simulation and Computing Program: The ASC SMG 2000 benchmark code (2001), `http://www.llnl.gov/asc/purple/benchmarks/limited/smg/`
18. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable Parallel Trace-based Performance Analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)

Grids for Scientific Computing: Minisymposium Abstract

Oxana Smirnova

Lund University, Sweden

Grid technologies are still evolving, with standards yet to be defined and reliable production-level solutions yet to be found. Nevertheless, Grid already stepped out of the cradle and slowly but steadily finds its way to the world of the modern information technologies. Early testers and adopters of this innovative technology are researchers in various fields of science, primarily those that traditionally require massive computational resources. Destined by the virtue of their occupation to investigate new phenomena, they provide most valuable feedback to the Grid technology developers, helping to shape the designs and define the roadmaps. Historically, researchers in High Energy Physics were the first to appreciate the Grid idea, not just as the consumers, but also as the key developers of many current solutions.

This minisymposium brings together several cases of using Grids for scientific computing. The presentations will cover the full range of scientific Grid computing activities, starting with development of core Grid software components, continuing to Grid-enabling of scientific applications, and on to providing computing infrastructure and resources. All these efforts contribute to addressing the Grand Challenges of Scientific Computing.

Roadmap for the ARC Grid Middleware

Paula Eerola[1], Tord Ekelöf[2], Mattias Ellert[2], Michael Grønager[3], John Renner Hansen[4], Sigve Haug[5], Josva Kleist[3,8], Aleksandr Konstantinov[5,6], Balázs Kónya[1], Farid Ould-Saada[5], Oxana Smirnova[1], Ferenc Szalai[7], and Anders Wäänänen[4]

[1] Experimental High Energy Physics, Institute of Physics, Lund University, Box 118, SE-22100 Lund, Sweden
oxana.smirnova@hep.lu.se
[2] Dept. of Radiation Sciences, Uppsala University, Box 535, SE-75121 Uppsala, Sweden
[3] NDGF, NORDUnet A/S, Kastruplundsgade 22-1, DK-2770 Kastrup, Denmark
[4] Niels Bohr Institute, Blegdamsvej 17, DK-2100, Copenhagen Ø, Denmark
[5] University of Oslo, Dept. of Physics, P. O. Box 1048, Blindern, NO-0316 Oslo, Norway
[6] Vilnius University, Institute of Material Science and Applied Research, Saulėtekio al. 9, Vilnius 2040, Lithuania
[7] Institute of National Information and Infrastructure Development NIIF/HUNGARNET, Victor Hugo 18-22, H-1132 Budapest, Hungary
[8] Dept. of Computer Science, Aalborg University, Fredrik Bajersvej 7E, DK-9220 Aalborg Ø, Denmark

Abstract. The Advanced Resource Connector (ARC) or the NorduGrid middleware is an open source software solution enabling production quality computational and data Grids, with special emphasis on scalability, stability, reliability and performance. Since its first release in May 2002, the middleware is deployed and being used in production environments. This paper aims to present the future development directions and plans of the ARC middleware in terms of outlining the software development roadmap.

Keywords: Grid, Globus, middleware, distributed computing, cluster, Linux, scheduling, data management.

1 Introduction

Advanced Resource Connector (ARC) [1] is a general purpose Grid middleware that provides a very reliable implementation of the fundamental Grid services, such as information services, resource discovery and monitoring, job submission and management, brokering and low-level data and resource management.

A growing number of research Grid infrastructure projects (Swegrid [2], Swiss ATLAS Grid [3], M-Grid of Finland [4], Nordic Data Grid Facility (NDGF) [5] etc.) are choosing ARC as their middleware. These resources effectively constitute one of the largest production Grids in the world, united by the common middleware base while having different operational modes and policies.

The development of the open source ARC middleware has been coordinated by the NorduGrid Collaboration [6]. This Collaboration has successfully initiated and takes an active part in several international Grid development and infrastructure projects, such as the EU KnowARC [7] and the Nordic Data Grid Facility. All such projects contribute to the ARC software. Due to these new initiatives and to the active community that has formed around the middleware, substantial development is planned and expected in coming years. This paper presents a common view on the future of the Advanced Resource Connector, incorporating the input from the major contributing projects and the community.

The roadmap contains development plans beyond the stable release version 0.6 of ARC. Relation to other middlewares, emerging standards and interoperability issues are out of scope of this paper.

2 ARC Overview

ARC middleware was created as a result of a research process started in 2001 by the Nordic High Energy Physics Community. The initial motivation was to investigate possibilities to set up a regional computational Grid infrastructure using the then existing solutions, primarily the EDG [9] prototype and the Globus Toolkit® [10]. Studies and tests conducted by NorduGrid showed that such solutions were not ready at that time to be used in a heterogeneous production environment, characteristic for the scientific and academic computing in the Nordic countries. Nordic scientific computing has a specific feature in a way that it is carried out by a large number of small and medium size facilities of a different kind and ownership, not by big supercomputing centers.

ARC was designed in 2002, having user requirements and experience with other Grid solutions in mind. An important requirement from the very start of ARC development has been to keep the middleware portable, compact and manageable both on the server and the client side.

In the stable ARC release version 0.6, the client package occupies only 14 Megabytes, it is available for most current Linux distributions and can be installed at any available location by a non-privileged user. For a computing service, only three main processes are needed: the specialized GridFTP service, the Grid Manager and the Local Information Service.

The ARC architecture is carefully planned and designed to satisfy the needs of end-users and resource providers. To ensure that the resulting Grid system stable by design, it was decided to avoid centralized services as much as possible and to identify only three mandatory components (see Figure 1):

1. The *Computing Service*, implemented as a GridFTP-*Grid Manager* pair of core services. The Grid Manager (GM) is instantiated at each computing resource's (typically a cluster of computers) front-end as a new service. It serves as a gateway to the computing resource through a GridFTP channel. GM provides an interface to the local resource management system, facilitates job manipulation, data management, allows for accounting and other essential functions.

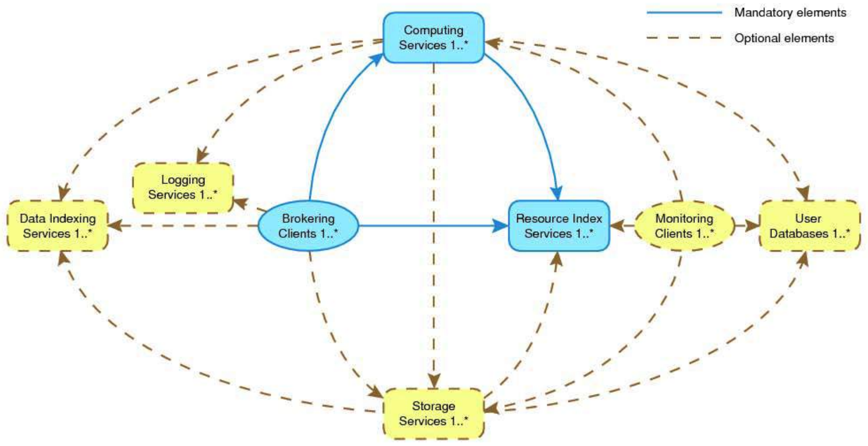


Fig. 1. Components of the ARC architecture. Arrows point from components that initiate communications towards queried servers.

2. The Information System is the basis for the Grid-like infrastructure. It is realized as a hierarchical distributed database: the information is produced and stored locally for each service (*computing, storage*), while the hierarchically connected *Index Services* maintain the list of known resources.
3. The *Brokering Client* is deployed as a client part in as many instances as users need. It is enabled with resource discovery and brokering capabilities, being able to distribute the workload across the Grid. It also provides client functionality for all the Grid services, and yet is required to be lightweight and installable by any user in an arbitrary location in a matter of few minutes.

In this scheme, the Grid is defined as a set of resources registering to the common information system. Grid users are those who are authorized to use at least one of the Grid resources by the means of Grid tools. Services and users must be properly certified by trusted agencies. Users interact with the Grid via their personal clients, which interpret the tasks, query the Information System whenever necessary, discover suitable resources, and forward task requests to the appropriate ones, along with the user's proxy and other necessary data. If the task consists of job execution, it is submitted to a computing resource, where the Grid Manager interprets the job description, prepares the requested environment, stages in the necessary data, and submits the job to the local resource management system. Grid jobs within the ARC system possess a dedicated area on the computing resource, called the *session directory*, which effectively implements limited sandboxing for each job. Location of each session directory is a valid URL and serves as the unique Job Identifier. In most configurations this guarantees that the entire contents of the session directory is available to the authorized persons during the lifetime of the Grid job. Job states are reported in the Information System. Users can monitor the job status and manipulate the jobs with the help of the client tools, that fetch the data from the Information System and forward the necessary instructions to the Grid Manager. The Grid Manager takes care of staging out and registering the data (if requested), submitting logging and accounting information, and eventually cleaning up the space used by the job. Client tools can also be used to retrieve job outputs at any time.

Figure 1 shows several optional components, which are not required for an initial Grid setup, but are essential for providing proper Grid services. Most important are the *Storage Services* and the *Data Indexing Services*. As long as the users' jobs do not manipulate very large amounts of data, these are not necessary. However, when Terabytes of data are being processed, they have to be reliably stored and properly indexed, allowing further usage. Data are stored at *Storage Elements* and can be indexed in a variety of third-party indexing databases. Other non-critical components are the *Grid Monitor* that provides an easy Web interface to the Information System, the *Logging Service* that stores historical system data, and a set of third-party *User Databases* that serve various Virtual Organisations.

3 The Roadmap

The described above solution has been in production use for many years, and in order to keep up with the growing users needs and with the Grid technology development, substantial changes have to be introduced, keeping the original fundamental design principles intact. ARC middleware development is driven in equal parts by the global Grid technology requirements and by customers' needs. Independently of the nature of the customer – an end-user or a resource owner – the guiding principles for the design are the following:

1. A Grid system, based on ARC, should have no single point of failure, no bottlenecks.
2. The system should be self-organizing with no need for centralized management.
3. The system should be robust and fault-tolerant, capable of providing stable round-the-clock services for years.
4. Grid tools and utilities should be non-intrusive, have small footprint, should not require special underlying system configuration and be easily portable.
5. No extra manpower should be needed to maintain and utilize the Grid layer.
6. Tools and utilities respect local resource owner policies, in particular, security-related ones.
7. Middleware development and upgrades must proceed in incremental steps, ensuring compatibility and reasonable co-existence of old and new clients and services during the extended transitional periods.

The long-term goal of the development is to make ARC to be able to support easy creation of dynamic Grid systems and to seamlessly integrate with tools used by different end-user communities. The main task taken care of by ARC will still be execution of computational jobs and data management, and the goal is to ease access to these services for potential users while retaining the relatively non-intrusive nature of the current ARC.

The architectural vision for future ARC development is quite traditional: it involves a limited number of core components, with computational and data management services on top. The core should be flexible enough to add general kinds of services, ranging from conventional ones for scientific computations, to more generic, like e.g. shared calendars. The resulting product should be, as before, a complete solution, ready to be used out of the box, simple to deploy, use and operate.

The post-0.6 major releases of ARC are expected to appear on yearly basis, with the version 1.0 scheduled for May 2007. Between the major releases, frequent incremental releases are planned, such that the newly introduced features and components could become available for early testers as soon as possible. Version 1.0 will introduce new interfaces for the current core ARC services and will also provide core libraries and a container for developing and hosting additional services. Version 2.0 will add numerous higher level components, such as the self-healing storage system and support for dynamic runtime environments via virtualization. Finally, with version 3.0 ARC will get extended with a scalable accounting service, enhanced brokering and job supervising service, and many other higher level functionalities.

Most new components will be developed anew, especially those constituting core services. Some existing services will be re-used or seamlessly interfaced, such as e.g. the conventional GridFTP server, VOMS or job flow control tools like Taverna [11].

While ARC versions 1.0 and above will see a complete protocol change, backwards compatibility to pre-0.6 versions will be provided for as long as reasonably needed by keeping old interfaces at the client side, and whenever necessary – at the server side as well.

3.1 ARC Version 1.0

ARC version 1.0 will capitalize on the existing in version 0.6 services and tools, and will mark preparation for transition to new core components. The following steps are foreseen, leading to ARC 1.0:

– *Standards Document*: this document will include an extended plan about how the ARC will implement essential OGF [12], OASIS [13], W3C [14], IETF [15] and other standards recommendations.
– *Architecture document*: will describe the extended new main components and services of ARC, with functionality including distributed storage, information system, virtual organization support, core libraries and the container. This architecture will mainly focus on the transformation of ARC middleware to the Web services based framework and will describe the implemented services in such a new system.

- *Web service core framework and container*: first of all, the main functions core libraries and the container will be implemented as described in the architecture document. The core framework will include such functions and methods as a common protocol (HTTP(S), SOAP, etc.) multiplexer, internal service communication channels, some part of authentication, operating system level resource management, common configuration system, logging and saving mechanism of internal states of services, etc. Meanwhile, the main service container will contain all the on-site manageable services.
- *Specifications for the Runtime Environment (RTE) Description Service and for the RTE Repository Service*: the current static and rather basic RTE support can be extended in many ways. The goal of this step is to identify the possibilities of such extensions and to define components and interfaces for the new components. The enhanced RTE system with dynamic and virtualized RTE support will be implemented in two steps.
- *Modular building and packaging framework*: current building and packaging procedures will be improved and optimized by introducing means for modular builds. Other improvements in the framework are foreseen, including transition to new version control and build systems.
- *Extended back-ends*: the reliability and robustness of the ARC middleware is the result of its very robust core back-end components. These back-ends form the layer between the Grid and the resources and have on many occasions proved superior to other resource managers in terms of performance, manageability and stability. This work item aims at further improving the performance, manageability and scalability of the ARC back-ends, adding support for additional batch systems, improving and standardising the batch system interfaces and offering better manageability and control of a Grid resource for the resource owner.
- *Initial Web service basic job management*: since job management is one of the most widely used components of current ARC, the first step to the Web service based system is to transform this component using current ARC capabilities to a Web service.
- *Web service clients*: to accommodate for transition of the job management to a Web service, relevant changes will be made in end-user clients and tools.
- *Enhanced treatment of RTEs via the Runtime Environment Description Service and RTE Repository*: after this step, the current static RTE system will be extended with a full semantic description of RTEs and with services that could help to collect and organise RTE packages and their descriptions. Simultaneously, a general framework for dynamic RTEs will become available as well.
- *Web service based elementary information index framework*: as a first step to reach one of the main goals, support for dynamic Grid creation (note also a P2P information system later), ARC will provide a Web service front-end to the current information indexing system.

3.2 ARC Version 2.0

ARC version 2.0 will see creation of new high-level components on top of the developing core services. The main foreseen steps towards this release are:

– *Taverna integration*: Taverna [11] is a workflow management system used strongly by the bioinformatics community. This step will see ARC and Taverna working together seamlessly.
– *flowGuide integration providing proof of industrial quality*: flowGuide [16] is a workflow management solution used in automotive industry. It will be adapted to be used in a Grid environment provided by ARC.
– *ARC – gLite gateway*: interoperability between ARC and gLite [17] is a top priority for a large group of traditional ARC customers. This step will enable a gateway-based solution for inter-Grid job submission.
– *Self-healing Storage: base components*: the new storage components of ARC will improve reliability and performance by incorporating automatic replication of data and indices with automatic fail-over, better integration between replica catalogues and storage elements and handling of collections of data.
– *On-site low level RTE management via RTE controller service based on virtualization*: will provide a flexible management system to automatize the RTE management on the computing resources.
– *Policy Enforcement and delegation engine implemented as a part of the container*: this step will extend the ARC security framework with even more fine grained security schemes. An enhanced access rights management framework will be developed, relying on the concept of delegation. A delegation language parser and policy engine will be selected and integrated into the ARC middleware.
– *MS Windows clients for main components*: since vast majority of end-users prefer to use Microsoft Windows for the client operating system, this step is necessary to bring Grid closer to customers, minimizing the acceptance threshold.
– *P2P-based information service*: the goal of this task is to investigate and propose novel, flexible Grid formation mechanisms which utilise the power of overlay network construction technologies of P2P networks. This work item aims to create next generation Grid information indices that fully support dynamic Grid formation. The new information backbones will be able to cope with the highly dynamical nature of Grids due to nodes unpredictably joining or leaving, the heavy load fluctuations and the inevitable node failures.
– *Improved resource control on the front-end*: quotas for data, jobs and VOs will be implemented within the container and core services.
– *Support for Extended JSDL*: ARC will be capable of dealing with complex I/O specification, dynamic RTE and authorisation policies etc.
– *Self-healing storage system with clients*: the core-level data management services will be extended to significantly reduce the manual user effort by providing a high-level, self-healing data storage service allowing the maintenance of user data (metadata) in a fault tolerant, flexible way. Based on this service the meta-data maintenance, replica management, and synchronisation

would be assisted by a Grid storage manager providing a single and uniform interface to the user.
- *Client-side developer library (next generation `arclib`)*: the existing `arclib` client-side developer library will be extended to take into account changes in the core services and addition of new higher level services.

3.3 ARC Version 3.0

ARC version 3.0 will introduce new services and extend the existing ones, using the new possibilities that will become available in course of development. It is too early to discuss the details, but one can already list some important tools and services:

- *Job migration service*: a set of services and techniques supporting migration of both queueing and running jobs.
- *Web service based monitoring*: system monitoring tools and utilities making use of Web service technologies.
- *Job supervising service*: a higher-level service capable of monitoring and eventually re-submitting jobs.
- *New brokering algorithms and services*: new brokering models will be implemented for dynamic Virtual Organisations, supporting e.g. benchmark-based brokering, push and pull models.
- *Accounting service relying on Service Level Agreements*: will implement a multi-level accounting system in a platform-independent way, based on Web service interfaces.
- *Wide range of portability of the server-side components*: will add support for operating systems currently not engaged in Grid structures, such as Solaris or Mac-OS.

4 Conclusion and Outlook

ARC middleware will see very dynamic development in coming years due to its open source approach and the ever growing developers community, that enjoys steady support from various funding agencies. The NorduGrid collaboration, that created ARC with the help of the Nordunet2 programme funding, will coordinate this development, assisted by several national and international projects.

The EU KnowARC project will be one of the major contributors, improving and extending the ARC middleware to become a next-generation Grid middleware conforming to community-based standard interfaces. It will also address interoperability with existing widely deployed middlewares. KnowARC also aims to get ARC included in several standard Linux distributions, contributing to Grid technology, and enabling all kinds of users, from industry to education and research, to easily set up and use this standards-based resource-sharing platform.

The Nordic Data Grid Facility (NDGF) project will be another major contributor to ARC development. It aims to create a seamless computing infrastructure for all Nordic researchers, leveraging existing, national computational resources and Grid infrastructures, and to achieve this, it employs a team of middleware developers, ensuring that user requirements will be met in the ARC middleware.

Other projects, such as the Nordunet3 programme, national Grid initiatives, smaller scale cooperation activities and even student projects, are also expected to provide contributions to future ARC development. With this in sight, and guided by a detailed roadmap, ARC has a perfect opportunity to grow into a popular, widely respected and used Grid solution.

References

1. Ellert, M., et al.: Advanced Resource Connector middleware for lightweight computational Grids. Future Generation Comp. Syst. 23(2), 219–240 (2007)
2. SWEGRID: the Swedish Grid testbed, `http://www.swegrid.se`
3. Gadomski, S., et al.: The Swiss ATLAS Computing Prototype, Tech. Rep. CERN-ATL-COM-SOFT-2005-007, ATLAS note (2005)
4. Material Sciences National Grid Infrastructure, `http://www.csc.fi/proj/mgrid/`
5. Nordic Data Grid Facility, `http://www.ndgf.org`
6. Ould-Saada, F.: The NorduGrid Collaboration. SWITCH journal 1, 23–24 (2004)
7. Grid-enabled Know-how Sharing Technology Based on ARC Services and Open Standards (KnowARC), `http://www.knowarc.eu`
8. EGEE gLite, gLite – Lightweight Middleware for Grid Computing, `http://glite.web.cern.ch/glite/`
9. Laure, E., et al.: The EU DataGrid Setting the Basis for Production Grids. Journal of Grid Computing 2(4), 299–400 (2004)
10. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. International Journal of Supercomputer Applications 11(2), 115–128 (1997)
11. Oinn, T., et al.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics Journal 20(17), 3045–3054 (2004)
12. Open Grid Forum, `http://www.ogf.org`
13. Organization for the Advancement of Structured Information Standards, `http://www.oasis-open.org`
14. The World Wide Web Consortium, `http://www.w3.org`
15. The Internet Engineering Task Force, `http://www.ietf.org`
16. flowGuide Workload Management Solution by science + computing ag, `http://www.science-computing.com/en/solutions/workflow_management.html`
17. Grønager, M., et al.: LCG and ARC middleware interoperability. In: Proceedings of Computing in High Energy Physics (CHEP06), Mumbai, India (2006)

Data Management for the World's Largest Machine

Sigve Haug[1], Farid Ould-Saada[2], Katarina Pajchel[2], and Alexander L. Read[2]

[1] Laboratory for High Energy Physics,
University of Bern, Sidlerstrasse 5,
CH-3012 Bern, Switzerland
sigve.haug@lhep.unibe.ch
[2] Department of Physics,
University of Oslo, Postboks 1048 Blindern,
NO-0316 Oslo, Norway
{farid.ould-saada, katarina.pajchel, a.l.read}@fys.uio.no
http://www.fys.uio.no/epf

Abstract. The world's largest machine, the Large Hadron Collider, will have four detectors whose output is expected to answer fundamental questions about the universe. The ATLAS detector is expected to produce 3.2 PB of data per year which will be distributed to storage elements all over the world. In 2008 the resource need is estimated to be 16.9 PB of tape, 25.4 PB of disk, and 50 MSI2k of CPU. Grids are used to simulate, access, and process the data. Sites in several European and non-European countries are connected with the Advanced Resource Connector (ARC) middleware of NorduGrid. In the first half of 2006 about 10^5 simulation jobs with 27 TB of distributed output organized in some 10^5 files and 740 datasets were performed on this grid. ARC's data management capabilities, the Globus Replica Location Service, and ATLAS software were combined to achieve a comprehensive distributed data management system.

1 Introduction

At the end of 2007 the Large Hadron Collider (LHC) in Geneva, often referred to as the world's largest machine, will start to operate [1]. Its four detectors aim to collect data which is expected to give some answers to fundamental questions about the universe, e.g. what is the origin of mass.

The data acquisition system of one of these detectors, the ATLAS detector, will write the recorded information of the proton-proton collision events at a rate of 200 events per second [2]. Each event's information will require 1.6 MB storage space [3]. Taking the operating time of the machine into account this will yield 3.2 PB of recorded data per year. The simulated and reprocessed data comes in addition. The estimated computing resource needs for 2008 are 16.9 PB tape storage, 25.4 PB disk storage and 50.6 MSI2k CPU.

The ATLAS experiment uses three grids to store, replicate, simulate, and process the data all over the planet : The LHC Computing Grid (LCG), the

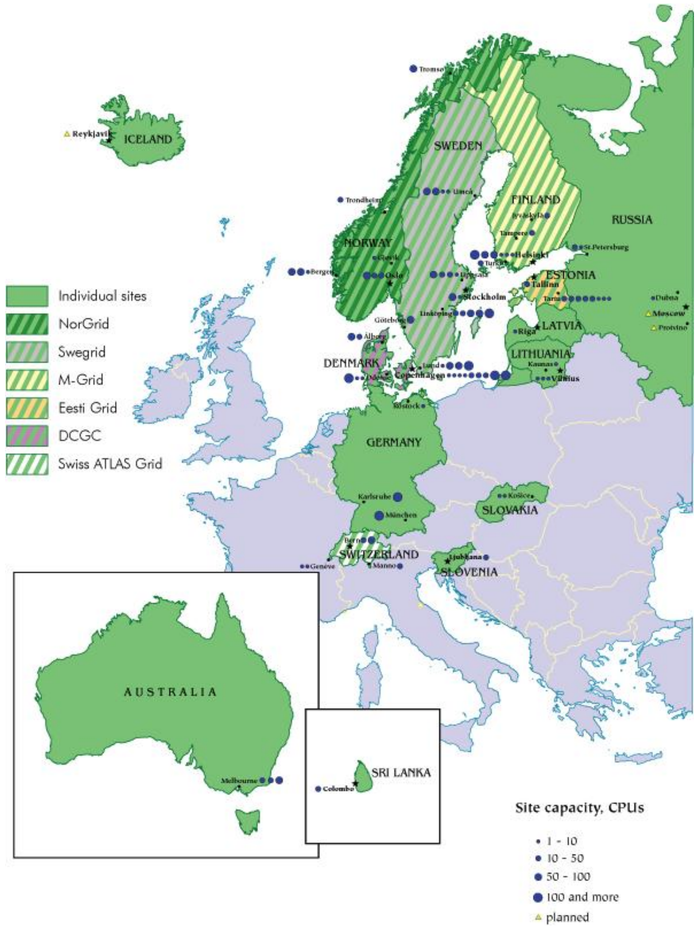


Fig. 1. Geographical snapshot of sites connected with ARC middleware (as of Dec. 2005). Many sites are also organized into national and or organizational grids, e.g. Swegrid and Swiss ATLAS Grid.

Open Science Grid (OSG), and the NorduGrid [4] [5] [6]. Here we report on the recent experience with the present distributed simulation and data management system used by the ATLAS experiment on NorduGrid. A geographical map of the sites connected by NorduGrid's middleware *The Advanced Resource Connector* (ARC) is shown in Figure 1. The network of sites which also have the necessary ATLAS software installed and thus are capable of running ATLAS computing tasks will in the following be called the ATLAS ARC Grid.

First, a description of the distributed simulation and data management system follows. Second, a report on the system performance in the period from November 2005 to June 2006 is presented. Then future usage, limitations, and needed improvements are commented. Finally, we recapitulate the performance of the ATLAS ARC Grid in this period and draw some conclusions.

2 The Simulation and Data Management System

The distributed simulation and data management system on the ATLAS ARC Grid can be divided into three main parts. First, there is the *production database* which is used for definition and tracking of the simulation tasks [7]. Second, there is the *Supervisor-Executor* instance which pulls tasks from the production database and submits them to the ATLAS ARC Grid. And finally, there are the ATLAS data management databases which collect the logical file names into datasets [8]. The Supervisor is common for all three grids. The Executor is unique for each grid and contains code to submit, monitor, postprocess and clean the grid jobs. In the case of the ATLAS ARC Grid, this simple structure relies on the full ARC grid infrastructure, in particular also a *Globus Replica Location Service* (RLS) which maps logical to physical file names [9].

The production database is an Oracle instance where job definitions, job input locations and job output names are kept. Further jobs' estimated resource needs, status, etc are stored.

The Supervisor-Executor is a Python application which is run by a user whose grid certificate is accepted at all ATLAS ARC sites. The Supervisor communicates with the production database and passes simulation jobs to the Executor in XML format. The Executor then translates the job descriptions into ARC's extended resource specification language (XRSL). Job-brokering is performed with attributes specified in the XRSL job-description and information gathered from the computing clusters with the ARC information system. In particular, clusters have to have the required ATLAS run time environment installed. This is an experiment-specific software package of about 5 GB which is frequently released. When a suitable cluster is found, the job is submitted. The ARC grid-manager on the front-end of the cluster downloads the input files, submits jobs to the local batch system and monitors them to their completion, and uploads the output of successful jobs. In this process the RLS is used to index both input and output files. The physical storage element (SE) for an output file is provided automatically by a storage service which obtains a list of potential SE's indexed by RLS. Thus neither the grid job executing on the batch node nor the Executor do any data movement and do not need to know explicitly where the physical inputs come from or where the physical outputs are stored.

When the Executor finds a job finished, it registers the metadata, e.g. a globally unique identifier and creation date, of the joboutput files in the RLS. It sets the desired grid access control list (gacl) on the files and reports back to the Supervisor and the production database.

Finally, the production database is periodically queried for finished tasks. For these the logical file names and their dataset affiliation are retrieved in order to register available datasets, their file content, state and locations in the ATLAS dataset databases. Hence, datasets can subsequentially be looked up for replication and analysis. The dataset catalogs provide the logical file names and the indexing service (from among the more than 20 index servers for the three grids of which the ATLAS computing grid is comprised) for the dataset to which the logical file is attached. The indexing service, i.e. the RLS on the ATLAS ARC Grid, provides the physical file location.

In short, the production on ATLAS ARC Grid is by design a fully automatic and light weight system which takes advantage of the inherent job-brokering and data management capabilities of the ARC middleware (RLS for indexing logical to physical filenames and storing metadata about files) and the ATLAS distributed data management system (a set of catalogs allowing replication and analysis on a dataset basis). See Reference [10] and [11] for detailed descriptions of the ATLAS and ARC data management systems.

3 Recent System Performance on the ATLAS ARC Grid

The preparation for the ATLAS experiment relies on detailed simulations of the physics processes, from the proton-proton collision, via the particle propagation through the detector material, to the full reconstruction of the particles' tracks. To a large extent this has been achieved in carefully planned time periods of operation, so-called Data Challenges. Many ARC sites have been providing

Table 1. ARC clusters which contributed to the ATLAS simulations in the period from November 2005 to June 2006. The number of jobs per site and the percentage of successful jobs are shown.

	Cluster	Number of jobs	Efficiency
1	ingrid.hpc2n.umu.se	6596	0.94
2	benedict.grid.aau.dk	5838	0.88
3	hive.unicc.chalmers.se	14211	0.84
4	pikolit.ijs.si	34106	0.83
5	bluesmoke.nsc.liu.se	9141	0.83
6	hagrid.it.uu.se	6654	0.81
7	grid00.unige.ch	624	0.79
8	morpheus.dcgc.dk	1329	0.76
9	grid.uio.no	2878	0.75
10	lheppc10.unibe.ch	3978	0.73
11	hypatia.uio.no	1542	0.70
12	sigrid.lunarc.lu.se	12038	0.70
13	alice.grid.upjs.sk	3	0.67
14	norgrid.ntnu.no	31	0.48
15	grid01.unige.ch	284	0.35
16	norgrid.bccs.no	286	0.35
17	grid.tsl.uu.se	6	0.00

Table 2. ARC Storage Elements and their contributions to the ATLAS Computing System Commissioning. Number of files stored by the ATLAS production in the period are shown in the third column. The fourth lists the total space occupied by these files. The numbers were extracted from the Replica Location Service *rls://atlasrls.nordugrid.org* on 2006-06-13.

Storage Element	Location	Files	TB
ingrid.hpc2n.umu.se	Umeaa	1217	0.2
se1.hpc2n.umu.se	Umeaa	14078	1.3
ss2.hpc2n.umu.se	Umeaa	70656	5.6
ss1.hpc2n.umu.se	Umeaa	74483	6.2
hive-se2.unicc.chalmers.se	Goteborg	10412	0.8
harry.hagrid.it.uu.se	Uppsala	38226	2.9
hagrid.it.uu.se	Uppsala	12620	1.6
storage2.bluesmoke.nsc.liu.se	Linkoping	6254	0.6
sigrid.lunarc.lu.se	Lund	14425	1.9
swelanka1.it.uu.se	Sri Lanka	1	< 0.1
grid.uio.no	Oslo	856	< 0.1
grid.ift.uib.no	Bergen	1	< 0.1
morpheus.dcgc.dk	Aalborg	252	< 0.1
benedict.grid.aau.dk	Aalborg	9426	1.3
pikolit.ijs.si:2811	Slovenia	25094	2.0
pikolit.ijs.si	Slovenia	21239	2.7
		299240	27.1

resources for these large scale production operations [12]. At the present time the third Data Challenge, or the Computing System Commissioning (CSC), is entering a phase of more or less constant production. As part of this constant production about 100 000 simulation jobs were run on ATLAS enabled ARC sites in the period from mid November 2005 to mid June 2006 where the end date just reflects the time of this report. Up to 17 clusters comprising about 1000 CPU's were used as a single resource for these jobs.

In Table 1 the clusters and their executed job shares are listed. Depending on their size, access policy, and competition with local users the number of jobs varies. In this period six countries provided resources. The Slovenian cluster, pikolit.ijs.si, was the largest contributor followed by the Swedish resources. The best clusters have efficiencies close to 90% (total ATLAS and grid middleware efficiency). This number reflects what can be expected in a heterogenious grid environment where not only different jobs and evolving software are used, but also the operational efficiency of the numerous computing clusters and storage services is a significant factor.

In Table 2 the number of output files and their integrated sizes are listed according to storage elements and locations. About 300 000 files with a total of

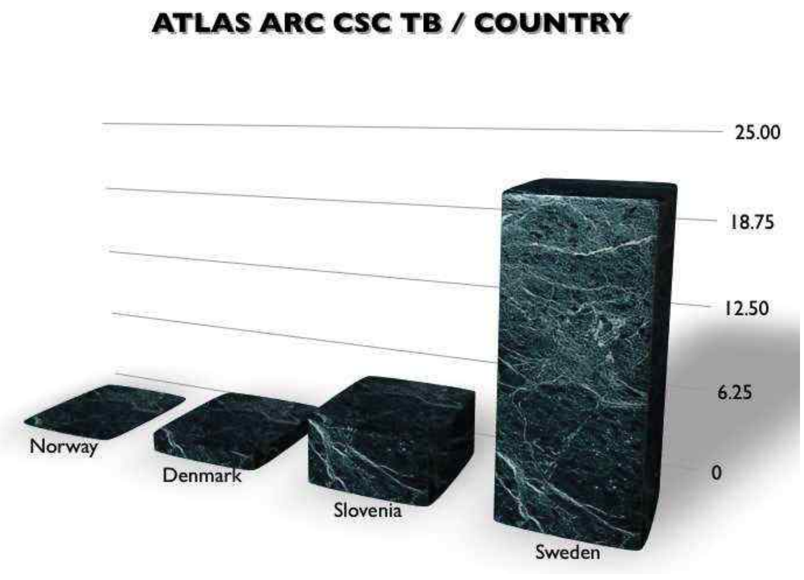


Fig. 2. TB per country. The graph visualizes the numbers in Table 2. In the period from November 2005 to June 2006 Sweden and Slovenia were the largest storage contributers to the ATLAS Computing System Commissioning. Only ARC storage is considered.

27 TB were produced and stored on disks at 11 sites in five different countries. This gives an average file size of 90 MB. The integrated storage contribution per country is shown in Figure 2. [1]

In the ATLAS production of simulated data (future data analysis will produce a different and more chaotic pattern) simulation is done in three steps. For each step input and output sizes vary. In the first step the physics in the proton-proton collisions is simulated, so-called event generation. These jobs have practically no input and output about 0.1 GB per job. In the second step the detector response to the particle interactions is simulated. These jobs use the output from the first step as input. They produce about 1 GB output per job. This output is again used as input for the last step where the reconstruction of the detector response is performed. A reconstruction job takes about 10 GB input in 10 files and produces an output of typically 1 GB. In order to minimize the number of files, it is foreseen to increase the file sizes (from 1 to 10 GB) as network capacity, disk sizes and tape systems evolve.

The outputs are normally replicated to at least one other storage element in one of the other grids and in the case of reconstruction outputs (the starting point of most physics analyses) to all the other large computing sites spread throughout the ATLAS grid. The output remains on the storage elements till a central ATLAS decision is made about deletion, most probably several years.

[1] This distribution is not representative for the previous data challenges.

Table 3. ATLAS Datasets on ARC Storage Elements as of 2006-06-13

Category	ARC	Total	ARC/Total	Description
All	739	3171	0.23	CSC + CTB + MC
CSC	489	2179	0.22	Computing System Commisioning
CTB	7	86	0.08	Combined Test Beam Production
MC	242	906	0.27	MC Production

Finally, the output files were logically collected into datasets, objects of analysis and replication. The 300 000 ATLAS files produced in this period and stored on ARC storage elements belong to 739 datasets in the period. The average number of files was then roughly 400, the actual numbers ranging from 50 to 10000. Table 3 shows the categories of datasets and their respective parts of the total numbers. The numbers in the ARC column were collected with the ATLAS DQ2 client, the numbers in the Total column with the PANDA monitor (*http://gridui02.usatlas.bnl.gov:25880/server/pandamon/query*). Since in the considered period the ATLAS ARC Grid's contribution to the total ATLAS Grid production is estimated to have been about 11 to 13%, the numbers indicate that rather shorter than average and long jobs were processed. [2]

4 Perspective, Limitations and Improvements

The limitations of the system must be considered in the context of its desired capabilities. At the moment the system manages some 10^3 jobs per day where each job typically needs less than a day to finish. The number of output files are about three times larger. In order to provide the ATLAS experiment with a significant production grid, the ATLAS ARC Grid should aim to cope with numbers of jobs another order of magnitude larger. In this perspective the ATLAS ARC Grid has no fundamental scaling limitations.

However, in order to meet the ambition several improvements are needed. First, the available amount of resources must increase. The present operation almost exhausts the existing. And since the resources are shared and with growing attraction to users, fair-sharing of the resources between local and grid and between different grid-users needs to be implemented. At the moment local users always have implicit first priority. And the grid-users are often mapped to a single local account so that they are effectively treated first-come first-serve.

Second, the crucial Replica Location Service provides the desired functionality with mapping from logical to physical file names, certificate authentication and bulk operations and is expected to be able to handle the planned scaling-up

[2] The Nordic share of the ATLAS computing resources is 7.5%, according to a memorandum of understanding.

of the system. However, the lack of perfect stability is an important problem which remains to be solved. Meanwhile, the persons running the Supervisor-Executor instances should probably have some administration privileges, e.g. the possibility to restart the service.

Third, further development should aim at some hours database independency. Both the production database and the data management databases now and then have some hours down time. This should cause problems other than delays in database registrations.

Continuous improvements in the ARC middleware ease the operation. However, in the ATLAS ARC Grid there are many independent clusters in production mode and not dedicated to ATLAS. Thus it is impractical to negotiate frequent middleware upgrades on all of them. Hence, the future system should rely as much as possible on the present features.

5 Conclusions

As part of the preparations for the ATLAS experiment at the Large Hadron Collider, large amounts of data are simulated on grids. The ATLAS ARC Grid, sites connected with NorduGrid's Advanced Resource Connector and having ATLAS software installed and configured for use by grid-jobs, now continuously contributes to this global effort.

In the period from November 2005 to June 2006 about 300 000 output files were produced on the ATLAS ARC Grid. Up to 17 sites in five different countries were used as a single batch facility to run about 100 000 jobs. Compared to previous usage, another layer of organization was introduced in the data management system. This enabled the concept of datasets, i.e. conglomerations of files, which are used as objects for data analysis and replication. The 27 TB output was collected into 740 datasets with the physical output distributed over eight significant sites in four countries.

Present experience shows that the system design can be expected to cope with the future load. Provided enough available resources, one person should be able to supervise about 10^4 jobs per day with a few GB of input and output data.

The present implementation of the ATLAS ARC Grid is lacking the ability to replicate ATLAS datasets to and from other grids via the ATLAS distributed data management tools [8] and there is no support for tape-based storage elements. These shortcomings will be addressed in the near future.

Acknowledgments. The indispensable work of the contributing resources' system administrators is highly appreciated.

References

1. The LHC Study Group: The Large Hadron Collider, Conceptual Design, CERN-AC-95-05 LHC (1995)
2. ATLAS Collaboration: Detector and Physics Performance Technical Design Report, CERN-LHCC-99-14 (1999)

3. ATLAS Collaboration: ATLAS Computing Technical Design Report, CERN-LHCC-2005-022 (2005)
4. Knobloch, J. (ed.): LHC Computing Grid - Technical Design Report, CERN-LHCC-2005-024 (2005)
5. Open Science Grid Homepage: http://www.opensciencegrid.org
6. NorduGrid Homepage: http://www.nordugrid.org
7. Goosesens, L., et al.: ATLAS Production System in ATLAS Data Challenge 2, CHEP 2004, Interlaken, contribution, no. 501
8. ATLAS Collaboration: ATLAS Computing Technical Design Report, CERN-LHCC-2005 -022, p. 115 (2005)
9. Nielsen, J., et al.: Experiences with Data Indexing Services supported by the NorduGrid Middleware, CHEP 2004, Interlaken, contribution, no. 253
10. Konstantinov, A., et al.: Data management services of NorduGrid, CERN-2005-002, vol. 2, p. 765 (2005)
11. Branco, M.: Don Quijote - Data Management for the ATLAS Automatic Production System, CERN-2005 -002, p. 661 (2005)
12. NorduGrid Collaboration: Performance of the NorduGrid ARC and the Dulcinea Executor in ATLAS Data Challenge 2, CERN-2005-002, vol. 2, p. 1095 (2005)

Meta-computations on the CLUSTERIX Grid

Roman Wyrzykowski[1], Norbert Meyer[2], Tomasz Olas[1], Lukasz Kuczynski[1], Bogdan Ludwiczak[2], Cezary Czaplewski[3], and Stanislaw Oldziej[3]

[1] Dept. of Comp. & Information Science, Czestochowa Univ. of Technology, Dabrowskiego 73, 42-200 Czestochowa, Poland
{roman, olas, lkucz}@icis.pcz.pl

[2] Poznan Supercomputing and Networking Center, Poland
{meyer, bogdanl}@man.poznan.pl

[3] Department of Chemistry, University of Gdansk, Poland
czarek@sun1.chem.univ.gda.pl

Abstract. In the first part, we present the concept and implementation of the National Cluster of Linux System (CLUSTERIX) – a truly distributed national computing infrastructure with 12 sites (64-bit Linux PC-clusters) located accross Poland. The second part presents our experience in adaptation of selected scientific applications to the cross-site execution as meta-applications, using the MPICH-G2 environment. The performance results of experiments confirm that CLUSTERIX can be an efficient platform for running meta-applications. However, harnessing its computing power needs to take into account the hierarchical architecture of the infrastructure.

1 Introduction

The National Cluster of Linux Systems, or shortly CLUSTERIX, is a truly distributed national computing infrastructure with 12 sites (local Linux PC-clusters) located accross Poland [1], [18]. These sites are connected by the Polish Optical Network PIONIER providing the dedicated 1 Gb/s bandwidth. Although the CLUSTERIX grid offers potentially large performance, the key questions facing computational scientists is how to effectively adapt their applications to such a complex and heterogeneous architecture.

Efficiently harnessing computing power of grids requires the ability to match requirements of applications with grid resources. Challenges in developing grid-enabling applications lie primarily [6] in the high degree of system heterogenity and dynamic behavior of grids. For example, communication between computers of the same site, connected by a high bandwidth and low latency local network, is much faster than communication between nodes of different sites provided by a wide area network characterized by a much high latency.

It makes programming HPC applications on grids a challenging problem [2], [14]. An important step in this direction is emergence of scientific-application-oriented grid middleware, such as MPICH-G2 [10] that implement the well-established MPI standard on top of grid infrastructure based on the Globus

Toolkit [4]. The MPICH-G2 environment significantly spares computational scientists from low-level details about communication handling, network topology, and resource management. However, in spite of these achievements, the development of efficient algorithms for large-scale computational problems that can exploit grids efficiently still remains an exceptionally challenging issue.

This paper presents our experience in adaptation of existing scientific applications to the CLUSTERIX grid environment. The proposed solution allows for running tasks accross several local clusters as meta-applications, using the MPICH-G2 middleware.

The paper is organized as follows. In Section 2, we shortly describe the architecture of the CLUSTERIX grid. How appplications are executed in the CLUSTERIX environment, it is presented in Section 3, while Section 4 is devoted to running meta-applications in CLUSTERIX using the MPICH-G2 tool. Section 5 presents two pilot meta-applications, as well as corresponding performance results. Conclusions are given in Section 6.

2 CLUSTERIX Grid Project

The main objective of the CLUSTERIX project [1], [18] is to develop mechanisms and tools that allow for deployment of a production grid. The CLUSTERIX infrastructure is a distributed PC-cluster (or meta-cluster) with 12 dedicated, local Linux clusters with 64-bit machines, connected via dedicated 1 Gb/s channels. The infrastructure has been tested on a set of pilot applications developed as a part of the project. The project has been implemented by 12 Polish partners, with Czestochowa University of Technology as a coordinator.

At this moment, the CLUSTERIX backbone includes 196 Intel Itanium2 processors (1.4 GHz, 3 MB cache) in 98 computational nodes. Each node (two-way SMP) is equipped with 4 GB or 8 GB RAM. The communication VLAN, using Gigabit Ethernet or InifiniBand, supports the message exchange between nodes, while connection of nodes to the NFS server is provided through the second VLAN based on Gigabit Ethernet. While users' jobs are allowed to be executed only on computational nodes, each local cluster is equipped with an access node (32-bit machine), where the local queuing system (currently OpenPBS) and components of CLUSTERIX middleware are running. All machines inside a local cluster are protected by a firewall, which is also used as a router for attachement of so-called dynamic clusters [7].

The grid middleware for CLUSTERIX has been developed as Open Source, and is based on the Globus Toolkit 2.4 and Web Services, with Globus 2.4 available in the Globus 3.2 distribution [4]. From the point of view of running applications in CLUSTERIX, the key components of the middleware are (Fig.1): (i) Grid Resource Management System (GRMS), which is CLUSTERIX metascheduler [12]; (ii) CLUSTERIX monitoring system JIMS [16]; (iii) CLUSTERIX Data Management System (CDMS) [11]; (iv) Virtual User System (VUS)[8].

Jobs are submitted to local batch systems, through the Globus facilities, using GRMS developed in the GridLab project [5]. The additional functionality of

GRMS developed for CLUSTERIX include: (i) prediction module, (ii) support for MPICH-G2, (iii) cooperation with JIMS and CDMS.

An important element of the CLUSTERIX backbone is the data storage system managed by CDMS, which has been developed in the project based on the analysis of users' requirements. When designing CDMS, a special attention has been paid to making the system high productive, secure, and user-friendly [11]. Before execution of an application, input data are fetched from storage elements and transferred to access nodes; after the execution output data are returned from access nodes to storage elements. Currently each storage element is equipped with 2 TB HDD.

CLUSTERIX is expecting to be a highly dynamic grid with many virtual organisations (VOs) and hundreds of users. The standard Globus authorisation mechanism requires configuring access for each grid user on each local cluster. This solution is neither scalable and nor sufficient. There is a need for authorisation based on the VO membership, so the administration burden is divided between resource administrator and VO manager. For that purpose, VUS middleware was developed. The main part of VUS is an authorisation subsystem, that replaces the standard Globus authorisation and it is located on each cluster access node. The basic authorisation is done by querying VO services (one service per VO) for their members [9]. VUS comprises of set of "virtual"-generic accounts (Linux accounts on cluster nodes), that are assigned to grid users for the time of their tasks. It is assured that only one user is logged on a account at the time, then the account may be assigned to another user, but the history of assignments is stored, e.g., in order to collect accounting or trace users actions.

3 Running Applications in CLUSTERIX Environment

The key role in running applications in CLUSTERIX plays the Grid Resource Management System – GRMS [12]. That is why, to avoid a single point of failure several instances of GRMS have to be launched. The main responsibility of this system is management of the whole process of remote submitting computational jobs to different batch systems in local clusters. The main functionality of GRMS include:

- ability to choose the best resource for the job execution, according to the job description;
- job suspending, resuming and migrating;
- providing access to the job status, and other information about jobs, e.g., name of host where the job is/was running;
- transferring input and output files.

The standard way of executing applications in the CLUSTERIX environment is shown in Fig.1. The user describes a job to be executed using the Internet portal. It generates and sends to GRMS a special file containing the job description as an XML document in the GRMS Job Description (GJD) format. GRMS chooses resources for the job execution (in one or more local clusters) which

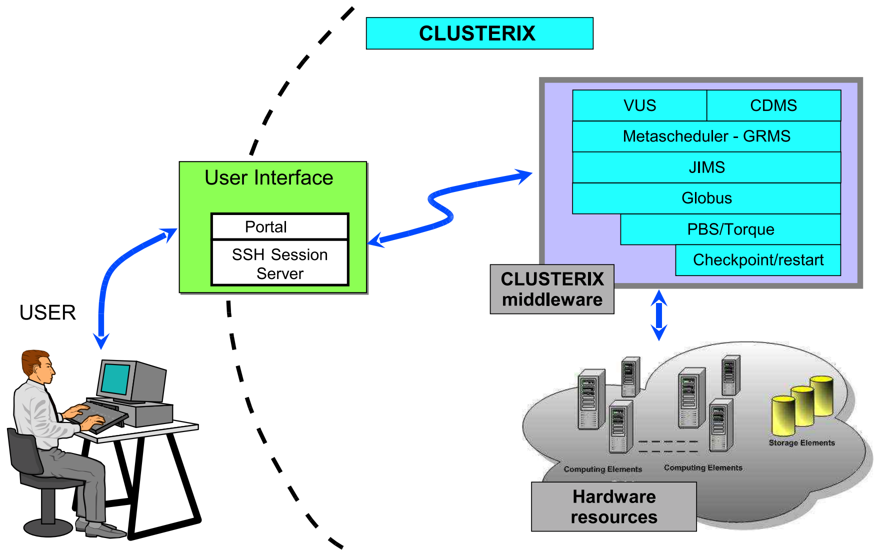


Fig. 1. Execution of applications in CLUSTERIX

satisfy the user's requirements. The job is then submitted to queuing systems in the local clusters, using the Globus facilities.

To be executed, the job needs to have access to Linux accounts on cluster nodes. In CLUSTERIX, VUS middleware is responsible for assignment of these accounts to the user only during the job execution. After transferring input data from CDMS, the job is executed on computational nodes of chosen local clusters. CDMS is also responsible for copying output files created as as result of the job execution. There is possibility to notify the user about success or failure of the job execution, using e.g. e-mail or sms.

The job description in the GJD format (Fig.2) is utilized by the user to express all the information necessary to handle the job, such as: resource requirements, location of executables and input/output files, etc. Each GRMS Job Description begins with <grmsjob> tag, containing "appid" attribute. This attribute serves as an application identifier, and is arbitrarily chosen by the user. There are two main elements in the job description: specification of the application executables, and description of resource requirements.

The <simplejob> element describes the main job to be executed by GRMS. This description specifies the application executables together with a set of parameters (given by <executable> element), as well as resource requirements (given by <resource> element). The <executable> element possesses an additional, "type" attribute which determines whether a single instance of the application (type="single") or multiple instances of the applications (type= "multiple") should be executed. When "type" is set to "mpi", it means that an MPI application will be executed; in this case an additional, "count" attribute

```
<grmsjob appid="psolidify">
  <simplejob>
    <resource>
      <localrmname>pbs</localrmname>
      <hostname>access.pcss.clusterix.pl</hostname>
    </resource>
    <executable type="mpi" count="8">
      <file name="exec" type="in">
        <url>gsiftp://access.wcss.clusterix.pl/~/myapp/psolidify</url>
      </file>
      <arguments>
        <value>250000.prl</value>
        <file name="250000.prl" type="in">
         <url>gsiftp://access.wcss.clusterix.pl/~/data/250000.prl</url>
        </file>
      </arguments>
      <stdout>
        <url>gsiftp://access.wcss.clusterix.pl/~/app1.out</url>
      </stdout>
    </executable>
  </simplejob>
</grmsjob >
```

Fig. 2. An example of MPI job description in the GJD format: the application is executed on 8 processors from the local cluster in Poznan, while input data and executable are transferred from the local cluster in Wroclaw, using GridFTP

specifies the required number of processors. The description of executables can also includes: locations of executables and input/output files (using <file> elements), input arguments (<arguments>), redirection of standard and error ouputs (<stdout> and <stderr> respectively), etc.

The <resource> element gives us possibility to determine resource requirements of the application. Among them are: name of machine on which the job should be executed (<hostname>), operating system (<ostype>, <osname>), type of requred batch system (<localrmname>), minimal memory (<memory>), minimal speed of processors (<cpuspeed>), network parameters (<bandwidth>, <latency>, ...), etc.

4 Meta-applications in CLUSTERIX

4.1 Using MPICH-G2

In the CLUSTERIX project, the MPICH-G2 middleware [10] is used as a grid-enabled implementation of the MPI standard. It allows for running multilevel parallel applications across many sites (local clusters) [6], [14]. MPICH-G2 extends the MPICH software to use Globus-family services. To improve performance, we use MPICH-based vendor implementations of MPI in local clusters.

CLUSTERIX has a hierachical architecture, with respect to both the memory access and communication. Inside SMP nodes, data are exchanged between processors through shared memory. SMP nodes are grouped into local clusters, and communications inside them are implemented using such network protocols as Gigabit Ethernet, Myrinet or InfiniBand. They are characterized by high bandwidths and small latencies (especially Myrinet and InfiniBand). Finally, local clusters connected by WAN are building blocks for the entire meta-cluster.

Taking into account the hierarchical architecture of CLUSTERIX, it is not a trivial task to adapt the existing applications for effective use in the meta-cluster. It requires parallelization on several levels corresponding to the meta-cluster architecture, taking into account the high level of heterogeneity in network performance between various subsystems of the meta-cluster (Table 1). In particular, there is a quite complex problem of minimizing the influence of less efficient networking between local clusters on the efficiency of calculations.

Table 1. Hierarchical architecture of CLUSTERIX

	latency	bandwidth	# processors
single node (MPICH-G2)		5.4 Gb/s	2
local cluster (vendor MPI)	104 μs	752 Mb/s	6 – 32
local cluster (MPICH-G2)	124 μs	745 Mb/s	6 – 32
meta-cluster (MPICH-G2)	10 ms	33 Mb/s	up to 196

4.2 Running Meta-applications in CLUSTERIX

To execute an MPICH-G2 application in CLUSTERIX, an adequate job description in the GJD format (Fig.3) is issued directly or using the portal. By setting type="mpichg2" in <executable> element, the application is recognized as an MPICH-G2 job. The subsequent, "count" attribute specifies the total number of processors required for the job execution.

In general, GRMS will find the best available clusters, which meet application requirements and whose total number of free computational nodes is not less the number of nodes requested in job description. Then the MPICH-G2 job will be submitted to those clusters by means of the Globus GRAM protocol and Globus DUROC. GRMS will split the job into subsets of MPI processes and submits every subset to the Globus gatekeeper on separate cluster. The gatekeepers passes the job's processes to the underlying cluster's queuing system, which will decide on which internal nodes they would be executed. So, GRMS has no influence on process-to-node mapping. It can only control the number of processes submitted to the chosen cluster.

GRMS acts in order to ensure the best jobs to cluster mapping, i.e. it aims to provide reliable jobs execution and resources usage leveling. To achieve that, GRMS needs valid and detailed information about the current state of the whole computational environment: the resources load levels, clusters (and queues in local queuing systems) with free nodes, etc. This is why GRMS has to cooperate

```
<grmsjob appid="psolidify-mpichg2" persistent="true">
  <simplejob>
    <resource>
      <hostname tileSize="4">access.pcss.clusterix.pl</hostname>
      <localrmname>pbs</localrmname>
    </resource>
    <resource>
      <hostname tileSize="8">access.pb.clusterix.pl</hostname>
      <localrmname>pbs</localrmname>
    </resource>
    <executable type="mpichg" count="2">
      <file name="clxintel" type="in">
        <url>cdms:///myapp/psolidify</url>
      </file>
      <arguments>
        <value>250000.prl</value>
        <file name="250000.prl" type="in">
         <url>cdms:///data/250000.prl</url>
        </file>
      </arguments>
      <stdout>
        <url>cdms:///app1.out</url>
      </stdout>
    </executable>
  </simplejob>
</grmsjob >
```

Fig. 3. An example of MPICH-G2 job description: (i) the application is executed on 4 processors from cluster in Czestochowa, and 8 processors from cluster in Poznan; (ii) CDMS is used to transfer files, instead of GridFTP

closely with infrastructure monitoring service, and why GRMS resource management quality depends on quality of information delivered by that service.

Other possibility that GRMS offers to its user is to specify directly in job description how many MPI processes shall be submitted to the given clusters. But in such case GRMS can not guarantee the proper job execution, e.g., some processes may be queued in local systems for too long and the application may fail due to timeouts.

The load partitioning among separate local cluster is carried out by multiple placement of <resource> element in the job description. How many processors should be utlized for the job execution on each cluster, it is specified by "tileSize" attribute of <resource> element. The sum of all the "tailSize" attributes nust be equal to the value of "count" attribute in <executable> element.

If user does not specify <resource> elements, then GRMS will pick the best resources basing on its internal algorithms. Similarly, if there are <resource> elements defined, but they do not define "tileSize" attributes, then GRMS will submit the job only to specified clusters, but it will decide about process distribution based on its algorithms and the available knowledge about the computational environment.

5 Testing Meta-applications

5.1 FEM Modelling of Castings Solidification

NuscaS is an object-oriented package for the FEM modeling, designed [17] at Czestochowa University of Technology to investigate thermomechanical phenomena. Its functionality includes also implementation on clusters. *NuscaS* is one of pilot applications adapted for execution in the CLUSTERIX environment.

In our tests, the problem geometry was meshed with 40613, 80401, 159613, 249925, 501001, and 750313 nodes. It should be noted that both in the sequential and parallel cases, the average from ten runs was used to estimate the sequential T_1 and parallel T_p execution times, where p is the number of processors. For each run, the time necessary to solve a system of linear equations using the conjugate gradient algorithm (CG) was measured.

Fig.4 presents the performance results achieved for a single site in Poznan, as well as two distant local clusters located evenly in Czestochowa and Poznan. In the single-site case, in spite of using only the Gigabit Ethernet the results of experiments are very good since the speedup is almost ideal. For the cross-site execution, the results could be considered as rather promising. For example, the speedup for the mesh with 750313 nodes is $S_p = 15.05$, for $p = 18$ processors (9 from Poznan, and 9 from Czestochowa). However, for small meshes the cross-site communication overheads becomes more significant, and decreases the speedup.

When adapting *NuscaS* to the hierarchical architecture of CLUSTERIX, the key points are: (i) to choose a modified version of the CG algorithm with only one synchronization point [13], which allows for reduction in the number of cross-site messages; (ii) to take advantage of overlapping of computation and communication when implementing the sparse matrix-vector multiplication in parallel [17].

For large meshes, the obtained values of speedup/efficiency are satisfactory for practical needs. At the same time, the execution of relatively small problems (e.g., with 40613 nodes) as meta-aplications is not reasonable because of too large cross-site communication overheads. For such problems, the use of resources of a single local cluster is sufficient in practice.

The comparison of the cross-site and single-site performance is shown in Fig.5. This comparison shows a loss in speedup when more than one local cluster is used. This negative effect is decreasing with the growth of the mesh size.

5.2 Prediction of Protein Structures

Proteins are macromolecules which are absolutely necessary for functioning of all known living organisms. Each protein has a unique 3D structure which determines its functions. The theoretical prediction of 3D structures of proteins

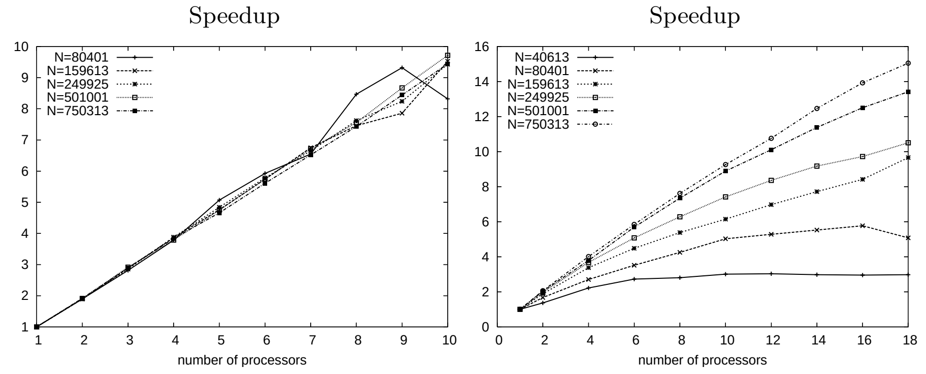


Fig. 4. Speedup for different mesh size versus number of processors achieved on a single cluster (on left hand) and two distant local clusters (on right hand)

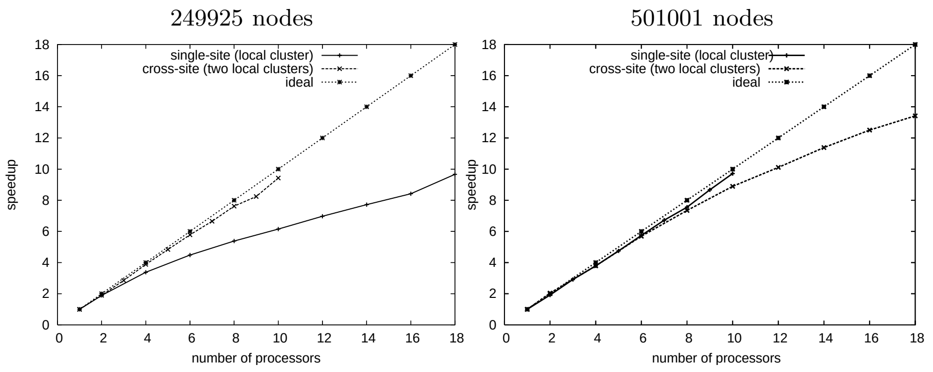


Fig. 5. Comparison of single-site and cross-site performance for two meshes with 249925 and 501001 nodes

solely from its sequence is a grand challenge of computational structural biology and theoretical chemistry.

Successful application of protein structure prediction methods based on energetic criteria depends on both an adequate approximation of the energy function, and an effective computational approach to global optimization of the conformational energy [15]. It is practically unfeasible to search conformational space with an all-atom potential function. In the UNRES force field, each amino-acid residue is represented by two interaction sites, namely the united peptide group and the united side chain. The UNRES force field is based on a cumulant expansion of the restricted free energy function of a polypeptide chain. One of the most effective procedures for the global optimization of protein structures is conformational space annealing (CSA), a hybrid global optimization method which combines genetic algorithm and a local gradient-based minimization.

Prediction of 3D structures of proteins, using CSA as a global optimization method and the UNRES force field, requires large computational resources. The CSA algorithm is applicable to large-scale parallel computing, and in this work it has been extended to the CLUSTERIX grid architecture using the MPICH-G2 tool. Parallelization of the CSA method by a master/worker approach has been enhanced by removing most of the synchronization steps, which makes it perform well also on hierarchical architectures such as in CLUSTERIX, and greatly enhances scalability. On hierarchical architectures slower communication between some workers and the master only slightly reduces efficiency of the code.

The results of benchmark simulations are shown in Table 2. It contains execution times for different number *p* of processors, taken from either a single local cluster in the Computing Center TASK (Gdansk), or different number of distant clusters located in TASK, Czestochowa (PCz), Wroclaw (WCSS) and Bialystok (PB). Comparison of data from single local cluster and two distant clusters shows only slight increase of execution times caused by slow cross-site communications. In case of three distant clusters there is a significant difference between execution times for 4+4+4 and 6+3+3+0, which can be explained in terms of a load-balancing problem for unequal distribution of work between different clusters.

Table 2. Performance results for single-site and cross-site execution of prediction of protein structures

TASK		PB+PCz		TASK+PB+PCz	
p	time[s]	p	time[s]	p	time[s]
2	5394	1+1	5483		
4	1752	2+2	1837	2+1+1	2083
8	767	4+4	777	4+2+2	1013
12	476	6+6	500	6+3+3	616
16	351			8+4+4	456
32	174			20+6+6	199

TASK+PB+PCz+WCSS	
p	time[s]
0+6+6+0	495
6+6+0+0	496
6+0+6+0	491
6+0+0+6	503
0+6+0+6	496
0+0+6+6	497
4+4+4+0	500
0+4+4+4	505
4+0+4+4	512

6 Conclusions and Further Works

This paper presents our experience in adaptation of selected scientific applications to their cross-site execution as meta-applications on the CLUSTERIX grid, using the MPICH-G2 environment. The performance results of numerical experiments with FEM modeling of castings solidification and prediction of protein structures confirm that CLUSTERIX can be an efficient platform for running numerical meta-aplications. However, harnessing its computing power is not a trivial task, and first of all needs to take into account the hierachical architecture of the infrastructure.

The performance results presented in this paper leave considerable room for further improvements in meta-applications scalability, especially in case when more than two CLUSTERIX local clusters are engaged in computations. In particular, a novel method for mapping FEM calculations onto meta-cluster architecture is under development in *NuscaS*. This method is based on using a two-level scheme of partitioning of FEM calculations, that gives way to match local clusters engaged in computations. Such an approach allows for further reduction in the number of cross-site messages.

References

1. CLUSTERIX Project Homepage, `http://www.clusterix.pl`
2. Boghosian, B., et al.: Nektar, SPICE and Vortonics: Using Federated Grids for Large Scale Scientific Applications. In: Workshop on Challenges of Large Applications in Distributed Environment, IEEE Catalog Number: 06EX1397, pp. 34–42 (2006)
3. Czaplewski, C., et al.: Improved conformational space annealing method to treat β-structure with the UNRES force-field and to enhance scalability of parallel implementation. Polymer 45, 677–686 (2004)
4. Globus Toolkit Homepage, `http://www.globus.org/toolkit`
5. GridLab: A Grid Application Toolkit and Testbed, `http://www.gridlab.org`
6. Dong, S., Karniadakis, G.E., Karonis, N.T.: Cross-site Computations on the TeraGrid. Computing in Science & Engineering 7(5), 14–23 (2005)
7. Frankowski, G., et al.: Integrating dynamic clusters in CLUSTERIX environment. In: Cracow Grid Workshop'05 Proceedings, pp. 280–292 (2006)
8. Jankowski, M., Wolniewicz, P., Meyer, N.: Virtual User System for Globus based grids. In: Cracow Grid Workshop'04 Proceedings, pp. 316–322 (2005)
9. Jankowski, M., Wolniewicz, P., Meyer, N.: Practical Experiences with User Account Management in Clusterix. In: Cracow Grid Workshop'05 Proceedings, pp. 314–321 (2006)
10. Karonis, N., Toonen, B., Foster, I.: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. Journal of Parallel and Distributed Computing 63(5), 551–563 (2003)
11. Kuczynski, L., Karczewski, K., Wyrzykowski, R.: CLUSTERIX Data Management System and Its Integration with Applications. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 240–248. Springer, Heidelberg (2006)
12. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Grid Multicriteria Job Scheduling with Resource Reservation and Prediction Mechanisms. In: Jozefowska, J., Weglarz, J. (eds.) Perspectives in Modern Project Scheduling, pp. 345–373. Springer, Heidelberg (2006)
13. Meisel, M., Meyer, A.: Hierarchically Preconditioned Parallel CG-Solvers with and without Coarse-Mesh Solvers Inside. SFB399-Preprint 97-20, Technische Universitat Chemnitz (1997)
14. Mirghani, B.Y., et al.: Development and Performance Analysis of a Simulation-Optimization Framework on TeraGrid Linux Clusters. In: Proc. LCI Int. Conf. on Linux Clusters: The HPC Revolution 2005, Chapel-Hill, NC (2005)
15. Scheraga, H.A., et al.: The protein folding problem: Global optimization of force fields. Frontiers in Bioscience 9, 3296–3323 (2004)

16. Wojtas, K., Wasilewski, L., Balos, K., Zielinski, K.: Discovery Service for JMX-Enabled Monitoring System –JIMS Case Study. In: Cracow Grid Workshop'05 Proceedings, pp. 148–157 (2006)
17. Wyrzykowski, R., Olas, T., Sczygiol, N.: Object-Oriented Approach to Finite Element Modeling on Clusters. In: Sørevik, T., Manne, F., Moe, R., Gebremedhin, A.H. (eds.) PARA 2000. LNCS, vol. 1947, pp. 250–257. Springer, Heidelberg (2001)
18. Wyrzykowski, R., Meyer, N., Stroinski, M.: Concept and Implementation of CLUSTERIX: National Cluster of Linux Systems. In: Proc. LCI Int. Conf. on Linux Clusters: The HPC Revolution 2005, Chapel-Hill, NC (2005)

Simulations of Materials: Minisymposium Abstract

Lars Nordström

Uppsala University, Sweden

Computer simulations of materials, or computer experiments, are strongly on the way to become an alternative to experiments in the strive to understand material properties. A basic ingredient in dealing with materials properties is their electronic structure. The methods to calculate the electronic structures from first principles (density functional methods) become more efficient and accurate, and are reaching a predict power with a large pace. With these accurate electronic structures one can calculate parameters, for example, molecular and spin dynamics.

This minisymposium presents work on developments of methods as well as applications of materials simulations.

Ab Initio Calculations of the Electronic Structure and Magnetism of Iron Porphyrin-Type Molecules: A Benchmarking Study

Pooja M. Panchmatia, Biplab Sanyal, and Peter M. Oppeneer

Department of Physics, Box 530, Uppsala University,
SE-751 21 Uppsala, Sweden
pooja.panchmatia@fysik.uu.se

Abstract. The iron porphyrin molecule is one of the most important biomolecules. In spite of its importance to life science, on a microscopic scale its electronic properties are not yet well-understood. In order to achieve such understanding we have performed an *ab initio* computational study of various molecular models for the iron porphyrin molecule. Our *ab initio* electronic structure calculations are based on the density functional theory (DFT) and have been conducted using both the Generalised Gradient Approximation (GGA) and the GGA+U approach, in which an additional Hubbard-U term is added for the treatment of on-site electron-electron correlations. In our investigations we have, first, optimised the molecular structures by computing the minimal-energy atomic distances, and second, benchmarked our computational approach by comparison to existing calculated results obtained by quantum-chemical methods. We have considered several models of ligated porphyrin (Cl and NH_3 ligated), as well as charged and non-charged molecules. In this way, the changes in the electronic, structural, and magnetic properties of the iron atom have been investigated as a function of the oxidation state and local environment of the iron atom. Our results for some of the model molecules reproduce the earlier quantum-chemical calculations done by Johansson and Sundholm [J. Chem. Phys. **120** (2003) 3229]. We find that the GGA+U approach provides a better description of the molecular electronic properties, which indicates that electron correlation effects on the iron are important and play an essential role, particularly for the spin moment on the iron atom. Also, we proceed beyond the relatively small molecular models to a larger, more realistic porphyrin molecule, for which we also find that the GGA+U results are in better agreement with experiments.

1 Introduction

Porphyrin rings are essential building blocks of many biological systems. In particular blood contains the common iron porphyrin form called the haem group, which is responsible for transporting oxygen to the body cells from the lungs. Each porphyrin ring is made up of four pyrolenine rings interconnected with a carbon bridge. In turn, each pyrolenine ring consists of a nitrogen atom and four carbon atoms [1]. The macrocyclic system is unsaturated with alternating single and double bonds all through the ring, which gives the molecule an overall lower total energy. This feature makes porphyrin systems chemically very stable [1]. A large class of porphyrin molecular rings exist and its number is continually growing [2]. This is mainly the result of the ring geometry, which has many positions where different functional groups can be substituted, which in turn influence and introduce many novel, useful properties. Some of these properties include, e.g., catalytic capabilities, interaction with specific molecules, the occurrence of magnetism, all of which correlate with the oxidation and reduction properties of the metal complex [3]. Synthetic porphyrin molecules have been used as cancer treatment agents and analytical reagents, too. Another important functionality of porphyrin rings is that they can be used effectively as diagnostic tools for malignant diseases such as malaria and cancer. Also, other investigations which deserve to be mentioned, concentrate on the use of porphyrin in molecular scale computing devices and switches. Clearly there is a great need to understand, on a quantum-chemical level, how and why these rings portray such a vast range of properties and hence be able to control these properties and employ them [4].

Experimental information concerning the electronic structure of porphyrin-type biomolecules has in recent years been derived from spectroscopic investigations (see, e.g., [5]). The two most popular spectroscopic methods for studying paramagnetic biomolecules are Electron Spin Resonance (ESR) and Nuclear Magnetic Resonance (NMR). For haem the g-tensors obtained from ESR measurements provide mainly the orbital occupation of the iron atom. The isotropic and anisotropic hyperfine-interaction constants can be used to estimate the spin densities. Spin distributions and electronic configurations can be derived from NMR spectroscopy [6,7,8].

A controversy between the ESR g-tensors requiring a whole unpaired dπ electron at the iron atom and NMR measurements suggesting up to 0.2 unpaired electrons dispersed in the porphyrin [9,10] has led to an increased interest in doing quantum chemical calculations on these systems. Several groups [11,12,13,14,15,18,19] have recently addressed these issues quantitatively using density functional theory-based approaches. A variety of hybrid functionals with split valence and exchange correlation and triple valence basis sets augmented with added polarisation functions have been used. Johansson and Sundholm [15] in particular, performed detailed calculations on small, porphyrin-type molecules, focusing thereby on two properties, the molecular spin-polarisation effect of oxidised iron porphyrins and the charge delocalisation effect upon reduction. They used DFT based method with the Becke's 1988 exchange functional [16] and the Lee, Yang, Parr correlation functional [17] (BLYP). Recent computational investigations [11,12,13,14,15] have revealed that calculated properties of Fe-porphyrin groups depend rather sensitively on seemingly small variations, such as peripheral molecular groups. Also, it has become clear that the computed electronic properties depend considerably on the type of quantum-chemical method used. Both findings signal that in order to be able to predict functional properties of porphyrin-type molecules highly-accurate, *ab initio* calculations have to be performed. However, at present it is not yet unambiguously clear which quantum chemical method can provide the best results.

Here we report a benchmarking computational study on Fe-porphyrin-type molecules. In the present investigation we first study four and five coordinated iron porphyrins, similar to those investigated by Johansson and Sundholm [15], which allows us to compare our results with those obtained previously with different computational schemes. The adopted porphyrin models are quite small (13-17 atoms) compared to the haem ring structure (37 atoms). In the calculations of Johansson and Sundholm the ring structure representing the porphyrin plane is more symmetry constrained than in the real system in both four and five coordinated cases. In our calculations we have, in contrast, allowed for a full structural relaxation of the molecules. Also, in all previous calculations quantum chemistry codes are used, whereas here we approach similar systems with the implementation of periodic boundary conditions. The use of the present code is advantageous as our future aim is to study porphyrin molecules supported on substrates and to understand the chemical and magnetic interactions between the substrate and the molecule as well as molecule-molecule interactions on the same substrate. The implementation of periodic boundary conditions is therefore vital to simulate a realistic surface extended over a considerable area.

2 Computational Details

Our calculations have been performed by an *ab initio* full-potential plane wave code (VASP) [21] using the Projector Augmented Wave (PAW) [22] method. This method has proven to work well for magnetic systems containing transition metals. A kinetic energy cut-off of 400 eV was used for the plane waves included in the basis set. The Perdew-Wang GGA (generalised gradient approximation) was used for the exchange-correlation [23] potential. We have also used the GGA+U approach, to include strong on-site electron-electron correlations explicitly through an additional Hubbard-U term, where U is the on-site Coulomb interaction parameter. As the value of U is not calculated *ab initio*, it is varied here in the range of 2 and 4 eV for iron. The exchange parameter was fixed to be 1 eV. Structural relaxations of the molecules were done by optimising the Hellmann-Feynman forces with a tolerance of 0.02 eV/Å. Local properties such as local density of states and local magnetic moments were calculated by projecting the wave functions onto spherical harmonics as described in Ref. [24]. For this work, the parallel version of the VASP code was implemented.

3 Results and Discussion

In the present investigation, several structural models were used to represent the ligated porphyrins both in the anionic and cationic states. Similar structural models were also adopted in the study of Johansson and Sundholm [15]. To represent the histidine bridge connecting the haem to the protein we used an

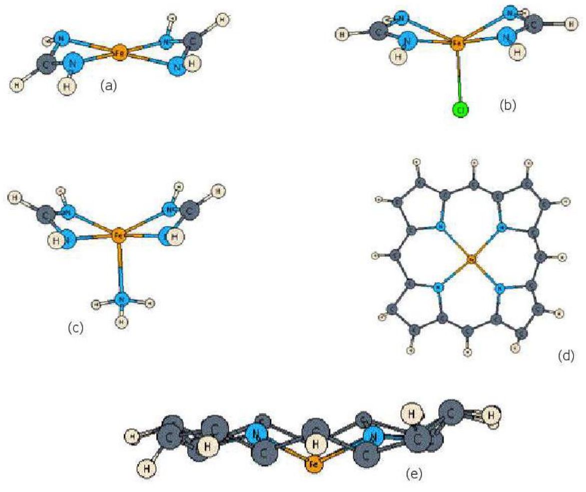


Fig. 1. Plots of the molecular models adopted for the ligated and unligated haem structures; (a) neutral haem model (FeR_2), (b) chlorinated haem model (FeR_2Cl), (c) amino-ligated haem model ($FeR_2(NH_3)$), (d) larger haem molecule, and (e) side-view of the larger haem molecule. The abbreviation R denotes here NHCHNH.

amino ligand NH_3 and for the oxidated state of the haemoglobin, a chloride ligand was used [2].

Figure 1 shows the different models used in this study, with four sets of molecules. Structures (a), (b) and (c) have been investigated in both neutral and charged forms. For the charged molecule, we used a compensating jellium background. We optimised the geometries of the molecules (a) and (b) starting from the structures proposed by Johansson and Sundholm [15], by minimising the Hellmann-Feynman forces. Structure (c) is similar to the proposed structure of Johansson and Sundholm, but ligated with a chlorine atom, while (d) is the experimental structure obtained from X-ray diffraction [25]. Our calculated optimised bond-lengths are given in Table 1. From the listed values in Table 1 it is clear that the GGA+U approach yields larger bond-lengths as compared to the GGA approach. This can be understood from the stronger iron-ligand bonding occurring for the more delocalised d states within the GGA calculations. We note, however, that the GGA+U values for the Fe–N bond-lengths are in very good agreement with the experimental numbers [25], which lends support to the GGA+U electronic structures.

In Table 2 we present the computed spin moments per iron atom, as well as the Fe $3d$ spin moments and $3d$ occupation numbers. Where available, the computed spin moments from Johansson and Sundholm are also given. Overall, there exists a reasonable agreement between the present results and those of Ref. [15]. These results are graphically displayed in Fig. 2. The same trends in the effect of the ionic state on the moments and Fe-$3d$ occupation numbers are observed. There are also some interesting differences between the calculated values. For

Table 1. Calculated bond distances (in Å) between the iron atom and the porphyrin nitrogen atoms (denoted as Fe-N_P) and the bond distance to the ligand nitrogen atom in the case of the amino-ligated porphyrin (denoted Fe-L) and, respectively, distances between the iron atom and chlorine atom for the chlorine ligated porphyrin (Fe-L). The calculated distances are listed for the GGA and the GGA+U approach, for $U = 4$ eV. Δz is the calculated vertical displacement of the iron atom out of the porphyrin plane due to the axial ligand (NH_3 or Cl).

Molecule	Fe–N_P		Fe–L		Δz	
	GGA	GGA+U	GGA	GGA+U	GGA	GGA+U
$Fe(R_2)$	1.94	1.98				
$Fe(R_2)^+$	1.91	1.92				
$Fe(R_2)NH_3$	2.12	2.15	2.2	2.2	0.77	0.77
$Fe(R_2)NH_3^+$	1.94	2.06	2.14	2.11	0.73	0.78
$Fe(R_2)Cl$	1.98	1.98	2.38	2.2	0.72	0.76
$Fe(R_2)Cl^-$	2.17	2.17	2.26	2.27	1.01	0.72
Haem	1.98	2.00				

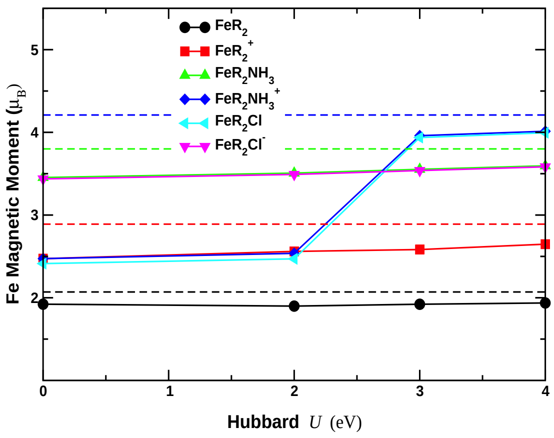


Fig. 2. Calculated spin magnetic moments on Fe of the various molecules versus the used computational method. The dashed horizontal lines represent results of Johansson and Sundholm, Ref. [15], where identical colours are used to denote computed moments for the same molecules.

some molecules, the GGA and GGA+U results are comparable, while for other molecules ($Fe(R_2)^+$, $Fe(R_2)NH_3$, $Fe(R_2)NH_3{}^+$) there are larger differences. For most molecules, the GGA+U results are closer to those of Ref. [15], particularly for higher values of U, as explicitly shown in Fig. 2. It can be understood from the calculated DOS why sometimes the effect of the additional Hubbard-U on the moments appears relatively small. Even though the Hubbard-U modifies the positions of the energy levels, the number of spin majority minus spin minority electrons does not change much. For the cationic amino-ligated por-

Table 2. Computed spin magnetic moment of iron (in μ_B), the 3d magnetic moment of the iron atom, and total d occupancy number of the iron atom. Results of three different computational approaches are compared, those of Johansson and Sundholm [15] and our calculations using the GGA and GGA+U (with $U = 4$ eV), respectively.

Molecule	Fe spin moment			Fe d spin moment			Fe d-occ. nr.		
	JS [15]	GGA	GGA+U	JS [15]	GGA	GGA+U	JS [15]	GGA	GGA+U
Fe(R_2)	2.07	1.94	1.96	1.93	1.92	1.94	6.43	6.22	6.23
Fe(R_2)$^+$	2.89	2.53	2.70	2.57	2.47	2.65	6.19	5.99	5.96
Fe(R_2)NH_3	3.80	3.50	3.64	3.54	3.45	3.60	6.21	6.00	5.99
Fe(R_2)NH_3^+	4.21	2.53	4.08	3.91	2.47	4.01	5.92	5.94	5.74
Fe(R_2)Cl		2.47	4.06		2.41	3.99		6.05	5.75
Fe(R_2)Cl$^-$		3.49	3.63		3.44	3.58		6.00	5.96
Haem		1.97	1.95		1.92	1.92		6.11	6.13

phyrin molecule our calculations differ from those of Johansson and Sundholm at lower values of U. Our calculations for the charged amino-ligated system and the non-charged chlorine-ligated system predict large magnetic moments where the high spin state (S=5/2) solution is expected according to Hund's rule. The total calculated spin moment per molecule is indeed 5 μ_B. For lower U values an intermediate spin state with S=3/2 is obtained.

The d-density of states (DOS) for these molecules are shown in Figure 3. The DOS of these two structures are presented showing both the spin up and spin

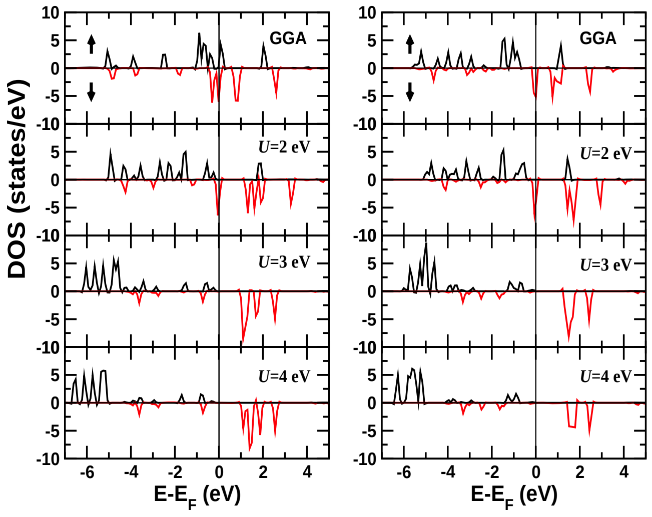


Fig. 3. Computed Fe d-DOS of a charged amino-ligated iron porphyrin molecule (left) and the non-charged chlorine-ligated porphyrin (right). The top panels give the GGA-calculated DOS, whereas the other panels show the DOS obtained from GGA+U calculations with the U as specified in the panel.

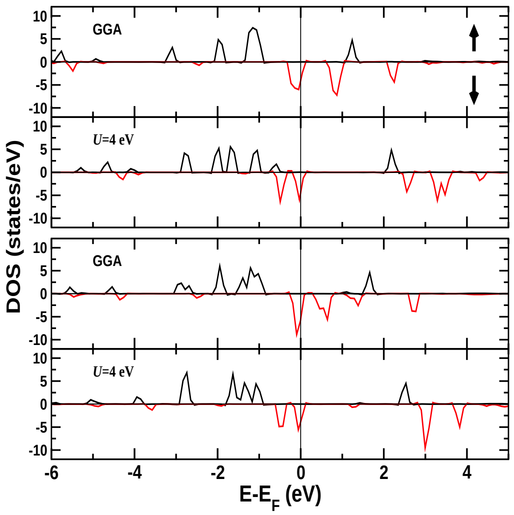


Fig. 4. Calculated Fe *d*-density of states (DOS) of the two neutral models (structures (a) and (d) in Fig. 1) of the iron porphyrin ring

down channels. The high minority spin DOS peak just below the Fermi energy for the GGA and GGA+U (U=2 eV) calculations is shifted downwards for U=3 and 4 eV and thereby completely removed from the Fermi level.

Figure 4 compares the neutral cases (structures (a) and (d)) for both the GGA and GGA+U (where U=4 eV) calculations. The presence of the pyrrole rings in the larger structure (d) does have an effect on the DOS of the iron atom. The addition of the Coulombic interaction parameter U, leads to a significant change in the spin down channel, especially for the larger model (d), however this is not reflected in the predicted magnetic moments shown in Table 2. We note, while there are some similarities in the DOS's of the two molecules for the GGA+U calculations, the GGA calculated DOS's show more pronounced differences. Thus, as far as the magnetic moments are concerned, the small molecule (a) gives a good representation of the haem molecule (d), yet some discrepancies remain.

4 Conclusions

We have studied various structural models of the Fe-porphyrin molecule using an *ab initio* computational scheme based on the density functional theory. A particular aim of our study is to benchmark electronic structure calculations for porphyrin molecules. Our present calculations, employing a full-potential code in combination with the GGA and GGA+U approaches, lead to spin moments and *d*-electron occupancies, which compare reasonably well with earlier results of Johansson and Sundholm [15]. Comparing the results from the GGA and

GGA+U calculations, we find that the GGA+U with U=3-4 eV provides a better description of the structural and electronic properties of these systems. The smaller adopted model structures reproduce several of the properties of the larger haem molecule. Having reproduced the earlier quantum-chemical results, our present investigation paves the ground for future, large scale studies of porphyrin systems on substrates.

Acknowledgements

We gratefully acknowledge grants for computer time from the Swedish National Infrastructure for Computing (SNIC). We thank Olle Eriksson and Heiko Wende for valuable suggestions. This work has been supported financially through the EU network "Novel Magneto-Optical Biosensors for Malaria Diagnosis" of the European Commission.

References

1. `http://www.chemistry.wustl.edu/~edudev/Labtutorials/Hemoglobin`
2. Scheidt, W.R., Gouterman, M.: In: Lever, A.B.P, Gray, H.B. (eds.) Iron Porphyrins (part 1). Physical Bioinorganic Chemistry Series, vol. 1(3), Addison-Wesley, Reading, MA (1983)
3. `http://www.bio.davidson.edu/Courses/Molbio/MolStudent`
4. `http://www.wiley.com/legacy/college/boyer/0470003790/`
5. Bertini, I., Luchinat, C., Parigi, G.: Solution NMR of Paramagnetic Molecules. Elsevier, Amsterdam (2001)
6. McGarvey, B.R.: Coord. Chem. Rev. 170, 75 (1998)
7. Walker, F.A., Reis, D., Balke, V.L.: J. Am. Chem. Soc. 106, 6888 (1984)
8. Walker, F.A.: Coord. Chem. Rev. 471, 185 (1999)
9. Horrocks, W.D., Greenberg, E.S.: Biochim. Biophys. Acta 491, 137 (1973)
10. Tan, H., Simonis, U., Walker, F.A.: J. Am. Chem. Soc. 116, 5784 (1994)
11. Dey, A., Ghosh, A.: J. Am. Chem. Soc. Comm. 124, 3206 (2002)
12. Conradie, J., Ghosh, A.: J. Phys. Chem. B. 107, 6486 (2003)
13. Johansson, M.P., Blomberg, M.R.A., Sundholm, D., Wikström, M.: Biochim. Biophys. Acta 1553, 183 (2002)
14. Johansson, M.P., Sundholm, D., Wikström, M.: J. Am. Chem. Soc. 124, 11771 (2002)
15. Johansson, M.P., Sundholm, D.: J. Chem. Phys. 120, 3229 (2003)
16. Becke, A.D.: Phys. Rev. A. 38, 3098 (1988)
17. Lee, C., Yang, W., Parr, R.G.: Phys. Rev. B. 37, 785 (1998)
18. Smith, D.M.A., Dupuis, M., Straatsma, T.P.: Mol. Phys. 103, 273 (2005)
19. Smith, D.M.A., Dupuis, M., Straatsma, T.P.: J. Am. Chem. Soc. 125, 2711 (2003)
20. Page, C.C., Moser, C.C., Chen, X., Dutton, P.L.: Nature 402, 47 (1999)
21. Kresse, G., Hafner, J.: Phys. Rev. B 47, R558 (1993), Kresse, G., Furthmüller, J.: Phys. Rev. B 54, 11169 (1996)
22. Blöchl, P.E.: Phys. Rev. B 50, 17953 (1994)
23. Perdew, J.P., Wang, Y.: Phys. Rev. B 45, 13244 (1992)
24. Eichler, A., Hafner, J., Furthmüller, J., Kresse, G.: Surf. Science 346, 300 (1996)
25. Fermi, G., Perutz, F.M., Shaanan, B., Fourme, R.: J. Mol. Biol. 175, 159 (1984)

Mechanical Properties of Random Alloys from Quantum Mechanical Simulations

Levente Vitos[1,2,3] and Börje Johansson[1,2]

[1] Applied Materials Physics, Department of Materials Science and Engineering, Royal Institute of Technology, SE-10044 Stockholm, Sweden
[2] Condensed Matter Theory Group, Physics Department, Uppsala University, SE-75121 Uppsala, Box 530, Sweden
levente.vitos@fysik.uu.se
[3] Research Institute for Solid State Physics and Optics, H-1525 Budapest, P.O. Box 49, Hungary

Abstract. Today, a direct determination of the mechanical properties of complex alloys from first-principles theory is not feasible. On the other hand, well established phenomenological models exist, which are suitable for an accurate description of materials behavior under various mechanical loads. These models involve a large set of atomic-level physical parameters. Unfortunately, in many cases the available parameters have unacceptably large experimental error bars. Here we demonstrate that computational modeling based on modern first-principles alloy theory can yield fundamental physical parameters with high accuracy. We illustrate this in the case of aluminum and transition metal alloys and austenitic stainless steels by computing the size and elastic misfit parameters, and the surface and stacking fault energies as functions of chemical composition.

1 Introduction

The mechanical properties represent the behavior of materials under applied forces. They are of vital importance in fabrication processes and use. Materials behavior are usually described in terms of stress or force per unit area and strain or displacement per unit distance. On the basis of stress and strain relations, one can distinguish elastic and plastic regimes. At small stress, the displacement and applied force obey Hook's law and the specimen returns to its original shape on uploading. Beyond the so called elastic limit, upon strain release the material is left with a permanent shape. Several models of elastic and plastic phenomena in solids have been established. For a detailed discussion of these models we refer to [1,2,3,4,5].

Within the elastic regime, the *single crystal elastic constants* and *polycrystalline elastic moduli* play the principal role in describing the stress-strain relation. Within the plastic regime, the importance of lattice defects in influencing the mechanical behavior of crystalline solids was recognized long time ago. Plastic deformations are primarily facilitated by dislocation motion and can occur at stress levels far below those required for dislocation-free crystals. Dislocation theory is a widely studied field within material science. Most recently, the mechanism of dislocation motion was also confirmed in complex materials [6].

The mechanical hardness represents the resistance of material to plastic deformation. It may be related to the yield stress separating the elastic and plastic regions, above which a substantial dislocation activity develops. In an ideal crystal dislocations can move easily because they experience only the weak periodic lattice potential. In real crystals, however, the motion of dislocations is impeded by obstacles, leading to an elevation of the yield strength. According to this scenario, the yield stress is decomposed into the Peierls stress, needed to move a dislocation in the crystal potential, and the solid-solution strengthening contribution, due to dislocation pinning by the randomly distributed solute atoms. The Peierls stress of pure metals is found to be approximately proportional to the *shear modulus* [5]. Dislocation pinning by random obstacles has been studied by classical theories [2,3,4] and it was found to be mostly determined by the *size misfit* and *elastic misfit* parameters. The concentration (c) dependence of the Peierls term is governed by that of the elastic constants, whereas the solid-solution strengthening contribution depends on concentration as $c^{2/3}$ [2,3,4].

Besides the above described bulk parameters, the formation energies of two-dimensional defects are also important in describing the mechanical characteristics of solids. The *surface energy*, defined as the excess free energy of a free surface, is a key parameter in brittle fracture. According to Griffith theory [5], the fracture stress is proportional to the square root of the surface energy, that is, the larger the surface energy is, the larger the load could be before the solid starts to break apart. Another important planar defect is the stacking fault in close-packed lattices, such as the face-centered cubic (fcc) or hexagonal close-packed (hcp) lattice. In these structures, the dislocations may split into energetically more favorable partial dislocations having Burgers vectors smaller than a unit lattice translation [1]. The partial dislocations are bound together and move as a unit across the slip plane. In the ribbon connecting the partials the original ideal stacking of close-packed lattice is faulted. The energy associated with this miss-packing is the *stacking-fault energy* (SFE). The balance between the SFE and the energy gain by splitting the dislocation determines the size of the stacking fault ribbon. The width of the stacking fault ribbon is of importance in many aspects of plasticity, as in the case of dislocation intersection or cross-slip. In both cases, the two partial dislocations have to be brought together to form an unextended dislocation before intersection or cross-slip can occur. By changing the SFE or the dislocation strain energy, wider or narrower dislocations can be produced and the mechanical properties can be altered accordingly. For instance, materials with high SFE permit dislocations to cross slip easily. In materials with low SFE, cross slip is difficult and dislocations are constrained to move in a more planar fashion. In this case, the constriction process becomes more difficult and hindered plastic deformation ensues. Designing for low SFE, in order to restrict dislocation movement and enhance hardness was adopted, *e.g.*, in transition metal carbides [7].

The principal problem related to modeling the mechanical properties of complex solid solutions is the lack of reliable experimental data of the alloying effects on the fundamental bulk and surface parameters. While the volume misfit parameters are available for almost all the solid solutions, experimental values of the elastic misfit parameters are scarce. There are experimental techniques to establish the polar dependence of the surface energy, but a direct measurement of its magnitude is not yet feasible [8]. In contrast to the surface energy, the stacking fault energy can be determined from experiments. For instance, one can find a large number of measurements on the stacking fault energy of austenitic stainless steels [9,10]. However, different sets of experimental data published on similar steel compositions differ significantly, indicating large error bars in these measurements.

On the theoretical side, the number of accurate calculations on solid solutions is also very limited. In fact, the complexity of the problem connected with the presence of disorder impeded any former attempts to calculate the above parameters from *ab initio* methods. Our ability to determine the physical parameters of solid solutions from first-principles has become possible with the Exact Muffin-Tin Orbitals (EMTO) method [11,12] based on the density functional theory [13] and efficient alloy theories [14]. Within this approach, we could reach a level of accuracy where many fundamental physical quantities of random alloys could be determined with an accuracy equal to or in many cases even better than experiments. The EMTO method has proved an accurate tool in the theoretical description of the simple and transition metal alloys [14,15,16,17,18,19] and, in particular, Fe-based random alloys [20,21,22,23,24]. In this work, we illustrate the possible impact of such calculations on modeling the mechanical properties of simple and transition metal binary alloys and austenitic stainless steels.

2 Theory

In this section, we briefly review the theory of the elastic constants, surface energy and stacking fault energy and give the most important numerical details used in the *ab initio* determination of these physical parameters.

2.1 Physical Parameters

Volume and elastic misfit parameters: The equilibrium volume $V(c)$ of a pseudo-binary alloy $A_{1-c}B_c$ is obtained from a Morse [25] or a Murnaghan [26] type of function fitted to the free energies $F(V)$ calculated for different volumes.

The elastic constants are the second order derivatives of the free energy with respect to the strain tensor e_{kl} $(k, l = 1, 2, 3)$, *viz.*

$$c_{ijkl} = \frac{1}{V}\frac{\partial F}{\partial e_{ij}\partial e_{kl}}, \quad (1)$$

where the derivatives are calculated at the equilibrium volume and at constant e's other than e_{ij} and e_{kl}. In a cubic system there are 3 independent elastic constants. Employing the Voigt notations, these are c_{11}, c_{12} and c_{44}. In a hexagonal crystal, there are 5 different elastic constants: c_{11}, c_{12}, c_{13}, c_{33} and c_{44}.

On a large scale, a polycrystalline material can be considered to be isotropic in a statistical sense. Such system is completely described by the bulk modulus B and the shear modulus G[1]. The only way to establish the *ab initio* polycrystalline B and G is to average the single crystal elastic constants c_{ij} by suitable methods based on statistical mechanics. A large variety of averaging methods has been proposed. According to Hershey's averaging method [27], for a cubic system the average shear modulus G is a solution of equation

$$G^3 + \alpha G^2 + \beta G + \gamma = 0, \tag{2}$$

where $\alpha = (5c_{11} + 4c_{12})/8, \beta = -c_{44}(7c_{11} - 4c_{12})/8, \gamma = -c_{44}(c_{11} - c_{12})(c_{11} + 2c_{12})/8$. This approach turned out to give the most accurate relation between cubic single-crystal and polycrystalline data in the case of Fe-Cr-Ni alloys [28]. For cubic crystals, the polycrystalline bulk modulus coincides with the single-crystal $B = (c_{11} + 2c_{12})/3$.

The size (ε_b) and elastic (ε_G) misfit parameters are calculated from the concentration dependent Burgers vector $b(c)$ or lattice parameter, and shear modulus $G(c)$ as

$$\varepsilon_b = \frac{1}{b(c)}\frac{\partial b(c)}{\partial c}, \text{ and } \varepsilon_G = \frac{1}{G(c)}\frac{\partial G(c)}{\partial c}. \tag{3}$$

According to Labusch-Nabarro model [3,4], solid solution hardening is proportional to $c^{2/3}{\varepsilon_L}^{4/3}$, where $\varepsilon_L \equiv \sqrt{{\varepsilon'_G}^2 + (\alpha\varepsilon_b)^2}$ is the Fleischer parameter, $\varepsilon'_G \equiv \varepsilon_G/(1 + 0.5|\varepsilon_G|)$ and $\alpha = 9 - 16$.

Surface energy: The surface free energy ($\gamma_{\rm S}$) represent the excess free energy per unit area associated with an infinitely large surface. For a random alloy, this can be calculated from the free energy of the surface region $F^{\rm S}(\{c^\alpha\})$ as

$$\gamma_{\rm S} = \frac{E_{\rm S}}{A_{\rm 2D}} = \frac{F_{\rm S}(\{c^\alpha\}) - F_{\rm B}(\tilde{c})}{A_{\rm 2D}}, \tag{4}$$

where $\{c^\alpha\}$ denotes the equilibrium surface concentration profile, $\alpha = 1, 2, ...$ is the layer index, $F_{\rm B}(\tilde{c})$ is the bulk free energy referring to the same number of atoms as $F_{\rm S}(\{c^\alpha\})$, and $\tilde{c}$ is set to be equal with the average concentration from the surface region. $A_{\rm 2D}$ is the area of the surface. The surface concentration profile is determined for each bulk concentration c by minimizing the free energy of the surface and bulk subsystems as described, *e.g.*, in [19].

Stacking fault energy: A perfect fcc crystal has the ideal $ABCABCAB$... stacking sequence, where the letters denote adjacent (111) atomic layers. The intrinsic stacking fault is the most commonly found fault in experiments on fcc metals. This fault is produced by a shearing operation described by the

[1] The Young modulus E and Poisson ratio ν are connected to B and G by the relations $E = 9BG/(3B + G)$ and $\nu = (3B - 2G)/(6B + 2G)$.

transformation $ABC \to BCA$ to the right-hand side of an (111) atomic layer, and it corresponds to the $ABCA\dot{C}\dot{A}\dot{B}\dot{C}$ stacking sequence, where the translated layers are marked by dot. The formation energy of an extended stacking fault is defined as the excess free energy per unit area, *i.e.*

$$\gamma_{\mathrm{SF}} = \frac{F_{\mathrm{SF}} - F_{\mathrm{B}}}{A_{\mathrm{2D}}}, \tag{5}$$

where F_{SF} and F_{B} are the free energies of the system with and without the stacking fault, respectively. Since the intrinsic stacking fault creates a negligible stress near the fault core, the faulted lattice approximately preserves the close packing of the atoms, and can be modeled by an ideal close-packed lattice. Within the third order axial interaction model [29], for the excess free energy we find $F_{\mathrm{SF}} - F_{\mathrm{B}} \approx F_{hcp} + 2F_{dhcp} - 3F_{fcc}$, where F_{fcc}, F_{hcp} and F_{dhcp} are the energies of fcc, *hcp* and double-*hcp* (*dhcp*) structures, respectively.

2.2 The *ab initio* Calculations

In the present application, the elastic constants have been derived by calculating the total energy[2] as a function of small strains δ applied on the parent lattice. For a cubic lattice, the two cubic shear constants, $c' = (c_{11} - c_{12})/2$ and c_{44}, have been obtained from volume-conserving orthorhombic and monoclinic distortions, respectively. Details about these distortions can be found in [15,16]. The bulk modulus B has been determined from the equation of state fitted to the total energies of undistorted cubic structure ($\delta = 0$). For a hexagonal lattice, at each volume V, the theoretical hexagonal axial ratio $(c/a)_0$ has been determined by minimizing the total energy $E(V, c/a)$ calculated for different c/a ratios close to the energy minimum. The hexagonal bulk modulus has been obtained from the equation of state fitted to the energy minima $E(V, (c/a)_0)$. The five hexagonal elastic constants have been obtained from the bulk modulus, the logarithmic volume derivative of $(c/a)_0(V)$, and three isochoric strains, as described in [30].

In the EMTO total energy calculations, the one-electron equations were solved within the scalar-relativistic and frozen-core approximations. To obtain the accuracy needed for the calculation of the elastic constants, we used about $\sim 10^5$ uniformly distributed $k-$points in the irreducible wedge of the Brillouin zone of the ideal and distorted lattices. In surface calculations, the close-packed fcc (111) surface was modeled by 8 atomic layers separated by 4 layers of empty sites simulating the vacuum region. The 2D Brillouin zone was sampled by $\sim 10^2$ uniformly distributed $k-$points in the irreducible wedge. The EMTO basis set included *sp* and *d* orbitals in the case of simple metal alloys, and *spd* and *f* orbitals for Ag-Zn, Pd-Ag and Fe-based alloys. The exchange-correlation was treated within the generalized gradient approximation (GGA) [31].

[2] For the temperature dependence of the elastic constants of random alloys the reader is referred to [18].

Table 1. Theoretical (present results) and experimental [32] equilibrium volume V (units of Bohr^3), hexagonal axial ratio $(c/a)_0$, and elastic constants (units of GPa) of the *hcp* $Ag_{0.3}Zn_{0.7}$ random alloy

$Ag_{0.3}Zn_{0.7}$	V	$(c/a)_0$	c_{11}	c_{12}	c_{13}	c_{33}	c_{44}
theory	110.8	1.579	110	56	63	129	27
experiment	104.3	1.582	130	65	64	158	41
percent error	6	0.2	15	14	2	18	34

3 Results

3.1 Misfit Parameters

Hexagonal Ag-Zn: We illustrate the accuracy of the present *ab initio* approach by comparing in Table 1 the theoretical results obtained for the $Ag_{0.3}Zn_{0.7}$ random alloy with experimental data [32]. The deviation between the theoretical and experimental equilibrium volume and equilibrium hexagonal axial ratio $(c/a)_0$ are 6 % and 0.2 %, respectively. The calculated elastic constants are somewhat small when compared with the measured values, but the relative magnitudes are well reproduced by the theory. Actually, the difference between the two sets of data is typical for what has been obtained for simple and transition metals in connection with the GGA for the exchange-correlation functional [31]. Therefore, the overall agreement between theory and experiment can be considered to be very satisfactory.

Aluminium alloys: On the left panel of Figure 1, the theoretical single-crystal elastic constants for Al-Mg are compared to the available experimental data [33]. We observe that the experimental value is slightly overestimated for c_{44} and c_{12} and underestimated for c_{11}. Such deviations are typically obtained for elemental metals. Nevertheless, for all three elastic constants we find that the variations of the theoretical values with concentration are in perfect agreement with the experimental data. In particular, we point out that both the experimental and theoretical c_{11} and c_{12} decrease whereas c_{44} slightly increases with Mg addition.

The calculated size misfit parameters (ε_b) for five Al-based solid solutions are compared to the experimental values on the right panel of Figure 1. An excellent agreement between the computed and experimental values is observed. This figure also shows the calculated elastic misfit parameter (ε_G) as a function of theoretical ε_b. According to these data, we can see that the elastic misfit for Mn, Cu and Mg contributes by $\sim$ 25% to $\varepsilon_L^{4/3}$, *i.e.* to the solid solution hardening, whereas this effect is below 3% in the case of Si and Zn.

Austenitic stainless steels: The volume and shear modulus of $Fe_{100-c-n}Cr_cNi_n$ alloys have been determined as functions of chemical composition for $13.5 < c < 25.5$ and $8 < n < 24$. The calculated composition-shear modulus map is presented in Figure 2 (left panel). Alloys with large shear modulus correspond to low and intermediate Cr ($< 20\%$) and low Ni ($< 15\%$) concentrations. Within this group of al-

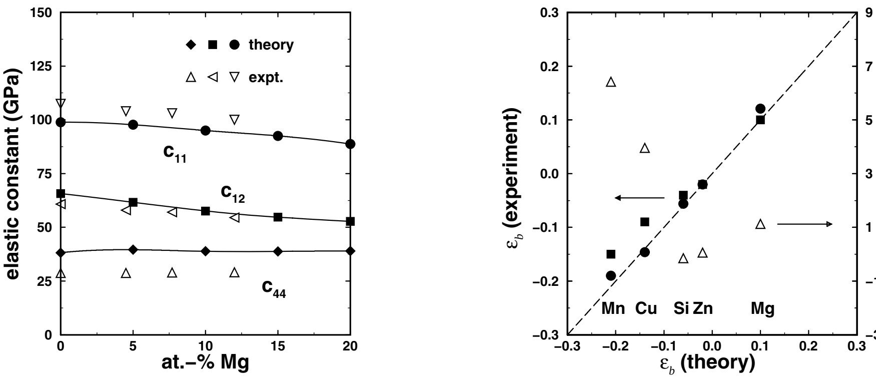


Fig. 1. Left panel: Composition dependence of the theoretical (present results) and experimental [33] single-crystal elastic constants of Al-Mg random alloys. Right panel: Misfit parameters for selected Al-based alloys (alloying elements shown at the bottom of the figure). Left axis: theoretical ε_b versus experimental ε_b (Ref. [34]: circles, Ref. [35]: squares); right axis: theoretical ε_G versus theoretical ε_b.

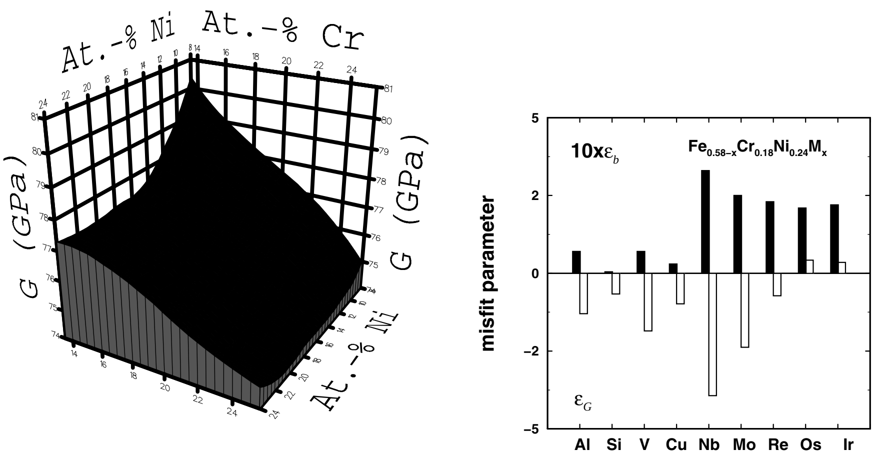


Fig. 2. Left panel: Calculated shear modulus of FeCrNi alloys as a function of the chemical composition. Right panel: Theoretical misfit parameters for $Fe_{58}Cr_{18}Ni_{24}$ alloy comprising a few percentage of Al, Si, V, Cu, Nb, Mo, Re, Os or Ir. Note that the size misfit parameters have been multiplied by a factor of 10.

loys G decreases monotonically with both Cr and Ni from a pronounced maximum of 81 GPa (near $Fe_{78}Cr_{14}Ni_8$) to approximately 77 GPa. The high Cr content alloys define the second family of austenites possessing the lowest shear moduli (< 75 GPa) with a minimum around $Fe_{55}Cr_{25}Ni_{20}$. The third family of austenites, with intermediate G values, is located at moderate Cr ($< 20\%$) and high Ni ($> 15\%$) concentrations, where G shows no significant chemical composition dependence.

The effect of alloying additions on the volume and shear modulus of alloy with composition $Fe_{58}Cr_{18}Ni_{24}$ is demonstrated on the right panel of Figure 2. It is found that Nb and Mo give the largest elastic misfit parameters ($|\varepsilon_G| = 3.9$ and 2.4, respectively). The size misfit is negligible for Al, Si, V and Cu, but it has a sizable value (between 0.21 and 0.33) for the 4d and 5d dopants. The Fleischer parameter is 5.4 for Nb, 4.1 for Mo, and $\sim$ 3.5 for the 5d elements. All the other dopants give $\varepsilon_L < 1.5$. Hence, assuming that the Labusch-Nabarro model is valid in the case of Fe-Cr-Ni alloys encompassing a few percent of additional elements, Nb and Mo are expected to yield the largest solid solution hardening. However, one should also take into account that the 4d metals, in contrast to the 5d metals, significantly decrease G and thus the Peierls stress [20]. Therefore, the overall hardening effect might be different from the one expressed merely via the Fleischer parameter.

3.2 Surface Energy of Pd-Ag Alloys

The surface energy and the top layer Ag concentration (c^1) of the *fcc* (111) surface of the Ag-Pd random alloy[3], calculated as a function of temperature and bulk Ag concentration, are plotted on the left panel of Figure 3. At 0 K, the surface energy is mainly determined by the pure Ag surface layer, which is reflected by an almost flat $E_\mathrm{S}(0\mathrm{K}, c) \approx E_{\mathrm{S,Ag}}$ line for $c \gtrsim 0.1$. With increasing temperature, $E_\mathrm{S}(T, c)$ converges towards the value estimated using a linear interpolation between end members. Note that the temperature dependence of E_S is very similar to that of c^1. Although, at intermediate bulk concentrations, the subsurface Ag concentration (c^2) shows strong temperature dependence [19], this effect is imperceptible in the surface energy. Therefore, the variation of the surface energy for a close-packed facet with temperature and bulk composition is, to a large extent, governed by the surface layer, and the subsurface layers play only a secondary role.

3.3 Stacking Fault Energy of Austenitic Stainless Steels

The calculated room-temperature SFE of Fe-Cr-Ni alloys is shown on the right panel of Figure 3 as a function of chemical composition. We can observe a strongly nonlinear composition dependence. This behavior is a consequence of the persisting local moments in austenitic steels, which are the take-off for many basic properties that the austenitic stainless steels exhibit [23]. The local magnetic moments disappear near the stacking faults, except in the high-Ni–low-Cr alloys, where they are comparable to those from the bulk. This gives a significant reduction of the magnetic fluctuation contribution to the SFE in the high-Ni–low-Cr alloys, but for the rest of the alloys, it is found that the magnetic part of γ_{SF} has the same order of magnitude as the total SFE . We have found that the most common austenitic steels, *i.e.* those with low-Ni and intermediate and

[3] For the effect of structural relaxation on the surface energy and equilibrium concentration profile the reader is referred to [36].

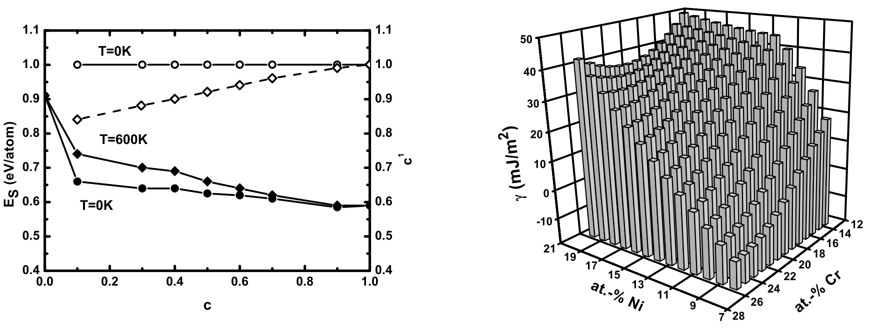


Fig. 3. Left panel: Surface energy (E_S) and top layer Ag concentration (c^1) for the *fcc* (111) surface of the Ag-Pd random alloy as functions of temperature and bulk Ag concentration. Lines are added to guide to the eye. Right panel: Stacking fault energy of Fe-Cr-Ni random alloys. The calculated SFE is shown for T = 300 K as a function of Cr and Ni content.

high-Cr content are in fact stabilized by the magnetic entropy. They possess $\gamma_{SF} \gtrsim 0$, and have enhanced hardness. The high-Ni–low-Cr alloys are more ductile compared to the rest of the compositions, since they have the largest γ_{SF}.

The role of the additional alloying elements has been investigated in Fe-Cr-Ni-M alloys encompassing a few percent of Nb or Mn. While the effect of Mn is found to be similar to that of Cr [9], the relative effect of Nb can be as large as $\sim 30\%$ [24]. However, the absolute effect of Nb on the SFE depends strongly on the initial composition. For instance, in alloys close to the *hcp* magnetic transition, Nb decreases the SFE [24]. Therefore, in a steel design process, both the alloying element and the composition of the host material are key parameters for predicting the role of alloying. This finding is in contrast to the widely employed models for compositional dependence of SFE, and it clearly demonstrates that no universal composition equations for the SFE can be established.

4 Summary

We have demonstrated that computational methods based on modern *ab initio* alloy theory can yield essential physical parameters for random alloys with an accuracy comparable to the experiment. These parameters can be used in phenomenological models to trace the variation of the mechanical properties with alloying additions.

Acknowledgments. The Swedish Research Council, the Swedish Foundation for Strategic Research and the Hungarian Scientific Research Fund (OTKA T046773 and T048827) are acknowledged for financial support.

References

1. Argon, A.S., Backer, S., McClintock, F.A., Reichenbach, G.S., Orowan, E., Shaw, M.C., Rabinowicz, E.: Metallurgy and Materials. Addison-Wesley Series. Addison-Wesley Publishing Company, Ontario (1966)
2. Fleischer, R.L.: Acta Metal 11, 203–209 (1963)
3. Labusch, R.: Acta Met. 20, 917–927 (1972)
4. Nabarro, F.R.N.: Philosophical magazine 35, 613–622 (1977)
5. Lung, C.W., March, N.H.: World Scientific Publishing Co. Pte. Ltd (1999)
6. Chisholm, M.F., Kumar, S., Hazzledine, P.: Science 307, 701–703 (2005)
7. Hugosson, H., Jansson, U., Johansson, B., Eriksson, O.: Science 293, 2423–2437 (2001)
8. Sander, D.: Current Opinion in Solid State and Materials Science 7, 51 (2003)
9. Schramm, R.E., Reed, R.P.: Met. Trans. A 6A, 1345–1351 (and references therein) (1975)
10. Miodownik, A.P.: Calphad 2, 207–226 (and references therein) (1978)
11. Andersen, O. K., Jepsen, O., Krier, G.: In: Kumar, V., Andersen, O. K., Mookerjee, A. (eds.) Lectures on Methods of Electronic Structure Calculations, World Scientific Publishing, Singapore, (1994) pp. 63–124
12. Vitos, L.: Phys. Rev. B 64, 014107(11) (2001)
13. Hohenberg, P., Kohn, W.: Phys. Rev. 136, B864–B871 (1964)
14. Vitos, L., Abrikosov, I.A., Johansson, B.: Phys. Rev. Lett. 87, 156401(4) (2001)
15. Taga, A., Vitos, L., Johansson, B., Grimvall, G.: Phys. Rev. B 71, 014201(9) (2005)
16. Magyari-Köpe, B., Grimvall, G., Vitos, L.: Phys. Rev. B 66, 064210, (2002) ibid 66, 179902 (2002)
17. Magyari-Köpe, B., Vitos, L., Grimvall, G.: Phys. Rev. B 70, 052102(4) (2004)
18. Huang, L., Vitos, L., Kwon, S.K., Johansson, B., Ahuja, R.: Phys. Rev. B 73, 104203 (2006)
19. Ropo, M., Kokko, K., Vitos, L., Kollár, J., Johansson, B.: Surf. Sci. 600, 904–913 (2006)
20. Vitos, L., Korzhavyi, P.A., Johansson, B.: Nature Materials 2, 25–28 (2003)
21. Dubrovinsky, L., et al.: Nature 422, 58–61 (2003)
22. Dubrovinsky, N., et al.: Phys. Rev. Lett. 95, 245502(4) (2005)
23. Vitos, L., Korzhavyi, P.A., Johansson, B.: Phys. Rev. Lett. 96, 117210(4) (2006)
24. Vitos, L., Nilsson, J.-O., Johansson, B.: Acta Materialia 54, 3821–3826 (2006)
25. Moruzzi, V.L., Janak, J.F., Schwarz, K.: Phys. Rev. B. 37, 790–799 (1988)
26. Murnaghan, F.D.: Proc. Nat. Acad. Sci. 30, 244–247 (1944)
27. Ledbetter, H.M.: J. Appl. Phys. 44, 1451–1454 (1973)
28. Ledbetter, H.M.: In: Levy, M., Bass, H.E., Stern, R.R. (eds.) Handbook of Elastic Properties of Solids, Liquids, Gases. Academic, San Diego, vol. III, p. 313 (2001)
29. Denteneer, P.J.H., van Haeringen, W.: J. Phys. C: Solid State Phys. 20, L883–L887 (1987)
30. Steinle-Neumann, G., Stixrude, L., Cohen, R.E.: Phys. Rev. B 60, 791–799 (1999)
31. Perdew, J.P., Burke, K., Ernzerhof, M.: Phys. Rev. Lett. 77, 3865–3868 (1996)
32. Matsuo, Y.: J. Phys. Soc. Japan 53, 1360–1365 (1984)
33. Gault, C., Dauger, A., Boch, P.: Physica Status Solidi (a) 43, 625–632 (1977)
34. King, H.W.: Journal of Materials Science 1, 79–90 (1966)
35. Pearson, W.B.: In: Raynor, G.V. (ed.) Handbook of lattice spacings and structures of metals and alloys, Pergamon Press Ltd., New York (1958)
36. Ropo, M.: Phys. Rev. B 74, 195401(4) (2006)

Novel Data Formats and Algorithms for Dense Linear Algebra Computations: Minisymposium Abstract

Fred G. Gustavson[1] and Jerzy Waśniewski[2]

[1] IBM T.J. Watson Research Center, New York, USA and Umeå University, Sweden
[2] Technical University of Denmark, Lyngby, Denmark

Recently, recursion was introduced into the area of Dense Linear Algebra with the intent of producing new algorithms as well as improving its existing algorithms. Recursion, via the divide-and-conquer paradigm, introduced variable blocking to complement the standard fixed block algorithms. In particular, new data structures for dense linear algebra became an active research area. This minisymposium focuses on both new data structures and/or new or existing algorithms for high performing dense linear algebra algorithms.

Cache Oblivious Matrix Operations Using Peano Curves

Michael Bader and Christian Mayer

Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
bader@in.tum.de, mail@christianmayer.de

Abstract. Algorithms are called *cache oblivious*, if they are designed to benefit from any kind of cache hierarchy—regardless of its size or number of cache levels. In linear algebra computations, block recursive techniques are a common approach that, by construction, lead to inherently local data access patterns, and thus to an overall good cache performance [3].

We present block recursive algorithms that use an element ordering based on a Peano space filling curve to store the matrix elements. We present algorithms for matrix multiplication and LU decomposition, which are able to minimize the number of cache misses on any cache level.

1 Cache Oblivious Algorithms in Linear Algebra

Due to the performance gap between the typical speed of main memory and the execution speed of (arithmetic) operations within modern CPUs, the efficient use of cache memory is nowadays essential for obtaining satisfying computational performance in all kinds of algorithms. There are, in principle, two main approaches to cache efficient programming: *cache aware* approaches will take a given algorithm, and modify data structures and program code such that it works better on a *given* hardware, where the number, size, and structure of caches are all known to the programmer. In contrast, *cache oblivious* approaches strive to design algorithms that will benefit from any presence of cache memory. Such algorithms usually require excellent locality properties of the data access pattern, which makes them inherently cache efficient.

An overview of cache aware and hybrid approaches in linear algebra, as well as of cache oblivious approaches in this field, has been given by Gustavson[4] or Elmroth et. al.[3]. For matrix computations, cache oblivious algorithms often focus on block recursive approaches, where matrices are recursively subdivided into smaller matrix blocks. The recursive cascade of block sizes makes sure that all cache sizes in a cache hierarchy will be efficiently used.

In [1], we presented a block recursive algorithm for matrix multiplication that combines block recursion with a *space filling curve* approach. We showed that storing matrix elements in an order given by a Peano space filling curve leads to an algorithm with excellent locality features. We also demonstrated that these locality features are a direct result of the properties of the Peano curve, and can not be obtained by similar approaches based on Morton order or Hilbert curves, for example.

In this paper we will present a block recursive algorithm for LU-decomposition that works on the same element ordering, and benefits from the same locality properties. In sections 2 to 4, we will recapitulate briefly the main ideas of the block recursive scheme for matrix multiplication on the Peano ordering. In section 5, we will introduce the block recursive scheme for the LU-decomposition. We will finish with first performance results and a summary in sections 6 and 7.

2 A Block Recursive Scheme for Matrix Multiplication

Consider the multiplication of two 3×3-matrices, such as given in equation (1), where the indices of the matrix elements indicate the order in which the elements are stored in memory.

$$\underbrace{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}}_{=:B} = \underbrace{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}}_{=:C} . \tag{1}$$

The scheme is similar to a column-major ordering, however, the order of the even-numbered columns has been inverted, which leads to a meandering scheme, which is also equivalent to the basic pattern of a Peano space filling curve. Now, if we examine the operations to compute the elements c_r of the result matrix, we notice that the operations can be executed in a very convenient order—from each operation to the next, an element is either reused or one of its direct neighbours in memory is accessed:

$$\begin{tikzcd}
c_0 \mathrel{+}= a_0 b_0 \arrow[d] & c_0 \mathrel{+}= a_6 b_2 \arrow[r] & c_5 \mathrel{+}= a_5 b_4 \arrow[d] & c_6 \mathrel{+}= a_0 b_6 \arrow[r] & c_6 \mathrel{+}= a_6 b_8 \arrow[d] \\
c_1 \mathrel{+}= a_1 b_0 \arrow[d] & c_1 \mathrel{+}= a_7 b_2 \arrow[u] & c_4 \mathrel{+}= a_4 b_4 \arrow[d] & c_7 \mathrel{+}= a_1 b_6 \arrow[u] & c_7 \mathrel{+}= a_7 b_8 \arrow[d] \\
c_2 \mathrel{+}= a_2 b_0 \arrow[d] & c_2 \mathrel{+}= a_8 b_2 \arrow[u] & c_3 \mathrel{+}= a_3 b_4 \arrow[d] & c_8 \mathrel{+}= a_2 b_6 \arrow[u] & c_8 \mathrel{+}= a_8 b_8 \\
c_2 \mathrel{+}= a_3 b_1 \arrow[d] & c_3 \mathrel{+}= a_8 b_3 \arrow[u] & c_3 \mathrel{+}= a_2 b_5 \arrow[d] & c_8 \mathrel{+}= a_3 b_7 \arrow[u] & \\
c_1 \mathrel{+}= a_4 b_1 \arrow[d] & c_4 \mathrel{+}= a_7 b_3 \arrow[u] & c_4 \mathrel{+}= a_1 b_5 \arrow[d] & c_7 \mathrel{+}= a_4 b_7 \arrow[u] & \\
c_0 \mathrel{+}= a_5 b_1 \arrow[r] & c_5 \mathrel{+}= a_6 b_3 \arrow[u] & c_5 \mathrel{+}= a_0 b_5 \arrow[r] & c_6 \mathrel{+}= a_5 b_7 \arrow[u] &
\end{tikzcd} \tag{2}$$

An algorithmic scheme with this spatial locality property can be obtained for any matrices of odd dimensions, as long as we adopt a meandering numbering scheme. However, cache efficiency requires *temporal* locality, as well, in the sense that matrix elements are reused within short time intervals, and will therefore not be removed from the cache by other data. To achieve temporal locality, we combine the scheme with a block recursive approach. Consequently, the element numbering is then also defined by a block recursive meandering scheme—which directly leads to a Peano space filling curve.

3 An Element Numbering Based on a Peano Space Filling Curve

Figure 1 illustrates the recursive scheme used to linearise the matrix elements in memory. It is based on a so-called *iteration* of a Peano curve. Four different block numbering patterns marked as P, Q, R, and S are combined in a way to ensure a contiguous numbering of the matrix elements—direct neighbours in memory will always be direct neighbours in the matrix as well.

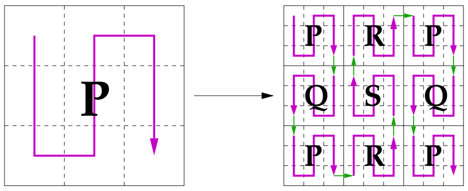


Fig. 1. Recursive construction of the Peano numbering scheme based on four block numbering patterns denoted by P, Q, R, and S

In its pure form, the numbering scheme only works for square matrices with dimensions that are a power of 3. To work with matrices of arbitrary size, we can for example use zero-padding to embed the given matrix into a quadratic $3^p \times 3^p$-matrix. When implementing the respective matrix algorithms, we need to make sure that no superfluous operations are performed on the padded zero-blocks.

An alternative approach is to stop recursion on larger matrix blocks with odd numbers of rows and columns. As any odd number can be written as a sum of three odd numbers of nearly the same size, this leads to a Peano numbering scheme that works for any odd-numbered dimensions. For even dimension, padding with only a single row and/or column of zeros is then sufficient[2].

4 A Block Recursive Scheme for Matrix Multiplication

Equation (3) shows the blockwise multiplication of matrices stored according to the proposed numbering scheme. Each matrix block is named with respect to its numbering scheme and indexed with the name of the global matrix and the position within the storage scheme:

$$\underbrace{\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}}_{=:\,A} \underbrace{\begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix}}_{=:\,B} = \underbrace{\begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}}_{=:\,C} . \qquad (3)$$

Analogous to the 3×3 multiplication in equation (1), we obtain an execution order for the individual block operations. The first operations are

$$P_{C0} \mathrel{+}= P_{A0} P_{B0} \quad \rightarrow \quad Q_{C1} \mathrel{+}= Q_{A1} P_{B0} \quad \rightarrow \quad P_{C2} \mathrel{+}= P_{A2} P_{B0} \quad \rightarrow \quad \ldots$$

If we only consider the ordering scheme of the matrix blocks, we obtain eight different types of block multiplications:

$$\begin{array}{llll} P \mathrel{+}= PP & Q \mathrel{+}= QP & R \mathrel{+}= PR & S \mathrel{+}= QR \\ P \mathrel{+}= RQ & Q \mathrel{+}= SQ & R \mathrel{+}= RS & S \mathrel{+}= SS\,. \end{array} \tag{4}$$

All eight types of block multiplication lead to multiplication schemes similar to that given in equation (2), and generate inherently local execution orders. Thus, we obtain a closed system of eight block multiplication schemes which can be implemented by a respective system of nested recursive procedures. The resulting algorithm has several interesting properties concerning cache efficiency:

1. the number of cache misses on an ideal cache can be shown to be asymptotically minimal;
2. on any level of recursion, after a matrix block has been used, either this block will be directly reused, or one of its direct neighbours in space will be accessed next; an interesting consequence of this property is that
3. precise knowledge for prefetching is available.

An extensive discussion of the resulting block-recursive multiplication algorithm and its properties can be found in [1]. There, we also demonstrate that a block-recursive scheme with such strict locality properties can only be obtained by using the Peano curve.

5 A Block Recursive Scheme for LU Decomposition

Based on the presented numbering scheme, we can also try to set up a block recursive algorithm for LU decomposition. Hence, consider the following decomposition of block matrices:

$$\underbrace{\begin{pmatrix} \underline{|P_{L0}} & 0 & 0 \\ Q_{L1} & \underline{|S_{L4}} & 0 \\ P_{L2} & R_{L3} & \underline{|P_{L8}} \end{pmatrix}}_{=:\,L} \underbrace{\begin{pmatrix} \overline{P_{U0}|} & R_{U5} & P_{U6} \\ 0 & \overline{S_{U4}|} & Q_{U7} \\ 0 & 0 & \overline{P_{U8}|} \end{pmatrix}}_{=:\,U} = \underbrace{\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}}_{=:\,A}\,. \tag{5}$$

In this notation, $\underline{|A}$ denotes a lower triangular matrix block A, while $\overline{B|}$ denotes an upper triangular matrix block B. Again, we need to derive a set of block operations, which we try to put into an execution order that preserves locality. However, in contrast to matrix multiplication, we now have to obey certain precedence rules. Unfortunately, these precedence rules deny us a scheme that is strictly memory local. However, we can still try to minimise the non-localities, which may lead us to the following scheme (for P-numbered matrix blocks):

1) $\lfloor\underline{P_{L0}}\,\overline{P_{U0}}\rceil = P_{A0}$ – LU decomp.	8) $\lfloor\underline{S_{L4}}\,\overline{S_{U4}}\rceil = S_{A4}$ – LU decomp.
2) $Q_{L1}\overline{P_{U0}}\rceil = Q_{A1}$ – solve for Q_{L1}	9) $Q_{A7} -= Q_{L1}P_{U6}$ – matr. mult.
3) $P_{L2}\overline{P_{U0}}\rceil = P_{A2}$ – solve for P_{L2}	10) $P_{A8} -= P_{L2}P_{U6}$ – matr. mult.
4) $\lfloor\underline{P_{L0}}\,P_{U6} = P_{A6}$ – solve for P_{U6}	11) $R_{L3}\overline{S_{U4}}\rceil = R_{A3}$ – solve for R_{L3}
5) $\lfloor\underline{P_{L0}}\,R_{U5} = R_{A5}$ – solve for R_{U5}	12) $\lfloor\underline{S_{L4}}\,Q_{U7} = Q_{A7}$ – solve for Q_{U7}
6) $R_{A3} -= P_{L2}R_{U5}$ – matr. mult.	13) $P_{A8} -= R_{L3}Q_{U7}$ – matr. mult.
7) $S_{A4} -= Q_{L1}R_{U5}$ – matr. mult.	14) $\lfloor\underline{P_{L8}}\,\overline{P_{U8}}\rceil = P_{A8}$ – LU decomp.

Note that an additional LU-decomposition scheme has to be derived for S-numbered blocks: $\lfloor\underline{S_{L4}}\,\overline{S_{U4}}\rceil = S_{A4}$. In addition, there are two further types of schemes to be derived:

1. Solve a matrix equation such as $Q_L\overline{P}\rceil = Q_A$ for the matrix Q_L, where $\overline{P}\rceil$ is an already computed upper triangular matrix; in the same manner solve $P_L\overline{P}\rceil = P_A$, $R_L\overline{S}\rceil = R_A$, and $S_L\overline{S}\rceil = S_A$ for P_L, R_L, and S_L, respectively.
2. Solve a matrix equation such as $\lfloor\underline{P}\,P_U = P_A$ for the matrix P_U, where $\lfloor\underline{P}$ is an already computed lower triangular matrix; in the same manner solve $\lfloor\underline{P}\,R_U = R_A$, $\lfloor\underline{S}\,Q_U = Q_A$, and $\lfloor\underline{S}\,S_U = S_A$, for R_U, Q_U, and S_U, respectively.

For both types of schemes, we again determine block recursive schemes based on the Peano ordering. In the usual manner, we try to reuse matrix blocks in the next block operation or use a matrix block that is a direct neighbour in memory. Again, the precedence relations do not allow an ideal scheme, such as for the matrix multiplication. The resulting schemes for PPP numbering are given in tables 1 and 2. All remaining block recursion schemes, including those for LU-decomposition, are listed in [6].

Finally, we obtain a system of nested block recursive schemes for the LU-decomposition, where the recursive schemes for matrix multiplication, and for the two schemes with the given triangular matrices are nested with the two schemes

Table 1. Block operations for solving the equation $P_LP_U = P_A$ for matrix P_U where the lower triangular matrix P_L is given

1) $\lfloor\underline{P_{L0}}\,P_{U0} = P_{A0}$ – solve for P_{U0}	10) $\lfloor\underline{S_{L4}}\,S_{U4} = S_{A4}$ – solve for S_{U4}
2) $Q_{A1} -= Q_{L1}P_{U0}$ – matr. mult.	11) $R_{A3} -= R_{L3}S_{U4}$ – matr. mult.
3) $P_{A2} -= P_{L2}P_{U0}$ – matr. mult.	12) $\lfloor\underline{P_{L8}}\,R_{U3} = R_{A3}$ – solve for R_{U3}
4) $\lfloor\underline{S_{L4}}\,Q_{U1} = Q_{A1}$ – solve for Q_{U1}	13) $\lfloor\underline{P_{L0}}\,P_{U6} = P_{A6}$ – solve for P_{U6}
5) $P_{A2} -= R_{L3}Q_{U1}$ – matr. mult.	14) $Q_{A7} -= Q_{L1}P_{U6}$ – matr. mult.
6) $\lfloor\underline{P_{L8}}\,P_{U2} = P_{A2}$ – solve for P_{U2}	15) $P_{A8} -= P_{L2}P_{U6}$ – matr. mult.
7) $\lfloor\underline{P_{L0}}\,R_{U5} = R_{A5}$ – solve for R_{U5}	16) $\lfloor\underline{S_{L4}}\,Q_{U7} = Q_{A7}$ – solve for Q_{U7}
8) $R_{A3} -= P_{L2}R_{U5}$ – matr. mult.	17) $P_{A8} -= R_{L3}Q_{U7}$ – matr. mult.
9) $S_{A4} -= Q_{L1}R_{U5}$ – matr. mult.	18) $\lfloor\underline{P_{L8}}\,P_{U8} = P_{A8}$ – solve for P_{U8}

Table 2. Block operations for solving the equation $P_L P_U = P_A$ for matrix P_L where the upper triangular matrix P_U is given

1) $P_{L0}\overline{P_{U0}}\rvert = P_{A0}$	– solve for P_{L0}	10) $R_{L3}\overline{S_{U4}}\rvert = R_{A3}$	– solve for R_{L3}
2) $Q_{L1}\overline{P_{U0}}\rvert = Q_{A1}$	– solve for Q_{L1}	11) $S_{L4}\overline{S_{U4}}\rvert = S_{A4}$	– solve for S_{L4}
3) $P_{L2}\overline{P_{U0}}\rvert = P_{A2}$	– solve for P_{L2}	12) $R_{L5}\overline{S_{U4}}\rvert = R_{A5}$	– solve for R_{L5}
4) $R_{A3} -= P_{L2}R_{U5}$	– matr. mult.	13) $P_{A8} -= R_{L3}Q_{U7}$	– matr. mult.
5) $S_{A4} -= Q_{L1}R_{U5}$	– matr. mult.	14) $Q_{A7} -= S_{L4}Q_{U7}$	– matr. mult.
6) $R_{A5} -= P_{L0}R_{U5}$	– matr. mult.	15) $P_{A6} -= R_{L5}Q_{U7}$	– matr. mult.
7) $P_{A6} -= P_{L0}P_{U6}$	– matr. mult.	16) $P_{L6}\overline{P_{U8}}\rvert = P_{A6}$	– solve for P_{L6}
8) $Q_{A7} -= Q_{L1}P_{U6}$	– matr. mult.	17) $Q_{L7}\overline{P_{U8}}\rvert = Q_{A7}$	– solve for Q_{L7}
9) $P_{A8} -= P_{L2}P_{U6}$	– matr. mult.	18) $P_{L8}\overline{P_{U8}}\rvert = P_{A8}$	– solve for P_{L8}

for the LU-decomposition. With respect to locality properties, the scheme is not quite as nice as that for matrix multiplication. However, it still profits from the locality properties of the Peano curve, and leads to a cache oblivious algorithm that ensures a very low number of cache misses.

Usually, LU-decomposition should be accompanied by (partial) pivoting. This is not addressed at the moment, because exchanging columns or rows in the Peano-ordered matrix-layout is a tedious task. We will give some hints on how to achieve pivoting in the conclusion, though (see section 7).

6 Performance

In this section, we compare the performance of `TifaMMy`[6], an implementation of the presented Peano algorithms, with that of Intel's Math Kernel Library[5]. Our implementation of `TifaMMy`, up to now, offers only very limited support for processors offering SIMD extensions, such as SSE operations on the Pentium architectures, for example. SIMD extensions can lead to a substantial performance gain, if algorithms can, for example, be vectorised. However, block recursion inhibits vectorisation, if it is performed until very small matrix block sizes, such as the present 3×3 blocks, are reached. The resulting lack of SSE support rather limits the achievable MFLOPS rates at the moment, and will thus be our focus of research in the imminent future (see the discussion in section 7). To level the currently existing SSE handicap to some extent, we measured the performance for different basic data structures: in addition to matrices of float or double precision floats, we also checked the running times for matrices of complex numbers, and even for using blocks of 4×4 floats as matrix elements.

Table 3 lists the cache hit rates for both the matrix multiplication and the LU-decomposition algorithm. These performance measurements were done on an Itanium 2 processor. The numbers indicate that the Peano-codes indeed cause

Table 3. Comparison of the cache hit rates of an implementation of the Peano-order algorithms, and of Intel's Math Kernel Library 7.2.1. (on an Itanium 2 processor, 1.3 GHz, with 256 Kbyte L2-cache and 3 Mbyte L3-cache).

multiplication	double		complex double	
dim.: 2048 × 2048	TifaMMy	MKL	TifaMMy	MKL
Level 2 references	$1.45 \cdot 10^{10}$	$2.47 \cdot 10^{9}$	$2.86 \cdot 10^{10}$	$4.99 \cdot 10^{9}$
Level 2 misses	$4.72 \cdot 10^{6}$	$1.38 \cdot 10^{8}$	$2.04 \cdot 10^{7}$	$2,85 \cdot 10^{8}$
Level 2 hit rate	99.97 %	94.39 %	99.93 %	94.29 %
Level 3 references	$9.35 \cdot 10^{6}$	$1.60 \cdot 10^{8}$	$3.57 \cdot 10^{7}$	$5.54 \cdot 10^{8}$
Level 3 misses	$1.29 \cdot 10^{6}$	$1.13 \cdot 10^{8}$	$3.09 \cdot 10^{6}$	$1.52 \cdot 10^{7}$
Level 3 hit rate	86.16 %	29.26 %	91.35 %	97.26 %

***LU*-decomposition**	double		complex double	
dim.: 2187 × 2187	TifaMMy	MKL	TifaMMy	MKL
Level 2 references	$5.81 \cdot 10^{9}$	$1.45 \cdot 10^{9}$	$1.14 \cdot 10^{10}$	$1.11 \cdot 10^{8}$
Level 2 misses	$9.32 \cdot 10^{6}$	$2.89 \cdot 10^{7}$	$3.29 \cdot 10^{7}$	$1.11 \cdot 10^{8}$
Level 2 hit rate	99.84 %	98.01 %	99.71 %	95.90 %
Level 3 references	$1.29 \cdot 10^{7}$	$3.44 \cdot 10^{7}$	$4.30 \cdot 10^{7}$	$1.98 \cdot 10^{8}$
Level 3 misses	$2.30 \cdot 10^{6}$	$3.38 \cdot 10^{6}$	$6.02 \cdot 10^{6}$	$8.62 \cdot 10^{6}$
Level 3 hit rate	82.12 %	90.18 %	85.99 %	95.65 %

very few cache misses. Both, on the level-2 and on the level-3 cache, the *number* of cache misses is lower than that of Intel's MKL. TifaMMy's level 3 *hit rate* (i.e. the percentage of cache hits) is actually worse than that of MKL, but this is only a consequence of the lower number of references—the absolute number of cache misses is lower by more than a factor of 2. It is not yet fully understood why the absolute number of references to the level-2 cache is higher for TifaMMy. It might result from a slight advantage of MKL's cache-aware approach for small matrix blocks.

Figure 2 shows the MFLOPS rates for matrix multiplication as measured on a Pentium 4 (3.4 GHz) and on the Itanium 2 (1.3 GHz) processor. As we can see from figure 4, the performance gap is to the most part explained by the lack of SSE optimisation. In that experiment, MKL was forced to use its default kernels, which do not use SSE in order to make them work on any Pentium processor. TifaMMy, which was also compiled without allowing SSE instructions, clearly outperforms the generic MKL code. Also, it can be seen from a comparison of figures 2 and 4 that TifaMMy takes only a minimal performance gain, when using SSE operations.

The effect of the suboptimal SSE optimisation can also be seen in figure 3. There, we used 4×4 matrix blocks of single-precision float as matrix elements, and implemented an SSE-optimised multiplication for these blocks. The MFLOPS rates are almost a factor of 2 higher than for data types double and complex double.

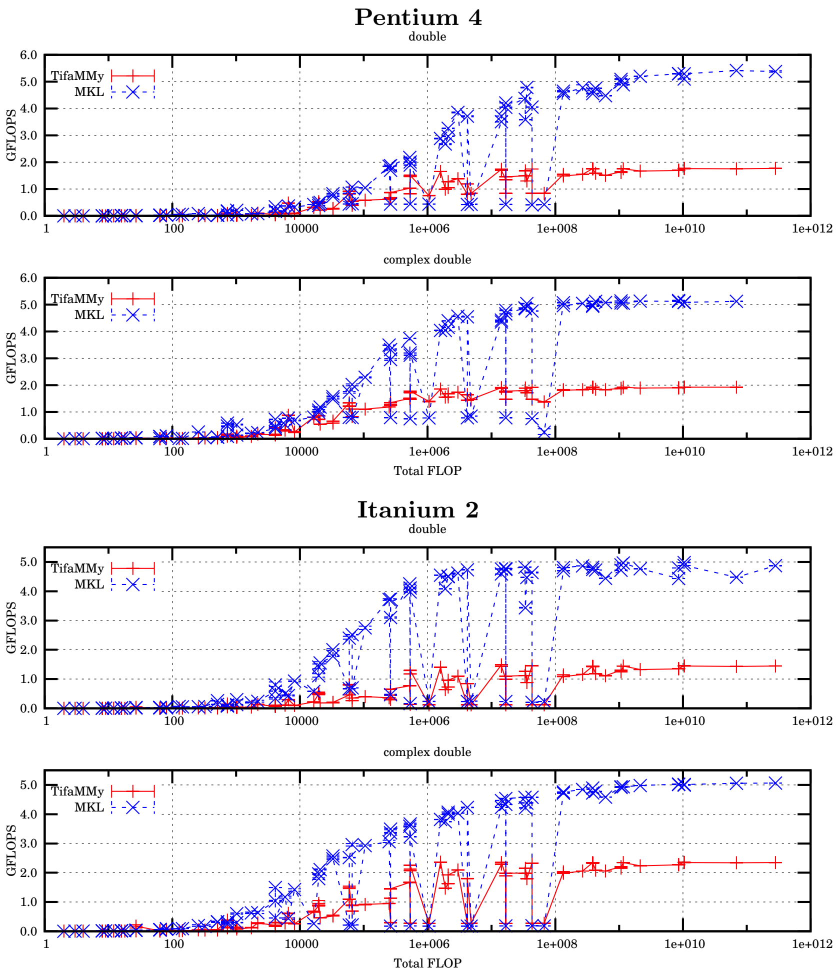


Fig. 2. Comparison of the MFLOPS rates of the matrix multiplication on a Pentium 4 and on an Itanium 2 processor

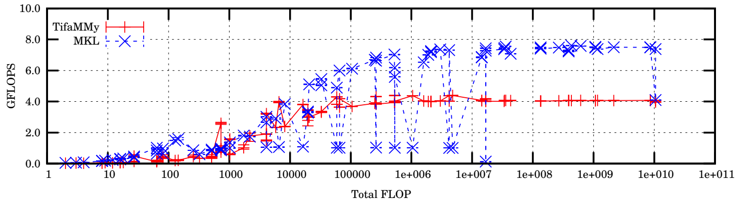


Fig. 3. Comparison of the MFLOPS rates of the matrix multiplication with 4×4-blocks of single-precision float on a Pentium 4 processor. The multiplication of the 4×4-blocks was implemented using hand-crafted SSE.

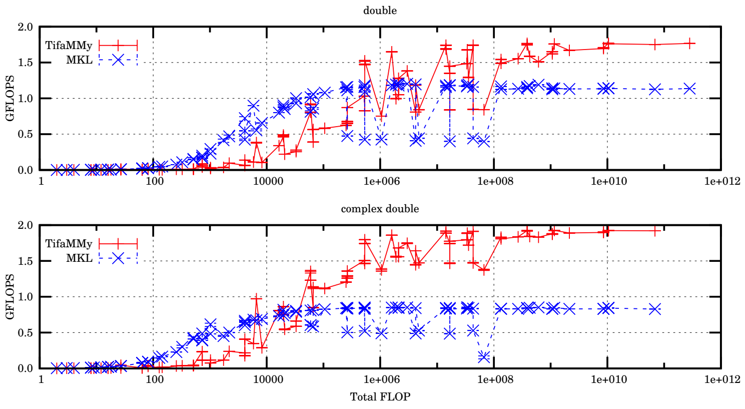


Fig. 4. Comparison of the non-SSE-MFLOPS rates of the matrix multiplication on a Pentium 4 and on an Itanium 2 processor. Here, MKL was forced to use its default kernel, which do not use SSE, and thus work on any Pentium processor. TifaMMy was compiled without allowing SSE operations.

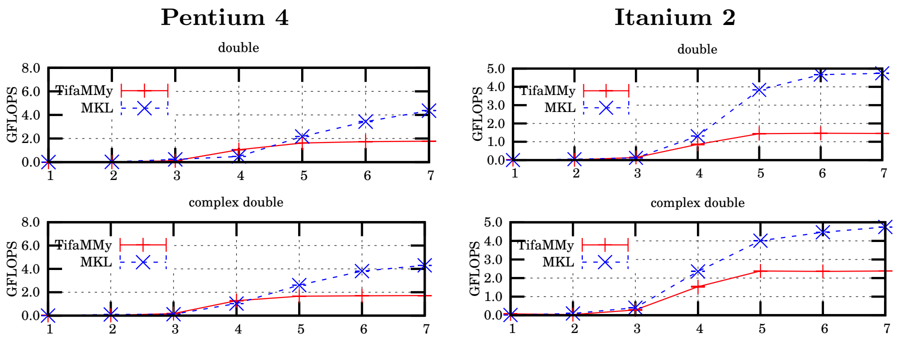


Fig. 5. Comparison of the MFLOPS rates of the LU-decomposition of $3^p \times 3^p$-matrices on a Pentium 4 and on an Itanium 2 processor

7 Conclusion and Current Research

The performance results indicate that our block recursive approach is indeed successful to minimise the cache misses for all cache levels during matrix multiplication, and also LU-decomposition. However, with respect to the use of SSE extensions, floating point registers, and even up to the first-level cache, they also indicate that switching to highly optimised implementations for operations on small matrix blocks – often called *kernels* – is necessary to get close to the processor's peak performance. This is quite in accordance with observations by other block recursive approaches [7].

Hence, our current research focus is to implement such a hybrid approach by integrating suitable hardware-aware multiplication and LU-decomposition kernels. This can, for example, be achieved by replacing the matrix elements in our scheme by matrix blocks in regular row- or column-major ordering, where the block size is tuned to fit the first level cache. Hardware-optimized kernel procedures are then called for the respective block matrix operations. First results indicate not only an even improved cache performance, but also a MFLOPS performance that at least rivals that of MKL. A further experience has been that the effort for hardware-aware implementation of these well-defined operations is comparably low. Thus, a large part of the implementation can stay *cache oblivious*, and only the inner kernels need to be *cache/hardware aware*.

The larger matrix blocks on the finest recursion level will also simplify the integration of partial pivoting. If suitable pivot elements can be found within the kernel blocks, then a block-oriented pivoting could be integrated into the cache-oblivious approach. This will, of course, not always guarantee numerical stability, but it would stay cache-friendly and make pivoting closer to partial pivoting.

References

1. Bader, M., Zenger, C.: Cache oblivious matrix multiplication using an element ordering based on a Peano curve. Linear Algebra and its Applications 417(2-3) (2006)
2. Bader, M., Zenger, C.: A Cache Oblivious Algorithm for Matrix Multiplication Based on Peano's Space Filling Curve. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 1042–1049. Springer, Heidelberg (2006)
3. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1) (2004)
4. Gustavson, F.G.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development 41(6) (1997)
5. Intel math kernel library (2005), `http://intel.com/cd/software/products/asmo-na/eng/perflib/mkl/`
6. Mayer, Ch.: Cache oblivious matrix operations using Peano curves, Diploma Thesis, TU München (2006), `http://tifammy.sourceforge.net/documentation.php`
7. Gunnels, J., Gustavson, F., Pingali, K., Yotov, K.: Is Cache-oblivious DGEMM Viable? In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 919–928. Springer, Heidelberg (2007)

Recursive Blocked Algorithms for Solving Periodic Triangular Sylvester-Type Matrix Equations

Robert Granat, Isak Jonsson, and Bo Kågström

Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{granat, isak, bokg}@cs.umu.se

Abstract. Recently, recursive blocked algorithms for solving triangular one-sided and two-sided Sylvester-type equations were introduced by Jonsson and Kågström. This elegant yet simple technique enables an automatic variable blocking that has the potential of matching the memory hierarchies of today's HPC systems. The main parts of the computations are performed as level 3 general matrix multiply and add (GEMM) operations. We extend and apply the recursive blocking technique to solving periodic Sylvester-type matrix equations. Successive recursive splittings are performed on 3-dimensional arrays, where the third dimension represents the periodicity of a matrix equation.

Keywords: Sylvester-type matrix equations, periodic matrix equations, recursion, blocking, level 3 BLAS, superscalar.

1 Introduction

The standard Sylvester equation $AX - XB = C$ has a *periodic* counter-part

$$\begin{aligned} A_k X_k - X_{k+1} B_k &= C_{k+1}, \qquad k = 1, \ldots, p-1, \\ A_p X_p - X_1 B_p &= C_1, \end{aligned}$$

where p is the periodicity of the matrix sequences, such that $A_{k+p} = A_k, B_{k+p} = B_k$ and $C_{k+p} = C_k$ [16,19]. In this contribution, we focus on recursive blocked algorithms for solving triangular periodic matrix equations, i.e., the matrix sequences A_k and B_k for $k = 1, \ldots, p$ are assumed to be in *periodic real Schur form* (PRSF) [4,10]. This means that $p-1$ of the matrices in each sequence are upper triangular and one matrix in each sequence, say A_r and B_s, $1 \leq r, s \leq p$, is quasi-triangular. The products of conforming diagonal blocks of the matrix sequences A_k and B_k contain the eigenvalues of the matrix products $A_1 A_2 \cdots A_p$ and $B_1 B_2 \cdots B_p$, respectively, where the 1×1 and 2×2 blocks on the main block diagonal of A_r and B_s correspond to real and complex conjugate pairs of eigenvalues of the corresponding matrix products.

Triangular matrix equations appear naturally in estimating the condition numbers of matrix equations and different eigenspace computations (e.g., see [14,15] and [2,17]), including decoupling and stability analysis. Periodic Sylvester-type matrix equations also appear in the context of eigenvalue reordering for computation and condition estimation of periodic invariant (deflating) subspaces of a matrix (pair) sequence [7,8,9]. To solve a triangular matrix equation is also a major step in the classical Bartels-Stewart method [1], which is our base-point for solving general non-reduced periodic matrix equations.

2 Recursive Algorithms for Periodic Triangular Matrix Equations

Our work includes novel recursive blocked algorithms for solving the most common one-sided and two-sided triangular periodic Sylvester-type matrix equations. In Table 1, a summary of the periodic matrix equations considered is displayed.

The classification in one-sided and two-sided matrix equations was introduced in [11,12] and is implicit in the definition of a matrix equation. A periodic matrix equation is *one-sided* if it only includes terms where the solution is involved in matrix products of two matrices, e.g., $\mathrm{op}(A_k)X_k$ or $X_k\mathrm{op}(A_k)$, where $\mathrm{op}(A_k)$ can be A_k or A_k^T. Similarly, a periodic matrix equation is *two-sided* if it includes matrix products of three matrices, e.g., $A_kX_kB_k^T$, where X_k is the solution sequence. This distinction relates to how blocks (subarrays) of matrices of the solution sequence (e.g., X_k in PSYCT) are used in updates of the right hand side matrices (C_k in PSYCT) in the recursive blocked algorithms. For example, our algorithms for two-sided matrix equations require more space and flops, compared to similar algorithms for one-sided equations.

For the two-sided equations, we only display one periodic transpose variant. The other variants can be derived by moving the transpose to the left multiplying matrices and replacing the periodic dependence $k+1$ by k and vice versa. For example, a second variant of PSYDT is

$$\begin{aligned} {A_k}^T X_{k+1} B_k - X_k &= C_k, \qquad k = 1, \ldots, p-1, \\ {A_p}^T X_1 B_p - X_p &= C_p. \end{aligned}$$

For the generalized equations in Table 1 we assume that the involved periodic matrix pairs, namely (A_k, D_k) and (B_k, E_k) in PGCSY, (A_k, C_k) and (B_k, D_k) in PGSYL, and (A_k, E_k) in PGLYCT and PGLYDT, are in *generalized* periodic real Schur form (GPRSF) (see, e.g., [4,10] for details).

By using and reusing recursive templates, we can solve all matrix equations listed in Table 1 utilizing only a small set of subroutines. By this, we mean that, e.g., the PLYCT problem can be largely solved by the PSYCT routine. Therefore, the efforts of optimizing the implementation can be concentrated on a few core routines.

In the following, we discuss only a few of the periodic matrix equations of Table 1 in some more detail.

The periodic matrix sequences are stored as 3-dimensional arrays, where the third dimension is the periodicity p of the matrix equation. The successive recur-

Table 1. Considered one-sided (top) and two-sided (bottom) periodic Sylvester-type matrix equations. Here, p is the periodicity of each equation and $1 \leq k < p-1$.

Name	Mnemonic	Matrix equation
Periodic continuous-time standard Sylvester	PSYCT	$\begin{cases} A_k X_k - X_{k+1} B_k = C_k \\ A_p X_p - X_1 B_p = C_p \end{cases}$
Periodic continuous-time standard Lyapunov	PLYCT	$\begin{cases} A_k X_k + X_{k+1} {A_k}^T = C_k \\ A_p X_p + X_1 {A_p}^T = C_p \end{cases}$
Periodic generalized coupled Sylvester	PGCSY	$\begin{cases} A_k X_k - Y_k B_k = C_k \\ D_k X_{k+1} - Y_k E_k = F_k \\ A_p X_p - Y_p B_p = C_p \\ D_p X_1 - Y_p E_p = F_p \end{cases}$
Periodic discrete-time standard Sylvester	PSYDT	$\begin{cases} A_k X_k {B_k}^T - X_{k+1} = C_k \\ A_p X_p {B_p}^T - X_1 = C_p \end{cases}$
Periodic discrete-time standard Lyapunov	PLYDT	$\begin{cases} A_k X_k {A_k}^T - X_{k+1} = C_k \\ A_p X_p {A_p}^T - X_1 = C_p \end{cases}$
Periodic generalized Sylvester	PGSYL	$\begin{cases} A_k X_k {B_k}^T - C_k X_{k+1} {D_k}^T = E_k \\ A_p X_p {B_p}^T - C_p X_1 {D_p}^T = E_p \end{cases}$
Periodic continuous-time generalized Lyapunov	PGLYCT	$\begin{cases} A_k X_k {E_k}^T + E_k X_{k+1} {A_k}^T = C_k \\ A_p X_p {E_p}^T + E_p X_1 {A_p}^T = C_p \end{cases}$
Periodic discrete-time generalized Lyapunov	PGLYDT	$\begin{cases} A_k X_k {A_k}^T - E_k X_{k+1} {E_k}^T = C_k \\ A_p X_p {A_p}^T - E_p X_1 {E_p}^T = C_p \end{cases}$

sive splittings are performed on the 3-dimensional arrays explicitly, leading to new types of data locality issues, compared to our previous work with RECSY [11,12,13].

2.1 Periodic Recursive Sylvester Solvers

Consider the real *periodic continuous-time Sylvester* (PSYCT) matrix equation

$$\begin{aligned} A_k X_k - X_{k+1} B_k &= C_{k+1}, \qquad k = 1, \ldots, p-1, \\ A_p X_p - X_1 B_p &= C_1, \end{aligned}$$

where the sequences A_k of size $M \times M$ and B_k of size $N \times N$ for $k = 1, \ldots, p$ are in PRSF form. The right hand sides C_k and the solution matrices X_k are of side $M \times N$. Depending on the dimensions M and N, we consider three ways of *recursive splitting.* First, we consider splitting of A_k by rows and columns and C_k by rows only. The second alternative is to split B_k by rows and columns and C_k by columns only. The third alternative is to split all three matrices by rows and columns. No matter which alternative is chosen, the number of flops is the same. Performance may differ greatly, though. Our algorithm picks the alternative that keeps matrices as "squarish" as possible, i.e., $1/2 < M/N < 2$, which guarantees a good ratio between the number of flops and the number of elements referenced.

Next, we consider the *periodic generalized continuous-time Lyapunov* (PGLYCT) equation

$$\begin{aligned} A_k X_k {E_k}^T + E_k X_{k+1} {A_k}^T &= C_k, \qquad k = 1, \ldots, p-1, \\ A_p X_p {E_p}^T + E_p X_1 {A_p}^T &= C_p, \end{aligned}$$

where the periodic matrix pair sequence (A_k, E_k) is in generalized periodic real Schur form (GPRSF), and C_k and X_k (overwrites C_k) are symmetric $M \times M$. Because of symmetry, there is only one way to split the equation, resulting in two triangular PGLYCT equations and one *generalized periodic Sylvester* (PGSYL) equation, all of which can be solved recursively using the following template:

$$\begin{aligned} A_{11}^{(k)} X_{11}^{(k)} {E_{11}^{(k)}}^T + E_{11}^{(k)} X_{11}^{(k+1)} {A_{11}^{(k)}}^T &= C_{11}^{(k)} \\ &\quad - A_{12}^{(k)} {X_{12}^{(k)}}^T {E_{11}^{(k)}}^T - E_{12}^{(k)} {X_{12}^{(k+1)}}^T {A_{11}^{(k)}}^T \\ &\quad - A_{11}^{(k)} X_{12}^{(k)} {E_{12}^{(k)}}^T - E_{11}^{(k)} X_{12}^{(k+1)} {A_{12}^{(k)}}^T \\ &\quad - A_{12}^{(k)} X_{22}^{(k)} {E_{12}^{(k)}}^T - E_{12}^{(k)} X_{22}^{(k+1)} {A_{12}^{(k)}}^T, \\ A_{11}^{(k)} X_{12}^{(k)} {E_{22}^{(k)}}^T + E_{11}^{(k)} X_{12}^{(k+1)} {A_{22}^{(k)}}^T &= C_{12}^{(k)} - A_{12}^{(k)} X_{22}^{(k)} {E_{22}^{(k)}}^T - E_{12}^{(k)} X_{22}^{(k+1)} {A_{22}^{(k)}}^T, \\ A_{22}^{(k)} X_{22}^{(k)} {E_{22}^{(k)}}^T + E_{22}^{(k)} X_{22}^{(k+1)} {A_{22}^{(k)}}^T &= C_{22}^{(k)}. \end{aligned}$$

Assuming that we have algorithms for solving PGLYCT and PGSYL (kernel solvers are discussed in Section 2.2), we start by solving for the sequence $X_{22}^{(k)}$, $k = 1, \ldots, p$ from the third (last) equation. After updating $C_{12}^{(k)}$ in the PGSYL equation (second above) with respect to $X_{22}^{(k)}$, $k = 1, \ldots, p$, we can solve for the sequence $X_{12}^{(k)}$. Finally, after updating $C_{11}^{(k)}$ in the first PGLYCT equation with respect to $X_{12}^{(k)}$ and $X_{22}^{(k)}$ for $k = 1, \ldots, p$, we solve for the sequence $X_{11}^{(k)}$.

The recursive template is now applied repeatedly to the three periodic matrix equations above (divide phase) until the subproblems are small enough, when kernel solvers are used for solving the node-leaf problems of the recursive tree. In the conquer phase, the tree is traversed level by level, finally producing the complete matrices X_k, $k = 1, \ldots, p$, i.e., the solution of PGLYCT.

2.2 Kernel Solvers for Leaf Problems

In each step of the recursive blocking, the original periodic matrix equation is reduced to several subproblems involving smaller and smaller matrices, and a great part of the computation emerges as standard matrix-matrix operations, such as general matrix multiply and add (GEMM) or triangular matrix multiply (TRMM) operations. At the end of the recursion tree, small instances of periodic matrix equations have to be solved. Each such matrix equation can be represented as a linear system $Zx = c$, where Z is a Kronecker product representation of the associated periodic Sylvester-type operator, and it belongs to the class of bordered almost block diagonal (BABD) matrices [6]. For example, the PSYCT matrix equation can be expressed as $Zx = c$, where the matrix Z of size $mnp \times mnp$ is

$$Z = \begin{bmatrix} B_p^T \otimes I_m & & & & I_n \otimes A_p \\ I_n \otimes A_1 & B_1^T \otimes I_m & & & \\ & \ddots & \ddots & & \\ & & & & \\ & & & I_n \otimes A_{p-1} & B_{p-1}^T \otimes I_m \end{bmatrix},$$

and

$$x = [\mathrm{vec}(X_1), \mathrm{vec}(X_2), \cdots, \mathrm{vec}(X_p)]^T, \quad c = [\mathrm{vec}(C_1), \mathrm{vec}(C_2), \cdots, \mathrm{vec}(C_p)]^T.$$

In the algorithm, recursion proceeds down to problem sizes of 1×1 to 2×2. For these problems, a compact form of the matrix Z which utilizes the sparsity structure of the problem is computed and the problem is solved using Gaussian elimination with partial pivoting (GEPP). These solvers are based on the superscalar kernels that were developed for the RECSY library [13]. Moreover, the block diagonal of the matrix Z sometimes (see, e.g., PGCSY in [8]) has a certain structure that can be exploited by the GEPP procedure. The memory usage for Z is $O(m^2n^2p)$, and the number of operations required to solve the problem is $O(m^3n^3p)$. In case of an ill-conditioned matrix Z, the Gaussian elimination is aborted when bad pivot elements are detected, an error condition is signaled, and the solution process is restarted with a new problem from a higher level of the recursion tree. This larger problem is then solved using LU with complete pivoting (GECP) on a non-compact form of Z, which in turn results in a p times larger memory requirement, namely $O(m^2n^2p^2)$ storage. However, since m and n are small, typically 1 or 2, this is an effective procedure for typical sizes of the periodicity p. The extra workspace can either be provided by the user or dynamically allocated.

2.3 Storage Layout of Matrices

For storage of regular dense matrices, there are two major linear variants: row-major ("C-style") and column-major ("Fortran-style"). In addition, several recursive blocked storage schemas have proven to give substantial performance improvements, see [5] and further references therein.

For the periodic matrix sequences, there are six (3!) different linear variants. One advantage of having the periodic dimension as the minor (innermost) dimension is better locality of the Z matrix in the kernel solver. However, this efficiently disables all use of standard level 3 BLAS. Therefore, we have the periodicity p as the outermost dimension. Column-major storage layout is used for each coefficient matrix $(A_k, B_k, C_k$ etc.) in the periodic matrix sequences.

3 Sample Performance Results

The recursive blocked algorithms for the periodic Sylvester-type equations have been implemented in Fortran 90, using the facilities for recursive calls of subprograms, dynamic memory allocation and threads. In this section, we present sample performance results of implementations of solvers for one-sided and two-sided equations executing on an AMD Opteron processor-based system. The system has a dual AMD Opteron 2.2 GHz processor, with a 64 kB level 1 cache, a 1024 kB level 2 cache and 8 GB memory per node. Theoretical peak performance is 4.4 Gflops/s per processor. The peak performance of DGEMM and other level 3 BLAS routines used vary between 3.0–3.5 Gflops/s. All performance numbers presented are based on uniprocessor computations.

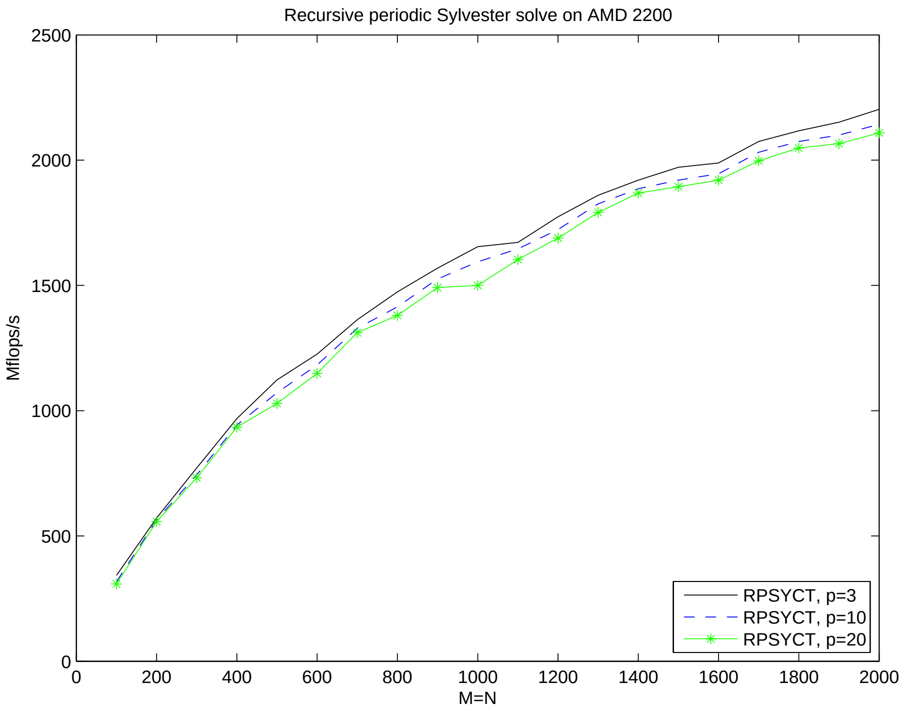


Fig. 1. Performance results for the one-sided PSYCT equation $(M = N)$ on AMD Opteron. The three graphs correspond to the periodicity $p = 3$, 10, and 20.

3.1 Performance of the Recursive Blocked PSYCT Solver

In Figure 1, performance graphs for the implementation of the recursive blocked PSYCT algorithm are displayed. The problem size ($M = N$) ranges from 100 to 2000, and the periodicity $p = 3, 10$, and 20. For large enough problems the performance approaches 70% of the DGEMM performance, which is on a level with the recursive blocked SYCT solver in RECSY [11,13]. For an increasing periodicity p the performance decreases only marginally.

3.2 Performance of the Recursive Blocked PGSYL Solver

In Figure 2, we show performance graphs for our implementation of the recursive blocked algorithm for the two-sided PGSYL equation. The performance results are somewhat inferior to the PSYCT performance, but still on a level with the recursive blocked GSYL solver in RECSY [12,13]. We also see that the relative decrease in performance with respect to an increasing periodiciy p is larger than for the PSYCT solver. Reasons for this degradation include that the PGSYL kernel solver is more complex and somewhat less efficient, and the two-sided updates in the recursive blocked algorithm result in extra operations compared

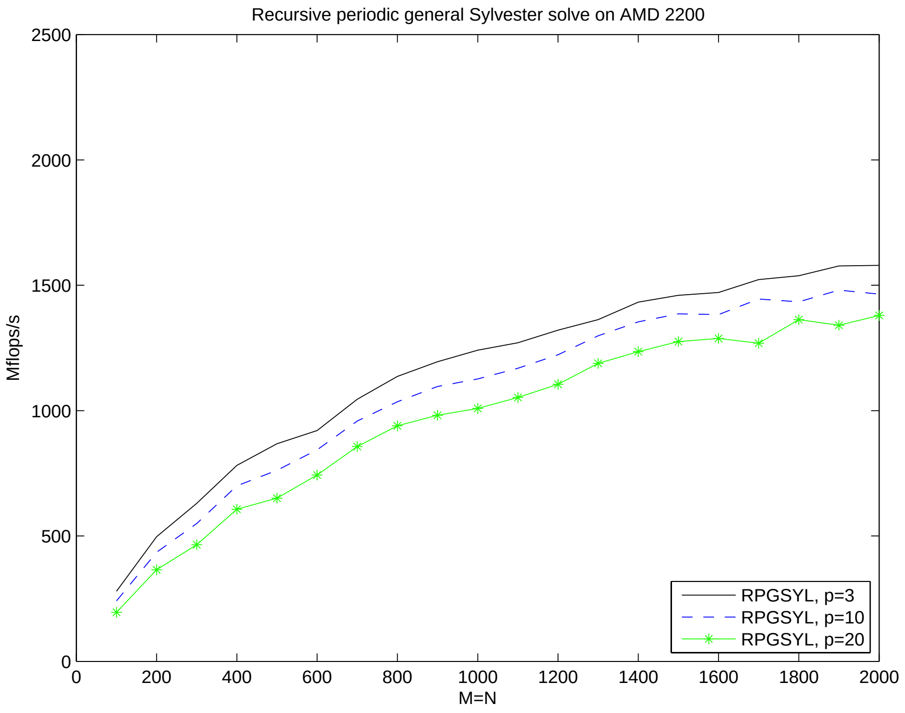


Fig. 2. Performance results for the two-sided PGSYL equation ($M = N$) on AMD Opteron. The three graphs correspond to the periodicity $p = 3$, 10, and 20.

to a true level 1 or level 2 algorithm. Such overhead never appear in the one-sided equations, but increases with p for two-sided equations. However, the use of efficient level 3 operations compensates for some of this computational overhead.

4 Conclusions and Future Work

We have presented novel recursive blocked algorithms for solving various periodic triangular matrix equations. Such equations stem from different applications with a periodic or seasonal behaviour, e.g., the study of periodic control systems [3], and discrete-time periodic (descriptor) systems [19] in particular.

Our recursive blocked algorithms are based on RECSY, an HPC library for the most common non-periodic matrix equations (see [11,12,13]). The performance results are on the level of RECSY, which confirm that the recursive blocking approach is an efficient way of solving periodic triangular Sylvester-type equations. The reason is three-fold: (*i*) recursion allows for good temporal locality; (*ii*) recursion enables the periodic matrix equations to be rewritten mainly as level 3 operations; (*iii*) novel superscalar kernel solvers deliver good performance for the small leaf-node problems. Our goal is to provide a complete periodic counterpart of the RECSY library. This study will also include alternative blocking techniques.

Acknowledgments

This research was conducted using the resources of the High Performance Computer Center North (HPC2N). Financial support has been provided by the Swedish Research Council under grant VR 621-2001-3284 and by the Swedish Foundation for Strategic Research under grant A3 02:128.

References

1. Bartels, R.H., Stewart, G.W.: Solution of the equation $AX + XB = C$. Comm. Assoc. Comput. Mach. 15, 820–826 (1972)
2. Benner, P., Mehrmann, V., Xu, H.: Perturbation analysis for the eigenvalue problem of a formal product of matrices. BIT 42(1), 1–43 (2002)
3. Bittanti, S., Colaneri, P. (eds.): Periodic Control Systems. In: Proceedings volume from the IFAC Workshop, August 27-28, 2001, Elsevier Science & Technology, Cernobbio-Como, Italy (2001)
4. Bojanczyk, A.W., Golub, G., Van Dooren, P.: The Periodic Schur Decomposition. Algorithms and Applications. In: Luk, F.T. (ed.) Proceedings SPIE Conference, vol. 1770, pp. 31–42 (1992)
5. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)
6. Fairweather, G., Gladwell, I.: Algorithms for Almost Block Diagonal Linear Systems. SIAM Review 44(1), 49–58 (2004)

7. Granat, R., Kågström, B.: Direct Eigenvalue Reordering in a Product of Matrices in Periodic Schur Form. SIAM J. Matrix Anal. Appl. 28(1), 285–300 (2006)
8. Granat, R., Kågström, B., Kressner, D.: Reordering the Eigenvalues of a Periodic Matrix Pair with Applications in Control. In: Proc. of 2006 IEEE Conference on Computer Aided Control Systems Design (CACSD), pp. 25–30 (2006) ISBN: 0-7803-9797-5
9. Granat, R., Kågström, B., Kressner, D.: Computing Periodic Deflating Subspaces Associated with a Specified Set of Eigenvalues. BIT Numerical Mathematics, December 2006 (submitted)
10. Hench, J.J., Laub, A.J.: Numerical solution of the discrete-time periodic Riccati equation. IEEE Trans. Automat. Control 39(6), 1197–1210 (1994)
11. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems — Part I: One-sided and coupled Sylvester-type matrix equations. ACM Trans. Math. Softw. 28(4), 392–415 (2002)
12. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems — Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. ACM Trans. Math. Softw. 28(4), 416–435 (2002)
13. Jonsson, I., Kågström, B.: RECSY — A High Performance Library for Sylvester-Type Matrix Equations. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 810–819. Springer, Heidelberg (2003)
14. Kågström, B., Poromaa, P.: LAPACK–Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs. ACM Trans. Math. Software 22, 78–103 (1996)
15. Kågström, B., Westin, L.: Generalized Schur methods with condition estimators for solving the generalized Sylvester equation. IEEE Trans. Automat. Control 34(4), 745–751 (1989)
16. Sreedhar, J., Van Dooren, P.: A Schur approach for solving some matrix equations. In: Helmke, U., Menniken, R., Saurer, J. (eds.) Systems and Networks: Mathematical Theory and Applications, Mathematical Research, vol. 77, pp. 339–362 (1994)
17. Sun, J.-G.: Perturbation bounds for subspaces associated with periodic eigenproblems. Taiwanese Journal of Mathematics 9(1), 17–38 (2005)
18. Varga, A.: Periodic Lyapunov equations: some applications and new algorithms. Internat. J. Control 67(1), 69–87 (1997)
19. Varga, A., Van Dooren, P.: Computational methods for periodic systems. In: Prepr. IFAC Workshop on Periodic Control Systems, Como, Italy, pp. 177–182 (2001)

Minimal Data Copy for Dense Linear Algebra Factorization

Fred G. Gustavson, John A. Gunnels, and James C. Sexton

IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
{fg2, gunnels, sextonjc}@us.ibm.com

Abstract. The full format data structures of Dense Linear Algebra hurt the performance of its factorization algorithms. Full format rectangular matrices are the input and output of level the 3 BLAS. It follows that the LAPACK and Level 3 BLAS approach has a basic performance flaw. We describe a *new* result that shows that representing a matrix A as a collection of square blocks will reduce the amount of data reformating required by dense linear algebra factorization algorithms from $O(n^3)$ to $O(n^2)$. On an IBM Power3 processor our implementation of Cholesky factorization achieves 92% of peak performance whereas conventional full format LAPACK `DPOTRF` achieves 77% of peak performance. All programming for our new data structures may be accomplished in standard Fortran, through the use of higher dimensional full format arrays. Thus, new compiler support may not be necessary. We also discuss the role of concatenating submatrices to facilitate hardware streaming. Finally, we discuss a *new* concept which we call the L1 / L0 cache interface.

1 Introduction

The current most commonly used Dense Linear Algebra (DLA) algorithms for serial and SMP processors have a performance inefficiency and hence they give sub-optimal performance. We indicate that Fortran and C two dimensional arrays are the main reason for the inefficiency. We show how to correct these performance inefficiencies by using New Data Structures (NDS) along with so-called kernel routines. These NDS generalize the current storage layouts for both the Fortran and C programming languages. One of these formats is packed format and we do not discuss it as a new result [18, 13, 19] about Rectangular Full Packed (RFP) format shows that packed format can be represented by RFP format. RFP format is full format and it and packed both use exactly the same amount of storage. However, SBP (Square Block Packed) format also replaces packed format and it is a main subject of this paper. Like RFP format it is a full format data structure and it uses only slightly more storage than RFP format.

The BLAS [22, 9, 10] (Basic Linear Algebra Subroutines) were introduced to make the algorithms of DLA performance-portable. Starting with LINPACK, [7] and progressing to LAPACK [4] the Level 1, 2, 3 BLAS were introduced. The suffix i in Level i refers to the number of nested "do loops" required to do the computation of a given BLAS. Almost all of the floating-point operations of

DLA algorithms are performed through the use of BLAS calls. If performance were directly proportional to operation count then performance would be truly portable. However, with today's deep memory hierarchies and other new architecural features, this is no longer the case. To understand the performance inefficiency of LAPACK algorithms, it suffices to discuss the Level 3 BLAS, `DGEMM` (Double precision GEneral Matrix Matrix). A relationship exists between the Level 3 BLAS and their usage in most of level 3 factorization routines. This relationship introduces a performance inefficiency in block based factorization algorithms and we will now discuss the Level 3 BLAS, `DGEMM` (Double precision GEneral Matrix Matrix) to illustrate this fact.

In [1, 5, 25, 14] design principles for producing a high performance Level 3 `DGEMM` BLAS are given. A key design principle for `DGEMM` is to partition its matrix operands into submatrices and then call a `DGEMM` L1 kernel routine multiple times on its submatrix operands. Another key design principle is to change the data format of the submatrix operands so that each call to the L1 kernel can operate at or near the peak Million FLoating point OPerations per Second (MFlops) rate. This format change and subsequent change back to standard data format is a cause of a performance inefficiency in `DGEMM`. The `DGEMM API` requires that its matrix operands be stored as standard Fortran or C two-dimensional arrays.

Any DLA Factorization Algorithm (DLAFA) of a matrix A calls `DGEMM` multiple times with *all* its operands being submatrices of A. For each call data copy will be done; therefore this unit cost gets multiplied by this number of calls. However, this overall cost can be eliminated by using the NDS to create a substitute for `DGEMM`; e.g. its analogous L1 kernel routine, which does *not* require the aforementioned data copy. So, as in [15, 17], for triangular matrices, we suggest that SBP format be used in concert with kernel routines.

This paper also describes a *new* concept which we call the L1 cache / L0 cache interface. We define a L0 cache as the register file of its floating point unit. Today, many architectures possess special hardware to support the streaming of data into the L1 cache from higher levels of memory [24, 21]. In fact with a large enough floating point register file it may be possible to do, say, a L2 or L3 cache blocking for a `DGEMM` kernel; ie, completely bypass the L1 cache. This is the case in [6] where a 6 by 6 register block for the `C` matrix can be used as this processor has 64 (32 dual SIMD) floating point registers. To do L0 register blocking we can concatenate tiny submatrices to faciltate streaming by reducing the number of streams. In effect, at the L0 level we have a concatenation of tiny submatrices behaving like a single long stride one vector that passes through L1 and into L0 in an optimal way. Sections 2, 2.1 and 2.2 give details about this technique. Using this extra level of blocking does not negate the benefits of using Square Blocks (SB). It is still essential that $\texttt{NB}^2$ elements of a SB be contiguous. However, the SBs are now no longer two dimensional Fortran or C arrays. We define a SB as *simple* when it is a two dimensional Fortran or C array. Using non-simple SBs as described here and in Section 2 allows us to claim that data copy for DLAFAs using SBs can be O(N^2) instead of O(N^3) which occurs when using Fortran or C two dimensional arrays.

Section 3 describes SB format for symmetric and triangular arrays. In this case one gets SBP format. Section 3.1 explains that SBP format is just as easy to use and to code for as is using standard full format for the same two purposes. Section 3.2 demonstrates a typical performance improvement one gets using simple SBP format over using standard full format. Similar performance results are attainable for non-simple SB, see [6]. Section 4 contains our main result about the reduction from $O(N^3)$ to $O(N^2)$ of data copy that is possible by using NDS; ie, either SB or SBP data format. The background material for this result is developed in Sections 2 and 3.

2 The Need to Reorder a Contiguous Square Block

NDS represent a matrix A as a collection of SB's of order `NB`. Each SB is contiguous in memory. In [23] it is shown that a contiguous block of memory maps best into L1 cache as it minimizes L1 and L2 cache misses as well as TLB misses for matrix multipy and other common row and column matrix operations. When using standard full format on a DLAFA one does an $O((N/NB)^2)$ amount of data copy in calling `DGEMM` in an outer do loop: `j=0,N-1,NB`. Over the entire DLAFA this becomes $O((N/NB)^3)$.

On some processors there are floating point multiple load and store instructions associated with the multiple floating point operations; see [1,6]. A multiple load / store operation requires that its multiple operands be contiguous in memory. The multiple floating point operations require their register operands to be contiguous; eg, see [6]. So, data that enters L1 may also have to be properly ordered to be able to enter L0 in an optimal way. Unfortunately, layout of a SB in standard row / column major order may *no longer* lead to an optimal way. In some cases it is sufficient to reorder a SB into submatrices which we call register blocks. Doing this produces a new data layout that will still be contiguous in L1 but can also be loaded into L0 from L1 in an optimal manner. Of course, the order and size in which the submatrices (register blocks) are chosen will be platform dependent.

2.1 A `DGEMM` Kernel Based on Square Block Format Partitioned into Register Blocks

In this contribution register blocks can be shown to be submatrices of a SB. This fact is important as it means that one can address these blocks in Fortran and C. To see this let A, B and C be three SB's and suppose we want to apply `DGEMM` to A, B and C. We partition A, B and C into conformable submatrices that are also register blocks. Let the sizes of the register blocks (submatrices) be $\texttt{kb} \times \texttt{mb}$, $\texttt{kb} \times \texttt{nb}$ and $\texttt{mb} \times \texttt{nb}$. Thus A^T, B and C are partitioned matrices of sizes $k_1 \times m_1$, $k_1 \times n_1$ and $m_1 \times n_1$ respectively.

The `DGEMM` kernel we want to compute is $C = C - A^T B$ where matrix multiply is stride one across the rows and columns of A and B respectively. (A^T will be

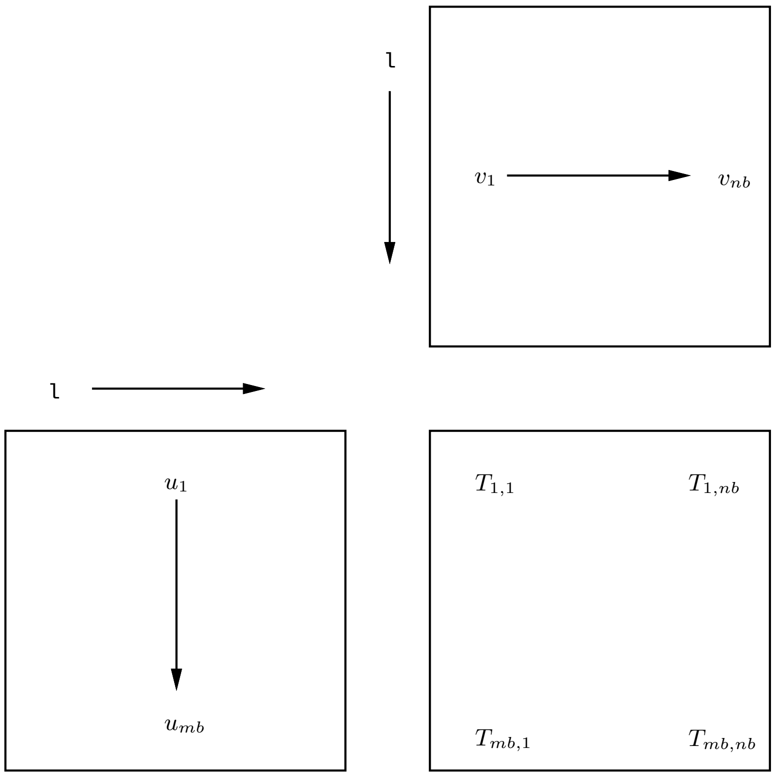


Fig. 1. Fundamental `GEMM` Kernel Building Block

stride one along rows as A is stride one along its columns.) Next, consider a fundamental building block of this `DGEMM` kernel; see Figure 1 It consists of multiplying k_1 register blocks of A^T by k_1 register blocks of B and summing them to form the update of a register block of C. The entire kernel will consist of executing $m_1 \times n_1$ fundamental building blocks in succession to obtain a near optimal kernel for `DGEMM`.

2.2 A Fundamental `DGEMM` Kernel Building Block and Hardware Streaming

If we use simple SB format we would need `mb` rows of A^T and `nb` columns of B and C to execute any fundamental building block. This would require $\texttt{mb} + 2\texttt{nb}$ stride one streams of matrix data to be present and working during the execution of a single building block. Many architectures do *n*ot possess special hardware to support this number of streams. Now the minimum number of streams is three; one each for matrix operands A, B and C. Is three possible? An answer emerges if one is willing to change the data structure away from simple SB order.

In Figure 1 we describe a data layout of a fundamental register block computation [1]. Initially, a register block of C is placed in $\mathtt{mb} \times \mathtt{nb}$ floating point registers $\mathtt{T(0:mb-1,0:nb-1)}$. An inner `do loop` on `l=0:K-1,kb` consists of performing K/kb sets of $mb \times nb$ independent dot products on `T`. For a given single value of `l`, vectors `u,v` of lengths `mb,nb` from A and B respectively are used to update $\mathtt{T} = \mathtt{T}-\ uv^T$. This update is a `DAXPY` outer product update consisting of $mb \times nb$ independent Floating Multiply-Adds (FMAs). However, and this is important, since the `T`'s are in registers there are *no* loads and stores of the `T`'s. The entire update is $\mathtt{T} = \mathtt{T}-\ \mathtt{A}^{\mathtt{T}}(\mathtt{0:K-1,i:i+mb-1}) \times \mathtt{B(0:K-1,j:j+nb-1)}$. If A and B were simple SB's we would need to access vectors `u,v` with stride `NB` and also there would be $\mathtt{mb}+\mathtt{nb}$ streams. Luckily, if we transpose $\mathtt{K} \times \mathtt{mb}\ A^T$ and $\mathtt{K} \times \mathtt{nb}\ B$ we will simultaneously access `u,v` stride one, just get *t*wo streams, and still be able to address A, B in the standard way. These two transpositions accomplish a matrix data rearrangement that allows for an excellent L1 / L0 interface of matrix data for the `DGEMM` kernel fundamental building block computation. We have just demonstrated that two streams are possible for A, B. By storing C as $m_1 \times n_1$ register blocks (submatrices) contiguously in a contiguous SB in the order they are accessed by the `DGEMM` kernel we will get a single stream for C.

3 SB Packed Formats Generalize Standard Full and Packed Formats

Square Block Packed (SBP) formats are a generalization of packed format for triangular arrays. They are also a generalization of full format for triangular arrays. A major benefit of the SBP formats is that they allow for level 3 performance while using about half the storage of the full array cases. For simple SBP formats of a triangular matrix A there are two parameters `TRANS` and `NB`, where usually $n \geq \mathtt{NB}$. For these formats, we first choose a block size, `NB`, and then we lay out the matrix elements in submatrices of order `NB`. Each SB can be in column-major order (`TRANS` = 'N') or row-major order (`TRANS` = 'T'). These formats support both `uplo` = 'L' or 'U'; we only cover the case `uplo` = 'L'. For `uplo` = 'L', the first vertical stripe is n by `NB` and it consists of n_1 SBs where $n_1 = \lceil n/\mathtt{NB} \rceil$. It holds the first trapezoidal n by `NB` part of L. Here we rename matrix A matrix L to remind the reader that our format is lower triangular. The next stripe has $n_1 - 1$ SBs and it holds the next trapezoidal n - `NB` by `NB` part of L, and so on, until the last stripe consisting of the last leftover triangle is reached. The total number of SBs, `nt1`, is $n_1(n_1+1)/2$ and the total storage of SBP format is $nt1 * NB^2$. In [3] we introduced Lower Hybrid SBP format in which the diagonal blocks were stored in packed format. Thus, no additional storage was required. We also provided a fast means to transform to this format using a buffer of size `n*NB`. Now we turn to full format storage. To get SBP format one simply sets $\mathtt{NB} = n$; ie, SBP format gives a single block triangle which happens to be full format.

[1] Compilers require that scalars be used to designate register usage. Also, we are using origin 1 in Fig. 1 and origin 0 in the text of Section 2.2.

3.1 Benefits of SB and SB Packed Formats

We believe a main use of SB formats is for symmetric and triangular arrays. We call these formats SB Packed (SBP). An innovation here is that one can translate, verbatim, standard packed or full factorization algorithms into a corresponding SBP format algorithm by replacing each reference to an i, j element of A by a reference to its corresponding SB submatrix. Because of this storage layout, the beginning of each SB is easily located. Another key feature of using SB's is that SBP format supports Level 3 BLAS. Hence, old, packed and full codes are easily converted into SBP format level 3 code. Therefore, one keeps "standard packed or full" addressing so the library writer/user can handle his own addressing in a Fortran/C environment. Figure 2 describes a RLA for block Cholesky factorization and illustrates what we have just said. For clarity, we assume that `n` is a multiple of `nb`. Lines 2, 4, 7 and 9 of Figure 2 are calls to kernel routines.

```
do j = 0, n-nb, nb
  factor a(j:j+nb-1,j:j+nb-1) ! kernel routine for potrf
  do i = j + nb, n-nb, nb
    a(i:i+nb-1,j:j+nb-1) =
      a(i:i+nb-1,j:j+nb-1)*aT(j:j+nb-1,j:j+nb-1) ! kernel trsm
  end do
  do i = j +nb, n-nb, nb  ! THE UPDATE PHASE
    a(i:i+nb-1,i:i+nb-1) = a(i:i+nb-1,i:i+nb-1) -
      a(i:i+nb-1,j:j+nb-1)*aT(i:i+nb-1,j:j+nb-1) ! kernel syrk
    do k = i + nb, n-nb, nb ! The Schur Complement update phase
      a(k:k+nb-1,i:i+nb-1) = a(k:k+nb-1,i:i+nb-1) -
        a(k:k+nb-1,j:j+nb-1)*aT(i:i+nb-1,j:j+nb-1) ! kernel gemm
    end do
  end do
end do
```

Fig. 2. Block Version of Right Looking Algorithm for Cholesky Factorization

3.2 Performance for SBP Cholesky

Performance results of SBP format for Cholesky factorization were taken from [17]. We only include one of the two graphical plots. To be fair we show the curves for Block Hybrid Cholesky which includes the cost of doing a data transformation from packed format to SBP format. For small N this cost is large, so we reduced this cost to zero by writing a Cholesky factor kernel for packed format; to distinguish this fact we call the resulting code with change over to SBP format BHC code. In Figure 3 the graphs plot MFlops versus matrix order N. Note that the x-axis is log scale; we let N range from 10 to 2000. In the comparison for BHC versus LAPACK we give four graphs: BHC, BHC + data transformation, `DPOTRF` and `DPPTRF`; we name these curves 1, 2, 3, 4. Data for the graphs were obtained on a 200 MHz IBM Power 3 with a peak performance of

800 MFlops. The performance of the BHC Cholesky algorithm of Figure 3 shows the data transformation does cost something. The actual crossover between the packed kernel and SBP format plus data transformation occurred at $N = 230$. For $N \leq 230$ curves 1 and 2 are identical. For $N \geq 230$ it pays to do the data transformation and the curves 1 and 2 separate. Curve 2 is faster than curve 3 for small N (up to four times faster) and more than 10 % faster for large N.

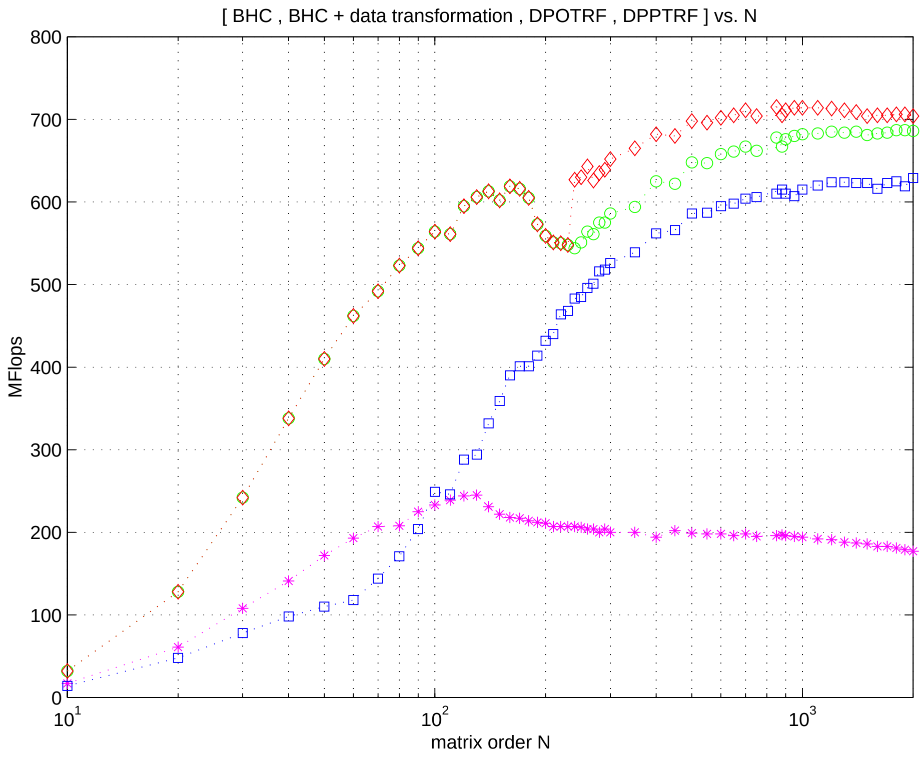


Fig. 3. Four Performance curves: diamond, circle, square, +

4 DLAFA's Using SB Format Require $O(n^2)$ Data Copy

We show this result by demonstrating it for a symmetric factorization as our focus is on Cholesky factorization and what we say about these factorizations applies to many other DLAFA's [8,15,12,11]. There are many Cholesky DLAFA's. We only mention left and right looking as well as hybrid and recursive [7,4,15, 16,2,3] ones. A (left, right) looking algorithm does the (least, most) amount of computation in the outer `do loop` of stage `j`, respectively; see Figure 2 where we use a right looking algorithm. A recursive algorithm uses the divide-and-conquer paradigm. A hybrid algorithm is a combination of left and right looking algorithms. The current version of LAPACK [4] uses a hybrid algorithm. The paper [3] examines some of these algorithm types using a variant of SBP format, packed recursive and standard full and packed formats. Performance studies on six platforms, Alpha, IBM P4, Intel x86, Itanium, SGI and SUN were made. Overall, the hybrid algorithm, using a variant of SB format, was best. However, it was not a clear winner. In [3], we did *not* call BLAS kernel routines. Instead, we either called the vendor or Atlas BLAS [25]. So, these BLAS probably did $O(N^3)$ data copy during Cholesky factorization.

4.1 Data Copy of a DLAFA Can Be $\mathbf{O(N^2)}$

The result we now give holds generally for Right Looking Algorithms (RLAs) for DLAFAs. And similar results hold for Left Looking Algorithms (LLAs). Here we shall be content with demonstrating that the Cholesky RLA on SBP format can be done by only using $\mathrm{O}(N^2)$ data copies. The $\mathrm{O}(N^3)$ part of the block Cholesky RLA has to do with the Schur Complement Update (SCU); ie, the inner `DGEMM do loop` over variable `k`; see Figure 2. We assume each call to `DGEMM` will do data copy on each of its three operands A, B and C. Now the number of C SB's that get SCUed over the entire RLA is $n_1(n_1+1)(n_1-1)/6$ where $n_1 = \lceil \mathtt{N/NB} \rceil$ and `N` is the order of A. It is therefore clear that $\mathrm{O}(N^3)$ data copies will occur.

In the case of simple SBs our result is obvious as no data copy occurs during execution of the RLA algorithm in Figure 2 because kernel routines of the BLAS are being called. So, there is only an initial reformating cost of full format A to SBP format of A, which is clearly $O(N^2)$. Also, as mentioned in Section 3 and 3.2 this initial reformating is optional. Now, consider non-simple SBs. In Sections 2.1 and 2.2 we indicated that it is now usually necessary to reformat each SB every time `DGEMM` is called if non-simple SB's are used. We now demonstrate that we can reduce this data copy cost to $\mathrm{O}(N^2)$. What we intend to do is to store the C operands of `DGEMM` in the register block format that was indicated in Sections 2.1 and 2.2. Hence, the format of these C operands is then fixed throughout this algorithm and no additional data copy occurs for them during the entire execution of this RLA; see Figure 2. And clearly, an initial formatting cost, if necessary, is only $\mathrm{O}(N^2)$. Now we examine the A and B operands of the SCU for the outer loop variable `j`. SB's $A(\mathtt{j} : \mathtt{n}_1, \mathtt{j})$ whose total is $\mathtt{n}_1 - \mathtt{j}$ are needed for the SCU as they constitute all the A, B operands of the SCU at iteration `j`. Summing from `j=1` to $\mathtt{j} = \mathtt{n}_1 - \mathtt{1}$ we find just $\mathtt{n}_1(\mathtt{n}_1 - \mathtt{1})/2$ SB's in all that need reformatting (data copying) over the course of this entire RLA; see Figure 2. And since there are both A and B operands we may have to double this amount to $n_1(n_1-1)$ SB's. However, in either case this amount of data copy is clearly $\mathrm{O}(N^2)$.

5 Summary and Conclusions

This paper demonstrates that the standard data structures of DLA can hurt the perfomance of its factorization algorithms. It indicates that by using NDS this performance loss can be lessened. Specifically, it describes SB and SBP format as a replacement of these standard data structures. SB and SBP data structures are shown to be easy to use and to code for. These two features are strong features of the standard data structures of DLA. SB and SBP formats have two desirable features that the standard data structures lack. SBP format uses near minimal storage for symmetric and triangular matrices whereas standard full format storage uses nearly double the minimal storage. Secondly, SB and SBP formats give DLAFAs better performance than standard full format does. Our main result, that DLAFAs require only $O(N^2)$ data copy, indicates partly why this is so. The use of standard full format requires $O(N^3)$ data copy by the level 3 BLAS being used by the DLAFA. We assumed these BLAS always did data copy on their submatrix operands. We discussed a new concept called the L1 / L0 cache interface. The existence of this interface showed one the necessity of introducing non-simple SBs in order to maintain high performance of `DGEMM` kernels on several new platforms. These non-simple SBs were able to fully exploit hardware streaming which is a feature of several new platforms.

References

1. Agarwal, R.C., Gustavson, F.G., Zubair, M.: Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. IBM Journal of Research and Development 38(5), 563–576 (1994)
2. Andersen, B.S., Gustavson, F.G., Wasnieski, J.: A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage. ACM TOMS 27(2), 214–244 (2001)
3. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Wasnieski, J.: A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. ACM TOMS 31(2), 201–227 (2005)
4. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide Release 3.0, SIAM, Philadelphia (1999), `http://www.netlib.org/lapack/lug/lapack_lug.html`
5. Bilmes, J., Asanovic, K., Whye Chin, C., Demmel, J.: Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In: Proceedings of International Conference on Supercomputing, Vienna, Austria (1997)
6. Chatterjee, S., et al.: Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. IBM Journal of Research and Development 49(2-3), 377–391 (2005)
7. Dongarra, J.J., Moler, C.B., Bunch, J.R., Stewart, G.W.: LINPACK Users' Guide Release 2.0. SIAM, Philadelphia (1979)
8. Dongarra, J.J., Gustavson, F.G., Karp, A.: Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. SIAM Review 26(1), 91–112 (1984)
9. Dongarra, J.J., Du Croz, J., Hammarling, S., Hanson, R.J.: An Extended Set of FORTRAN Basic Linear Algebra Subprograms. TOMS 14(1), 1–17 (1988)
10. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms. TOMS 16(1), 1–17 (1990)

11. Elmroth, E., Gustavson, F.G., Kagstrom, B., Jonsson, I.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)
12. Gunnels, J., Gustavson, F.G., Henry, G., van de Geijn, R.: Formal linear algebra methods environment (FLAME). ACM TOMS 27(4), 422–455 (2001)
13. Gunnels, J.A., Gustavson, F.G.: A New Array Format for Symmetric and Triangular Matrices. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 247–255. Springer, Heidelberg (2006)
14. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: A Family of High-Performance Matrix Multiplication Algorithms. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 256–265. Springer, Heidelberg (2006)
15. Gustavson, F.G.: Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. IBM Journal of Research and Development 41(6), 737–755 (1997)
16. Gustavson, F.G., Jonsson, I.: Minimal Storage High Performance Cholesky via Blocking and Recursion. IBM Journal of Research and Development 44(6), 823–849 (2000)
17. Gustavson, F.G.: High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. IBM Journal of Research and Development 47(1), 31–55 (2003)
18. Gustavson, F.G.: New Generalized Data Structures for Matrices Lead to a Variety of High performance Dense Linear Algorithms. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 11–20. Springer, Heidelberg (2006)
19. Gustavson, F.G., Wasniewski, J.: Rectangular Full Packed Format for LAPACK Algorithms Timings on Several Computers. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 570–579. Springer, Heidelberg (2007)
20. IBM: IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. IBM Pub. No. SA22-7272-04 (December 2001)
21. Kalla, R., Sinharoy, B., Tendler, J.: Power 5. HotChips-15, August 17-19, 2003, Stanford, CA (2003)
22. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic Linear Algebra Subprograms for Fortran Usage. TOMS 5(3), 308–323 (1979)
23. Park, N., Hong, B., Prasanna, V.K.: Tiling, Block Data Layout, and Memory Hierarchy Performance. IEEE Trans. Parallel and Distributed Systems 14(7), 640–654 (2003)
24. Sinharoy, B., Kalla, R.N., Tendler, J.M, Kovacs, R.G., Eickemeyer, R.J., Joyner, J.B.: POWER5 System Microarchitecture. IBM Journal of Research and Development 49(4/5), 505–521 (2005)
25. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing (1-2), 3–35 (2001)

Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage

Fred G. Gustavson[1,2], Lars Karlsson[2], and Bo Kågström[2]

[1] IBM T. J. Watson Research Center,
Yorktown Heights, NY 10598, USA
fg2@us.ibm.com
[2] Department of Computing Science and HPC2N, Umeå University,
SE-901 87 Umeå, Sweden
{larsk, bokg}@cs.umu.se

Abstract. We present three algorithms for Cholesky factorization using minimum block storage for a distributed memory (DM) environment. One of the distributed square block packed (SBP) format algorithms performs similar to ScaLAPACK PDPOTRF, and our algorithm with iteration overlapping typically outperforms it by 15–50% for small and medium sized matrices. By storing the blocks contiguously, we get better performing BLAS operations. Our DM algorithms are not sensitive to cache conflicts and thus give smooth and predictable performance. We also investigate the intricacies of using rectangular full packed (RFP) format with ScaLAPACK routines and point out some advantages and drawbacks.

1 Introduction

Dense linear algebra routines that are implemented in a distributed memory environment typically use a 2D block cyclic layout (BCL), with ScaLAPACK being one example of a library that uses BCL for all routines [3]. A BCL can provide effective load balance for many algorithms. The mapping of matrix elements to processors does not prescribe how they are later stored on each processor. The approach taken by the ScaLAPACK library is to store each elementary block as a submatrix of a column major 2D array (standard Fortran array) [3]. Another approach is to store each elementary block contiguously, for example as a column major block 2D array.

Storing elementary blocks contiguously has at least three advantages. They will map very well into L1 cache and level 3 operations involving such blocks will therefore tend to achieve high performance and minimize memory traffic. Another benefit is that moving a block can be done by one contiguous memory transfer. In this contribution we use *square elementary blocks* (called a square block, or SB) to store the local matrix. Furthermore, we store only the triangular part of the block matrix to achieve minimum block storage for symmetric matrices. We call this *square block packed* (SBP) format.

We identify an inefficiency in straightforward data parallel implementations, e.g., the implementation of the Cholesky factorization in ScaLAPACK (routine `PDPOTRF`) and develop an iteration overlapping data parallel implementation which removes much of the idling and thus decreases execution time.

2 Near Minimal Storage in a Serial Environment

A recently proposed format for storing triangular or symmetric matrices is called *rectangular full packed* (RFP) (see [8] for details). This format takes many slightly different forms. Figure 1 illustrates a lower triangular matrix. The matrix is

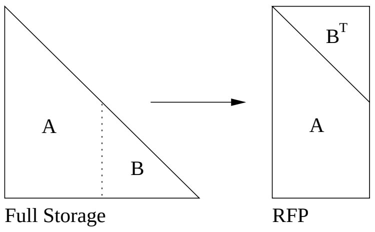


Fig. 1. Illustration of rectangular full packed format

partitioned into two submatrices A and B. The triangular matrix B^T is merged along the diagonal with A. As can be seen, this new matrix can be stored as a standard full format rectangular array with no waste of memory.

Another format for near minimal storage is a generalization of a standard column major format. The matrix is divided into square blocks and the format is based on storing each such block in a contiguous memory area. The blocks can then be stored for example in either a row or column major ordering. Figure 2

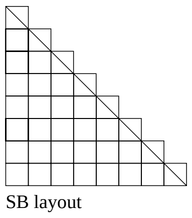


Fig. 2. Illustration of square block packed format

illustrates the square blocks. The elements above the diagonal of the diagonal blocks are wasted storage. By picking the block size to one, we see that we get either the standard row or column major format. For details on this format see [7].

3 Minimum Block Storage in a Distributed Environment

In this section we describe how RFP and SBP can be used in a distributed memory environment. We show how both approaches give a nearly minimum block storage.

3.1 A Distributed SBP Algorithm for Cholesky Factorization

In this contribution we consider the blocked algorithmic variant of Cholesky factorization described in Algorithm 1. We note that this algorithmic variant is used in ScaLAPACK [4]. In a distributed environment with a 2D block cyclic

Algorithm 1. Standard blocked Cholesky factorization

```
1: for each panel left to right do
2:     Partition $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11}$ is NB×NB
3:     Factorize $A_{11} = LL^T$ using unblocked algorithm
4:     Update panel $A_{21} := A_{21}L^{-T}$ using triangular solver
5:     Update trailing matrix $A_{22} := A_{22} - A_{21}A_{21}^T$ using symmetric rank-$k$ update
6:     Continue with $A = A_{22}$
7: end for
```

layout with block size `NB`×`NB`, block A_{11} resides on one processor, block A_{21} on one processor column, and A_{22} generally resides on all processors. By using parallel triangular solve and symmetric rank-k update routines Algorithm 1 will achieve scalable performance due to good load balance and because most of the computation is in step 5 which is easy to parallelize. However, steps 3 and 4 do not utilize all processors effectively. One variant of this algorithm is to start the next iteration before the current iteration has finished step 5 (see [7] for more details). This is possible by noting that the first updated column panel of the new pivot from step 5 will be used as the only input for step 3 and 4 of the next iteration.

A major problem with a straightforward parallel implementation of Algorithm 1 is the idle time introduced when processors implicitly synchronize after each iteration. This idle time is caused both by slight load imbalances and the work in steps 3 and 4 that are not performed on all processors. By using the iteration overlapping algorithm this idle time will be eliminated if the communication of data between steps 3 and 4 can be carried out while still doing useful work in updates.

The data dependencies in Algorithm 1 are simple. Output from step 3 is input for step 4 whose output in turn is input for step 5. As for the first dependency a column broadcast is all that is needed. The second dependency requires a somewhat more complicated communication pattern and is now described briefly. All subblocks of A_{21} are broadcasted along the processor rows. Once a subblock of

A_{21} reaches the processor holding the diagonal block of that row it is broadcasted along its processor column. One can show that after this, each processor holds the blocks of A_{21} and A_{21}^T that it needs for step 5. In our implementation these blocks are stored in two block buffer vectors W and S, where W (for West border vector) holds blocks of A_{21} and S (for South border vector) holds blocks of A_{21}^T.

We have studied how overlapping two successive pivot steps can affect the performance of our parallel implementation. Our implementation is described in Algorithm 2. The overlapping in Algorithm 2 happens during the execution

Algorithm 2. Cholesky with iteration overlap

```
 1: for each panel left to right do
 2:    Partition global $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11}$ is NB×NB
 3:    if process holds $A_{11}$ then
 4:       Factorize $A_{11} = LL^T$ using serial algorithm
 5:       Column broadcast the block $L$
 6:    end if
 7:    if process column holds $A_{21}$ then
 8:       Receive the block $L$
 9:       Partition $S_1 = \begin{bmatrix} S_A & S_B \end{bmatrix}$, where $S_A$ is NB×NB
10:       Update $A_{21} := A_{21} - W_1 S_A$
11:       Scale $A_{21} := A_{21} L^{-T}$
12:       Start communication of $A_{21}$ using buffers $W_2$ and $S_2$ (sender)
13:       Update $A_{22} := A_{22} - W_1 S_B$
14:    else {all other process columns}
15:       Start communication of $A_{21}$ using buffers $W_2$ and $S_2$ (receiver)
16:       Update $A := A - W_1 S_1$
17:    end if
18:    Move (symbolically) $W_1 := W_2$ and $S_1 := S_2$ {there is no data movement}
19: end for
```

of steps 10 to 13. Taken together, steps 10 and 13 perform a complete update. The execution order of the straightforward algorithm would put steps 10 and 13 together, and step 11 after both. Executing step 11 before 13 allows the communication needed for the subsequent update to take place during the update in step 13 (and while the other processors execute step 16).

In Figure 3, we illustrate by example how our local matrices are stored in practice. The blocks are stored columnwise in a one-dimensional block vector indexed by a column pointer (CP) array. The entries in CP are pointers to the first block of each block column.

Figure 4 shows how the two sets of buffers are used in Algorithm 2. The light shaded blocks are those used for the update of iteration i. The darker shaded blocks are those computed during iteration i for use in iteration $i + 1$. After the panel factorization the communication algorithm is started and it will broadcast the panel and its transpose to all processors with this data stored in the second

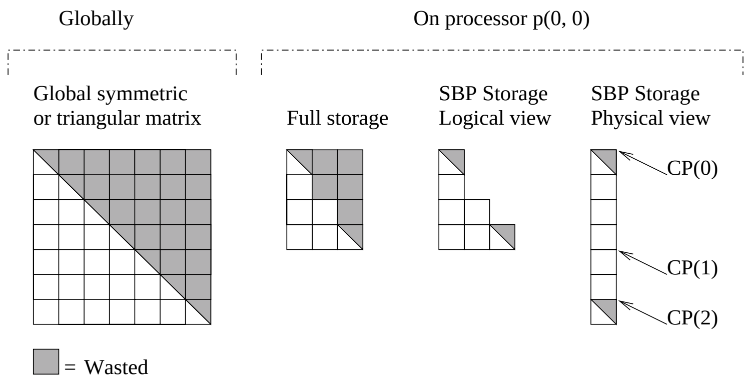


Fig. 3. Illustration of how a 7×7 block global matrix is laid out on a 2×3 mesh in SBP format and addressed with its column pointer (CP) array. The full size of the global matrix is `7NB`×`7NB`.

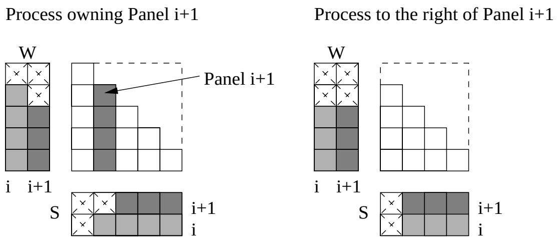


Fig. 4. Data layout for the SBP with double sets of W and S border vectors

set of buffers. While this communication takes place, the first set of buffers is used to finish the update of iteration i.

3.2 A Distributed RFP Algorithm for Cholesky Factorization

Because of the good performance achievable with RFP format in a serial environment (see [9]) we investigated its extension to parallel environments via using ScaLAPACK and PBLAS. Algorithm 3 gives the details of the RFP Cholesky algorithm. The limitations of PBLAS and ScaLAPACK do not generally allow matrices to begin inside an elementary block; each submatrix must be block aligned. Therefore, we use the RFP format on the block level, introducing some wasted storage and thus achieve minimum block storage while still being able to use RFP with existing routines.

The RFP format could be used with an algorithm similar to the one we used with SBP. Such an RFP algorithm would probably achieve similar performance to the SBP algorithm so we did not develop any implementation of it.

Algorithm 3. RFP Cholesky with ScaLAPACK/PBLAS routines

```
1: Matrix $A$ is in RFP format: $A = \begin{bmatrix} A_{11} \backslash A_{22}^T \\ A_{21} \end{bmatrix}$
2: Factor $A_{11} = LL^T$ using ScaLAPACK routine PDPOTRF
3: Update panel $A_{21} := A_{21}L^{-T}$ using PBLAS routine PDTRSM
4: Update trailing matrix $A_{22} := A_{22} - A_{21}A_{21}^T$ using PBLAS routine PDSYRK
5: Factor $A_{22} = LL^T$ using ScaLAPACK routine PDPOTRF
```

4 Related Work on DM Cholesky Factorization

We briefly discuss other packed storage schemes for DM environments.

D'Azevedo and Dongarra suggested in 1997 a storage scheme where the elementary blocks are mapped to the same processor as in the full storage case, but only the non-redundant blocks are stored [6]. Each block column is stored as a submatrix the same way as it would in full storage. The result is that each block column is a regular ScaLAPACK matrix and can be used as such. Note that the blocks will be mapped to the same processors as the SBP format, but the local processor storage layout is different. Benefits include routine reuse via PBLAS and ScaLAPACK routines. However, some new PBLAS routines seem to be required to handle the packed storage [6]. Furthermore, their results indicate that the performance varies wildly with input, making performance extrapolation difficult.

Recently, Marc Baboulin et al. presented a storage scheme which uses relatively large square blocks consisting of at least LCM(p, q)[1] elementary blocks [2]. This format also supports code reuse via PBLAS and ScaLAPACK. The granularity is limited to the distributed block size, which means less possibility to save memory. For the Cholesky factorization routines, the chosen block sizes for performance measurements were between 1024 and 10240. This resulted in a departure from their minimum storage by as much as 7–13%. Using their minimum allowed distributed block size would bring this percentage down to about 1–3% but at the cost of longer execution times.

5 Performance Results and Comparison

In this section we give some performance related results. We compare RFP, SBP and ScaLAPACK routines and analyze the differences that we observed.

All tests were performed on the *Sarek* cluster at HPC2N. It consists of 190 HP DL145 nodes, with dual AMD Opteron 248 (2.2GHz) processor and 8 GB memory per node. The AMD Opteron 248 processor has a 64 kB instruction and 64 kB data L1 Cache (2-way associative) and a 1024 kB unified L2 Cache (16-way associative). The cluster's operating system is Debian GNU/Linux 3.1 and we used Goto BLAS 0.94 throughout.

[1] The least common multiple of the integers a and b (written LCM(a, b)) is the smallest integer that is a multiple of both a and b.

Table 1. Execution times for PDPOTRF and the SBP algorithm with iteration overlap for various square grid sizes. The block size NB is set to 100.

N	2x2	3x3	4x4	5x5	6x6	7x7
4000	2.13/0.86	1.48/0.63	1.04/0.66	0.79/0.68	0.63/0.64	0.57/0.65
8000	14.80/0.92	8.29/0.80	5.33/0.79	3.97/0.77	3.15/0.71	2.64/0.73
12000		25.20/0.83	16.30/0.80	10.90/0.84	8.27/0.80	7.11/0.78
16000		57.30/0.84	34.50/0.85	24.00/0.85	18.30/0.80	13.90/0.85
20000			65.00/0.85	43.90/0.86	33.00/0.81	25.90/0.84
24000					53.90/0.84	42.30/0.85

Table 1 shows selected times for both PDPOTRF and the SBP algorithm with iteration overlap. Each cell has the form X/y, where X is the time (in seconds) of the PDPOTRF routine and y = Y/X, where Y is the time for the SBP algorithm. The same block size was used for both implementations. We identify two trends. First of all, the relative gain by overlapping increases with the number of processors since the idle time is introduced on the entire mesh. The bigger the mesh the more idle time we can remove by overlapping. Second, the relative gain decreases with increasing problem sizes. This is expected because the dominant operation is the trailing matrix update (with $\mathcal{O}\left(N^3\right)$ flops) whereas the operations causing idle time (the panel factorization) make up for only $\mathcal{O}\left(N^2\right)$ flops.

Table 2. Execution time for PDPOTRF and the RFP algorithm using ScaLAPACK routines for various grid sizes

N	2x2	3x3	4x4	5x5	6x6	7x7
4000	2.13/1.26	1.48/1.16	1.04/1.18	0.79/1.44	0.63/1.42	0.58/1.34
8000	14.80/*1.48*	8.29/1.25	5.33/1.23	3.97/1.37	3.15/1.33	2.64/1.33
12000		25.20/*1.41*	16.30/1.14	10.90/1.34	8.27/1.33	7.11/1.29
16000		57.30/1.16	34.50/*1.34*	24.00/1.28	18.30/1.20	13.90/1.34
20000			65.00/1.13	43.90/*1.40*	33.00/1.22	25.90/1.25

Table 2 is similar to Table 1 but shows selected times for PDPOTRF and our RFP algorithm which uses four calls to ScaLAPACK/PBLAS routines. Each cell has the form X/y, where X is the time (in seconds) of the PDPOTRF routine and y = Y/X, where Y is the time for the RFP algorithm. As can be seen from this table the RFP algorithm has typically a 10–30% longer execution time. By tracing the execution of the algorithm we found two substantial causes for this overhead. The performance of the BLAS operations issued by the RFP algorithm was less efficient than was typical for the other algorithms we tested. Moreover, there are more synchronization points in the RFP algorithm due to the two ScaLAPACK and two PBLAS calls on problems half the size. This amplifies the communication overhead and load imbalance. Taken together, this would probably explain most of the time differences we observed. One interesting detail to note in Table 2 is that when the local matrix dimension is 4000 the RFP algorithm experienced a dramatic loss in performance (emphasized by italics in Table 2). This is caused by a cache effect because the leading dimension is actually 4100 which is close to $2^{12} = 4096$; also the L1 cache on Sarek is only 2-way set associative.

The block size mainly affects performance of the BLAS operations and the load balance. Larger blocks tend to give good BLAS performance but less load balance. For the SBP algorithm the block size is intimately related to BLAS performance because then all `GEMM` calls are on matrices of order `NB`. The ScaLAPACK algorithm is less dependent on the block size because of the fewer and larger PBLAS operations. Table 3 gives an idea of how the block size relates to

Table 3. Impact of block size on performance (measured in Gflops/s per processor) for ScaLAPACK `PDPOTRF` and our overlapping SBP algorithm

NB	PDPOTRF	Overlapping
25	2.08	1.91
50	2.12	2.57
75	2.09	2.75
100	2.15	3.04
125	2.24	3.06
150	2.13	3.04

performance for both of these algorithms. The processor mesh was 2×3 and the order of the matrix was `N`=6000. On Sarek we see that when we approach a block size of 100 we get close to optimal performance, whereas the block size does not matter much for the ScaLAPACK routine. The gap in performance between the two routines is mainly due to less idling in the overlapping routine.

Finally, we note that our overlapping SBP algorithm could be modified so that it updates first and factorizes the next panel afterwards. This makes the algorithm essentially equal to the straightforward implementation but with a different data format. We implemented this variant too and found that as expected it gave performance nearly identical to the ScaLAPACK algorithm.

6 Future Work

We outline some future directions of development. Our overlapping algorithm relies on the idea that the task of trailing matrix update can be divided into two tasks: the first panel on the column of processors holding the pivot and the rest of the panels on all processors. This allows us to have two iterations on the same processor, but three is not possible. A solution is to further divide the tasks. The trailing matrix update could for example be divided into one task for each block column. Instead of waiting for data it now becomes attractive to do smaller tasks instead. The order of the tasks thus becomes non-deterministic because it would depend on processor interactions. To get a clean implementation it might be necessary to use a style reminiscent of a work pool.

The overlapping algorithm relies heavily on the interleaving of communication and updates. One consequence of the overlapping is that more workspace is needed. In general each ongoing iteration will require its own W and S buffer. It is preferable to have many iterations ongoing because in that way more work is kept at each processor and chances for idling will get reduced. The concept of lookahead in factorization algorithms has been addressed several times (cf. [1,5,7]) and recently in [10]. The emphasis of the latter contribution is that a dynamic lookahead is most appropriate. A large lookahead is not feasible in a DM environment because of the large workspace required. Setting a fixed cap (or dynamic relative to a fixed workspace) on the number of iterations may be a feasible solution.

Our work provides an argument for the inclusion of *nonblocking collective* communication routines in communication libraries. The de-facto industry standard MPI has substantial support for nonblocking point-to-point communication but collectives are all blocking. Our implementation emulates nonblocking collectives by repeatedly testing for individual completion of nonblocking point-to-point operations. This complicates the code and probably comes at a higher cost than would have been the case if nonblocking collectives existed as part of the library.

7 Conclusion

We have implemented and compared three algorithms and data formats for minimum block storage in distributed memory environments using a 2D block cyclic data layout.

In a serial environment, the RFP format is an attractive choice [9]. However, the straightforward generalization of serial RFP algorithms has some weaknesses.

The SBP format was implemented and tested with two algorithm variants. One resembles ScaLAPACK's `PDPOTRF` but makes no use of PBLAS or ScaLAPACK routines, and one overlaps iterations. We have demonstrated that performance at least as good as the ScaLAPACK algorithm is attainable, and for the overlapping variant far better performance, especially for small and medium sized matrices, was achieved.

The ideas that we explored in this work can be applied to many other algorithms as well. Two examples very similar to the Cholesky factorization are the LU and QR factorizations.

Acknowledgements. This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

1. Agarwal, R.C., Gustavson, F.G.: A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In: Wright, M. (ed.) Aspects of Computation on Asynchronous and Parallel Processors, pp. 217–221. IFIP, North-Holland, Amsterdam (1989)
2. Baboulin, M., Giraud, L., Gratton, S., Langou, J.: A distributed packed storage for large parallel calculations. Technical Report TR/PA/05/30, CERFACS, Toulouse, France (2005)
3. Blackford, L.S., et al.: ScaLAPACK user's guide. SIAM Publications (1997)
4. Choi, J., Dongarra, J.J., Ostrouchov, S., Petitet, A.P., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Scientific Programming 5(3), 173–184 (1996)
5. Dackland, K., Elmroth, E., Kågström, B.: A ring–oriented approach for block matrix factorizations on shared and distributed memory architectures. In: Sincovec, R.F., et al. (eds.) SIAM Conference on Parallel Processing for Scientific Computing, pp. 330–338. SIAM Publications (1993)
6. D'Azevedo, E., Dongarra, J.: Packed storage extension for ScaLAPACK. Technical Report UT-CS-98-385 (1998)
7. Gustavson, F.: Algorithm compiler architecture interaction relative to dense linear algebra. Technical Report RC 23715, IBM Thomas J. Watson Research Center (September 2005)
8. Gustavson, F.: New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 11–20. Springer, Heidelberg (2006)
9. Gustavson, F.G., Wasniewski, J.: Rectangular Full Packed Format for LAPACK Algorithms Timings on Several Computers. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 570–579. Springer, Heidelberg (2007)
10. Kurzak, J., Dongarra, J.: Implementing Linear Algebra Routines on Multi-core Processors with Pipelining and a Look Ahead. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 147–156. Springer, Heidelberg (2007)

In-Place Transposition of Rectangular Matrices

Fred G. Gustavson[1] and Tadeusz Swirszcz[2]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
fg2@us.ibm.com
[2] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
swirszcz@mini.pw.edu.pl

Abstract. We present a new Algorithm for In-Place Rectangular Transposition of an m by n matrix A that is efficient. In worst case it is $O(N \log N)$ where $N = mn$. It uses a bit-vector of size `IWORK` words to further increase its efficiency. When `IWORK=0` no extra storage is used. We also review some of the other existing algorithms for this problem. These contributions were made by Gower, Windley, Knuth, Macleod, Laffin and Brebner (ACM Alg. 380), Brenner (ACM Alg. 467), and Cate and Twigg (ACM Alg. 513). Performance results are given and they are compared to an Out-of-Place Transposition algorithm as well as ACM Algorithm 467.

1 Introduction

We present a new algorithm that requires little or no extra storage to transpose a m by n rectangular (non-square) matrix A in-place. We assume that A is stored in the standard storage format of the Fortran and C programming languages. We remark that many other programming languages use this same standard format for laying out matrices. One can prove that it requires $O(N \log N)$ operations in worst case where $N = mn$. It uses a bit-vector of size `IWORK` words to further increase its efficiency. When `IWORK=0` no extra storage is used. When `IWORK = m*n/ws` where `ws` is the word size the algorithm has $O(N)$ complexity.

Matrix A^T is an n by m matrix. Now both A and A^T are simultaneously represented by either A or A^T. Also, in Fortran, A and A^T are stored stride one by column. An application determines which format is best and frequently, for performance reasons, both formats are used. Currently, Dense Linear Algebra libraries do *not* contain in-place transpose algorithms when $m \neq n$.

Our algorithm is based on following the cycles of a permutation P of length $q = mn - 1$. This permutation P is defined by the mapping of A onto A^T that is induced by the standard storage layouts of Fortran and C. Thus, if one follows a cycle of P then one must eventually return to the beginning point of this cycle of P. By using a bit vector one can tag which cycles of P have been visited and then a starting point for each new cycle is easily determined. The cost of this algorithm is easily seen to be $O(q)$ which is minimal. Now, we go further and remove the bit vector. Thus, we need a method to distinguish between a new cycle and a previous cycle (the original reason for the bit vector). A key observation is that every new cycle has a minimum starting value. If we traverse a proposed new cycle and we find an iterate whose value is less than the current starting value we know that the cycle we are generating has already been generated. We can therefore abort and go on to the next starting value. On the other hand, if we return to the original starting value, thereby completing a cycle, where every iterate is larger than this starting value we are assured that a new cycle has been found and we can therefore record it. Our algorithm is based on this simple idea which is originally due to Gower [2,7]. In [7] Knuth does an in depth complexity analysis of Gower's Algorithm as it applied to in-situ permutation in general. Knuth showed that in-situ permutation had an average case complexity of $O(n \log n)$ but worst case of $O(n^2)$. However, in the special case when both P and P^{-1} are known (this includes matrix transposition), the worst case is reduced to $O(n \log n)$. Now, using P and P^{-1} is equivalent to searching a cycle in both directions which we did by the BABE (Burn At Both Ends) programming technique.

References [6,4,8,9,10,13] all emphasize the importance of the fundamental mapping $P(k) = mk \bmod q$. Here $0 < k < q$, $i = \lfloor k/n \rfloor$, $j = k - ni$ and $P(k) = i + mj$ is the location of A_{ij} assuming Fortran storage order and 0-origin subscripting of A_{ij}. A_{ij} moves to $(A^T)_{ij}$ under this mapping.

Our algorithm Modified In-Place Transpose (`MIPT`) is closely related to ACM Alg. 467 [8]. However, algorithm `MIPT` has the following four new features. First, we use the BABE approach which makes our code provably $O(N \log N)$ in worst case. Second, `MIPT` removes a bug in ACM Alg. 467. Third, `MIPT` stores a bit vector in an integer array as opposed to using an integer array to just store 0 or 1. Fourth, the BABE inner loop of `MIPT` has been made more efficient than the inner loop of ACM Alg. 467.

To remove the bug, we did not use the mapping $P(k)$. We used instead the Fortran statement `KBAR=M*K-Q*(K/N)`. The map $P(k)$ can cause destructive integer overflow whereas the Fortran statement does *not*.

Our algorithm `MIPT` stores a bit vector in integer array `MOVE` of size `IWORK` instead of having each element of `MOVE` hold 0 or 1. Thus, this gives a factor of 32 storage gain over ACM Algs. 380, 467, and 513. Our experiments indicate that a fixed size for `IWORK`, say 100 words, is always a good idea. Finally, we have made several other changes to ACM Alg. 467 which we describe in Section 3.

In Section 2, we describe our basic key idea first discovered by Gower and later used by most of our references. In Section 3, we fully describe Alg. `MIPT`. It uses our discovery of a duality result which is a key feature of Alg. `MIPT`. We call attention to Theorem 7 of [9] which proves that if a self dual cycle exists then the dual cycle mechanism used in our Alg. `MIPT` (also in ACM Algs. 467 and 513) meets "in the middle" and so the recording of dual and self cycles can be merged into single piece of code. In Section 4, we prove that $\bar{k} =$ `KBAR`. Section 5 gives performance studies. Section 6 traces the history of research on the subject of in-place transpose.

2 The Basic In-Place Transposition Algorithm IPT

```
ALGORITHM IPT (m,n,A)
DO cnt = 1, mn-2
  k = P(cnt)
  DO WHILE (k > cnt)
    k = P(k)
  END DO
  IF (k = cnt) then
    Transpose that part
    of A which is in the
    new cycle just found
  ENDIF
END DO
```

We have just described Gower's algorithm; see pages 1 and 2 of [2] and page 2 of [7]. The algorithm IPT does extra computation in its inner while loop. Here is an observation. Sometimes the algorithm IPT completes while cnt is still on the first column of A; i.e. before cnt reaches m. However, when P has small cycles this causes cnt to become much larger before IPT completes. Also, one can compute the number of one-cycles in P for any matrix A. This is a greatest common divisor (gcd) computation and there are $1 + gcd(m-1, n-1)$ one cycles. Since non-trivial one cycles always occur in the interior of A; ie, for large values of cnt, knowing their number can sometimes drastically reduce the cost of running IPT. To see this, note that the outer loop of IPT runs from 1 to $mn-2$. If one records the total cycle count so far tcc then one can leave the outer loop when tcc reaches mn. We now rename the modification of IPT that uses the gcd logic and the tcc count our basic algorithm IPT.

3 Descriptive Aspects of Algorithm MIPT

A programming technique called BABE can be used to speed up the inner while loop of IPT. BABE traverses the while loop from both ends and thus requires use of both P and P^{-1}. This additionally guarantees that MIPT will have a worst case complexity of $O(N \log N)$. Use of BABE allows one to use functional parallelism; [11]. More importantly, matrix elements are *not* accessed during the inner while loop and so no cache misses occur. In fact the inner while loop consists entirely of register based fixed point instructions and hence the inner loop will perform at the peak rate of many processors. The actual transposition part of IPT runs very very slowly in comparison to the inner loop processing: Any cycle of P usually accesses the elements of A in a completely random fashion. Hence, a cache miss almost always occurs for each element of the cycle; thus, the whole line of the element is brought into cache and the remaining elements of the line usually never get used. On the other hand, an out-of-place transpose algorithm allows one to bring the elements of A into and out of cache in a fairly structured way. These facts illustrate the principle of "trading storage to increase performance".

Let $\bar{k} = P(k)$ and $l = q - k$. One can show that $P(l) = q - \bar{k}$. Thus, let cnt generate a cycle and suppose that iterate $l = q - \texttt{cnt}$ does not belong to this cycle. Then $q - \texttt{cnt}$ also generates a cycle. This result shows that a duality principle exists for P. The criterion for cnt is $j \geq \texttt{cnt}$ for every j in the cycle. For $q - \texttt{cnt}$ the criterion is $j \leq q - \texttt{cnt}$ for every j in the companion cycle.

The value q is the key to understanding P and hence our algorithm. Let k be the storage location of $A_{ji}, 0 < k < q$. One can show $P(k) = mk \bmod q$. $P(k)$ is the storage location of A_{ij}. If d is a divisor of q, then every iterate of d also divides q. Hence, when cnt starts at d one can alternatively look at $\bar{k} = mod(nk, q/d)$ where cnt begins at 1. So, we can partition $0 < k < q$ into a disjoint union over the divisors of d of q. For each d, we can apply a suitable variant of algorithm IPT, called algorithm MIPT, as a routine that is called for each d of q. This master algorithm drastically reduces the operation count of the inner while loop of algorithm IPT. These remarks also describe several of the features of ACM Alg. 467. Not mentioned is our use of integer array MOVE to hold a bit vector (Alg. 467 uses the elements of its integer MOVE array to hold either a 0 or 1). We mention our clever use of combining the search for a possible new cycle with its dual cycle when such a dual cycle exists. When a dual cycle does not exist, ACM Algs. 467 and 513 use self duality and meet in the middle. However, when a dual cycle exists, the two new cycles are found using just P ; see also Theorem 7 on page 106 of [9]. Thus, to use our BABE approach we needed a fast way to determine, apriori, when a cycle had a dual cycle.

3.1 Apriori Determination of Dual / Self Dual Cycles

The results of Knuth [7] and Fich et. al. [12] allowed us to prove that our BABE approach was sufficient to guarantee an $O(N \log N)$ running time of Algorithm MIPT. However, we need a way to modify the current Algorithm MIPT to do this: Every divisor d of q is a cycle minima. Hence, at no additional cost, one finds CL the cycle length of minima d. Knowing CL one can use Theorem 7 of [9] which states that a cycle is self dual if and only if $n^{\mathrm{CL}/2} = q - 1$. This computation is very cheap if one uses powers-of-two doubling.

3.2 Some Details on Algorithm MIPT

We now describe the overall features of Algorithm MIPT by further contrasting it to the three earlier algorithms [6,8,9].

Both Brenner [8] and later Cate and Twigg [9] improved Laflin and Brebner's algorithm [6] by combining the dual and self dual cases into a single processing case. Brenner went further when he applied some elementary Abelian group theory: the natural numbers $0 < i < mn$ partition into n_d Abelian groups where n_d is the number of divisors d of $\phi(q)$; ϕ is Euler's phi function. We have $q = \sum_{d|q} ord(G_d)$ and $ord(G_d) = \phi(q/d)$. We and he both recognized that this partition can sometimes greatly reduce the time spent in the search for cycle minima. However, the inner loop now becomes a double loop where its inner loop must run over *only* the numbers relatively prime to $\phi(q/d)$. This complication forces the overhead of the inner inner loop to increase. However, sometimes the cycle minima search is drastically reduced; in those cases the overall processing time of the new double inner loop is greatly reduced. All three algorithms [6,8,9] combine a variant of our bit vector algorithm with Gower's algorithm. By variant we mean they all use an integer sized array of size `IWORK` to just hold a bit. They all say that setting the length of `IWORK` equal to $(m+n)/2$ is a good choice. Now by using a bit vector we gain a factor of 32 over their use of integer array `MOVE`. However, locating and accessing / testing a bit in a 32 bit word has a higher processing cost than a simple integer compare against 0 / 1. Quite simply, the bit vector approach is more costly per unit invocation. However, since one gains a factor of 32 in the size of `IWORK` at no additional storage cost the number of times the costly inner loop will be entered will be lessened and overall, in some cases, there will be a big gain in the processing time of Algorithm `MIPT`.

4 Integer Overflow and the Fortran mod Function

Previously, we have seen that $P(k) = mk \bmod q$ where $0 < k < q$, $i = \lfloor k/n \rfloor$, $j = k - ni$ and $P(k) = i + mj$ is the location of A_{ij} assuming Fortran storage order and 0-origin subscripting of A_{ij}. A_{ij} moves to A^T_{ij} under this mapping. Now, one can safely use this ij direct definition of $P(k)$ in a Fortran program as integer overflow cannot occur. But, by using the Fortran mod function to compute $P(k)$ instead, one can increase the speed of computing $P(k)$ and hence increase the speed of the inner loop processing of Algorithm `MIPT`. But integer overflow can occur when using the Fortran mod formula for $P(k)$. And this Fortran mod formula then computes an *unwanted* result. More bluntly, the use of the Fortran mod formula produces a bug in Brenner's ACM Alg. 467! In ACM Algs. 380 and 513 the authors compute `m*k-q*(k/n)` as the value for $P(k)$. Now, integer overflow also does occur. However, despite the overflow occuring, the Fortran formula `m*k-q*(k/n)` computes $P(k)$ correctly. Both [6,9] fail to note this possibility of overflow although [9] states that the Fortran formula `m*k-q*(k/n)`computed faster than the CDC 6600 system modulo function. Further thought would have shown why this is so as these two formulas must compute different results in the case of integer overflow. In any case, we now prove in the Lemma below that `m*k-q*(k/n)` computes $P(k)$ correctly.

Lemma: Given are positive integers m and n. Let $q = mn - 1$ such that $q \leq 2^{31} - 1$. Let $0 < k < q$ with k relatively prime to q. Set $\bar{k} = mk \bmod q$ and $i = \lfloor k/n \rfloor$. Then $\bar{k} = mk - iq$. Let `KBAR=M*K-(K/N)*Q` be a Fortran statement where `I, K, M, N, Q` and `KBAR` are `INTEGER*4` Fortran variables. We shall prove `KBAR` $= \bar{k}$.

Proof: Put $j = k - ni$. Then $\bar{k} = i + jm = mq - iq$ as simple calculations show. Let $mk = q_1 \times 2^{32} + r_1$ and $iq = q_2 \times 2^{32} + r_2$ with $0 \leq r_l < 2^{31}$, $l = 1, 2$. Let $r_l = b_l \times 2^{31} + s_l$, $l = 1, 2$. Here $0 \leq b_l \leq 1$ and $0 \leq s_l \leq 2^{31}$. Again simple calulations show that $\bar{k} = s_1 - s_2$ when $b_1 = b_2$ and $\bar{k} = 2^{31} + s_1 - s_2$ when

$b_1 \neq b_2$. Also, $0 < \bar{k} < q$ is representable by INTEGER*4 Fortran variable KBAR. Suppose $b_1 = b_2 = 0$. Then M*K equals s_1, -I*Q equals $-s_2$ and the Fortran sum KBAR equals $s_1 - s_2 = \bar{k}$. Also, if $b_1 = b_2 = 1$, then M*K equals $-(2^{31} - s_1)$ and -I*Q equals $2^{31} - s_2$. Their algebraic sum equals $s_1 - s_2 = \bar{k}$ which also equals the Fortran sum KBAR. Note that when $b_1 = b_2$, KBAR is the sum of terms of mixed sign and overflow will not occur. Now suppose $b_1 = 0$ and $b_2 = 1$. Then M*K equals s_1 and -I*Q equals $2^{31} - s_2$. Both operands are positive and their Fortran sum KBAR = their actual sum $\bar{k}$. Since $\bar{k}$ is a 32 bit positive integer no overflow will occur. Finally, let $b_1 = 1$ and $b_2 = 0$. Then M*K equals $-(2^{31} - s_1)$ and -I*Q equals $-s_2$. After setting $-2^{31} = -2^{32} + 2^{31}$ we find the actual sum equals $-2^{32} + \bar{k}$. However, in this case, for 32 bit arithmetic, overflow always occurs and thus KBAR $= \bar{k}$. This completes the proof.

5 Performance Studies of MIPT

In the first experiment, run only on a Power 5, ACM Alg. 467 was compared to MIPT. 190 matrices of row and column sizes from 50 to 1000 in steps of 50 were generated and one experiment was conducted. We made two runs where IWORK = 0 and 100. When IWORK = 0 both codes do *not* use their arrays MOVE as their sizes are zero. When IWORK = 100, MIPT uses 3200 entries of k whereas ACM Alg. 467 uses 100 entries of k to reduce their respective inner loop searches. We ruled out the cases when $m = n$ as both codes are then the same. There are 19*20/2 = 190 cases in all. We label these cases from 1 to 190 starting at m=1000, n=950. Thus m=1000, n=950:50:-50 are cases 1 to 19, m= 950, n=900:50:-50 are cases 20 to 37 and so on down to m=100, n=50 which is case 190. Our performance results as functions of m and n are essentially random. Results in both Tables are separated into good and bad cases. The good cases mean that MIPT is faster than ACM Alg. 467 and the bad cases mean the opposite. The good and bad cases are ordered into buckets of percent. A good case in bucket 10 means that MIPT is between 10 to 11 % faster than Alg. 467. In general a good case in bucket j means that MIPT is between j % to (j+1) % faster than Alg. 467. A result in bucket j (bad or good) translates to a factor of 1 + j/100 to 1 + (j+1)/100 times faster. If j >= 100, then one of the algorithms is more than twice as fast as the other.

```
--------------------TABLE 1 ( IWORK = 0 )------------------------------
   5 BAD CASES  as  2 Triples of %j :  # of Problems :  Problem Numbers
 0:1: 142
 1:4: 18 81 139 147
-----------------------------------------------------------------------
  185 GOOD CASES as 5 Triples of %j :  # of Problems :  Problem Numbers
 0:12: 38 51 60 86 87 107 113 125 127 128 142 155
 1:36: 8 16 21 23 24 27 30 36 55 62 65 71 74 78 93 95 101 102 104 115 129
 130 135 137 146 156 160 167 170 171 172 173 177 180 183 184
 2:62: 5 9 11 13 14 15 20 26 28 29 31 32 33 34 41 45 46 49 50 54 58 61 63
 66 72 75 77 79 80 85 88 89 92 94 103 105 108 109 110 117 118 119 120 122
 126 131 132 134 136 138 143 144 148 152 162 163 168 169 176 178 185 188
 3:72: 1 2 3 4 6 7 10 12 17 19 22 25 37 39 40 43 44 47 48 52 53 56 57 59
 64 67 68 69 70 73 76 82 83 84 90 91 96 97 98 99 100 106 111 112 114 116
 123 124 133 140 141 145 149 150 151 153 154 157 158 159 161 164 165 174
 175 179 181 182 186 187 189 190
 4:3: 35 121 166
-------------------------------------------------------------------------
```

The results of Table 1 demonstrate that Algorithm MIPT is always marginally, about 3 %, faster than ACM Alg. 467.

```
 --------------------TABLE 2 ( IWORK = 100 )----------------------------
    76 BAD CASES as 12 Triples  of %j :  # of Problems :  Problem Numbers
  0:20:21 29 32 33 38 42 46 49 60 74 75 76 80 87 113 125 128 139 147 164
  1:20:9 16 24 27 31 45 61 65 78 81 88 105 118 121 136 137 143 146 150 156
  2:9:62 79 84 93 95 107 117 127 166 | 3 4:102 115 119 129 | 4 2:34 160
  5:5:108 134 148 167 177 | 6:2:69 171 | 7:2:98 152 | 8:4:70 111 172 176
  11:2:112 174 | 15:2:123 135 | 16:4:37 54 161 178
    13 more BAD CASES as 13 Pairs of     ( %j  Problem Number )
  (10 173), (12 185), (13 145), (14 179), (19 162), (21 182), (25 99)
  (29 180), (30 188), (38 184), (40 187), (54 189), (90 190)
 ----------------------------------------------------------------------
   60 GOOD CASES as  9 Triples of %j :  # of Problems :  Problem Numbers
  0:27:5 7 8 28 30 41 47 51 55 58 64 71 77 83 86 94 96 100 101 104 120
         130 131 132 142 149 155
  1:14:4 17 23 26 36 44 57 67 110 114 122 140 141 170
  2:4:20 25 56 90 |  3:4:52 72 157 163 |  4:3:63 68 109 | 22:2:22 175
  93:2:1 169 |  102:2:2 6 |  114:2:15 138
    41 more GOOD CASES as 41 Pairs of    ( %j  Problem Number )
  ( 5 18), ( 7 116), (11 35), (18 91), (27 126), (34 10), (39 11), (40 12),
  (46 92), (47 133), (48 59), (53 159), (56 124), (58 50), (62 14),
  (66 66), (70 48), (71 158), (72 73), (76 43), (77 151), (82 183),
  (84 186), (87 82), (89 144), (92 39), (94 89), (95 19), (97 85), (104 13),
  (108 181), (110 53), (111 153), (112 3), (113 154), (116 168), (132 40),
  (139 97), (144 103), (156 165), (186 106)
 ----------------------------------------------------------------------
```

The results of Table 2 demonstrate that Algorithm MIPT benefits from using a bit-vector approach as opposed to using an integer array whose elements represent just single bits. The unit overhead cost of bit processing is greater than a straight integer-compare of zero or one. The 89 bad-case results are due to the effect of the higher unit cost being greater than the savings afforded by inner loop testing. For the 101 good cases, there are more savings afforded by inner loop testing in MIPT and these savings more than offset the losses of higher unit cost bit processing. Note that there are 55 bad cases and 56 good cases whose percentage is less than 5. So, in these 101 cases performance is about equal. In the remaining 89 cases of which 45 are good and 34 are bad there are 16 good cases and no bad cases that are more than twice as fast.

In the second experiment, done on an IBM Power 5 and a PowerPC 604, Algorithm `MIPT` was compared to ESSL subroutine `DGETMO` which is an out-of-place transpose routine. The major cost of Alg. `MIPT` over `DGETMO` has to do with accessing a whole line of A with only one element being used. Now the PowerPC 604 has line size 4 whereas the Power 5 has line size 16. And qualitatively, the two runs show this as the ratios are better on the Power 5.

```
-----------TABLE 3 on the Power 5 ( IWORK = 100 )----------------------
  190 CASES   13 Triples  of 100j% :  # of Problems :  Problem Numbers
4:22: 35 36 67 68 69 95 96 97 120 121 122 139 140 141 142 143 155 156
      157 158 170 171
5:28: 20 30 32 33 34 51 52 64 65 66 91 93 94 98 109 110 116 117 118
      119 136 137 138 149 150 159 160 172
6:45: 1 17 21 22 23 24 25 26 27 28 29 31 50 55 56 57 58 59 60 61 62 63
      83 86 87 88 89 90 92 107 108 111 113 114 115 123 132 146 147 148
      151 152 163 176 177
7:31: 16 38 39 41 42 43 44 47 48 49 53 70 81 82 84 100 101 102 104 105
      106 129 130 131 134 153 164 166 167 173 181
8:19: 2 3 4 8 19 37 40 45 46 54 80 103 127 128 133 161 165 174 178
9:26: 5 6 7 9 10 11 12 13 14 71 72 73 74 75 76 77 78 79 112 125 126 135
      145 162 179 185
10:3: 99 144 182 | 11:5: 85 168 180 184 188 | 12:1: 124
13:4: 15 154 186 189 | 14:1: 175 | 15:1: 169 |16:3: 183 187 190 | 46:1: 18
------------------------------------------------------------------------
-----------TABLE 4 on the PowerPC 604 ( IWORK = 100 )-------------------
  190 CASES   4 Triples  of 100j% :  # of Problems :  Problem Numbers
0:39: 19 37 53 54 68 70 84 98 112 113 123 134 135 142 143 145 152 153
      161 162 165 166 167 168 171 172 173 174 176 177 178 179 181 182
      185 186 188 189 190
1:128: 8 10 11 12 13 14 15 16 17 27 28 29 30 31 32 33 34 35 36 41 43 44
       45 46 47 48 49 50 51 52 55 57 58 59 60 61 62 63 64 65 66 67 69 71
       72 74 75 76 77 78 79 80 81 82 83 85 86 87 88 89 90 91 92 93 94 95
       96 97 99 100 101 102 103 104 105 106 107 108 109 110 111 114 115
       116 117 118 119 120 121 122 124 125 126 127 128 129 130 131 132
       133 136 137 138 139 140 141 144 146 147 148 149 150 151 154 155
       156 157 158 159 160 163 164 169 170 175 183 184 187
2:22: 1 2 3 4 5 6 7 9 18 20 21 22 23 24 25 26 38 39 40 42 56 73 | 4:1: 180
 -----------------------------------------------------------------------
```

In Tables 3 and 4 we have made each bucket 100 times larger. So, a result in bucket `j` translates to a factor of `1 + j` to `1 + j+1` times faster. If `j >= 1`, then `DGETMO` is more than twice as fast as Alg. `MIPT`. One can see, on the Power 5 in Table 3, that `DGETMO` is more than five times faster than `MIPT` for all 190 cases. It is more than ten times faster for 45 cases and for one case it is 48 times faster. On the PowerPC604 `DGETMO` is again always faster. This time there are 39 cases where it is not twice as fast, 128 cases where it is between 2 and 3 times faster, 22 cases where it is between 3 to 4 times faster and one case where it is 5.9 times faster. So, Tables 3 and 4 corroborate the remarks made in paragraph one of Section 3.

In Section 1 we described our bit vector algorithm that was based on our Algorithm IPT (see Section 2). On the Power 5 we compared its performance on our set of 190 matrices against our Algorithm MIPT with IWORK=100. Surprisingly, MIPT was faster in 189 cases and in 15 cases it was more than twice as fast.

6 Discussion of Prior In-Place Transpostion Algorithms

We thought our algorithms were new; the second printing of Knuth, Vol. 1, Fundamental Algorithms gave us this impression, see [4]. In [4], we became aware of problem 12 of section 1.3.3 which posed the subject of our paper as an exercise. The solution of problem 12 gave outlines of bit-vector algorithms but gave no specific details. Additionally, Berman's algorithm [1], the algorithm of Pall and Seiden and ACM Algorithm 302 by Boothroyd [3] were cited. However, one of our key discoveries was *not* mentioned: the use of the bit vector could be removed; however, at the cost of many additional computer operations. Later, in [13], Knuth additionally cites Brenner and ACM Algorithm 467, [8], Windley, [2], himself, [7], Cate and Twigg, [9] and Fich, Munro, Poblete, [12].

Windley's [2] paper gave three solutions to the in-place transposition problem which was an exercise to students taking the Cambridge University Diploma in Numerical Analysis and Automatic Computing. Berman's algorithm was noted but not considered. The first solution by M. Fieldhouse was an $O(m^2n^2)$ algorithm. J. C. Gower produced our basic algorithm IPT by *first* discovering one of our key discoveries. Gower also used the count of the elements transposed to speed up his algorithm. The third algorithm, due to Windley, was a variant of Gower's Algorithm which for each k, $0 \leq k < mn$ placed one element of the transpose in its correct position. Macleod, in [5], discusses in-situ permutation that governed the matrix transposition. He presents a modification of Gower's Algorithm which he believed to be "the most efficient yet devised". He notes that its performance varied from reasonable to poor depending on the permutation governing matrix transposition.

ACM Alg. 380, in Laflin and Brebner [6], also uses Gower's and our later key discovery. It also uses another discovery of ours: a duality principal. In [6], duality was called symmetric. Laflin and Brebner [6] were first to describe the duality result. Finally, ACM Alg. 380 uses an integer array MOVE of length IWORK. The purpose of array MOVE was to reduce the cost of the additional computer operations imposed by using the key discovery. ACM Alg. 380 gives empirical evidence that IWORK should be set to $(m + n)/2$. We note the use of IWORK produces a hybrid algorithm that combines the bit vector approach with the use of the key idea. Brenner's ACM Alg. 467 is an improvement of ACM Alg. 380. Cate and Twigg [9] discuss some theoretical results related to in-situ transposition and use these to accelerate ACM Algs. 302 and 380. ACM Alg. 302, see [3] was an improvement of Windley's algorithm. Finally, in [10] Leathers presents experimental evidence that ACM Alg. 513 is inferior to ACM Alg. 467 and laments the publication of ACM Alg. 513. In [12], Fich, Munro and Poblete give new computational complexity results on permuting in-place. When both

P and P^{-1} are known they give in their Figure 5 on page 269 a modified version of Gower's Algorithm. We now briefly indicate why ACM Alg. 467 is faster than ACM Algs. 380 and 513. All three algorithms use array `MOVE` to reduce the use of their costly inner loops. If `IWORK` $= mn$ then one is using our bit vector algorithm and one completely avoids using this inner loop. Unfortunately, these three algorithms are then no longer in-situ. ACM Algs. 467 and 513 use the duality principle in an optimal way whereas ACM Alg. 380 does not. Finally, ACM Alg. 467 additionally recognizes when q has divisors d. When q has several divisors d the costly inner loop search can be greatly reduced. We also discovered these facts. In [9], Cate and Twigg note that the Fortran statement for `KBAR` is faster than using the system function for $P(k)$. For this reason, ACM Alg. 380 and 513 avoided a latent bug whereas ACM Alg. 467 did not. We find it a curious fact, which we proved, that the formula for `KBAR` works in the case of integer overflow whereas the Fortran definition of mod forces extra operations that, when integer overflow occurs, compute an unwanted result.

References

1. Berman, M.F.: A Method for Transposing a Matrix. J. Assoc. Comp. Mach. 5, 383–384 (1958)
2. Windley, P.F.: Tranpsoing matrices in a digital computer. Comput. J. 2, 47–48 (1959)
3. Boothroyd, J.: Alg. 302: Transpose vector stored array. Comm. ACM 10(5), 292–293 (1967)
4. Knuth, D.: In: The Art of Computer Programming (2nd printing), 1st edn., vol. 1, Addison-Wesley, Reading (1969) Problem 12, page 180 and Solution to Problem 12. page 517
5. Macleod, I.D.G.: An Algorithm For In-Situ Permutation. The Austrialian Computer Journal 2(1), 16–19 (1970)
6. Laflin, S., Brebner, M.A.: Alg. 380: In-situ transposition of a rectangular matrix. Comm. ACM 13, 324–326 (1970)
7. Knuth, D.: Matematical Analysis of Algorithms, Information Processing 71, Invited Papers-Foundations. North-Holland Publishing Company (1972)
8. Brenner, N.: Matrix Transposition in Place. Comm. ACM 16(11), 692–694 (1973)
9. Cate, E.G., Twigg, D.W.: Algorithm 513: Analysis of In-Situ Transposition. ACM TOMS 3(1), 104–110 (1977)
10. Leathers, B.L.: Remark on Algorithm 513: Analysis of In-Situ Transposition. ACM TOMS 5(4), 520 (1979)
11. Agarwal, R.C., Gustavson, F.G., Zubair, M.: Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. IBM Journal of Research and Development 38(5), 563–576 (1994)
12. Fich, F.E., Munro, J.I., Poblete, P.V.: Permuting In Place. SIAM Journal of Computing 24(2), 266–278 (1995)
13. Knuth, D.: In: The Art of Computer Programming (4th printing), 3rd edn., vol. 1. Addison-Wesley, Reading Mass (1997) Problem 12, page 182 and Solution to Problem 12. page 523

Rectangular Full Packed Format for LAPACK Algorithms Timings on Several Computers

Fred G. Gustavson[1] and Jerzy Waśniewski[2]

[1] IBM T.J. Watson Research Center, Yorktown Heights NY 10598, USA
fg2@us.ibm.com

[2] Informatics & Mathematical Modeling, Technical University of Denmark, DK-2800 Lyngby, Denmark
jw@imm.dtu.dk

Abstract. We describe a new data format for storing triangular and symmetric matrices called RFP (Rectangular Full Packed). The standard two dimensional arrays of Fortran and C (also known as full format) that are used to store triangular and symmetric matrices waste nearly half the storage space but provide high performance via the use of level 3 BLAS. Standard packed format arrays fully utilize storage (array space) but provide low performance as there are no level 3 packed BLAS. We combine the good features of packed and full storage using RFP format to obtain high performance using L3 (level 3) BLAS as RFP is full format. Also, RFP format requires exactly the same minimal storage as packed format. Each full and/or packed symmetric/triangular routine becomes a single new RFP routine. We present LAPACK routines for Cholesky factorization, inverse and solution computation in RFP format to illustrate this new work and to describe its performance on the IBM, Itanium, NEC, and SUN platforms. Performance of RFP versus LAPACK full routines for both serial and SMP parallel processing is about the same while using half the storage. Performance is roughly one to a factor of 33 for serial and one to a factor of 100 for SMP parallel times faster than LAPACK packed routines. Existing LAPACK routines and vendor LAPACK routines were used in the serial and the SMP parallel study, respectively. In both studies vendor L3 BLAS were used.

1 Introduction

Recently many new data formats for matrices have been introduced for improving the performance Dense Linear Algebra (DLA) algorithms. Two ACM TOMS papers [2,1] and the survey article [5] give an excellent overview. Since then at least two new ones have emerged, [6] and the subject matter of this paper, RFP format.

2 Description of Rectangular Full Packed Format

We describe RFP (Rectangular Full Packed) format. It represents a standard packed array as a full 2D array. By using RFP format the performance of LAPACK's [3] packed format routines becomes equal to or better than their full array counterparts. RFP format is a variant of Hybrid Full Packed (HFP) format [4]. RFP format is a rearrangement of a Standard full rectangular Array (SA) holding a symmetric / triangular matrix A into a compact full storage Rectangular Array (AR) that uses the minimal storage $NT = n(n+1)/2$. Note also that the transpose of the matrix in array AR also represents A. Therefore, level 3 BLAS can be used on AR or its transpose. In fact, with the equivalent LAPACK algorithm, using array AR or its transpose instead of array SA, gives slightly better performance. Therefore, this offers the possibility to replace all packed or full LAPACK routines with equivalent LAPACK routines that work on array AR or its transpose.

3 Cholesky Factorization Using Rectangular Full Packed Format

RFP format is a standard full array of size $NT = n(n+1)/2$ that holds a symmetric / triangular matrix A of order n. It is closely related to HFP format, see [4], which represents A as the concatenation of two standard full arrays whose total size is also NT. A basic simple idea leads to both formats. Let A be an order n symmetric matrix. Break A into a block 2×2 form

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} \tag{1}$$

where A_{11} and A_{22} are symmetric. Clearly, we need only store the lower triangles of A_{11} and A_{22} as well as the full matrix A_{21}. When $n = 2k$ is even, the lower triangle of A_{11} and the upper triangle of A_{22}^T can be concatenated together along their main diagonals into an $(k+1) \times k$ dense matrix. This last operation is the crux of the basic simple idea. The off-diagonal block A_{21} is $k \times k$, and so it can be appended below the $(k+1) \times k$ dense matrix. Thus, the lower triangle of A can be stored as a single $(n+1) \times k$ dense matrix AR. In effect, each block matrix A_{11}, A_{21} and A_{22} is now stored in 'full format'. This means all entries of AR can be accessed with constant row and column strides. So, the full power of LAPACK's block level 3 codes are now available for RFP format which uses the minimum amount of storage. Finally, AR^T which is $k \times (n+1)$ has the same desirable properties. In the right part of Figures 1 and 2 with $n = 7$ and $n = 6$ we have introduced colors and horizontal lines to try to visually delineate triangles T_1, T_2 representing lower, upper triangles of symmetric matrices A_{11}, A_{22}^T respectively and square or near square S_1 representing matrix A_{21}. After each $a_{i,j}$ we have added its position location in the arrays A and AR.

We now describe a 2 by 2 block algorithm (BA) that naturally suggests itself for use on RFP format. A has a block 2×2 form Cholesky factorization

LAPACK full data format
n = 7
memory needed: n × n = 49

Rectangular full packed
n = 7, memory needed:
n × (n+1)/2 = 28

$$\left(\begin{array}{cccc|ccc}
a_{1,1_1} & \diamond & \diamond & \diamond & \diamond & \diamond & \diamond \\
a_{2,1_2} & a_{2,2_9} & \diamond & \diamond & \diamond & \diamond & \diamond \\
a_{3,1_3} & a_{3,2_{10}} & a_{3,3_{17}} & \diamond & \diamond & \diamond & \diamond \\
a_{4,1_4} & a_{4,2_{11}} & a_{4,3_{18}} & a_{4,4_{25}} & \diamond & \diamond & \diamond \\
a_{5,1_5} & a_{5,2_{12}} & a_{5,3_{19}} & a_{5,4_{26}} & a_{5,5_{33}} & \diamond & \diamond \\
a_{6,1_6} & a_{6,2_{13}} & a_{6,3_{20}} & a_{6,4_{27}} & a_{6,5_{34}} & a_{6,6_{41}} & \diamond \\
a_{7,1_7} & a_{7,2_{14}} & a_{7,3_{21}} & a_{7,4_{28}} & a_{7,5_{35}} & a_{7,6_{42}} & a_{7,7_{49}}
\end{array}\right), \quad
\left(\begin{array}{cccc}
a_{1,1_1} & \underline{a_{5,5_8}} & a_{6,5_{15}} & a_{7,5_{22}} \\
a_{2,1_2} & a_{2,2_9} & \underline{a_{6,6_{16}}} & a_{7,6_{23}} \\
a_{3,1_3} & a_{3,2_{10}} & a_{3,3_{17}} & \underline{a_{7,7_{24}}} \\
a_{4,1_4} & a_{4,2_{11}} & a_{4,3_{18}} & a_{4,4_{25}} \\ \hline
a_{5,1_5} & a_{5,2_{12}} & a_{5,3_{19}} & a_{5,4_{26}} \\
a_{6,1_6} & a_{6,2_{13}} & a_{6,3_{20}} & a_{6,4_{27}} \\
a_{7,1_7} & a_{7,2_{14}} & a_{7,3_{21}} & a_{7,4_{28}}
\end{array}\right)$$

A matrix A

A matrix AR

Fig. 1. Rectangular Full Packed format if n is odd

$$LL^T = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \tag{2}$$

where L_{11} and L_{22} are lower triangular. Equation (2) is the basis of a 2 by 2 BA on RFP. We now describe this by using existing LAPACK routines and level 3 BLAS (see figure 3). The BA with block sizes k and $k+1$ where $k = \lceil n/2 \rceil$ or $k = n/2$ is: see equations (1), (2) and Figures 1, 2, 3.

LAPACK full data format
n = 6
memory needed: n × n = 36

rectangular full packed
n = 6, memory needed:
(n+1) × n/2 = 21

$$\left(\begin{array}{ccc|ccc}
a_{1,1_1} & \diamond & \diamond & \diamond & \diamond & \diamond \\
a_{2,1_2} & a_{2,2_8} & \diamond & \diamond & \diamond & \diamond \\
a_{3,1_3} & a_{3,2_9} & a_{3,3_{15}} & \diamond & \diamond & \diamond \\
a_{4,1_4} & a_{4,2_{10}} & a_{4,3_{16}} & a_{4,4_{22}} & \diamond & \diamond \\
a_{5,1_5} & a_{5,2_{11}} & a_{5,3_{17}} & a_{5,4_{23}} & a_{5,5_{29}} & \diamond \\
a_{6,1_6} & a_{6,2_{12}} & a_{6,3_{18}} & a_{6,4_{24}} & a_{6,5_{30}} & a_{6,6_{36}}
\end{array}\right), \quad
\left(\begin{array}{ccc}
\underline{a_{4,4_1}} & a_{5,4_8} & a_{6,4_{15}} \\
a_{1,1_2} & \underline{a_{5,5_9}} & a_{6,5_{16}} \\
a_{2,1_3} & a_{2,2_{10}} & \underline{a_{6,6_{17}}} \\
a_{3,1_4} & a_{3,2_{11}} & a_{3,3_{18}} \\ \hline
a_{4,1_5} & a_{4,2_{12}} & a_{4,3_{19}} \\
a_{5,1_6} & a_{5,2_{13}} & a_{5,3_{20}} \\
a_{6,1_7} & a_{6,2_{14}} & a_{6,3_{21}}
\end{array}\right)$$

A matrix A

A matrix AR

Fig. 2. Rectangular Full Packed format if n is even

This covers RFP format when uplo = 'L', and n is odd (figure 1) and even (figure 2). For uplo = 'U', for n even and odd similar layouts exist. Also, all of these layouts have associated layouts for AR^T.

We now consider performance aspects of using RFP format in the context of using LAPACK routines on triangular matrices stored in RFP format. The above BA should perform about the same as the corresponding full format LAPACK routine. This is because both the BA code and the corresponding LAPACK code are nearly the same and both data formats are full format. Therefore, the BA code should outperform the corresponding LAPACK packed code by about the

```
1. factor $L_{11}L_{11}^T = A_{11}$;
   call POTRF('L',k,AR(1,1),n,info)
2. solve $L_{21}L_{11}^T = A_{21}$;
   call TRSM('R','L','T','N',k-1,k, &
   one,AR(1,1),n,AR(k+1,1),n)
3. update $A_{22}^T := A_{22}^T - L_{21}L_{21}^T$;
   call SYRK('U','N',k-1,k,-one, &
   AR(k+1,1),n,one,AR(1,2),n)
4. factor $U_{22}^T U_{22} = A_{22}^T$;
   call POTRF('U',k-1,AR(1,2),n,info)
```

For n odd, and $k = \lceil n/2 \rceil$

```
1. factor $L_{11}L_{11}^T = A_{11}$;
   call POTRF('L',k,AR(2,1),n+1,info)
2. solve $L_{21}L_{11}^T = A_{21}$;
   call TRSM('R','L','T','N',k,k,one, &
   AR(2,1),n+1,AR(k+2,1),n+1)
3. update $A_{22}^T := A_{22}^T - L_{21}L_{21}^T$;
   call SYRK('U','N',k,k,-one, &
   AR(k+2,1),n+1,one,AR(1,1),n+1)
4. factor $U_{22}^T U_{22} = A_{22}^T$;
   call POTRF('U',k,AR(1,1),n+1,info)
```

For n even, and $k = n/2$

Fig. 3. The Cholesky Factorization Algorithm using RFP format

same margin as does the corresponding LAPACK full code. In [4] performance results for HFP format on the IBM Power 4 Processor is given. Those results are similar to what we obtained for RFP format. Here the gain of full RFP code over packed LAPACK code is about a factor of one to 36 for serial processing and about a factor of one to 97 for SMP parallel processing.

4 A Performance Study Using RFP Format

There are 11 tables giving performance results of LAPACK and RFP routines. The LAPACK routines POTRF, PPTRF, POTRI, PPTRI, POTRS and PPTRS are compared with the RFPTRF, RFPTRI and RFPTRS for Cholesky factorization, inverse and solution respectively. In all cases real long precision arithmetic

Table 1. Performance in Mflops of Cholesky Factorization on SUN UltraSPARC-III computer

n	RFP				LAPACK			
	NO TRANS		TRANS		POTRF		PPTRF	
	U	L	U	L	U	L	U	L
50	456	520	516	482	460	464	291	294
100	753	813	829	768	612	827	399	369
200	946	979	997	955	933	1150	455	370
400	1208	1231	1158	1183	1081	1244	483	339
500	1173	1227	1138	1186	1121	1340	511	343
800	1316	1318	1189	1269	1256	1310	522	324
1000	1275	1318	1281	1303	1313	1406	530	288
1600	1350	1387	1358	1312	1405	1234	502	223
2000	1264	1367	1403	1323	1360	1491	394	163
4000	1287	1450	1537	1263	1392	1565	300	153

Table 2. Performance in Mflops of Cholesky Inversion on SUN UltraSPARC-III computer

n	RFP				LAPACK			
	NO TRANS		TRANS		POTRI		PPTRI	
	U	L	U	L	U	L	U	L
50	379	379	380	381	330	328	318	338
100	698	696	699	700	698	707	412	446
200	1012	989	1008	997	1052	1030	467	558
400	1290	1223	1229	1263	1229	1212	452	606
500	1290	1276	1238	1330	1285	1276	448	595
800	1446	1445	1330	1356	1343	1325	408	566
1000	1428	1337	1442	1436	1378	1318	404	531
1600	1418	1372	1369	1396	1142	1317	262	450
2000	1387	1333	1370	1536	1400	1366	242	394
4000	1460	1408	1389	1453	1421	1395	201	288

Table 3. Performance in Mflops of Cholesky Solution on SUN UltraSPARC-III computer

r	n	RFP				LAPACK			
h		NO TRANS		TRANS		POTRS		PPTRS	
s		U	L	U	L	U	L	U	L
100	50	1132	1123	1153	1135	1103	1069	353	353
100	100	1193	1163	1237	1211	1262	1262	478	478
100	200	1477	1478	1500	1490	1280	1235	557	554
100	400	1494	1505	1514	1534	1150	1149	582	582
100	500	1466	1443	1436	1445	1217	1229	560	569
100	800	1503	1505	1535	1469	1151	1096	528	526
100	1000	1553	1524	1499	1576	1089	1125	513	513
160	1600	1595	1564	1603	1577	1155	1121	421	403
200	2000	1600	1636	1610	1615	1105	1087	347	338
400	4000	1666	1668	1696	1665	1080	1084	292	290

(double precision) is used. Results were obtained on several different computers using everywhere the vendor level 3 and level 2 BLAS.

Due to space limitations, we cannot present all of our timing results. We noticed a few anomalies in the performance runs for POTRS on SUN in Table 3, the PP runs on NEC (Tables 7, 8 and 9) and the PP and PO runs on SUN SMP parallel, Table 11. We have re-run these cases and have obtained the same results. At this time we do *not* have a rational explanation for these anomalies. Finally, our timings do *not* include the cost of sorting any LAPACK data formats to RFP data formats and vice versa.

The tables from 1 to 9 show the performance comparison in Mflops of factorization, inversion and solution on SUN UltraSPARC-III (clock rate: 1200 MHz; L1 cache: 64 kB 4-way data, 32 kB 4-way instruction, and 2 kB Write, 2 kB Prefetch; L2 cache: 8 MB; TLB: 1040 entries), ia64 Itanium (CPU: Intel

Table 4. Performance in Mflops of Cholesky Factorization on ia64 Itanium computer

n	RFP				LAPACK			
	NO TRANS		TRANS		POTRF		PPTRF	
	U	L	U	L	U	L	U	L
50	781	771	784	771	1107	739	495	533
100	1843	1788	1848	1812	1874	1725	879	825
200	3178	2869	2963	3064	2967	2871	1323	1100
400	3931	3709	3756	3823	3870	3740	1121	1236
500	4008	3808	3883	3914	4043	3911	1032	1257
800	4198	4097	4145	4126	3900	4009	612	1127
1000	4115	4038	4015	3649	3769	3983	305	697
1600	3851	3652	3967	3971	3640	3987	147	437
2000	3899	3716	3660	3660	3865	3835	108	358
4000	3966	3791	3927	4011	3869	4052	119	398

Table 5. Performance in Mflops of Cholesky Inversion on ia64 Itanium computer

n	RFP				LAPACK			
	NO TRANS		TRANS		POTRI		PPTRI	
	u	l	u	l	u	l	u	l
50	633	659	648	640	777	870	508	460
100	1252	1323	1300	1272	1573	1760	815	810
200	2305	2442	2431	2314	2357	2639	1118	1211
400	3084	3199	3188	3094	3152	3445	1234	1363
500	3204	3316	3329	3218	3400	3611	1239	1382
800	3617	3741	3720	3640	3468	3786	1182	1268
1000	3611	3716	3637	3590	3456	3790	767	946
1600	3721	3802	3795	3714	3589	3713	500	609
2000	3784	3812	3745	3704	3636	3798	473	596
4000	3822	3762	3956	3851	3760	3750	467	614

Table 6. Performance in Mflops of Cholesky Solution on ia64 Itanium computer

r h s	n	RFP				LAPACK			
		NO TRANS		TRANS		POTRS		PPTRS	
		u	l	u	l	u	l	u	l
100	50	2409	2412	2414	2422	3044	3018	725	714
100	100	3305	3301	3303	3303	3889	3855	1126	1109
100	200	4149	4154	4127	4146	4143	4127	1526	1512
100	400	4398	4403	4416	4444	4469	4451	1097	1088
100	500	4313	4155	4374	4394	4203	4093	1054	1045
100	800	3979	3919	4040	4051	3969	4011	692	720
100	1000	3716	3608	3498	3477	3630	3645	376	372
160	1600	3892	3874	4020	3994	4001	4011	188	182
200	2000	4052	4073	4040	4020	4231	4203	119	119
400	4000	4245	4225	4275	4287	4330	4320	115	144

Table 7. Performance in Mflops of Cholesky Factorization on SX-6 NEC computer with Vector Option

n	RFP				LAPACK			
	NO TRANS		TRANS		POTRF		PPTRF	
	u	l	u	l	u	l	u	l
50	206	200	225	225	365	353	57	238
100	721	728	789	788	1055	989	120	591
200	2028	2025	2005	2015	1380	1639	246	1250
400	3868	3915	3078	3073	1763	3311	479	1975
500	4483	4470	4636	4636	4103	4241	585	2149
800	5154	5168	4331	4261	3253	4469	870	2399
1000	5666	5654	5725	5703	5144	5689	1035	2474
1600	6224	6145	5644	5272	5375	5895	1441	2572
2000	6762	6788	6642	6610	6088	6732	1654	2598
4000	7321	7325	7236	7125	6994	7311	2339	2641

Table 8. Performance in Mflops of Cholesky Inversion on SX-6 NEC computer with Vector Option

n	RFP				LAPACK			
	NO TRANS		TRANS		POTRI		PPTRI	
	u	l	u	l	u	l	u	l
50	152	152	150	152	148	145	91	61
100	430	432	428	432	313	310	194	126
200	950	956	940	941	636	627	404	249
400	1850	1852	1804	1806	1734	1624	722	470
500	2227	2228	2174	2181	2180	2029	856	572
800	3775	3775	3668	3686	3405	3052	1186	842
1000	4346	4346	4254	4263	4273	3638	1342	985
1600	5313	5294	5137	5308	5438	4511	1690	1361
2000	6006	6006	5930	5931	5997	4832	1854	1536
4000	6953	6953	6836	6888	7041	4814	1921	2122

Itanium2: 1.3 GHz, cache: 3 MB on-chip L3 cache), and NEC SX-6 computer (8 CPU's, per CPU peak : 8 Gflops, per node peak : 64 Gflops, vector register length: 256). The tables 10 and 11 show the SMP parallelism of these subroutines on the IBM Power4 (clock rate: 1300 MHz; two CPUs per chip; L1 cache: 128 KB (64 KB per CPU) instruction, 64 KB 2-way (32 KB per CPU) data; L2 cache: 1.5 MB 8-way shared between the two CPUs; L3 cache: 32 MB 8-way shared (off-chip); TLB: 1024 entries) and SUN UltraSPARC-IV (the same hardware parameters as SUN UltraSPARC-III except the clock rate: 1350 MHz) computers respectively. They compare SMP times of RFPTRF, POTRF and PPTRF. The tables 10 and 11 also show the times of the four operations (POTRF, TRSM, SYRK and again POTRF) inside the new algorithm RFPTRF.

Table 9. Performance in Mflops of Cholesky Solution on SX-6 NEC computer with Vector Option

r h s	n	RFP				LAPACK			
		NO TRANS		TRANS		POTRS		PPTRS	
		U	L	U	L	U	L	U	L
100	50	873	870	889	886	1933	1941	88	88
100	100	2173	2171	2200	2189	3216	3236	181	179
100	200	4236	4230	4253	4245	4165	4166	352	347
100	400	5431	5431	5410	5408	5302	5303	648	644
100	500	5563	5562	5568	5567	5629	5632	783	779
100	800	6407	6407	6240	6240	5569	5593	1132	1128
100	1000	6578	6578	6559	6558	6554	6566	1325	1320
160	1600	6781	6805	6430	6430	6799	6809	1732	1727
200	2000	7568	7569	7519	7519	7406	7407	1920	1914
400	4000	7858	7858	7761	7761	7626	7627	2414	2410

Table 10. Performance Times and MFLOPS of Cholesky Factorization on an IBM Power 4 computer using SMP parallelism on 1, 5, 10 and 15 processors. Here vendor codes for Level 2 and 3 BLAS and POTRF are used, ESSL library version 3.3. PPTRF is LAPACK code. UPLO = 'L'.

n	n pr oc	Mflp	Times						
		RFPTRF		in RFPTRF				LAPACK	
				POTRF	TRSM	SYRK	POTRF	POTRF	PPTRF
1000	1	2695	0.12	0.02	0.05	0.04	0.02	0.12	0.94
	5	7570	0.04	0.01	0.02	0.01	0.01	0.03	0.32
	10	10699	0.03	0.01	0.01	0.01	0.00	0.02	0.16
	15	18354	0.02	0.00	0.01	0.00	0.00	0.01	0.11
2000	1	2618	1.02	0.13	0.38	0.38	0.13	0.97	8.74
	5	10127	0.26	0.04	0.10	0.09	0.04	0.24	3.42
	10	17579	0.15	0.02	0.06	0.05	0.03	0.12	1.65
	15	23798	0.11	0.02	0.04	0.04	0.01	0.13	1.11
3000	1	2577	3.49	0.45	1.33	1.28	0.44	3.40	30.42
	5	11369	0.79	0.11	0.28	0.30	0.11	0.71	11.76
	10	19706	0.46	0.06	0.19	0.16	0.05	0.38	6.16
	15	29280	0.31	0.05	0.12	0.10	0.04	0.26	4.28
4000	1	2664	8.01	1.01	2.90	3.09	1.01	7.55	75.72
	5	11221	1.90	0.26	0.68	0.72	0.24	1.65	25.73
	10	21275	1.00	0.13	0.39	0.36	0.12	0.86	13.95
	15	31024	0.69	0.09	0.28	0.24	0.08	0.59	10.46
5000	1	2551	16.34	2.04	6.16	6.10	2.04	15.79	154.74
	5	11372	3.66	0.45	1.37	1.44	0.40	3.27	47.76
	10	22326	1.87	0.25	0.78	0.62	0.22	1.73	28.13
	15	32265	1.29	0.17	0.53	0.45	0.14	1.16	20.95

Table 11. Performance in Times and Mflops of Cholesky Factorization on SUN UltraSPARC-IV computer with a different number of Processors, testing the SMP Parallelism. PPTRF does not show any SMP parallelism. UPLO = 'L'.

n	n proc	Mflp	Times	Times	Times	Times	Times	Times	Times
		RFPTRF		in RFPTRF				LAPACK	
				POTRF	TRSM	SYRK	POTRF	POTRF	PPTRF
1000	1	1587	0.21	0.03	0.09	0.07	0.03	0.19	1.06
	5	4762	0.07	0.02	0.02	0.02	0.02	0.07	1.13
	10	5557	0.06	0.01	0.01	0.02	0.02	0.06	1.12
	15	5557	0.06	0.02	0.01	0.01	0.02	0.06	1.11
2000	1	1668	1.58	0.22	0.63	0.52	0.22	1.45	11.20
	5	6667	0.40	0.07	0.13	0.13	0.07	0.38	11.95
	10	8602	0.31	0.06	0.07	0.11	0.07	0.25	11.24
	15	9524	0.28	0.06	0.06	0.08	0.08	0.23	11.66
3000	1	1819	4.95	0.62	1.98	1.72	0.63	4.86	45.48
	5	6872	1.31	0.20	0.42	0.48	0.20	1.38	55.77
	10	12162	0.74	0.14	0.22	0.21	0.16	0.76	46.99
	15	12676	0.71	0.14	0.16	0.30	0.16	0.61	45.71
4000	1	1823	11.70	1.52	4.62	4.01	1.55	11.86	112.52
	5	7960	2.68	0.40	0.94	0.92	0.42	2.74	112.77
	10	14035	1.52	0.26	0.47	0.49	0.30	1.61	112.53
	15	17067	1.25	0.24	0.37	0.35	0.29	1.29	111.67
5000	1	1843	22.61	2.92	8.76	8.00	2.93	23.60	218.94
	5	8139	5.12	0.77	1.81	1.80	0.74	5.45	221.58
	10	14318	2.91	0.50	0.97	0.93	0.51	3.11	214.54
	15	17960	2.32	0.45	0.72	0.68	0.47	2.40	225.08

5 Summary and Conclusions

This paper describes RFP format as a standard minimal full format for representing both symmetric and triangular matrices. Hence, these matrix layouts are a replacement for both the standard formats of DLA, namely full and packed storage. These new layouts possess three good features: they are supported by level 3 BLAS and LAPACK full format routines, and they require minimal storage.

Acknowledgments

The results in this paper were obtained on five computers, an IBM, two SUN, Itanium and NEC computers. The IBM machine belongs to the Center for Scientific Computing at Aarhus, the SUN machines to the Danish Technical University, and the Itanium and NEC machines to the Danish Meteorological Institute. We would like to thank Bernd Dammann for consulting on the SUN systems; Niels Carl W. Hansen for consulting on the IBM system; and Bjarne Stig Andersen for obtaining the results on the Itanium and NEC computers.

References

1. Andersen, B.S., Gunnels, J., Gustavson, F.G., Reid, J.K., Waśniewski, J.: A fully portable high performance minimal storage hybrid format Cholesky algorithm. TOMS 31, 201–227 (2005)
2. Andersen, B.A., Gustavson, F.G., Waśniewski, J.: A recursive formulation of Cholesky factorization of a matrix in packed storage. TOMS 27(2), 214–244 (2001)
3. Anderson, E., et al.: LAPACK Users' Guide Release 3.0, SIAM, Philadelphia (1999), `http://www.netlib.org/lapack/`
4. Gunnels, J.A., Gustavson, F.G.: A New Array Format for Symmetric and Triangular Matrices. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 247–255. Springer, Heidelberg (2006)
5. Elmroth, E., Gustavson, F.G., Kågström, B., Jonsson, I.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)
6. Herrero, J.R.: A Framework for Efficient Execution of Matrix Computations, PhD thesis, Universitat Politècnica de Catalunya (May 2006)

Using Non-canonical Array Layouts in Dense Matrix Operations[⋆]

José R. Herrero and Juan J. Navarro

Computer Architecture Dept., Univ. Politècnica de Catalunya
C/ Jordi Girona 1-3, D6, ES-08034 Barcelona, Spain
{josepr, juanjo}@ac.upc.edu

Abstract. We present two implementations of dense matrix multiplication based on two different non-canonical array layouts: one based on a hypermatrix data structure (HM) where data submatrices are stored using a recursive layout; the other based on a simple block data layout with square blocks (SB) where blocks are arranged in column-major order. We show that the iterative code using SB outperforms a recursive code using HM and obtains competitive results on a variety of platforms.

1 Introduction

A matrix representation is a method used by a computer language to store matrices of more than one dimension in memory. Fortran and C use different schemes. Fortran uses "Column Major", in which all the elements for a given column are stored contiguously in memory. C uses "Row Major", which stores all the elements for a given row contiguously in memory. These two schemes are considered canonical storage. The default column/row-major order used by programming languages such as Fortran and C limits locality to a single dimension.

As processor speed continues to increase relative to memory speed, locality optimizations get to be a significant performance issue for algorithms operating on large matrices. Data has to be reused in cache as effectively as possible: locality has to be exploited. In low associativity caches conflicts should be avoided or reduced. A considerable amount of research has been conducted towards achieving an effective use of memory hierarchies. Although we tried to be complete we are sure we missed some references. Performance can only be obtained by matching the algorithm to the architecture and vice-versa [1,2]. Conventional techniques such as *cache bloking* or *tiling* [3,4,5,6], *precopying* [3,7] and *padding* [7,8] have been used extensively. In addition, alternative storage formats have been proposed to address the issue of locality. Next, we provide an overview of such work.

1.1 Serial Dense Codes Using Non-canonical Array Layouts

A submatrix storage was proposed in [9] with the purpose of minimizing the page faulting occurring in a paged memory system. The authors partition matrices

[⋆] This work was supported by the Ministerio de Educación y Ciencia of Spain (TIN2004-07739-C02-01).

into square submatrices, keeping one submatrix per page, obtaining orders of magnitude improvement in the number of page faults for several common matrix operations.

In the last ten years there have been several studies on the application of non-canonical array layouts in uniprocessor environments. Recursivity has been introduced into linear algebra codes. Block recursive codes for dense linear algebra computations appear to be well-suited for execution on machines with deep memory hierarchies because they are effectively blocked for all levels of the hierarchy [10,11]. Unfortunately, some block recursive algorithms do not interact well with the TLB [12]. This has led to the eruption of new storage formats [13,14,15,16,17,18].

Different authors refer to a given data layout using different names. A data layout where matrices are stored as submatrices which are in turn stored by columns has been named as Submatrix storage in [9], BC in [13,2], SB in [10,16,19,20,21], 4D in [22], BDL in [23], and TDL in [24]. In this document we will refer to such data layout as SB (Square Block Format). This name reflects the square nature of the submatrices and is the one used most extensively in the literature.

Some studies have focused on the use of quadtrees or Space Filling Curves (SFC) for serial dense codes. In [25], a recursive matrix multiplication algorithm with quadtrees was presented. This algorithm uses recursion down to the level of single array elements which causes a dramatic loss of performance. Later, the same authors improved performance by stopping recursion at 8×8 blocks [26]. In [27] the authors have experimented with five different SFCs (U, X, Z, Gray and Hilbert) on the matrix multiplication algorithm. The performance reported was similar for all five. Morton (Z) order has relative simplicity in calculating block addresses compared to the other orderings and is often the order of choice. In spite of its relative simplicity compared to other layouts, calculation of addresses in Morton layout is expensive. There are several indexing techniques which differ in their structure, but which all induce Morton order: Morton, level-order, and Ahnentafel indexing. These indexing schemes require bit manipulation unless a lookup table is precomputed [22]. Bit masks can be used when dimensions are powers of two [28]. However, this requires padding. In [29] the authors show how the strong locality features of Peano SFCs can be used to produce cache oblivious matrix operations.

Tiling can also be applied to non-canonical data layouts. In [23] the authors show that improved cache and TLB performance can be achieved when tiling is applied to both Block Data Layout (BDL) and Morton layout. In their experiments matrix multiplication with an iterative code using BDL was often faster than a recursive code using Morton layout. As we will comment below, our results agree with this: our iterative tiled algorithm working on SB outperforms the recursive code operating on hypermatrices. Authors have also investigated on tile size selection for non-canonical array layouts [30,23,31] and have come to similar conclusions to the case of canonical storage: blocks should target the level 1 cache.

2 A Bottom-Up Approach

We have studied two data structures for dense matrix computations: a Hypermatrix data structure [32] and a Square Block Format [16]. We present them in section 3. In both cases we drive the creation of the structure from the bottom: the inner kernel fixes the size of the data submatrices. Then the rest of the data structure is produced in conformance. We do this because the performance of the inner kernel has a dramatic influence in the overall performance of the algorithm. Thus, our first priority is to use the best inner kernel at hand. Afterwards, we can adapt the rest of the data structure (in case hypermatrices are used) and/or the computations.

2.1 Inner Kernel Based on Our Small Matrix Library (SML)

In previous papers [33,24] we presented our work on the creation of a Small Matrix Library (SML): a set of routines, written in Fortran, specialized in the efficient operation on matrices which fit in the first level cache. The advantage of our method lies in the ability to generate very efficient inner kernels by means of a good compiler. Working on regular codes for small matrices, most of the compilers we have used in different platforms create very efficient inner kernels for matrix multiplication. We use the matrix multiplication routine within our SML as the inner kernel of our general matrix multiplication codes.

3 Non-canonical Array Layouts

In this section we briefly describe two non-canonical data layouts: a hypermatrix scheme and a simple square block format.

3.1 Hypermatrix Structure

We have used a data structure based on a hypermatrix (HM) scheme [32], in which a matrix is partitioned recursively into blocks of different sizes. The HM structure consists of N levels of submatrices, where N is an arbitrary number. In order to have a simple HM data structure which is easy to traverse we have chosen to have blocks at each level which are multiples of the lower levels. The top *N-1* levels hold pointer matrices which point to the next lower level submatrices. Only the last (bottom) level holds data matrices (see Figure 1). Data matrices are stored as dense matrices and operated on as such. Hypermatrices can be seen as a generalization of quadtrees. The latter partition each matrix precisely into four submatrices [34].

We have used a HM on dense Cholesky factorization and matrix multiplication with encouraging results. In [35] we showed that the use of orthogonal blocks [36] was beneficial to obtain performance. However, this approach presents some overhead following pointers and recursing down to the data submatrix level. There are also difficulties in the parallelization [37]. For these reasons we have also experimented with a Square Block Format.

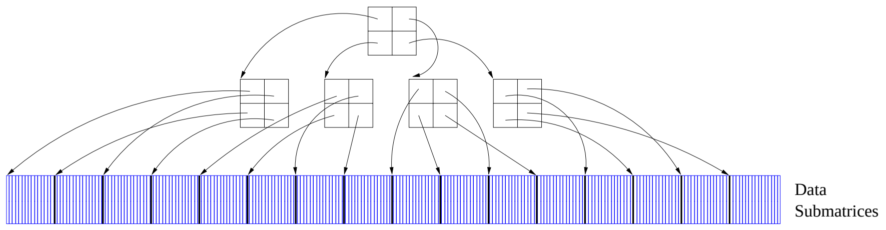


Fig. 1. Hypermatrix: example with two level of pointer matrices

3.2 Square Block Format

The overhead of a dense code based on recursion and hypermatrices together with the difficulties to produce efficient parallel code based on this data structure, has led us to experiment with a different data structure. We use a simple Square Block Format (SB) [16] stored as a 4D array. It corresponds to a 2D data layout of submatrices kept in column-major order as can be seen in the left part of Figure 2. We use a simple data structure which is described graphically in the right part of Figure 2. The shaded area at the top represents padding introduced to force data alignment.

Using this data structure we were able to improve the performance of our matrix multiplication code, obtaining very competitive results. Our code implements tiling and we use a code generator to create different loop orders. We present the results obtained with the best loop order found.

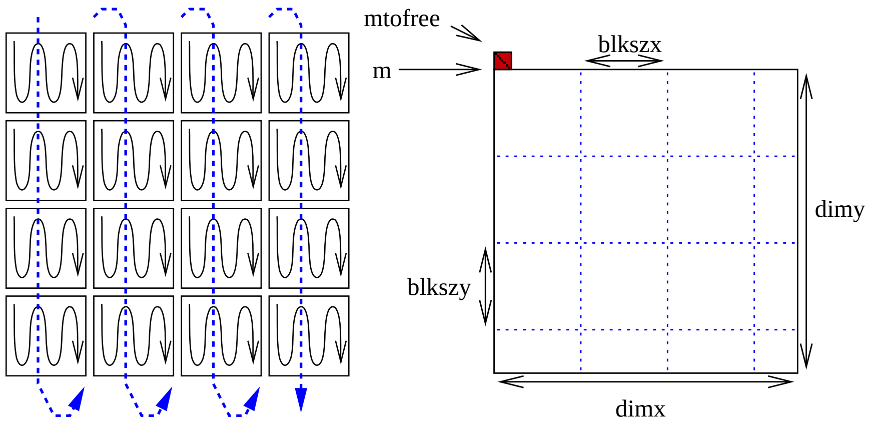


Fig. 2. Square Block Format

4 Results

We present results for matrix multiplication on three platforms. The matrix multiplication performed is $C = C - A^T \times B$. Each of the following figures shows the results of DGEMM in ATLAS [38], Goto [39] or the vendor BLAS, and our code based on SB format and our SML. Goto BLAS are known to obtain excellent performance on Intel platforms. They are coded in assembler and targeted to each particular platform. The dashed line at the top of each plot shows the theoretical peak performance of the processor. Some plots show the performance obtained with the dense codes based on the hypermatrix (HM) scheme. It can be seen on the plots that SB outperforms HM.

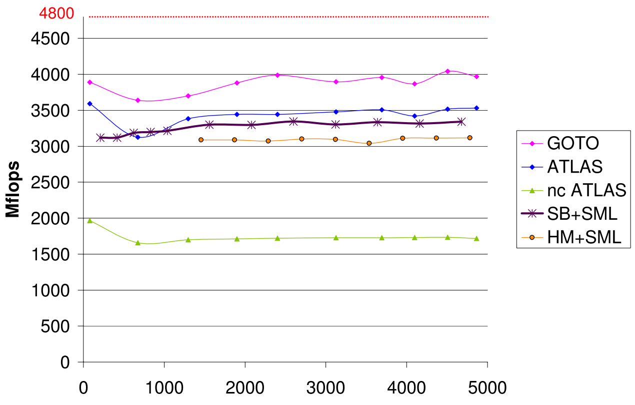


Fig. 3. Performance of DGEMM on Intel Pentium 4 Xeon

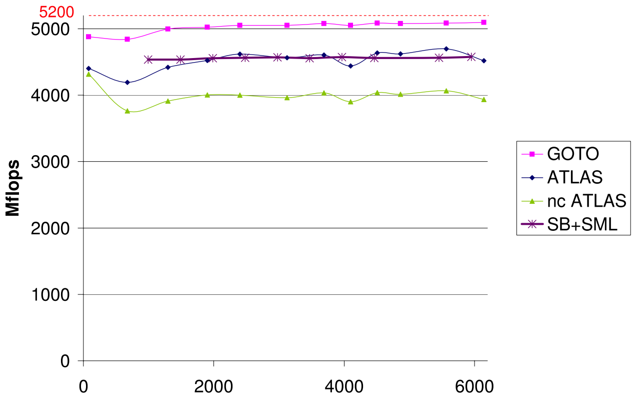


Fig. 4. Performance of DGEMM on Intel Itanium 2

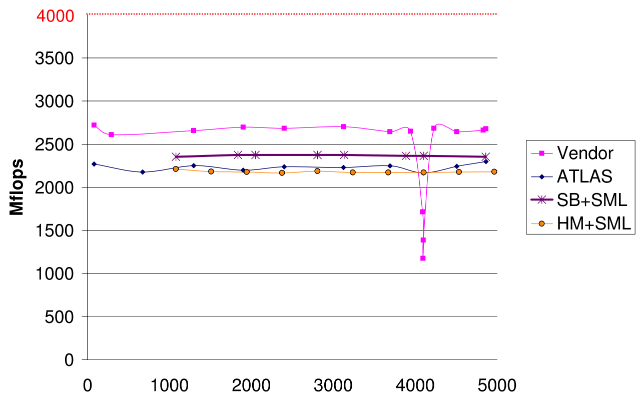


Fig. 5. Performance of DGEMM on Power 4

For the Intel machines (figures 3 and 4) we have included the Mflops obtained with a version of the ATLAS library where the hand-made codes were not enabled at ATLAS installation time[1]. We refer to this code in the graphs as 'nc ATLAS'. We can observe that in both cases ATLAS performance drops heavily. SB with SML kernels obtain performance close to that of ATLAS on the Pentium 4 Xeon, similar to ATLAS on the Itanium 2, and better than ATLAS on the Power4 (Figure 5). For the latter we show the Mflops obtained by the vendor DGEMM routine which outperforms both ATLAS and our code based on SB. We can see that even highly optimized routines provided by the vendor may be significantly slower under certain circumstances. For instance, some large leading dimensions can be particularly harmful and produce lots of TLB misses if data is not precopied. At the same time, data precopying must be performed selectively due to the overhead incurred at execution time [8]. These problems can be avoided using non-canonical array layouts.

5 Conclusions

The results obtained with an iterative code working on a Square Block Format surpass those obtained with a recursive code which uses a hypermatrix. This happens even when the upper levels of the hypermatrix are defined so that the upper levels of the memory hierarchy are properly exploited. This is due to the overhead caused by recursion. We conclude that a simple Square Block

[1] Directory `tune/blas/gemm/CASES` within the ATLAS distribution contains about 90 files which are, in most cases, written in assembler, or use some instructions written in assembler to do data prefetching. Often, one (or more) of these codes outperform the automatically generated codes. The best code is (automatically) selected as the inner kernel. The use of such hand-made inner kernels has improved significantly the overall performance of ATLAS subroutines on some platforms.

format provides a good way to exploit locality and iterative codes can outperform recursive codes. Our results agree with those presented in [23,40].

References

1. Agarwal, R.C., Gustavson, F.G., Zubair, M.: Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. IBM J. Res. Dev. 38, 563–576 (1994)
2. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46, 3–45 (2004)
3. IBM: ESSL Guide and Reference for IBM ES/3090 Vector Multiprocessors (1986) Order No. SA 22-9220 (Febuary 1986)
4. Gallivan, K., Jalby, W., Meier, U., Sameh, A.: Impact of hierarchical memory systems on linear algebra algorithm design. Int. J. of Supercomputer Appl. 2, 12–48 (1988)
5. Irigoin, F., Triolet, R.: Supernode partitioning. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 319–329. ACM Press, New York (1988)
6. Wolfe, M.: More iteration space tiling. In: ACM (ed.) Supercomputing '89, Reno, Nevada, November 13-17, 1989, pp. 655–664. ACM Press, New York (1989)
7. Lam, M., Rothberg, E., Wolf, M.: The cache performance and optimizations of blocked algorithms. In: Proceedings of ASPLOS '91, pp. 67–74 (1991)
8. Temam, O., Granston, E.D., Jalby, W.: To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In: Supercomputing, pp. 410–419 (1993)
9. McKellar, A.C., Coffman, J.E.G.: Organizing matrices and matrix operations for paged memory systems. Communications of the ACM 12, 153–165 (1969)
10. Gustavson, F.G.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM J. Res. Dev. 41, 737–756 (1997)
11. Toledo, S.: Locality of reference in LU decomposition with partial pivoting. SIAM J. Matrix Anal. Appl. 18, 1065–1081 (1997)
12. Ahmed, N., Pingali, K.: Automatic generation of block-recursive codes. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 368–378. Springer, Heidelberg (2000)
13. Gustavson, F., Henriksson, A., Jonsson, I., Kågström, B.: Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In: Kagström, B., Elmroth, E., Waśniewski, J., Dongarra, J.J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 195–206. Springer, Heidelberg (1998)
14. Andersen, B.S., Gustavson, F.G., Karaivanov, A., Marinova, M., Wasniewski, J., Yalamov, P.Y.: LAWRA: Linear algebra with recursive algorithms. In: Sørevik, T., Manne, F., Moe, R., Gebremedhin, A.H. (eds.) PARA 2000. LNCS, vol. 1947, pp. 38–51. Springer, Heidelberg (2001)
15. Andersen, B.S., Wasniewski, J., Gustavson, F.G.: A recursive formulation of Cholesky factorization of a matrix in packed storage. ACM Transactions on Mathematical Software (TOMS) 27, 214–244 (2001)

16. Gustavson, F.G.: New generalized data structures for matrices lead to a variety of high-performance algorithms. In: Engquist, B. (ed.) Simulation and visualization on the grid: Parallelldatorcentrum, Kungl. Tekniska Högskolan, proceedings 7th annual conference. Lecture Notes in Computational Science and Engineering, vol. 13, pp. 46–61. Springer, Heidelberg (1974)
17. Andersen, B.S., Gunnels, J.A., Gustavson, F., Wasniewski, J.: A recursive formulation of the inversion of symmetric positive defite matrices in packed storage data format. In: Fagerholm, J., Haataja, J., Järvinen, J., Lyly, M., Råback, P., Savolainen, V. (eds.) PARA 2002. LNCS, vol. 2367, pp. 287–296. Springer, Heidelberg (2002)
18. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Waśniewski, J.: A fully portable high performance minimal storage hybrid format Cholesky algorithm. ACM Transactions on Mathematical Software 31, 201–227 (2005)
19. Gustavson, F.G.: High-performance linear algebra algorithms using new generalized data structures for matrices. IBM J. Res. Dev. 47, 31–55 (2003)
20. Gustavson, F.G.: New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 11–20. Springer, Heidelberg (2006)
21. Gustavson, F.G.: Algorithm Compiler Architecture Interaction Relative to Dense Linear Algebra. Technical Report RC23715 (W0509-039), IBM, T.J. Watson (2005)
22. Chatterjee, S., Jain, V.V., Lebeck, A.R., Mundhra, S., Thottethodi, M.: Nonlinear array layouts for hierarchical memory systems. In: Proceedings of the 13th international conference on Supercomputing, pp. 444–453. ACM Press, New York (1999)
23. Park, N., Hong, B., Prasanna, V.K.: Tiling, block data layout, and memory hierarchy performance. IEEE Trans. Parallel and Distrib. Systems 14, 640–654 (2003)
24. Herrero, J.R., Navarro, J.J.: Compiler-optimized kernels: An efficient alternative to hand-coded inner kernels. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganà, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3984, pp. 762–771. Springer, Heidelberg (2006)
25. Frens, J.D., Wise, D.S.: Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In: Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program, SIGPLAN Notices, pp. 206–216 (1997)
26. Wise, D.S., Frens, J.D.: Morton-order matrices deserve compilers' support. Technical Report TR 533, Computer Science Department, Indiana University (1999)
27. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast parallel matrix multiplication. In: Proc. of the 11th annual ACM symposium on Parallel algorithms and architectures, pp. 222–231. ACM Press, New York (1999)
28. Athanasaki, E., Koziris, N.: Fast indexing for blocked array layouts to improve multi-level cache locality. In: Interaction between Compilers and Computer Architectures, pp. 109–119 (2004)
29. Bader, M., Mayer, C.: Cache oblivious matrix operations using Peano curves. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 521–530. Springer, Heidelberg (2007)
30. Valsalam, V., Skjellum, A.: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurrency and Computation: Practice and Experience 14, 805–839 (2002)

31. Athanasaki, E., Koziris, N., Tsanakas, P.: A tile size selection analysis for blocked array layouts. In: Interaction between Compilers and Computer Architectures, pp. 70–80 (2005)
32. Fuchs, G., Roy, J., Schrem, E.: Hypermatrix solution of large sets of symmetric positive-definite linear equations. Comp. Meth. Appl. Mech. Eng. 1, 197–216 (1972)
33. Herrero, J.R., Navarro, J.J.: Automatic benchmarking and optimization of codes: an experience with numerical kernels. In: Int. Conf. on Software Engineering Research and Practice, pp. 701–706. CSREA Press (2003)
34. Wise, D.S.: Representing matrices as quadtrees for parallel processors. Information Processing Letters 20, 195–199 (1985)
35. Herrero, J.R., Navarro, J.J.: Adapting linear algebra codes to the memory hierarchy using a hypermatrix scheme. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 1058–1065. Springer, Heidelberg (2006)
36. Navarro, J.J., Juan, A., Lang, T.: MOB forms: A class of Multilevel Block Algorithms for dense linear algebra operations. In: Proceedings of the 8th International Conference on Supercomputing, pp. 354–363. ACM Press, New York (1994)
37. Herrero, J.R., Navarro, J.J.: A study on load imbalance in parallel hypermatrix multiplication using OpenMP. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 124–131. Springer, Heidelberg (2006)
38. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Supercomputing '98, pp. 211–217. IEEE Computer Society Press, Los Alamitos (1998)
39. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Univ. of Texas at Austin (2002)
40. Gunnels, J.A., Gustavson, F.G., Pingali, K., Yotov, K.: Is cache-oblivious DGEMM viable? In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 919–928. Springer, Heidelberg (2007)

New Data Distribution for Solving Triangular Systems on Distributed Memory Machines[⋆]

Przemysław Stpiczyński

Department of Computer Science, Maria Curie–Skłodowska University
Pl. M. Curie-Skłodowskiej 1, PL-20-031 Lublin, Poland
przem@hektor.umcs.lublin.pl

Abstract. The aim is to present a new data distribution of triangular matrices that provides steady distribution of blocks among processes and reduces memory wasting compared to the standard block-cyclic data layout used in the ScaLAPACK library for dense matrix computations. A new algorithm for solving triangular systems of linear equations is also introduced. The results of experiments performed on a cluster of Itanium 2 processors and Cray X1 show that in some cases, the new method is faster than corresponding PBLAS routines PSTRSV and PSTRSM.

1 Introduction: Block-Cyclic Distribution

The problem of solving triangular systems of linear equations belongs to the most important tasks of computational linear algebra, and a large number of papers has been published about handling such systems on distributed memory architectures [3,8,9,11]. The most popular solvers, namely PSTRSV and PSTRSM for solving $Ax = b$ and $AX = B$ (for multiple right hand sides) respectively, belong to Level 2 and Level 3 PBLAS libraries designed as a part of ScaLAPACK [2]. They use the block-cyclic data distribution of matrices onto rectangular process grids. A process in a $P \times Q$ grid can be referenced by its coordinates (p, q), where $0 \le p < P$ and $0 \le q < Q$. The general class of block-cyclic distributions can be obtained by matrix partitioning like

$$A = \begin{pmatrix} A_{11} & \dots & A_{1M} \\ \vdots & & \vdots \\ A_{M1} & \dots & A_{MM} \end{pmatrix} \tag{1}$$

where each block A_{ij} is $m_b \times m_b$. Let $loc(A_{ij})$ denote coordinates (location) of A_{ij} in a grid. The blocks are mapped to processes by assigning A_{ij} to the process whose coordinates in a $P \times Q$ grid are

$$loc(A_{ij}) = ((i-1) \ \mathbf{mod} \ P, (j-1) \ \mathbf{mod} \ Q) \,. \tag{2}$$

[⋆] The work has been sponsored by the KBN grant 6T11 2003C/06098. The use of Cray X1 from the Interdisciplinary Center for Mathematical and Computational Modeling (ICM) of the Warsaw University is kindly acknowledged.

When A is a triangular (or symmetric) matrix, then only the lower (or upper) triangle is distributed, so about half of the storage is wasted (Figure 1). To avoid memory wasting, some improvements have been proposed [4,1,7]. However, these solutions still support block-cyclic data distribution, thus the performance of the algorithms `PSTRSV` and `PSTRSM` cannot be easily improved.

A_{11}	
$A_{31}\ A_{33}$	A_{32}
$A_{51}\ A_{53}\ A_{55}$	$A_{52}\ A_{54}$
$A_{71}\ A_{73}\ A_{75}\ A_{77}$	$A_{72}\ A_{74}\ A_{76}$
A_{21}	A_{22}
$A_{41}\ A_{43}$	$A_{42}\ A_{44}$
$A_{61}\ A_{63}\ A_{65}$	$A_{62}\ A_{64}\ A_{66}$
$A_{81}\ A_{83}\ A_{85}\ A_{87}$	$A_{82}\ A_{84}\ A_{86}\ A_{88}$

Fig. 1. Block-cyclic distribution of a lower triangular matrix over a 2×2 rectangular process grid. About 50% of the memory is wasted.

2 New Data Distribution

First let us consider a special case of the block-cyclic distribution which gives a better performance when the number of available processors is small (less than nine [2]), or especially when we want to solve triangular systems with a relatively small number of right hand sides, namely the block-cyclic distribution over a $P \times 1$ grid.

The system of linear equations $AX = B$, $A \in \mathbb{R}^{m\times m}$, $B \in \mathbb{R}^{m\times n}$ can be rewritten in the following block form

$$\begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & \vdots & \ddots & \\ A_{M1} & \dots & \dots & A_{MM} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_M \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix}, \tag{3}$$

and then the data distribution can be depicted as in Figure 2. It is clear that the block size m_b should be small enough to minimize memory wasting. On the other hand, m_b should be large enough to utilize the cache memory and provide a reasonable performance of BLAS operations performed on blocks of A [5], thus choosing the optimal block size leads to a tradeoff between performance and memory. Moreover, such a distribution leads to a kind of unbalance. For example, the first processor holds six blocks, while the last holds twelve (Figure 2).

In [12] we have introduced a new method for the distribution of symmetric and triangular matrices on orthogonal memory multiprocessors. The method can significantly reduce memory wasting and can be adopted for distributing blocks

A_{11} A_{51} A_{52} A_{53} A_{54} A_{55}	B_1 B_5
A_{21} A_{22} A_{61} A_{62} A_{63} A_{64} A_{65} A_{66}	B_2 B_6
A_{31} A_{32} A_{33} A_{71} A_{72} A_{73} A_{74} A_{75} A_{76} A_{77}	B_3 B_7
A_{41} A_{42} A_{43} A_{44} A_{81} A_{82} A_{83} A_{84} A_{85} A_{86} A_{87} A_{88}	B_4 B_8

Fig. 2. Block-cyclic distribution of a triangular matrix A and rectangular matrix B over a 4×1 grid

A_{ij} of a triangular (or symmetric) matrix over a $P \times 1$ process grid. Let us assume that $M = 2P$ and define the new data distribution as

$$loc(A_{ij}) = \begin{cases} (i-1, 0) & \text{for } i = 1, \ldots, P \\ (2P - i, 0) & \text{for } i = P+1, \ldots, 2P. \end{cases} \tag{4}$$

Figure 2 shows the example of such a distribution for $P = 4$. Simply, we divide the lower (or upper) triangle into two parts. The number of block rows is $2P$. The blocks from the upper part are distributed from the first to the last processor and the blocks from the lower part are distributed bottom up. Note that each processor holds exactly the same number of blocks of A. Also, BLAS routines (used locally) can operate on larger blocks, so one can expect that calls to such routines will be more efficient.

A_{11} A_{81} A_{82} A_{83} A_{84} A_{85} A_{86} A_{87} A_{88}	B_1 B_8
A_{21} A_{22} A_{71} A_{72} A_{73} A_{74} A_{75} A_{76} A_{77}	B_2 B_7
A_{31} A_{32} A_{33} A_{61} A_{62} A_{63} A_{64} A_{65} A_{66}	B_3 B_6
A_{41} A_{42} A_{43} A_{44} A_{51} A_{52} A_{53} A_{54} A_{55}	B_4 B_5

Fig. 3. New distribution of a triangular matrix A and rectangular matrix B over a 4×1 grid

The block size is determined by the number of processors, namely all square blocks A_{ij}, $i < M$, are $m_b \times m_b$, where

$$m_b = \lfloor m/(2P) \rfloor. \tag{5}$$

The leading dimension of all blocks A_{Mj} is

$$m'_b = (m - (M-1)m_b). \tag{6}$$

Thus, the block size of A_{MM} is $m'_b \times m'_b$, while other blocks A_{Mj} are $m'_b \times m_b$. All blocks B_j, $j < M$, are $m_b \times n$, while B_M is $m'_b \times n$.

Figure 4 shows local data structures for triangular matrices. Each process allocates a two dimensional array with the same leading dimension equal to m_b defined by (5) or (6) in case of the process $(0, 0)$. Nonzero blocks of A and B are stored consecutively using conventional storage by columns. Note that the

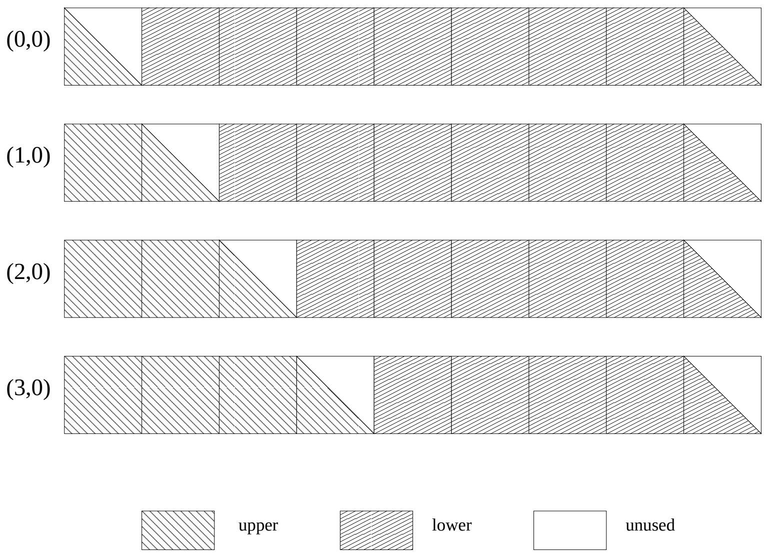


Fig. 4. New data distribution: local data structures for triangular matrices. "`upper`" blocks represent A_{ij} with $i \le P$, while "`lower`" blocks with $i > P$.

unused parts are relatively small. Each process holds exactly $2P+1$ blocks of a triangular matrix and approximately one block is unused. Thus, about

$$\frac{100}{2P+1}\%$$

of the memory is wasted. For $P = 4, 8, 16$, we get respectively 11%, 6%, and 3% of memory wasted.

It should be pointed out that the the new data distribution has a disadvantage. It does not allow the reuse of standard ScaLAPACK routines. The conversion between the standard block-cyclic data layout and the new distribution requires a data redistribution that can decrease the efficiency of the algorithms.

3 Implementation and Experimental Results

The following Algorithm 1 is a block version of the fan-out algorithm presented in [8]. It solves systems of linear equations $AX = B$, where $A \in \mathbb{R}^{m\times m}$ and $B \in \mathbb{R}^{m\times n}$, just like the Level 3 PBLAS routine `PSTRSM`. Note that for $n = 1$, it performs the same computational task as the Level 2 PBLAS routine `PSTRSV`. Our new Algorithm 2 is a distributed SPMD version of Algorithm 1, using the new data distribution (4).

The method has been implemented in Fortran and tested on a cluster of 16 Itanium 2 processors (1.3 GHz, 3 MB cache, approximately 5 Gflops peak performance, 8 dual processor nodes, Gbit Ethernet, running under Linux), using

Algorithm 1. Solving triangular systems (3)

```
Require: nonsingular, lower triangular matrix $A \in \mathbb{R}^{m\times m}$ and general matrix $B \in \mathbb{R}^{m\times n}$
Ensure: $B = A^{-1}B$
 1: for $i = 1$ to $M$ do
 2:    $B_i \leftarrow A_{ii}^{-1}Bi$ {STRSM}
 3:    for $j = i+1$ to $M$ do
 4:       $B_j \leftarrow B_j - A_{ji}B_i$ {SGEMM}
 5:    end for
 6: end for
```

Algorithm 2. Process $(i,0)$, $i = 0,\dots,P-1$, for solving triangular systems (3)

```
Require: nonsingular, triangular matrix $A \in \mathbb{R}^{m\times m}$ and general matrix $B \in \mathbb{R}^{m\times n}$
    distributed using the new data distribution (4)
Ensure: $B = A^{-1}B$
 1: $M \leftarrow 2P$
 2: for $j = 1$ to $P$ do
 3:    if $loc(A_{jj}) = (i,0)$ then
 4:       $B_j \leftarrow A_{jj}^{-1}B_j$ {STRSM}
 5:       send $B_j$ to all $\{(k,0):\ k = 0,\dots,P-1 \wedge k \neq i\}$
 6:       $B_{2P-j+1} \leftarrow B_{2P-j+1} - A_{2P-j+1,j}B_j$ {SGEMM}
 7:    else
 8:       receive $X$ from $(loc(A_{jj}),0)$
 9:       if $i > loc(A_{jj})$ then
10:          $B_{i+1} \leftarrow B_{i+1} - A_{i+1,j}X$ {SGEMM}
11:       end if
12:       $B_{2P-i} \leftarrow B_{2P-i} - A_{2P-i,j}X$ {SGEMM}
13:    end if
14: end for
15: for $j = P+1$ to $M-1$ do
16:    if $loc(A_{jj}) = (i,0)$ then
17:       $B_j \leftarrow A_{jj}^{-1}B_j$ {STRSM}
18:       send $B_j$ to all $\{(k,0):\ k = 0,\dots,i-1\}$
19:    else
20:       if $i < loc(A_{jj})$ then
21:          receive $X$ from $(loc(A_{jj}),0)$
22:          $B_{i+1} \leftarrow B_{i+1} - A_{i+1,j}X$ {SGEMM}
23:       end if
24:    end if
25: end for
26: if $loc(A_{MM}) = (i,0)$ then
27:    $B_M \leftarrow A_{MM}^{-1}B_M$ {STRSM}
28: end if
```

Intel Fortran Compiler and Math Kernel Library as the efficient implementations of BLAS [5] and on 8 MSPs of Cray X1. Each Cray X1 processor, namely MSP, comprises four SSPs. Each SSP consists of a fast vector processor and a very slow (400 MHz) scalar processor for scalar operations. The MSPs support multistreaming, which means that vectors can be processed across all four SSPs. Four MSPs (i.e. one node) operate in the symmetric multiprocessing mode, so they have access to shared memory. To access more MSPs, a user can use MPI, SHMEM, CAF or UPC [10]. The peak performance of one MSP is 18 Gflops. In the case of the Itanium cluster, the libraries PBLAS [2] (routines `PSTRSV` and `PSTRSM`) and BLACS [6] (based on MPI, used for communication) have been downloaded from the Netlib. For the Cray we have used corresponding routines provided by Cray.

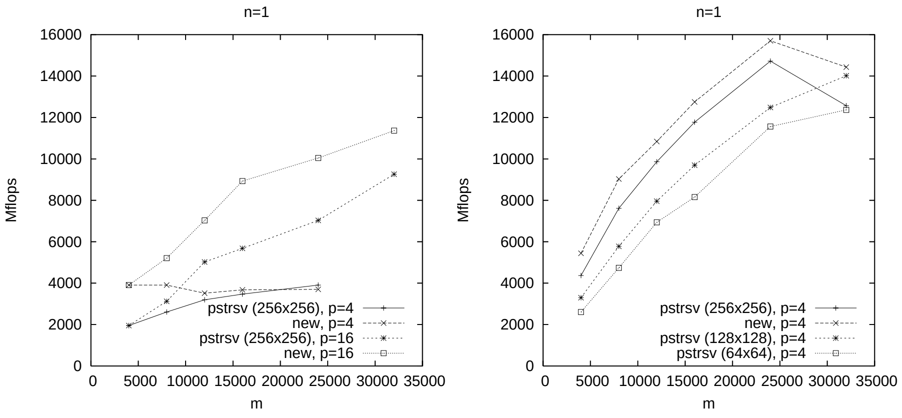


Fig. 5. The performance (in Mflops) of the PBLAS routine `PSTRSV` and the new method on a cluster of Itanium 2 (left) and Cray X1 (right) for various matrix sizes (m) and $n = 1$, obtained using a $P \times 1$ grid

We have compared the performance of the new method with the performance of the routine `PSTRSV` (Figure 5). On the Itanium cluster, the best performance of `PSTRSV` is achieved for the block size 256×256. The new method is faster on 4 and 16 processors. The results are similar on the Cray X1. Figure 6 shows the performance of `PSTRSM` (optimal block size 64×64) and the new solver on the Itanium cluster. We can observe that the performance of the new solver is comparable to the performance of `PSTRSM`, but when we increase the size of the matrix and the number of processors, the performance of the new solver is explicitly better. Note that for smaller number of processors, the performance of the new solver is a little bit worse. Figure 7 compares the performance of the new solver to the performance of `PSTRSM` (for the optimal block size 128×128) on Cray X1. One can observe that Algorithm 2 outperforms `PSTRSM` when $P = 8$. This is a consequence of the reduced communication overhead. The new algorithm operates on larger matrices, so the total number of broadcasts (lines 5, 18) is reduced in comparison to the routines `PSTRSV` and `PSTRSM` which operates on smaller blocks.

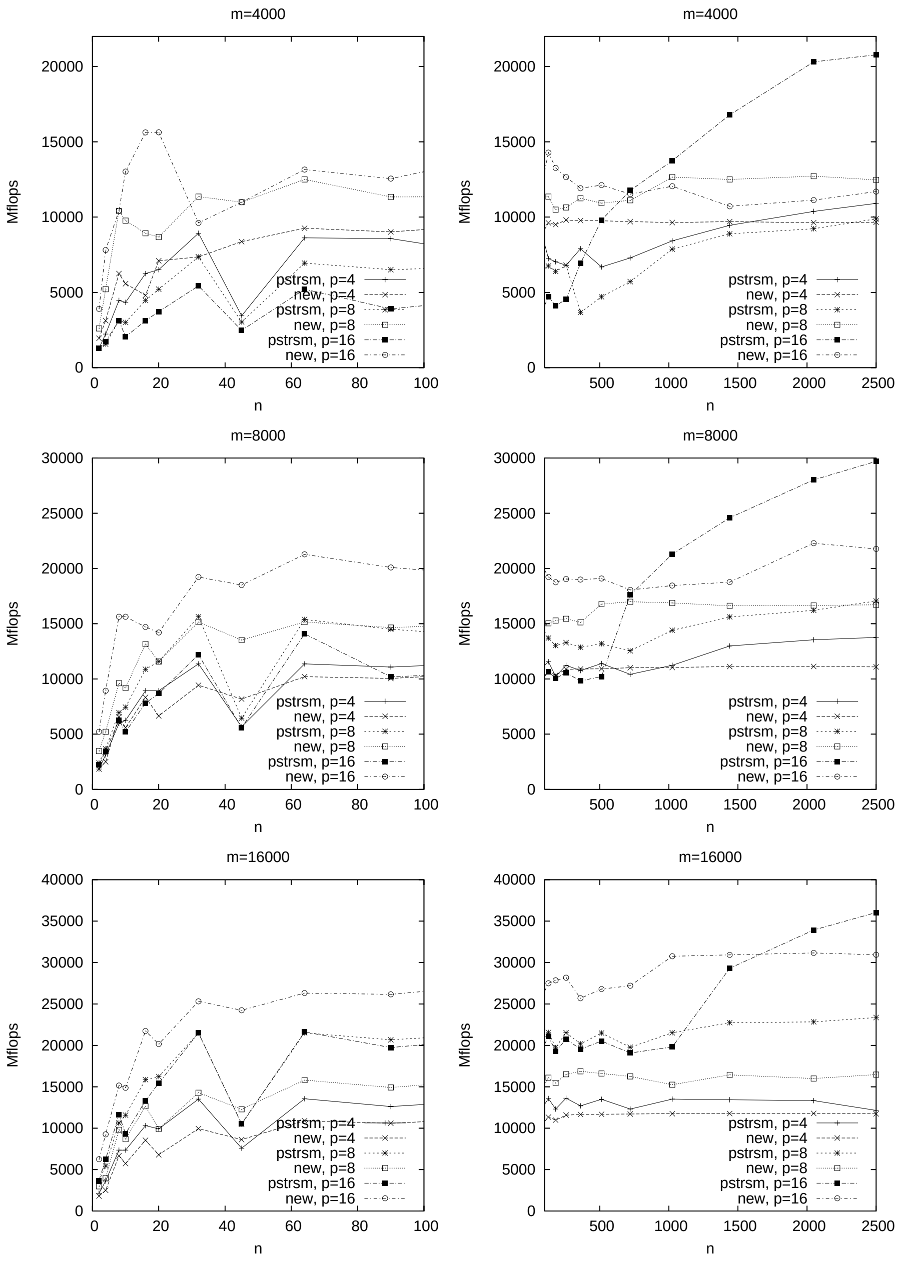


Fig. 6. The performance (in Mflops) of the PBLAS routine PSTRSM and the new method on a cluster of Itanium 2 for various matrix sizes $n = 2, \ldots, 2500$, $m = 4000, 8000, 16000$ and numbers of processors $p = 4, 8, 16$

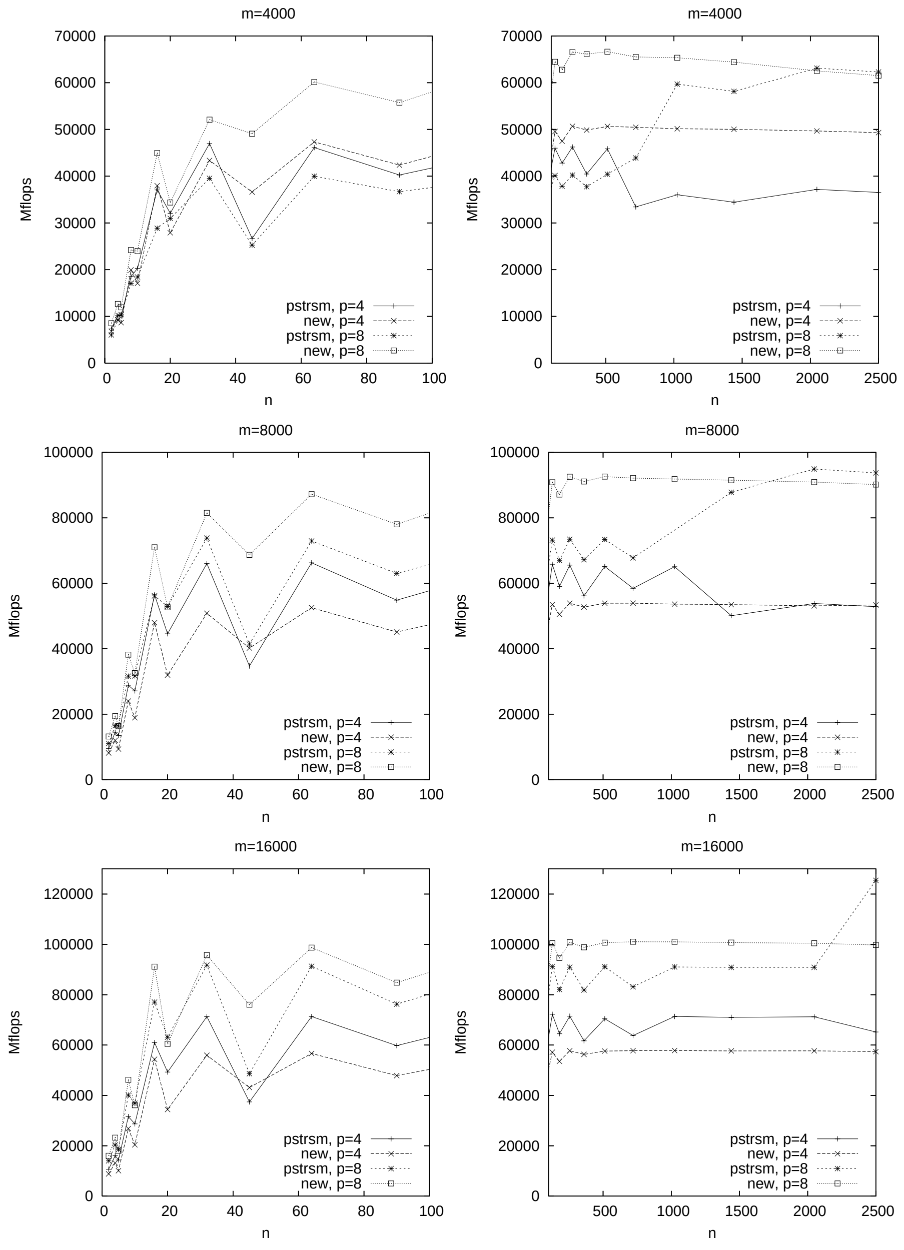


Fig. 7. The performance (in Mflops) of the PBLAS routine `PSTRSM` and the new method on Cray X1 for various matrix sizes $n = 2, \ldots, 2500$, $m = 4000, 8000, 16000$ and numbers of processors $p = 4, 8$

4 Conclusions and Future Work

The new data format allows each processor to allocate the same amount of memory in two dimensional arrays and only a small part is wasted. In some cases, the new solver is faster than corresponding PBLAS routines `PSTRSV` and `PSTRSM`. In the future we will consider the implementation of other PBLAS routines for triangular matrices. The same idea can be applied for symmetric matrices and some well known algorithms like the Cholesky factorization.

Acknowledgments

The author would like to thank the anonymous referees for valuable discussions and suggestions.

References

1. Baboulin, M., Giraud, L., Gratton, S., Langou, J.: A distributed packed storage for large dense parallel in-core calculations. Technical Report TR/PA/05/30, CERFACS (2005)
2. Blackford, L., et al.: ScaLAPACK User's Guide. SIAM, Philadelphia (1997)
3. Chaudron, M.R., van Duin, A.C.: The formal derivation of parallel triangular system solvers using a coordination-based design method. Parallel Comput. 24, 1023–1046 (1998)
4. D'Azevedo, E.F., Dongarra, J.J.: LAPACK Working Note 135: Packed storage extension for ScaLAPACK (1998)
5. Dongarra, J., Duff, I., Sorensen, D., Van der Vorst, H.: Solving Linear Systems on Vector and Shared Memory Computers. SIAM, Philadelphia (1991)
6. Dongarra, J.J., Whaley, R.C.: LAPACK Working Note 94: A user's guide to the BLACS, vol. 1.1 (1997), `http://www.netlib.org/blacs`
7. Gustavson, F.G., Karlsson, L., Kagstrom, B.: Three algorithms for Cholesky factorization on distributed memory using packed storage. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 550–559. Springer, Heidelberg (2007)
8. Heath, M., Romine, C.: Parallel solution of triangular systems on distributed memory multiprocessors. SIAM J. Sci. Statist. Comput. 9, 558–588 (1988)
9. Li, G., Coleman, T.F.: A new method for solving triangular systems on distributed-memory message-passing multiprocessors. SIAM J. Sci. Stat. Comput. 10, 382–396 (1989)
10. Netwok Computer Services Inc.: The AHPCRC Cray X1 primer, `http://www.ahpcrc.org/publications/Primer.pdf`
11. Romine, C., Ortega, J.: Parallel solutions of triangular systems of equations. Parallel Comput. 6, 109–114 (1988)
12. Stpiczyński, P.: Parallel Cholesky factorization on orthogonal multiprocessors. Parallel Computing 18, 213–219 (1992)

The Design of a New Out-of-Core Multifrontal Solver

John K. Reid and Jennifer A. Scott

Computational Science and Engineering Department,
Rutherford Appleton Laboratory,
Chilton, Oxfordshire, OX11 0QX, England, UK
{j.k.reid, j.a.scott}@rl.ac.uk

Abstract. Direct methods for solving large sparse linear systems of equations are popular because of their generality and robustness. Their main weakness is that the memory they require increases rapidly with problem size. We discuss the design and development of a new multifrontal solver that aims to circumvent this problem by allowing both the system matrix and its factors to be stored externally. We highlight some of the key features of our new out-of-core package, in particular its use of efficient implementations of dense linear algebra kernels to perform partial factorizations and its memory management system. We present numerical results for some large-scale problems arising from practical applications.

1 Introduction

The popularity of direct methods for solving large sparse linear systems of equations $Ax = b$ stems from their generality and robustness. Indeed, if numerical pivoting is properly incorporated, direct solvers rarely fail for numerical reasons; the main reason for failure is a lack of memory. Although increasing amounts of main memory have enabled direct solvers to solve many problems that were previously intractable, their memory requirements generally increase much more rapidly than problem size so that they can quickly run out of memory. Buying a machine with more memory is an expensive and inflexible solution since there will always be problems that are too large for the chosen quantity of memory. Using an iterative method may be a possible alternative but for many of the "tough" systems that arise from practical applications, the difficulties involved in finding and computing a good preconditioner can make iterative methods infeasible. Another possibility is to use a direct solver that is able to hold its data structures on disk, that is, an out-of-core solver.

The advantage of using disk storage is that it is many times cheaper than main memory per megabyte, making it practical and cost-effective to add tens or hundreds of gigabytes of disk space to a machine. By holding the main data structures on disk, well implemented out-of-core direct solvers are very reliable since they are much less likely than in-core solvers to run out of memory.

The idea of out-of-core linear solvers is not new (see, for example, [3],[8] and, recently, [2], [10]). Our aim is to design and develop a sparse symmetric out-of-core solver for inclusion within the mathematical software library HSL [6]. Our new Fortran 95 solver, which is called `HSL_MA77`, implements an out-of-core multifrontal algorithm. It is a uniprocessor code, but most of the arithmetic is performed by Level-3 BLAS kernels (see Section 5) that typically make use of any shared-memory parallelism. The first release of `HSL_MA77` is for positive-definite systems and performs a Cholesky factorization. The second release will have an option that incorporates numerical pivoting using 1×1 and 2×2 pivots, which will extend the package to symmetric indefinite problems. An important feature of the package is that all input and output to disk is performed through a set of Fortran subroutines that manage a virtual memory system so that actual input/output occurs only when really necessary. We describe this memory management system and highlight other key features of the new package, including its use of efficient implementations of dense linear algebra kernels.

2 Introduction to the Multifrontal Method

The multifrontal method is a variant of sparse Gaussian elimination and involves the matrix factorization

$$A = (PL)D(PL)^T,$$

where P is a permutation matrix and L is lower triangular. In the positive-definite case, $D = I$; in the indefinite case, D is block diagonal with blocks of size 1×1 and 2×2. Solving $Ax = b$ is completed by performing forward elimination followed by back substitution. The basic multifrontal algorithm for element problems is summarised in Figure 1. The assemblies can be held as

```
Given a pivot sequence:
 do for each pivot
     Assemble all elements that involve the pivot into a full matrix (the frontal
        matrix)
     Perform static condensation, that is, eliminate the pivot variable and any
        others that do not appear in any elements not yet assembled;
     Treat the reduced matrix as a new element (a generated element)
 end do
```

Fig. 1. The basic multifrontal algorithm

a tree, called an *assembly* tree. Each *node* of the assembly tree represents an element. When a pivot is eliminated and a generated element created, a node is added to the tree whose children are the elements that involve the pivot (these may be original elements and/or generated elements). If the nodes of the tree are ordered using a depth-first search, the generated elements required at each stage are the most recently generated ones of those so far unused. This makes it convenient to use a stack for temporary storage during the factorization. This alters the pivot sequence, but the arithmetic is identical apart from the round-off effects of reordering the assemblies.

The multifrontal method can be extended to non-element problems by regarding row i of A as a packed representation of a 1×1 element (the diagonal a_{ii}) and, for each $a_{ij} \neq 0$, a 2×2 element of the form

$$A^{(ij)} = \begin{pmatrix} 0 & a_{ij} \\ a_{ij} & 0 \end{pmatrix}.$$

When i is chosen as pivot, the 1×1 element plus the subset of 2×2 elements $A^{(ij)}$ for which j has not yet been selected as a pivot must be assembled. Rows that have an identical pattern may be treated together by grouping the corresponding variables into *supervariables.*

Multifrontal data structures

The multifrontal method needs data structures for the original matrix A, the frontal matrix, the stack of generated elements, and the matrix factor. An out-of-core method writes the columns of the factor to disk as they are computed and may also allow either the stack or both the stack and the frontal matrix to be held on disk. If the stack and frontal matrix are held in main memory and only the factors written to disk, the method performs the minimum possible input/output for an out-of-core method: it writes the factor data to disk once and reads it once during backsubstitution or twice when solving for further right-hand sides (once for the forward substitution and once for the backsubstitution). However, for very large problems, it may also be necessary to hold the stack or the stack and the frontal matrix on disk. Main memory requirements can be reduced further by holding the original matrix data on disk. `HSL_MA77` allows the original matrix data plus the factor and the stack to be held on disk; currently the frontal matrix is held in main memory.

3 Overview of the Structure of `HSL_MA77`

`HSL_MA77` is designed to solve one or more sets of sparse symmetric equations $AX = B$. To offer users flexibility in how the matrix data is held, A may be input in either by rows or by square symmetric elements (such as in a finite-element calculation). A reverse communication user interface is used, with control being returned to the calling program for each row or element. This keeps the memory requirements for the initial matrix to a minimum and gives the user maximum freedom as to how the original matrix data is held.

Given a pivot sequence, the multifrontal method can be split into a number of distinct phases:

– An analyse phase that uses the index lists for the rows or elements and the pivot sequence to construct the assembly tree. It also calculates the work and

storage required for the subsequent numerical factorization (in the indefinite case, based on the assumption that there are no delayed pivots).
– A factorization phase that uses the assembly tree to factorize the matrix and (optionally) solve systems of equations (in the indefinite case, the pivot order may have to be modified to maintain numerical stability).
– A solve phase that performs forward elimination followed by back substitution using the stored matrix factors.

The `HSL_MA77` package has separate routines for each of these phases but does not include routines for computing the pivot order. We note that there already exist a number of packages that can be used compute a nested dissection ordering (for example, the METIS graph partitioning package [7]) or a minimum degree based ordering (a number of variants are offered by the analyse phase of the HSL solver `MA57` [4]).

4 Virtual Memory Management

A key part of the design of `HSL_MA77` is that all input and output to disk is performed through a set of Fortran subroutines that manage a virtual memory system so that actual input/output occurs only when really necessary. This set of subroutines is available within HSL as the package `HSL_OF01`.

`HSL_OF01` provides read and write facilities from and to one or more direct-access files. To minimise the actual input-output operations, a single in-core buffer is used for reals and a single in-core buffer is used for integers. The advantage of having one buffer is that the available memory is dynamically shared between the files according to their needs at each stage of the computation. Each buffer is divided into *pages* that are all of the same size, which is also the size of each file record. All actual input-output is performed by transfers of pages between the buffer and records of the file.

The data in a file is addressed as a virtual array, that is, as if it were a very large array. The package allows any contiguous section of the virtual array to be read or written, without regard to page boundaries. The virtual array is permitted to be too large to be accommodated on a single file, in which case secondary files are used. These may reside on different devices. We refer to a primary file and its secondaries as a *super-file.*

The most recently accessed pages of the virtual array are held in the buffer. For each buffer page, the index of the super-file and the page number within the virtual array are stored. Wanted pages are found quickly with the help of a simple hashing function, and hash clashes are resolved by holding doubly-linked lists of pages having identical hash codes.

Once the buffer is full and another page is wanted, the least recently accessed buffer page is freed. It is identified quickly with the aid of a doubly-linked list of pages in order of access. This requires that whenever a page is accessed, it is removed from its old position in the doubly-linked list and inserted at the front. A special test is made for the page already being at the front since it can happen that there are many short reads and writes that fit within a single page.

A flag is kept for each page to indicate whether it has changed since its entry into the buffer so that only those that have been changed need be written to file when they are freed. On each call of `OF01_read` or `OF01_write`, all wanted pages that are in the buffer are accessed before those that are not in the buffer in order to avoid freeing a page that may be needed.

Because of the importance of assembly steps in the frontal method, we provide an option in `OF01_read` to add a section of the virtual array into an array under the control of a map.

The efficiency of reading to and writing from files using `HSL_OF01` depends on the number of pages in the buffer and the size of each page. These are parameters (`npage` and `lpage`) under the control of the user. Numerical experimentation has shown that as the buffer size increases the timings generally reduce but, for a given buffer size, the precise choice of `npage` and `lpage` is not critical, with a range of values giving similar performances. However, using either a small number of pages or a small page size can adversely effect performance. Based on our experiments, we have chosen the default values for these control parameters to be 1600 and 2^{12}, respectively, giving a buffer size of 6.6×10^6 reals or integers.

Option for in-core working

If the buffer is big enough, `HSL_OF01` will avoid any actual input/output, but there remain the overheads associated with copying data to and from the buffer. This is particularly serious in the solve phase for a single right-hand side since each datum read during the forward or backward substitution is used only once. We have therefore included the option of replacing the super-files by arrays. The user can specify the initial sizes of these arrays and a limit on their total size. If an array is found to be too small, the code attempts to reallocate it with a larger size. If this breaches the overall limit or if the allocation fails because of insufficient available memory on the computer being used, the contents of the array are written to a super-file and the memory that was used by the array is freed, resulting in a combination of super-files and in-core arrays being used. If a user specifies the total size without specifying the initial sizes of the individual arrays, the code automatically selects these sizes.

5 Kernel Code for Handling Full-Matrix Operations

The frontal matrix is a dense matrix that may be expressed in the form

$$\begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix},$$

where the fully-summed variables correspond to the rows and columns of F_{11}. In the positive-definite case, the operations can be blocked as the Cholesky factorization

$$F_{11} = L_{11} L_{11}^T,$$

the update operation

$$L_{21} = F_{21} L_{11}^{-T},$$

and the calculation of the generated element

$$S_{22} = F_{22} - L_{21}L_{21}^T.$$

Within HSL_MA77, we rely on a modification of the work of Andersen, Gunnels, Gustavson, Reid and Wasniewski [1] on the Cholesky factorization of a positive-definite full symmetric matrix to perform these operations and the corresponding forward and backward substitution operations. Andersen et. al. pack the upper or lower triangular part of the matrix into a *block hybrid* format that is as economical of storage as packing by columns but is able to take advantage of Level-3 BLAS. It divides the matrix into blocks, all of which are square and of the same size nb except for the blocks at the bottom which may have less rows. Each block is ordered by rows and the blocks are ordered by block columns. It is illustrated by the lower triangular case with order 9 and block size $nb = 3$ in Figure 2. Note that each of the blocks is held contiguously in memory.

The factorization is programmed as a sequence of blocks steps that involve the factorization of a block on the diagonal, the solution of a triangular set of equations with a block as it right-hand side, or the multiplication of two blocks. Andersen et. al. [1] have written a special kernel for the factorization of a block on the diagonal that uses blocks of size 2 to reduce traffic to the registers. The Level-3 BLAS DTRSM and DGEMM are available for the other two steps.

2a. Lower Packed Format

0									
1	10								
2	11	19							
3	12	20	27						
4	13	21	28	34					
5	14	22	29	35	40				
6	15	23	30	36	41	45			
7	16	24	31	37	42	46	49		
8	17	25	32	38	43	47	50	52	
9	18	26	33	39	44	48	51	53	54

2b. Lower Blocked Hybrid Format

0									
1	2								
3	4	5							
6	7	8	27						
9	10	11	28	29					
12	13	14	30	31	32				
15	16	17	33	34	35	45			
18	19	20	36	37	38	46	47		
21	22	23	39	40	41	48	49	50	
24	25	26	42	43	44	51	52	53	54

Fig. 2. Lower Packed and Blocked Hybrid Formats

We have chosen to work with the lower packed format so that it is easy to separate the pivoted columns that hold part of the factor from the other columns that must be assembled into the parent element. The modification involves limiting the eliminations to the fully-summed columns, the first p, say. The factorization of the frontal matrix takes the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & \\ & S_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix}$$

where L_{11} is lower triangular and both F_{11} and L_{11} have order p. We use the lower blocked hybrid format for the lower triangular part of both F_{11} and F_{22}.

Again, each is held by blocks of order nb, except that the final block may be smaller. The rectangular matrix F_{21} is held as a block matrix with matching block sizes. During factorization, these matrices are overwritten by the lower triangular parts of L_{11} and S_{22} and by L_{21}. We will call this format for F and its factorization the *double blocked hybrid* format.

The modified code is collected into the module HSL_MA54. It has facilities for rearranging a lower triangular matrix in lower packed format (that is, packed by columns) to double blocked hybrid format and vice-versa, for partial factorization, and for partial forward and back substitution, that is, solving equations of the form

$$\begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} X = B \quad \text{and} \quad \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix} X = B$$

and the corresponding equations for a single right-hand side b and solution x.

It is efficient to retain the factor in block hybrid format for the forward and backward substitution operations but HSL_MA77 reorders the matrix S_{22} back to lower packed format for the assembly operations at the parent node since the block structure at the parent is very unlikely to be the same.

6 Numerical Experiments

In this section, we illustrate the performance of HSL_MA77. A set of 26 positive-definite test problems taken from the University of Florida Sparse Matrix Collection (www.cise.ufl.edu/research/sparse/matrices/) were used in our experiments. Full details are given in [9]. The numerical results were obtained using double precision (64-bit) reals on a single 3.6 GHz Intel Xeon processor of a Dell Precision 670 with 4 Gbytes of RAM. The g95 compiler with the -O option was used with the ATLAS BLAS and LAPACK (math-atlas.sourceforge.net).

Times for each phase

In Table 1, we report the wall clock times for each phase of HSL_MA77 for a subset of our test problems. The input time is the time taken to input the matrix data using the reverse communication interface and the ordering time is the time to compute the pivot sequence using a nested dissection ordering. MA77_solve(k) is the time for the solve phase when called with k right-hand sides. The complete solution time is for a single right-hand side.

Comparisons with MA57

We now compare the performance of HSL_MA77 with that of the well-known HSL sparse direct solver MA57 [4]. MA57 is also a multifrontal code and is designed to solve both positive-definite and indefinite problems in assembled form; out-of-core facilities are not offered. In Figures 3 and 4, we compare the factorize and solve times for MA57 and for HSL_MA77 in-core (using arrays in place of files) with those for HSL_MA77 out-of-core. The figures show the ratios of the MA57 and HSL_MA77 in-core times to the HSL_MA77 out-of-core times. MA57 failed to solve problems 2, 11, 12, and 15 because of insufficient memory. For each of the

Table 1. Times for the different phases of HSL_MA77

Phase	af_shell3	cfd2	fullb	pwtk	thread
Input	2.18	0.42	0.55	1.32	0.45
Ordering	2.57	3.54	1.30	1.39	0.66
MA77_analyse	2.18	3.71	0.63	1.01	0.68
MA77_factor(0)	70.5	29.3	82.0	32.6	24.2
MA77_factor(1)	81.5	34.4	88.3	36.9	27.0
MA77_solve(1)	15.7	6.23	11.5	8.00	3.65
MA77_solve(10)	20.8	7.44	14.2	9.89	3.81
MA77_solve(100)	73.1	23.7	45.4	34.0	10.3
Complete solution	91.8	43.5	95.9	44.6	29.8

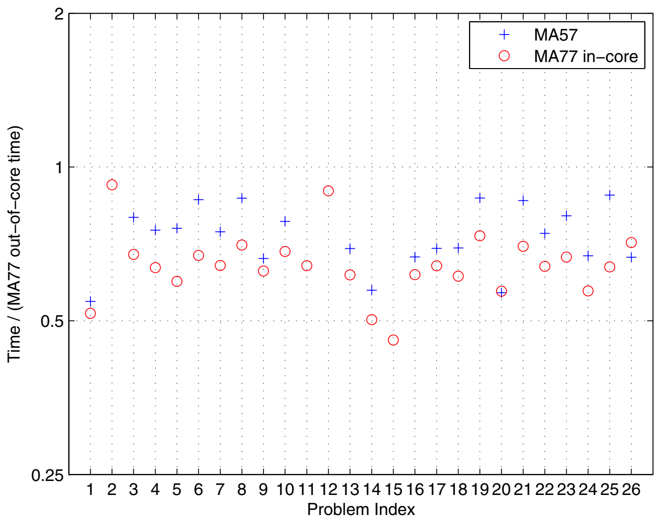


Fig. 3. The ratios of the MA57 and HSL_MA77 in-core factorize times to the HSL_MA77 out-of-core factorize times

problems, HSL_MA77 was able to successfully use a combination of arrays and files. Further details of the test examples that could not be solved in core are given in Table 2.

It is clear from Figure 3 that the factorization times for HSL_MA77 in-core are almost always faster than those for MA57 and that working out-of-core generally decreases the speed by about 30 per cent. For the solution phase with a single right-hand side, the penalty for working out-of-core is much greater because the ratio of data movement to arithmetic operations is significantly higher than for the factorization. This is evident in Figure 4. There are no HSL_MA77 in-core solve times shown for cases 2, 11, 12, and 15 because the main real super-file was not held in-core.

Table 2. Characteristics of the test problems that could not be solved in core. n is the order of A. $nz(A)$ and $nz(L)$ denote the number of entries in A and L, respectively, in millions. $front$ denotes the maximum order of frontal matrix.

	Identifier	n	$nz(A)$	$nz(L)$	$front$
2.	`audikw_1`	943,695	39.298	1283.170	11223
11.	`inline_1`	503,712	18.660	186.192	3261
12.	`ldoor`	952,203	23.737	164.531	2436
15.	`nd24k`	72,000	28.716	322.601	11363

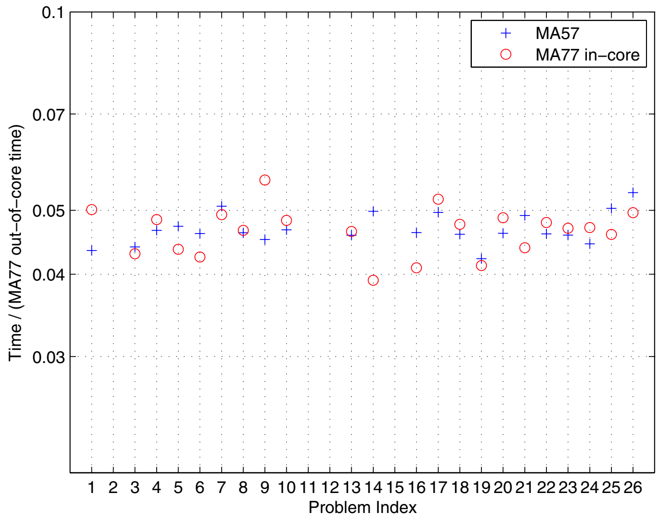


Fig. 4. The ratios of the `MA57` and `HSL_MA77` in-core solve times to the `HSL_MA77` out-of-core factorize times

7 Summary

In this paper, we have discussed some of the key features of a new out-of-core direct solver. The solver is designed for both positive-definite and indefinite problems but, although we have already written preliminary code for the indefinite case, work still remains to be done on developing an efficient indefinite kernel. The first release of our solver will thus be for positive-definite systems only. `HSL_MA77` together with the packages `HSL_OF01` and `HSL_MA54` that handle the out-of-core working and perform the dense linear algebra computations, respectively, will be included within the next release of HSL. Further details may be found in [9].

References

1. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Wasniewski, J.: A fully portable high performance minimal storage hybrid format Cholesky algorithm. ACM Transactions on Mathematical Software 31, 201–208 (2005)
2. Dobrian, F., Pothen, A.: A comparison between three external memory algorithms for factorising sparse matrices. In: Proceedings of the SIAM Conference on Applied Linear Algebra (2003)
3. Duff, I.S.: Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. SIAM J. Scientific and Statistical Computing 5, 270–280 (1984)
4. Duff, I.S.: MA57– a new code for the solution of sparse symmetric definite and indefinite systems. ACM Transactions on Mathematical Software 30, 118–144 (2004)
5. Guermouche, A., L'Excellent, J.-Y.: Optimal memory minimization algorithms for the multifrontal method. Technical Report RR5179, INRIA (2004)
6. HSL: A collection of Fortran codes for large-scale scientific computation (2004), See http://hsl.rl.ac.uk/
7. Karypis, G., Kumar, V.: METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - version 4.0 (1998), See http://www-users.cs.umn.edu/~karypis/metis/
8. Reid, J.K.: TREESOLV, a Fortran package for solving large sets of linear finite-element equations. Report CSS 155, AERE Harwell (1984)
9. Reid, J.K., Scott, J.A.: An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory (2006)
10. Rotkin, V., Toledo, S.: The design and implementation of a new out-of-core sparse Cholesky factorization method. ACM Transactions on Mathematical Software 30(1), 19–46 (2004)

Cholesky Factorization of Band Matrices Using Multithreaded BLAS[⋆]

Alfredo Remón, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí

Depto. de Ingeniería y Ciencia de Computadores,
Universidad Jaume I,
ES-12.071–Castellón, Spain
{remon, quintana, gquintan}@icc.uji.es

Abstract. In this paper we present two routines for the Cholesky factorization of band matrices that target (parallel) SMP architectures and expose coarser-grain parallelism than their LAPACK counterpart. Experimental results on two different parallel platforms show the benefits of the new approaches.

Keywords: Band matrices, Cholesky factorization, LAPACK, multithreaded BLAS, Symmetric Multiprocessor (SMP).

1 Introduction

Exploiting the structure of the coefficient matrix for band linear systems yields huge savings, both in number of computations and storage space. This is recognized in LAPACK [1], which includes a blocked routine for the Cholesky factorization of symmetric positive definite band (SPDB) matrices.

In this paper we analyze the parallel efficiency of the (double-precision) routine DPBTRF in LAPACK for the Cholesky factorization of SPDB matrices stored in packed symmetric format. As the parallelism in LAPACK is extracted by using multithreaded implementations of BLAS, the fragmentation (partitioning) of the operations in the LAPACK routine may reduce its parallel performance. Our approach to overcome this problem consists in *merging* operations so that parallelism with coarser-grain is exposed. In order to do so, our routine DPBTRF_A requires a simple modification of the storage scheme for band matrices so that a few rows of zeros are added to the bottom of the matrix. By doing some additional copies and manipulation of the matrix blocks, our routine DPBTRF_B does not require this workspace, at the expense of some performance loss.

The paper is structured as follows. The blocked routine in LAPACK for the factorization of SPDB matrices, xPBTRF, is reviewed in Section 2. Our new routines are then presented in Section 3. Experiments on two Intel® SMP architectures, based on Xeon™ and Itanium2™ processors, show the benefits of

⋆ This research was supported by the DAAD programme Acciones Integradas HA2005-0081, the CICYT project TIN2005-09037-C02-02 and FEDER, and project No. P1B-2004-6 of the *Fundación Caixa-Castelló/Bancaixa and UJI.*

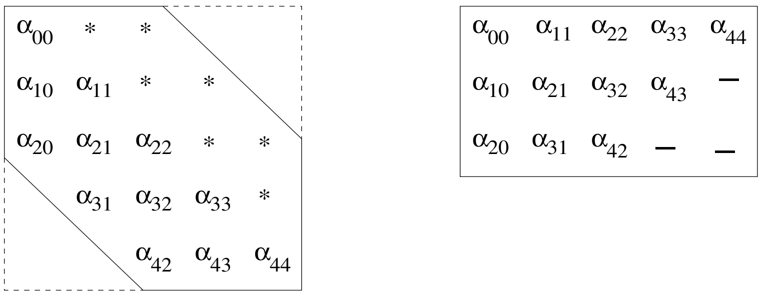


Fig. 1. Symmetric 5×5 band matrix with bandwidth $k = 2$ (left) and packed storage used in LAPACK (right). The '$*$' symbols denote the symmetric entries of the matrix.

merging operations in Section 4. Finally, some concluding remarks are given in Section 5.

2 LAPACK Blocked Routine for the Band Cholesky Factorization

Given a SPDB matrix $A \in \mathbb{R}^{n\times n}$, with bandwidth k, routine xPBTRF from LAPACK obtains a decomposition of this matrix into either

$$A = U^T U \quad \text{or} \quad A = LL^T,$$

where the Cholesky factors $U, L \in \mathbb{R}^{n\times n}$ are, respectively, upper and lower triangular with the same bandwidth as A. We only consider hereafter the latter decomposition, but the elaboration that follows is analogous for the upper triangular case.

The LAPACK routines employ a packed format to save storage: due to the symmetry of A, only its lower (or upper) triangular part is stored, following the pattern illustrated in Figure 1 (right). Upon completion of the factorization, the elements of L overwrite the corresponding entries of A in the packed matrix.

In order to describe routine xPBTRF, we will assume for simplicity that the algorithmic block size, b, is an exact multiple of k. Consider now the partitionings

$$A = \left(\begin{array}{c||c||c} A_{TL} & \star & \\ \hline\hline A_{ML} & A_{MM} & \star \\ \hline\hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c||c|c} A_{00} & \star & \star & & \\ \hline\hline A_{10} & A_{11} & \star & \star & \\ \hline A_{20} & A_{21} & A_{22} & \star & \star \\ \hline\hline & A_{31} & A_{32} & A_{33} & \star \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \qquad (1)$$

where $A_{TL}, A_{00} \in \mathbb{R}^{m\times m}$, $A_{11}, A_{33} \in \mathbb{R}^{b\times b}$, and $A_{22} \in \mathbb{R}^{k\times k}$, with $k = k - b$. (The "$\star$" symbol in A denotes the symmetric quadrant (block) of the matrix and will not be referenced.) With this partitioning, A_{31} is upper triangular. It is also important to realize that, due to the packed storage used in LAPACK, the entries that would be "logically" occupied by the strictly lower triangular part of A_{31}, "physically" correspond to entries (from the subdiagonals) of A_{11}.

Routine xPBTRF corresponds to what is usually known as a right-looking algorithm; that is, an algorithm where, at a certain iteration, A_{TL} has been already factorized, and A_{ML} and A_{MM} have been updated correspondingly [2]. In order to move forward in the computation by b rows/columns, during the current iteration of the routine the following operations are carried out (the annotations to the right of these operations correspond to the name of the LAPACK/BLAS routines that are used to perform them):

1. Compute the (dense) Cholesky factorization
$$A_{11} = L_{11}L_{11}^T. \qquad \text{(xPOTF2)} \qquad (2)$$
2. Solve the triangular linear systems
$$A_{21}(= L_{21}) := A_{21}L_{11}^{-T}, \qquad \text{(xTRSM)} \qquad (3)$$
$$A_{31}(= L_{31}) := A_{31}L_{11}^{-T}. \qquad \text{(xTRSM)} \qquad (4)$$
3. Compute the updates:
$$A_{22} := A_{22} - L_{21}L_{21}^T, \qquad \text{(xSYRK)} \qquad (5)$$
$$A_{32} := A_{32} - L_{31}L_{21}^T, \qquad \text{(xGEMM)} \qquad (6)$$
$$A_{33} := A_{33} - L_{31}L_{31}^T. \qquad \text{(xSYRK)} \qquad (7)$$

Due to A_{31}/L_{31} having only their upper triangular parts stored, the operations in (4), (6), and (7) in the actual implementation need special care, as described next. In order to solve the triangular linear system in (4), a copy of the upper triangular part of A_{31} is first obtained in an auxiliary workspace W of dimension $b \times b$, with its subdiagonal entries set to zero; the BLAS-3 solver xTRSM then yields $W(= L_{31}) := WL_{11}^{-T}$. The operation in (6) is computed as a general matrix product using BLAS-3 kernel xGEMM as $A_{32} := A_{32} - WL_{21}^T$. The update in (7) is obtained using BLAS-3 kernel xSYRK as $A_{33} := A_{33} - WW^T$ and the upper triangular part of W is written back to A_{31} once this is done.

LAPACK is coded in Fortran 77 and there is no dynamic allocation of memory. Instead of asking the user to provide the auxiliary space (as usual in many routines from LAPACK), the implementation of xPBTRF includes a local array of dimension $b_{\max} \times b_{\max}$ to store W. The actual value of $b_{\max}$ can be set during the installation of LAPACK and limits the values of b that can be used during the execution.

In our notation, after these operations are carried out, A_{TL} (the part that has been already factorized) grows by b rows/columns so that

$$A = \left(\begin{array}{c||c||c} A_{TL} & \star & \\ \hline\hline A_{ML} & A_{MM} & \star \\ \hline\hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c|c||c} A_{00} & \star & \star & & \\ \hline A_{10} & A_{11} & \star & \star & \\ \hline\hline A_{20} & A_{21} & A_{22} & \star & \star \\ \hline & A_{31} & A_{32} & A_{33} & \star \\ \hline\hline & & A_{42} & A_{43} & A_{44} \end{array}\right) \qquad (8)$$

in preparation for the next iteration.

From this particular implementation we observe that:

– Provided $b \ll k$, a major part of the floating-point arithmetic operations (flops) in (2)–(7) are performed in terms of the rank-k update in (5) (see also (1)) so that, by calling a tuned implementation of xSYRK, high performance is to be expected from the LAPACK blocked routine.
– No attempt is made to exploit the structure of W in the computations corresponding to (4), (6), and (7), as there is no BLAS kernel that fits such operations in case A_{31}/L_{31} are (upper) triangular.
– The additional work space W and the extra copies are only required in order to use kernels from current implementations of BLAS to perform operations (4), (6), and (7).

3 New Routines for the Band Cholesky Factorization

The exposition of the algorithm underlying routine xPBTRF in the previous section shows that, in order to advance the computation by b rows/columns, two invocations of routine xTRSM, two invocations of routine xSYRK, and an invocation of xGEMM are necessary for the updates in Steps 2 and 3. This is needed because of the special storage pattern used for band matrices in LAPACK. As all parallelism is extracted in LAPACK from calling multithreaded implementations of BLAS routines, the fragmentation of the computations that need to be performed in a single iteration of the algorithm into several small operations (and a large one) is potentially harmful for the efficiency of the codes.

In this section we describe how to merge the operations corresponding to Steps 2–3 so that higher performance is likely to be obtained on SMP architectures. Our first routine requires additional storage space in the data structure containing A so that b rows of zeros are present at the bottom of the matrix. By doing some extra copies and manipulation of the matrix blocks, the second routine does not require this workspace.

3.1 Routine xPBTRF_A

Consider the data structure containing A (see Figure 1) is padded with $b_{\max}$ ($\geq b$) rows at the bottom with all the entries in this additional space initially set to zero. (Interestingly, that corresponds, e.g., to the structure that would be necessary in the LAPACK storage scheme to keep a band matrix with square blocks of order $b_{\max}$ in the lower band. Block band matrices are frequently encountered in practice as well, but no specific support is provided for them in the current version of LAPACK.) Then, Steps 2–3 in xPBTRF are transformed in routine xPBTRF_A as follows:

2. Solve the (single) triangular linear system:

$$\begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix} \left(= \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \right) := \begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix} L_{11}^{-T}. \qquad \text{(xTRSM)} \qquad (9)$$

There is no longer need for workspace W nor copies to/from it as the additional rows at the bottom accommodate for the elements in the strictly lower triangle of A_{31}/L_{31}.

3. Compute the (single) update:

$$\begin{pmatrix} A_{22} & \star \\ A_{32} & A_{33} \end{pmatrix} := \begin{pmatrix} A_{22} & \star \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}^T . \qquad (\text{xSYRK}) \qquad (10)$$

3.2 Routine xPBTRF_B

The previous approach, though efficient in the sense of grouping as many computations as possible in coarse-grain blocks, requires a non-negligible workspace. The following routine, xPBTRF_B, also groups the computations at the expense of two additional copies. The idea behind it is simple: The problem lies in the operations involving A_{31} and L_{31} as only the upper triangular of these blocks is actually stored, but the operations require them to be full square blocks. Operating without taking this into account would therefore procure an erroneous decomposition. Now, consider the symmetric packed storage used by the LAPACK routines. We can operate on A_{31} and L_{31} as if they were full square blocks if we preserve the data that physically lies in their strictly lower triangular parts (and which corresponds to the subdiagonals of A_{11}). Therefore, we propose the following modification of Steps 2–3 in routine xPBTRF_B, which also aims at clustering blocks but does not require storage for b rows:

2. Obtain $W := \text{STRIL}(A_{31})$, a copy of the space that would be physically occupied by the strictly lower triangular part of A_{31} and set $\text{STRIL}(A_{31}) := 0$. (Due to the storage format, after these operations $W = \text{STRIL}(A_{11})$ and $\text{STRIL}(A_{11}) = 0$.) Then, solve the triangular linear system:

$$\begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix} \left(= \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \right) := \begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix} L_{11}^{-T}. \qquad (\text{xTRSM}) \qquad (11)$$

   Now, a copy of the physical storage that would be overwritten in this step is kept in W.

3. Compute the (single) update:

$$\begin{pmatrix} A_{22} & \star \\ A_{32} & A_{33} \end{pmatrix} := \begin{pmatrix} A_{22} & \star \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}^T , \qquad (\text{xSYRK}) \qquad (12)$$

   and restore $\text{STRIL}(A_{31})$ ($= \text{STRIL}(A_{11})$) from W.

Following LAPACK xPBTRF approach, the workspace W is a local array of routine xPBTRF_B and is not provided by the user.

4 Experimental Results

All experiments in this section were performed using IEEE double-precision (real) arithmetic and band matrices of order n=10,000, with bandwidth size ranging from $k = 1$ to 2,000. Provided $n \gg k$, the performance of the routines is only determined by k and the block size b. In the evaluation, for each bandwidth dimension, we employed values from 1 to 200 to determine the optimal block size, b_{opt}; only those results corresponding to b_{opt} are shown.

We report the performance of the routines on two different SMP architectures, with 2 and 4 processors; see Table 1. Two threads were employed on XEON and 4 threads on ITANIUM. As the efficiency of the kernels in BLAS is crucial, for each platform we use the implementations listed in Table 2.

Table 1. SMP architectures employed in the evaluation

Platform	Architecture	#Proc.	Frequency (GHz)	L2 cache (KBytes)	L3 cache (MBytes)	RAM (GBytes)
XEON	Intel Xeon	2	2.4	512	–	1
ITANIUM	Intel Itanium2	4	1.5	256	4	4

Table 2. Software employed in the evaluation

Platform	BLAS	Compiler	Optimization Flags	Operating System
XEON	GotoBLAS 1.00 MKL 8.0	`gcc` 3.3.5	`-O3`	Linux 2.4.27
ITANIUM	GotoBLAS 0.95mt MKL 8.0	`icc` 9.0	`-O3`	Linux 2.4.21

Figure 2 reports the results from a detailed evaluation which shows how the time is distributed among the different operations within LAPACK routine DPBTRF: (factorization of) A_{11}, (update of) A_{21}, etc. Two main observations can be drawn from this experiment: The factorization of A_{11} is a major performance bottleneck for the routine when Goto BLAS is employed. A closer inspection revealed this to be due to the low performance of the multithreaded implementation of DGEMV, invoked from DPOTF2 during the factorization of A_{11}, when the size of this block ($b \times b$) is small. A similar inspection concluded that MKL employs a single thread in such situation and therefore avoids this bottleneck. A second source of performance degradation is the update of A_{31}, both for MKL and Goto BLAS. Although in theory the time required to update this block should be negligible, the experiments show that this is not the case, especially on XEON.

Figure 3 illustrates the MFLOPs (millions of flops per second) of the original codes DPBTRF in the lines labeled as `Goto BLAS` and `MKL BLAS`. In order to overcome the problem with Goto BLAS, we factorize A_{11} using routine DPOTF2 with the code in kernel DGEMV directly inlined. This ensures that a single thread is used during this operation and obtains an important performance enhancement, as shown in the top two plots of Figure 3 by the lines labeled as `Goto BLAS_inline`.

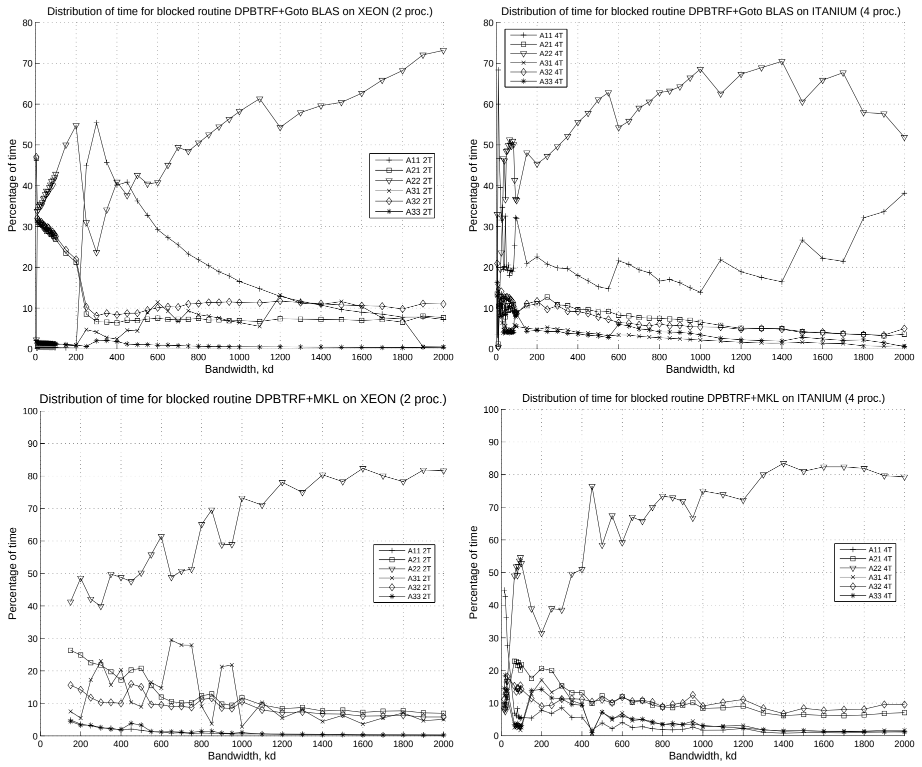


Fig. 2. Distribution of time among the different operations involved in routine DPBTRF on XEON (left) and ITANIUM (right) using multithreaded Goto BLAS (top) and MKL BLAS (bottom)

The two new routines that we propose, DPBTRF_A and DPBTRF_B, merge the update of A_{31} with that of A_{21}, and the updates of A_{32} and A_{33} with that of A_{22}. This allows to combine the operations of the routine so that only single invocations of DTRSM and DSYRK are required per iteration to operate on the subdiagonal blocks of the band matrix. On the other hand, routine DPBTRF_B requires some extra copies (overhead), which are not present in routine DPBTRF_A, so that we can expect the performance of the former to be slightly lower. The actual improvement that is obtained in the performance is illustrated by the lines labeled as `Goto BLAS_A_inline`, `Goto BLAS_B_inline`, `MKL BLAS_A`, and `MKL BLAS_B` in Figure 3. Let us comment each one of the cases:

– `Goto` BLAS on XEON: Routines `Goto BLAS_A_inline` and `Goto BLAS_B_inline` offer similar performances which are significantly higher than that of the LAPACK routine with inline (`Goto BLAS_inline`). Starting from k=1,100, routine `Goto BLAS_B_inline` slightly outperforms `Goto BLAS_A_inline`.

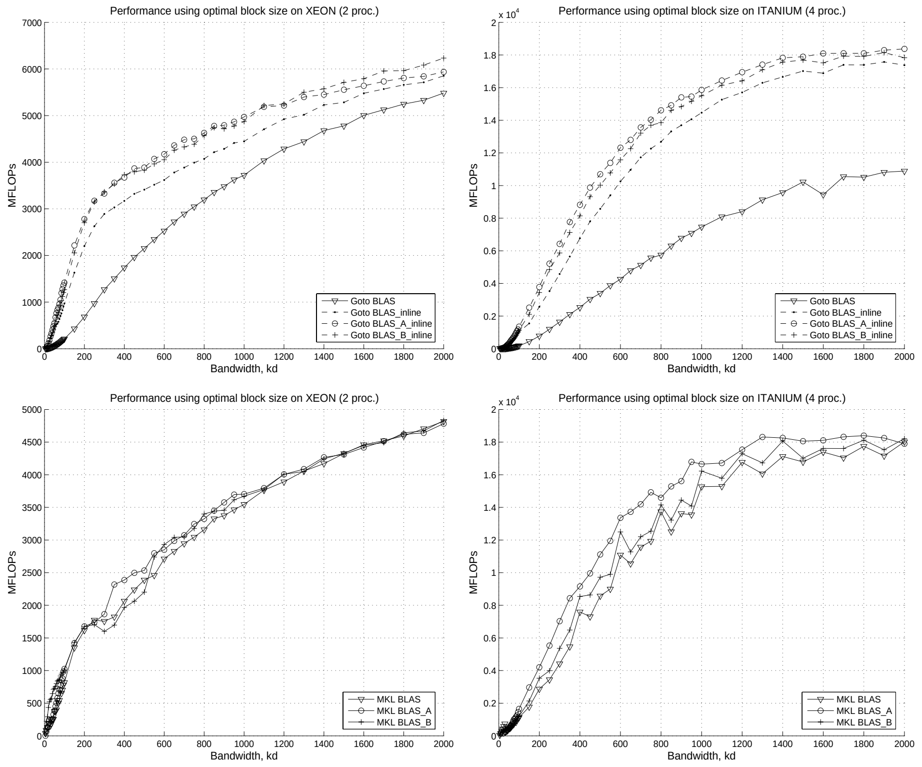


Fig. 3. Performance of routine DPBTRF on XEON (left) and ITANIUM (right) using multithreaded Goto BLAS (top) and MKL (bottom)

- `Goto BLAS` on ITANIUM: The efficiency of routine `Goto BLAS_A_inline` is moderately higher than that of `Goto BLAS_B_inline` and both are consistently higher than that of the LAPACK routine with inline.
- `MKL BLAS` on XEON: The MFLOPs rate attained with this implementation of BLAS is considerably lower than that reported for Goto BLAS on the same architecture. Routine `MKL BLAS_A` offers higher rates for matrices with moderate bandwidth. For the larger bandwidth sizes, all three routines deliver similar performance.
- `MKL BLAS` on ITANIUM: The asymptotic performance is close to that obtained with Goto BLAS. `MKL BLAS_A` yields higher performance than the LAPACK routine (`MKL BLAS`). The efficiency of `MKL BLAS_B` lies between those of the other two routines.

5 Conclusions

We have presented two routines for the Cholesky factorization of a band matrix that reduce the number of calls to BLAS per iteration so that coarser-grain parallelism is exposed. The routines are thus potentially better suited to exploit the architecture of SMP systems.

Several other (minor) conclusions have been extracted from our experience with band codes:

– BLAS does not support all the functionality that is needed for the Cholesky factorization of band matrices. However, even if these kernels were available, exploiting the structure of A_{31} splits the computations in several parts and usually results in lower performance when multiple processors are used.
– The performance of BLAS-1 and BLAS-2 is much more important than in general for other dense routines and the optimal block size for the blocked routines needs careful tuning.
– Operations that are considered minor in current LAPACK routines will need to be reconsidered for future multicore architectures and some LAPACK routines may even need to be recoded.

Compared with the message-passing programming paradigm in ScaLAPACK, a parallel implementation based on LAPACK+multithreaded BLAS presents a much better ease of use. Besides, this second approach offers programmers the opportunity of improving existing sequential codes evolving the codes to be more scalable.

Acknowledgments

We thank K. Goto for his help with the Goto BLAS implementations.

References

1. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
2. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical linear Algebra for high-performance computers. SIAM, Philadelphia, PA (1998)

A Master-Worker Type Eigensolver for Molecular Orbital Computations

Tetsuya Sakurai[1,3], Yoshihisa Kodaki[2], Hiroto Tadano[3], Hiroaki Umeda[3,4], Yuichi Inadomi[5], Toshio Watanabe[3,4], and Umpei Nagashima[3,4]

[1] Department of Computer Science, University of Tsukuba, Tsukuba 305-8573, Japan
sakurai@cs.tsukuba.ac.jp
[2] Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba 305-8573, Japan
[3] Core Research for Evolutional Science and Technology, Japan Science and Technology Agency, Japan
[4] Computational Science Research Division, National Institute of Advanced Industrial Science and Technology
[5] Computing and Communications Center, Kyushu University, Fukuoka 812-8581, Japan

Abstract. We consider a parallel method for solving generalized eigenvalue problems that arise from molecular orbital computations. We use a moment-based method that finds several eigenvalues and their corresponding eigenvectors in a given domain, which is suitable for master-worker type parallel programming models. The computation of eigenvalues using explicit moments is sometimes numerically unstable. We show that a Rayleigh-Ritz procedure can be used to avoid the use of explicit moments. As a test problem, we use the matrices that arise in the calculation of molecular orbitals. We report the performance of the application of the proposed method with several PC clusters connected through a hybrid MPI and GridRPC system.

1 Introduction

Generalized eigenvalue problems arise in many scientific and engineering applications. Several methods for such eigenvalue problems are building sequences of subspaces that contain the desired eigenvectors. Krylov subspace based techniques are powerful tools for large-scale eigenvalue problems [1,2,8,9]. The relations among Krylov subspace methods, moment-matching approach and Padé approximation are shown in [2].

In this paper we consider a parallel method for finding several eigenvalues and eigenvectors of generalized eigenvalue problems in a grid computing environment. A master-worker type algorithm is efficient to obtain a good performance with distributed computing resources. However, it is not easy to adjust eigensolvers for such type algorithms because of iterative processes of solvers.

We use a method using a contour integral proposed in [11] to find eigenvalues that lie inside a given domain. In this method, a small matrix pencil that has only the desired eigenvalues is derived by solving systems of linear equations. Since these linear systems can be solved independently, we can obtain a master-worker type parallel algorithm. In [12], a parallel implementation of the method using a GridRPC system is presented.

In the method [11], complex moments obtained by a contour integral are used. The explicit use of moments sometimes causes numerical instability. To improve the numerical stability, we apply a Rayleigh-Ritz procedure with a subspace obtained by a contour integral for some vectors.

As a test problem, we used the matrices that arise in the calculation of molecular orbitals obtained from the fragment molecular orbital (FMO) method [3]. The FMO method has been proposed as a method for calculating the electronic structure of large molecules such as proteins. The molecular orbitals of the FMO method are obtained by solving the generalized eigenvalue problem for the Fock matrix calculated with the total density matrix of the FMO method. In this eigenvalue problem, we need several number of eigenvectors related to chemical activities.

Since the target molecule is divided into small fragments, the process to generate matrices for molecular orbitals using FMO has a good performance with distributed PC clusters. If we can use an eigensolver which also has a good parallel performance in a grid environment, whole processes can be done in a same computing environment. We report the performance of the application of the proposed method with several PC clusters connected through a hybrid MPI and GridRPC system.

2 A Master-Worker Type Parallel Method

In this section, we briefly describe a master-worker type method for generalized eigenvalue problems presented in [11,12]. This method, which is based on a moment-based root finding method [4], finds several eigenvalues that are located inside given circles. The error analysis of the method [4] is considered in [5,10].

Let $A, B \in \mathbb{C}^{n\times n}$, and let $\lambda_1, \lambda_2, \dots, \lambda_n$ be eigenvalues of the matrix pencil (A, B). For nonzero vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{C}^n$, we define

$$f(z) := \boldsymbol{u}^{\mathrm{H}}(zB - A)^{-1}B\boldsymbol{v}$$

with a complex parameter z. Let $\boldsymbol{p}_j$ and $\boldsymbol{q}_j$ be left and right eigenvectors corresponding to an eigenvalue λ_j, respectively. Then $f(z)$ can be expressed as

$$f(z) = \sum_{j=1}^{n} \frac{\nu_j}{z-\lambda_j} + g(z),$$

where $\nu_j = (\boldsymbol{u}^{\mathrm{H}}\boldsymbol{q}_j)(\boldsymbol{p}_j^{\mathrm{H}}B\boldsymbol{v})$, $g(z)$ is a polynomial of degree K, and K is a maximum size of Jordan blocks of B ([11]).

Let Γ be a circle with radius ρ centered at γ. Suppose that m distinct eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_m$ are located inside Γ. Define the complex moments

$$\mu_k := \frac{1}{2\pi\mathrm{i}} \int_{\Gamma} (z-\gamma)^k f(z)\,\mathrm{d}z, \quad k = 0, 1, \dots. \tag{1}$$

By approximating the integral of (1) via the N-point trapezoidal rule, we obtain the following approximations:

$$\mu_k \approx \hat{\mu}_k := \frac{1}{N}\sum_{j=0}^{N-1}(\omega_j - \gamma)^{k+1} f(\omega_j), \quad k = 0, 1, \ldots, \tag{2}$$

where

$$\omega_j := \gamma + \rho\, \mathrm{e}^{\frac{2\pi \mathrm{i}}{N}\left(j+\frac{1}{2}\right)}, \quad j = 0, 1, \ldots, N-1.$$

Let the $m \times m$ Hankel matrices $\hat{H}_m$ and $\hat{H}_m^<$ be

$$\hat{H}_m := [\hat{\mu}_{i+j-2}]_{i,j=1}^m \quad \text{and} \quad \hat{H}_m^< := [\hat{\mu}_{i+j-1}]_{i,j=1}^m.$$

The approximate eigenvalues $\hat{\lambda}_1, \hat{\lambda}_2, \ldots, \hat{\lambda}_m$ are obtained from the pencil $(\hat{H}_m^<, \hat{H}_m)$.

Let $\hat{\boldsymbol{w}}_j$ be an eigenvector of the pencil $(\hat{H}_m^<, \hat{H}_m)$. The approximations for right eigenvectors are obtained by

$$\hat{\boldsymbol{q}}_j = [\hat{\boldsymbol{s}}_0, \hat{\boldsymbol{s}}_1, \ldots, \hat{\boldsymbol{s}}_{m-1}]\hat{\boldsymbol{w}}_j, \quad j = 1, 2, \ldots, m, \tag{3}$$

where

$$\hat{\boldsymbol{s}}_k := \frac{1}{N}\sum_{j=0}^{N-1}(\omega_j - \gamma)^{k+1}\boldsymbol{y}_j, \quad k = 0, 1, \ldots, m-1, \tag{4}$$

and

$$\boldsymbol{y}_j = (\omega_j B - A)^{-1}\boldsymbol{v}, \quad j = 0, 1, \ldots, N-1. \tag{5}$$

The approximate residues are evaluated by

$$\hat{\nu}_j = (\hat{\mu}_0, \hat{\mu}_1, \ldots, \hat{\mu}_{m-1})\hat{\boldsymbol{w}}_j, \quad j = 1, 2, \ldots, m.$$

The accuracy of the results is sometimes improved by computing $M(> m)$ eigenvalues instead of m. In this case, we remove $\hat{\lambda}_j$ when it is located outside of Γ, or corresponding residue $|\hat{\nu}_j|$ is very small.

In order to evaluate the value of $f(z)$ at $z = \omega_j$, $j = 0, 1, \ldots, N-1$, we solve N linear systems (5) for $\boldsymbol{y}_j$, $j = 0, 1, \ldots, N-1$. When matrices A and B are large, the computational costs for solving the linear systems are dominant in the method. Since these linear systems are independent for each j, they are solved on remote servers in parallel.

This method can be extended for the case that several circular regions are given simultaneously. Suppose that N_c circles are given. Then we solve $N \times N_\mathrm{c}$ systems of linear equations

$$(\omega_j^{(k)} B - A)\boldsymbol{y}_j^{(k)} = \boldsymbol{v}, \quad j = 0, 1, \ldots, N-1, \quad k = 1, 2, \ldots, N_\mathrm{c},$$

where $\omega_j^{(k)}$, $j = 0, 1, \ldots, N-1$ are equi-distributed points on the kth circle with the center $\gamma^{(k)}$ and the radius $\rho^{(k)}$.

Since A and B are common in each system of linear equations, we initially send these data to each server. In order to solve another equation on the same server, a scalar parameter ω_j is sent. In this approach, we do not need to exchange data between remote servers. Therefore, the present method is suitable for master-worker programming models. A hybrid implementation by using a grid RPC system Ninf-G [7] and MPI is presented in [13].

3 Modification by the Rayleigh-Ritz Procedure

In this section, we show a method for improving the accuracy of the eigenpairs using a Rayleigh-Ritz procedure.

Let $A \in \mathbb{R}^{n\times n}$ be symmetric, and let $B \in \mathbb{R}^{n\times n}$ be symmetric positive definite. Let $Z \in \mathbb{R}^{n\times M}$ be an orthonormal basis of a certain subspace. The Rayleigh-Ritz procedure is shown below:

Rayleigh-Ritz Procedure

1. Construct an orthonormal basis Z.
2. Form $\tilde{A} = Z^{\mathrm{T}}AZ$ and $\tilde{B} = Z^{\mathrm{T}}BZ$.
3. Compute the eigenpairs $(\theta_j, \boldsymbol{w}_j)$ $(1 \le j \le M)$ of $(\tilde{A}, \tilde{B})$.
4. Set $\boldsymbol{x}_j = Z\boldsymbol{w}_j, j = 1, 2, \ldots, M$.

If the subspace Z includes the subspace spanned by the m eigenvectors $\boldsymbol{q}_1$, $\boldsymbol{q}_2, \ldots, \boldsymbol{q}_m$, then some of the Ritz values θ_j of the projected pencil $(\tilde{A}, \tilde{B})$ are taken as approximate eigenvalues for the original pencil (A, B) with corresponding Ritz vectors $\boldsymbol{x}_j$. We can derive various methods by a choice of Z.

Now we show a method to provide a subspace using a contour integral. For a nonzero vector $\boldsymbol{v} \in \mathbb{R}^n$, we define

$$\boldsymbol{s}_k := \frac{1}{2\pi \mathrm{i}} \int_{\Gamma} (z-\gamma)^k (zB - A)^{-1} B\boldsymbol{v}\, \mathrm{d}z, \quad k = 0, 1, \ldots, M-1.$$

When $M \ge m$ and Z is the orthonormal basis of the space spanned by $\{\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{M-1}\}$, then m Ritz values are $\lambda_1, \lambda_2, \ldots, \lambda_m$ ([14]). This implies that the eigenvalues located inside Γ can be obtained with vectors obtained by the contour integral.

By approximating the contour integral via the N-point trapezoidal rule, we obtain the following approximations for $\boldsymbol{s}_k$:

$$\boldsymbol{s}_k \approx \hat{\boldsymbol{s}}_k := \frac{1}{N} \sum_{j=0}^{N-1} (\omega_j - \gamma)^{k+1} (\omega_j B - A)^{-1} B\boldsymbol{v}, \quad k = 0, 1, \ldots, M-1,$$

where N is a positive integer. Since $\hat{\boldsymbol{s}}_k$ suffer from the quadrature error which arises from eigenvalues located outside the circle, we take the size of the subspace larger than the exact number of the eigenvalues inside the circle. Thus we set, in practice,

$$Z \in \mathrm{span}(\hat{\boldsymbol{s}}_0, \hat{\boldsymbol{s}}_1, \ldots, \hat{\boldsymbol{s}}_{M-1}),$$

with $M(> m)$, and this approach is efficient to decrease the influence of the quadrature error similar to the method in the previous section.

By using the Rayleigh-Ritz procedure, we can avoid the explicit use of the moments in the computation of the approximate eigenvalues and eigenvectors.

4 Numerical Examples

In the first example, we compare the numerical accuracy of the method using explicit moments and the modified method with Rayleigh-Ritz procedure in case that the eigenvalues are given analytically. The matrices were

$$A = I_n, \quad B = \begin{pmatrix} 5 & -4 & 1 & & & & \\ -4 & 6 & -4 & 1 & & & \\ 1 & -4 & 6 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 6 & -4 & 1 \\ & & & 1 & -4 & 6 & -4 \\ & & & & 1 & -4 & 5 \end{pmatrix},$$

where I_n is the $n \times n$ identity matrix, and $n = 1,000,000$. The exact eigenvalues are given by

$$\lambda_j^* = \frac{1}{16 \cos^4 \left(\dfrac{j\pi}{2(n+1)} \right)}, \quad j = 1, 2, \ldots, n.$$

Computation was performed with double precision arithmetic in MATLAB. The systems of linear equations were solved by a direct solver of MATLAB function. The parameter were $N = 32$ and $M = 24$, and the interval was $[2.0000, 2.0002]$. The results are given in Table 1 and 2. The relative error for $\hat{\lambda}_i$ was given by

$$\min_{1 \leq j \leq n} \frac{|\hat{\lambda}_i - \lambda_j^*|}{|\lambda_j^*|},$$

and the residual was given by $\|A\hat{\boldsymbol{q}}_j - \hat{\lambda}_i B\hat{\boldsymbol{q}}_j\|_2$. In the case of explicit moment, only 6 eigenvalues were found, while 7 eigenvalues were included in the circle. We can see that the error was improved by using the Rayleigh-Ritz procedure.

The second example was derived from the molecular orbital computation of a model DNA [3]. The size of the matrices was $n = 1,980$. The parameters were $N = 32$ and $M = 24$. We put four circles on the interval $[-0.200, -0.175]$, and 30 eigenvalues were found. The relative error was evaluated by comparing with the results obtained by a MATLAB function 'eig'. The numerical results are given in Table 3.

Next, we show the wall-clock time using a GridRPC system. In order to evaluate the performance of the method for the situation in which several PC clusters are employed via a wide area network, we regarded some parts of a large-size PC cluster system as distributed PC clusters.

Table 1. Numerical results of the method using Rayleigh-Ritz procedure

Approximation	Relative Error	Residual
2.0000159105	4.4×10^{-16}	6.7×10^{-13}
2.0000430289	2.2×10^{-16}	8.7×10^{-13}
2.0000701478	1.8×10^{-15}	5.2×10^{-11}
2.0000972671	1.1×10^{-15}	8.6×10^{-13}
2.0001243870	8.9×10^{-16}	1.1×10^{-11}
2.0001515073	2.2×10^{-16}	5.8×10^{-13}
2.0001786281	2.2×10^{-16}	5.7×10^{-13}

Table 2. Numerical results of the method using explicit moments

Approximation	Relative Error	Residual
2.0000159085	1.0×10^{-9}	5.3×10^{-8}
2.0000430351	3.1×10^{-9}	1.4×10^{-7}
2.0000972715	2.2×10^{-9}	2.4×10^{-7}
2.0001243930	3.0×10^{-9}	4.8×10^{-7}
2.0001515063	4.9×10^{-10}	1.4×10^{-6}
2.0001786283	1.3×10^{-10}	1.9×10^{-7}

Experiments were performed on the AIST F32 Super Cluster of the National Institute of Advanced Industrial Science and Technology. The node of the cluster system was a 3.06-GHz Xeon with 4 GB of RAM. The client machine was a 3.0-GHz Pentium 4 with 2 GB of RAM. The client was connected to servers via 100Base-TX Ethernet. The program was implemented with a GridRPC system, Ninf-G ver. 2.4.0 [7].

The procedure to solve N systems was performed on remote servers. Since A, B and $\boldsymbol{v}$ are common in each system of linear equations, we only need to send these data at the first time to each server. In the hybrid algorithm [13], matrix data are sent from a client to a remote server when a Ninf-G process is started. Each Ninf-G process employs a specified number of MPI processes, and matrix data are delivered to all processes. By this approach, we can reduce the number of data transfer from a client to servers via a wide area network compared with the case in which all remote servers receive matrix data directly from a client. To solve another equation on a remote server, a scalar parameter ω_j is sent.

The test matrices were derived from computation of the molecular orbitals of lysozyme (129 amino-acid residues, 1,961 atoms) with 20,758 basis functions [3]. The structure of the lysozyme molecule has been determined experimentally, and we added counter-ions and water molecules around the lysozyme molecule in order to simulate in vivo conditions. The size of both A and B was $n = 20,758$, and the number of nonzero elements was $10,010,416$. A was symmetric, and B was symmetric positive definite.

The original matrices were given as dense matrices. To decrease the size of data to transfer from a client to servers, we dropped elements of A and B when $|a_{ij}| + |b_{ij}| \leq \delta \times \max_{i,j}(|a_{ij}| + |b_{ij}|)$ with small number δ. In Table 4, we show

Table 3. Error and residual for model DNA

Interval	Approximate eigenvalues	Relative Error	Residual
$[-0.175, -0.164]$	-0.1653821623	2.7×10^{-15}	9.1×10^{-14}
	-0.1660179534	1.6×10^{-14}	4.4×10^{-11}
	-0.1665276422	8.2×10^{-15}	2.7×10^{-12}
	-0.1671209292	1.9×10^{-14}	2.5×10^{-11}
	-0.1701971456	5.4×10^{-15}	1.0×10^{-12}
	-0.1713425781	9.6×10^{-15}	4.2×10^{-12}
	-0.1720255816	5.5×10^{-15}	5.6×10^{-13}
	-0.1734650856	6.9×10^{-15}	1.6×10^{-13}
$[-0.188, -0.176]$	-0.1800584518	6.2×10^{-16}	2.1×10^{-12}
	-0.1804114492	7.7×10^{-16}	1.5×10^{-12}
	-0.1836742198	3.9×10^{-15}	4.9×10^{-12}
	-0.1849530396	1.4×10^{-15}	3.5×10^{-11}
	-0.1855093053	9.3×10^{-15}	2.0×10^{-9}
	-0.1855909662	5.1×10^{-15}	2.4×10^{-10}
$[-0.199, -0.190]$	-0.1924234809	3.1×10^{-14}	3.8×10^{-11}
	-0.1930532724	2.3×10^{-15}	2.2×10^{-11}
	-0.1938675180	8.6×10^{-16}	4.1×10^{-11}
	-0.1947494181	1.3×10^{-15}	5.1×10^{-11}
	-0.1957682188	2.6×10^{-15}	4.2×10^{-10}
	-0.1960484797	6.7×10^{-15}	1.8×10^{-9}
	-0.1967218965	9.6×10^{-15}	9.7×10^{-10}
	-0.1970427773	3.2×10^{-15}	6.4×10^{-11}
$[-0.207, -0.200]$	-0.2005919579	3.7×10^{-15}	3.4×10^{-13}
	-0.2008357178	4.4×10^{-15}	5.2×10^{-13}
	-0.2036556294	6.7×10^{-15}	3.9×10^{-10}
	-0.2038003108	7.8×10^{-15}	2.4×10^{-10}
	-0.2039320476	2.4×10^{-14}	2.9×10^{-10}
	-0.2048059882	5.4×10^{-15}	1.1×10^{-9}
	-0.2048242514	4.5×10^{-15}	8.1×10^{-10}
	-0.2050547248	2.6×10^{-14}	1.8×10^{-10}

Table 4. Ratio of nonzero elements (%)

δ	10^{-16}	10^{-14}	10^{-12}	10^{-10}	10^{-8}	10^{-6}
A	16.22	11.50	7.18	3.65	1.34	0.42
B	2.19	1.74	1.33	0.94	0.49	0.29
$\lvert A\rvert + \lvert B\rvert$	16.22	11.51	7.18	3.65	1.35	0.44

the ratio of nonzero elements with drop tolerance δ for A, B and $|A| + |B|$. In the numerical experiments below, we take $\delta = 10^{-9}$ by some experiments, and A and B are treated as sparse matrices.

Since the matrix $\omega_j B - A$ with complex ω_j is complex symmetric, the COCG method [15] with incomplete Cholesky factorization with a complex shift [6] was used. In this case, a complex shift was effective to decrease the number of iterations. We used the same precondition matrix in each process of Ninf-G to

Table 5. Wall-clock times (seconds)

Number of MPI	Number of Ninf-G processes				
processes	1	2	4	6	8
1	1351	670	351	265	240
2	901	447	238	182	159
4	529	265	142	117	82
8	348	173	86	84	50

save the computational costs. The stopping criterion for the relative residual was 10^{-12}. Computation was performed in double-precision arithmetic. The elements of $\boldsymbol{v}$ were distributed randomly on the interval $[0, 1]$ by a random number generator.

The intervals $[-0.24, -0.18]$ and $[0.16, 0.22]$ were covered by eight circles. These intervals include the energy levels of the highest occupied molecular orbitals (HOMO) and the lowest unoccupied molecular orbitals (LUMO), which are key factors in the amount of energy needed to add or remove electrons in a molecule. The parameters were chosen as $N = 48$ and $M = 20$.

We observed the wall-clock time in seconds with various combinations of the numbers of Ninf-G processes and MPI processes. The time required to load the input matrices into the memory of the client and that required to start up the Ninf-G and MPI processes were not included. The results are listed in Table 5. Twenty eigenvalues and corresponding eigenvectors were obtained in 50 seconds with the combination of eight Ninf-G processes and eight MPI processes (64 processors).

In these computations, the preconditioning matrix was computed in serial code, and was computed once in each Ninf-G process. This causes the load imbalance between MPI processes. The use of a parallel preconditioner will improve the performance. The application to larger size of matrices derived from practical problems will be our future work.

Acknowledgements

This work was supported in part by CREST of the Japan Science and Technology Agency (JST).

References

1. Arnoldi, W.E.: The principle of minimized iteration in the solution of the matrix eigenproblem. Quarterly of Appl. Math. 9, 17–29 (1951)
2. Bai, Z.: Krylov subspace techniques for reduced-order modeling of large-scale dynamical systems. Appl. Numer. Math. 43, 9–44 (2002)
3. Inadomi, Y., Nakano, T., Kitaura, K., Nagashima, U.: Definition of molecular orbitals in fragment molecular orbital method. Chem. Phys. Letters 364, 139–143 (2002)

4. Kravanja, P., Sakurai, T., Van Barel, M.: On locating clusters of zeros of analytic functions. BIT 39, 646–682 (1999)
5. Kravanja, P., Sakurai, T., Sugiura, H., Van Barel, M.: A perturbation result for generalized eigenvalue problems and its application to error estimation in a quadrature method for computing zeros of analytic functions. J. Comput. Appl. Math. 161, 339–347 (2003)
6. Magolu, M.M.M.: Incomplete factorization-based preconditionings for solving the Helmholtz equation. Int. J. Numer. Meth. Engng. 50, 1088–1101 (2001)
7. Ninf-G: `http://ninf.apgrid.org/`
8. Ruhe, A.: Rational Krylov algorithms for nonsymmetric eigenvalue problems II: matrix pairs. Lin. Alg. Appl. 197, 283–295 (1984)
9. Saad, Y.: Iterative Methods for Large Eigenvalue Problems. Manchester University Press, Manchester (1992)
10. Sakurai, T., Kravanja, P., Sugiura, H., Van Barel, M.: An error analysis of two related quadrature methods for computing zeros of analytic functions. J. Comput. Appl. Math. 152, 467–480 (2003)
11. Sakurai, T., Sugiura, H.: A projection method for generalized eigenvalue problems. J. Comput. Appl. Math. 159, 119–128 (2003)
12. Sakurai, T., Hayakawa, K., Sato, M., Takahashi, D.: A parallel method for large sparse generalized eigenvalue problems by OmniRPC in a grid environment. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 1151–1158. Springer, Heidelberg (2006)
13. Sakurai, T., Kodaki, Y., Umeda, H., Inadomi, Y., Watanabe, T., Nagashima, U.: A hybrid parallel method for large sparse eigenvalue problems on a grid computing environment using Ninf-G/MPI. In: Lirkov, I., Margenov, S., Waśniewski, J. (eds.) LSSC 2005. LNCS, vol. 3743, pp. 438–445. Springer, Heidelberg (2006)
14. Sakurai, T.: A Rayleigh-Ritz type method for large-scale generalized eigenvalue problems. In: Abstract of 1st China-Japan-Korea Joint Conference on Computational Mathematics, Sapporo, pp. 50–52 (2006)
15. van der Vorst, H.A., Melissen, J.B.M.: A Petrov-Galerkin type method for solving $Ax = b$, where A is a symmetric complex matrix. IEEE Trans. on Magnetics 26, 706–708 (1990)

Bioinformatics and Computational Biology: Minisymposium Abstract

Ann-Charlotte Berglund Sonnhammer and Sverker Holmgren

Uppsala University, Sweden

The fast development of new experimental biotechnologies has resulted in an avalanche of genomic data. As a consequence, the life sciences are becoming more quantitative, and information technology, mathematics and statistics are being incorporated in a natural way. New fields such as bioinformatics and computational biology have emerged and are now key players in the effort to unravel the information hidden in the genomes. With the vast and ever growing amounts of data, it has become necessary to employ parallel programming, and grid techniques to accelerate the data processing and analysis. Although the bioinformatics and computational biology fields are still in their infancy when it comes to using parallel computers and grids, at least compared to other fields such as computational chemistry and physics, they have already emerged as major users of high-throughput computing resources.

Using Parallel Computing and Grid Systems for Genetic Mapping of Quantitative Traits

Mahen Jayawardena[1,2], Kajsa Ljungberg[1], and Sverker Holmgren[1]

[1] Division of Scientific Computing, Department of Information Technology,
Uppsala University, Box 337, SE-751 05 Uppsala, Sweden
{kajsa.ljungberg, sverker.holmgren}@it.uu.se
[2] University of Colombo School of Computing, Colombo, Sri Lanka
mahen@cmb.ac.lk

Abstract. We present a flexible parallel implementation of the exhaustive grid search algorithm for multidimensional QTL mapping problems. A generic, parallel algorithm is presented and a two-level scheme is introduced for partitioning the work corresponding to the independent computational tasks in the algorithm. At the outer level, a static block-cyclic partitioning is used, and at the inner level a dynamic pool-of-tasks model is used. The implementation of the parallelism at the outer level is performed using scripts, while MPI is used at the inner level. By comparing to results from the SweGrid system to those obtained using a shared memory server, we show that this type of application is highly suitable for execution in a grid framework.

Keywords: QTL analysis, grid computing.

1 Genetic Mapping of Quantitative Traits

Many important traits in animals and plants are quantitative in nature. Examples include body weight and growth rate, susceptibility to infections and other diseases, and agricultural crop yield. Hence, understanding the genetic factors behind quantitative traits is of great importance. The regions in the genome affecting a quantitative trait can be found by analysis of the genetic composition of individuals in experimental populations. The genetic regions are also known as Quantitative Trait Loci (QTL), and the procedure of finding these is called QTL mapping. A review of QTL mapping methods is given in [13].

In QTL mapping, a statistical model for how the genotypes of the individuals in the population affect the trait is exploited. The data for the model is produced by experiments where the genotypes are determined at a set of marker loci in the genome. This data is input to a QTL mapping computer code, where the computation of the model fit and the search for the most probable positions of the QTL are implemented using numerical algorithms. Once the most probable QTL is determined, further computations are needed to establish the statistical significance of the result.

It is generally believed that quantitative traits are governed by an interplay between multiple QTL and environmental factors. However, using a model where multiple QTL are searched for simultaneously makes the statistical analysis very computationally demanding. Finding the most likely position of d QTL influencing a trait corresponds to a d-dimensional global optimization problem, where the evaluation of the objective function is performed by computing the statistical model fit for a given set of d QTL positions in the genome. So far, standard QTL mapping software [18,6,25,7] has used an exhaustive grid search for solving the global optimization problem. This type of algorithm is robust, but the computational requirement grows exponentially with d. This results in that often only mapping of a single QTL ($d = 1$) can be easily performed. Models with multiple QTL are normally fitted using for example a forward selection technique where a sequence of one-dimensional exhaustive searches are performed. In this type of procedure, the identified QTL are successively included as known quantities when searching for additional QTL. However, it is not clear how accurate this technique is for general QTL models. Lately, the interest in simultaneous mapping of multiple QTL has increased. Partly, the interest is motivated by analyses of real data sets [26,27,12] where certain interactions [11] between pairs of QTL have been found to only be detectable by solving the full two-dimensional optimization problem.

2 Efficient Computational Schemes for QTL Analysis

A popular approach for computing the model fit is the linear regression method [17,14,23], where a single least-squares problem is solved for each objective function evaluation. Efficient numerical algorithms for solving these least-squares problems in the QTL mapping setting are considered in [20,19]. Because of the exponential growth in work using the standard exhaustive grid search, algorithms for the global optimization problem for simultaneous mapping of multiple QTL have received special attention. Previously used optimization methods for QTL mapping problems include a genetic optimization algorithm, implemented for $d = 2$ using library routines [8], and an algorithm based on the DIRECT [16] scheme, implemented for $d = 2$ and $d = 3$ [21] and later improved to include a more efficient local search [22]. For multi-dimensional QTL searches, these new optimization algorithms are many orders of magnitude faster than an exhaustive grid search. The results indicate that the new algorithms in [22] enable simultaneous mapping of up to six QTL ($d = 6$) using a standard computer.

The purpose of the work presented in the rest of this paper is three-fold:

1. We want to be able to perform at least a few high-dimensional QTL mapping computations using the very costly exhaustive grid search. For real experimental data, we do not know the true optimal QTL positions a priori. Using results from exhaustive grid searches for representative data sets and models, we can evaluate the accuracy (and efficiency) of the more elaborate optimization methods mentioned above.

2. The extreme computational cost for performing the exhaustive grid searches for high-dimensional QTL mapping problems makes is necessary to implement a parallel computer code. This code will later provide a basis also for the implementation of the more efficient optimization schemes in a variety of high performance computing environments.
3. The structure of the multidimensional QTL search indicates that it can be efficiently implemented in a computational grid environment. Using a flexible parallel implementation, it is possible to investigate if this conjecture is valid.

3 Parallelization of Search Algorithms for Multiple QTL

In Section 5, we describe a flexible parallel implementation of a scheme for simultaneous mapping of several QTL, using the linear regression statistical method and an exhaustive grid search for finding the best model fit. More details about the problem setting can be found for example in [21,22]. In the experiments presented in Section 6, we use the parallel code to search for potential QTL positions in data from an experimental intercross between European wild boars and white domestic pigs consisting of 191 individuals [5]. The pig genome has 18 chromosomes, and its total length is $\sim$ 2300 cM[1]. In the computations we use models with two and three QTL, including both marginal and epistatic effects[2]. In this paper, we use this set of data and models as representative examples. We do not consider the relevance of the models used, nor do we consider the problem of which statistical model to use. Also, we do not attempt to establish the statistical significance of the results, and we do not draw any form of genetic implications from the computations. However, the code described in this paper provides a basis for future studies of all these issues, and for performing complete QTL mapping analyses using models including many QTL.

The search for the best QTL model fit should in principle be solved by optimizing over all positions x in a d-dimensional hypercube where the side is given by the size of the genome. The genome is divided into C chromosomes, resulting in that the search space hypercube consists of a set of C^d d-dimensional unequally sized *chromosome combination boxes*, cc-boxes. A cc-box can be identified by a vector of chromosome numbers $c = [c_1 \quad c_2 \ldots c_d]$, and consists of all x for which x_j is a point on chromosome c_j. The ordering of the loci does not affect the model fit, and this symmetry can be used to reduce the search space. We can restrict the search to cc-boxes identified by non-decreasing sequences of chromosomes. In addition, in cc-boxes where two or more edges span the same chromosome, for example $c = [1 \quad 8 \quad 8]$, we need only consider a part of the box.

Since genes on different chromosomes are unlinked, the objective function is normally discontinuous at the cc-box boundaries. This means that the QTL search could be viewed as essentially consisting of $n \approx C^d/2$ independent global

[1] A standard unit of genetic distance is *Morgan* [M]. However, distances are often reported in centi-Morgan [cM].

[2] Marginal effects are additive, i.e. the combined effect from two loci equals the sum of the individual effects. For epistatic effects, the relationship is nonlinear.

optimization problems, one for each cc-box included in the search space. Note that this fact must also be acknowledged when using more advanced optimization algorithms. For example, it is of course not possible to utilize derivative information across a cc-box boundary. This partitioning of the problem is a natural basis for a straight-forward parallelization of multi-dimensional QTL searches:

```
do (in parallel) i=1:n
  l_sol(i) = global_optimization(cc-box(i));
end
Find the global solution among l_sol(:);
```

The final (serial) operation only consists of comparing n objective function values, and the work is negligible compared to the work performed within the parallel loop. This type of parallelization was also used in [9] for mapping of single QTL.

Since the objective function evaluations (model fit computations) are rather expensive, the work for performing global minimization in a cc-box is almost exclusively determined by the number of calls to the objective function evaluation routine. For an exhaustive grid search algorithm, this number is known a priori. However, since the size of the different cc-boxes varies a lot (the chromosomes have very different lengths), the work will be very different for different boxes. Hence, the two main issues when implementing the algorithm above is load-balancing and granularity for the parallel loop.

We use an equidistant 1 cM grid for the exhaustive grid search, and exploit the symmetry of the search space to reduce the number of grid points. In Table 1, the total number of objective function evaluations and the total number of cc-boxes, n, for our pig data example are given for searches using models including different number of QTL d. For this example, the number of objective function evaluations for the different cc-boxes ranges from 400 to about 25000.

Table 1. The total number of objective function evaluations for different number of QTL in the model

number of QTL (d)	cc-boxes (n)	function evaluations
2	171	$2.645 \cdot 10^6$
3	1140	$2.089 \cdot 10^9$
4	5985	$1.260 \cdot 10^{12}$
5	26334	$0.612 \cdot 10^{15}$

4 Computational Grids and Other Parallel Computer Systems

Grid computing has been a buzz word in the computing community for some years, and numerous research projects involving grid systems and grid computations have been initiated. Still, a common view by computational science researches is that much work remains for grid systems to be mature enough for practical benefit. However, within a grid computing framework, inhomogeneous networks of commodity class computer and clusters can be used for performing large-scale computational tasks. This is especially valuable for research institutions where funding for large-scale, advanced HPC hardware can not be easily found.

The concept of a computational grid is very suitable for multiple instances of serial applications in need of a throughput computing capacity. If the communication requirement is limited and not heavily dependent on low communication latency, general computational grids can also be used for executing a parallel job utilizing web- or file-based communication. Moreover, if the grid is built up from standard parallel computer systems (clusters) it could be used for efficient execution of parallel jobs with higher communication needs, given that the parallel job is scheduled within a single cluster.

From the description of the generic parallel algorithm for multi-dimensional QTL search presented in Section 2 it is clear that this algorithm is suitable for implementation in a grid system. A very small amount of data is communicated, and communication only has to occur at two points in the algorithm.

The code for QTL search described in this paper was implemented to be easily ported to a large variety of parallel computer systems. In Section 6, we present experiments using a grid system and a shared memory server. Running the code to a standard cluster would be easy. As remarked earlier, our code will provide the basis for implementing a more complete QTL mapping software, using more efficient search algorithms, in a grid framework. Another initiative along these lines is the GridQTL project [15].

As an example of a computational grid we use is the SweGrid system [1]. SweGrid consists of six clusters distributed over the six national HPC centers in Sweden, and connected via the 1 Gbit SUNET national network. Each cluster has 100 nodes, where each node is equipped with a 2.8 GHz Intel P4 processor and 2 GB RAM. Within the clusters, the nodes are interconnected by standard gigabit Ethernet. The clusters run different versions of Linux, and the grid middleware Nordugrid [2] is used for submitting and scheduling jobs between clusters. Within the clusters, the jobs are scheduled using a standard queuing system, for example PBS. The Nordugrid jobs are specified in so called XRSL-files. Jobs parallelized with MPI can be submitted to the grid for execution within a cluster, but MPI parallelization is currently not possible across clusters.

As an example of a shared memory server, we use a SunFire 15k system at the Uppsala University HPC center UPPMAX [3]. The partition of the system used for our experiments consists of 32 UltraSPARC III+ CPUs running at 900 MHz, all sharing a 48 GB primary memory. The system runs Solaris 9, and the parallel jobs are submitted using the GridEngine N1 queuing system. For our QTL search application, the much higher memory bandwidth of the SunFire system actually results in that the serial performance is better than on the SweGrid system even though the clock speed of the CPU is much lower.

5 Implementation

Our implementation of the exhaustive grid search for multi-dimensional QTL problems is based on existing serial codes written in both C and Fortran 95. These codes use the efficient objective function evaluation algorithms described in [20]. When using the serial codes in the package it is possible to choose between different global optimization algorithms, including both exhaustive search and the much faster algorithms in [21]. As we remarked earlier, we will later include also these algorithms in the parallel version of the code.

The parallel implementation uses a hybrid, two-level scheme for partitioning the tasks corresponding to global optimization in cc-boxes over a set of parallel processors. On the outer level a static partitioning of tasks corresponding to blocks of cc-boxes is used, and on the inner level dynamic partitioning of tasks corresponding to single cc-boxes is exploited.

For the outer, static level a separate code partitions the cc-boxes over p_s different jobs. This is performed by defining a single-index ordering of the cc-boxes and assigning blocks of indices in this ordering to different jobs. The user specifies the block size, and the distribution of tasks is performed using a standard block-cyclic partitioning. As remarked earlier the number of objective function evaluations per cc-box is known a priori. It would be possible to take this information into account and use a partitioning algorithm of the type presented in [24]. However, the goal is to also include the more elaborate global optimization algorithms mentioned in Section 2 in the parallel code package in the future. In this case, the work per cc-box is not known beforehand, and we do not want the implementation to be dependent on this type of information.

The preparation code stores the description of the partitioning in p_s input files, which are then used by p_s instances of the computational code. A job control script submits the p_s computational jobs and wait for them to finish. In the Nordugrid environment the computational jobs are started using XRSL-files, and when using a standard queuing system some other form of script describing the jobs are used. In our grid implementation, we currently do not include any error handling for jobs that do not finish in a reasonable amount of time or do not finish at all. In an initialization step, the computational code reads its input file to get information about for which cc-boxes it should perform global optimization. When an instance of the computational code has performed the tasks assigned to it, it writes its local result (the optimal function value and the corresponding position vector x) to a result file. Finally, when all jobs have finished, a small separate code compares the local results and outputs the global optimum. In this context it should be noted that the amount of data used by the computational code is rather small, and data distribution is not an issue for the grid implementation.

The inner, dynamic level of parallelization is implemented using MPI. Each of the p_s instances of the computational code is executed as a $p_d + 1$-process MPI job, where the partitioning of the cc-boxes over the processes is performed using a master-slave scheme[3]. The master process maintains a queue of tasks, each consisting of global optimization in a single cc-box. These tasks are dynamically handed out to the p_d worker processes as soon as they have completed their previous task. At this level, the communication of cc-box specifications and results is of course implemented using MPI routines.

Using a hybrid scheme of the type described above gives us the flexibility to use only static or dynamic load balancing, or a combination of the two. Also, since we have used different parallelization tools for the two levels, we can easily port the code to a variety of different computer systems and configurations. In the experiments performed below we only present results for pure static or dynamic load balancing. However, on the SweGrid system, it would be possible to run the code in the hybrid mode where the Nordugrid middleware is used to submit a small number of MPI jobs, each consisting of a number of processes that are executed locally on one of the six clusters.

6 Results

In this section, we present results for QTL searches where three QTL are simultaneously included in the model ($d = 3$). The code is not limited to these type of problems, but before performing any very demanding experiments for problems where $d > 3$ we want to investigate the parallel behavior for problems that do not require extensive amounts of resources. Also, expensive high-dimensional searches should be carefully planned so that the results derived could be interesting also from a genetics perspective.

We start by presenting results for the shared memory server, where we compare the two strategies for load balancing implemented in the code. These results can also be used as a reference when assessing the performance on the grid system studied later. The run times reported are wall-clock timings, and the experiments were performed on a lightly loaded system.

In Table 2, timings are presented using both static and dynamic partitioning of the work as described in Section 5. When employing static partitioning, the preparation code was used to compute a block-cyclic partitioning of the tasks corresponding to the cc-boxes over p_s instances of the computational code, and no MPI parallelization was used at the inner level ($p_d = 1$). When using dynamic partitioning, a single instance of the computational code is run ($p_s = 1$), and the tasks corresponding to the cc-boxes are distributed one-by-one to p_d MPI processes using the dynamic pool-of-tasks scheme. For this problem with $d = 3$, the number of cc-boxes involved in the optimization is 1140, and the load balance for the static scheme is rather good. The table shows that the performance of the two schemes is very similar.

Next we present results for the SweGrid system. Here, we focus on the timings for the three-QTL model ($d = 3$). On this systems, the experiments had to be performed under regular load conditions and using the regular scheduling

[3] In the special case when $p_d = 1$, a single computational process is initiated and MPI is not used.

Table 2. Timings for $d = 3$ on the SMP. Static (left) and dynamic (right) partitioning

p_s $(p_d = 1)$	Speedup	Runtime [s]	p_d $(p_s = 1)$	Speedup	Runtime [s]
1	1	535813	1	1	535730
2	1.99	269450	2	1.97	272242
4	3.94	135905	4	3.84	139439
8	7.78	68979	8	7.49	71561
16	14.97	35804	16	15.18	35287

policies. Some indication of the load of the SweGrid system can be given by the idle time, i.e., the time the job has to wait in queue before it is started. However, the idle time also depends on the scheduling policy of the queuing systems at the local clusters, especially if MPI-job are used. In Table 3, timings for the static and dynamic work partitioning schemes are presented. Note that when using dynamic partitioning, a $p_d + 1$-processor MPI-job is executed on one of the SweGrid clusters. Since the static work partitioning scheme exploits XRSL-

Table 3. Timings for $d = 3$ on SweGrid. Static (left) and dynamic (right) partitioning

p_s $(p_d = 1)$	Speedup	Idle time [s]	Runtime [s]	p_d $(p_s = 1)$	Speedup	Idle time [s]	Runtime [s]
1	1	1020	955800	1	1	180	941400
2	2.03	240	471180	2	2.01	180	469140
4	3.93	360	243240	4	4.00	540	235200
8	7.56	28440	12480	8	8.02	240	117420
16	14.20	240	67320	16	8.99	181160	104760
30	24.36	600	39240	30	29.49	140760	31920
60	47.98	5460	19920	60	43.11	118440	21840

scripts for executing multiple instances of a serial code, it is similar to schemes used in other major grid applications, for example by the LHC Cern project [4]. From the results in Table 3, it is also clear that the scheduling policies used in the queuing systems at the SweGrid clusters favors this type usage of the grid. Parallel MPI-jobs often have to wait in queue a long time before they are started. To some degree, this time could be reduced by giving parallel jobs higher priorities. However, when the load on the system is high it is probably difficult to improve the situation very much.

7 Conclusions

A major aim of this paper was to present a flexible parallel implementation of multidimensional QTL search. When using the standard exhaustive grid search algorithm for solving the global optimization problem for finding the best QTL model fit, it is easy to partition the work into a large number of independent tasks corresponding to cc-boxes in the search space. Also, the work per cc-box can be accurately estimated a priori, and the experimental results in Section 6 show that both static and dynamic work partitioning schemes are efficient.

Another major aim was to investigate if it possible to solve the multidimensional QTL search problem efficiently on a computational grid. Again, the results presented in Section 6 show that this is indeed the case. The speedup achieved when using the SweGrid system is very similar to when a shared memory server is used. For our implementation, the static partitioning scheme resulted in shorter turn-around times than the dynamic scheme. The reason for this is that the static version uses serial jobs which are efficiently scheduled on available processor nodes by the Nordugrid middleware, resulting in short queuing times for the jobs. Our implementation of the dynamic scheme uses MPI, and thus a number of processors must be available on a cluster before the local queuing system can start the execution of the job. When the number of processors requested is large, this results in long queuing times. On SweGrid, our static *partitioning* of work results in dynamic *execution* of the corresponding computational tasks while our dynamic partitioning in a sense results in static execution.

Finally, it should be noted that the parallelization technique presented in this paper is adopted for efficient execution of a single, multidimensional QTL search. For other QTL analysis settings, other schemes for parallelization should be used. For example, multiple instances of non-expensive searches where $d = 1$ and possibly also $d = 2$ are perfectly suitable for serial execution in a grid environment. Another class of problems arises in analyses where low-dimensional ($d = 1$ or $d = 2$) QTL searches are performed for a single set of genetic data, but for a large number of phenotypes. Here, the phenotypes can arise from microarray experiments or from permuted data sets used in significance testing. In this case, it is more efficient to parallelize the computations over the phenotypes, as described in [10]

References

1. `http://www.swegrid.se`
2. `http://www.nordugrid.org`
3. `http://www.uppmax.uu.se`
4. `http://lhc.web.cern.ch/lhc/`
5. Andersson, L., Haley, C., Ellegren, H., Knott, S., Johansson, M., Andersson, K., Andersson-Eklund, L., Edfors-Lilja, I., Fredholm, M., Hansson, I.: Genetic mapping of quantitative trait loci for growth and fatness in pigs. Science 263, 1771–1774 (1994)
6. Basten, C., Weir, B., Zeng, Z.-B.: QTL Cartographer, Version 1.15. Department of Statistics, North Carolina State University, Raleigh, NC (2001)
7. Broman, K., Wu, H., Sen, S., Churchill, G.: R/qtl: QTL mapping in experimental crosses. Bioinformatics 19, 889–890 (2003)
8. Carlborg, Ö., Andersson, L., Kinghorn, B.: The use of a genetic algorithm for simultaneous mapping of multiple interacting quantitative trait loci. Genetics 155, 2003–2010 (2000)
9. Carlborg, Ö., Andersson-Eklund, L., Andersson, L.: Parallel computing in regression interval mapping. Journal of Heredity 92(5), 449–451 (2001)

10. Carlborg, Ö., de Koning, D.-J., Manly, K., Chesler, E., Williams, R., Haley, C.: Methodological aspects of the genetic dissection of gene expression. Bioinformatics 21, 2383–2393 (2005)
11. Carlborg, Ö., Haley, C.S.: Epistasis: too often neglected in complex trait studies? Nature Reviews Genetics 5, 618–625 (2004)
12. Carlborg, Ö., Kerje, S., Schütz, K., Jacobsson, L., Jensen, P., Andersson, L.: A global search reveals epistatic interaction between QTL for early growth in the chicken. Genome Research 13, 413–421 (2003)
13. Doerge, R.: Mapping and analysis of quantitative trait loci in experimental populations. Nature Reviews Genetics 3, 43–52 (2002)
14. Haley, C., Knott, S.: A simple regression method for mapping quantitative trait loci in line crosses using flanking markers. Heredity 69, 315–324 (1992)
15. Institute of Evolutionary Biology, Roslin Institute and National e-Science Centre, Edinburgh. GridQTL, `http://www.gridqtl.org.uk/`
16. Jones, D., Perttunen, C., Stuckman, B.: Lipschitzian optimization without the Lipschitz constant. Journal of Optimization Theory and Application 79, 157–181 (1993)
17. Knapp, S., Bridges, W., Birkes, D.: Mapping quantitative trait loci using molecular marker linkage maps. Theoretical and Applied Genetics 79, 583–592 (1990)
18. Lincoln, S., Daly, M., Lander, E.: Mapping genes controlling quantitative traits with MAPMAKER/QTL 1.1. Technical report, Whitehead Institute, Technical report. 2nd edn. (1992)
19. Ljungberg, K.: Efficient evaluation of the residual sum of squares for quantitative trait locus models in the case of complete marker genotype information. Technical Report 2005-033, Division of Scientific Computing, Department of Information Technology, Uppsala University (2005)
20. Ljungberg, K., Holmgren, S., Carlborg, Ö.: Efficient algorithms for quantitative trait loci mapping problems. Journal of Computational Biology 9(6), 793–804 (2002)
21. Ljungberg, K., Holmgren, S., Carlborg, Ö.: Simultaneous search for multiple QTL using the global optimization algorithm DIRECT. Bioinformatics 20, 1887–1895 (2004)
22. Ljungberg, K., Mishchenko, K., Holmgren, S.: Efficient algorithms for multidimensional global optimization in genetic mapping of complex traits. Technical Report 2005-035, Division of Scientific Computing, Department of Information Technology, Uppsala University (2005)
23. Martinez, O., Curnow, R.: Estimating the locations and the sizes of effects of quantitative trait loci using flanking markers. Theoretical and Applied Genetics 85, 480–488 (1992)
24. Olstad, B., Manne, F.: Efficient partitioning of sequences. IEEE Transactions on Computers 44(11), 1322–1326 (1995)
25. Seaton, G., Haley, C., Knott, S., Kearsey, M., Visscher, P.: QTL express: mapping quantitative trait loci in simple and complex pedigrees. Bioinformatics 18, 339–340 (2002)
26. Shimomura, K., Low-Zeddies, S., King, D., Steeves, T., Whiteley, A., Kushla, J., Zemenides, P., Lin, A., Vitaterna, M., Churchill, G., Takahashi, J.: Genome-wide epistatic interaction analysis reveals complex genetic determinants of circadian behavior in mice. Genome Research 11, 959–980 (2001)
27. Sugiyama, F., Churchill, G., Higgins, D., Johns, C., Makaritsis, K., Gavras, H., Paigen, B.: Concordance of murine quantitative trait loci for salt-induced hypertension with rat and human loci. Genomics 71, 70–77 (2001)

Partial Approximation of the Master Equation by the Fokker-Planck Equation

Paul Sjöberg

Division of Scientific Computing, Department of Information Technology, Uppsala University, P.O. Box 337, SE-751 05 Uppsala, Sweden
Paul.Sjoberg@it.uu.se

Abstract. The *chemical master equation* (CME) describes the probability for each internal state of the cell or rather the states of a model of the cell. The number of states grows exponentially with the number of chemical species in the model, since each species corresponds to one dimension in the state space. The CME can be approximated by a *Fokker-Planck equation* (FPE), which can be solved numerically cheaper than the CME. The FPE approximation of the full CME is not always appropriate, while it can be suitable for a subspace in the state space. In order to exploit the lower cost of the FPE approximation a method for splitting the state space in two subspaces where one is approximated by the FPE and one remains unapproximated is presented. A biologically relevant problem in four dimensions is solved as an example.

1 Introduction

Chemical reactions are often accurately described with a system of ordinary differential equations that is called the reaction-rate equations. Each equation accounts for the change in concentration for a chemical species. This system of equations describes a completely deterministic evolution of the chemical reactor.

The reaction-rate equations assume large number of molecules and that the system is close to chemical equilibrium. These are reasonable assumptions in chemical engineering, but not in molecular biology. When the reactor volume is small with low copy numbers of some of the reacting molecules such a deterministic model is not always good enough. The discreteness of reactions and molecules introduces a noise that cannot be disregarded [20]. Even if copy numbers are comparatively large there can be large fluctuations [5].

Stochastic models describe the system in a probabilistic way, where the probability for every possible state of the system is considered. This is equivalent to determining the frequency of each state in an ensemble of cells, while the reaction-rate equations is a model for the evolution of the mean state in the ensemble. The theoretical importance of stochastic models has been known for long time (see for instance the review [17]), but in recent years the experimental results have been gathering as well [1], [6], [8] and as experimental methods are developed to examine single molecules in single cells [3], [21] more experiments can be expected.

The *chemical master equation* (CME) is used to describe the time-evolution of the probability density function of the state of the model. It is expensive to solve numerically. This paper describes a splitting of the CME into one part that is approximated by a *Fokker-Planck equation* (FPE) and one that is not.

2 Stochastic Biochemical Models

The physical foundation for the master equation as a stochastic model of biochemical reactions requires some assumptions to be valid [11]. The basic assumption for our model is that the system is well approximated by a Markov process, that is a stochastic process with no memory. It is only the current state that may affect the probability of the next reaction.

Definition 1. *Let the system contain N chemical species X_i, $i = 1 \dots N$ and M reaction channels R_ν, $\nu = 1 \dots M$. Denote the number of X_i-molecules by $x_i \in [0, \infty)$. Let the state of the system be determined by $\mathbf{x} = (x_1, x_2, \dots, x_N)^T$. Define $c_\nu \delta t$ as the probability for reaction R_ν between a given set of reactant molecules in the next time interval δt. Also define $h_\nu(\mathbf{x})$ as the number of such reactive sets and the propensity of reaction R_ν, $w_\nu(\mathbf{x}) = h_\nu(\mathbf{x}) c_\nu$.*

We assume that the biological cell is a homogeneous mixture of the reactive molecules and some solvent. This is necessary due to the basic assumption since there is no spatial component in the state variable and the process has no memory. The rationale for the well-stirred assumption is that the reactor volume can be considered to be well-stirred if each reactive molecule collides a large number of times with inert molecules between subsequent reaction events. The result is that there is an equal probability for any two molecules to collide, which is necessary for a reaction to take place.

A reaction is specified by a *reaction propensity* $w_\nu(\mathbf{x})$, which is defined as the probability for reaction R_ν per unit time, and stoichiometric coefficients $\mathbf{s} = (s_1, s_2, \dots, s_N)^T$ determining the change in molecule numbers as the reaction is fired. A state change can be written

$$\mathbf{x} \to \mathbf{x} + \mathbf{s}\,.$$

The propensity depends on the probability of collision and the probability for reaction given that a collision has occurred. The latter probability depends on the velocity distribution of the reactant molecules (hence the temperature), the activation energy of the reaction, the probability of the molecules to be positioned in a reactive orientation and so on.

The probability for collision of molecules is naturally dependent on the volume of the reactor. Cells grow and divide which makes the propensities time dependent and introduce additional noise due to the random splitting of molecules between daughter cells. This complication is not difficult to include in the presented framework. Here a constant volume assumption is made to avoid these technicalities.

3 Methods

3.1 The Chemical Master Equation

Consider a system as defined in Definition 1. Let $-\mathbf{n}_\nu$ denote the stoichiometric coefficients of reaction R_ν. Summing over all reactions we now can write the CME [20]:

$$\frac{dp(\mathbf{x},t)}{dt} = \sum_{\nu=1}^{M} w_\nu(\mathbf{x}+\mathbf{n}_\nu)p(\mathbf{x}+\mathbf{n}_\nu,t) - \sum_{\nu=1}^{M} w_\nu(\mathbf{x})p(\mathbf{x},t)\,. \tag{1}$$

The CME states that the change in probability for state $\mathbf{x}$ is simply the probability to reach state $\mathbf{x}$ from any other state $\mathbf{x}+\mathbf{n}_\nu$ (first sum), minus the probability to leave state $\mathbf{x}$ for any other state (second sum).

The size of the state space grows exponentially with the number of molecular species in the model. To compute probability density functions for larger and larger biochemical models, the aim must be to allow solution of CMEs with state spaces of higher and higher dimension. In that perspective, the impact of parallelization is modest. It will add one or two dimensions to the set of solvable problems. For larger problems it is necessary to use approximations.

3.2 The Fokker-Planck Approximation

For many problems the discrete CME has too many degrees of freedom to be computationally tractable, even for rather low dimensions. If it is replaced by a continuous approximation the discretization of the approximation has substantially fewer degrees of freedom.

By Taylor expansion of the CME and truncation after the second order terms the following FPE is obtained [20]

$$\frac{\partial p(\mathbf{x},t)}{\partial t} = \sum_{\nu=1}^{M}\left(\sum_{i=1}^{N} n_{\nu i}\frac{\partial(w_\nu(\mathbf{x})p(\mathbf{x},t))}{\partial x_i} + \sum_{i=1}^{N}\sum_{j=1}^{N}\frac{n_{\nu i}n_{\nu j}}{2}\frac{\partial^2(w_\nu(\mathbf{x})p(\mathbf{x},t))}{\partial x_i \partial x_j}\right)$$
$$\equiv \sum_{\nu=1}^{M} A_\nu p\,, \tag{2}$$

where $n_{\nu i}$ is the i:th element of $\mathbf{n}_\nu$.

3.3 Merging the Discrete CME and the Continuous FPE

Consider a CME for a system as described in Definition 1. Let $\mathbb{X}$ be the space of all physical states (i.e. fulfilling $x_i \geq 0$ and other restrictions due to the model). Divide the variables into two subsets X and Y such that $\operatorname{span} X \cup Y = \mathbb{X}$ so that Y is suitable for FPE approximation, but X is not. Split the stoichiometric coefficients accordingly so that μ_ν and η_ν are vectors containing the elements of $\mathbf{n}_\nu$ corresponding to the variables in X and Y, respecetively.

The CME (1) for a splitting where $\mathbf{x} \in \operatorname{span} X$ and $\mathbf{y} \in \operatorname{span} Y$ is now written

$$\frac{\partial p}{\partial t}(\mathbf{x},\mathbf{y},t) = \sum_{\nu} w_\nu(\mathbf{x}+\mu_\nu,\mathbf{y}+\eta_\nu)p(\mathbf{x}+\mu_\nu,\mathbf{y}+\eta_\nu) - \sum_{\nu} w_\nu(\mathbf{x},\mathbf{y})p(\mathbf{x},\mathbf{y},t)\,.$$

Introduce $q_\nu(\mathbf{x},t) = w_\nu(\mathbf{x})p(\mathbf{x},t)$ and apply the FPE approximation to the variables in Y

$$\begin{aligned}\frac{\partial p}{\partial t}(\mathbf{x},\mathbf{y},t) &= \sum_{\nu} q_\nu(\mathbf{x}+\mu_\nu,\mathbf{y}+\eta_\nu,t) - q_\nu(\mathbf{x},\mathbf{y},t)\\ &= \sum_{\nu} q_\nu(\mathbf{x}+\mu_\nu,\mathbf{y}+\eta_\nu,t) - q_\nu(\mathbf{x}+\mu_\nu,\mathbf{y},t) + q_\nu(\mathbf{x}+\mu_\nu,\mathbf{y},t) - q_\nu(\mathbf{x},\mathbf{y},t)\\ &= \sum_{\nu}\left(\sum_i \eta_{\nu i}\frac{\partial q_\nu}{\partial y_i} + \frac{1}{2}\sum_i\sum_j \eta_{\nu i}\eta_{\nu j}\frac{\partial^2 q_\nu}{\partial y_i \partial y_j}\right)(\mathbf{x}+\mu_\nu,\mathbf{y},t) +\\ &\qquad \sum_{\nu} q_\nu(\mathbf{x}+\mu_\nu,\mathbf{y},t) - q_\nu(\mathbf{x},\mathbf{y},t)\,. \end{aligned} \tag{3}$$

For example, if the sets X and Y have one member each, $x \in X$ and $y \in Y$ where $x \in \{0,1,\ldots,k\}$, and $max_\nu|\mu_\nu| = 1$ then (3) can be written as the following sum of block-diagonal matrices (zero elements not written)

$$p = \begin{pmatrix} p(0,y)\\ p(1,y)\\ \vdots\\ p(k,y)\end{pmatrix}, \qquad \frac{\partial p}{\partial t} = \sum_{\nu,\mu_\nu=0}\begin{pmatrix} A_{\nu,0} & & & \\ & A_{\nu,1} & & \\ & & \ddots & \\ & & & A_{\nu,k}\end{pmatrix} p$$

$$+ \sum_{\nu,\mu_\nu=1}\begin{pmatrix} -w_\nu(0,y) & A_{\nu,1}+w_\nu(1,y) & & \\ & -w_\nu(1,y) & A_{\nu,2}+w_\nu(2,y) & \\ & \ddots & \ddots & \\ & & -w_\nu(k-1,y) & A_{\nu,k}+w_\nu(k,y)\\ & & & -w_\nu(k,y)\end{pmatrix} p$$

$$+ \sum_{\nu,\mu_\nu=-1}\begin{pmatrix} -w_\nu(0,y) & & \\ A_{\nu,0}+w_\nu(0,y) & -w_\nu(1,y) & \\ \ddots & \ddots & \\ & A_{\nu,k-1}+w_\nu(k-1,y) & -w_\nu(k,y)\end{pmatrix} p\,,$$

where $A_{\nu,i}$ is the FPE operator from (2) for the y-variable and constant $x = i$. Since negative molecule numbers are nonsense, $w_\nu(0,y) = 0$ if $\mu_\nu = 1$. The numerical boundary at the truncation of the state space at k can be chosen more freely, but here $w_\nu(k,y) = 0$ if $\mu_\nu = -1$, assuming the probability at this boundary is zero.

3.4 Discretization of the Fokker-Planck Equation

The FPE is discretized and solved on a grid that is considerably coarser than the state space [4]. A finite volume method (see [18]) is used to discretize the state space in the subspace where the master equation is approximated with an FPE. Reflecting barrier boundary conditions [7] are used which prescribe that there is no probability current over the boundary.

3.5 The Stochastic Simulation Algorithm (SSA)

The well-known *stochastic simulation algorithm* (SSA) was proposed by Gillespie in 1976 [10] for simulation of stochastic trajectories through the state space of chemical reaction networks. Since then several improvements, approximations and extensions [2] [9], [12], [16], have been made to the original algorithm that we summarize here. SSA can be used to solve the unapproximated master equation and is relevant for comparison to the FPE-method with respect to accuracy and efficiency.

SSA was not designed for the purpose of solving the CME, which at the time seemed "virtually intractable, both analytically and numerically" [10]. That statement is not out of date for the vast majority of CME problems, but numerical solution for small reaction networks has become feasible.

Define the system as in Definition 1. The probability that the next reaction is fired in the interval $(t+\tau, t+\tau+\delta t)$ and is of type R_λ is

$$p(\tau,\lambda)\delta t = w_\lambda \exp(-\sum_{\nu}^{M} w_\nu \tau).$$

SSA samples $p(\tau,\lambda)$ in order to take a statistically correct Monte Carlo step. The algorithm for generating a trajectory is:

1. Initialize the state $\mathbf{x}$, compute reaction propensities $w_\nu(\mathbf{x})$, $\nu = 1 \dots M$ and set $t = 0$.
2. Generate a sample (τ, λ) from the distribution $p(\tau,\lambda)$
3. Increase time by τ and change the state according to reaction R_λ
4. Store population and check for termination, otherwise go to step 2

By simulating an ensemble of trajectories the state probability distribution can be estimated and a solution to the master equation is obtained. The error of SSA can be estimated in a statistical sense. With a certain probability the error is within the prescribed error bound [18].

3.6 Computational Efficiency

The convergence rate of solution by the FPE approximation is derived in [18]. For a certain error ϵ the computational work for the SSA is

$$W_{SSA}(\epsilon) = C_{SSA}\,\epsilon^{-2},$$

while the work for the FPE is

$$W_{FPE}(\epsilon) = C_{FPE}\,\epsilon^{-\left(\frac{N}{r}+\frac{1}{s}\right)},$$

where C_{SSA} and C_{FPE} are independent of ϵ, N is the dimension of the problem and r and s are the order of accuracy of the space- and time-discretizations, respectively. For some systems, numerical solution of the FPE approximation can be much more efficient than SSA [18]. If, like here, second order accurate discretizations schemes are used in time and space, less work is needed for FPE when $N < 3$ and a small ϵ. For the hybrid method N is the number of variables in Y.

High order schemes have a great potential of allowing solution of larger problems, but the dimensionality of the feasible problems will still be low compared to the actual number of dimensions in most molecular biological models. There are no principal impediments for using higher order schemes.

4 An Example Problem

To demonstrate a problem where the partial approximation is useful, consider a gene which is regulated directly by its product. The gene products binds cooperatively at two sites, S_1 and S_2 in the regulatory region of the gene. There are five molecular species in the model (notation within parenthesis): the gene product - a metabolite (M), the gene with S_1 and S_2 unoccupied (DNA), the gene with S_1 occupied by an M-molecule (DNA_M), the gene with both S_1 and S_2 occupied by M-molecules (DNA_{2M}) and mRNA (RNA). The number of molecules is denoted by the token for the corresponding species in lowercase characters, i.e. m, dna, dna_M, dna_{2M} and rna. Figure 1 shows the reactions of the model. A metabo-

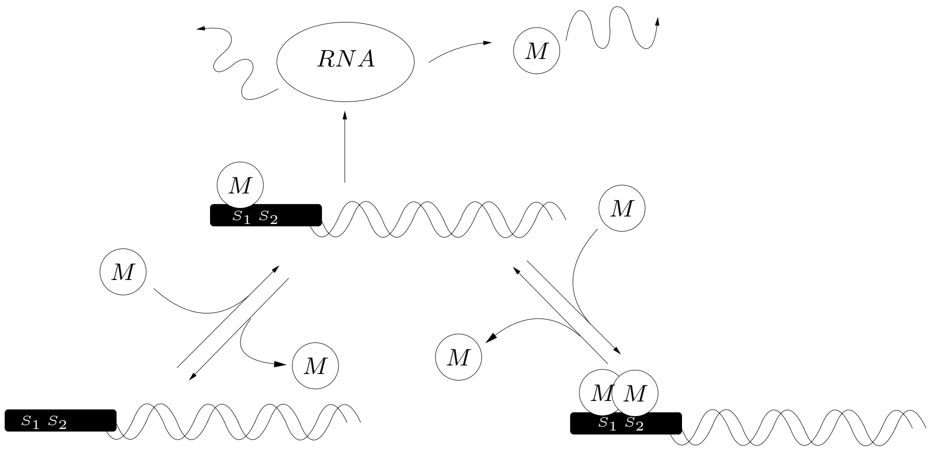


Fig. 1. The reaction scheme of the example

lite bound to S_1 activates transcription while a metabolite bound to S_2 blocks RNA polymerase and shuts down mRNA production. Cooperativity is necessary for metabolite binding to S_2 and so strong that a metabolite bound to S_1 will not unbind while S_2 is occupied. The metabolite and mRNA are actively degraded. This example is a simplified version of the transcription regulation example in [13].

No reactions change the number of gene copies. There are exactly two copies during the entire simulation, i.e. the probability for $dna + dna_M + dna_{2M} = 2$ is 1. Therefore the state space is actually a four-dimensional surface in the five-dimensional state space and we can reduce the state description to $\tilde{\mathbf{x}} = (dna, dna_M, rna, m)^T$ and substitute $dna_{2M} = 2 - dna - dna_M$ in the propensity functions. Table 1 lists the reactions of the reduced model. The symbol $\emptyset$ denotes that no molecules are created in the reaction.

Table 1. The reactions of the example system

Reaction	**Stoichiometric Coeff.**	**Propensity**
$RNA \xrightarrow{w_1} RNA + M$	$(0, 0, 0, 1)^T$	$w_1 = 0.05 \cdot rna$
$M \xrightarrow{w_2} \emptyset$	$(0, 0, 0, -1)^T$	$w_2 = 0.001 \cdot m$
$DNA_M \xrightarrow{w_3} RNA + DNA_M$	$(0, 0, 1, 0)^T$	$w_3 = 0.1 \cdot dna_M$
$RNA \xrightarrow{w_4} 0$	$(0, 0, -1, 0)^T$	$w_4 = 0.005 \cdot rna$
$DNA + M \xrightarrow{w_5} DNA_M$	$(-1, 1, 0, -1)^T$	$w_5 = 0.02 \cdot dna \cdot m$
$DNA_M \xrightarrow{w_6} DNA + M$	$(1, -1, 0, 1)^T$	$w_6 = 0.5 \cdot dna$
$DNA_M + M \xrightarrow{w_7} DNA_{2M}$	$(0, -1, 0, -1)^T$	$w_7 = 2 \cdot 10^{-4} \cdot dna_M \cdot m$
$DNA_{2M} \xrightarrow{w_8} DNA_M + M$	$(0, 1, 0, 1)^T$	$w_8 = 1 \cdot 10^{-11} \cdot (2 - dna - dna_M)$

5 Experiments

The example in Section 4 was simulated for 35 minutes (approximately one cell generation time) using SSA and the hybrid approach. The two variables *dna* and dna_M were represented by the CME part (X) and *rna* and m were represented by the FPE part of the hybrid method. The FPE part (Y) was discretized in the $rna \times m$-plane using 20×60 cells of equal length in the m-dimension and of increasing length in the RNA-dimension. Let h_k be the length of the k:th cell in the RNA-dimension. The step lengths were determined by $h_k = (1 + \theta)\, h_{k-1}$, $i = 2 \ldots 20$ and $h_1 = 0.5$. For time integration of (3) the implict second order *backward differentiation formula* scheme (BDF-2) [14] was used. The time step was chosen adaptively according to [15] with an error tolerance, measured in the L_1-norm, of one percent in each time step. The system of equations that arise in each time step in BDF-2 is solved using BiCGSTAB [19]. The implicit method is suitable for handling the ubiquitous stiffness of molecular biological systems. As initial data, a normal distribution in the $rna \times m$-plane was truncated at the boundaries of the computational domain and rescaled. The mean $E(\mathbf{x})$ and variance $V(\mathbf{x})$ of the normal distribution *before* truncation was $E(\mathbf{x}) = (1, 5)^T$ and $V(\mathbf{x}) = (1, 1)^T$. The probability for being in a state where $dna = 2$ and $dna_M = 0$ was set to 1.

For SSA the simulation was used to compute mean and variance, but not an approximation of the probability density function. The initial state of each trajectory was sampled from the initial distribution that was used for the CME-FPE-hybrid simulation.

5.1 Results

Due to numerical approximation errors the solution is slightly negative at some parts of the state space. In order to calculate the mean and standard deviation those negative values are set to zero. Figure 2 shows the means and standard deviations for the hybrid method compared to what is obtained by SSA using 10^4 trajectories. Figure 3 shows a projection of the probability density function.

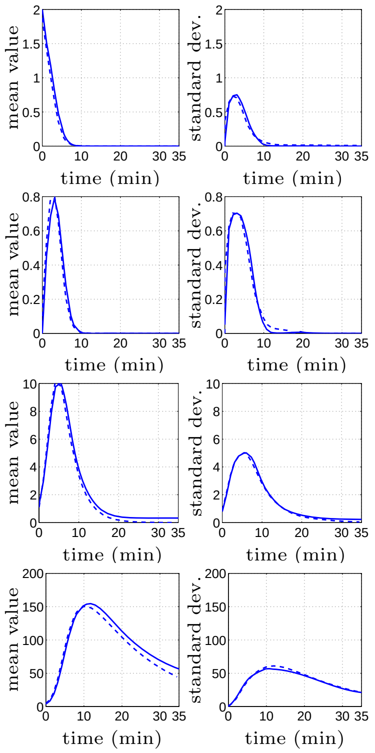


Fig. 2. Mean values (*left*) and standard deviations (*right*) for, from top to bottom, *dna*, dna_M, *rna* and *m*. The hybrid solution (*solid line*) and the SSA solution (*broken line*) is plotted for each case.

Isolines of

$$\tilde{p}(rna, m) = \sum_{dna, dna_M} p(dna, dna_M, rna, m)$$

are plotted at four different times as indicated in the figure.

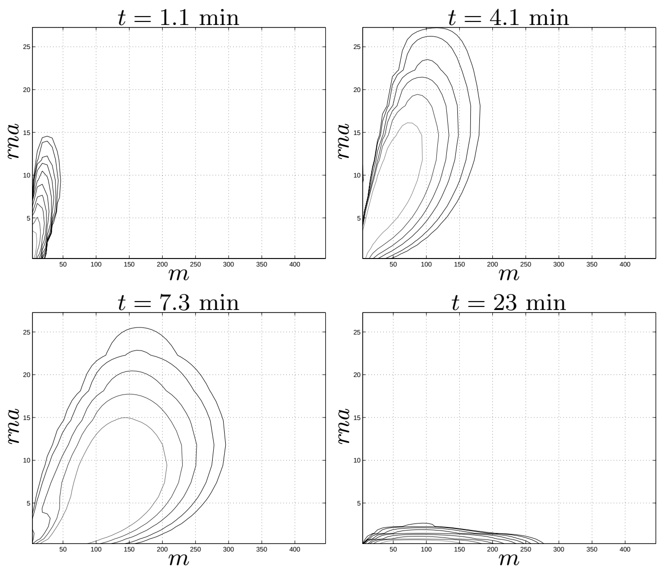


Fig. 3. Snapshots of the numerical solution projected on the $rna \times m$-plane

6 Conclusion

The splitting of the state space in two subspaces where one subspace is suitable for approximation with the chemical master equation and one is not, extends the range of problems that can solved numerically using the FPE approximation of the master equation.

The main reason for introducing this hybrid method is that the FPE often is a good approximation for some, but not all, dimensions in the state space. Typically genes and gene configurations will not be well approximated. Such dimensions need to be resolved by more grid points than the actual number of states in that dimension. Furthermore the error in the FPE approximation is bounded by the third derivatives [18]. For low copy numbers the probability of a single dominating state is high resulting in probability peaks that cannot be well approximated due to large third derivatives. There is also a point in avoiding approximations when they are not needed.

Acknowledgements

I want to thank Per Lötstedt and Johan Elf for helpful discussions. This work was funded by the Swedish Research Council, the Swedish National Graduate School in Scientific Computing and the Swedish Foundation for Strategic Research.

References

1. Blake, W.J., Kærn, M., Cantor, C.R., Collins, J.J.: Noise in eukaryotic gene expression. Nature 422, 633–637 (2003)
2. Bratsun, D., Volfson, D., Tsimring, L.S., Hasty, J.: Delay-induced stochastic oscillations in gene regulation. Proc. Natl. Acad. Sci. USA 102(41), 14593–14598 (2005)
3. Cai, L., Friedman, N., Xie, X.S.: Stochastic protein expression in individual cells at the single molecule level. Nature 440, 358–362 (2006)
4. Elf, J., Lötstedt, P., Sjöberg, P.: Problems of high dimensionality in molecular biology. In: Hackbusch, W. (ed.) High-dimensional problems - Numerical treatment and applications, Proceedings of the 19th GAMM-Seminar, Leipzig, pp. 21–30 (2003), available at `http://www.mis.mpg.de/conferences/gamm/2003/`
5. Elf, J., Paulsson, J., Berg, O.G., Ehrenberg, M.: Near-critical phenomena in intracellular metabolite pools. Biophys. J. 84, 154–170 (2003)
6. Elowitz, M.B., Levine, A.J., Siggia, E.D., Swain, P.S.: Stochastic gene expression in a single cell. Science 297, 1183–1186 (2002)
7. Gardiner, C.W.: Handbook of Stochastic Methods, 2nd edn. Springer, Heidelberg (1985)
8. Gardner, T.S., Cantor, C.R., Collins, J.J.: Construction of a genetic toggle switch in Escherichia coli. Nature 403, 339–342 (2000)
9. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. J. Phys. Chem. 104, 1876–1889 (2000)
10. Gillespie, D.T.: A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. J. Comput. Phys. 22, 403–434 (1976)
11. Gillespie, D.T.: A rigorous derivation of the chemical master equation. Physica A 188, 404–425 (1992)
12. Gillespie, D.T.: Approximate accelerated stochastic simulation of chemically reacting systems. J. Chem. Phys. 115(4), 1716 (2001)
13. Goutsias, J.: Quasiequilibrium approximation of fast reaction kinetics in stochastic biochemical systems. J. Chem. Phys. 184102 (2005)
14. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations, Nonstiff Problems, 2nd edn. Springer, Heidelberg (1993)
15. Lötstedt, P., Söderberg, S., Ramage, A., Hemmingsson-Frändén, L.: Implicit solution of hyperbolic equations with space-time adaptivity. BIT 42, 134–158 (2002)
16. Lu, T., Volfson, D., Tsimring, L., Hasty, J.: Cellular growth and division in the Gillespie algorithm. Syst. Biol. 1, 121–128 (2004)
17. McQuarrie, D.A.: Stochastic approach to chemical kinetics. J. Appl. Prob. 4, 413–478 (1967)
18. Sjöberg, P., Lötstedt, P., Elf, J.: Fokker-Planck approximation of the master equation in molecular biology. Comput. Visual. Sci. 2006 (accepted for publication)
19. van der Vorst, H.A.: BiCGSTAB: A fast and smoothly converging variant of the Bi-CG for the solution of nonsymmetric linear systems. SIAM J. Sci and Stat. Comp 13, 631–644 (1992)
20. van Kampen, N.G.: Stochastic Processes in Physics and Chemistry, 2nd edn. Elsevier, Amsterdam (1992)
21. Yu, J., Xiao, J., Ren, X., Lao, K., Xie, X.S.: Probing gene expression in live cells, one protein molecule at a time. Science 311, 1600 (2006)

Design, Construction and Use of the FISH Server

Jeanette Tångrot[1,2], Lixiao Wang[1], Bo Kågström[2,3], and Uwe H. Sauer[1]

[1] Umeå Centre for Molecular Pathogenesis
[2] Department of Computing Science
[3] High Performance Computing Center North (HPC2N),
Umeå University, SE-901 87 Umeå, Sweden
{jeanette, bokg}@cs.umu.se, {lixiao, uwe}@ucmp.umu.se

Abstract. At the core of the FISH (**F**amily **I**dentification with **S**tructure anchored **H**idden Markov models, saHMMs) server lies the midnight ASTRAL set. It is a collection of protein domains with low mutual sequence identity within homologous families, according to the structural classification of proteins, SCOP. Here, we evaluate two algorithms for creating the midnight ASTRAL set. The algorithm that limits the number of structural comparisons is about an order of magnitude faster than the all-against-all algorithm. We therefore choose the faster algorithm, although it produces slightly fewer domains in the set. We use the midnight ASTRAL set to construct the structure-anchored Hidden Markov Model data base, saHMM-db, where each saHMM represents one family. Sequence searches using saHMMs provide information about protein function, domain organization, the probable 2D and 3D structure, and can lead to the discovery of homologous domains in remotely related sequences.

The FISH server is accessible at `http://babel.ucmp.umu.se/fish/`.

1 Introduction

Genome sequencing projects contribute to an exponential increase of available DNA and protein sequences in data bases. Millions of sequence entries contain remarks such as “hypothetical”, “unidentified”, or “unknown”. It is therefore crucial to develop accurate automated sequence annotation methods. For proper characterization of newly sequenced proteins it is important to associate them with homologous proteins of well characterized functions and possibly high quality three dimensional (3D) structures. Proteins are modular and can harbour many domains. Consequently, it is advisable to characterize the constituent domains rather than the protein as a whole. Existing resources, such as Pfam [1], Superfamily [7], SMART [6] and others, provide the user with versatile tools for domain identification. Common for these methods is that they use protein sequence alignments that include as many sequences as possible, even with high sequence identity of up to 95%, to construct hidden Markov models, HMMs. At the core of our approach lies a data base of structure-anchored hidden Markov models, saHMMs. In contrast to the other methods, we derive structure anchored multiple sequence alignments, saMSAs, exclusively from multiple structure superimpositions of protein domains within SCOP families [9]. Only spatial distance criteria are considered to find matching residues and to deduce the multiple sequence alignments from which the saHMMs are built. Great care is taken to ensure sequence diversity among the domains by including only such members with a mutual sequence identity below a certain cut-off value. We call the data set containing the low mutual sequence identity domains the "midnight ASTRAL set", since it was derived using the ASTRAL compendium [2]. We have made the saHMM data base, saHMM-db, publicly available through the FISH server, which has been introduced and briefly described earlier [13]. FISH, which stands for Family Identification with Structure-anchored HMMs, is a versatile server for the identification of domains in protein sequences. Here, we describe the algorithms behind the server in more detail, in particular the creation of the midnight ASTRAL set. In addition, we present a layout of the cross-linking of the underlying data bases and describe in more detail how to use the server.

2 The Midnight ASTRAL Set and Selection Algorithms

The midnight ASTRAL set is the non-redundant collection of representative domains used to construct the saHMMs. In order to maximize the sequence variability within each SCOP domain family [9], we included only domains with low mutual sequence identities, below the "twilight zone" curve, $p^I(L,0)$ [10],[8]:

$$p^I(L,n) = n + \begin{cases} 100 & for \;\; L \le 11, \\ 480 \cdot L^{-0.32 \cdot \left(1+e^{-L/1000}\right)} & for \;\; 11 < L \le 450, \\ 19.5 & for \;\; L > 450. \end{cases} \tag{1}$$

The function $p^I(L,0)$ defines the limit of percent sequence identity for clearly homologous protein sequences, as a function of the alignment length L.

To construct the midnight ASTRAL set, representative domains must be selected for each of the 2845 SCOP families belonging to true classes. Individual families can harbour as few as one domain and as many as 1927 domains. We have evaluated two methods for selecting saHMM-members into the midnight ASTRAL set. Both methods are modified versions of the algorithms described by Hobohm *et al.* [4]. The algorithms select, for each SCOP family, only those domains that were determined by X-ray crystallography to a resolution of 3.6 Å or better, and have mutual sequence identities equal to or less than $p^I(L,0)$.

Within each family we construct pairwise structural superimpositions in order to obtain the percent sequence identities. The coordinate files of the domains are obtained from the ASTRAL compendium [2] corresponding to SCOP version 1.69 [9]. We have evaluated several structure alignment programs, and found that, currently, MUSTANG [5] results in the best performing saHMMs (to be published elsewhere). In case the program fails to align two structures, the pair of domains is treated like a pair with too high sequence identity. As a minimum requirement for building an saHMM, the SCOP domain family must be represented by at least two structures. Therefore, all families with only one representative were excluded from the midnight ASTRAL set.

All computations were done in parallel, using up to 20 processors on the HPC2N Linux cluster Seth. The compute nodes on Seth are AMD Athlon MP2000+ with 1GB of memory per dual node, connected in a high-speed SCALI network.

2.1 Algorithm 1 for Selecting saHMM-Members

Algorithm 1 is designed to limit the number of structural comparisons. It works by removing one of the domains in a pair from further consideration, if the mutual sequence identity falls above $p^I(L, 0)$.

Outline of Algorithm 1

```
1. Collect all family members with < 3.6 Å resolution into to-be-checked set.
2. Take domain d1 from to-be-checked set, place in select set.
3. For each other domain d2 in to-be-checked set.
   (a) Pairwise structural alignment of d1 and d2 to determine sequence identity $sI$ and alignment length $L$.
   (b) If $sI > p^I(L, 0)$ then dToRemove = selectOne(d1,d2).
       i. place dToRemove in to-remove set.
       ii. if dToRemove = d1 repeat from 2.
4. Repeat from 2 until no more domains remain in to-be-checked set.
```

In order to retain the highest quality structures for constructing optimal structure superimpositions as the basis for the saHMMs, the algorithm selects the domain with the better resolution. In cases where the resolution values of the structures to be compared are too similar, i.e., they differ by less than 10% of their average, we exclude the domain with the higher mean thermal factor, B-factor. This rule applies in particular to domains extracted from the same PDB (Protein Data Bank) file. The mean B-factor reflects the data quality and is here calculated as the arithmetic mean of the B-factors for all C_α atoms within the domain. The function `selectOne` is used to select which domain to remove in case of high sequence identity.

Outline of function `selectOne`

```
1. Read in domains to compare: d1 and d2
2. if |resolution(d1) - resolution(d2)| < 0.1·mean(resolution(d1), resolution(d2))
   (a) if the mean B-factor for d1 is smaller than the mean B-factor of d2, then set dToRemove = d2
   (b) else set dToRemove = d1
3. else if resolution of d2 is poorer than that of d1, then set dToRemove = d2
4. else set dToRemove = d1
```

After the first round of selection, all the preliminary discarded protein domains stored in the to-remove set are again compared to all domains in the select set, in order to assure that only domains with sequence identities above $p^I(L,0)$ are permanently discarded. The rationale behind this additional step is that in the process of removing domains, it is possible that a domain A is removed due to high sequence identity to domain B. If B is later removed due to high sequence identity to domain C, it could be that A and C have low mutual sequence identity. Thus A must be compared with C, and in case the identity is equal to or less than $p^I(L,0)$ both A and C must be kept.

2.2 Algorithm 2 for Selecting saHMM-Members

We evaluated a second algorithm, called Algorithm 2, which is designed to maximize the number of representative domains. Using Algorithm 2, one first fills an $n \times n$ score matrix M based on all-against-all structural comparisons of all n members within a particular SCOP family. An entry M_{ij} is a measure of the level of sequence identity and the relative data quality of domains d_i and d_j, and is defined as:

$$M_{ij} = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{if} \quad sI \le p^I(L,0), \\ 1 + 1/n & \text{if} \quad d_j = \texttt{dToRemove}, \\ 1 - 1/n & \text{if} \quad d_i = \texttt{dToRemove}. \end{cases} \tag{2}$$

Which domain to remove in case of too great sequence identity is determined using the same procedure `selectOne` as described for Algorithm 1. To select representative domains using M, we remove in each step the domain similar to most other domains, until no more similarities can be detected. The domain d_k, corresponding to row index k in M, which is similar to most other domains is the one with the highest row sum:

$$k = \text{argmax}_i\left(\sum_j M_{ij}\right). \tag{3}$$

Removing the domain d_k from the set corresponds to setting elements $M_{ki} = 0$ and $M_{ik} = 0$ for all i, including the diagonal element M_{kk}. The process is finished when $\max_i(\sum_j M_{ij}) = 1$. The representative domains are those with 1 on the diagonal ($M_{yy} = 1$ for all representatives y). For reasons described in Algorithm 1, all removed domains are checked once more against all selected domains to make sure that no representatives were mistakenly discarded.

2.3 Comparing Algorithm 1 and Algorithm 2

Calculations using Algorithm 1 result in 3129 domains in the midnight ASTRAL set, representing 850 different SCOP domain families. These families cover 65% of the SCOP domains and correspond to 30% of the SCOP families belonging to true classes. Algorithm 2 gives 3293 domains in the midnight ASTRAL set, which represent 894 SCOP domain families. These families cover about 60% of SCOP domains and correspond to 31% of the true class SCOP families.

The advantage of Algorithm 2 is that it produces more saHMM-members for the midnight ASTRAL set. However, it is time expensive due to the all-against-all structural comparisons, which cause the problem to scale quadratically with the number of domains. It was not practical to use Algorithm 2 for the four very largest families, each harbouring more than 600 domains. Even so, the computing time used to select representative domains with Algorithm 2 exceeded the total time used by Algorithm 1 by an order of magnitude. We therefore decided against Algorithm 2, and will from now on use Algorithm 1 to select saHMM-members, even though Algorithm 1 results in a slightly reduced coverage of SCOP families.

2.4 Analysis of the Midnight ASTRAL Set

In Fig. 1(a) the distribution of lengths of domains within the midnight ASTRAL set selected with Algorithm 1 is displayed. The sharp peak shows that the most common sequence length of the saHMM-members is about 100 residues. The length varies from 21 amino acids for the shortest domain up to 1264 residues for the longest. In Fig. 1(b) the distribution of resolutions at which the structures of the domains were determined is displayed. The majority of the crystal structures from which the domains are extracted fall into the resolution range between 1.5 to 2.5Å. This assures a high confidence in the determined structures.

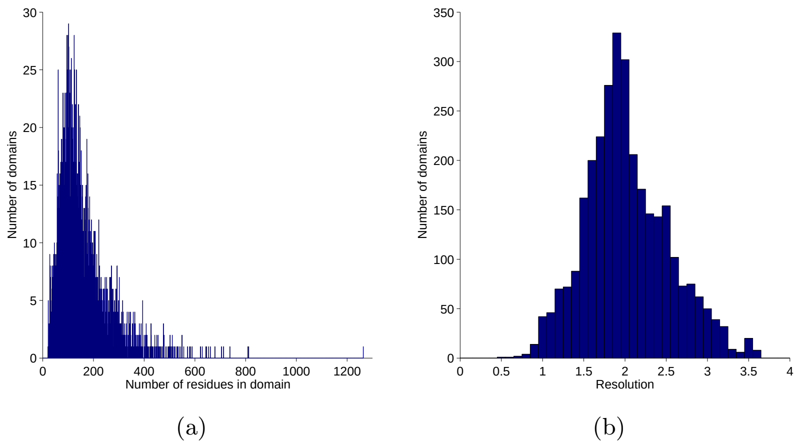


Fig. 1. Distribution of (a) sequence lengths and (b) resolutions among domains in the midnight ASTRAL set

3 The saHMM Data Base

The construction of structure-anchored Hidden Markov Models, saHMMs, requires three major steps. First, the non-redundant midnight ASTRAL set must be generated as was described above. Then a multiple 3D superimposition of the peptide chains of these domains, called the saHMM-members, is constructed. By using only spatial criteria to compare their structures, it is possible to match those amino acids that are from different chains and in close spatial vicinity, into a structure anchored multiple sequence alignment (see also [12]). The final step involves building the saHMMs from the deduced structure-anchored multiple sequence alignment.

The coordinate files of the saHMM-members are obtained from the ASTRAL compendium corresponding to SCOP version 1.69. The domains are superimposed with MUSTANG [5] and the saHMMs are built using HMMER 2.2g [3].

We implemented several Perl programs in order to automate the process from raw SCOP family classification of domains, through the construction of the midnight ASTRAL set, to the creation and testing of the saHMMs. The programs perform tasks such as detecting and correcting inconsistencies between the notations used in SCOP and the ASTRAL coordinate files, standardizing the notation used in the coordinate files and parsing of results to convert output from one program to input for another.

3.1 Coverage of SCOP

Since at least two structures are needed for superimposition, and because of the stringent sequence identity restrictions, our collection of saHMMs currently includes 850 saHMMs, which cover about 30% of the 2845 SCOP families belonging to true classes and 65% of the 67210 domain sequences. We expect these numbers to improve due to the exponential increase of deposited 3D structures.

4 The FISH Server

4.1 Design of the FISH Server

Flat file data bases were imported into a relational data base (MySQL implemented on a Linux platform) and cross-linked (Fig. 2). The user interface is written in Perl, PHP, and JavaScript and integrated with the Apache web server.

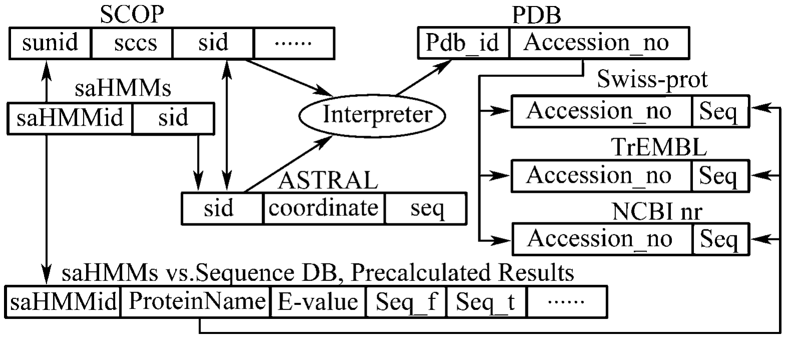


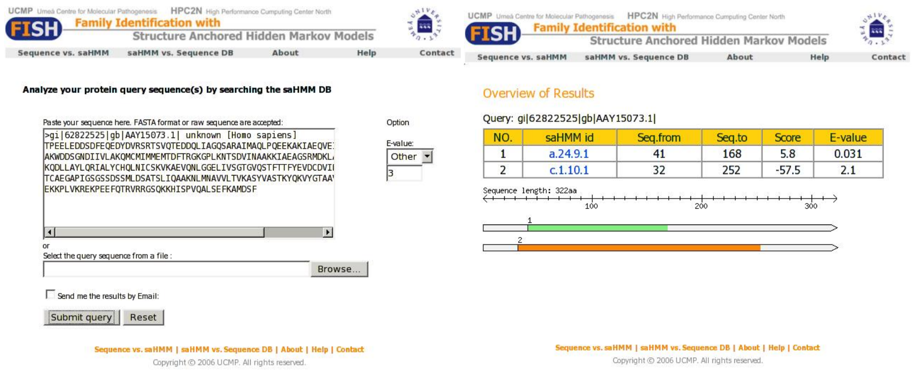
Fig. 2. Schematic view of the data base cross-linking used in the FISH server

The user inputs a query via the web interface. The query interpreter analyzes the input, using the collection of saHMMs. The cross-link engine merges information from the associated data bases with the results of the query. The results assembler presents the outcome of the search to the user via the web interface. The search results can also be sent to the user by e-mail in the form of a www-link and are stored on the server for 24 hours.

4.2 How to Use the FISH Server

Sequence Searches vs. the saHMM-db

Using the FISH server, a user can compare a query sequence with all models in the saHMM-db. Matches obtained in such a search provide the user with a classification on the SCOP family level and outline structurally defined, putative domain boundaries in the query sequence. This information is useful for sequence annotation, to design mutations, to identify soluble domains, to find structural templates for homology modelling and possibly for structure determination by molecular replacement.


UCMP Umeå Centre for Molecular Pathogenesis
HPC2N High Performance Cumputing Center North
FISH Family Identification with Structure Anchored Hidden Markov Models
Sequence vs. saHMM
saHMM vs. Sequence DB
About
Help
Contact

Analyze your protein query sequence(s) by searching the saHMM DB

Paste your sequence here. FASTA format or raw sequence are accepted:

```
>gi|62822525|gb|AAY15073.1| unknown [Homo sapiens]
TPEELEDDSDFEQEDYDVRSRTSVQTEDDQLIAGQSARAIMAQLPQEEKAKIAEQVE
AKWDDSGNDIIVLAKQMCMIMMEMTDFTRGKGPLKNTSDVINAAKKIAEAGSRMDKL
KQDLLAYLQRIALYCHQLNICSKVKAEVQNLGGELIVSGTGVQSTFTTFYEVDCDVI
TCAEGAPIGSGSSDSSMLDSATSLIQAAKNLMNAVVLTVKASYVASTKYQKVYGTAA
EKKPLVKREKPEEFQTRVRRGSQKKHISPVQALSEFKAMDSF
```

Option

E-value: Other

3

or

Select the query sequence from a file :

☐ Send me the results by Email:

Browse...
Submit query
Reset
Sequence vs. saHMM | saHMM vs. Sequence DB | About | Help | Contact
Copyright © 2006 UCMP. All rights reserved.

(a)

Overview of Results

Query: gi|62822525|gb|AAY15073.1|

NO.	saHMM id	Seq.from	Seq.to	Score	E-value
1	a.24.9.1	41	168	5.8	0.031
2	c.1.10.1	32	252	-57.5	2.1

Sequence length: 322aa

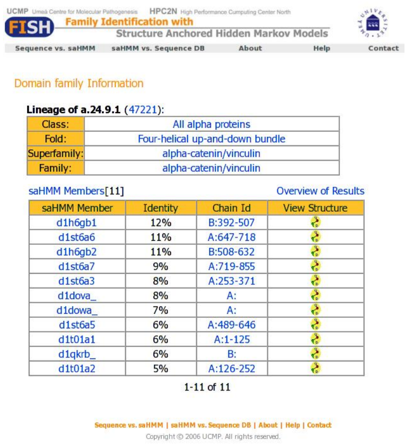
100
200
300
1
2
Sequence vs. saHMM | saHMM vs. Sequence DB | About | Help | Contact
Copyright © 2006 UCMP. All rights reserved.

(b)

Fig. 3. Sample (a) input and (b) results pages from a sequences vs saHMMs search

Fig. 3(a) displays an example of the input page. The user enters one or more query sequences and can select an E-value cut-off for the results. The E-value of a hit is the expected number of false matches having at least the same score as the hit, and hence is a measure of the confidence one can have in the hit. The closer the E-value is to zero, the more the match can be trusted. In the 'overview of results' page (Fig. 3(b)) the list of matches is sorted in increasing order with respect to the E-value, up to the chosen cut-off. When selecting one entry from the list, the family specific information for that match is displayed (Fig. 4(a)). The top table provides information about the SCOP classification. It is followed by a table listing all saHMM-members of this family together with details about, for example, the percent sequence identity of the query sequence aligned to the member. For each saHMM-member, it is possible to view the structure of the selected domain in an interactive Java window, as shown in Fig. 4(b).

Below the list of matches in the 'overview of results' page (Fig.3(b)) is a horizontal bar graph representation of the query sequence, where matches are marked as coloured ranges. A light green range corresponds to an E-value of 0.1 or less, a yellow range to $0.1 \leq$ E-value ≤ 1.0 and an orange range for E-values above 1.0. Each coloured range links to a pairwise alignment of the query sequence and the saHMM consensus. The user has the option to display a multiple sequence alignment of the query sequence and the saHMM-member sequences in different formats. In addition, it is possible to reach a list with pairwise comparisons of the query and each saHMM-member. All alignments are anchored on the saHMM.


UCMP Umeå Centre for Molecular Pathogenesis
HPC2N High Performance Computing Center North
FISH
Family Identification with
Structure Anchored Hidden Markov Models
Sequence vs. saHMM
saHMM vs. Sequence DB
About
Help
Contact

Domain family Information

Lineage of a.24.9.1 (47221):

Class:	All alpha proteins
Fold:	Four-helical up-and-down bundle
Superfamily:	alpha-catenin/vinculin
Family:	alpha-catenin/vinculin

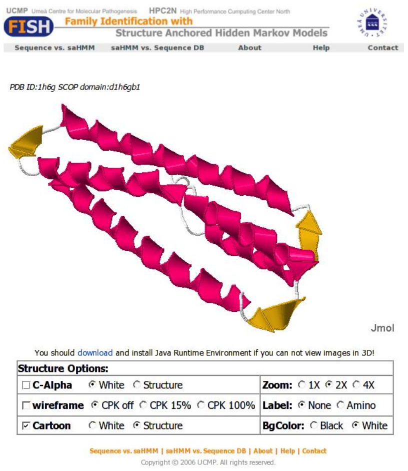
saHMM Members[11] Overview of Results

saHMM Member	Identity	Chain Id	View Structure
d1h6gb1	12%	B:392-507	
d1st6a6	11%	A:647-718	
d1h6gb2	11%	B:508-632	
d1st6a7	9%	A:719-855	
d1st6a3	8%	A:253-371	
d1dova_	8%	A:	
d1dowa_	7%	A:	
d1st6a5	6%	A:489-646	
d1t01a1	6%	A:1-125	
d1qkrb_	6%	B:	
d1t01a2	5%	A:126-252	

1-11 of 11

Sequence vs. saHMM | saHMM vs. Sequence DB | About | Help | Contact
Copyright © 2006 UCMP. All rights reserved.

(a)


UCMP Umeå Centre for Molecular Pathogenesis
HPC2N High Performance Computing Center North
FISH
Family Identification with
Structure Anchored Hidden Markov Models
Sequence vs. saHMM
saHMM vs. Sequence DB
About
Help
Contact
Jmol

PDB ID:1h6g SCOP domain:d1h6gb1

You should download and install Java Runtime Environment if you can not view images in 3D!

Structure Options:	
☐ **C-Alpha** [✓ White] Structure	**Zoom:** 1X [✓ 2X] 4X
☐ **wireframe** [✓ CPK off] CPK 15% CPK 100%	**Label:** [✓ None] Amino
☑ **Cartoon** White [✓ Structure]	**BgColor:** Black [✓ White]

Sequence vs. saHMM | saHMM vs. Sequence DB | About | Help | Contact
Copyright © 2006 UCMP. All rights reserved.

(b)

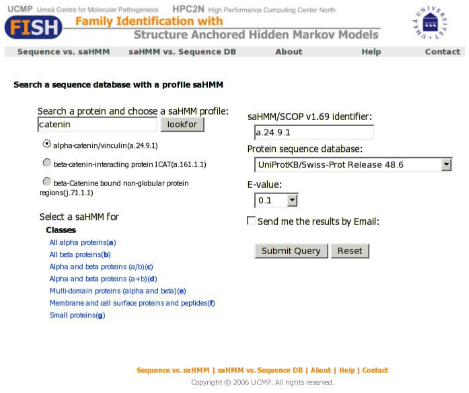
Fig. 4. Example pages displaying (a) the domain family information of the top hit from Fig. 3(b) and (b) the structure view of the domain with highest sequence identity compared to the query sequence

saHMM Searches vs. a Sequence Database

Furthermore, the FISH server allows the user to employ individual saHMMs for searching against a sequence data base to find those proteins that harbour a certain domain, independent of sequence identity and annotation status. For this purpose, the user can choose a particular saHMM from a list of available models and specify against which data base to perform the search. Currently, the Swiss-Prot, TrEMBL and the non-redundant data base, nr, from NCBI are available for searching. In addition, a user has the option to upload his/her own sequence database, as long as its size does not exceed 2 MB. In this way it is possible to identify previously un-annotated sequences on the domain family level, even in case of very low sequence identities, below $p^I(L, 0)$. For each match, the user obtains the corresponding sequence entry, as well as pairwise and multiple sequence alignments of the matched sequence and the saHMM-members, anchored on the saHMM. Information about the domain family used for searching is also easily available.

A search with a single saHMM vs. SwissProt can take from 15 minutes up to about nine hours. Searching TrEMBL, which is about ten times larger, takes considerably longer. In order to minimize the time a user has to wait for the results, we pre-calculated the searches of all 850 saHMMs vs. SwissProt, TrEMBL and nr using an E-value cut-off of 100. Depending on the E-value choice of the user, the results are extracted and presented up to that value. The computations were done in parallel, by searching the databases with several saHMMs concurrently, using up to 20 processors on the HPC2N Linux cluster Seth.

Fig. 5 shows an example of (a) the input page and (b) the results page of a search with an saHMM versus a sequence database. In the example, SwissProt was used. The results of the search are represented in form of a list sorted by E-value up to the user-specified cut-off.


UCMP Umeå Centre for Molecular Pathogenesis
HPC2N High Performance Computing Center North
FISH
Family Identification with Structure Anchored Hidden Markov Models
Sequence vs. saHMM
saHMM vs. Sequence DB
About
Help
Contact
lookfor
Submit Query
Reset
Sequence vs. saHMM | saHMM vs. Sequence DB | About | Help | Contact
Copyright © 2006 UCMP. All rights reserved.

Search a sequence database with a profile saHMM

Search a protein and choose a saHMM profile: catenin

(✓) alpha-catenin/vinculin(a.24.9.1)

( ) beta-catenin-interacting protein ICAT(a.161.1.1)

( ) beta-Catenine bound non-globular protein regions(j.71.1.1)

Select a saHMM for
Classes
- All alpha proteins(**a**)
- All beta proteins(**b**)
- Alpha and beta proteins (a/b)(**c**)
- Alpha and beta proteins (a+b)(**d**)
- Multi-domain proteins (alpha and beta)(**e**)
- Membrane and cell surface proteins and peptides(**f**)
- Small proteins(**g**)

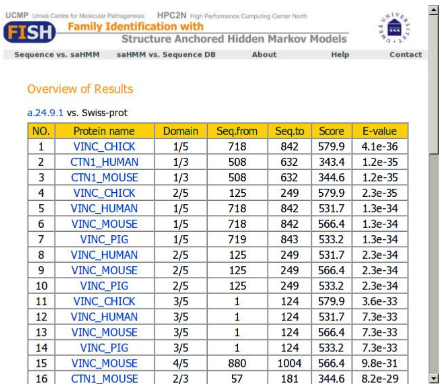
saHMM/SCOP v1.69 identifier: a.24.9.1

Protein sequence database: UniProtKB/Swiss-Prot Release 48.6

E-value: 0.1

☐ Send me the results by Email:

(a)


UCMP Umeå Centre for Molecular Pathogenesis
HPC2N High Performance Computing Center North
FISH
Family Identification with Structure Anchored Hidden Markov Models
Sequence vs. saHMM
saHMM vs. Sequence DB
About
Help
Contact

Overview of Results

a.24.9.1 vs. Swiss-prot

NO.	Protein name	Domain	Seq.from	Seq.to	Score	E-value
1	VINC_CHICK	1/5	718	842	579.9	4.1e-36
2	CTN1_HUMAN	1/3	508	632	343.4	1.2e-35
3	CTN1_MOUSE	1/3	508	632	344.6	1.2e-35
4	VINC_CHICK	2/5	125	249	579.9	2.3e-35
5	VINC_HUMAN	1/5	718	842	531.7	1.3e-34
6	VINC_MOUSE	1/5	718	842	566.4	1.3e-34
7	VINC_PIG	1/5	719	843	533.2	1.3e-34
8	VINC_HUMAN	2/5	125	249	531.7	2.3e-34
9	VINC_MOUSE	2/5	125	249	566.4	2.3e-34
10	VINC_PIG	2/5	125	249	533.2	2.3e-34
11	VINC_CHICK	3/5	1	124	579.9	3.6e-33
12	VINC_HUMAN	3/5	1	124	531.7	7.3e-33
13	VINC_MOUSE	3/5	1	124	566.4	7.3e-33
14	VINC_PIG	3/5	1	124	533.2	7.3e-33
15	VINC_MOUSE	4/5	880	1004	566.4	9.8e-31
16	CTN1_MOUSE	2/3	57	181	344.6	8.2e-29

(b)

Fig. 5. Example of (a) input and (b) results of a search with the catenin saHMM (a.24.9.1) vs SwissProt version 1.69. Only the top part of the results page is shown.

5 Conclusions

The foundation of the structure-anchored hidden Markov model method is the 3D superimposition of carefully chosen domains representing the SCOP domain family to be modelled. For the selection of the representative domains, called the saHMM-members, we evaluated two algorithms, Algorithm 1 and Algorithm 2. Even though the use of Algorithm 2 results in 164 more saHMM-members in the midnight ASTRAL set, which leads to 44 more saHMMs, we prefer Algorithm 1 since it is more than an order of magnitude faster and can handle even the largest families in a reasonable amount of time. The resulting saHMMs together constitute the saHMM-db, which covers 30% of the SCOP families and

65% of the domains belonging to true classes. So far, every new SCOP release has lead to new saHMMs and has increased the number of saHMM-members for many families. As the number of deposited structures grows, we anticipate that the saHMM-db will cover more of SCOP. In addition, we expect that new domain sequences will be added to families, which in turn increases the number of saHMM-members and improve saHMMs with only few saHMM-members. The saHMM-db is publicly available through the FISH server, which is a powerful and versatile tool with dual function. On the one hand, the user can perform sequence searches versus the saHMM-db, and possibly obtain matches even for remote homologues, within the "midnight zone" of sequence alignments. On the other hand, the user can choose one of the saHMMs to perform a search against a protein sequence data base. Since the saHMMs are based on structure anchored sequence alignments and the structures of all representatives are known, the alignment of a sequence to the saHMM-members can be used to draw conclusions about the secondary and tertiary structures of the sequence.

Acknowledgements. We thank Åke Sandgren for assistance with parallel computing and Marek Wilczynski for support with the FISH server. Part of this research was conducted using the HPC2N cluster resources. B.K. acknowledges the partial support for this project by the Swedish Foundation for Strategic Research grant A3.02:13. U.H.S. acknowledges the partial support for this project by a grant from the Knowledge Foundation (KK-Stiftelsen).

References

1. Bateman, A., Coin, L., Durbin, R., Finn, R.D., Hollich, V., Griffiths-Jones, S., Khanna, A., Marshall, M., Moxon, S., Sonnhammer, E.L.L., Studholme, D.J., Yeats, C., Eddy, S.R.: The Pfam protein families database. Nucleic Acids Research 32, D138–D141 (2004)
2. Chandonia, J.-M., Hon, G., Walker, N.S., Lo Conte, L., Koehl, P., Levitt, M., Brenner, S.E.: The ASTRAL Compendium in 2004. Nucleic Acids Research 32, D189–D192 (2004)
3. Eddy, S.R.: Profile Hidden Markov Models. Bioinformatics 14, 755–763 (1998)
4. Hobohm, U., Scharf, M., Schneider, R., Sander, C.: Selection of representative protein data sets. Protein Science I, 409–417 (1992)
5. Konagurthu, A.S., Whisstock, J.C., Stuckey, P.J., Lesk, A.M.: MUSTANG: A multiple structural alignment algorithm. PROTEINS: Structure, Function, and Bioinformatics 64, 559–574 (2006)
6. Letunic, I., Copley, R.R., Pils, B., Pinkert, S., Schultz, J., Bork, P.: SMART 5: domains in the context of genomes and networks. Nucleic Acids Research 34, D257–D260 (2006)
7. Madera, M., Vogel, C., Kummerfeld, S.K., Chothia, C., Gough, J.: The SUPERFAMILY database in 2004: additions and improvements. Nucleic Acids Research 32, D235–D239 (2004)
8. Mika, S., Rost, B.: UniqueProt: creating representative protein sequence sets. Nucleic Acids Research 31, 3789–3791 (2003)

9. Murzin, A.G., Brenner, S.E., Hubbard, T., Chothia, C.: SCOP: a structural classification of proteins database for the investigation of sequences and structures. Journal of Molecular Biology 247, 536–540 (1995)
10. Rost, B.: Twilight zone of protein sequence alignments. Protein Engineering 12, 85–94 (1999)
11. Russell, R.B., Barton, G.J.: Multiple Protein Sequence Alignment From Tertiary Structure Comparison: Assignment of Global and Residue Confidence Levels. PROTEINS: Structure, Function, and Genetics 14, 309–323 (1992)
12. Tångrot, J.: The Use of Structural Information to Improve Biological Sequence Searches. Lic. Thesis, UMINF-03.19. Dept. of Comput. Sci., Umeå Univ. (2003)
13. Tångrot, J., Wang, L., Kågström, B., Sauer, U.H.: FISH – family identification of sequence homologues using structure anchored hidden Markov models. Nucleic Acids Research 34, W10–W14 (2006)

Scientific Visualization and HPC Applications: Minisymposium Abstract

Matt Cooper and Anders Ynnerman

Linköping University, Sweden

High Performance Computing (HPC) produces enormous amounts of data. This simple truth has been the perennial bane of the HPC user and there is no sign of the problem going away. The results of the computational process are often large data sets in the form of molecular structures and property fields, fluid density and velocity fields, particle positions and momenta or any of a diverse host of other types all sharing the single property that they are large and so difficult to interpret. In the case of many computational methods it is, in addition, often useful to retain state information, perhaps as large as the final output, from each step of the computational process for a post-mortem analysis of the optimization or for computational steering. The size of the data produced often scales with the problem size, the problem size typically increases with the available computational power and so the ever-growing improvement in computational power is likely to continue to make this problem more difficult as time progresses.

To aid in the analysis of complex multidimensional, multivariate and often time-varying data, visualization systems exploiting computer graphics and complex interaction mechanisms are becoming required tools but the size of these data sets presents unique problems in the efficient processing and graphical rendering of representations of the data which will permit their interpretation.

This mini-symposium brings together researchers and developers from a range of disciplines, both experts in the scientific fields which produce the data for interpretation and experts in the techniques which are being used to provide the visualization tools for this work. We hope this diverse panel will stimulate an interesting and informative discussion with the audience, giving some ideas about how visualization is likely to change in the future and how visualization systems need to change to meet the needs of the user community.

Interactive Volume Visualization of Fluid Flow Simulation Data

Paul R. Woodward, David H. Porter, James Greensky, Alex J. Larson, Michael Knox, James Hanson, Niranjay Ravindran, and Tyler Fuchs

University of Minnesota, Laboratory for Computational Science & Engineering, 499 Walter Library, 117 Pleasant St. S. E., Minneapolis, Minnesota 55455, U.S.A.
{paul, dhp, mikeknox}@lcse.umn.edu, {jjgreensky, jphansen}@gmail.com, {lars1671, ravi0022, fuch0057}@umn.edu

Abstract. Recent development work at the Laboratory for Computational Science & Engineering (LCSE) at the University of Minnesota aimed at increasing the performance of parallel volume rendering of large fluid dynamics simulation data is reported. The goal of the work is interactive visual exploration of data sets that are up to two terabytes in size. A key system design feature in accelerating the rendering performance from such large data sets is replication of the data set on directly attached parallel disk systems at each rendering node. Adaptation of this system for interactive steering and visualization of fluid flow simulations as they run on remote supercomputer systems introduces special additional challenges which will briefly be described.

Keywords: scientific visualization, interactive steering, fluid flow simulation.

1 Introduction

The visualization of fluid flow simulations using perspective volume rendering provides a very natural means of understanding flow dynamics. This is especially true in flow simulations, like those in astrophysics, that do not involve complex confining walls and surfaces. However, volume rendering is data intensive, and therefore even a modest flow simulation can easily result in quite a large amount of data to be visualized. Our team at the University of Minnesota's Laboratory for Computational Science & Engineering (LCSE) has been exploiting volume rendering for fluid flow visualization for many years. In this paper we report our most recent efforts to accelerate this process so that even very large data sets can be interactively explored at full PowerWall resolution. Our earlier efforts at full resolution visualization were based principally on off-line generation of movie animations. The challenge of speeding up perspective volume visualization has been addressed by several groups. Like other teams (see, for example, [1-2] and references therein), our hierarchical volume rendering software HVR (www.lcse.umn.edu/hvr) utilizes a multiresolution, tree-based data format to render only coarsened data when there is little variation within a region or when the region occupies very few pixels. Like many other groups we use PC graphics engines to accelerate the parallel ray tracing calculations that we long ago performed on vector CPUs like the Cray-2, then later implemented on SGI Reality Engines. This is in contrast to ray tracing on large parallel machines, as described in [4], which can be interactive when the data fits into the memory. Unlike more comprehensive graphics software packages like VisIt and SciRun [5], we concentrate exclusively on volume rendering at the LCSE and focus on the special problems that come with very large, multi-terabyte data sets that do not fit into the memories of essentially any available rendering resource. This focus has led us to our present system design, which exploits full data set replication at multiple rendering nodes as a key performance enabling feature.

2 Interactive Volume Rendering of Multi-TB Data

To set up for a session of data exploration, we generally decide upon a limited set of variables we think will be useful and then process the sequence of compressed dumps from the simulation to create "HV-file" sequences for these variables that incorporate the multiresolution data structure that our renderer, HVR, requires for performance. A medium-scale run, which we used to demonstrate our system at the IEEE Visualization conference in October, 2005, in an exhibit booth sponsored by our industrial partner, Dell, (see Figure 1) serves as a good example. From this run, which took about 6 weeks on a 32-processor Unisys ES7000 in our lab, we saved about 500 compressed dumps of about 10 GB each. It took a few days to process these dumps into sets of 1.27 GB HV-files for the vorticity, the divergence of velocity, and the entropy. These HV-files thus constituted a data set of about 1.9 TB. This data set was sufficiently small that we could place a full copy of it at each node of our 14-node rendering cluster. Replication of the data to be explored is a key element of our system design. Each node has a directly attached set of twelve 400 GB SATA disks striped using a 3ware 12-channel disk controller and delivering up to 400 MB/sec sustained data transfers. The capacity at each node therefore allows 2 TB to be declared as scratch space and used for replication of whatever data might be under study at the time. Fourteen different 2-TB data sets can be held on the system while a user explores one of them replicated on all nodes. Our experience with building inexpensive fast disk systems over many years indicates that data replication is the only surefire way to guarantee the fast, sustained data delivery on every node from a shared data set that is required for interactive volume rendering. This system design allows each local disk subsystem to stream data unimpeded by requests from other nodes for different portions of the shared data. Locally attached disks are also considerably less expensive than storage area networks of dedicated storage servers.

Although every node has an entire copy of the data under study, a node needs at any one time to read only a small portion of it. Each HV-file has an octree structure of voxel bricks, which overlap in order to enable seamless image ren-

Fig. 1. Jim Greensky and David Porter control 8 nodes of our system at IEEE Vis2005

dering. Although our software enables an initial rendering in low resolution to be continuously improved, this strategy is not well suited to our large PowerWall display, shown in Figure 2, which measures 25 feet across. The resolution of an image rendering is very obvious on a display of this size. Therefore, on the PowerWall we generally would prefer to continue looking at the previous high quality rendering while waiting for the next high quality image to appear. Our solution is thus to optimize the throughput of the pipeline from data on disk through to the image on the screen with multiple renderers assigned their own segments of the final image. In order to speed up the rate at which voxel bricks can be loaded as 3-D textures into our Nvidia graphics cards, we have decreased the size of these bricks from our previous standard of 128^3 to 64^3 voxels. Although this caused the data streaming rate from disk to drop by about 20%, the overall system pipeline speed increased. The smaller voxel bricks do, however, increase the efficiency of the parallel rendering by allowing each HVR volume rendering server to read and render a section of the flow more closely approximating that contained in its viewing frustum. For true interactive visualization of our example 1.9 TB data set, the time to produce a new, full resolution PowerWall image from data not in memory but on disk is the most essential measure of success. Ten nodes of our cluster can produce a 13 Mpixel PowerWall image of a billion-voxel data snap shot in just over 1 second. If we then choose to look at this same

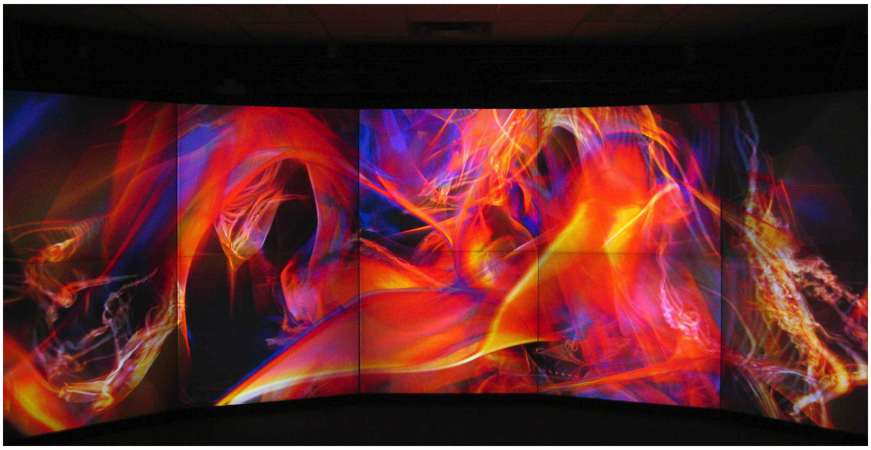
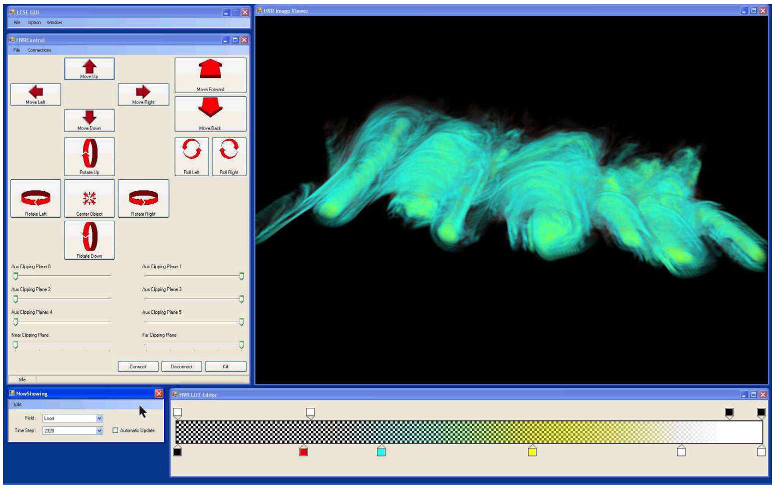


Fig. 2. A rendering of our decaying Mach 1 turbulence vorticity data on the PowerWall

data from a different viewing position and/or with a different color and opacity mapping, we can generate the next image much faster, since our software takes advantage of the voxel bricks that are already loaded into each graphics card. We are now experimenting with increasing the interactivity by concentrating our 14 nodes of rendering power on less than the full 10 image panels of our PowerWall. This allows us to trade off interactivity with image size and resolution.

3 Remote Flow Visualization and Simulation Steering

Our software (see www.lcse.umn.edu/hvr) supports the visualization cluster members rendering portions of an image that they send to a remote node to be stitched together and displayed on a single screen. This rendering is generally faster, since the screen has many fewer pixels than the PowerWall, and it is therefore more interactive. This mode was used in the exhibit booth at IEEE Vis2005, as shown in Figure 1. Just 8 of our 14 nodes were able to render images from billion-voxel HV-files on disk at 1920×1200 resolution in about 0.7 sec. Later in the year, we combined this capability with a high performance implementation of our PPM gas dynamics code running on 512 or 1024 CPUs of the Cray XT3 at the Pittsburgh Supercomputer Center (PSC) to enable interactive steering and visualization of an ongoing gas dynamics simulation. We worked with the PSC team, and in particular Nathan Stone, who provided a utility, PDIO [6], that allowed us to write tiny quarter-MB files from each of the 512 CPUs on the machine that PDIO managed and transported to a node of our visualization cluster at the LCSE in Minnesota. Our software then constructed HV-files from the stream of little files, which landed on a RAM disk, and broadcast the HV-files to 10 rendering nodes connected to the PowerWall. However, the rendering nodes could also transmit portions of images to another machine located anywhere on the Internet, and that machine could display the result on its monitor, as shown in Figure 3. Figure 3 shows the image viewer pane, the opacity and color map control pane, the field (i.e. fluid state variable) and time level control pane, and viewpoint and clipping plane control pane. A separate GUI controls the simulation on the Cray XT3, which the user can pause, abort, restart, or instruct to alter the frequency of data file output. Running on 1024 CPUs at a little over 1 Tflop/s, this application computes a flow simulation of shear instability, as shown, to completion in about 20 minutes. A reenactment of this TeraGrid Conference demonstration shown in Figure 3 using the actual data generated by the XT3 at the event arriving according to its actual time stamps (every 10 time steps, or about every 3 to 5 seconds, new voxel data arrived in the LCSE) and showing the interactive control can be downloaded as a Windows .exe file from www.lcse.umn.edu/TeraGridDemo.



Field : Lvort
Time Step : 2320
☐ Automatic Update

Fig. 3. LCSE-PSC demonstration of interactive supercomputing, TeraGrid Conf

Grid resolutions of 512^3 cells enable 30 to 40 minute simulations when computing with PPM at 1 Tflop/s, and this is a good length of time for an interactive session. The PDIO software enabled us to obtain 55 MB/sec throughput on the University of Minnesota's Internet connection during the day and almost twice this much at night. This allows the transmission of a 128 MB voxel snap shot of the flow in a single fluid state variable in about 3 seconds. While a new snap shot is being transmitted, the interactive capabilities of our visualization cluster allow the user to view the previous snap shot from different angles and with different color and opacity settings using the control panes shown in Figure 3. The user may also explore the data that has accumulated to this point. Responsiveness is good, because the data size is small. However, if one wishes to view a different fluid state variable from the one that was sent, one can only request that this be transmitted in subsequent snap shots; it can be made available at this same time level only if multiple variables are transmitted. This would be possible on a 10 Gbit/s connection.

As computer power and networking bandwidth grow, we anticipate that interactive flow visualization will be used more and more for steering simulations as they run. The data sets associated with runs that are short enough to be steered today are still under one TB, but as petascale computing systems are put into place this will change.

This work has been supported by NSF CISE RR (CNS-0224424) and MRI (CNS-0421423) grants, through grant DE-FG02-03ER25569 from the MICS program of the DoE Office of Science, through a donation of an ES-7000 computer from Unisys Corp., and by local support to the Laboratory for Computational Science & Engineering (LCSE) from the University of Minnesota's Digital Technology Center and Minnesota Supercomputer Institute. Our fluid dynamics simulations have also been supported by grants of time on the NSF TeraGrid cluster at NCSA and at the Pittsburgh Supercomputing Center. We gratefully acknowledge the work of Nathan Stone and Raghu Reddy at the Pittsburgh Supercomputer Center, who worked closely with us to make the data transfers from our fluid dynamics application on the Cray XT3 to a node of our local visualization cluster go smoothly and very fast using Stone's PDIO utility.

References

1. Ahearn, S., Daniel, J.R., Gao, J., Ostrouchov, G., Toedte, R.J., Wang, C.: Multiscale data visualization for computational astrophysics and climate dynamics at Oak Ridge National Laboratory. J. of Physics: Conf. Series 46, 550–555 (2006)
2. Shen, H.-W.: Visualization of large scale time-varying scientific data. J. of Physics: Conf. Series 46, 535–544 (2006)
3. LCSE volume rendering and movie making software is available at `http://www.lcse.umn.edu/hvr`
4. Parker, S., Parker, M., Livnat, Y., Sloan, P.-P., Hansen, C., Shirley, P.: Interactive ray tracing for volume visualization. IEEE Transactions on Visualization and Computer Graphics 5, 238–250 (1999)
5. Bethel, W., Johnson, C., Hansen, C., Parker, S., Sanderson, A., Silva, C., Tricoche, X., Pascucci, V., Childs, H., Cohen, J., Duchaineau, M., Laney, D., Lindstrom, P., Ahern, S., Meredith, J., Ostrouchov, G., Joy, K., Hamann, B.: VACET: Proposed SciDAC2 Visualization and Analytics Center for Enabling Technologies. J. of Physics: Conf. Series 46, 561–569 (2006)
6. Stone, N.T.B., Balog, D., Gill, B., Johanson, B., Marsteller, J., Nowoczynski, P., Porter, D., Reddy, R., Scott, J.R., Simmel, D., Sommerfield, J., Vargo, K., Vizino, C.: PDIO: High-Performance Remote File I/O for Portals-Enabled Compute Nodes. In: Proc, Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV (June 2006), also available at `http://www.psc.edu/publications/tech_reports/PDIO/PDIO-PDPTA06.pdf`
7. Nystrom, N., Weisser, D., Lim, J., Wang, Y., Brown, S. T., Reddy, R., Stone, N. T., Woodward, P.R., Porter, D.H., Di Matteo, T., Kale, L.V., Zheng, G.: Enabling Computational Science on the Cray XT3. In: Proc. CUG (Cray User Group) Conference, Zurich (May 2006)

Software Tools for Parallel CFD Applications: Minisymposium Abstract

Xing Cai and Hans Petter Langtangen

Simula Research Laboratory and Oslo University, Norway

Computational Fluid Dynamics (CFD) is an important research field with a broad spectrum, requiring a close interplay between mathematical modeling, numerical methods, software development, and high performance computing. Programming CFD applications is an inherently difficult task due to many factors related to numerics and software. For modern CFD applications, use of parallel computers is a must, which makes the CFD code development even more challenging. Rapid and flexible programming of a parallel CFD application calls for re-usable serial and parallel software components plus a modular overall parallel framework that allows an easy coupling between different components. Moreover, more attention needs to be put on securing the performance of the software components. This minisymposium thus attempts to shed some light into the recent developments in respect of CFD-related numerical methods, programming techniques and software libraries.

The Iterative Solver Template Library

Markus Blatt and Peter Bastian

Institut für Parallele und Verteilte Systeme,
Universität Stuttgart, Universitätstr. 38, 70569 Stuttgart, Germany
{Markus.Blatt, Peter.Bastian}@ipvs.uni-stuttgart.de
http://www.dune-project.org

Abstract. The numerical solution of partial differential equations frequently requires the solution of large and sparse linear systems. Using generic programming techniques in C++ one can create solver libraries that allow efficient realization of "fine grained interfaces", i.e. with functions consisting only of a few lines, like access to individual matrix entries. This prevents code replication and allows programmers to work more efficiently. We present the "Iterative Solver Template Library" (ISTL) which is part of the "Distributed and Unified Numerics Environment" (DUNE). It applies generic programming in C++ to the domain of iterative solvers of linear systems stemming from finite element discretizations. Those discretizations exhibit a lot of structure. Our matrix and vector interface supports a block recursive structure. Each sparse matrix entry can itself be either a sparse or a small dense matrix. Based on this interface we present efficient solvers that use the recursive block structure via template metaprogramming.

1 Introduction

The numerical solution of partial differential equations (PDEs) frequently requires solving large and sparse linear systems. Naturally, there are many libraries available for doing sparse matrix/vector computations, see [7] for a comprehensive list.

The widely available Basic Linear Algebra Subprograms (BLAS) standard has been extended to cover also sparse matrices [5]. The standard uses procedural programming style and offers only a FORTRAN and C interface. "Fine grained" interfaces, i.e. with functions consisting only of a few lines of code, such as access to individual matrix elements, impose an efficiency penalty here, as the relative cost for indirect function calls becomes huge.

Generic programming techniques in C++ or Ada offer the possibility to combine flexibility and reuse ("efficiency of the programmer") with fast execution ("efficiency of the program"). They allow the compiler to apply optimizations even for "fine grained" interfaces via static function typing. These techniques were pioneered by the Standard Template Library (STL), [16]. Their efficiency advantage for scientific C++ was later demonstrated by the Blitz++ library [6]. For an introduction to generic programming for scientific computing see [2,17].

Application of these ideas to matrix/vector operations is available with the Matrix Template Library (MTL), [13,15] and to iterative solvers for linear systems with the Iterative Template Library (ITL), [12].

In contrast to these libraries the "Iterative Solver Template Library" (ISTL), which is part of the "Distributed and Unified Numerics Environment" (DUNE), [3,8], is designed specifically for linear systems stemming from finite element discretizations. The sparse matrices representing these linear systems exhibit a lot of structure, e.g.:

– Certain discretizations for systems of PDEs or higher order methods result in matrices where individual entries are replaced by small blocks, say of size 2×2 or 4×4, see Fig. 1(a). Dense blocks of different sizes e.g. arise in *hp* Discontinuous Galerkin discretization methods, see Fig. 1(b). It is straightforward and efficient to treat these small dense blocks as fully coupled and solve them with direct methods within the iterative method, see e.g. [4].
– Equation-wise ordering for systems results in matrices having an $n \times n$ block structure where n corresponds to the number of variables in the PDE and the blocks themselves are large and sparse. As an example we mention the Stokes system, see Fig. 1(d). Iterative solvers such as the SIMPLE or Uzawa algorithm use this structure.
– Other discretizations, e.g. those of reaction/diffusion systems, produce sparse matrices whose blocks are sparse matrices of small dense blocks, see Fig. 1(c).
– Other structures that can be exploited are the level structure arising from hierarchical meshes, a p-hierarchical structure (e.g. decomposition in linear and quadratic part), geometric structure from decomposition in subdomains or topological structure where unknowns are associated with nodes, edges, faces or elements of a mesh.

Our library takes advantage from this natural block structure at compile time and supports the recursive block structuredness in a natural way.

Other libraries, like MTL, provide the blockings as views to the programmer. As this is done dynamically the block structure cannot be used efficiently in custom generic algorithms. In the Optimized Sparse Kernel Interface (OSKI), see [18], the sparse matrices are stored as scalar matrices, too. Here the user can

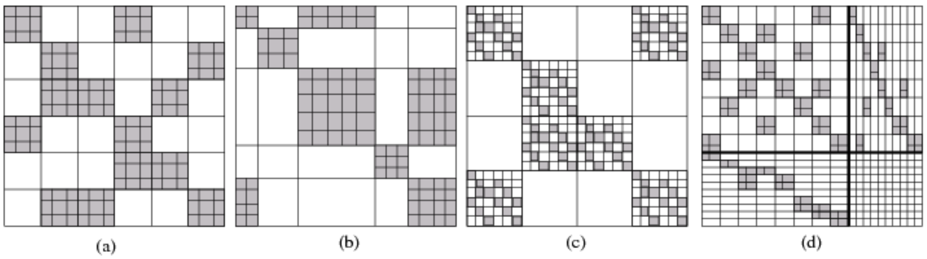


Fig. 1. Block structure of matrices arising in the finite element method

provide hints about the dense block sizes which are used at runtime to tune the solvers.

In the next section we describe the matrix and vector interface that represents this recursive block structure via templates. In Sect. 3 we show how to exploit the block structure using template metaprogramming at compile time. Finally we sketch the high level iterative solver interface in Sect. 4.

2 Matrix and Vector Interface

The interface of our matrices is designed according to what they represent from a mathematical point of view. The vector classes are representations of vector spaces while the matrix classes are representations of linear maps between two vector spaces.

2.1 Vector Spaces

We assume the reader is familiar with the concept of vector spaces. Essentially a vector space over a field $\mathbb{K}$ is a set V of elements (called vectors) along with vector addition $+ : V \mapsto V$ and scalar multiplication $\cdot : \mathbb{K} \times V \mapsto V$ with the well known properties. See your favourite textbook for details, e.g. [11].

For our application the following way of construction plays an important role: Let V_i, $i = 1, 2, \ldots, n$, be a normed vector space of dimension n_i with a scalar product, then the n-nary Cartesian product

$$V := V_1 \times V_2 \times \ldots \times V_n = \{(\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n) | \boldsymbol{v}_1 \in V_1, \boldsymbol{v}_2 \in V_2, \ldots, \boldsymbol{v}_n \in V_n\} \quad (1)$$

is again a normed vector space of dimension $\sum_{i=1}^{n} n_i$ with the canonical norm and scalar product.

Treating $\mathbb{K}$ as a vector space itself we can apply this construction recursively starting from the field $\mathbb{K}$.

While for a mathematician every finite dimensional vector space is isomorphic to $\mathbb{R}^k$ for an appropriate k, for our application it is important to know how the vector space was constructed recursively by the procedure described in (1).

Vector Classes. To express the construction of the vector space by n-nary products of other vector spaces ISTL provides the following classes:

FieldVector. The `template<class K, int n> FieldVector<K,n>` class template is used to represent a vector space $V = \mathbb{K}^n$ where the field is given by the type `K`. `K` may be `double`, `float`, `complex<double>` or any other numeric type. The dimension given by the template parameter `n` is assumed to be small.

Example: Use `FieldVector<double,2>` for vectors with a fixed dimension 2.

BlockVector. The `template<class B> BlockVector<B>` class template builds a vector space $V = B^n$ where the "block type" B is given by the template parameter

Table 1. Types of vector classes

expression	return type
field_type	The type of the field of the represented vector space, e.g. double.
block_type	The type of the vector blocks.
size_type	The type used for the index access and size operations.
block_level	The block level of the vector, e.g. 1 for `FieldVector`, 2 for `BlockVector<FieldVector<K>,n>`.
`Iterator`	The type of the iterator.
`ConstIterator`	The type of the immutable iterator.

`B`. `B` may be any other class implementing the vector interface. The number of blocks n is given at run-time.

Example: `BlockVector<FieldVector<double,2> >` can be used to define vectors of variable size where each block in turn consists of two `double` values.

VariableBlockVector. The `template<class B> VariableBlockVector'` class can be used to construct a vector space having a two-level block structure of the form $V = B^{n_1} \times B^{n_2} \times \ldots \times B^{n_m}$, i.e. it consists of m blocks $i = 1, \ldots, m$ and each block in turn consists of n_i blocks given by the type `B`. In principle this structure could be built also with the previous classes but the implementation here is more efficient. It allocates memory in one big array for all components. For certain operations it is more efficient to interpret the vector space as $V = B^N$, where $N = \sum_{i=1}^{m} n_i$.

Vectors are containers. Vectors are containers over the base type `K` or `B` in the sense of the Standard Template Library. Random access is provided via `operator[](int i)` where the indices are in the range $0, \ldots, n-1$ with the number of blocks n given by the `N` method. Here is a code fragment for illustration:

```
typedef Dune::FieldVector<std::complex<double>,2> BType;
Dune::BlockVector<BType> v(20);
v[3][0] = 2.56;
v[3][1] = std::complex<double>(1,-1);
```

Note how one `operator[]()` is used for each level of block recursion.

Sequential access to container elements is provided via iterators. The `Iterator` class provides read/write access while the `ConstIterator` class provides read-only access. The type names are accessed via the `::`-operator from the scope of the vector class.

A uniform naming scheme enables writing of generic algorithms. See Table 1 for the types provided in the scope of any vector class.

2.2 Linear Maps

For a matrix representing a linear map (or homomorphism) $A : V \mapsto W$ from vector space V to vector space W the recursive block structure of the matrix rows and columns immediately follows from the recursive block structure of the vectors representing the domain and range of the mapping, respectively. As a natural consequence we designed the following matrix classes:

Matrix classes. Using the construction in (1) the structure of our vector spaces carries over to linear maps in a natural way.

FieldMatrix. The `template<class K, int n> FieldMatrix<K,n,m>` class template is used to represent a linear map $M : V_1 \to V_2$ where $V_1 = \mathbb{K}^n$ and $V_2 = \mathbb{K}^m$ are vector spaces over the field given by template parameter `K`. `K` may be `double`, `float`, `complex<double>` or any other numeric type. The dimensions of the two vector spaces given by the template parameters `n` and `m` are assumed to be small. The matrix is stored as a dense matrix. Example: Use `FieldMatrix<double,2,3>` to define a linear map from a vector space over doubles with dimension 2 to one with dimension 3.

BCRSMatrix. The `template<class B> BCRSMatrix` class template represents a sparse matrix where the "block type" B is given by the template parameter `B`. `B` may be any other class implementing the matrix interface. The matrix class uses a compressed row storage scheme.

VariableBCRSMatrix. The `template<class B> VariableBCRSMatrix'` class can be used to construct a linear map between two vector spaces having a two-level block structure $V = B^{n_1} \times B^{n_2} \times \ldots \times B^{n_k}$ and $W = B^{m_1} \times B^{m_2} \times \ldots \times B^{m_l}$. Both are represented by the `template<class B> VariableBlockVector'` class, see Sect. 2.1. This is not implemented yet.

Matrices are containers of containers. Matrices are containers over the matrix rows. The matrix rows are containers over the type `K` or `B` in the sense of the Standard Template Library. Random access is provided via `operator[](int i)` on the matrix to the matrix rows and on the matrix rows to the matrix columns (if present). Note that except for `FieldMatrix`, which is a dense matrix, `operator[]` on the matrix row triggers a binary search for the column.

For sequential access use `RowIterator` and `ColIterator` for read/write access or `ConstRowIterator` and `ConstColIterator` for read-only access to rows and columns, respectively.

As with the vector interface a uniform naming convention enables generic algorithms. See Table 2 for the most important names.

3 Block Recursive Algorithms

A basic feature of the concept described by the matrix and vector classes, is their recursive block structure. Let A be a matrix with block level $l > 1$ then each block A_{ij} can be treated as (or actually is) a matrix itself. This recursiveness can be exploited in a generic algorithm using the defined `block_level` of the matrix and vector classes.

Table 2. Type names in the matrix classes

expression	**return type**
field_type	The type of the field of the vector spaces we map from and to
block_type	The type representing the matrix components
row_type	The container type of the rows.
size_type	The type used for index access and size operations
block_level	The block recursion level, e.g. 1 for `FieldMatrix` and 2 for `BCRSMatrix<FieldMatrix<K,m,n> >`.
RowIterator	The type of the mutable iterator over the rows
ConstRowIterator	Ditto, but immutable
ColIterator	The type of the mutable iterator over the columns of a row.
ConstColIterator	Ditto, but immutable

Note that we do not use recursive blocked algorithms on the dense matrix blocks, as described in [9], as the dense blocks resulting from finite element discretizations will generally be small.

Most preconditioners can be modified to honor this recursive structure for a specific number of block levels k. They then work as normal on the offdiagonal blocks, treating them as traditional matrix entries. For the diagonal values a special procedure applies: If $k > 1$ the diagonal is treated as a matrix itself and the preconditioner is applied recursively on the matrix representing the diagonal value $D = A_{ii}$ with block level $k - 1$. For the case that $k = 1$ the diagonal is treated as a matrix entry resulting in a linear solve or an identity operation depending on the algorithm.

In the formulation of most iterative methods upper and lower triangular and diagonal solves play an important role. ISTL provides block recursive versions of these generic building blocks using template metaprogramming, see Table 3 for a listing of these methods. In the table matrix A is decomposed into $A = L+D+U$, where L is a strictly lower block triangular, D is a block diagonal and U is a strictly upper block triangular matrix. $d = b - Ax$ denotes the current residual used to calculate the update v to the current guess x. An arbitrary block recursion level can be given by an additional parameter. If this parameter is omitted it defaults to 1.

Using the same block recursive template metaprogramming technique, kernels for the residual formulations of simple iterative solvers are available in ISTL. The number of block recursion levels can again be given as an additional argument. See the second part of Table 3 for a list of these kernels.

4 Solver Interface

The solvers in ISTL do not work on matrices directly. Instead we use an abstract operator concept. This allows for using matrix-free operators, i.e. operators that are not stored as matrices in any form. Thus our solver algorithms can easily be

Table 3. Iterative Solver Kernels

function	**computation**
block triangular and block diagonal solves	
`bltsolve(A,v,d)`	$v = (L+D)^{-1}d$
`bltsolve(A,v,d,`ω`)`	$v = \omega(L+D)^{-1}d$
`ubltsolve(A,v,d)`	$v = L^{-1}d$
`ubltsolve(A,v,d,`ω`)`	$v = \omega L^{-1}d$
`butsolve(A,v,d)`	$v = (D+U)^{-1}d$
`butsolve(A,v,d,`ω`)`	$v = \omega(D+U)^{-1}d$
`ubutsolve(A,v,d)`	$v = U^{-1}d$
`ubutsolve(A,v,d,`ω`)`	$v = \omega U^{-1}d$
`bdsolve(A,v,d)`	$v = D^{-1}d$
`bdsolve(A,v,d,`ω`)`	$v = \omega D^{-1}d$
iterative solves	
`dbjac(A,x,b,`ω`)`	$x = x + \omega D^{-1}(b - Ax)$
`dbgs(A,x,b,`ω`)`	$x = x + \omega(L+D)^{-1}(b - Ax)$
`bsorf(A,x,b,`ω`)`	$x_i^{k+1} = x_i^k + \omega A_{ii}^{-1}\left[b_i - \sum_{j<i} A_{ij}x_j^{k+1} - \sum_{j\geq i} A_{ij}x_j^k\right]$
`bsorb(A,x,b,`ω`)`	$x_i^{k+1} = x_i^k + \omega A_{ii}^{-1}\left[b_i - \sum_{j\leq i} A_{ij}x_j^{k} - \sum_{j>i} A_{ij}x_j^{k+1}\right]$

turned into matrix-free solvers just by plugging in matrix-free representations of linear operators and preconditioners.

4.1 Operators

The base class `template<class X, class Y> LinearOperator` represents linear maps. The template parameter `X` is the type of the domain and `Y` is the type of the range of the operator. A linear operator provides the methods `apply(const X& x, Y& y)` and `applyscaledadd(field_type alpha, const X& x, Y& y)` performing the operations $y = A(x)$ and $y = y + \alpha A(x)$, respectively. The subclass `template<class M, class X, class Y> AssembledLinearOperator` represents linear operators that have a matrix representation. Conversion from any matrix into a linear operator is done by the class `template<class M, class X, class Y> MatrixAdapter`.

4.2 Scalar Products

For convergence tests and the stopping criteria, Krylow methods need to compute scalar products and norms on the underlying vector spaces. The base class `template<class X> Scalarproduct` provides methods `field_type dot(const X& x, const X&y)` and `double norm(const X& x)` to calculate these. For sequential programs use `template<class X> SeqScalarProduct` which simply maps this to functions of the vector implementations.

Table 4. Preconditioners

class	implements	s/p	recursive
`SeqJac`	Jacobi method	s	x
`SeqSOR`	successive overrelaxation (SOR)	s	x
`SeqSSOR`	symmetric SSOR	s	x
`SeqILU`	incomplete LU decomposition (ILU)	s	
`SeqILUN`	ILU decomposition of order N	s	
`Pamg::AMG`	algebraic multigrid method	s/p	
`BlockPreconditioner`	Additive overlapping Schwarz	p	

4.3 Preconditioners

The `template<class X, class Y> Preconditioner` provides the abstract base class for all preconditioners in ISTL. The method `void pre(X& x, Y& b)` has to be called before applying the preconditioner. Here `x` is the left hand side and `b` is the right hand side of the operator equation. The method may, e.g. scale the system, allocate memory or compute an (I)LU decomposition. The method `void apply(X& v, const Y& d)` applies one step of the preconditioner to the system $A(\boldsymbol{v}) = \boldsymbol{d}$. Here `d` should contain the current residual and `v` should use 0 as the initial guess. Upon exit of the method `v` contains the computed update to the current guess, i.e. $\boldsymbol{v} = M^{-1}\boldsymbol{d}$ where M is the approximate of the operator A characterizing the preconditioner. The method `void post(X& x)` should be called after all computations to give the preconditioner the chance to clean up allocated resources.

See Table 4 for a list of available preconditioner.They have the template parameters `M` representing the type of the matrix they work on, `X` representing the type of the domain, `Y` representing the type of the range of the linear system. The block recursive preconditioners are marked with “x” in the last column. For them the recursion depth is specified via an additional template parameter `int l`. The column labeled “s/p” specifies whether they support **s**equential and/or **p**arallel mode.

4.4 Solvers

All solvers are subclasses of the abstract base class `template<class X, class Y> InverseOperator` representing the inverse of an operator from the domain of type `X` to the range of type `Y`. The actual solve of the system $A(\boldsymbol{x}) = \boldsymbol{b}$ is done in the method `void apply(X& x, Y& b, InverseOperatorResult& r)`. In the `InverseOperatorResult` object some statistics about the solution process, e.g. iteration count, achieved residual reduction, etc., are stored. All solvers only use methods of instances of `LinearOperator`, `ScalarProduct` and `Preconditioner`. These are provided in the constructor.

See Table 5 for a list of available solvers. All solvers are template classes with a template parameter `X` providing them with the vector implementation used.

Table 5. ISTL Solvers

class	implements
`LoopSolver`	only apply preconditioner multiple time
`GradientSolver`	preconditioned gradient method
`CGSolver`	preconditioned conjugate gradient method
`BiCGStab`	preconditioned biconjugate gradient stabilized method

Table 6. Performance Tests

(a) scalar product

N	500	5000	50000	500000	5000000
MFLOPS	896	775	167	160	164

(b) daxpy operation $y = y + \alpha x$

500	5000	50000	500000	5000000
936	910	108	103	107

(c) Matrix-vector product, 5-point stencil, b: block size

N, b	100,1	10000,1	1000000,1	1000000,2	1000000,3
MFLOPS	388	140	136	230	260

(d) Damped Gauß-Seidel ($N = 10^6$, $b = 1$)

	C	ISTL
time / it. [s]	0.17	0.18

4.5 Parallel Solvers

Instead of using parallel data structures (matrices and vectors) that (implicitly) know the data distribution and communication patterns like in PETSc [14,1] we decided to decouple the parallelization from the data structures used. Basically we provide an abstract consistency model on top of our linear algebra. This is hidden in the parallel implementations of the interfaces of `LinearOperator`, `Scalarproduct` and `Preconditioner`, which assure consistency of the data (by communication) for the `InverseOperator` implementation. Therefore the same Krylow method algorithms work in parallel and sequential mode.

Based on the idea proposed in [10] we implemented parallel overlapping Schwarz preconditioners with inexact (sequential) subdomain solvers and a parallel algebraic multigrid preconditioner together with appropriate implementations of `LinearOperator` and `Scalarproduct`. Nonoverlapping versions are currently being worked on.

Note that using this approach it is easy to switch from the currently implemented MPI version to new parallel programming paradigms that might be needed on new platforms.

4.6 Performance Evaluation

We evaluated the performance of our implementation on a Pentium 4 Mobile 2.4 GHz processor with a measured memory bandwidth of 1084 MB/s for the daypy operation ($x = y + \alpha z$) in Tables 6. The code was compiled with the GNU C++ compiler version 4.0 with -O3 optimization. In the tables N is the number of unknown blocks (equals the number of unknowns for the scalar cases in Tables 6(a), 6(b), 6(d), b is the size of the dense blocks. All matrices are sparse matrices of dense blocks. The performance for the scalar product, see Table 6(a), and the daxpy operation, see Table 6(b) is nearly optimal and for large N the limiting factor is clearly the memory bandwidth. Table 6(c) shows that we take advantage of cache reusage for matrices of dense blocks with block size $b > 1$. In Table 6(d) we compared the generic implementation of the Gauss Seidel solver in ISTL with a specialized C implementation. The measured times per iteration show that there is no significant lack of computational efficiency due to the generic implementation.

References

1. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2004)
2. Barton, J.J., Nackman, L.R.: Scientific and Engineering C++. Addison-Wesley, London (1994)
3. Bastian, P., Droske, M., Engwer, C., Klöfkorn, R., Neubauer, T., Ohlberger, M., Rumpf, M.: Towards a unified framework for scientific computing. In: Kornhuber, R., Hoppe, R., Périaux, J., Widlund, O., Pironneau, O., Xu, J. (eds.) Domain Decomposition Methods in Science and Engineering. LNCSE, vol. 40, pp. 167–174. Springer, Heidelberg (2005)
4. Bastian, P., Helmig, R.: Efficient fully-coupled solution techniques for two-phase flow in porous media. Parallel multigrid solution and large scale computations. Adv. Water Res. 23, 199–216 (1999)
5. BLAST Forum. Basic linear algebra subprograms technical (BLAST) forum standard (2001), `http://www.netlib.org/blas/blast-forum/`
6. Blitz++. `http://www.oonumerics.org/blitz/`
7. Dongarra, J.: List of freely available software for linear algebra on the web (2006), `http://netlib.org/utk/people/JackDongarra/la-sw.html`
8. DUNE. `http://www.dune-project.org/`
9. Elmroth, E., Gustavson, F., Jonsson, I., Kagström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46(1), 3–45 (2004)
10. Haase, G., Langer, U., Meyer, A.: The approximate dirichlet domain decomposition method. part i: An algebraic approach. Computing 47, 137–151 (1991)
11. Hefferson, J.: Linear algebra. in the web (May 2006), `http://joshua.amcvt.edu/`
12. Iterative template library, `http://www.osl.iu.edu/research/itl/`
13. Matrix template library, `http://www.osl.iu.edu/research/mtl/`
14. PETSc, `http://www.mcs.anl.gov/petsc/`
15. Siek, J., Lumsdaine, A.: A modern framework for portable high-performance numerical linear algebra. In: Langtangen, H.P., Bruaset, A.M., Quak, E. (eds.) Advances in Software Tools for Scientific Computing, LNCSE, vol. 10, pp. 1–56. Springer, Heidelberg (1974)
16. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, London (1997)
17. Veldhuizen, T.: Techniques for scientific C++. Technical report, Indiana University, Computer Science Department (1999)
18. Vuduc, R., Demmel, J.W., Yelick, K.A.: Oski: A library of automatically tuned sparse matrix kernels. Journal of Physics Conference Series 16, 521–530 (2005)

EulFS: A Parallel CFD Code for the Simulation of Euler and Navier-Stokes Problems on Unstructured Grids

Aldo Bonfiglioli[1], Bruno Carpentieri[2], and Masha Sosonkina[3]

[1] Dip.to di Ingegneria e Fisica dell'Ambiente, University of Basilicata, Potenza, Italy
ba001ing@unibas.it
[2] Karl-Franzens University, Institut of Mathematics and Scientific Computing, Graz, Austria
bruno.carpentieri@uni-graz.at
[3] Ames Laboratory/DOE, Iowa State University, Ames, USA
masha@scl.ameslab.gov

Abstract. We present results with a parallel CFD code that computes steady-state solutions of the Reynolds-Favre averaged Navier-Stokes equations for the simulation of the turbulent motion of compressible and incompressible Newtonian fluids. We describe solution techniques used for the discretization, algorithmic details of the implementation and report on preliminary experiments on 2D and 3D problems, for both internal and external flow configurations.

1 The Physical Problem

The simulation of fluid dynamic problems relies on the solution of Euler and Navier-Stokes equations, a set of partial differential equations that describe the fundamental conservation laws of mass, motion quantity and energy applied to continuous flows. Approximate time-dependent values of the conserved variables may be computed by decomposing the computational domain $\Omega \subseteq \mathbb{R}^d (d = 2, 3)$ into a finite set of nonoverlapping control volumes C_i and discretizing the conservation laws for each infinitesimal volume. The discretization results in a system of nonlinear equations that can be solved using Newton's method. Given a control volume C_i, fixed in space and bounded by the control surface ∂C_i with inward normal $\mathbf{n}$, the integral, conservation law form of the mass, momentum, energy and turbulence transport equations can be concisely written as:

$$\int_{C_i} \frac{\partial \mathbf{U}_i}{\partial t}\, dV = \oint_{\partial C_i} \mathbf{n} \cdot \mathbf{F}\, dS - \oint_{\partial C_i} \mathbf{n} \cdot \mathbf{G}\, dS + \int_{C_i} \mathbf{S}\, dV \qquad (1)$$

where we denote by $\mathbf{U}$ the vector of conserved variables. For compressible flows, we have $\mathbf{U} = (\rho, \rho e, \rho \mathbf{u}, \tilde{\nu})^T$, and for incompressible, constant density flows, $\mathbf{U} = (p, \mathbf{u}, \tilde{\nu})^T$. Throughout this paper, the standard notation is adopted for the kinematic and thermodynamic variables: $\mathbf{u}$ is the flow velocity, ρ is the density, p

is the pressure (divided by the constant density in incompressible, homogeneous flows), T is the temperature, e and h are the specific total energy and enthalpy, respectively, $\tilde{\nu}$ is a scalar variable related to the turbulent eddy viscosity ν via a damping function. The operators $\mathbf{F}$ and $\mathbf{G}$ represent the inviscid and viscous fluxes, respectively. For compressible flows,

$$\mathbf{F} = \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u}h \\ \rho\mathbf{u}\mathbf{u} + p\mathbf{I} \\ \tilde{\nu}\mathbf{u} \end{pmatrix}, \quad \mathbf{G} = \frac{1}{\mathrm{Re}_\infty} \begin{pmatrix} 0 \\ \mathbf{u}\cdot\boldsymbol{\tau} + \nabla q \\ \boldsymbol{\tau} \\ \frac{1}{\sigma}\left[(\nu+\tilde{\nu})\,\nabla\tilde{\nu}\right] \end{pmatrix},$$

and for incompressible, constant density flows,

$$\mathbf{F} = \begin{pmatrix} a^2\mathbf{u} \\ \mathbf{u}\mathbf{u} + p\mathbf{I} \\ \tilde{\nu}\mathbf{u} \end{pmatrix}, \quad \mathbf{G} = \frac{1}{\mathrm{Re}_\infty} \begin{pmatrix} 0 \\ \tau \\ \frac{1}{\sigma}\left[(\nu+\tilde{\nu})\,\nabla\tilde{\nu}\right] \end{pmatrix}.$$

Finally, $\mathbf{S}$ is the source term, which has a non-zero entry only in the row corresponding to the turbulence transport equation:

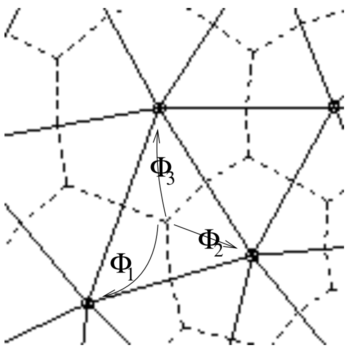
$$\mathbf{S} = \begin{pmatrix} 0 \\ 0 \\ \mathbf{0} \\ c_{b1}\left[1 - f_{t2}\right]\tilde{S}\tilde{\nu} + \frac{1}{\sigma Re}\left[c_{b2}\left(\nabla\tilde{\nu}\right)^2\right] + \\ -\frac{1}{Re}\left[c_{w1}f_w - \frac{c_{b1}}{\kappa^2}f_{t2}\right]\left[\frac{\tilde{\nu}}{d}\right]^2 + Re f_{t1}\Delta U^2 \end{pmatrix}.$$

In the case of high Reynolds number flows, turbulence effects are accounted for by the Reynolds-Favre averaged Navier-Stokes (RANS) equations that are obtained from the Navier-Stokes (NS) equations by means of a time averaging procedure. The RANS equations have the same structure as the NS equations with an additional term, the Reynolds' stress tensor, that accounts for the effects of the turbulent scales on the mean field. A closure problem arises, since the Reynolds' stresses require modeling. Using Boussinesq's hypothesis as the constitutive law for the Reynolds' stresses amounts to link the Reynolds' stress tensor to the mean velocity gradient through a scalar quantity which is called turbulent (or eddy) viscosity. With Boussinesq's approximation, the RANS equations become formally identical to the NS equations, except for an "effective" viscosity (and turbulent thermal conductivity), sum of the laminar and eddy viscosities, which appears in the viscous terms of the equations. In the present work, the turbulent viscosity is modeled using the Spalart-Allmaras one-equation model [7]. Despite the non-negligible degree of empiricism introduced by turbulence modeling, it is recognized that the solution of the RANS equations still remains the only feasible approach to perform computationally affordable simulations of problems of engineering interest on a routine basis.

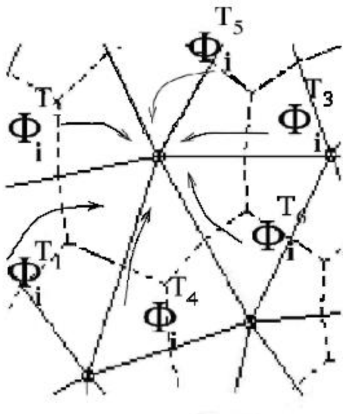
2 Solution Techniques

The compressible RANS equations are discretized in space using Fluctuation Splitting (or residual distribution) schemes [9]. Introduced in the early eighties by

P.L. Roe, and successively further developed by a number of groups worldwide, this class of schemes shares common features with both Finite Element (FE) and Finite Volume (FV) methods. Just as with iso-P1 FE, the dependent variables are stored in the vertices of the mesh, which is made of triangles in two space dimensions (2D) and tetrahedra in three (3D), and assumed to vary linearly in space. Control volumes (C_i) are drawn around each gridpoint by joining (in 2D) the centroids of gravity of the surrounding cells with the midpoints of all the edges that connect that gridpoint with its nearest neighbors. An example of these polygonal shaped control volumes (so-called median dual cells) is shown by dashed lines in Fig. 1. Using a FV-type approach, the integral form of the governing equations (1) is discretized over each control volume C_i; however, rather than calculating fluxes by numerical quadrature along the boundary ∂C_i of the median dual cell, as would be done with conventional FV schemes, the flux integral is first evaluated over each triangle (tetrahedron) in the mesh and then splitted among its vertices, see Fig. 1(a). Gridpoint i will then collect fractions Φ_i^T of the flux balances of all the elements by which it is surrounded, as schematically shown in Fig. 1(b).



(a) The flux balance of cell T is scattered among its vertices.



(b) Gridpoint i gathers the fractions of cell residuals from the surrounding cells.

Fig. 1. Residual distribution concept

This approach leads to a space-discretized form of Eq. (1) that reads:

$$\int_{C_i} \frac{\partial \mathbf{U}_i}{\partial t}\, dV = \sum_{T \ni i} \Phi_i^T$$

where

$$\Phi^T = \oint_{\partial T} \mathbf{n} \cdot \mathbf{F}\, dS - \oint_{\partial T} \mathbf{n} \cdot \mathbf{G}\, dS + \int_T \mathbf{S}\, dV$$

is the flux balance evaluated over cell T and Φ_j^T is the fraction of cell residual scattered to its jth vertex. Conservation requires that the sum over the vertices of a given cell T of the splitted residuals Φ_j^T equals the total flux balance, *i.e.*, $\sum_{j \in T} \Phi_j^T = \Phi^T$. The properties of the scheme will depend upon the criteria used to distribute the cell residual: distributing the convective flux balance along the characteristic directions gives the discretization an upwind flavour, while the distribution of the viscous flux balance can be shown to be equivalent to a standard Galerkin FE discretization. It should also be stressed that, since the dependent variables are continuous across the element interfaces, the Riemann problem model, so commonly used in most FV discretizations, is not adopted in the present framework. As a consequence, this rather unusual approach to a FV-type discretization leads to a set of discrete equations that shows closer resemblance [3] to FE Petrov-Galerkin schemes rather than to FV ones. Full details concerning the spatial discretization can be found in [2].

The discretization of the governing equations in space leads to the following system of ordinary differential equations:

$$\mathbf{M}\frac{d\mathbf{U}}{dt} = \mathbf{R}(\mathbf{U}) \tag{2}$$

where t is the physical time variable. In Eq. (2), $\mathbf{M}$ is the mass matrix and $\mathbf{R}(\mathbf{U})$ represents the spatial discretization operator, or nodal residual, which vanishes at steady state. We solve Eq. (2) in pseudo-time until a stationary solution is reached. Since we are interested in steady state solutions, the mass matrix can be replaced by a diagonal matrix $\mathbf{V}$, whose entries are the volumes (areas in 2D) of the median dual cells. The residual vector $\mathbf{R}(\mathbf{U})$ is a (block) array of dimension equal to the number of meshpoints times the number of dependent variables, m; for compressible flows and a one-equation turbulence model $m = d + 3$ where d is the spatial dimension. The i-th entry of $\mathbf{R}(\mathbf{U})$ represents the discretized equation of the conservation laws for meshpoint i:

$$\mathbf{R}_i = \sum_{T \ni i} \Phi_i^T = \sum_{j=1}^{\mathcal{N}_i} \left(\mathbf{C}_{ij} - \mathbf{D}_{ij}\right) \mathbf{U}_j. \tag{3}$$

In Eq. (3) the second summation ranges over the set $\mathcal{N}_i$ of nodes surrounding (and including) meshpoint i, and the matrices $\mathbf{C}_{ij}$ and $\mathbf{D}_{ij}$, respectively, account for the contribution of the inviscid and viscous terms in the governing equations; their detailed form is given in [2]. If the time derivative in equation (2) is approximated using a two-point, one-sided finite difference formula:

$$\frac{d\mathbf{U}}{dt} = \frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t},$$

then an explicit scheme is obtained by evaluating $\mathbf{R}(\mathbf{U})$ at time level n and an implicit scheme if $\mathbf{R}(\mathbf{U})$ is evaluated at time level $n + 1$. In this latter case, linearizing $\mathbf{R}(\mathbf{U})$ about time level n, *i.e.*

$$\mathbf{R}(\mathbf{U}^{n+1}) = \mathbf{R}(\mathbf{U}^n) + \left(\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right)^n \left(\mathbf{U}^{n+1} - \mathbf{U}^n\right) + H.O.T.$$

we obtain the following implicit scheme:

$$\left(\frac{1}{\Delta t^n} \mathbf{V} - \mathbf{J} \right) \left(\mathbf{U}^{n+1} - \mathbf{U}^n \right) = \mathbf{R}(\mathbf{U}^n), \qquad (4)$$

where we denote by $\mathbf{J}$ the Jacobian of the residual $\frac{\partial \mathbf{R}}{\partial \mathbf{U}}$. Eq. (4) represents a large nonsymmetric (though structurally symmetric) sparse linear system of equations to be solved at each pseudo-time step for the update of the vector of the conserved variables. Due to the compact stencil $\mathcal{N}_i$ of the schemes, the sparsity pattern of the Jacobian matrix coincides with the graph of the underlying unstructured mesh, *i.e.* it involves only one-level neighbours. On average, the number of non-zero (block) entries per row equals 7 in 2D and 14 in 3D.

3 *EulFS*

We present experiments with an academic code [2] that simulates the turbulent motion of compressible and incompressible Newtonian fluids on 2D and 3D computational domains. The code computes the steady-state solution of the RANS equations for both inviscid (Euler's equations) and viscous (Navier-Stokes equations) fluids, for internal and external flow configurations using unstructured grids. The numerical schemes adopted in the package are *fluctuation splitting* for the spatial discretization of the conservation equations and *pseudo-time stepping* for the time discretization. The analytical evaluation of the Jacobian matrix, though not impossible, is rather cumbersome [4] so that two alternatives have been adopted in the present implementation: one is based on an analytically calculated, but approximate Jacobian, the other uses a numerical approximation of the "true" Jacobian, obtained using one-sided finite differences formulae. In both cases, the individual entries of the Jacobian matrix are computed and stored in memory.

The approximate (or Picard) linearization amounts to compute a given Jacobian entry as follows:

$$\mathbf{J}_{ij} \approx \mathbf{C}_{ij} - \mathbf{D}_{ij},$$

i.e. it neglects the dependence of the convective and diffusive matrices upon the solution, see Eq. (3).

In the finite difference (FD) approximation, the individual entries of the vector of nodal unknowns are perturbed by a small amount ϵ and the nodal residual is then recomputed for the perturbed state $\hat{\mathbf{U}}_j$. The Jacobian entries are then approximated using one-sided FD formulae:

$$\mathbf{J}_{ij} \approx \frac{1}{\epsilon} \left(\mathbf{R}_i(\hat{\mathbf{U}}_j) - \mathbf{R}_i(\mathbf{U}_j) \right).$$

Although the Jacobian matrix can be assembled using a single loop over all cells [4], its evaluation is computationally expensive, as it requires $(d+1) \times m$ residual evaluations. Therefore, the use of the FD approximation to the true Jacobian matrix pays off only if a considerable reduction in number of iterations can be achieved over simpler iterative schemes. This can be obtained by exploiting the quadratic convergence of Newton's rootfinding method, which is recovered from Eq. (4) as the timestep approaches infinity. However, the error reduction in Newton's method is quadratic only if the initial guess is within a sufficiently small neighborhood of the steady state. This circumstance is certainly not met if the numerical simulation has to be started from "scratch", so that in practice the following two-step approach is adopted. In the early stages of the calculation, the turbulent transport equation is solved in tandem with the mean flow equations: the mean flow solution is advanced over a single time step using an approximate (Picard) Jacobian while keeping turbulent viscosity frozen, then the turbulent variable is advanced over one or more pseudo-time steps using a FD Jacobian with frozen mean flow variables. Due to the uncoupling between the mean flow and turbulent transport equations, this procedure will eventually converge to steady state, but never yields quadratic convergence. Therefore, once the solution has come close to steady state, a true Newton strategy is adopted: the mean flow and turbulence transport equation are solved in fully coupled form and the Jacobian is computed by FD. Also, the size of the time-step needs to be rapidly increased to recover Newton's algorithm and, according to the Switched Evolution Relaxation (SER) strategy proposed in [5], this is accomplished by letting Δt^n in Eq. (4) vary with the L_2 norm of the residual at the initial and current time-step as:

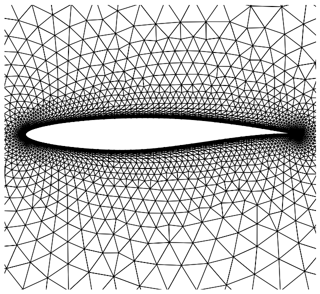
$$\Delta t^n = \Delta t \frac{||\mathbf{R}(\mathbf{U}^0)||_2}{||\mathbf{R}(\mathbf{U}^n)||_2},$$

where Δt is the time-step computed using the stability criterion of the explicit time integration scheme.

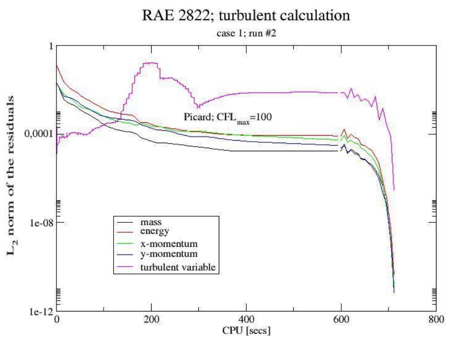
The code is implemented using the PETSc library [1] and has been ported to different parallel computer architectures (SGI/Cray T3E-900, SUN E3500, DEC Alpha, SUN Solaris and IBM RS6000, Linux Beowulf cluster) using the MPI standard for the message-passing communications. The simulations presented in this study are run on a Linux Beowulf cluster.

Experiments on the RAE Problem

The first test-case that we consider is the compressible, subsonic flow past the two-dimensional RAE2822 profile. Free-stream conditions are as follows: Mach number $M_\infty = 0.676$, Reynolds' number based on chord: $\mathrm{Re}_C = 5.7 \cdot 10^6$, angle of attack: $\alpha_\infty = 2.40°$. The computational mesh, which is shown in Fig. 2(a), is made of 10599 meshpoints and 20926 triangles. The simulation is started from uniform flow and the solution is advanced in pseudo-time using approximate linearization. Once the L_2 norm of the residual has been reduced below a pre-set threshold, the fully coupled approach is put in place. The convergence history towards steady-state is shown in Fig. 2(b): only seven Newton iterations are required to reduce the L_2 norm of the residuals (mass, energy, x and y momentum) to machine zero. For the inner linear solver, we use GMRES(30) [6] preconditioned by an incomplete LU factorization with pattern selection strategy



(a) Computational mesh for the RAE2822 aerofoil.



(b) L_2 residual norms versus solution time.

Fig. 2. Experiments on the RAE2822 airfoil

based on the level of fill. The set $\mathcal{F}$ of fill-in entries to be kept for the approximate lower triangular factor $\mathcal{L}$ is given by

$$\mathcal{F} = \{\, (k,i) \mid lev(l_{k,i}) \leq \ell \,\} \ ,$$

where integer ℓ denotes a user specified maximal fill-in level. The level $lev(l_{k,i})$ of the coefficient $l_{k,i}$ of $\mathcal{L}$ is defined as follows:

Initialization

$$lev(l_{k,i}) = \begin{cases} 0 \ \text{ if } \ l_{k,i} \neq 0 \ \text{ or } \ k = i \\ \\ \infty \text{ otherwise} \end{cases}$$

Factorization

$$lev(l_{k,i}) = \min \{\, lev(l_{k,i}) \, , \, lev(l_{i,j}) + lev(l_{k,j}) + 1 \,\} \ .$$

The resulting preconditioner is denoted by $ILU(\ell)$. A similar strategy is adopted for preserving sparsity in the upper triangular factor $\mathcal{U}$. We take $x_0 = 0$ as initial guess for GMRES and the stopping criterion as to reduce the original residual by 10^{-5}, so that it can then be related to a norm-wise backward error. In Tables 1 and 2, we report on the number of iterations and the solution time (in seconds for a single-processor execution), respectively, obtained with $ILU(\ell)$ for different levels of fill-in from zero to four. Note that, in Fig. 2(b), the results are for $ILU(2)$. It may be observed from Table 1, that the $ILU(\ell)$ variants with a non-zero fill level lead to a more uniform iteration numbers needed to solve the linear system in each Newton step.

A blocking strategy is incorporated in the construction of the preconditioner to exploit the inherent block structure of the coefficient matrix of the linear system. Block row operations replace standard row operations and diagonal blocks are inverted instead of the diagonal entries. The block size is equal to five, that is the number of fluid dynamic variables in each computational node. We observe that the use of block storage schemes combined with optimized routines available in PETSc for inverting the small diagonal blocks enables to save both memory and setup costs. For this small problem, blocking of the physical variables reduces the solution time of each linear system by a factor of three. Finally, the curves of the pressure distribution along the airfoil, reported in Fig. 3(a), show a very good agreement between the simulation and the experimental data.

Table 1. Iterations for solving the RAE problem using GMRES(30) preconditioned by block $ILU(\ell)$

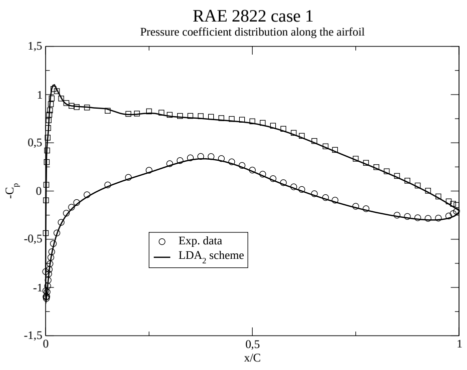
Newton iter	Iterations for $ILU(\ell)$				
	0	1	2	3	4
1	56	35	22	19	17
2	115	55	29	23	21
3	111	60	44	27	23
4	85	53	32	22	20
5	56	43	23	19	16
6	116	55	36	24	19
7	82	35	25	20	17

Table 2. CPU time for solving the RAE problem using GMRES(30) preconditioned by block $ILU(\ell)$

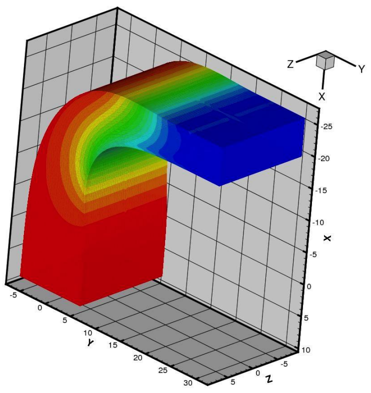
Newton iter	Solution time (sec)				
	0	1	2	3	4
1	1.7	1.2	0.9	0.9	1.0
2	3.3	1.7	1.1	1.1	1.1
3	3.2	1.9	1.6	1.2	1.2
4	2.5	1.7	1.2	1.0	1.1
5	1.6	1.4	0.9	0.9	0.9
6	3.4	1.7	1.4	1.1	1.0
7	2.4	1.2	1.0	0.9	0.9

Experiments on the Stanitz Elbow

The three-dimensional test case that we have examined deals with the internal compressible flow through the so-called Stanitz elbow. The simulation reproduces experiments [8] conducted in early 1950's at the National Advisory Committee for Aeronautics (NACA), presently NASA, to study secondary flows in an accelerating, rectangular elbow with 90° of turning. The chosen flow conditions correspond to a Mach number in the outlet section of 0.68 and Reynolds' number $4.3 \cdot 10^5$. Figure 3(b) shows the geometry along with the computed static pressure contours. The computational mesh consists of 156065 meshpoints and 884736 tetrahedral cells. The simulation has been run on 16 processors of a Linux Beowulf cluster. Figures 4(a)–4(b) show the convergence history of the iterative solution; we report on experiments with an Additive Schwarz preconditioner (see, *e.g.* [6]) with overlap=2 for GMRES, where the diagonal blocks are approximately inverted using $ILU(1)$. It can be seen that the decoupled strategy which integrates the mean flow variables using Picard linearization and the turbulent transport equation using Newton linearization is not robust enough to obtain residual convergence to machine epsilon. Thus it needs to be accelerated using a true Newton algorithm close to the steady state. The only problem is to determine an effective strategy for switching from one integration scheme to the other. In the present implementation, we adopt a criterium based on a user-defined tolerance for the residual reduction and a maximum number of (Picard)



(a) RAE problem: comparison with experimental data.

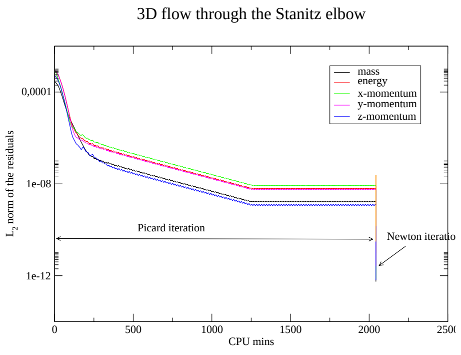


(b) Stanitz elbow geometry and computed static pressure contours.

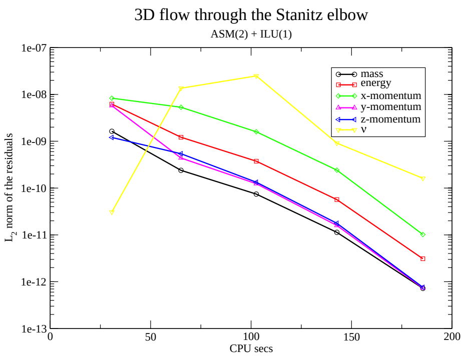
Fig. 3. The RAE and "Stanitz" elbow problems

iterations. Although maybe not optimal, this strategy proves to be fairly robust on the reported experiments. In Table 3, we report on comparative results with respect to number of iterations and solution time (in seconds) with a block Jacobi preconditioner. For the Additive Schwarz method, we performed tests with different values of overlap observing very similar results.

Although the solution of the RANS equation may require much less computational effort of other simulation techniques like LES (Large Eddy Simulation) and DNS (Direct Numerical Simulation), severe numerical difficulties may arise when the mean flow and turbulence transport equation are solved in fully coupled form, the Jacobian is computed *exactly* by means of FD and the size of the time-step is rapidly increased to recover Newton's



(a) Convergence history for Picard iterations using ASM(2)+ILU(1).



(b) Convergence history for Newton iterations using ASM(2)+ILU(1).

Fig. 4. Experiments on the Stanitz elbow

Table 3. Solution cost for solving the Stanitz problem using GMRES(30)

Newton iter	BJ+ILU(1)		ASM(2)+ILU(1)	
	Iter	CPU time	Iter	CPU time
1	72	10.1	49	8.8
2	121	16.7	76	12.3
3	147	20.0	98	15.8
4	175	23.5	112	17.8
5	200	26.9	127	20.1

algorithm. Indeed, on 3D unstructured problems reported successful experiments are not numerous in the literature. The code is still in a development stage but the numerical results are encouraging. Perspectives of future research include enhancing both the performance and the robustness of the code on more difficult configurations, and designing a multilevel incomplete LU factorization preconditioner for solving the inner linear system, that preserves the inherent block structure of the coefficient matrix and scales well with the number of unknowns.

References

1. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: PETSc home page (1998), `http://www.mcs.anl.gov/petsc`
2. Bonfiglioli, A.: Fluctuation splitting schemes for the compressible and incompressible Euler and Navier-Stokes equations. IJCFD 14, 21–39 (2000)
3. Carette, J.-C., Deconinck, H., Paillère, H., Roe, P.L.: Multidimensional upwinding: its relation to finite elements. International Journal for Numerical Methods in Fluids 20, 935–955 (1995)
4. Issman, E.: Implicit Solution Strategies for Compressible Flow Equations on Unstructured Meshes. PhD thesis, Université Libre de Bruxelles (1997)
5. Mulder, W., van Leer, B.: Experiments with an Implicit Upwind Method for the Euler Equations. Journal of Computational Physics 59, 232–246 (1985)
6. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
7. Spalart, P.R., Allmaras, S.R.: A one-equation turbulence model for aerodynamic flows. La Recherche-Aerospatiale 1, 5–21 (1994)
8. Stanitz, J.D., Osborn, W.M., Mizisin, J.: An experimental investigation of secondary flow in an accelerating, rectangular elbow with 90° of turning. Technical Note 3015, NACA (1953)
9. van der Weide, E., Deconinck, H., Issman, E., Degrez, G.: A parallel, implicit, multidimensional upwind, residual distribution method for the Navier-Stokes equations on unstructured grids. Computational Mechanics 23, 199–208 (1999)

Making Hybrid Tsunami Simulators in a Parallel Software Framework

Xing Cai[1,2] and Hans Petter Langtangen[1,2]

[1] Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway
{xingca, hpl}@simula.no
[2] Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, NO-0316 Oslo, Norway

Abstract. Extending the philosophy of additive Schwarz algorithms, we propose a hybrid framework that allows different subdomains to use different mathematical models, different spatial discretizations, different local mesh types, and even different serial codes. This hybrid software framework is implemented using object-oriented techniques, such that existing serial codes are easily reused after being equipped with the standard interface of a generic subdomain solver. The resulting hybrid parallel tsunami simulator thus has full flexibility and extensibility. The focus of this paper is on the software design of the framework, with an illustrating example of application.

1 Introduction and Motivation

Computing the propagation of waves in the open sea is a key issue in tsunami simulation. When an entire ocean is the solution domain, this computational task becomes extremely challenging, both due to the huge amount of computations needed and due to the fact that different physics are valid in different regions. For example, the effect of dispersion is important for modeling wave propagation over an vast region with large water depth (see e.g. [5,10]). Moreover, in regions where water depth rapidly changes or close to the coastlines, nonlinear effects become important. Both the above factors mean that using a simple wave propagation model (such as the shallow water formulation (3)-(4) to be given later) over an entire ocean domain may seriously affect the accuracy (see e.g. [8,6]). Therefore, the computations should preferably adopt a hybrid strategy, i.e., using advanced mathematical models in small local areas where needed, while applying simple models to the remaining large regions. This is for achieving an acceptable balance between computational efficiency and accuracy.

Although tsunami simulations often employ the non-dispersive standard shallow water equations, we have in this work applied a set of Boussinesq equations, which are capable of modeling weakly dispersive and nonlinear waves:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot (H + \alpha\eta)\nabla\phi + \epsilon H \left(\frac{1}{6}\frac{\partial \eta}{\partial t} - \frac{1}{3}\nabla H \cdot \nabla\phi\right)\nabla H = 0, \tag{1}$$

$$\frac{\partial \phi}{\partial t} + \frac{\alpha}{2}\nabla\phi \cdot \nabla\phi + \eta - \frac{\epsilon}{2} H \nabla \cdot \left(H \nabla \frac{\partial \phi}{\partial t}\right) + \frac{\epsilon}{6} H^2 \nabla^2 \frac{\partial \phi}{\partial t} = 0, \tag{2}$$

Equation (1) is called the continuity equation, and Equation (2) is a variant of the Bernoulli (momentum) equation. In the above equations, η and ϕ are the primary unknowns denoting, respectively, the water surface elevation and the velocity potential. The water depth H is assumed to be a function of the spatial coordinates x and y. In (1)-(2) the effect of dispersion and nonlinearity is controlled by the two dimensionless constants ϵ and α, respectively. For more mathematical and numerical details, we refer to [10,7,3]. Note that by choosing $\epsilon = \alpha = 0$, we recover the widely used linear shallow water equations:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot (H \nabla \phi) = 0, \tag{3}$$

$$\frac{\partial \phi}{\partial t} + \eta = 0. \tag{4}$$

The Boussinesq equations (1)-(2) can be considered a compromise between the computationally too expensive Navier-Stokes equations and the simple shallow water model (3)-(4). The numerical algorithm for solving (1)-(2) typically consists of a time-stepping process that solves the following two semi-discretized equations per time level:

$$\begin{aligned} \frac{\eta^{\ell} - \eta^{\ell-1}}{\Delta t} &+ \nabla \cdot \Bigg(\left(H + \alpha \frac{\eta^{\ell-1} + \eta^{\ell}}{2} \right) \nabla \phi^{\ell - \frac{1}{2}} \\ &+ \epsilon H \left(\frac{1}{6} \frac{\eta^{\ell} - \eta^{\ell-1}}{\Delta t} - \frac{1}{3} \nabla H \cdot \nabla \phi^{\ell-\frac{1}{2}} \right) \nabla H \Bigg) = 0, \end{aligned} \tag{5}$$

$$\begin{aligned} \frac{\phi^{\ell+\frac{1}{2}} - \phi^{\ell-\frac{1}{2}}}{\Delta t} &+ \frac{\alpha}{2} \nabla \phi^{\ell-\frac{1}{2}} \cdot \nabla \phi^{\ell+\frac{1}{2}} - \frac{\epsilon H}{2} \nabla \cdot \left(H \frac{\nabla \phi^{\ell+\frac{1}{2}} - \nabla \phi^{\ell-\frac{1}{2}}}{\Delta t} \right) \\ &+ \frac{\epsilon H^2}{6} \frac{\nabla^2 \phi^{\ell+\frac{1}{2}} - \nabla^2 \phi^{\ell-\frac{1}{2}}}{\Delta t} = -\eta^{\ell}. \end{aligned} \tag{6}$$

The above numerical scheme has adopted centered differences in the temporal direction and an associated staggered temporal grid [9]. The superscript ℓ in (5)-(6) denotes the time level. For the spatial discretization, both finite elements and finite differences can be used, depending on whether or not unstructured (and adaptively refined) meshes are needed to resolve the details of the water depth and/or the shape of coastlines. We mention that the shallow water model (3)-(4), being a special case of the Boussinesq equations (1)-(2), can be discretized in the temporal direction in the same fashion as in (5)-(6), likewise for the subsequent spatial discretization. The difference is that the resulting numerical strategy for solving (3)-(4) is often of an explicit nature (no need to solve linear systems), giving rise to an extremely fast algorithm. In contrast, the numerical strategy for (1)-(2) is of an implicit nature, meaning that linear systems must be solved for both (5) and (6) at every discrete time level. Moreover, unstructured finite element meshes will incur more computation time, in comparison with solving linear systems related to (5)-(6) on uniform finite difference spatial meshes. This is due to the complex data structure and indirect memory access that are used by the finite element codes. Therefore, in respect of software, the coding complexity and computational cost also suggest that advanced mathematical models and unstructured computational meshes should only be applied to local small areas where necessary.

2 Parallelization by a Subdomain-Based Approach

Parallel computing is essential for simulating wave propagation over an entire ocean, because a huge number of degrees of freedom are often needed. As we have discussed above, different physics are valid in different regions, calling for a computationally resource-aware parallelization. More specifically, in regions where nonlinear and/or dispersive effects are important, existing serial software for Boussinesq equations (1)-(2) should be applied. Likewise can existing serial software for linear shallow water equations (3)-(4) be used in the remaining regions.

Such a parallelization strategy is most easily realized by using subdomains, such that the entire spatial domain Ω is decomposed into a set of overlapping subdomains $\{\Omega_s\}_{s=1}^{P}$. Mathematically, this idea of parallelization was first conceived in the additive Schwarz algorithms, see [11]. In a generic setting, where a partial differential equation (PDE) is expressed as

$$\mathcal{L}_{\Omega}(u) = f_{\Omega},$$

the Schwarz algorithm consists of an iterative process generating $u^0, u^1, \ldots, u^k$ as a series of approximate solutions. During Schwarz iteration k, each subdomain first *independently* updates its local solution through

$$\mathcal{L}_{\Omega_s}(u_s^k) = f_{\Omega_s}^{k-1}. \tag{7}$$

Note that notation $f_{\Omega_s}^{k-1}$ means a right-hand side due to restricting f_{Ω} within Ω_s while making use of the latest global approximation u^{k-1} on the internal boundaries of Ω_s. When all the subdomains have finished solving (7), the new global solution u^k is composed by "sewing together" the subdomain local solutions $u_1^k, u_2^k, \ldots, u_P^k$.

Equation (7) thus opens for the possibility of using different local solvers in different subdomains. Taking the idea of additive Schwarz one step further, we can also apply different mathematical models in different subdomains. Therefore, different serial codes may be deployed regionwise. In the context of solving (5)-(6), these two equation can each use a series of the above Schwarz iterations at every discrete time level. The same set of subdomains should be used for both (5) and (6), to avoid unnecessary cross-subdomain data shuffle.

3 An Object-Oriented Implementation

To implement a hybrid parallel tsunami simulator as argued above, we resort to object-oriented programming techniques. For simplicity, the implementation can be extended from a generic library of Schwarz algorithms for solving PDEs, such as that described in [1]. Our objective is a flexible design, such that existing serial wave propagation codes can be easily integrated into a hybrid parallel simulator of wave propagation. We will use C++ syntax in the following text, but the object-oriented strategy is equally implementable using another language such as Python, see e.g. [2].

3.1 A Generic Schwarz Framework

Before explaining the overall design of an object-oriented hybrid parallel tsunami simulator, it is necessary to briefly repeat the generic library of Schwarz algorithms, as described in [1]. Let us assume that the generic Schwarz library consists of two generic components, say, `class SubdomainSolver` and `class Administrator`. The purpose of the generic base class `SubdomainSolver` is to declare on beforehand a generic interface of all concrete subdomain solvers, which can later be inserted into the generic Schwarz framework. The generic interface is namely a set of virtual member functions without concrete implementation. For example, `createLocalMatrix` is a virtual function meant for setting up the subdomain matrix associated with discretizing (7), and function `solveLocal` is meant for solving the discretized form of (7) during each Schwarz iteration. The actual computational work is of course realized inside a concrete subclass of `SubdomainSolver`, which implements the virtual member functions such as `createLocalMatrix` and `solveLocal`, either by a cut-and-paste of old serial codes or more elegantly as a wrapper of an existing solver class.

Regarding the generic base class `Administrator`, the purpose is also to implement on beforehand a common set of functions, some of them as virtual member functions, useful later in a concrete parallel PDE solver based on additive Schwarz iterations. The typical functions of `Administrator` deal with, e.g., checking the global convergence among subdomains and invoking required inter-subdomain communication, all of which are independent of specific PDEs.

3.2 Designing a Parallel Tsunami Simulator

Now it is time for us to present the design of a hybrid parallel tsunami simulator. To maintain flexibility, while considering the special features with solving the Boussinesq water wave equations (1)-(2) (recall that (3)-(4) is a special case), we introduce a new generic class `SubdomainBQSolver`. The class is derived as a subclass of `SubdomainSolver` to implement all the virtual member functions of `SubdomainSolver`, while introducing a small set of new virtual functions. This is because, e.g., the two semi-discretized equations (5)-(6) both need to be solved using additive Schwarz iterations, thus requiring the `solveLocal` function to contain two versions, one for (5) and the other for (6). The structure of function `SubdomainBQSolver::solveLocal` may therefore be as follows:

```
if (solve_4_continuity)
  return solveContinuityEq ();
else
  return solveBernoulliEq ();
```

Here, `solve_4_continuity` is a flag indicating which equation, (5) or (6), is the current solution target. We note that the two new virtual member functions `solveContinuityEq` and `solveBernoulliEq` are left open for concrete subclasses to later insert their actual code of computation.

Once the new generic class `SubdomainBQSolver` is ready, a new class with name `HybridBQSolver` is derived as a subclass from base class `Administrator`. All the virtual functions of `Administrator` are implemented in `HybridBQSolver`, which also implements the time-stepping process that solves (5)-(6) at each time level. This in turn relies on an object of `SubdomainBQSolver` to perform the actual subdomain work for solving (7) during each Schwarz iteration. We remark that an object of `HybridBQSolver` and an object of a subclass of `SubdomainBQSolver` will be deployed on each processor during a parallel simulation, and inter-subdomain communication is handled between the objects of `HybridBQSolver`. The remaining programming work needed to implement an actual parallel tsunami simulator is mainly in form of deriving concrete subclass(es) of `SubdomainBQSolver`, best illustrated by the following case study. A hybrid parallel simulator arises when objects of different subclasses of `SubdomainBQSolver` are deployed on different subdomains.

4 Case Study

We have two existing serial software codes: (1) an advanced C++ finite element solver named `class Boussinesq` applicable for unstructured meshes, and (2) a legacy F77 finite difference code applicable for uniform meshes. Both codes are hard to parallelize following the standard approach of inserting MPI commands directly into linear algebra operations. This is especially true for the legacy F77 code, which has a tangled internal data structure. Our objective is to build a hybrid parallel tsunami simulator based on these two codes, in a straightforward and effective way. To this end two light-weight new classes are programmed:

`class SubdomainBQFEMSolver` and `class SubdomainBQFDMSolver`

Here, class `SubdomainBQFEMSolver` uses double inheritance, as subclass of both `SubdomainBQSolver` and `Boussinesq`, so that it inherits the computational functionality from `Boussinesq` and at the same time is accepted by `HybridBQSolver` as a subdomain solver. Similarly, class `SubdomainBQFDMSolver` is derived from `SubdomainBQSolver` and at the same time "wraps up" the F77 subroutines of the legacy code inside its `solveLocal` function.

Using such a hybrid software framework, a parallel tsunami simulator has been built for the 2004 Indian Ocean tsunami. The entire spatial domain is depicted in Fig. 1, where the epicenter is located at position $(1,1)$. Moreover,

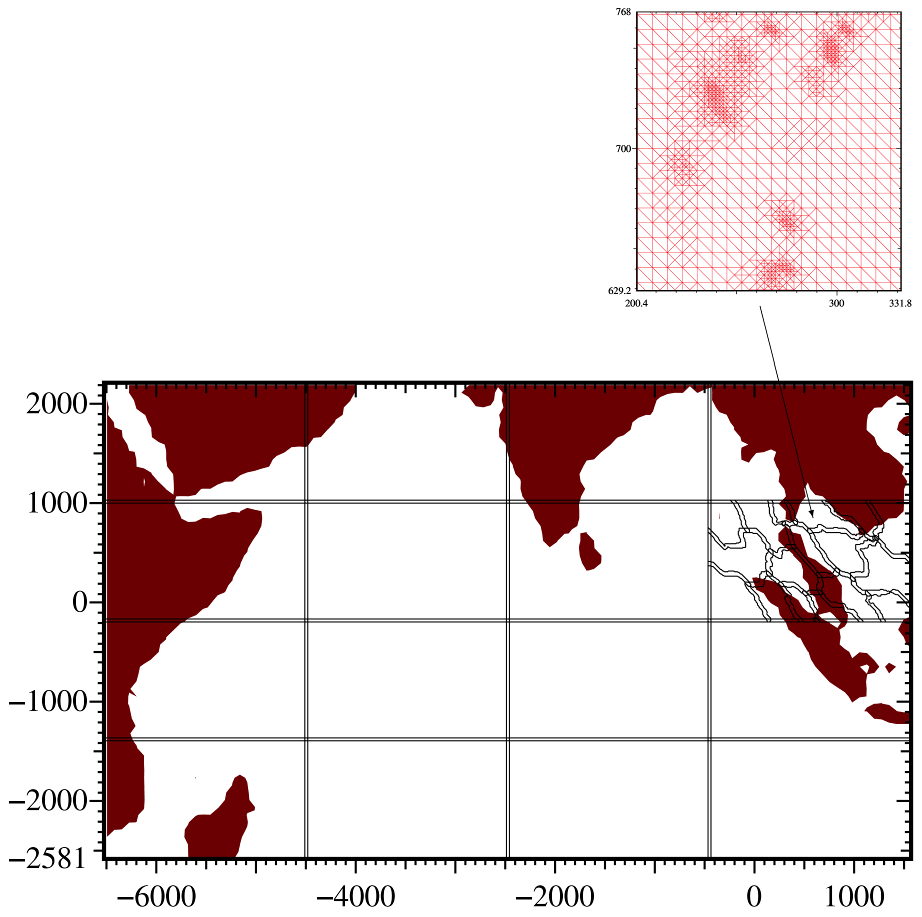


Fig. 1. An example of partitioning the Indian Ocean domain into a mixture of rectangular and complex-shaped subdomains. Finite differences are used by the rectangular subdomains to carry out the spatial discretization, whereas the complex-shaped subdomains use finite elements and adaptively refined local meshes.

the figure also shows different types of local meshes, i.e., uniform local meshes in the rectangular subdomains and adaptively refined local meshes in the complex-shaped subdomains. Different spatial discretizations (finite differences and finite elements) can thus be deployed in different regions. The simulation results have been reported in [4].

5 Concluding Remarks

We have explained a hybrid software framework for parallelizing and, at the same time, combining different existing serial codes. Such a parallelization strategy is numerically inspired by the additive Schwarz algorithms, while implementationally enabled by object-oriented programming techniques. The approach is particularly attractive for creating parallel simulators of wave propagation, as many old serial wave codes exist but are otherwise difficult to be parallelized. For ocean-scale simulations, the advantage of such a hybrid parallel simulator is that small areas of difficulty can be handled by subdomains that are equipped with an advanced mathematical model and a sophisticated numerical solver, whereas the remaining vast regions are handled by a simple mathematical model and fast code.

Future work will apply the software approach from this paper to other aspects of tsunami simulation, for instance, run-up of waves on beaches. Quite some sophisticated serial codes have been developed for the run-up problem, and these are hard to parallelize well. Our suggested approach makes parallelization feasible with little work. Of even more importance is the fact that reuse of very well-tested codes contributes to high reliability in a new hybrid, parallel simulator. For each new problem such as wave run-up it always remains, however, to investigate whether the additive Schwarz algorithm is capable of delivering satisfactory parallel efficiency.

References

1. Cai, X.: Overlapping domain decomposition methods. In: Langtangen, H.P., Tveito, A. (eds.) Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming, pp. 57–95. Springer, Heidelberg (2003)
2. Cai, X., Langtangen, H.P.: Parallelizing PDE solvers using the Python programming language. In: Bruaset, A.M., Tveito, A. (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers, Lecture Notes in Computational Science and Engineering, vol. 51, pp. 295–325. Springer, Heidelberg (2006)
3. Cai, X., Pedersen, G.K., Langtangen, H.P.: A parallel multi-subdomain strategy for solving Boussinesq water wave equations. Advances in Water Resources 28, 215–233 (2005)
4. Cai, X., Pedersen, G.K., Langtangen, H.P., Glimsdal, S.: Parallel simulation of tsunamis using a hybrid software approach. In: Joubert, G.R., et al. (eds.) Parallel Computing: Current & Future Issues of High-End Computing; Proceedings of the International Conference ParCo 2005, pp. 383–390. John von Neumann Institute for Computing (2006)
5. Gjevik, B., Pedersen, G., Dybesland, E., Harbitz, C.B., Miranda, P.M.A., Baptista, M.A., Mendes-Victor, L., Heinrich, P., Roche, R., Guesmia, M.: Modeling tsunamis from earthquake sources near Gorringe Bank southwest of Portugal. J. Geophysical Research 102, 931–949 (1997)
6. Glimsdal, S., Pedersen, G., Atakan, K., Harbitz, C.B., Langtangen, H.P., Løvholt, F.: Propagation of the Dec. 26, 2004, Indian Ocean Tsunami: Effects of dispersion and source characteristics. Int. J. Fluid Mech. Research 33, 15–43 (2006)
7. Glimsdal, S., Pedersen, G.K., Langtangen, H.P.: An investigation of overlapping domain decomposition methods for one-dimensional dispersive long wave equations. Advances in Water Resources 27, 1111–1133 (2004)

8. Okal, E.A., Synolakis, C.E.: Source discriminants for near-field tsunamis. Geophysical Journal International 158(3), 899–912 (2004)
9. Pedersen, G.: Three-dimensional wave patterns generated by moving disturbances at transcritical speeds. J. Fluid Mech. 196, 39–63 (1988)
10. Pedersen, G., Langtangen, H.P.: Dispersive effects on tsunamis. In: Proceedings of the International Conferance on Tsunamis, Paris, France, pp. 325–340 (1999)
11. Smith, B.F., Bjørstad, P.E., Gropp, W.: Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, Cambridge (1996)

HPC-MW: A Problem Solving Environment for Developing Parallel FEM Applications

Satoshi Ito[1] and Hiroshi Okuda[2]

[1] Collabolate Research Center for Computational Science and Technology, Institute of Industrial Science, The University of Tokyo, Komaba 4-6-1, Meguro-ku, 153-8505 Tokyo, Japan
sito@fsis.iis.u-tokyo.ac.jp
[2] Research into Artifacts, Center for Engineering (RACE), The University of Tokyo, Kashiwanoha 5-1-5, Kashiwa, 153-8505 Tokyo, Japan
okuda@race.u-tokyo.ac.jp

Abstract. Computational analysis requires huge amounts of computational resources. However, for the usual developer, parallelization is difficult as it requires special programming skills. In addition, since recently available architectures vary from PC-clusters to SMP clusters, the optimization strategies to make the best use of the hardware performance are no longer unique. In the present study, we develop a problem solving environment, which supports parallelization and optimizations for easily developing parallel FEM code. The load of code development on this PSE is evaluated. The parallel efficiency of the developed code is also discussed.

1 Introduction

Parallel computing involves unique techniques such as domain decomposition, message passing, and vectorization. The additional work required for implementing these techniques is extremely burdensome for application developers, often resulting in very time consuming development and buggy code. In addition, since recently available architectures vary from PC-clusters to SMP clusters, the optimization strategies used to efficiently exploit the available hardware vary from case to case. For this reason, the approach of Problem Solving Environment (PSE) [1,2,3,4,5,6,7,8,9] has been investigated by several researchers.

PSEs for software development support are categorized as being of the library-type or the middleware-type. The library-type PSE only provides some of the important functions in the applications as external subroutines. ScaLapack [6] is a package of linear matrix solvers, which are optimized for distributed-memory parallel computers. MpCCI [7] enables different analysis codes to be coupled by handling data exchange on the interface.

The middleware-type PSE gives some guideline for developing analysis code. POOMA [8] is a templated C++ library for developing parallel simulation code of partial differential equations. PCP [9] is a platform for developing parallel analysis codes. By building legacy code onto the sample code, parallelization is achieved.

PSE for software development support must provide three features: versatility, easy implementation, and optimization. Versatility means flexibility for algorithms and governing equations. The library-type PSE has advantages with respect to optimization and easy implementation, but a disadvantage with respect to versatility. The middleware-type PSE has advantages with respect to versatility and/or easy implementation. Therefore, we developed a new PSE "HPC middleware" (HPC-MW) [10] which provides the above three features.

2 HPC-MW

As mentioned previously, the HPC middleware (HPC-MW) must provide versatility, easy implementation, and optimization. For versatility, the finite element method is employed as the base of this PSE because FEM is a general discretization method of partial differential equations.

2.1 Easy Implementation

For easy implementation, two requirements must be satisfied: provision of FEM common procedures and automatic parallelization.

The FEM consists of four main functions: I/O, construction of the global stiffness matrix, calculation of the right-hand side vector, and the solver. By providing these functions, the application developer can concentrate on the algorithm. The domain decomposition method (DDM) is employed for parallelization. In this method, message massing is needed for some processes.

I/O. Input data must be distributed for each processor in parallel computing. There are two approaches for data distribution. In the entire data approach, a certain processor reads the data in its entirety and distributes the data to other processors. The other approach is the distributed data approach, in which each processor reads decomposed data. The latter is inevitable in large-scale analysis. Output is simply performed for each processor by writing the results to a file.

Construction of the Global Stiffness Matrix. This function is composed of three processes: Jacobian calculation, calculation of the element matrix, and assembling the element matrix to the global matrix. Jacobian calculation depends only on the type of element so that it should be included in HPC-MW.

Assembling the element matrix to the global stiffness matrix must be provided because the CRS format (described in Section 2.2) [11] has to be used.

Calculation of the Right-Hand Side Vector. In this function, operations such as matrix-vector multiplication and vector inner product are needed. These operations also require the CRS format. Thus, subroutines for matrix-vector multiplication and vector inner product must be supported. The block matrix method is employed so that the subroutines used for matrix-vector multiplication have some types that correspond to the typical block size.

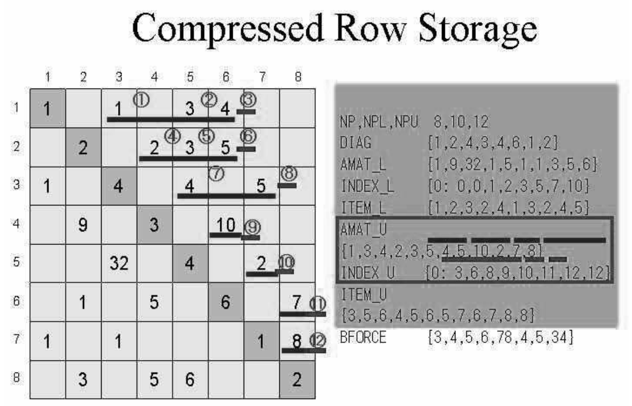
Table 1. Examples of HPC-MW subroutines

I/O	
`hpcmw_get_mesh`	Input mesh data
`hpcmw_write_result`	Output result data
Construction of the global matrix	
`hpcmw_mat_con`	Create CRS table
`hpcmw_matrix_allocate`	Allocate of a matrix memory
`hpcmw_Jacob`	Calculate shape functions
`hpcmw_matrix_assemble`	Assemble an element matrix to global matrix
Calculation of the right-hand side vector	
`hpcmw_matrix_vector`	Matrix-vector multiplication
`hpcmw_vector_innerProduct`	Vector inner product
Solver	
`hpcmw_solver_11`	Linear matrix solvers for CG, GMRES, etc

The Solver. The solver dominates the performance of the developed code, thus it is significant. Many optimized solvers, such as CG and GMRES, are prepared in HPC-MW. Powerful preconditioners are also included.

Table 1 shows examples of subroutines from HPC-MW. All functions inevitable for the FEM procedure are successfully provided.

For parallelization, message passing should be included in HPC-MW. The functions that require message passing include matrix-vector multiplication, the vector inner product, and the solvers. By adapting these functions for message passing, the application developer is free from parallelization.



```
NP,NPL,NPU   8,10,12
DIAG         [1,2,4,3,4,6,1,2]
AMAT_L       [1,9,32,1,5,1,1,3,5,6]
INDEX_L      [0: 0,0,1,2,3,5,7,10]
ITEM_L       [1,2,3,2,4,1,3,2,4,5]
AMAT_U
[1,3,4,2,3,5,4,5,10,2,7,8]
INDEX_U      [0: 3,6,8,9,10,11,12,12]
ITEM_U
[3,5,6,4,5,6,5,7,6,7,8,8]
BFORCE       [3,4,5,6,78,4,5,34]
```

Fig. 1. An example of compressed row storage (CRS) format is shown. In this format, a matrix is stored by three one-dimensional arrays. One stores the number of column of the component, and the other stores the number of non-zero components in the row and the other stores the value of the component.

2.2 Optimization

Since recently available architectures vary from PC-clusters to SMP clusters, the optimization strategies used to efficiently exploit the available hardware vary from case to case. However, the optimization strategy always depends on the data structure. Here, the data structure indicates how to store the data on the memory. In HPC-MW, CRS format [11] is employed.

Figure 1 shows an example of the CRS format. This format is profitable for memory usage and computational cost in large-scale problems because it stores only the non-zero components of a matrix. The process that uses the CRS format is matrix-vector multiplication and assembling the element matrix to the global matrix. These two processes were optimized previously.

3 Example of Application Development

In this section, we explain how to apply HPC-MW for the development of analysis code. Here, incompressible fluid analysis code was developed. This phenomenon is appropriate for evaluating the developed PSE because of its characteristics of non-linearity and time-dependency.

Figure 2 shows the utilization image of HPC-MW. The method by which to use HPC-MW will be explained in the following sections.

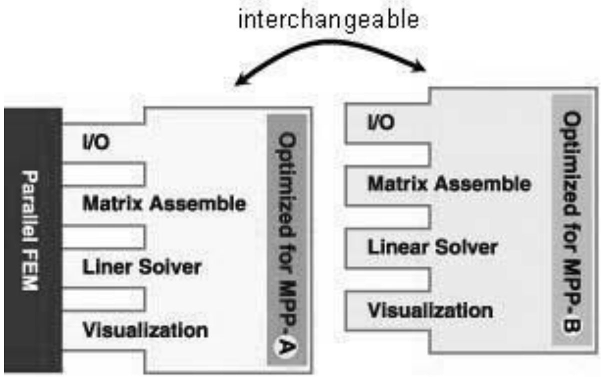
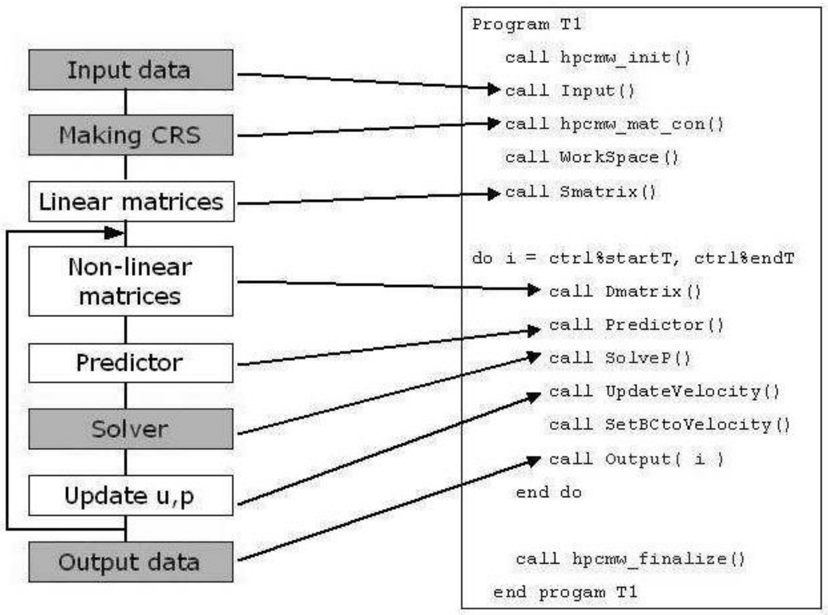


Fig. 2. This figure shows the utilization image of HPC-MW. Optimized parallel code is generated by special language/compiler based on analysis data and H/W information.

3.1 Algorithms and Development Process

The governing equations are the Navier-Stokes equations and the continuum equation. The P1P1 finite element is employed for spatial discretization, and the predictor-multicorrector method [12] is applied for temporal discretization. For stabilization,the SUPG/PSPG method [12] is also employed. Navier-Stokes



```
Program T1
   call hpcmw_init()
   call Input()
   call hpcmw_mat_con()
   call WorkSpace()
   call Smatrix()

do i = ctrl%startT, ctrl%endT
     call Dmatrix()
     call Predictor()
     call SolveP()
     call UpdateVelocity()
     call SetBCtoVelocity()
     call Output( i )
   end do

   call hpcmw_finalize()
  end progam T1
```

Fig. 3. Flowchart and the main part of the source of the developed code. This algorithm includes seven processes: Input, CRS data, Construction of Global Matrix, Predictor, Solver, and Output. The four processes that are colored in the figure are entirely supported by HPCMW functions.

equations are applied in the SUPG, and the continuum equation is applied in the PSPG, as follows.

$$\int_{\Omega} w_i \cdot \rho \left(\frac{\partial u_i}{\partial t} + u_j u_{i,j} - f_i \right) d\Omega - \int_{\Omega} w_i \sigma_{ij,j} d\Omega$$
$$+ \sum_e \int_{\Omega_e} \tau u_j w_{i,j} \cdot \left[\rho \left(\frac{\partial u_i}{\partial t} + u_j u_{i,j} - f_i \right) - \sigma_{ik,k} \right] d\Omega = 0. \tag{1}$$

$$\int_{\Omega} q \;\; u_{i,i} d\Omega + \sum_e \int_{\Omega_e} \tau \frac{1}{\rho} q_{,j} \left[\rho \left(\frac{\partial u_i}{\partial t} + u_j u_{i,j} - f_i \right) - \sigma_{ij,j} \right] d\Omega = 0. \tag{2}$$

$$\sigma_{ij} = -p\delta_{ij} + \mu \left(u_{i,j} + u_{j,i} \right). \tag{3}$$

Here, u denotes the velocity, p denotes the pressure, ρ denotes the density, w denotes the weight function for the Navier-Stokes equations, q denotes the weight function for the continuum equation, and τ denotes the stabilization coefficient, and μ denotes viscousity coefficient, and δ denotes Kronecker delta. Figure 3 shows a flowchart and the source of the main part of the developed code. According to Fig. 3, the algorithm and the structure of the source are well synchronized.

The program source of the mass matrix construction is shown below.

Program source of mass matrix construction

```
subroutine MakeMass

  do iElem = 1, hpcMESH%n_elem
     localMAT = 0.0
     call hpcmw_Jacob_341(iElem, volume, b, c, d)

     do i = 1, 4
        do j = 1, 4
           if ( i == j ) then
           localMAT(i,j,1) = localMAT(i,j,1) + rho * volume / 10.0
           else
           localMAT(i,j,1) = localMAT(i,j,1) + rho * volume / 20.0
           end if
        end do
     end do

    call hpcmw_matrix_assemble(iElem, 4, 1, localMAT, massMAT)

  end do

end subroutine MakeMass
```

Equation (4) expresses the weak-form of the element mass matrix equation. Here, both α and β denote the local node numbers, ρ denotes the density, V_e denotes element volume, and N denotes the shape function. This integration can be expanded as Eq. (4) if a P1P1 tetrahedral element is employed.

$$M_{\alpha\beta} = \int N_\alpha N_\beta d\Omega = \rho \frac{V_e}{20} \begin{bmatrix} 2 & & & 1 \\ & 2 & & \\ & & 2 & \\ 1 & & & 2 \end{bmatrix} . \tag{4}$$

In order to calculate the mass matrix, the following three steps are required:

1. Calculate V_e.
2. Calculate element mass matrices using ρ and V_e.
3. Assemble the element matrices to the global matrix.

According to the program source,V_e was calculated by the subroutine `hpcmw_Jacob_341` (step 1), and the calculated element matrix was assembled to the global matrix by the subroutine `hpcmw_matrix_assemble` automatically (step 3).

4 Numerical Results

4.1 Parallel Efficiency

The parallel efficiency of the developed code was measured. The simulated model is a lid-driven cavity flow. The number of nodes was 1,000,000, and the number of d.o.f.fs was 4,000,000. The specifications of the PC-cluster are shown in Table 2. Figure 4 shows the speed-up of the solver provided by HPC-MW. The definition of the speed-up is given by Eq. (5), in which S denotes the speed-up, T denotes the calculation time, and the subscript n denotes the number of processors. A parallel efficiency of approximately 94% is achieved with 24 processors for a mid-sized model.

$$S = \frac{8T_8}{T_n} \tag{5}$$

Table 2. Specification of the PC-cluster

CPU	Xeon 2.8GHz
Memory	2 Gbyte
Network	Myrinet

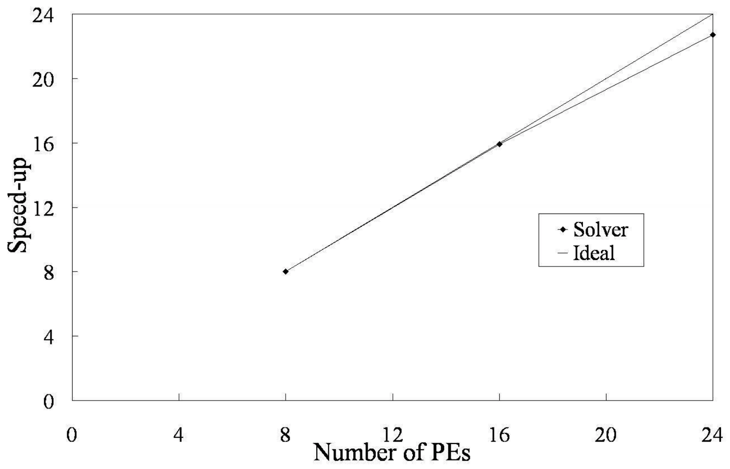


Fig. 4. Speed-up of the solver. The cases of 8, 16 and 24 processors are shown.

4.2 Development Efficiency

Table 3 shows the number of steps in the developed code. The number of steps is 1,812. We estimate the number of steps that are covered by HPC-MW functions. The estimation is 2,450. Thus, we consider the total steps of the developed code as 4,250.

According to the estimation, approximately 57% of the steps are covered by HPC-MW.

Table 3. Number of steps in the developed code

	Number of steps	Ratio(%)
Total	4250	100
Algorithm	1812	42.6
HPC-MW	2438	57.4
Details of HPC-MW		
CRS data	245	5.76
Matrix assemble	55	1.29
Jacobian	62	1.46
Matrix-vector multiplication	571	13.4
Solver (with precond.)	1388	32.6

5 Conclusions

A PSE for an FEM application developer was designed and developed. Various procedures common to FEM are provided by the PSE, and parallelization is automatically achieved. An incompressible fluid analysis code was developed using this PSE. The developed PSE saved approximately 57% with respect to the number of steps of developed code. A parallel efficiency of approximately 94% was obtained for a mid-sized model run on a PC-cluster.

Acknowledgments. This research was performed as part of the "Revolutionary Simulation Software for the 21st century (RSS21)" project supported by the next-generation IT program of the Ministry of Education, Culture, Sports, Science and Technology (MEXT).

References

1. Houstis, E.N, Rice, J.R., Gallopoulos, E., Bramley, R. (eds.): Enabling Technologies for Computational Science. Kluwer Academic Publishers, Dordrecht (2000)
2. Boonmee, C., Kawata, S.: Computer-Assisted Simulation Environment for Partial-Differential-Equation Problem: 1. Data Structure and Steering of Problem Solving Process. Trans. of the Japan Soc. for Compt. Engnrg. and Science Paper No. 1998001 (1998)
3. Boonmee, C., Kawata, S.: Computer-Assisted Simulation Environment for Partial-Differential-Equation Problem: 2. Visualization and Steering of Problem Solving Process. Trans. of the Japan Soc. for Compt. Engng. and Science Paper No. 1998002 (1998)
4. Hirayama, Y., Ishida, J., Ota, T., Igaki, M., Kubo, S., Yamaga, S.: Physical Simulation using Numerical Simulation Tool PSILAB. In: The 1st Problem Solving Environment Workshop, pp. 1–7 (1998)
5. Kawata, S., Fujita, A., Machide, S., Hirama, T., Yoshimoto, K.: Computer-Assisted Simulation System NCAS and a Future PSE. In: Proc. the 2nd Problem Solving Environment (PSE) Workshop, pp. 13–18 (1999)
6. `http://www.netlib.org/scalapack/`

7. http://www.scai.fraunhofer.de/mpcci.html
8. http://acts.nersc.gov/pooma/
9. Tezuka, A., Matsumoto, J., Matsubara, K.: Development of Common Software Platform on Parallel Computations for Discretized Numerical Schemes and its Application to Finite Element Fluid Dynamics. Int. J. Comp. Fluid Dyn. 18(4), 347–354 (2004)
10. http://www.fsis.iis.u-tokyo.ac.jp/en/theme/hpc/
11. Nakajima, K.: Preconditioned Iterative Linear Solvers for Unstructured Grids on the Earth Simulator. In: 7th International Conference on High Performance Computing and Grid in Asia Pacific Region (2004)
12. Tezduyar, T.E.: Incompressible flow computations with stabilized bilinear and linear equal-order-interpolation velocity-pressure elements. Compt. Methods in Appl. Mech. and Engng. 95, 221–242 (1992)

SyFi - An Element Matrix Factory

Kent-Andre Mardal

Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway
kent-and@simula.no
http://www.simula.no/portal_memberdata/kent-and

Abstract. SyFi is an open source C++ library for defining and using variational forms and finite elements based on symbolic representations of polygonal domains, degrees of freedom and polynomial spaces. Once the finite elements and variational forms are defined, they are used to generate efficient C/C++ code.

1 Introduction

SyFi [15], which stands for Symbolic Finite Elements, is a C++ library for finite element computations. SyFi is equipped with a Python interface by using SWIG [16]. It relies on the symbolic mathematics library GiNaC [9] and the Python interface to GiNaC called Swiginac [17]. SyFi, Swiginac, and GiNaC are open source.

This paper is only a short overview of the SyFi project in the context of finite element methods for the incompressible Navier-Stokes equations. A more comprehensive description of the project can be found on its webpage `http://syfi.sf.net`, which contains a tutorial, a complete reference and the source code. We will show various code snippets in this paper. The complete code examples can be found in the subdirectory `para06` in the SyFi source code tree.

There are quite a few other projects that are similar in various respects to SyFi. Within the FEniCS [4] project there are two Python projects: FIAT [6] and FFC [5]. FIAT is a Python module for defining finite elements while FFC generates C++ code based on a high–level Python description of variational forms. The DSEL project [3] employs high–level C++ programming techniques such as expression templates and meta-programming for defining variational forms, performing automatic differentiation, interpolation and more. Sundance [14] is a C++ library with a powerful symbolic engine which supports automatic generation of a discrete system for a given variational form. Analysa [1], GetDP [8], and FreeFem++ [7] define domain-specific languages for finite element computations. The main difference between SyFi and the other projects is that it uses a high level symbolic framework in Python to generate efficient C/C++ code.

A key point in the design of SyFi is, as already mentioned, that we want to employ symbolic mathematics and code generation in place of the numerics. The powerful symbolic engine GiNaC and the combination of the high–level languages C++ and Python have so far proven to be a solid platform. Consider for instance, the computation of an entry in the mass matrix

$$A_{ij} = \int_{\hat{T}} N_i N_j \, D \, dx, \tag{1}$$

where D is the Jacobian of the geometry mapping between the reference element $\hat{T}$ and the global element T, and $\{N_i\}_i$ are the finite element basis functions. If the geometry mapping is affine, the computation of one matrix entry based on (1) will result in a real number times D. Because everything but the multiplication with D (in case of a mass matrix) can be precomputed, the generated code will be very efficient compared to traditional codes, which typically implements a loop over quadrature points and numerical evaluation of the finite element basis functions. See also `mass.py` for a demonstration of such code generation.

As will be explained later, other advantages of this approach include an easy way of defining finite elements, and straightforward computation of the Jacobian in the case of nonlinear PDEs.

2 Using Finite Elements and Evaluating Variational Forms

One main goal with SyFi has been that it should be a tool with *strong support for differentiation and integration of polynomials on polygonal domains*, which are basic ingredients both when defining finite elements and using finite elements to define variational forms. Many finite elements have been implemented in SyFi. Of particular importance for the simulation of incompressible fluids are the continuous and discontinuous Lagrangian elements of arbitrary order and the Crouzeix-Raviart element [2]. However, also the H(div)-Raviart-Thomas elements [13] and the H(curl)- Nedelec elements [11,12] of arbitrary order have been implemented. We will come back to the construction of finite elements in Section 4. In this section we concentrate on the usage of already implemented elements.

We construct the commonly used Taylor–Hood element $\mathbb{P}_2^2 - \mathbb{P}_1$ as follows (see also `div.py`),

```
from swiginac import *
from SyFi import *

polygon = ReferenceTriangle()

v_element = VectorLagrangeFE(polygon,2)
v_element.set_size(2)
v_element.compute_basis_functions()

p_element = LagrangeFE(polygon,1)
p_element.compute_basis_functions()
```

The polygonal domain here is a reference triangle, but it may be a line, a triangle, a square, a tetrahedron or a box. Furthermore, these geometries are not limited to typical reference geometries. For instance, we may construct the elements on a global triangle defined by the points (x_0, y_0), (x_1, y_1), and (x_2, y_2) where $x_0, \dots, y_2$ might be both numbers and/or symbols. The following code shows the Taylor–Hood element on a triangle defined in terms of the symbols $x_0, \dots, y_2$, (see also `div_global.py`),

```
x0 = symbol("x0"); y0 = symbol("y0")
x1 = symbol("x1"); y1 = symbol("y1")
x2 = symbol("x2"); y2 = symbol("y2")

p0 = [x0,y0]; p1 = [x1,y1]; p2 = [x2,y2]

polygon = Triangle(p0,p1,p2)
v_element = VectorLagrangeFE(polygon,2)
v_element.set_size(2)
v_element.compute_basis_functions()

p_element = LagrangeFE(polygon,1)
p_element.compute_basis_functions()
```

The computed basis functions are standard polynomials also in this case, although they depend on $x_0, \dots, y_2$. These polynomials can be added, multiplied, differentiated, integrated etc. in the standard way (within a symbolic framework). Consider for example, the computation of the divergence constraint,

$$B_{ij} = \int_T \operatorname{div} \boldsymbol{N}_i \, L_j \, dx,$$

where $\boldsymbol{N}_i$ and L_j are the basis functions for the velocity and pressure elements, respectively, and T is a polygonal domain. This matrix can be computed as follows (see also `div_global.py`):

```
... construct the element

for i in range(0,v_element.nbf()):
    for j in range(0,p_element.nbf()):
        integrand = div(v_element.N(i))*p_element.N(j)
        Bij =  polygon.integrate(integrand)
```

Another example that demonstrates the power of this approach, in which we utilize a symbolic mathematics engine, is the computation of the Jacobian of the nonlinear convection-diffusion equations that typically appear in incompressible flow simulations. Let

$$\boldsymbol{F}_i = \int_T (\boldsymbol{u} \cdot \nabla \boldsymbol{u}) \cdot \boldsymbol{N}_i + \nabla \boldsymbol{u} : \nabla \boldsymbol{N}_i \, dx,$$

where $\boldsymbol{u} = \sum_k u_k \boldsymbol{N}_k$. Then,

$$\boldsymbol{J}_{ij} = \frac{\partial \boldsymbol{F}_i}{\partial u_j} = \frac{\partial}{\partial u_j} \int_T (\boldsymbol{u} \cdot \nabla \boldsymbol{u}) \cdot \boldsymbol{N}_i + \nabla \boldsymbol{u} : \nabla \boldsymbol{N}_i \, dx. \tag{2}$$

The computation of such Jacobian matrices and the implementation of corresponding simulation software are usually tedious and error-prone. It seems that one main reason for this difficulty is the gap between the computations done by hand and the corresponding numerical algorithm to be implemented. After all, the computation of (2) only involves straightforward operations. SyFi aims at closing this gap. We will now show the code for computing (2) with SyFi. The complete source code is in `conv-diffusion.py`. First, we compute the finite elements as shown in the previous example. Secondly, we compute the $\boldsymbol{F}_i$ and differentiate to get the Jacobian:

```
u, ujs = sum("u", fe)

for i in range(0,fe.nbf()):

  # compute diffusion term
  fi_diffusion = inner(grad(u), grad(fe.N(i)))

  # compute convection term
  uxgradu = (u.transpose()*grad(u)).evalm()
  fi_convection = inner(uxgradu, fe.N(i), True)

  # add together diffusion and convection
  fi = fi_diffusion + fi_convection

  # compute the integral
  Fi = polygon.integrate(fi)

  for j in range(0,fe.nbf()):
    # differentiate to get the Jacobian
    uj = ujs.op(j)
    Jij = diff(Fi, uj)
    #print out the Jacobian
    print "J[%d,%d]=%s;\n"%(i,j,Jij)
```

The output from `conv-diffusion.py` is:

```
J[0,0]=1+1/24*u2-1/12*u1-1/24*u5-1/6*u0-1/24*u4-1/24*u3;
J[0,1]=-1/12*u0+1/12*u4;
J[0,2]=-1/2+1/12*u2+1/24*u0+1/24*u4;
J[0,3]=-1/24*u0+1/24*u4;
J[0,4]=-1/2+1/24*u2+1/12*u1+1/24*u5-1/24*u0+1/24*u3;
...
```

We can now extend the above code such that it also can include the Ostwald–de Waele power-law viscosity model, i.e.,

$$\boldsymbol{F}_i^p = \int_T (\boldsymbol{u} \cdot \nabla \boldsymbol{u}) \cdot \boldsymbol{N}_i + \mu(\boldsymbol{u}) \nabla \boldsymbol{u} : \nabla \boldsymbol{N}_i \, dx,$$

where $\mu = \mu_0 \|\nabla \boldsymbol{u}\|^n$. The Jacobian matrix is then

$$\boldsymbol{J}_{ij}^p = \frac{\partial \boldsymbol{F}_i^p}{\partial u_j}.$$

The only thing we need to change then in the above script is the diffusion term (see also `conv-diffusion-power-law.py`):

```
  # nonlinear power-law diffusion term
  mu = inner(grad(u), grad(u))
  fi_diffusion = mu0*pow(mu,n)*inner(grad(u), grad(fe.N(i)))
```

In addition, we also need to declare n and μ_0 to be either symbols or numbers.

3 Code Generation for Quadrature Based FEM Systems

SyFi can also be used to generate C++ code for other FEM systems. We will here consider code generation for finite element basis functions in a format specified by the user. Other code generation examples can be found in the SyFi tutorial and source code, where code for creating both PyCC and Epetra matrices for various problems are generated. Furthermore, notice that one can print the expressions out in either of the formats: ASCII, C, LaTeX, and Python.

The following code demonstrates how C code for the basis functions is generated (see also `code_gen_simple.py`):

```
polygon = ReferenceTriangle()
fe = LagrangeFE(polygon,2)
fe.compute_basis_functions()

N_string = ""
for i in range(0,fe.nbf()):
    N_string += "    N[%d]=%s;\n"% (i, fe.N(i).printc())

c_code = """
void basis2D(double N[%d], double x, double y) {
%s
} """ % (fe.nbf(), N_string)

print c_code
```

Notice that C code for the expressions is generated with the function `printc`. The output from `code_gen_simple.py` is:

```
void basis2D(double N[6], double x, double y) {
    N[0]=pow(-y-x+1.0,2.0)-(-y-x+1.0)*y-(-y-x+1.0)*x;
    N[1]=4.0*(-y-x+1.0)*x;
    N[2]=-y*x+(x*x)-(-y-x+1.0)*x;
    N[3]=4.0*(-y-x+1.0)*y;
    N[4]=4.0*y*x;
    N[5]=-y*x+(y*y)-(-y-x+1.0)*y;
}
```

Finally, notice that to change the above code to produce code for, e.g., 5th order elements all you need to do is change the degree of the element i.e.,

```
polygon = ReferenceTriangle()
fe = LagrangeFE(polygon,5)
fe.compute_basis_functions()

...
```

4 Defining a Finite Element in SyFi

Defining a finite element may of course be more technical than using it, in particular for advanced elements. Furthermore, the implementation shown below involves more of GiNaC and SyFi than the earlier examples, so the reader should have access to both the SyFi and GiNaC tutorial. The elements implemented in SyFi so far have mostly been implemented in C++ since they then will be available in both C++ and Python (by using SWIG).

We will describe the implementation of an element recently added to SyFi. The element was introduced in [10]. The special feature of this element is that it works well for both Darcy and Stokes types of flow.

The definition of the element is as follows,

$$\boldsymbol{V}(T) = \{\boldsymbol{v} \in \mathbb{P}_3^2 \ : \ \operatorname{div} \boldsymbol{v} \in \mathbb{P}_0, \ (\boldsymbol{v} \cdot \boldsymbol{n}_e)|_e \in \mathbb{P}_1 \ \forall e \in E(T)\},$$

where T is a given triangle, $E(T)$ is the edges of T, $\boldsymbol{n}_e$ is the normal vector on edge e, and $\mathbb{P}_k$ is the space of polynomials of degree k and $\mathbb{P}_k^d$ the corresponding vector space. The degrees of freedom are,

$$\begin{aligned} &\int_e (\boldsymbol{v} \cdot \boldsymbol{n})\tau^k \, d\tau, \quad k = 0, 1, && \forall e \in E(T), \\ &\int_e (\boldsymbol{v} \cdot \boldsymbol{t}) \, d\tau, && \forall e \in E(T). \end{aligned}$$

The definition of the element is more complicated than most of the common elements. Still, we will show that it can be implemented in SyFi in about 100 lines of codes. We will compute this element in four steps:

1. Constructing the polynomial space $\boldsymbol{V}(T)$.
2. Specifying the constraints.
3. Specifying the degrees of freedom.
4. Solving the resulting linear system of equations.

Considering the first step, SyFi implements the Bernstein polynomials (in barycentric coordinates) with the functions `bernstein` and `bernsteinv`, for scalar and vector polynomials, respectively. The `bernstein` functions returns a list (`lst`) with the items:

– The polynomial $(a_0 x + a_1 y + a_2(1 - x - y) + \ldots)$.
– The variables $(a_0, a_1, a_2, \ldots)$.
– The polynomial basis $(x, y, 1 - x - y, \ldots)$.

In the following we construct $\mathbb{P}_3^2$:

```
Triangle triangle
ex V_space = bernsteinv(2, 3, triangle, "a");
ex V_polynomial = V_space.op(0);
ex V_variables = V_space.op(1);
```

Here V_space is the above mentioned list, V_polynomial contains the polynomial, and V_variables contains the variables.

In the second step we first specify the constraint $\operatorname{div} \boldsymbol{v} \in \mathbb{P}_0$:

```
lst equations;
ex divV = div(V);
ex_ex_map b2c = pol2basisandcoeff(divV);
ex_ex_it iter;
// div constraints:
for (iter = b2c.begin(); iter != b2c.end(); iter++) {
    ex basis = (*iter).first;
    ex coeff= (*iter).second;
    if ( coeff != 0 && ( basis.degree(x) > 0
         || basis.degree(y)  > 0 )  ) {
        equations.append( coeff == 0 );
    }
}
```

Here, the divergence is computed with the div function. The divergence of a function in $\mathbb{P}_3^2$ is in $\mathbb{P}_2$. Hence, it is on the form $b_0+b_1x+b_2y+b_3xy+b_4x^2+b_5y^2$. In the above code we find the coefficients b_i, as expressions involving the above mentioned variables a_i and the corresponding polynomial basis, with the function pol2basisandcoeff. Then we ensure that the only coefficient which is not zero is b_0.

The next constraints $(\boldsymbol{v}\cdot\boldsymbol{n}_e)|_e \in \mathbb{P}_1$ are implemented in much of the same way as the divergence constraint. We create a loop over each edge e of the triangle and multiply $\boldsymbol{v}$ with the normal $\boldsymbol{n}_e$. Then we substitute the expression for the edge, i.e., in mathematical notation $|_e$, into $\boldsymbol{v}\cdot\boldsymbol{n}$. After substituting the expression for these lines to get $(\boldsymbol{v}\cdot\boldsymbol{n}_e)|_e$, we check that the remaining polynomial is in $\mathbb{P}_1$ in the same way as we did above.

```
// constraints on edges:
for (int i=1; i<= 3; i++) {
    Line line = triangle.line(i);
    symbol s("s");
    lst normal_vec = normal(triangle, i);
    ex Vn = inner(V, normal_vec);
    Vn = Vn.subs(line.repr(s).op(0)).subs(line.repr(s).op(1));
    b2c = pol2basisandcoeff(Vn,s);
    for (iter = b2c.begin(); iter != b2c.end(); iter++) {
        ex basis = (*iter).first;
        ex coeff= (*iter).second;
        if ( coeff != 0 &&  basis.degree(s) > 1 )
        {
            equations.append( coeff == 0 );
        }
    }
}
```

In the third step we specify the degrees of freedom. First, we specify the equations coming from $\int_e (\boldsymbol{v}\cdot\boldsymbol{n})\tau^k d\tau, k=0,1$ on all edges. To do this we need to create a loop over all edges, and on each edge we create the space of linear

Bernstein polynomials in barycentric coordinates on e, i.e., $\mathbb{P}_1(e)$. Then we create a loop over the basis functions τ^k in $\mathbb{P}_1(e)$ and compute the integral $\int_e (\boldsymbol{v}\cdot\boldsymbol{n})\tau^k \, d\tau$.

```
// dofs related to the normal on the edges
for (int i=1; i<= 3; i++) {
    Line line = triangle.line(i);
    lst normal_vec = normal(triangle, i);
    ex P1_space = bernstein(1, line, istr("a",i));
    ex P1 = P1_space.op(2);
    ex Vn = inner(V, normal_vec);

    ex basis;
    for (int j=0; j< P1.nops(); j++) {
        basis = P1.op(j);
        ex integrand = Vn*basis;
        ex dofi =  line.integrate(integrand);
        dofs.insert(dofs.end(), lst(line.vertex(0),
                line.vertex(1), j));
        ex eq = dofi == numeric(0);
        equations.append(eq);

    }
}
```

Finally, the degrees of freedom $\int_e (\boldsymbol{v} \cdot \boldsymbol{t}) d\tau$ can be implemented in basically the same fashion as the previously described degrees of freedom. To summarize, we have now specified 20 equations which is precisely the number of unknowns in $\mathbb{P}_3^2$. Hence, the space $\boldsymbol{V}(T)$ is uniquely defined, what remains is simply to solve a linear system with 20 equations and 20 unknowns. The complete source code is in `Robust.cpp`.

5 Summary

In this paper we have tried to demonstrate that symbolic mathematics combined with code generation can be an alternative to the traditional numerical approach for implementing finite elements and finite element methods. By combining Python, C++ and legacy libraries we have created a library which is both easy to use and powerful enough for advanced methods and complicated PDEs. Furthmore, the generated code is often efficient compared to the traditional quadrature based approach.

References

1. Bagheri, B., Scott, L.R.: Analysa software package, `http://people.cs.uchicago.edu/~ridg/al/aa.html`
2. Crouzeix, M., Raviart, P.A.: Conforming and non–conforming finite element methods for solving the stationary stokes equations. RAIRO Anal. Numér. 7, 33–76 (1973)
3. Prud'homme, C.: DSEL software package, `http://www.hpc2n.umu.se/para06/papers/paper_147.pdf`

4. Dupont, T., Hoffman, J., Jansson, J., Johnson, C., Kirby, R.C., Knepley, M., Larson, M., Logg, A., Scott, R., Wells, G.N.: FEniCS software package, http://www.fenics.org
5. Logg, A.: FFC software package, http://www.fenics.org/ffc/
6. Kirby, R.C.: FIAT software package, http://www.fenics.org/fiat/
7. Pironneau, O., Hecht, F., Hyaric, A.L.: FreeFEM software package, http://www.freefem.org/ff++/index.htm
8. Dular, P., Geuzaine, C.: GetDP software package, http://www.geuz.org/getdp/
9. Bauer, C., Dams, C., Frink, A., Kisil, V.V., Kreckel, R., Sheplyakov, A., Vollinga, J.: GiNaC - is not a CAS, http://www.ginac.de
10. Mardal, K.-A., Tai, X.-C., Winther, R.: A robust finite element method for Darcy–Stokes flow. SIAM J. Numer. Anal. 40, 1605–1631 (2002)
11. Nédélec, J.-C.: Mixed finite elements in R^3 35(3), 315–341(October 1980)
12. Nédélec, J.-C.: A new family of mixed finite elements in R^3 50(1), 57–81 (November 1986)
13. Raviart, P.A., Thomas, J.M.: A mixed finite element method for 2-order elliptic problems. Matematical Aspects of Finite Element Methods (1977)
14. Long, K.: Sundance software package, http://software.sandia.gov/sundance/
15. Mardal, K.-A.: SyFi - Symbolic Finite Elements, http://www.fenics.org/syfi
16. Beazley, D., et al.: SWIG - Simplified Wrapper and Interface Generator, http://www.swig.org
17. Skavhaug, O., Certik, O.: Swiginac - Python interface to GiNaC, http://swiginac.berlios.de/
18. Heroux, M., et al.: Trilinos, http://software.sandia.gov/trilinos/

Life: Overview of a Unified C++ Implementation of the Finite and Spectral Element Methods in 1D, 2D and 3D

Christophe Prud'homme

Université Joseph Fourier, Laboratoire Jean Kuntzmann,
51 rue des Mathématiques, 38041 Grenoble, France
christophe.prudhomme@ujf-grenoble.fr

Abstract. This article presents an overview of a unified framework for finite element and spectral element methods in 1D, 2D and 3D in C++ called Life. The objectives of this framework are quite ambitious and could be expressed in various ways: *(i)* the creation of a versatile mathematical kernel allowing for easily solving problems using different techniques thus allowing testing and comparing methods, e.g. cG versus dG, *(ii)* the creation of a *small* and *manageable* library which shall nevertheless encompass a wide range of numerical methods and techniques, and *(iii)* build mathematical software that follows closely the mathematical abstractions associated with the partial differential equations to be solved.

1 Introduction and Basic Principles

This article presents an overview of a unified framework for finite element and spectral element methods in 1D, 2D and 3D in C++, called Life. The objectives of this framework is quite ambitious and could be expressed in various ways such as *(i)* the creation of a versatile mathematical kernel easily solving problems using different techniques thus allowing testing and comparing methods, e.g. continuous Galerkin (cG) versus discontinuous Galerkin (dG) , *(ii)* the creation of a *small* and *manageable* library which shall nevertheless encompass a wide range of numerical methods and techniques, *(iii)* build mathematical software that follows closely the mathematical abstractions associated with partial differential equations (PDE) — which is often not the case, for example the design could be physics oriented, — and *(iv)* the creation of a library entirely in C++ allowing to create complex multi-physics applications such as fluid-structure interaction or mass transport in haemodynamics — the rationale being that these applications are computing intensive and the use of an interpreted language such as Python would not be satisfying though in many simpler cases that would simplify and accelerate the development.

Now we describe a few requirements or features and general considerations that the library tries to satisfy:

- The syntax, the semantics and the pragmatics of the library are very close to the mathematics; the design and implementation should be mathematics oriented. *C++* supports very well multiple paradigms design and offers a wide range of solutions for a given problem. Generic programming, OO programming, meta-programming are such paradigms and they are definitely very useful when dealing with mathematical abstractions.
- The library shall remain small and manageable and make use wherever possible of established libraries. In particular, it uses most of the Boost[1] libraries. To name but a few: ublas, lambda, mpl, preprocessor, serialization, multi_index.
- The library should compare reasonably well with other similar frameworks performance wise. Partial specialization in *C++* [2] allows to select efficient algorithms at compile time. Proper benchmarking shall be available soon.
- The library should achieve a certain level of numerical type independence: that is being able to use different numerical types such as the standard ones — `float`, `double`, `long double` or `std::complex` , — double-double and quad-double, see [13], arbitrary precision, see [6] or interval arithmetic; numerical classes are then parametrized by the numerical type and mathematical functions are available in a unified way. To achieve this, it uses the Boost.preprocessor library and provides through a `math::` namespace a set of mathematical functions for all supported numerical types. One should note that the library takes care of type deduction in operations using different numerical types.
- Wherever geometry is concerned, the dimension — up to 3D for now, but it is open to higher dimensions — is a template parameter of the data structures.
- Wherever possible, computing intensive operations are expressed in an algebraic way and use when possible appropriate efficient libraries, the computations should be mostly driven by linear algebra. For standard numerical types, libraries like BLAS and LAPACK can then be used for efficient calculations. Again partial specialization in *C++* [2] is at work here.
- The library delegates linear algebra in the sense that well maintained third party libraries are used through unified interfaces using the Facade design pattern, see [10]. At the moment, the library supports gmm [24], Boost.ublas, PETSc [7] and Trilinos [12].
- The library should provide the tools to generate parallel applications. The parallelisation of the library is built on top of the PETSc and Trilinos frameworks, (Par)METIS [15] and it provides iterators over the partitioned mesh.

There are many freely available libraries which offer the capabilities described previously to a certain extent. To name but a few: the Freefem software family [11,22], the Fenics project [19,18], Getdp [9] or Getfem++ [24], or libraries or frameworks such as LifeV (*C++*) [1,21], deal.II(*C++*) [8], Sundance (*C++*) [20], Analysa (Scheme) [5]. Either they rely on a domain specific language (Python, the freefem language, ...) when it comes to describe the PDE to solve, or they

[1] `http://www.boost.org`

are geometry or dimension dependent, or they are not so expressive with respect to the mathematics, i.e. the mathematics are hidden by programming details.

The main part of the article will describe the general ideas in various areas that drove the design and implementation: first we shall present the polynomial library, then the function spaces, the operators or forms and the domain specific language embedded into *C++* for variational formulations.

2 A Polynomial Library

The polynomial library is composed of various bricks: *(i)* the geometrical entities or convexes *(ii)* the prime basis in which we express subsequently the polynomials, *(iii)* the definition and construction of point sets in convexes such as quadrature point sets and finally *(iv)* polynomials and finite elements.

2.1 Convexes

The supported convexes are the simplices and quadrilaterals in $nD, n = 1, 2, 3$. Higher dimensions will eventually be implemented. Prisms and pyramids are not yet supported. The convexes are described geometrically in a standard way in terms of their subentities vertices, edges, faces, volumes. Then they are wrapped around two classes `Reference` and `Real` which define the interface for the reference convex and the convex in the real space. Again, the crucial albeit standard point is the decomposition in subentities and the capacity to iterate over the entities of a convex or of the same topological dimension inside a convex, e.g. iterate over the edges of a tetrahedron.

2.2 Prime Basis: L_2 Orthonormal Polynomials

In order to express polynomials, we need to choose a prime basis. Often one chooses the canonical one — also known as the moment or monomial basis. — Recent work by R.C. Kirby [16,17] proposed to use the Dubiner polynomials as a prime basis on the simplex. We extended these ideas on the quadrilaterals using the Legendre polynomials. Other examples of prime basis being used are the Bernstein polynomials. Our framework uses the Dubiner or Legendre basis as the default prime basis. This choice simplifies the construction of finite elements thanks to the L_2 orthogonality property of these polynomials which allows for: *(i)* easily extracting a basis spanning a subspace of a polynomial space — which corresponds to extract a range of coefficients, — *(ii)* simplifying some operations like numerical integration or the L_2 projection which is now explicit and *(iii)* better numerical stability.

A brief digression on the Dubiner and Legendre polynomials: they are constructed by taking the products of principal functions based on the Jacobi polynomials denoted $P_p^{\alpha,\beta}$ on $[-1;\ 1]$ where $\alpha, \beta > -1$ and satisfying the L_2 orthogonality relationship:

$$\int_{-1}^{1} (1-x)^\alpha (1+x)^\beta P_p^{\alpha,\beta}(x) P_q^{\alpha,\beta}(x) = C_{\alpha,\beta} \delta_{pq} \tag{1}$$

The case $\alpha = \beta = 0$ recovers the Legendre polynomials. The Dubiner polynomials are constructed by introducing a collapsed coordinate system and proper weighting functions associated with some Jacobi polynomials to recover the L_2 orthogonality, see [14] page 101 for more details. In practice, the prime basis is normalized.

2.3 Point Sets on Convexes

Now we turn to the construction of point sets in the reference convex. Point sets are represented by a matrix, they are parametrized by the associated convex and the numerical type. Recall that the convex is decomposed in vertices, edges, faces, volumes, a similar decomposition is done for the point sets: points are constructed and associated to their respective entities on which they reside. This is crucial when considering continuous Galerkin formulations for example.

The type of point sets supported are *(i)* the equidistributed point set, *(ii)* the warpblend point sets on simplices proposed by T. Warburton [25], *(iii)* Fekete points in simplices, see [14], *(iv)* standard quadrature rules in simplices and finally *(v)* Gauss, Gauss Radau and Gauss Lobatto and combinations in simplices and quadrilaterals. It should be noted that the last family is constructed from the computation of the zero of the Legendre polynomials on $[-1;\ 1]$ including eventually the boundary vertices $-1, 1$ for the Radau and Lobatto flavors. The kernel of the polynomial library is the construction of Jacobi polynomials and the computation of their zeros.

Warpblend and Fekete points are used with nodal basis which, when constructed at these points, present much better interpolation properties — lower Lebesgue constant, see [14] — Note that the Gauss-Lobatto points are Fekete points in the quadrilateral — i.e. they maximize the determinant of the associated generalized Vandermonde matrix.

2.4 Polynomial Set

After introducing the ingredients in the previous sections necessary to the construction of polynomials on simplices and quadrilaterals, we can now focus on the polynomial abstraction.

They are template classes parametrized by the prime basis in which they are expressed and the field type in which they have their values: scalar, vectorial or matricial. Their interface provides a number of operations such as evaluation and derivation at a set of points, extraction of polynomials from a polynomial set or components of a polynomial or polynomial set when the `FieldType` is `Vectorial` or `Matricial`.

One critical operation is the construction of the gradient of a polynomial or a polynomial set expressed in the prime basis. This usually requires solving a linear system where the matrices entries are given by the evaluation of the prime basis and its derivatives at a set of points. Again the choice of set of points is crucial here to avoid ill-conditioning and loss of accuracy. We choose Gauss-Lobatto points for quadrilaterals and Warpblend or Fekete points for simplicies as they provide a much better conditioning for the Vandermonde matrix. A trick is to do these computations using higher precision types, e.g. `dd_real`, and then fall back in the end to the required numerical type, e.g. `double`. This trick provides a appreciable gain in accuracy.

2.5 Finite Elements and Other Polynomial Basis

Now we turn to finite elements(FE) which we define and construct in a reference element. The reference FE are usually described by the triplet $(K, \mathbb{P}, \Sigma)$ where K is a convex, $\mathbb{P}$ the polynomial space and Σ the dual space. We have already seen the ingredients for K and $\mathbb{P}$, it remains to describe Σ. Σ is a set of functionals — degrees of freedom — taking their values in $\mathbb{P}$ with values in a scalar, vectorial or matricial field.

Several types of functionals can then be instantiated which merely require basic operations like evaluation at a set of points, derivation at a set of points, exact integration or numerical integration. Here are some examples of functionals:

- Evaluation at a point $x \in K$, $\ell_x : p \rightarrow p(x)$
- Derivation at a point $x \in K$ in the direction d, $\ell_{x,d} : p \rightarrow \dfrac{\partial p}{\partial x_d}(x)$
- Moment integration associated with a polynomial $q \in \mathbb{P}(K)$, $\ell_q : p \rightarrow \int_K pq$

A functional is represented algebraically by a vector whose entries result from the application of the functional to the prime basis in which we express the polynomials. Then applying the functional to a polynomial is just a scalar product between the coefficient of this polynomial in the prime basis by the vector representing the functional.

For example the Lagrange element is the finite element $(K, \mathbb{P}, \Sigma = \{\ell_{x_i}, x_i \in X \subset K\})$ such that $\ell_{x_i}(p_j) = \delta_{ij}$ where p_j is a Lagrange polynomial and $X = \{x_i\}$ is a set of points defined in the convex K, for example the equispaced, warpblend or Fekete point sets. Other FE such as $\mathbb{P}_{1,2}$-bubble, $\mathbb{RT}_k$, $\mathbb{N}_k$ or polynomials are constructed likewise though they are more complex.

3 Meshes, Function Spaces and Operators

In the previous section, we have described roughly the mono-domain construction of polynomials, now we turn to the ingredients for the multi-domain construction. We start with some remarks on the mesh data, then the function spaces abstraction and finally the concept of forms.

3.1 Mesh Data Structures

While performing integration and projection, it is common to be able to extract parts of the mesh. We wish to extract easily subsets of convexes out of the total set constituting the mesh.

The mesh data structure uses the Boost.Multi_index library[2] to store the elements, elements faces, edges and points. This way the mesh entities are indexed either by their ids, the process id — i.e. the id given by MPI in a parallel context, by default the current process id — to which they belong, their markers — material properties, boundary ids..., — their location — whether the entity is internal or lies on the boundary of the domain. — Other indices could certainly be defined, however those previous four already allow a wide range of applications.

Thanks to Boost.Multi_index, it is trivial to retrieve pairs of iterators over the entities — elements, faces, edges, points — containers depending on the usage context. The pairs of iterators are then turned into a range, see Boost.Range[3], to be manipulated by the integration and projection tools.

A number of free functions are available that hide all details about the mesh class to concentrate only on the relevant parts, here are two examples:

– `elements(<mesh>,<pid>)` the set of convexes constituting the `<mesh>` associated with the current process id
– `markedfaces(<mesh>,<marker>,<pid>)` the set of faces marked by `<marker>` of the `<mesh>` and belonging to the process id `<pid>`

3.2 Function Spaces and Functions

Recall that we want the framework to follow the mathematical abstraction as closely as possible. We therefore introduce naturally the `FunctionSpace` abstraction which is parametrized by the mesh type, the basis type and the numerical type, see listing 1.1. Its role is to construct the degrees of freedom table, embed the creation of elements of the space, and store interpolation data structures such as localization trees.

Functions spaces can be also defined as a product of other function spaces. This is very powerful to describe complex multiphysics problems when coupled with operators/forms described in the next section. Extracting subspaces or component spaces are part of the interface. The most important feature is that it embeds the definition of element which allows for strict definition of an `Element` of a `FunctionSpace` and thus ensures the correctness of the code.

An element has its representation as a vector — also in the case of product of multiple spaces. — The vector representation is parametrized by one of the linear algebra backends presented in the introduction — gmm, PETSc or Trilinos — that will then be subsequently used to solve the PDE. Other supported operations are interpolation and extraction of components — be it a product of function spaces element or a vectorial/matricial element, — see listing 1.1 for an example of `FunctionSpace::Element`.

[2] `http://www.boost.org/libs/multi_index/doc/index.html`
[3] `http://www.boost.org/libs/range/index.html`

3.3 Operators and Forms

One last concept needed to have the variational formulation language mathematically expressive is the notion of forms. They follow closely their mathematical counterparts: they are template classes with arguments being the space or product of spaces they take as input and the algebraic representation of these forms. In what follows, we consider only the case where the linear and bilinear forms are represented by vectors and matrices respectively — we could consider also vector-free and matrix-free representations.

The crucial point is that the linear and bilinear form classes are the glue between their algebraic representation and the variational formulation which is stored in a complex tree-like data structure that we denote `Expr`, it will

- Fill the matrix with non-zero entries depending on the approximation space(s) and the mathematical expression;
- Allow a per-component/per-space construction(blockwise);
- Check that the numerical types of the expression and the representation are consistent
- When `operator=( Expr const& )` is called, the expression is evaluated and will fill the representation entries

With the high level concepts described we can now focus on the variational formulation language.

4 A Variational Formulation Language Embedded in *C++*

In order to express the PDE problems, libraries or applications rely either on a domain specific language usually written in an interpreted language such as Python or a home made one or on high level interfaces. These interfaces or languages are desirable for several reasons: teaching purposes, solving complex problems with multiple physics and scales or rapid prototyping of new methods, schemes or algorithms. The goal is always to hide (ideally all) technical details behind software layers and provide only the relevant components required by the user or programmer.

In the library, a domain specific language *embedded* — later written DSEL — in *C++* has been implemented. This language for numerical integration and projection has been proposed by the author in [23]. The DSEL approach has advantages over generating a specific *external* language like in case *(i)* : compiler construction complexities can be ignored, other libraries can concurrently be used which is often not the case of specific languages which would have to also develop their own libraries and DSELs inherit the capabilities of the language in which they are written. However, DSELs are often defined for one particular task inside a specific domain [26] and implementation or parts of implementation are not shared between different DSELs. Here, we have a language that shares the expression tree construction between integration, projection and automatic differentiation.

The implementation of the language relies on standard techniques such as expression templates [26,2,3,4,21] and an involved two stages evaluation of the expression tree. Without dwelling too much upon the internals of the language, it is interesting to note that it supports some optimisation with respect to the product of N function spaces. Indeed let's consider the statements in listing 1.1: depending on the block to be filled by the local/global assembly process, some terms must be equated to 0. The language detects that at compile time and thanks to the g++ compiler it will not consider the terms which are not applying to the currently assembled block, see [23] for more details.

5 Example: A Navier-Stokes Solver for the Driven Cavity

We wish to solve the 3D incompressible Navier-Stokes equations [14] in a domain $\Omega = [0,1]^3$ in parallel with a $\mathbb{P}_2/\mathbb{P}_1$ discretization for the velocity and pressure

Listing 1.1. A 3D incompressible Navier-Stokes example

```
// mesh of linear tetrahedron
typedef Mesh<Simplex<3,1> > mesh_t;
// define and partition the mesh if in parallel
mesh_t mesh; mesh.partition();
typedef fusion::vector<Lagrange<3,2,Vectorial>,
                       Lagrange<3,1,Scalar> > basis_t;
typedef FunctionSpace<mesh_t,basis_t> space_type;
space_type V_h( mesh );
space_type::Element<trilinos> U( V_h ), V( V_h );
// views for U and V: velocity
space_type::element<0,trilinos>::type u = U.element<0>();
space_type::element<0,trilinos>::type v = V.element<0>();
// views for U and V: pressure
space_type::element<1,trilinos>::type p = U.element<1>();
space_type::element<1,trilinos>::type q = V.element<1>();
// integration method : Gauss points on the Simplex in 3D
// that integrates exactly polynomials of degree 5
IM<3,5,double,Simplex,Gauss> im;
for( double t = dt; t < T; t+= dt ) {
  // ops: id, dx,dy,dz, grad...
  // when no suffix, it identifies test basis functions
  // the t suffix identifies trial basis functions
  // the v suffix denotes the interpolated value
  // integrate(<set of elements>,<integration>,<expression>);
  backend<trilinos>::vector_type F;
  form( V_h, F) f =
    integrate( elements(mesh), im,
     (idv(u) - dt*gradv(p))*id(v) );
  backend<trilinos>::matrix_type A;
  form( V_h, V_h, A) a =
    integrate( elements(mesh), im,
     nu*dt*dot(gradt(u), grad(v)) + idt(u)*id(v) +
     dt*(dot( idv(u), gradt(u) ))*id(v)
     - idt(p)*div(v) + id(q)*divt(u) )+
    on( 10, u, F, oneX() )+ // 10 identifies Γ1
    on( 20, u, F, 0 ); // 20 identifies Γ2
  A.close(); // final assembly
}
```

respectively, that is to say, $(u,p) \in X_h \times M_h$ such that for all $(v,q) \in X_h \times M_h$, $\int_\Omega \frac{\partial u}{\partial t} v + \nu \nabla u \cdot \nabla v + u \cdot \nabla u v - \nabla \cdot v \; p = 0$ with $\int_\Omega \nabla \cdot u \; q = 0$, $u_{\Gamma_2} = (1,0,0)^T$ on Γ_2 which is the top face of unit domain Ω and $u_{\Gamma_1} = (0,0,0)^T$ on $\Gamma_1 = \partial\Omega \backslash \Gamma_2$ and X_h and M_h being adequate spaces the velocity and pressure respectively. As a parallel linear algebra backend, Trilinos is used.

6 Conclusion and Main Results

Life is a mathematical library with an emphasis on *(i)* approximation methods while the linear algebra is delegated to third party libraries and *(ii)* the ability to solve partial differential equations using a language embedded in *C++* very close to the mathematics. This allows the user to concentrate on the problem at hand and manipulate high level abstractions and not being bothered by the programming details. Life will also become the new mathematical kernel of the LifeV project (`www.lifev.org`) whose objective is to develop solvers for multiscale multiphysics applications in particular in haemodynamics.

Acknowledgments

The author wishes to acknowledge the chair of Prof. A. Quarteroni at EPFL which funded this research, and E. Burman, C. Winkelmann, G. Steiner and G. Pena from EPFL for the many fruitful discussions we had.

References

1. Lifev: a finite element library, `http://www.lifev.org`
2. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. C++ in Depth Series. Addison-Wesley Professional (2004)
3. Aubert, P., Di Césaré, N.: Expression templates and forward mode automatic differentiation. In: Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U. (eds.) Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science, ch. 37, pp. 311–315. Springer, New York (2001)
4. Aubert, P., Di Césaré, N., Pironneau, O.: Automatic differentiation in C++ using expression templates and application to a flow control problem. Computing and Visualisation in Sciences 3(4), 197–208 (2001)
5. Bagheri, B., Scott, R.: Analysa (2003), `http://people.cs.uchicago.edu/~ridg/al/aa.ps`
6. Bailey, D.H., Hida, Y., Jeyabalan, K., Li, X.S., Thompson, B.: C++/fortran-90 arbitrary precision package, `http://crd.lbl.gov/~dhbailey/mpdist/`
7. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual, Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2004)
8. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II Differential Equations Analysis Library, Technical Reference, `http://www.dealii.org`

9. Dular, P., Geuzaine, C.: Getdp: a general environment for the treatment of discrete problems, `http://www.geuz.org/getdp`
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley Professional Computing Series. Addison Wesley, London (1995)
11. Hecht, F., Pironneau, O.: FreeFEM++ Manual. Laboratoire Jacques Louis Lions (2005)
12. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the trilinos project. ACM Trans. Math. Softw. 31(3), 397–423 (2005)
13. Hida, Y., Li, X.S., Bailey, D.H.: Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 9472 (October 2000), `http://crd.lbl.gov/~dhbailey/mpdist/`
14. Karniadakis, G.E., Sherwin, S.J.: Spectral/*hp* element methods for CFD. In: Numerical Mathematics and Scientific Computation, Oxford University Press, New York (2005)
15. Karypis, G., Kumar, V.: Parallel Multilevel k-way Partitioning Schemes for Irregular Graphs. Technical Report 036, Minneapolis, MN 55454 (May 1996)
16. Kirby, R.C.: Algorithm 839: Fiat - a new paradigm for computing finite element basis functions. ACM Trans. Math. Software 30(4), 502–516 (2004)
17. Kirby, R.C.: Optimizing fiat with level 3 blas. ACM Trans. Math. Software (2005)
18. Kirby, R.C., Logg, A.: A compiler for variational forms. Technical report, Chalmers Finite Element Center (May 2005), `www.phi.chalmers.se/preprints`
19. Logg, A., Hoffman, J., Kirby, R.C., Jansson, J.: Fenics (2005), `http://www.fenics.org/`
20. Long, K.: Sundance: Rapid development of high-performance parallel finite-element solutions of partial differential equations, `http://software.sandia.gov/sundance/`
21. Di Pietro, D.A., Veneziani, A.: Expression templates implementation of continuous and discontinuous galerkin methods. Computing and Visualization in Science 2005 (Submitted)
22. Pino, S.D., Pironneau, O.: FreeFEM3D Manual. Laboratoire Jacques Louis Lions (2005)
23. Prud'homme, C.: A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. Scientific Programming 14(2), 81–110 (2006)
24. Renard, Y., Pommier, J.: Getfem++: Generic and efficient c++ library for finite element methods elementary computations, `http://www-gmm.insa-toulouse.fr/getfem/`
25. Warburton, T.: An explicit construction for interpolation nodes on the simplex. Journal of Engineering Mathematics 2005 (submitted)
26. Veldhuizen, T.: Using C++ template metaprograms 7(4), 36–43 (1995) (S. Lippman (ed.) Reprinted in C++ Gems)

Parallel Program Complex for 3D Unsteady Flow Simulation

Eugene V. Shilnikov

Institute for Mathematical Modeling RAS, 125047, Miusskaya sq. 4A, Moscow, Russia
shiva@imamod.ru

Abstract. A parallel program complex for 3D viscous gas flow simulation is presented. This complex is based on explicit finite difference schemes, which are constructed as an approximation of conservation laws (control volume method), and oriented towards use of locally refined grids. A special algorithm and utility for nested grid partitioning was created. The principles of program construction permits one to introduce new types of boundary conditions and change the finite difference scheme as well as the governing equation system. Introducing new face types and writing new subroutines for flux calculations may reach it. The scalability of the program complex was investigated on different subsonic and supersonic problems. The calculations were made on different types of cluster computer systems. The parallelization efficiency was more than 90% for 40 processors and more than 60% for 600 processors.

Keywords: Unsteady viscous gas flows, Parallel program complex, Kinetically consistent finite difference schemes, Turbulent flows, Parallelization efficiency.

1 Introduction

Essentially unsteady and turbulent regimes of viscous gas flows have received increasing attention from researchers, motivated in part by the importance of unsteadiness in industrial problems arising in turbomachinery and aeronautics. Unsteady flow phenomena which occur frequently behind relatively slender bluff structures are of great practical interest. First of all this is connected with the possible destructive influence of the acoustic pressure oscillations occurring in the gas flow upon mechanical properties of the vehicle construction elements, especially in the resonant case. Under certain conditions such flows may be characterized by quasi-regular self-induced pressure oscillations. Their frequency, amplitude and harmonic properties depend upon the body shape and external flow conditions. In the case of symmetric geometry at relatively small Reynolds numbers, numerical simulation based on 2D unsteady Navier - Stokes equations is quite successful. At high Reynolds numbers, which are more relevant in practice, 3D stochastic turbulent fluctuations are superimposed on the quasi-periodic 2D unsteady motion. So numerical simulation must be three-dimensional even for simple flow geometry. The numerical simulation of a detailed structure of unsteady viscous compressible 3D flows with high Reynolds numbers is possible only by the use of high performance parallel computer systems. This demands the development of specialized parallel software. This software must have a good scalability (with respect to the number of processors and to the problem size), portability and robustness.

2 Choice of Numerical Method

The use of parallel computer systems with distributed memory architecture determines the choice of numerical method. The opinion is widely spread that we have to use only implicit schemes for viscous gas flow simulation, because of their good stability properties. That is quite right for stationary or slow flows. But in the case of essentially unsteady flows, we have to receive detailed information about the high frequency oscillations of the gas dynamic parameters. This fact limits the time step acceptable by the accuracy requirements. For many interesting problems these limitations are very strong, although they may not ensure the stability of an explicit scheme. On the other hand, each time step of implicit schemes usually demands much more calculations. Besides, these schemes are difficult for parallel implementation from the viewpoint of efficiency. All these factors together may neutralize the advantages of implicit schemes. So for such problems, the explicit difference schemes seem to be preferable because of their simplicity. Our program complex is based on explicit finite difference schemes, which are constructed as an approximation of conservation laws (control volume method). The explicit kinetically consistent finite difference (KCFD) schemes [1] were selected for realization.

KCFD schemes belong to the class of kinetic schemes. Nowadays the kinetic or Boltzmann schemes are very popular in computational fluid dynamics [2] – [4]. The kinetic schemes differ from other algorithms primarily in the fact that the basis for their derivation is the discrete models for the one-particle distribution function. Schematically, the process of deriving the kinetic schemes can be represented as the following chain: the Boltzmann equation – the discrete model for one-particle distribution function – the averaging of the discrete models with the collision vector components. As a result we obtain the discrete equations for gas dynamic parameters. Traditionally, another logical chain is used: the Boltzmann equation – the averaging of the Boltzmann equation with the collision vector components and the gas dynamics equations as a result – the discretization of gas dynamics equations. Compared with the traditional algorithms, the processes of averaging and discretization are interchanged in the case of kinetic schemes. The basic assumptions used for KCFD schemes construction are that one particle distribution function (and the macroscopic gas dynamic parameters too) have small variations on the distances compatible with the average free path length l, and the distribution function has Maxwellian form just after molecular collisions. Proceeding from these assumptions, we may suppose that one particle distribution function has a Maxwellian form constant on cells of a size equal to the free path length. The dissipative terms of KCFD schemes are formed by two different "half-Maxwellian" distribution functions taken from both sides of a cell face.

The variant of KCFD schemes named Corrected KCFD scheme, may be considered as an approximation of Navier - Stokes equations with some additional terms. For 3D case in Cartesian coordinates it can be written in the following form:

$$\frac{\hat{\rho}-\rho}{\Delta t}+(\rho u_j)_{\overset{\circ}{x}_j}=(\tau_j(\rho u_j^2+p)_{\bar{x}_j})_{x_j},$$

$$\frac{\widehat{\rho u_i}-\rho u_i}{\Delta t}+(\rho u_i u_j+p\delta_{ij})_{\overset{\circ}{x}_j}=\frac{\mathrm{M}}{\mathrm{Re}}\left[(\mu u_i)_{\bar{x}_j}(1+\frac{\delta_{ij}}{3})\right]_{x_j}+(\tau_j(\rho u_i u_j^2+3pu_i\delta_{ij})_{\bar{x}_j})_{x_j},$$

$$\frac{\hat{E}-E}{\Delta t}+(u_j(E+p))_{\overset{\circ}{x}_j}=\frac{\mathrm{M}}{2\mathrm{Re}}\left[\mu(\boldsymbol{u}^2+\frac{u_j^2}{3})_{\bar{x}_j}\right]_{x_j}+\frac{\gamma\mathrm{M}}{\mathrm{RePr}(\gamma-1)}\left[\mu\left(\frac{p}{\rho}\right)_{\bar{x}_j}\right]_{x_j}+$$
$$+\left[\tau_j\left(u_j^2(E+2.5p)\right)_{\bar{x}_j}\right]_{x_j}+\left[\tau_j\frac{p}{\rho(\gamma-1)}p_{\bar{x}_j}\right]_{x_j}.$$

Here $i,j=1,2,3$, summation by index j is supposed, ρ – density, p – pressure, u_i – three velocity components, $E=\dfrac{p}{\rho(\gamma-1)}+\dfrac{\boldsymbol{u}^2}{2}$ – total energy, $\boldsymbol{u}^2=u_1^2+u_2^2+u_3^2$, γ – specific ratio, μ – viscosity, M, RE, PR – Mach, Reynolds and Prandtl numbers, $A_{\overset{\circ}{x}}$, $A_{\bar{x}}$, A_x, – central, backward and forward finite difference first order spatial derivatives, Δt – time step, $\hat{A}$ – variable at the next time moment. All gas dynamic variables are defined in the cell centers.

Additional dissipative terms in these equations contain factor $\tau_j=\Delta x_j/(2\mathrm{U})$, where Δx_j – spatial step along j coordinate and $\mathrm{U}=\sqrt{\gamma p/\rho+\boldsymbol{u}^2}$. They can be interpreted as efficient numerical stabilizers which provide smoothness of the solution on the distance l. It is interesting to remark that analogous stabilizers were obtained independently in [3]. Multiplying these equations by cell volume $\Delta x_1\Delta x_2\Delta x_3$, we may rewrite them in the form of an approximation of conservation laws. In this form, the variation of gas dynamic parameters in a cell (control volume) is defined as a sum of fluxes through all cell faces. This scheme has a Courant-like stability condition ($\Delta t\sim\Delta x_{min}$) giving the opportunity to use very fine meshes to study the fine flow structure. The successful experience in solving various gas dynamic problems by means of KCFD schemes showed that they describe viscous heat conducting flows as good as schemes for Navier - Stokes equations, where the latter are applicable. In addition, this KCFD scheme permits us to calculate oscillating regimes in super- and transonic gas flows, which are very difficult to model by means of other algorithms.

Modeling some problems of mathematical physics demand high accuracy resolution of the solution particularities in small regions. Such situations may occur, for example, in detonation or combustion processes, solitons motion, flow around a body with shape singularities etc. This problem may be avoided by the use of unstructured meshes, but the convenience and simplicity of difference schemes on regular grids enforced us to use multiblock grids where different subregions have their own grids. A variant of such an approach, namely the use of nested (or locally refined) grids is chosen. The main idea is to divide one or more cells of regular (maybe rough) grid into some amount of small cells. These cells are rectangular too. If these new cells are not sufficiently small, we may repeat the refining procedure and divide some of them into smaller cells and so on. We use explicit difference schemes, which are constructed as an approximation of conservation laws. So, the main problem is approximation of fluxes between cells of different sizes. The construction of this approximation and the parallel realization of explicit finite difference schemes for the numerical solution of gas dynamic problems on nested grids were discussed in [5].

3 Parallel Implementation

The basic idea for the program complex constructed, was ultimately to simplify the parallel program providing its lucidity. For this reason all complicated but not time-consuming operations are addressed to sequential preprocessing utilities as far as possible. The program complex structure is defined by four relatively independent tasks:

– problem/model formulation, including a description of the computation field geometry, the initial and boundary conditions specifications and the grid generation;
– grid partitioning and data preprocessing;
– main computational block execution;
– data postprocessing.

As a rule, only the third of these tasks must be parallel. Separate sequential programs may execute all other jobs.

Parallel realization is based on a geometry parallelism principle. Each processor makes calculations in its own separate subdomain. It has to exchange data with processors working in adjacent subregions. Only information from boundary cells of each subregion is to be sent, and the exchange is local. In the case when the difference scheme is written in the form of conservation laws, the approximation of gas dynamic equations comes to the approximation of conservative variables (density ρ, momentum $\rho\mathbf{U}$ and total energy E) fluxes through cell faces. In order to reach the algorithm homogeneity, different types of boundary conditions (no slip, symmetry, impermeability, inlet, outlet conditions etc.) are also written as fluxes of conservative variables, though region bound. Each cell face is supplied with an attribute indicating its type: an inner face, various boundary faces, a face between cells of different size (result of local mesh refining), a ghost face i.e. a face between cells laying outside the computational region. This attribute determines which subroutine that must calculate the fluxes through the face. The face attributes, as well as a description of the problem geometry and grid information, are contained in a special text file, which is prepared by sequential preprocessing utility.

Another utility divides the 3D computational region with rectangular bounds (in i-j-k space) into the required number of rectangular subdomains according to the multistep algorithm described in [5]. It is based on the utility contained in the previous version of our program package [6]. This utility divides the 3D box-shaped region into the required number of rectangular subdomains with approximately equal volumes. The following modification of this partitioning algorithm is proposed.

The computational region is expanded up to parallelepiped. Each cell of the original regular mesh is ascribed a weight 1. Additional cells get zero weights, and their faces is set to the "ghost" attribute. Zero weights are also ascribed to ghost cells which are inside the computational region (cells in the solid body, for example). If a cell was divided into N small cells during mesh refining, it is ascribed a weight N. Now we may calculate the number of cells which must be in each subregion. That is equal to the sum of all weights divided by the number of processors. After that the computational region is divided into the needed number of subdomains with approximately equal total weights. If the mesh refinement is too deep, some of the original cells weights may be too large for one subdomain. Meeting such a cell, we treat it separately using the same algorithm again. After allotting the needed number of small cells to a subdomain, we subtract this number from the weight of the treated large cell. This procedure is repeated until the residual weight becomes less than the number needed for one processor. Note that some subregions may contain each other. In this case, the cell faces of the inner subregion get the attribute of the ghost face in the outer domain. If subdomains are overlapping, their common faces get the ghost attribute in one of the regions. The permissibility of overlapping subregions is useful for achieving equal number of cells when all domains have a box shape. As a result, this utility creates a text file that describes the 3D subregions in terms of grid node numbers, it also lists the neighbors to each subregion and the information needed for the organizing of inter processor communications.

According to this, the main computational module of the parallel program is uniform for all processors. Each processor calculates fluxes through all faces in its subregion. Having equal number of faces in each subdomain, the homogeneity of the algorithm automatically provides load balancing of processors. For a regular grid, an equal number of cells provides an equal number of faces, except for faces between cells of different size. The number of such faces, which bound the zones of local mesh refining, is not very large. Some imbalance is also possible because different face types demand different amounts of calculations. However, a large number of cells in subdomains leads to a levelling of these differences. At last, small differences among weights of subregions are possible. It is unavoidable because the result of division of the total weight by the number of processors is not always an integer. These differences do not affect efficiency dramatically, if the maximum weight is close to average one.

Note that such program construction permits us to introduce new types of boundary conditions and change as finite difference scheme as governing equation system (changing the coordinate system for example).

4 Program Package Testing

The program complex was tested on a problem of supersonic($M_\infty = 2$) and subsonic ($M_\infty = 0.135$) viscous compressible gas flow around a square cylinder at $Re = 20000$. The inlet of the computational region was located at $x/D = -5$ and the outlet at $x/D = 15$, where $D = 1$ is cylinder diameter. The origin of the coordinate system was in the center of the cylinder's front surface. The length of the cylinder was equal to $4D$. The height and width of the computational region were equal to $20D$. A locally refining computational grid was used. The original coarse grid with a maximum cell size equal to 0.2 was refined near the cylinder walls so that the minimum cell size was 0.001. The total amount of grid cells was about 8000000. The no slip conditions were posed on the cylinder walls. The far field boundary conditions, based on splitting of the convective fluxes by Steger – Warming [7], were used.

The supersonic flow is stationary in our case. The calculated pressure distribution in the symmetry plane $z = 0$ for this flow is presented in Fig. 1. The flow picture is typical and not very interesting.

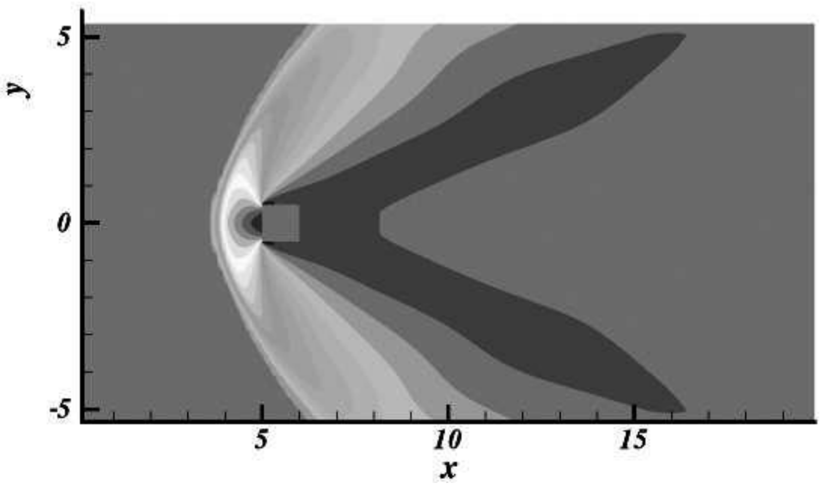


Fig. 1. Pressure distribution for a supersonic flow

The subsonic flow is much more complicated and interesting. At the beginning of the calculations the symmetric flow is formed, but after a while this symmetry is broken, due to vortex shedding from four cylinder's back edges, and the flow becomes quasi-periodic. A very complicated flow structure is formed behind the rear face of the cylinder. The stream traces near the cylinder in the same symmetry plane, but for different flow phases, are shown in Fig. 2. The character of the stream traces in the $y = 0$ plane is analogous with this. The distance between time instances approximately corresponds to half a period of the main oscillation mode. The pressure distribution is plotted in the background.

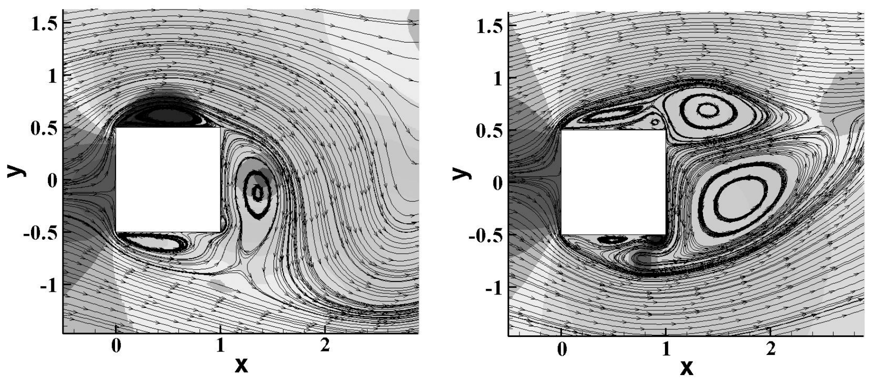


Fig. 2. Stream traces in the $z = 0$ section

The pictures in Fig. 2 are quite natural for the flow type under consideration and are similar to those presented in [8] & [9], where detailed experimental data for such flow are described.

In [10], the results of numerical simulations of the case studied in [9] are presented. The comparison of the calculation results obtained with different turbulence models in the 2D problem formulation was also made in this paper. Our results are in good agreement with these results. The comparison of calculated and experimental integral characteristics, such as the mean value of the drag coefficient $\overline{c_D}$, the amplitude of middle-range oscillations of lift coefficient c'_L and the Strouhal number Sh corresponding to these oscillations are summarized in Table 1. It is necessary to note, that our results were obtained without introducing any turbulence model.

Table 1. Force coefficients and Strouhal numbers

	$\overline{c_D}$	c'_L	Sh
Experimental results [9]	2.05 – 2.19	—	0.135 –0.139
Calculated results [10]	1.56 – 2.11	0.30 – 1.18	0.129 – 0.146
Our results	2.09	0.37	0.137

The amplitude spectrum of the lift coefficient oscillations is presented in Fig. 3. The largest peak in the plot (at $Sh \approx 0.1$) corresponds to the main periodic component of the flow oscillations.

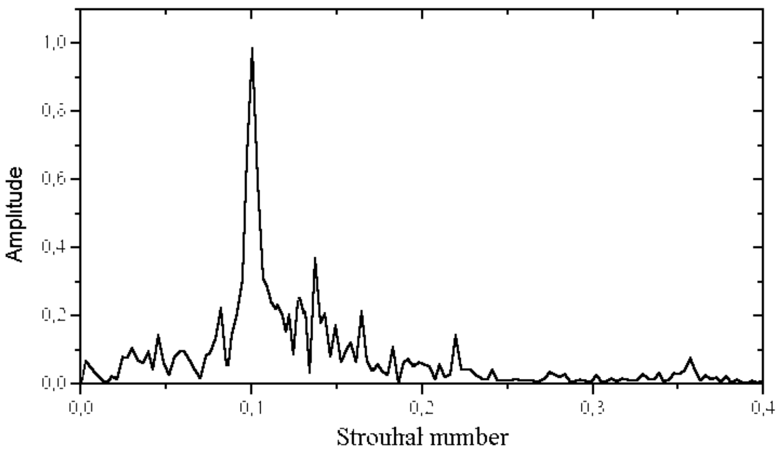


Fig. 3. Spectrum of the lift coefficient

5 Efficiency Investigation

The scalability of the program complex was investigated on different types of cluster multiprocessor computer systems (a 768-processor MCS-1000M computer system equipped with 667MHz 64-bit 21164 EV67 Alpha processors and a 906-processor MCS-15000 computer system equipped with 2.2GHz PPC970FX processors). The parallelization efficiency was measured for different number of processors. The results can be found in Table 2.

Table 2. Efficiency (%) dependence on the number of processors

N	1	2	10	40	160	320	600
MCS-1000M	—	100	97	92.8	79.9	70.1	61.4
MCS-15000	100	98.5	96.3	92	80.1	72.1	62.7

Because of lack of memory, such a large problem can't be solved on one processor of the MCS-1000M system. That's why the efficiency for this system was computed with respect to the calculations made on 2 processors.

The scaling with respect to the problem size was also inspected. Our grid was doubled in each direction, so the total number of cells became 8 times larger than previous and amounted to $\sim 6 \times 10^7$. Such a problem is too large even for one processor of the MCS-15000 system, so we can't measure efficiency directly. The following method is used: We specify some test job, for example 2000 time steps of our program. Let $t_s^{(N)}$ and $t_l^{(N)}$ be the calculation times needed for this job on "small" and "large" grids respectively (N is the number of processors). The total computational work for the large problem is 8 times greater than for the small one. If the efficiency doesn't depend on the problem size, we must receive $f = t_l^{(N)}/t_s^{(N)} = 8$ for each N. The diminishing of this factor corresponds to the efficiency increase for the large problem. The values of this factor for some numbers of processors are presented in Table 3.

Table 3. Computational time increase for enlarged grid

N	10	40	100	200	400	600
f	7.96	7.83	7.71	7.48	7.13	6.69

These results show that for $N < 100$ our factor is really close to 8, but for greater N it diminishes. This effect is connected with the increase of computational time with respect to exchange time for each processor. So, the increasing of the total number of grid nodes leads to efficiency growth. For example, factor 6.69 for $N = 600$ means a 75% efficiency for the "large" problem in contrast to a 62.7% efficiency for the "small" problem with the same number of processors.

Acknowledgments

This work was supported by Russian Foundation for Basic Research (grants No. 06-01-00187, 05-07-90230, 05-01-00510).

References

1. Chetverushkin, B.N.: On improvement of gas flow description via kinetically-consistent difference schemes. In: Chetverushkin, B.N., et al. (eds.) Experimental modeling and computation in flow, turbulence and combustion, vol. 2, pp. 27–37. Wiley, Chichester (1997)
2. Perthame, B.: The kinetic approach to the system of conservation laws. Recent advances in partial differential equations. Res. Appl. Math. Masson, Paris 30 (1992)
3. Oñate, E., Manzam, M.: Stabilization techniques for finite element analysis for convective-diffusion problem, vol. 183. Publication CIMNE (2000)
4. Succi, S.: The lattice Boltzmann equations for fluid dynamics and beyond. Clarendon press, Oxford (2001)
5. Shilnikov, E.V.: Viscous gas flow simulation on nested grids using multiprocessor computer systems. In: Proceedings of Parallel Computational Fluid Dynamics Conference, Moscow, Russia, 2003, pp. 110–115. Elsevier Science, BV (2004)

6. Shilnikov, E.V., Shoomkov, M.A.: Parallel Program Package for 3D Unsteady Flows Simulation. In: Joubert, G.R., et al. (eds.) Parallel Computing. Advances and current Issues. Proceedings of international conference ParCo2001, pp. 238–245. Imperial College Press, London (2002)
7. Hirsh, C.: Numerical computation of internal and external flows. In: Fundamentals of numerical discretization, vol. 1, John Wiley & Sons, A Wiley Interscience Publication, Chichester (1988)
8. Lin, D.A., Rody, W.: The flapping shear layer formed by flow separation from the forward corner of a square cylinder. J. Fluid Mech. 267, 353–376 (1994)
9. Lin, D.A., Einav, S., Rody, W., Park, J.-H.: A laser-Doppler velocimetry study of ensemble-averaged characteristics of the turbulent near wake of a square cylinder. J. Fluid Mech. 304, 285–319 (1995)
10. Bosch, G., Rodi, W.: Simulation of vortex shedding past a square cylinder with different turbulence models. Int. J. Numer. Meth. Fluids 28, 601–616 (1998)

Multi-scale Physics: Minisymposium Abstract

Mats G. Larson

Umeå University, Sweden

Many problems of interest in science and engineering involve several types of physics possibly interacting over several scales in time and space. This minisymposia presents recent progress on numerical methods for solving problems involving multiple scales and/or physics. In particular, topics include adaptive procedures for tuning critical discretization parameters, technology for treating multiphysics problems on different meshes, modeling of unresolved fine scale behavior using multiscale and homogenization approaches. Applications range from simulation of electrical activity in the human heart to fluid and solid mechanics.

Simulation of Multiphysics Problems Using Adaptive Finite Elements

Fredrik Bengzon, August Johansson, Mats G. Larson, and Robert Söderlund

Department of Mathematics, Umeå University, SE-901 87 Umeå, Sweden
mats.larson@math.umu.se
http://www.math.umu.se/

Abstract. Real world applications often involve several types of physics. In practice, one often solves such multiphysics problems by using already existing single physics solvers. To satisfy an overall accuracy, it is critical to understand how accurate the individual single physics solution must be. In this paper, we present a framework for a posteriori error estimation of multiphysics problems and derive an algorithm for estimating the total error. We illustrate the technique by solving a coupled flow and transport problem with application in porous media flow.

1 Introduction

Many industrial problems involve several different physical entities. A common example from engineering is to determine the expansion of a material subject to temperature change. A problem of this kind is referred to as a multiphysics problem, because it involves different kinds of physics. In the example, we have heat conductivity and elasticity theory. Solving multiphysics problems is often done by connecting existing single physics solvers into a network and applying some splitting scheme. Although each of the solvers may be optimized with respect to accuracy and computational resources to perform its task, connecting them does not necessarily imply global accuracy of the desired quantity, nor efficient use of resources. The cause of this is the difficulty of choosing suitable discretizations for each of the involved solvers. To determine an adequate discretization, it is necessary to understand the interaction of solver errors when connected in a network.

In this paper, we develop an algorithm to automatically determine individual and locally adaptive meshes for each single physics solver involved in a general multiphysics simulation. The analysis is based on a posteriori error estimates using duality techniques, allowing for error bounds on a user specific, possibly nonlinear, output functional. The finite element method is used for discretizing each problem. We illustrate the ideas by solving a coupled time dependent pressure transport problem. We note that such decoupled algorithms are suitable for implementation in distributed environments where each problem is solved on separate processors or computers.

Duality based a posteriori error estimates are available for many single physics problems, see for example [1], [2], and [3]. However, for multiphysics problems, a posteriori error estimates are scarce. The reader may cf. [6] by the authors for a general approach with application to stationary MEMS simulations. See also [7] and [5] for other applications.

This paper is organized as follows. We introduce the concept of a network of finite element solvers in Section 2. Error representation formulas are derived and the algorithm for computing such in a network is presented in Section 3. To illustrate the use of the algorithm, we close the paper by solving a coupled flow and transport problem in Section 4.

2 Finite Element Network Solvers

Single Physics Solver. Let $\mathcal{P}$ be the set of physics in a multiphysics problem. We assume that each physics $p \in \mathcal{P}$ can be described by a well posed abstract variational problem of the form: Find $u_p \in \mathcal{V}_p$ such that

$$a_p(u_p, v) = l_p(v) \text{ for all } v \in \mathcal{V}_p, \tag{1}$$

where $\mathcal{V}_p$ is a suitable function space, and $a_p(\cdot,\cdot)$ and $l_p(\cdot)$ are possibly time dependent linear forms. The input to these linear forms are usually parameters describing the properties of the physics, but very often it is also output from the other physics. Certainly we cannot expect any of these input data to be exact, since the physical parameters may be measured only up to a certain level of accuracy, and since the solution to any other physics is computed and therefore prone to numerical errors. To take this into account we introduce approximate forms, $\tilde{a}_p(\cdot,\cdot)$ and $\tilde{l}_p(\cdot)$, which are computed using the inexact inputs. As a consequence, the abstract finite element discretization of the physics reads: Find $u_p^h \in \mathcal{V}_p^h$ such that

$$\tilde{a}_p(u_p^h, v) = \tilde{l}_p(v) \text{ for all } v \in \mathcal{V}_p^h, \tag{2}$$

where $\mathcal{V}_p^h$ is a suitable finite dimensional subspace of $\mathcal{V}_p$.

Multiphysics Solver. We think of a multiphysics solver as a network of connected single physics solvers. Assuming we have solvers for each of the physics $p \in \mathcal{P}$ of a multiphysics problem, we determine dependencies between the solvers by defining a directed graph: Let $\mathcal{P}$ represent the vertices of a graph and draw directed edges from physics q to physics p if p depends on q. We will by $\mathcal{D}_p$ denote the set of all in-neighbours to vertex p (i.e., all vertices having a directed edge to p). To simplify things a bit, we assume that the graph does not contain any cycles. However, if two physics do form a cycle (corresponding to a mutual data dependency) we can reduce the graph by considering them as a single problem, see Figure 1.

3 A Posteriori Error Estimates

A Posteriori Error Estimate for a Single Physics Solver. We now derive an error representation formula for the error in a nonlinear functional in terms of

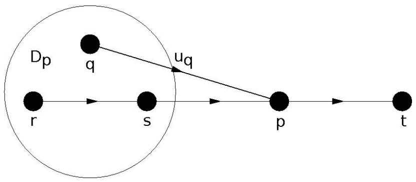


Fig. 1. A network of single physics solvers comprising a multiphysics solver. Each node represents a single physics finite element solver. Solver p depends on input data, that is, the solutions u_q, from the three solvers in $\mathcal{D}_p = \{q, r, s\}$. The arrows indicate the direction of information flow.

the discretization error and the modeling errors caused by erroneous input data. Let $m(\cdot)$ be a given functional on $\mathcal{V}_p$. We note the elementary identity

$$m(v) - m(w) = \bar{m}(v - w) \equiv \int_0^1 m'(vs + w(1-s))(v - w)\, ds. \tag{3}$$

Note also that the functional $\bar{m}(\cdot)$ depends on both v and w, that is, $\bar{m}(\cdot) = \bar{m}(v, w; \cdot)$.

To control the error in $m(\cdot)$ we introduce the dual problem: Find $\phi_p \in \mathcal{V}_p$ such that

$$\bar{m}(v) = \tilde{a}_p(v, \phi_p) \text{ for all } v \in \mathcal{V}_p. \tag{4}$$

Using (3) and plugging $v = u_p - u_p^h$ into (4) we obtain

$$\begin{aligned}
m(u) - m(u_p^h) &= \bar{m}(u_p - u_p^h) && (5)\\
&= \tilde{a}_p(u_p - u_p^h, \phi_p) && (6)\\
&= \tilde{a}_p(u_p, \phi_p) - \tilde{a}_p(u_p^h, \phi_p) && (7)\\
&= \tilde{a}_p(u_p, \phi_p) - a_p(u_p, \phi_p) && (8)\\
&\qquad + a_p(u_p, \phi_p) - \tilde{a}_p(u_p^h, \phi_p) &&\\
&= \tilde{a}_p(u_p, \phi_p) - a_p(u_p, \phi_p) && (9)\\
&\qquad + l_p(\phi_p) - \tilde{l}_p(\phi_p) &&\\
&\qquad + \tilde{l}_p(\phi_p - \pi_p\phi_p) - \tilde{a}_p(u_p^h, \phi_p - \pi_p\phi_p), &&
\end{aligned}$$

where we used the definition of the dual problem and the Galerkin orthogonality property (2) to subtract the interpolant $\pi_p\phi_p \in \mathcal{V}_p^h$.

Introducing the residual $\tilde{R}(u_p^h) \in \mathcal{V}_p^*$, where $\mathcal{V}_p^*$ is the dual of $\mathcal{V}_p$, as follows

$$(\tilde{R}_p(u_p^h), v) = \tilde{l}_p(v) - \tilde{a}_p(u_p^h, v) \text{ for all } v \in \mathcal{V}_p, \tag{10}$$

we finally get the following error representation formula

$$\begin{aligned}
m(u) - m(u_p^h) &= (\tilde{R}(u_p^h), \phi_p - \pi_p\phi_p) && (11)\\
&\qquad + \tilde{a}_p(u_p, \phi_p) - a_p(u_p, \phi_p) + l_p(\phi_p) - \tilde{l}_p(\phi_p). &&
\end{aligned}$$

From (11) it is obvious that the first term, $(\tilde{R}(u_p^h), \phi_p - \pi_p \phi_p)$, can be interpreted as a discretization error, while the other terms, $\tilde{a}_p(u_p, \phi_p) - a_p(u_p, \phi_p) + l_p(\phi_p) - \tilde{l}_p(\phi_p)$, are modelling errors due to errors in input data.

In practice, a computable estimate is used to bound the discretization error. For example, using the dual weighted residual method, see [1], we end up with the estimate

$$(\tilde{R}(u_p^h), \phi_p - \pi_p \phi_p) \leq \sum_{K \in \mathcal{K}_p} \rho_K^p \omega_K^p, \tag{12}$$

where ρ_K^p is a computable element residual, ω_K^p is a weight accounting for the effect of the dual problem, and K is a finite element within the mesh $\mathcal{K}_p$. We return to specific forms of the residual and weight below.

A Posteriori Error Estimate for a Multiphysics Solver. Suppose that we wish to compute a certain functional $m_p(u_p)$ of the solution u_p to problem $p \in \mathcal{P}$, which is part of a multiphysics problem. Then not only do we need to investigate which problems we have to solve to be able to compute u_p, but we also need to know how the error propagates between these problems. We shall now formulate a simple algorithm for this task.

Recalling that the set of problems feeding data into problem p is given by $\mathcal{D}_p$, we conclude that the data needed to compute the forms $a_p(\cdot,\cdot)$ and $l_p(\cdot)$ are determined by the solutions u_q for $q \in \mathcal{D}_p$. Therefore, let $v_{\mathcal{D}_p}$ be a data entry of $\mathcal{V}_{\mathcal{D}_p} = \bigoplus_{q \in \mathcal{D}_p} \mathcal{V}_q$ and define the nonlinear functional $M_p : \mathcal{V}_{\mathcal{D}_p} \rightarrow \mathbf{R}$, by

$$M_p(v_{\mathcal{D}_p}) = l_p(v_{\mathcal{D}_p}; \phi_p) - a_p(v_{\mathcal{D}_p}; u_p, \phi_p), \tag{13}$$

where u_p and ϕ_p are the solutions to the primal (1) and dual (4) problems, respectively, and we have also emphasized the dependence on the data $v_{\mathcal{D}_p}$. We then note that the modeling error in (11) can be written

$$M_p(u_{\mathcal{D}_p}) - M_p(u_{\mathcal{D}_p}^h) = \tilde{a}_p(u_p, \phi_p) - a_p(u_p, \phi_p) + l_p(\phi_p) - \tilde{l}_p(\phi_p), \tag{14}$$

where $u_{\mathcal{D}_p} = \bigoplus_{q \in \mathcal{D}_p} u_q$ is a vector containing the solutions to the problems feeding input data into problem p and $u_{\mathcal{D}_p}^h$ is the vector of corresponding finite element approximations. Further, using a version of the identity (3) with vector valued arguments we have

$$\begin{aligned} M_p(u_{\mathcal{D}_p}) - M_p(u_{\mathcal{D}_p}^h) &= \sum_{q \in \mathcal{D}_p} \int_0^1 \frac{\partial M_p}{\partial u_q}(s u_{\mathcal{D}_p} + (1-s) u_{\mathcal{D}_p}^h)(u_q - u_q^h)\, ds && (15) \\ &= \sum_{q \in \mathcal{D}_p} \bar{m}_q(u_q - u_q^h), && (16) \end{aligned}$$

where we have defined the linear functionals $\bar{m}_q : \mathcal{V}_q \rightarrow \mathbf{R}$ by

$$\bar{m}_q(u_q - u_{q,h}) = \int_0^1 \frac{\partial M_p}{\partial u_q}(s u_{\mathcal{D}_p} + (1-s) u_{\mathcal{D}_p}^h)(u_q - u_q^h)\, ds. \tag{17}$$

From this it follows that the modeling error can be expressed as a sum of functionals of the errors $u_q - u_q^h$, $q \in \mathcal{D}_p$. Thus to control the overall error in the functional m_p, we need to control the errors in functionals $\bar{m}_q$ for $q \in \mathcal{D}_p$ as well as the discretization error in problem p. Now each problem q can be handled in the same way using duality based a posteriori error analysis for functional $\bar{m}_q$. Iterating the procedure until no problems are left, we obtain a formula for the error in functional m_p in terms of weighted residuals of all problems in $\mathcal{D}_p$. This reasoning leads us to Algorithm 1.

Algorithm 1

```
1: Given a functional $m_p(\cdot)$ for problem $p$, set $\mathcal{S} = \{p\}$ and $I = (\tilde{R}(u_p^h), \phi_p - \pi_p \phi_p)$.
2: Set $\mathcal{S} = \cup_{q \in \mathcal{S}} \mathcal{D}_q$.
3: For each problem in $\mathcal{S}$, compute a functional $m_q(\cdot)$ from (17) to be controlled.
4: For each problem $q \in \mathcal{S}$ compute $(\tilde{R}(u_q^h), \phi_q - \pi_q \phi_q)$ corresponding to functional
   $m_q(\cdot)$ and set $I = I + (\tilde{R}(u_q^h), \phi_q - \pi_q \phi_q)$.
5: If $\mathcal{S}$ is empty $m_p(u - u_p^h) = I$, otherwise goto 2.
```

4 A Multiphysics Example

Contaminant Transport in a Darcy Flow. Let us consider a concrete multiphysics problem taken from porous media flow.

The transport of a contaminant through a porous medium Ω saturated with fluid is governed by

$$\dot{u}_C + \nabla \cdot (\beta u_C - \epsilon \nabla u_C) + \gamma u_C = 0, \quad t > 0, \tag{18}$$

where u_C is the concentration of the contaminant, ϵ is a diffusion tensor, γ is the absorption rate of the porous medium, and β is the advection field (i.e., the velocity of the pore fluid).

We assume (18) to be subject to the natural boundary condition

$$n \cdot (\beta u_C - \epsilon \nabla u_C) = \kappa (u_C - g_{C,\infty}) + g_{C,0}, \tag{19}$$

where n is the outward unit normal of the boundary $\partial\Omega$, κ is the permeability of the boundary, $g_{C,\infty}$ is the concentration of the ambient medium, and $g_{C,0}$ is the influx.

In this paper we restrict attention to steady state flow fields characterized by *Darcy's law*,

$$\beta = -a \nabla u_P, \tag{20}$$

where a is the permeability of the porous medium and u_P is the fluid pressure. If the fluid is incompressible and if the porous solid is non-deformable, then $\nabla \cdot \beta = 0$. It follows that

$$-\nabla \cdot (a \nabla u_P) = 0. \tag{21}$$

In addition to (21) we impose the boundary conditions

$$u_P|_{\Gamma_P^D} = g_P, \qquad n \cdot \nabla u_P|_{\Gamma_P^N} = 0, \tag{22}$$

where Γ_D and Γ_N are two parts of the boundary such that $\partial\Omega = \Gamma_D \cup \Gamma_N$ and $\Gamma_D \cap \Gamma_N = \emptyset$, and g_P is a given pressure.

Equations (18) and (21) constitute a weakly coupled multiphysics problem, since the pressure flux provides the convection field for the advective transport. The weak coupling stems from the fact that this is a one way exchange of data, that is, (21) provides data for (18) but not vice versa.

Weak Form of the Multiphysics Problem. The weak formulation of (21) reads: Find $u_P \in \mathcal{V}_{P,g_P^D} = \{v \in H^1(\Omega) \, : \, v|_{\Gamma_P^D} = g_P^D\}$ such that

$$a_P(u_P, v) = 0 \text{ for all } v \in \mathcal{V}_{P,0}, \tag{23}$$

where $\mathcal{V}_{P,0} = \{v \in H^1(\Omega) \, : \, v|_{\Gamma_P^D} = 0\}$, and the linear form $a_P(\cdot,\cdot)$ is defined by

$$a_P(v, w) = (a\nabla v, \nabla w). \tag{24}$$

The weak formulation of (18) reads: Find $u_C \in H^1(\Omega) \times L_2(0,T)$ such that

$$a_C(u_C, v) = l_C(v) \text{ for all } v \in H^1(\Omega) \times L_2(0,T), \tag{25}$$

where space-time linear forms $a_C(\cdot,\cdot)$ and $l_C(\cdot)$ are defined by

$$a_C(v,w) = \int_0^T ((\dot{u}_C, v) - (\beta u_C, \nabla v) \tag{26}$$
$$+ (\epsilon\nabla v, \nabla w) + (\gamma u, v) + (\kappa v, w)_{\partial\Omega})\, dt,$$
$$l_C(v) = \int_0^T (\kappa g_{C,\infty} - g_{C,0}, v)_{\partial\Omega}\, dt. \tag{27}$$

Meshes, Finite Element Spaces, and the Transfer Operator. In order to formulate the numerical methods we need to define some function spaces. Let $\mathcal{K} = \{K\}$ be a partition of Ω into hexahedral elements K with longest side h_K, and let $\mathcal{V}^h = \mathcal{V}^h(\mathcal{K})$ be the space of continuous piecewise trilinear functions defined on $\mathcal{K}$. Further, let $0 = t_0 < t_1 < t_2 < \ldots < t_N = T$ be a uniform partition of the time interval $I = (0,T)$ into N subintervals $I_n = (t_n - t_{n-1})$ of equal length $\delta t = t_n - t_{n-1}$, $n = 1, 2, \ldots, N$, and let P_d^q and P_c^q be the spaces of piecewise discontinuous and piecewise continuous polynomials of degree q on this partition, respectively.

We use hanging nodes to accomplished local mesh refinement. In doing so, the global conformity of the trilinear spaces is preserved by eliminating the hanging nodes via interpolation between neighboring regular nodes, cf. e.g. [1].

Each single physics problem is discretized using individually adapted meshes, and data is transferred between these using the transfer operator $\pi_{pq} : \mathcal{V}_{p,h} \to \mathcal{V}_{q,h}$, defined by $\pi_{pq}v = \pi_q v$, where $\pi_q : C(\Omega) \to \mathcal{V}_{q,h}$ is the usual trilinear nodal interpolation operator for continuous functions. In particular π_{pq} is the identity operator when $\mathcal{V}_p^h \subset \mathcal{V}_q^h$.

The Finite Element Methods. The finite element method for (21) reads: Find $u_P^h \in \mathcal{V}_{P,g_P^D}^h$ such that

$$a_P(u_P^h, v) = 0 \text{ for all } v \in \mathcal{V}_{P,0}^h. \tag{28}$$

The finite element method for (18) reads: Find $u_C^h \in \mathcal{V}_T^h \times P_c^1$ such that

$$\tilde{a}_C(u_C^h, v) = l_C(v) \text{ for all } v \in \mathcal{V}_T^h \times P_d^0, \tag{29}$$

where $\tilde{a}_C(\cdot,\cdot)$ is the approximate linear form obtained by approximating the advection field β, by the computable approximation $\beta^h = -a\pi_{PC}\nabla u_P^h$. Note that π_{PC} is the transfer operator from the pressure mesh to the transport mesh. In the case of small diffusion ϵ, we stabilize the above finite element method with a streamline diffusion term (see e.g. [4]).

A Posteriori Error Estimates. To control the error $u_C - u_C^h$ at a specific point X, let ψ be a positive function with localized support around X and define the linear functional $m_C(v) = \int_0^T (v,\psi)\, dt$. Introducing the dual transport problem

$$m_C(v) = \tilde{a}_C(v, \phi_C), \tag{30}$$

we have, following (11), the error representation

$$m_C(u_C - u_C^h) = l_C(\phi_C) - \tilde{a}_C(u_C^h, \phi_C) + m_P(u_P - u_P^h), \tag{31}$$

where, see [7],

$$m_P(u_P - u_P^h) = (-a\nabla(u_P - u_P^h), \int_0^T (u_C \nabla\phi_C)\, dt). \tag{32}$$

The functional $m_P(\cdot)$ is the modeling error from the pressure flux (i.e., the approximative advection field β^h). Thus, to control this error, we introduce the dual pressure problem

$$m_P(v) = a_P(v, \phi_P), \tag{33}$$

which, following (11) again, gives the error representation

$$m_P(u_P - u_P^h) = l_P(\phi_P) - a_P(u_P^h, \phi_P). \tag{34}$$

Note that we do not get any modeling error here since the input data for the pressure solver is assumed to be exact. Of course, we do however get a discretization error.

As mentioned earlier, one can estimate the discretization error via the dual weighted residual method. We use this method to compute appropriate element indicators, which are used to select the most error prone elements for adaptive mesh refinement.

For the pressure problem we have the element indicator $\rho_K^P\omega_K^P$, where

$$\rho_K^P(u_P^h) = \|\nabla\cdot a\nabla u_P^h\|_K + \tfrac{1}{2}h_K^{-1/2}\|[n\cdot a\nabla u_P^h]\|_{\partial K}, \tag{35}$$

and

$$\omega_K^P(\phi_P) = \|\phi_P - \pi\phi_P\|_K + h_K^{1/2}\|\phi_P - \pi\phi_P\|_{\partial K}. \tag{36}$$

For the transport problem we consider the time averaged element indicator $\int_0^T \rho_K^C \omega_K^C \, dt$, where

$$\rho_K^C(u_T^h) = \|\dot{u}_C^h + \nabla \cdot (\beta^h u_C^h - \epsilon \nabla u_C^h) + \gamma u_C^h\|_K + \tfrac{1}{2} h_K^{-1/2} \|\epsilon [n \cdot \nabla u_C]\|_{\partial K}, \tag{37}$$

and

$$\omega_K^C(\phi_C) = (\delta t \|\dot{\phi}_C\|_K + h_K \|\phi_C\|_K). \tag{38}$$

Thus for this problem the overall solution strategy takes the following form:

1. Compute the pressure u_P^h.
2. Compute the concentration u_C^h.
3. Compute the dual concentration ϕ_C^h.
4. Compute the dual pressure ϕ_P^h.
5. Evaluate the element indicators $\rho_K^C(u_C^h)\omega_K^C(\phi_C^h)$ and $\rho_K^P(u_P^h)\omega_K^P(\phi_P^h)$.
6. Refine the meshes for the pressure and transport solvers.
7. Repeat until satisfactory results are obtained.

Numerical Example. To evaluate the method of Algorithm 1, we present the following numerical example of coupled Darcy flow and contaminant transport.

The computational geometry is a channel with a bend and three obstacles, see Figure 2. To create a pressure drop and consequently a pressure flux, we apply a prescribed pressure of $g_P^D = \pm 1$ on the inflow and outflow boundary of

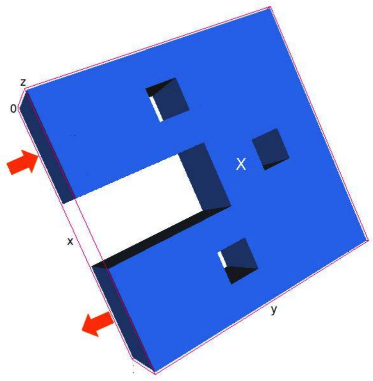


Fig. 2. The computational geometry, a channel with a bend and three obstacles. The bounding box is of dimension $[0,8] \times [0,8] \times [0,2]$. Arrows indicate regions of inflow and outflow. The capital letter X denotes the randomly chosen point $(4,5,1)$ where we want to control the contamination concentration u_C.

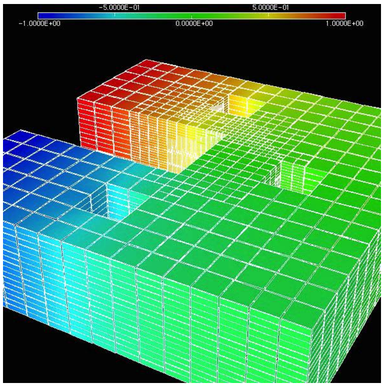


Fig. 3. The computed pressure u_P^h and adapted mesh after five refinements, using the duality based error indicator $\rho_K^P \omega_K^P$

the channel, respectively. For simplicity we assume that the permeability of the channel is unity (i.e., $a = 1$).

Taking the gradient of the computed pressure we get the advection field for the contamination transport. However since we use different meshes for the pressure and transport solvers, we must first interpolate the pressure to the transport mesh and then compute the advective field. The transport solver implements a Crank-Nicolson time stepping scheme combined with a streamline diffusion stabilized finite element method. The stabilization enables us to choose $\epsilon = 0.01$ and $\gamma = 0$. At the inflow we prescribe a net influx of $g_{C,0} = 1$, and at the outflow we use a transparent, outflow boundary condition with $\kappa = \beta \cdot n$, $g_{C,\infty} = g_{C,0} = 0$. All other boundaries are assumed to be totally absorbing, which corresponds to letting $\kappa \to \infty$. In practice this is implemented by setting κ to a big number. The evolution of the contaminant is started from a zero initial condition for u_C and simulated until final time $T = 100$, when a steady state occurs. Figure 4 show a series of snapshots of the computed contamination concentration u_C^h.

The aim of the computation is to control the time average error around the point $X = (4, 5, 1)$, which lies in the vicinity of the center obstacle. Therefore,

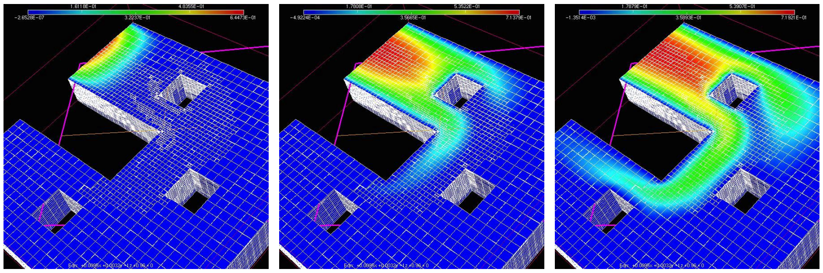


Fig. 4. Snapshots of the contamination concentration u_C^h at times $t = 5$, 25, and 50

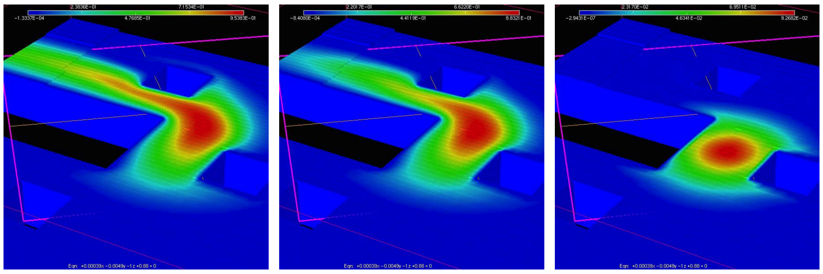


Fig. 5. Snapshots of the transport dual ϕ_C^h at times $t = 50$, 25, and 0

we let $\psi = \exp(-c|X - x|^2)$, where $c = 100$ is a constant. Solving the dual transport problem, which runs backwards in time, we get the transport dual ϕ_C^h of Figure 5.

Given the dual transport solution, we solve the dual pressure problem for ϕ_P^h, see Figure 6.

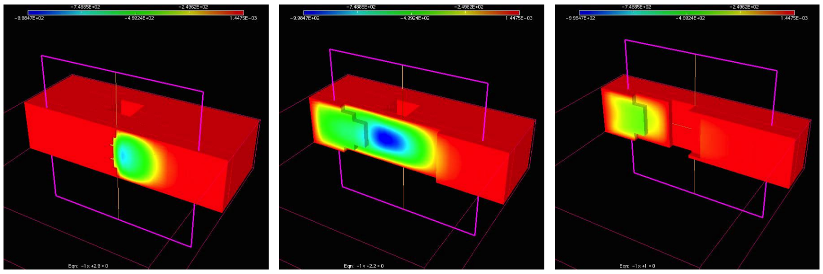


Fig. 6. Slice plot of the (time independent) dual pressure ϕ_P^h. Note that ϕ_P^h has a rather localized support indicating that the region where the pressure must be accurately computed is quite small.

Given the primal and dual solutions we compute the element indicators, refine the meshes, and start the simulation again from the initial time $t = 0$.

As is evident from Figures 3 and 4, our duality based adaptive algorithm is able to identify and refine the regions relevant for accurate computation of the contamination concentration around the point X. The dual pressure, for instance, clearly indicates the inflow region and the region around the first obstacle as important to get a correct pressure flux for the subsequent advective transport while the other areas are of less importance and thus a coarse grid can be used there.

5 Conclusions

In this paper we have developed a framework for adaptive simulation of multiphysics problems based on available adaptive single physics finite element solvers.

We assume that the single physics solvers are adaptive and can control the accuracy in given functionals of the solution using duality based a posteriori error estimates together with mesh refinement. Given an output quantity of interest in the multiphysics problem the framework identifies functionals for the involved single physics solvers that need to be controlled to achieve overall accuracy in the quantity of interest. Note, in particular, that different quantities of interest result in different accuracy requirements on the individual single physics solvers.

We illustrate our technique on a coupled time dependent pressure transport problem. The example illustrates a simple coupling and the use of individual mesh refinement in the two involved solvers with the overall goal of computing a localized weighted average of the solution.

So far only a few works have been published on adaptive methods for multiphysics applications and therefore many problems still remain. For instance, further extensions to more challenging applications involving different scales in space and time, where adaptivity is crucial, should be investigated.

References

1. Bangerth, W., Rannacher, R.: Adaptive finite element methods for differential equations. Birkhäuser Verlag (2003)
2. Estep, D.J., Larson, M.G., Williams, R.D.: Estimating the error of numerical solutions of systems of reaction-diffusion equations. Mem. Amer. Math. Soc. 146 (2000)
3. Eriksson, K., Estep, D., Hansbo, P., Johnson, C.: Introduction to adaptive methods for differential equations. Acta Numer. 105–158 (1995)
4. Eriksson, K., Estep, D., Hansbo, P., Johnson, C.: Computational Differential Equations. Studentlitteratur (1996)
5. Grätsch, T., Bathe, K.-J.: Goal-oriented error estimation in the analysis of fluid flows with structural interactions. Comput. Method. Appl. Mech. Engin. (Article in press)
6. Larson, M.G., Bengzon, F.: Adaptive Finite Element Approximation of Multiphysics Problems. Comm. Num. Method. Engin. (to appear)
7. Larson, M.G., Målqvist, A.: Goal oriented adaptivity for coupled flow and transport problems with applications in oil reservoir simulation. The Finite Element Center Preprint Series, 2006-3. Comput. Method. Appl. Mech. Engin. (to appear), `http://www.femcenter.org`

A New Domain Decomposition Approach Suited for Grid Computing

Juan A. Acebrón[1], Raúl Durán[2], Rafael Rico[2], and Renato Spigler[3]

[1] Departament d'Enginyeria Informàtica i Matemàtiques, Universitat Rovira i Virgili, Av. Països Catalans 26, ES-43007 Tarragona, Spain
juan.acebron@urv.cat

[2] Departamento de Automática, Escuela Politécnica, Universidad de Alcalá de Henares, Crta. Madrid-Barcelona, Km 31.600, ES-28871 Alcalá de Henares, Madrid, Spain
raul.duran@uah.es,rafael.rico@uah.es

[3] Dipartimento di Matematica, Università "Roma Tre", 1, Largo S.L. Murialdo, IT-00146 Rome, Italy
spigler@mat.uniroma3.it

Abstract. In this paper, we describe a new kind of domain decomposition strategy for solving linear elliptic boundary-value problems. It outperforms the traditional ones in complex and heterogeneous networks like those for grid computing. Such a strategy consists of a hybrid numerical scheme based on a probabilistic method along with a domain decomposition, and full decoupling can be accomplished. While the deterministic approach is strongly affected by intercommunication among the hosts, the probabilistic method is scalable as the number of subdomains, i.e., the number of processors involved, increases. This fact is clearly illustrated by an example, even operating in a grid environment.

1 Introduction

It is well-known that a number of applications can benefit from operating in a grid environment [4,5,18]. A grid system could be exploited for solving very large problems, which are otherwise intractable even with the existing supercomputers, but latency will be a crucial issue. All experiments show this. The most successful case of grids is that of handling huge amounts of data, as in the celebrated example of the SETI project (SETI@home), and other similar to it. In these applications latency does not play any negative role. Due to the high degree of system heterogeneity and high latency, however, only few applications seem to be computationally advantageous. Indeed, parallel scientific computing to handle problems based on the solution of partial differential equations (PDEs) is currently at a crossroad. Among the several numerical methods proposed in the literature for solving boundary-value problems for PDEs, domain decomposition methods seem to be particularly well-suited for parallel architectures.

The main idea consists in decoupling the original problem into several subproblems. More precisely, the given domain is divided into a number of subdomains, and the task of the numerical solution on such separate subdomains is then assigned to different processors or computational sites. However, the computation cannot run independently for each subdomain, because the latter are coupled to each other through internal interfaces where the solution is still unknown. Therefore, the processors have to exchange data along these interfaces at each computational time step, resulting in a degradation of the overall performance. In fact, due to the global character of the PDE boundary-value problem, the solution cannot be obtained even at a single point inside the domain prior to solving the entire problem. Consequently, certain iterations are required across the chosen (or prescribed) interfaces in order to determine approximate values of the solution sought inside the original domain. There exist two approaches for a domain decomposition, depending on whether the domains overlap or do not overlap; see [16,17], e.g. Given the domain, the subproblems are coupled, hence some additional numerical work is needed, and therefore it is doubtful whether full scalability might be attained as the number of the subdomains increases unboundedly.

Moreover, implementing such a method on a parallel architecture involves partitioning the given mesh into the corresponding subdomains, balancing the computational load as well as minimizing the communication overhead at the same time. For this purpose, traditionally, graph partitioning strategies have been used [10,11]. Such strategies rest on considering the problem's domain as a graph made of weighted vertices and communication edges. Then, the balancing problem can be viewed as a problem of partitioning a graph by balancing the computational load among subdomains while minimizing the communication edge-cut. However, most of the traditional partitioners are no longer suitable for the emerging grid strategies [12]. In fact, these algorithms only consider balancing computational load and reducing communication overhead, while in practice the data movement cost may be a crucial issue in view of the high communication latency on a grid.

In order to overcome this drawback, several strategies have been already proposed. In [14,15], an unbalanced domain decomposition method was considered. The idea there consists of hiding communication latency by performing useful computations, and assigning a smaller workload to processors responsible for sending messages outside the host. This approach goes in the right direction, but other ways are currently in progress. For instance, two graph partitioners, suited for grid computing, i.e., capable to take into account the peculiarities of grid systems, with respect to both, CPUs and network heterogeneity, are MiniMax [8] and PaGrid [9]. These algorithms try to minimize the overall CPU time instead of balancing computational load, minimizing intercommunication at the same time. In a homogeneous environment, these two actions are equivalent, but in the heterogeneous case, things are very different. A parallel version of METIS, ParMETIS [10,11], is only capable to handle CPUs heterogeneity.

In this paper, we propose a method recently developed by some of the authors and others [1,2]. It has been proven to be rather successful in homogeneous parallel architectures. In Section 2, some generalities about the method are discussed. In Section 3, a numerical example is shown, where the performance in a grid environment was tested. In the final section, we summarize the high points of the paper.

2 The Probabilistic Method

The core of the probabilistic method is based on combining a probabilistic representation of solutions to elliptic or parabolic PDEs with a classical domain decomposition method (DD). This approach can be referred to as a "*probabilistic domain decomposition*" (for short, PDD) method. This approach allows to obtain the solution at some points, internal to the domain, without first solving the entire boundary-value problem. In fact, this can be done by means of the probabilistic representation of the solution. The basic idea is to compute only few values of the solution by Monte Carlo simulations on certain chosen interfaces, and then interpolate to obtain continuous approximations of it on such interfaces. The latter can then be used as boundary values to decouple the problem into subproblems, see Fig. 1. Each such subproblem can then be solved *independently* on a separate processor. Clearly, neither communication among the processors nor iteration across the interfaces are needed. Moreover, the PDD method does not even require balancing. In fact, after decomposing the domain into a number of subdomains, each problem to be solved on them will be totally independent of the others. Hence, each problem can be solved by a single host. Even though some hosts may end the computation much later than others, the results obtained from the faster hosts are correct, and can be immediately used, when necessary.

Fault tolerance of algorithms is an essential issue in grid environments, since a typical characteristic of a grid is to allow dynamically aggregation of resources what results in a strong modification of the structure of the system itself. Hence, all classical methods, which require continuous communications among the various processors involved, will be seriously affected and the overall performance in general degraded. Even a single processor exiting the system at some point, will in general abort the running process, since all the remaining processors must wait for the results expected from such processor. The probabilistic method, instead, is unaffected by this kind of events, since all subproblems are fully decoupled and the unaffected processors may continue processing their task.

3 Numerical Examples

Here we present some numerical examples, aiming at comparing the performance achieved by a classical deterministic domain decomposition method and the PDD method, in a grid environment. To this purpose, the Globus Toolkit's services and the Globus-based MPI library (MPICH-G2) [7,19] have been used. We built

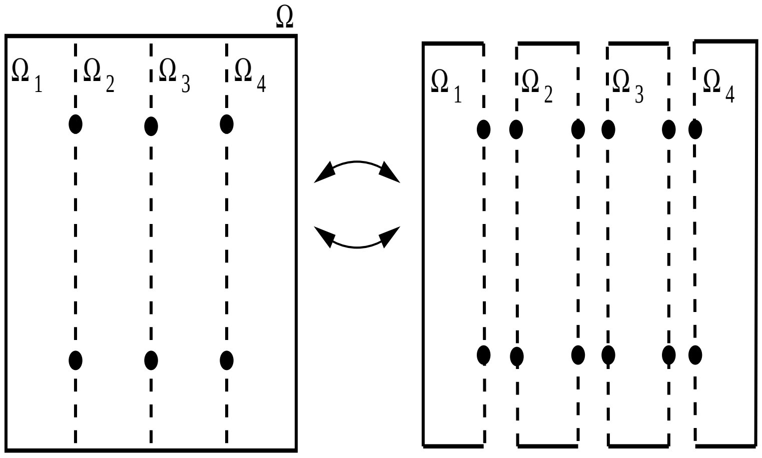


Fig. 1. Sketchy diagram illustrating the numerical method, splitting the initial domain Ω into four subdomains, $\Omega_1, \Omega_2, \Omega_3, \Omega_4$

the highly heterogeneous computing system sketched in Fig.2, made with 16 PCs AMD Duron processors (below referred to as of type Cluster A), 1.3 GHz, linked to each other through a Fast Ethernet connection, plus 2 other PCs, one being an AMD Athlon XP 2400+ (type B), the other an AMD-K6, 500 MHz (type C). These two PCs are also linked through a Fast network, but they are linked to the previous group of 16 PCs through a 10 Ethernet connection (which is much slower than the other, as is well known). We chose the example of the Dirichlet problem

$$u_{xx} + (6x^2 + 1)u_{yy} = 0 \qquad \text{in } \Omega = (0,1) \times (0,1) \tag{1}$$

with the boundary data

$$u(x,y)|_{\partial\Omega} = \left[(x^4 + x^2 - y^2)/2\right]_{\partial\Omega} \tag{2}$$

the solution being $u(x,y) = (x^4 + x^2 - y^2)/2$.

This problem has been solved discretizing it by finite differences with size $\Delta x = \Delta y = 1/N$, $N = 1200$.

In Fig. 3(a) and 3(b), the pointwise numerical error is shown, made correspondingly to the DD and the PDD methods. For the former, only two nodes on which interpolation was made have been used on each interface. Parameters were chosen conveniently in order to attain a comparable error for both methods.

The deterministic algorithm we adopted is extracted from the numerical package pARMS [13], where the overlapping Schwarz method with a FGMRES iterative method has been chosen [6,13]. This was preconditioned with ILUT as local solver. To split the given linear algebraic system corresponding to the full discretized problem into a number of subproblems, and solve them independently, in parallel, it is necessary to accomplish a mesh partitioning. This should

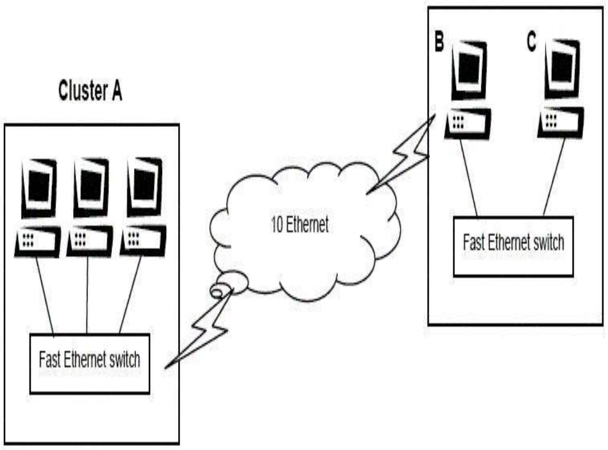


Fig. 2. Diagram showing the Grid environment in which the algorithms have been tested

be done balancing the overall computational load (according to the processors' heterogeneity), minimizing, at the same time, the intercommunication occurring among the various processors. As already pointed out, operating in a grid environment, this task is by far more challenging than in a homogeneous hosts setting.

At this point, we used ParMETIS as a partitioner, configuring it according to the characteristics of the particular grid we adopted. For instance, it is possible to assign the computational load according to the CPU performance of each available processor. With this, the CPU heterogeneity is taken into account. As for the intercommunication problem, we observe that minimizing communications overhead may not correspond to minimizing graph's edge-cuts, since one should weight somehow the importance of every specific edge connecting pairs of nodes of the mesh. This importance depends on the specific problem to be solved, i.e., on the physics embodied in the problem, besides the geometry of the domain and the discretization underlying the chosen algorithm (such as finite differences, finite elements, etc.). Clearly, different partitions of the given domain lead to different linear algebraic problems on each subdomain. In particular, using iterative methods, the local problems will be characterized by matrices with different condition numbers.

On each subdomain, the time required for the computations is proportional to the size of the problem, i.e., to the number of nodes or the square of the matrix dimension, times the condition number of such a matrix. The latter determines the number of iterations required to achieve a prescribed accuracy in the iterative local solution. Recall that the time spent for computations on each subdomain (i.e., due to a given processor) is that spent per iteration times the number of iterations, and the former is proportional to the number of nodes. Of course, using direct (instead of iterative) methods, the condition numbers of the local

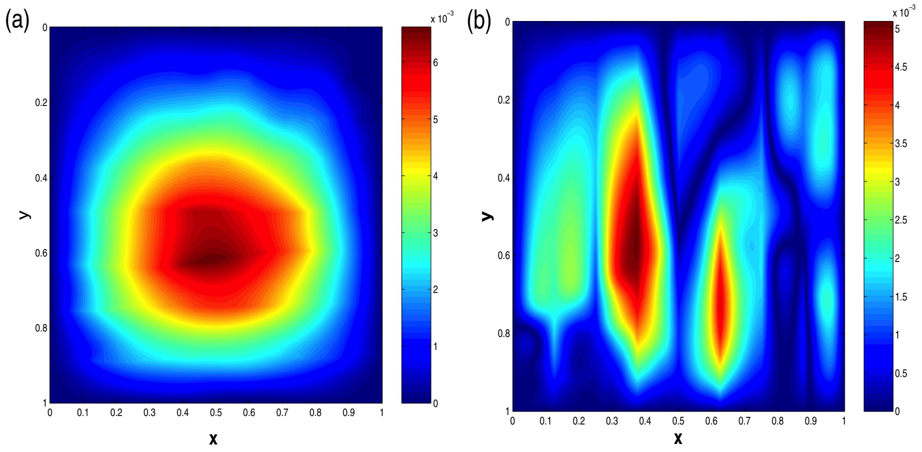


Fig. 3. Pointwise numerical error in: (a) the DD algorithm, and (b) the PDD algorithm

matrices, A_i in $A_i x_i = b_i$, $i = 1, \ldots, p$ (p being the number of subdomains) are only important in the amplification of errors in the entries of A_i and b_i. Direct methods, however, are used only when the local problems are small, typically with A_i of dimension not larger that few hundreds.

The number of nodes in each subdomain could be chosen according to the specific performance of each CPU (the more powerful the CPU is, more nodes are assigned to it). The condition number of A_i is a much more delicate issue, since it also depends on the physics, i.e., in our problems, on the coefficients of the PDE.

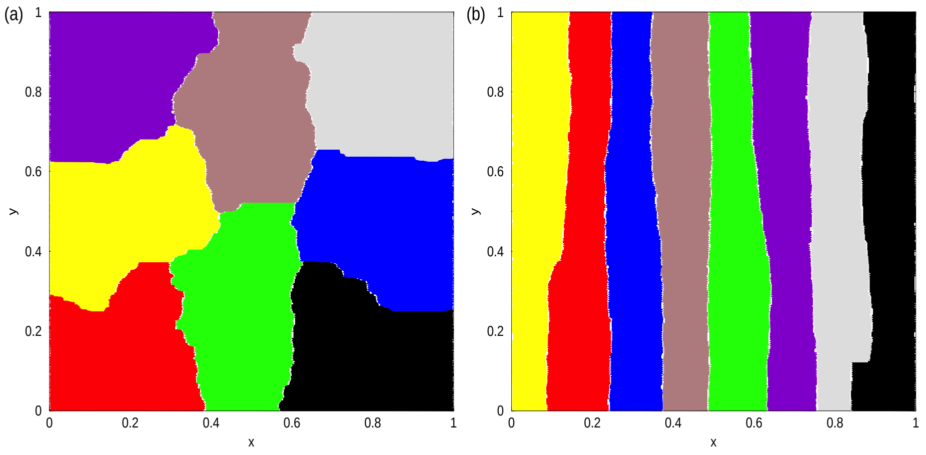


Fig. 4. Partitions of the given domain into 8 subdomains, and assigned to 8 processors for two different configurations: (a) Only geometry was taken into account, (b) Both geometry and physics was considered

Table 1. CPU time in seconds for the heterogeneous system

$Processors$	$PDD(unbalanced)$	$PDD(balanced)$	$DD(A)$	$DD(B)$
8	88.67	43.18	372.72	282.73
16	46.36	25.67	300.86	221.30
18	32.52	18.44	270.21	302.75

In the specific example above, in view of the asymmetry in x and y, the particular partition of the domain is very relevant, requiring very different iteration numbers. In Fig. 4(a), numerical experiments pertaining to the example below, show a partition which takes into account only the geometry of the problem. In Fig. 4(b), instead, a different partition has been obtained which takes into account the physics as well, having introduced such dependence in the weights of the edges. These weights increase with x, in the y direction. Therefore, in the configuration underlying Fig. 4(b), the total number of iterations required is smaller than in case of Fig. 4(a) (we needed 220 versus 294 iterations). From this example it appears clearly that minimizing the edge-cuts as in the configuration in Fig. 4(a) does not minimize the overall computational time, since the iteration number in such a case is larger than that needed in Fig. 4(b), see Table 1.

Table 2. CPU time in seconds for the homogeneous subsystem (Cluster type A)

$Processors$	PDD	$DD(A)$	$DD(B)$
8	38.78	169.71	132.35
16	20.82	242.01	197.49

In the PDD approach, we discretized the local problems by finite differences, and solved the corresponding linear algebraic systems by the same iterative method. Here, the partitioning has been done as shown in Fig. 1, which corresponds, somehow, to the *optimal* one found above for the DD.

As in the deterministic DD, CPU heterogeneity can be taken into account, hence all subdomains may have different area (and thus are characterized by different computational load). Moreover, in this case, due to the full decoupling among the various subdomains, the iteration times as well as the iteration numbers pertaining to every subdomain are easily computable. This is clearly shown in Fig. 5, where 16 processors have been used: all but the processors labelled by $p = 2$ and $p = 3$ correspond to type B and C, respectively. Note that the type C processor is the slowest one, hence the time needed is longer, and on the other hand to obtain the entire solution one has to wait for it to finish. However, other than in the deterministic DD, the PDD allows to use the solution computed in the various subdomains as soon as the corresponding processors complete their task. In addition, being now the subdomains fully uncoupled from each other, this allow us to minimize easily the overall CPU time, assigning a higher computational load to the faster processors, and reducing that to the slower ones. In Fig. 5, the iteration times and the number of iterations required by the local

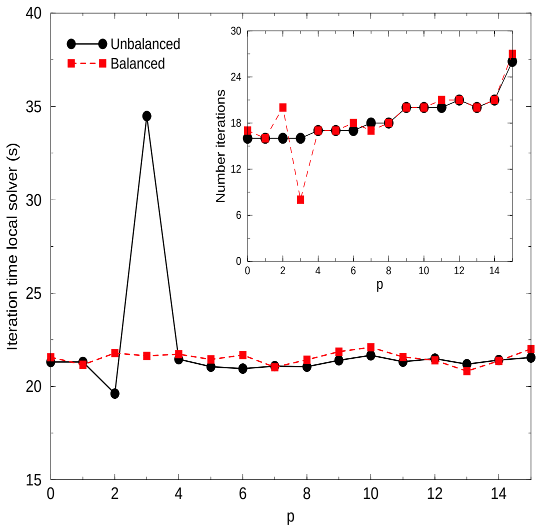


Fig. 5. Iteration times and number for each subdomain in the PDD algorithm, being computed with 16 processors

solver (labelled by p), have been displayed in case of 16 processors. A comparison is made balancing the computational load, according to the heterogeneity of the system, and without any balancing.

Table 1 shows the performance of both methods, the PDD, and DD. For the latter, two different partitions, labelled by A and B (see Fig. 4), have been used. The results for the PDD algorithm were obtained, balancing or not balancing the computational load. The two methods have been compared correspondingly to about the same maximum error, 10^{-3} (see Fig. 3). Note that the PDD algorithm scales well, as well as the DD in configuration A. However, the performance of the DD algorithm seems to be much better in configuration B, for a low number of processors, degrading when more processors are included in the system. In any case, the PDD always outperforms the DD method, the results being even more striking when the number of processors is higher. This fact is clearly due to the high intercommunication overload inherent to the DD algorithm.

In Table 2, the CPU times required to solve the problem in the subsystem Cluster of type A by the PDD and by the DD method, are shown for two different number of processors. Note that DD does not scale for both partitioning configurations, in contrast to the results shown in Table 1 for the whole heterogeneous system. The advantage in terms of CPU time achieved with the PDD method in comparison to the DD scheme, has now been reduced, even though is still important.

4 Observations and Conclusions

In view of the previous results, grid computing can be considered as an emerging alternative system, that could compete with the most powerful supercomputers

[3], in both, data parallelization *and* scientific computing. In the latter case, however, the development of suitable algorithms is still in its infancy. The probabilistic domain decomposition proposed in this paper is a simple but promising method, going in this direction. Monte Carlo methods, which are one of the ingredients of our approach, are known to be "embarrassingly parallel", but more, they seem to be fully scalable, fault tolerant, and well suited to heterogeneous computing, in particular to that special case represented by grid computing.

Acknowledgements

This work was completed during a visit of J.A.A. in Rome, supported by the Short-Term Mobility program of the Italian CNR for the year 2006. J.A.A. also acknowledges support from the Spanish Ramón y Cajal Program. R.D., and R.R were funded through grant UAH PI2005/072 and CAM-UAH2005/042. We thank J. Cardoso, and E. Moratilla for their invaluable technical support and in developing the Grid computer system environment in which our algorithms were tested.

References

1. Acebrón, J.A., Busico, M.P., Lanucara, P., Spigler, R.: Domain decomposition solution of elliptic boundary-value problems via Monte Carlo and quasi-Monte Carlo methods. SIAM J. Sci. Comput. 27, 440–457 (2005)
2. Acebrón, J.A., Busico, M.P., Lanucara, P., pigler, R.: Probabilistically induced domain decomposition methods for elliptic boundary-value problems. J. Comput. Phys. 210, 421–438 (2005)
3. Acebrón, J.A., Spigler, R.: Supercomputing applications to the numerical modeling of industrial and applied mathematics problems. J. Supercomputing (in press)
4. Boghosian, B.M., Coveney, P.V.: Scientific applications of Grid computing, vol. 7, pp. 10–13. IEEE CS Press, Los Alamitos (2005)
5. Dong, S., Karniadakis, G.E., Karonis, N.T.: Cross-site computations on the TeraGrid, vol. 7, pp. 14–23. IEEE CS Press, Los Alamitos (2005)
6. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra on High-Performance Computers. SIAM, Philadelphia (1998)
7. Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: Jin, H., Reed, D., Jiang, W. (eds.) NPC 2005. LNCS, vol. 3779, pp. 2–13. Springer, Heidelberg (2005)
8. Harvey, D.J., Das, S.K., Biswas, R.: Design and performance of a heterogeneous grid partitioner. Algorithmica 45, 509–530 (2006)
9. Huang, S., Aubanel, E., Virendrakumar, C.B.: PaGrid: A mesh partitioner for computational grids. J. Grid Computing 4, 71–88 (2006)
10. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20, 359–392 (1998)
11. Karypis, G., Kumar, V.: ParMETIS–parallel graph partitioning and field–reducing matrix ordering, `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`

12. Li, Y., Lan, Z.: A survey of load balancing in grid computing. In: Zhang, J., He, J.-H., Fu, Y. (eds.) CIS 2004. LNCS, vol. 3314, pp. 280–285. Springer, Heidelberg (2004)
13. Li, Z., Saad, Y., Sosonkina, M.: pARMS: a parallel version of the algebraic recursive multilevel solver. Numerical Linear Algebra with Applications 10, 485–509 (2003)
14. Otero, B., Cela, J.M., Badia, R.M., Labarta, J.: A domain decomposition strategy for grid environments. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3241, pp. 353–361. Springer, Heidelberg (2004)
15. Otero, B., Cela, J.M., Badia, R.M., Labarta, J.: Performance analysis of domain decomposition applications using unbalanced strategies in grid environments. In: Zhuge, H., Fox, G.C. (eds.) GCC 2005. LNCS, vol. 3795, pp. 1031–1042. Springer, Heidelberg (2005)
16. Toselli, A., Widlund, O.: Domain Decomposition Methods - Algorithms and Theory. Springer Series in Computational Mathematics, vol. 34 (2005)
17. Quarteroni, A., Valli, A.: Domain decomposition methods for partial differential equations. Oxford Science Publications, Clarendon Press (1999)
18. Distributed Computing, Science, Special Issue, 308, 809–821 (2005)
19. (2006), http://www.globus.org

Parallelization of the Mesh Refinement Algorithm of the FDEM Program Package

Torsten Adolph and Willi Schönauer

Forschungszentrum Karlsruhe, Institute for Scientific Computing,
Hermann-von-Helmholtz-Platz 1, 76344 Karlsruhe, Germany
{torsten.adolph,willi.schoenauer}@iwr.fzk.de
http://www.fzk.de/iwr

Abstract. The Finite Difference Element Method (FDEM) program package is a robust and efficient black-box solver. It solves by a Finite Difference Method arbitrary non-linear systems of elliptic and parabolic partial differential equations under arbitrary non-linear boundary conditions on arbitrary domains in 2-D and 3-D, given by a FEM mesh. From formulas of different order, we get an easy access to the discretization error. By the knowledge of this error, the mesh may be refined locally to reduce the error to a prescribed relative tolerance.

For the refinement of the elements, we first determine the refinement nodes because of either error or data organization reasons. The refinement of an element is based on halving its edges. We explain the difficulties in parallelizing the mesh refinement algorithm on distributed memory parallel computers where the processors have only local data and the refinement must be synchronized by the message passing paradigm.

Keywords: Finite difference method, Unstructured grid, Non-linear PDEs, Error estimate, Mesh refinement, Black-box solver, Parallelization.

1 Introduction

From recent textbooks about the numerical solution of partial differential equations (PDEs) (cf. [1], [2]), we learn that there are basically three main methods for the numerical solution of PDEs: The first method is the finite difference method (FDM) that dominated the early development of numerical analysis of PDEs. In the 1960s, the finite element method (FEM) was introduced by engineers, and over the last decades this method has become a widely used numerical method for PDEs. The third method is the finite volume method that is between the FDM and the FEM (widely used in CFD). We use an FDM on an unstructured FEM mesh which we call Finite Difference Element Method (FDEM).

Based on a 1-D domain decomposition of the grid, we are able to state clear rules that fix the ownership of the nodes and elements by the processors. The passing of messages to a neighbour processor during the mesh refinement is always split up into two parts.

The organization of this paper is as follows: In Section 2, the details of FDEM are presented, Section 3 describes the mesh refinement algorithm and its parallelization and, in Section 4, we present some results.

2 The Finite Difference Element Method

We want to solve elliptic and parabolic non-linear systems of PDEs in 2-D and 3-D with arbitrary non-linear boundary conditions (BCs) where we use an unstructured mesh on an arbitrary domain. The domain may be composed of several subdomains where we may have different systems of PDEs. The solutions of the subdomains are coupled by coupling conditions. We want a robust black-box solver with a reliable error estimate that we also use for the order control and for a local mesh refinement.

We discuss the solution method in 2-D; the extension to 3-D is too extensive so that we refer to [5] for details. The most general operator that we admit for the PDEs and BCs in 2-D, with the unknown solution $u(t,x,y)$, has the form

$$Pu \equiv P(t,x,y,u_t,u_x,u_y,u_{xx},u_{yy},u_{xy}) = 0 \tag{1}$$

where u and Pu are vectors with l components (system of l PDEs). If we include t and u_t, the system is parabolic, otherwise it is elliptic.

A basic paper on FDEM is [3], a progress report is [4]. A detailed report is available online, see [5].

2.1 The Generation of Difference and Error Formulas

For the generation of the difference and error formulas, we make use of a finite difference method of order q which means local approach of the solution u by a polynomial of consistency order q. The 2-D polynomial of order q is

$$P_q(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4xy + a_5y^2 + a_6x^3 + \cdots + a_{m-1}y^q \tag{2}$$

This polynomial has m coefficients a_0 to a_{m-1} where $m = (q+1)\cdot(q+2)/2$. For the determination of these m coefficients, we need m nodes with coordinates (x_0,y_0) to (x_{m-1},y_{m-1}). For example, for $q=2$ we need $m=6$ nodes.

In order to get explicit difference formulas, we make use of the principle of the influence polynomials. For a node i the influence polynomial $P_{q,i}$ of order q is defined by

$$P_{q,i}(x,y) = \begin{cases} 1 \text{ for } (x_i,y_i) \\ 0 \text{ for } (x_j,y_j),\ j \neq i \end{cases} \tag{3}$$

This means that the influence polynomial $P_{q,i}$ has function value 1 in node i and 0 in the other $m-1$ nodes. Then the discretized solution u which we denote by u_d (the index d means "discretized") can be represented by

$$u_d(x,y) := P_q(x,y) = \sum_{i=0}^{m-1} u_i \cdot P_{q,i}(x,y) \tag{4}$$

By the evaluation of $P_{q,i}$ for a grid point $x = x_j, y = y_j$, we obtain the coefficients of an interpolation polynomial at a node j. The difference formulas are the partial derivatives of the interpolation polynomial P_q, i.e. we have to differentiate (4). For example, for the difference formula for u_x which we denote by $u_{x,d}$ we get

$$u_{x,d} := \frac{\partial P_q(x,y)}{\partial x} = \sum_{i=0}^{m-1} u_i \cdot \frac{\partial P_{q,i}(x,y)}{\partial x} \tag{5}$$

One of the most critical sections is how we choose the m nodes on an unstructured FEM mesh. The nodes are collected in rings around the central node, see Fig. 1. Here we use logical masks to get the next neighbour ring of a given ring from the element list (gives nodes of an element) and the inverted element list (gives elements in which a node occurs). We do not only collect m nodes up to the

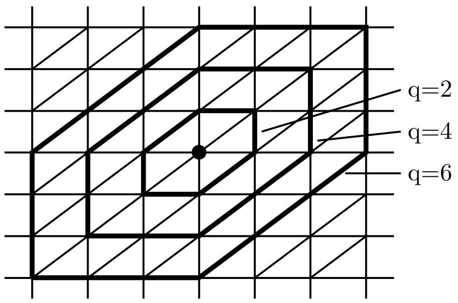


Fig. 1. Illustration for ring search

consistency order q but $m+r$ nodes up to order $q+\Delta q$ because there may be linear dependencies. A second criterion is that we collect at least $q + 2$ rings (because of the error formula). This results in $m + r$ equations for the m coefficients. We want to have nodes in the difference stars that are close to the central node. Therefore, we arrange the equations according to the ring structure and allow the crossing of a ring limit only if the current pivot element $|pivot| \leq \varepsilon_{pivot}$. The parameters Δq and ε_{pivot} determine the quality of the difference and error formulas and therefore are the key for the whole solution process. As we must determine m influence polynomials that generate the unit matrix as the right-hand sides, we must invert the matrix in reality, see [3, eqs. (27), (28)].

2.2 The Estimate of the Discretization Error

As we have formulas of arbitrary order q, we get an easy access to the estimate of the discretization error. If we denote e.g. the difference formula of order q for the derivative u_x by $u_{x,d,q}$, the error estimate d_x is defined by

$$d_x = u_{x,d,q+2} - u_{x,d,q} \tag{6}$$

i.e. by the difference to the order $q+2$. Because the exact discretization error is the difference of the derivative and the difference formula of order q, we see that the derivative is replaced by a "better" formula for the estimate which holds only for sufficiently fine grid. Equation (6) is the key for our explicit error access.

2.3 The Error Equation

Pu (1) is an arbitrary non-linear function of its arguments. Therefore, we linearize system (1) with the Newton Raphson method and then discretize the resulting linear Newton-PDE replacing e.g.

$$u_x \Leftarrow u_{x,d} + d_x \tag{7}$$

and analogously the other derivatives. After linearizing also in the discretization errors, we finally get the error equation for the overall error Δu_d:

$$\begin{aligned} \Delta u_d = \quad & \Delta u_{Pu} + \Delta u_{D_t} + \Delta u_{D_x} + \Delta u_{D_y} + \Delta u_{D_{xy}} \quad \text{(level of solution)} \\ = Q_d^{-1} \cdot & [(Pu)_d \ + \ D_t \ + \ \{D_x \ + \ D_y \ + \ D_{xy}\}] \quad \text{(level of equation)} \end{aligned} \tag{8}$$

Here, Q_d denotes the large sparse matrix resulting from the discretization. The inverse Q_d^{-1} is never explicitly computed as it is a full matrix. Instead, the system is solved iteratively. $(Pu)_d$ is the discretized Newton residual and the D_μ are discretization error terms that result from the linearization in the d_μ, e.g.

$$D_x = \frac{\partial Pu}{\partial u_x} \cdot d_x + \frac{\partial Pu}{\partial u_{xx}} \cdot d_{xx} \tag{9}$$

In the parentheses of the second row of (8) we have error terms that can be computed "on the level of equation" and that are transformed by Q_d^{-1} to the "level of solution". These corresponding errors on the solution level are arranged above their source terms. So the overall error Δu_d has been split up into the parts that result from the corresponding terms on the level of equation.

The only correction that is applied is the Newton correction Δu_{Pu} that results from the Newton residual $(Pu)_d$. It is computed from

$$Q_d \Delta u_{Pu} = (Pu)_d \tag{10}$$

The other error terms in the first row of (8) are only used for the error control. If we applied these terms, we had no error estimate any more. This approach also implies that we can explicitly follow the effect of a discretization error to the level of solution.

2.4 Parallelization

The numerical solution of large PDE problems needs much computation time and memory. Therefore, we need an efficiently parallelized program that is executed on distributed memory parallel computers with message passing (MPI).

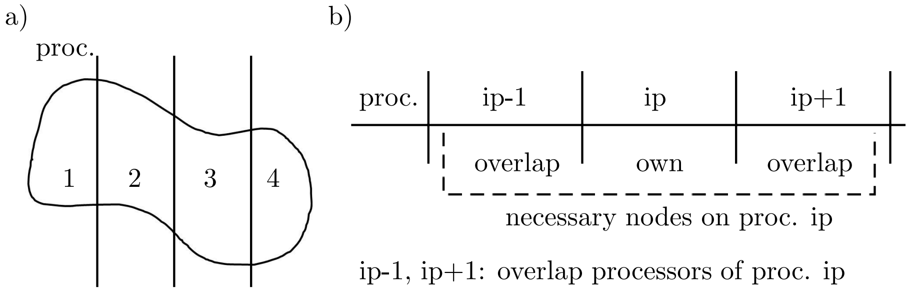

a)
proc.
1
2
3
4
b)
proc.
ip-1
ip
ip+1
overlap
own
overlap
necessary nodes on proc. ip
ip-1, ip+1: overlap processors of proc. ip

Fig. 2. Illustration for 1-D domain decomposition with overlap

In FDEM, we re-sort the nodes for their x-coordinate and distribute them in np equal parts on the np processors which results in a 1-D domain decomposition, see Fig. 2a). The elements are distributed correspondingly: an element is owned by the processor that owns its leftmost node. If we want to execute the generation and evaluation of the difference and error formulas and the computation of the matrix Q_d and the r.h.s. $(Pu)_d$ purely local without communication, we also have to store on processor *ip* node and element information of its left and right neighbour(s) which is indicated as overlap, see Fig. 2b).

3 The Algorithm of the Mesh Refinement

For the structure of the space, we use linear triangles in 2-D and tetrahedrons in 3-D. The mesh refinement consists of two main parts. First, we have to determine all elements that have to be refined, either because of the error or because of the refinement cascade that becomes necessary for data organization reasons. The second part consists of the refinement of the chosen elements. We first halve the edges of each refinement element by creating a mid-point on the edge and afterwards combine the nodes of an element to 4 new elements in 2-D and 8 new elements in 3-D, respectively.

3.1 Refinement Nodes and Elements

The user prescribes a global relative tolerance *tol* for the solution, and the refinement process is stopped if

$$\|\Delta u_d\| / \|u_d\| \leq tol \tag{11}$$

holds. For the control of the solution process, we need a corresponding value *tolg* on the level of equation: We transform *tol* to *tolg* like the Newton correction to the Newton residual. The value of *tolg* is compared to the maximum of the space key error terms of the l components of a node. If for a node i

$$\max_{\nu=1}^{l} |\{D_x + D_y + D_{xy}\}_{i,\nu}| \leq s_{grid} \cdot tolg \tag{12}$$

with a safety factor s_{grid} does not hold, the node becomes a refinement node. All elements a refinement node belongs to become refinement elements. We enter the value *true* into a logical array *refel* in the row of the local number of the refinement element and the column corresponding to its refinement stage (the original elements have refinement stage 0, the refinement stage is increased by 1 with each refinement), see Table 1. The array *refel* is the key for the whole refinement process.

Table 1. Shape of the logical array *refel*

element number	belongs to ref. stage	*refel*			
		stage 0	stage 1	stage 2	stage 3
1	1	false	true	false	false
2	0	true	false	false	false
3	3	false	false	false	true
⋮	⋮	⋮	⋮	⋮	⋮
ne_l	0	true	false	false	false

For reasons of data organization, we allow only 3 nodes on an edge of an element, so we must avoid that in an element a fourth node is generated on an edge by the refinement of a neighbour element. This is done by the preceding refinement of the concerned element and is denoted by refinement cascade. The difficulties of the refinement cascade result from the parallelization of FDEM. An element is refined by the processor that owns the element, and an element is owned by the processor that owns the leftmost node of this element. As the refinement nodes or the elements that evoke the refinement cascade may not be owned by the same processors that own the resulting refinement elements, communication becomes necessary, see Fig. 3. Here, node 1 on processor ip_4 is

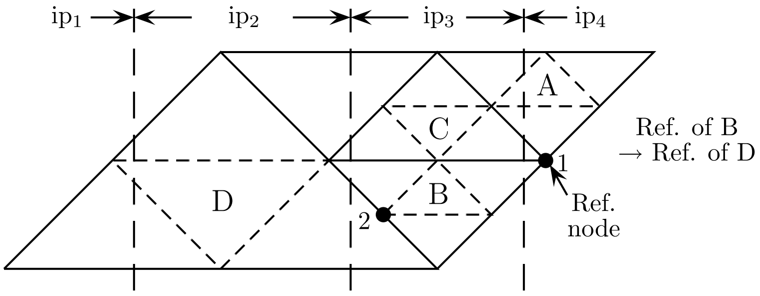


Fig. 3. Illustration of the refinement cascade on $np = 4$ processors

a refinement node because of the error. Therefore, elements A, B and C (solid lines) have to be refined. By the refinement of element B, there would be created a fourth node (node 2) on the right edge of element D. As the refinement node 1 is owned by processor ip_4, but neither element A nor B nor C are elements owned by processor ip_4, this processor has to send the information about the necessary refinement of the three elements to processor ip_3 (for element A) and processor ip_2 (for elements B and C). If we look for the neighbours of element B on processor ip_2, we see that the neighbour element D has a lower refinement stage and therefore has to be refined because of the refinement cascade. But element D is on processor ip_1, so that processor ip_2 has to send the information about the refinement of element D to processor ip_1.

So obviously a sophisticated algorithm is necessary for two reasons: first, data exchange may be necessary not only to the direct neighbour processors but to several of the overlap processors whose data are also stored on the own processor. Second, the elements that have to be refined because of the refinement cascade may evoke new refinement elements, i.e. the cascade may continue. This may cause further communication.

We proceed as following: Each processor determines the elements that must be refined because of the error. Then we determine the elements that are owned by overlap processors. The element numbers of these elements are collected in an array and afterwards sent to the corresponding processors. There they are entered into the local *refel* array, and we determine the new refinement elements that have to be refined because of the refinement cascade. Again, the refinement elements that are owned by overlap processors are determined, and the element numbers are sent to the corresponding overlap processors. This process stops if there are not any refinement elements at the processor borders that are owned by an overlap processor.

After the last step of the refinement cascade, all refinement elements are known, and during the refinement process no new refinement elements are added.

3.2 Refinement of the Elements

The refinement of the elements is carried out in refinement stages. We start with the largest elements, after their refinement the second largest elements are refined etc. As the refinement of an element is based on the halving of its edges, we first have to transform the refinement element list into a refinement edge list where the information for a refinement edge consists of the following data: local node numbers of the endpoints, local element number, local position within the element, number of neighbour elements at the edge, local numbers of the neighbour elements and (to be added later) the new node number of the mid-point. The reason why we need all this data for an edge is that the unit “edge” does not exist in FDEM. An edge is identified by its two endpoints, and the new mid-point is generated by the processor that owns the edge, i.e. the processor that owns the leftmost endpoint of the edge.

We have two arrays for the refinement edge information, one for the local edges and one for the edges that are in the overlap, i.e. that are owned by a neighbour processor. For each neighbour processor, there is one column in this array. Here, we have to enter global node and element numbers.

Therefore, we need communication again, i.e. the information needed for the generation of a new mid-point must be sent to the processor that owns the refinement edge. Having generated the mid-points of all received edges, this processor sends back the new node numbers. After the mid-points of the refinement edges in the overlap have got their node numbers, the new mid-points are generated for the refinement edges that are owned by the processor that also owns the refinement element. Afterwards each of the newly-created nodes gets assigned coordinates, consistency order and function values, and we check if there have been created any new boundary nodes.

Each refinement element then has 3 (4) corner nodes and 3 (6) mid-points on the edges in 2-D (3-D). We separate an element in such a way that 4 (8) new elements result, see Fig. 4.

In each refinement stage, the information what happened to the refinement elements of the previous stages must be available on the concerned processors. Therefore, we have a sophisticated communication algorithm to provide the necessary information for each processor.

The mesh refinement algorithm is described in detail in [6].

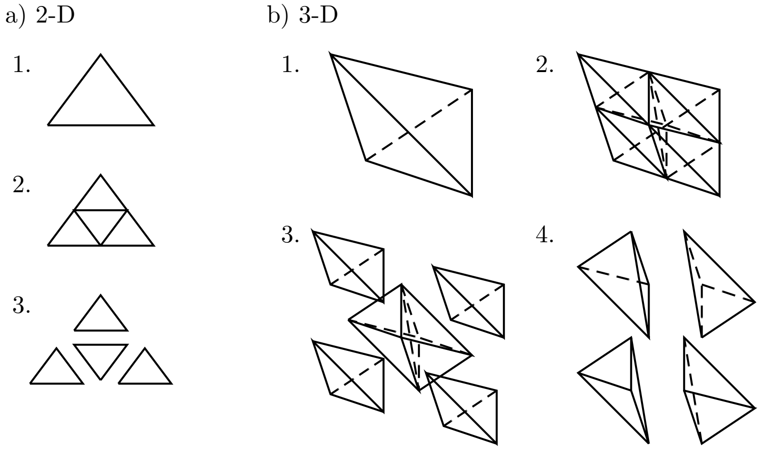


Fig. 4. Illustration of the separation of an element

3.3 Communication Patterns

We developed communication patterns that allow a very efficient data exchange. The data transfer is split up into two parts. First we only send the number of nodes, elements etc. to the overlap processors so that on each processor the send and receive commands can be set in such a way that it is known on each processor from which processor it will receive a message and that we may also compute the corresponding message length. We are also able to allocate the length of the

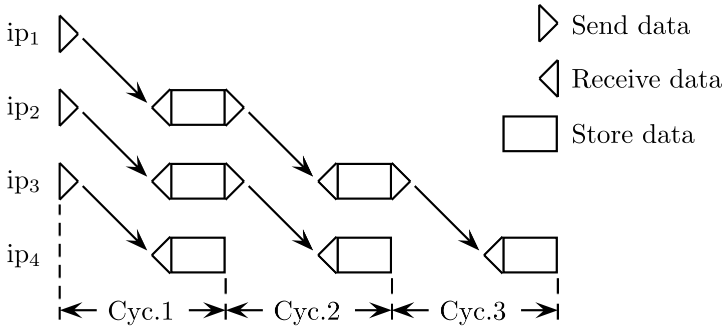


Fig. 5. Illustration of the passing of the message lengths to the right for $np = 4$ processors and $np_{max,r} = 3$

receive buffer by the required length so that we save storage. This preparing step needs $np_{max,r} - 1$ cycles at most, see Fig. 5 for the communication to the right, where $np_{max,r}$ is the maximum number of right overlap processors. Then the information from a processor ip_j has arrived at processor $ip_{j+np_{max,r}}$.

In a second step, the actual data exchange takes place. We only send messages to overlap processors data must be sent to, and by individual message lengths we save further communication time. We also save computation time because we avoid that those overlap processors receiving a superfluous message would try to extract some useful data from that message afterwards. This second part also consists of $np_{max,r}$ steps, where we send the necessary messages from processor ip_j to processor ip_{j+i} in step i if $j \leq np - i$ holds.

Moreover, instead of sending an individual message for each piece of information, we do not start the communication between two processors until we have collected in a buffer all data that has to be sent to the target processor. This reduces the consumed startup time drastically.

4 Results

In order to see the influence of the grid, we make a series computation with 6 different grids on a 4×1 domain. The characteristics of the 6 grids are shown in Table 2. The number of grid points in x- and y-direction is doubled from one grid to the other. This results in the fourfold number of total grid points and approximately the fourfold number of elements. After the first computation cycle, the mesh is completely refined. As we quadruple the number of nodes, we also quadruple the number of processors so that the number of nodes (and elements) on a processor is the same for each computation.

The computations have been carried out on the IBM Blue Gene/L at the Forschungszentrum Jülich, Germany, with 700 MHz PowerPC 440 processors with a peak performance of 2800 MFLOPS.

In the last column of Table 2, the CPU sec. of master processor 1 for the pure mesh refinement is given. One can see that the mesh refinement scales well, but

Table 2. Characteristics of the 6 grids for the mesh refinement

Grid no.	Dimension	Original grid Number of nodes	Original grid Number of elements	Refined grid Number of nodes	Refined grid Number of elements	No. of proc.	CPU sec. for ref.
1	512 × 128	65,536	129,794	260,865	519,176	1	1.95
2	1,024 × 256	262,144	521,730	1,046,017	2,086,920	4	1.93
3	2,048 × 512	1,048,576	2,092,034	4,189,185	8,368,136	16	2.04
4	4,096 × 1,024	4,194,304	8,378,370	16,766,977	33,513,480	64	2.13
5	8,192 × 2,048	16,777,216	33,533,954	67,088,385	134,135,816	256	2.20
6	16,384 × 4,096	67,108,864	134,176,770	268,394,497	536,707,080	1024	2.28

for higher numbers of processors the communication overhead gradually affects the performance as time increases slightly. On the other hand, the number of nodes in y-direction is doubled from one grid to the other. However, a higher number of grid points in y-direction means increasing message lengths during the mesh refinement as there are more edges at the processor borders for which we have to exchange the refinement edge information so that the results in Table 2 are acceptable.

References

1. Langtangen, H.P.: Computational Partial Differential Equations. In: Texts in Computational Science and Engineering 1, 2nd edn., Springer, Heidelberg (2003)
2. Larsson, S., Thomée, V.: Partial Differential Equations with Numerical Methods. In: Texts in Applied Mathematics, vol. 45, Springer, Heidelberg (2003)
3. Schönauer, W., Adolph, T.: How WE solve PDEs. Journal of Computational and Applied Mathematics 131, 473–492 (2001)
4. Schönauer, W., Adolph, T.: How we make the FDM more flexible than the FEM. In: Vigo-Aguiar, J., Wade, B.A. (eds.) Computational and Mathematical Methods in Science and Engineering. Proceedings of the CMMSE-2002, Deposito Legal (Spain) S.1026-2002, vol. II, pp. 313–322 (2003) (Also in: Journal of Computational and Applied Mathematics 158(1), 157–167 (2003))
5. Schönauer, W., Adolph, T.: FDEM: The Evolution and Application of the Finite Difference Element Method (FDEM) Program Package for the Solution of Partial Differential Equations. Universität Karlsruhe (TH) (2005), Available at `http://www.rz.uni-karlsruhe.de/rz/docs/FDEM/Literatur/fdem.pdf`
6. Adolph, T.: The Parallelization of the Mesh Refinement Algorithm in the Finite Difference Element Method. Doctoral Thesis. Universität Karlsruhe (TH) (2005), Available at `http://www.rz.uni-karlsruhe.de/rz/docs/FDEM/Literatur/par_mra_fdem.pdf`

Load Balancing for the Numerical Solution of the Navier-Stokes Equations

Gregory Karagiorgos[1], Petros Katsafados[3], Andreas Kontarinis[2], Nikolaos M. Missirlis[1], and Filippos Tzaferis[1]

[1] Department of Informatics, University of Athens,
Panepistimiopolis, GR-15784,
Athens, Greece
{greg, nmis, ftzaf}@di.uoa.gr
http://parallel.di.uoa.gr
[2] ankont@mycosmos.com
[3] Department of Physics, University of Athens,
Panepistimiopolis, GR-15784,
Athens, Greece
pkatsaf@mg.uoa.gr
http://parallel.di.uoa.gr

Abstract. In this paper, we simulate the performance of a load balancing scheme. In particular, we study the application of the Extrapolated Diffusion(EDF) method for the efficient parallelization of a simple *atmospheric* model. It involves the numerical solution of the steady state Navier-Stokes(NS) equations in the horizontal plane and random load values, corresponding to the *physics* computations, in the vertical plane. For the numerical solution of NS equations, we use the local Modified Successive Overrelaxation (LMSOR) method with local parameters thus avoiding the additional cost caused by the global communication of the involved parameter ω in the classical SOR method. We have implemented an efficient domain decomposition technique by using a larger number of processors in the areas of the domain with heavier work load. With our balancing scheme, a gain of approximately 45% in execution time is achieved, in certain cases.

1 Introduction

In this paper we study the application of the Diffusion method for the efficient parallelization of a simple model. It involves the numerical solution of the steady state Navier-Stokes(NS) equations in the horizontal plane and random load values, corresponding to the *physics* computations, in the vertical plane. Our intention is to study the performance of a load balancing scheme under more realistic conditions than in previous studies [12,13]. Up to now the problem of load balancing has been studied solely, namely without any combination of horizontal and vertical computations. In this paper it is the first time, where (i) an optimal version of the diffusion method is used and (ii) the migration of load transfer is also implemented using only local communication as opposed to schedules which require global communication.

The paper is organized as follows. In Section 2 we present the Extrapolated Diffusion(EDF) method, which possesses the fastest rate of convergence among its other counterparts [16]. In Section 3 we describe our simple *atmospheric model* and we present the use of the local Modified Successive Overrelaxation (LMSOR) method for the numerical solution of NS equations. In Section 4 we describe a domain decomposition technique, which initially assigns a square domain to each processor in a mesh network. However, load balancing imposes a modified domain decomposition. As this increases the communication complexity, we propose a load transfer to balance the load along the row processors only. This domain decomposition technique assigns a larger number of processors in the areas of the domain with heavier load, thus increasing the efficiency of load balancing. Finally, in section 5 we present our simulation results.

2 The Extrapolated Diffusion (EDF) Method

Let us consider an arbitrary, undirected, connected graph $G = (V, E)$. This graph represents our processor network, where its nodes, namely the processors, are denoted by $v_i \in V$ and its edges (links) are $(v_i, v_j) \in E$, when v_i, v_j are neighbors. Furthermore, we assign a weight $u_i \geq 0$ to each node, which corresponds to the computational load of v_i. The processor graph reflects the inter-connection of the subdomains of a mesh that has been partitioned and distributed amongst processors (see Figure 1). In this graph each node represents a subdomain and two nodes are linked by an edge if the corresponding subdomains share edges of the mesh.

The Extrapolated Diffusion (EDF) method for the load balancing has the form [1,4]

$$u_i^{(n+1)} = u_i^{(n)} - \tau \sum_{j \in A(i)} c_{ij} \left(u_i^{(n)} - u_j^{(n)} \right) \tag{1}$$

where c_{ij} are diffusion parameters, $A(i)$ is the set of the nearest neighbors of node i of the graph $G = (V, E)$, $u_i^{(n)}$, $i = 0, 1, 2, \ldots, |V|$ is the load after the n-th iteration on node i and $\tau \in \mathbb{R}\backslash\{0\}$ is a parameter that plays an important role in the convergence of the whole system to the equilibrium state. The overall workload distribution at step n, denoted by $u^{(n)}$, is the transpose of the vector $(u_1^{(n)}, u_2^{(n)}, \ldots, u_{|V|}^{(n)})$ and $u^{(0)}$ is the initial workload distribution. In matrix form (1) becomes

$$u^{(n+1)} = M u^{(n)} \tag{2}$$

where M is called the *diffusion matrix*. The elements of M, m_{ij}, are equal to τc_{ij}, if $j \in A(i)$, $1 - \tau \sum_{j \in A(i)} c_{ij}$, if $i = j$ and 0 otherwise. With this formulation, the features of diffusive load balancing are fully captured by the iterative process (2) governed by the diffusion matrix M. Also, (2) can be written as $u^{(n+1)} = (I - \tau L)u^{(n)}$, where $L = BWB^T$ is the *weighted Laplacian* matrix of the graph, W is a diagonal matrix of size $|E| \times |E|$ consisting of the coefficients

c_{ij} and B is the vertex-edge incident matrix. At this point, we note that if $\tau = 1$, then we obtain the Diffusion (DF) method proposed by Cybenko [4] and Boillat [1], independently. If $W = I$, then we obtain the special case of the DF method with a single parameter τ *(unweighted Laplacian).* In the unweighted case and for network topologies such as chain, 2D-mesh, nD-mesh, ring, 2D-torus, nD-torus and nD-hypercube, optimal values for the parameter τ that maximize the convergence rate have been derived by Xu and Lau [21,22]. However, the same problem, in the weighted case was solved recently [16]. Next, we consider the weighted case. The diffusion matrix of EDF can be written as

$$M = I - \tau L, \qquad L = D - A \tag{3}$$

where $D = diag(L)$ and A is the weighted adjacency matrix. Because of (3), (2) becomes $u^{(n+1)} = (I - \tau D)\, u^{(n)} + \tau A u^{(n)}$ or in component form

$$u_i^{(n+1)} = \left(1 - \tau \sum_{j \in A(i)} c_{ij}\right) u_i^{(n)} + \tau \sum_{j \in A(i)} c_{ij} u_j^{(n)}, i = 1, 2, \ldots, |V| \tag{4}$$

The diffusion matrix M must have the following properties: nonnegative, symmetric and stochastic [4,1]. The eigenvalues of L are $0 = \lambda_1 < \lambda_2 \leq \ldots \leq \lambda_n$. In case $c_{ij} =$ constant, the optimum value of τ is attained at [20,23]

$$\tau_o = \frac{2}{\lambda_2 + \lambda_n}$$

and the corresponding minimum value of the convergence factor

$$\gamma(M) = \max\{|1 - \tau\lambda_n|, |1 - \tau\lambda_2|\}$$

is given by

$$\gamma_o(M) = \frac{P(L) - 1}{P(L) + 1}, \text{ where } P(L) = \frac{\lambda_n}{\lambda_2}$$

which is the P-condition number of L. Note that if $P(L) \gg 1$, then the rate of convergence of the EDF method is given by

$$R(M) = -\log \gamma_o(M) \simeq \frac{2}{P(L)}$$

which implies that the rate of convergence of the EDF method is a decreasing function of $P(L)$. The problem of determining the diffusion parameters c_{ij} such that EDF attains its maximum rate of convergence is an active research area [5,9,16]. Introducing the set of parameters $\tau_i, i = 1, 2, \ldots, |V|$, instead of a fixed parameter τ in (4), the problem moves to the determination of the parameters τ_i in terms of c_{ij}. By considering local Fourier analysis [15,16] we were able to determine good values (near the optimum) for τ_i. These values become optimum(see Table 1) in case the diffusion parameters are constant in each dimension and satisfy the relation $c_j^{(2)} = \sigma_2 c_i^{(1)}$, $i = 1, 2, \ldots, N_1$, $j = 1, 2, \ldots, N_2$,

Table 1. Formulae for the optimum τ_o and $\gamma_o(M)$ (E: Even, O: Odd)

N_1	N_2	Case	τ_o	$\gamma_o(M)$
E	E	1	$\left(3+2\sigma_2-\cos\frac{2\pi}{N_1}\right)^{-1}$	$\frac{1+2\sigma_2+\cos\frac{2\pi}{N_1}}{3+2\sigma_2-\cos\frac{2\pi}{N_1}}$
O	O	2	$\left(2+\sigma_2(1+\cos\frac{\pi}{N_2})+\cos\frac{\pi}{N_1}-\cos\frac{2\pi}{N_1}\right)^{-1}$	$\frac{\cos\frac{\pi}{N_1}+\cos\frac{2\pi}{N_1}+\sigma_2(1+\cos\frac{\pi}{N_2})}{2+\sigma_2(1+\cos\frac{\pi}{N_2})+\cos\frac{\pi}{N_1}-\cos\frac{2\pi}{N_1}}$
E	O	3	$\left(3-\cos\frac{2\pi}{N_1}+\sigma_2(1+\cos\frac{\pi}{N_2})\right)^{-1}$	$\frac{1+\cos\frac{2\pi}{N_1}+\sigma_2(1+\cos\frac{\pi}{N_2})}{3-\cos\frac{2\pi}{N_1}+\sigma_2(1+\cos\frac{\pi}{N_2})}$
O	E	4	$\left(2+2\sigma_2+\cos\frac{\pi}{N_1}-\cos\frac{2\pi}{N_1}\right)^{-1}$	$\frac{2\sigma_2+\cos\frac{\pi}{N_1}-\cos\frac{2\pi}{N_1}}{2+2\sigma_2+\cos\frac{\pi}{N_1}-\cos\frac{2\pi}{N_1}}$

where $\sigma_2=\frac{1-\cos\frac{2\pi}{N_1}}{1-\cos\frac{2\pi}{N_2}}$ and $c_i^{(1)}$, $c_j^{(2)}$ are the row and column diffusion parameters, respectively, of the torus [16]. Also, in [16] it is proven that at the optimum stage[1]

$$RR_\infty(EDF)\simeq 2RR_\infty(DF)$$

which means that EDF is twice as fast as DF for stretched torus, that is a torus with either $N_1\gg N_2$ or $N_2\gg N_1$.

In order to further improve, by an order of magnitude, the rate of convergence of EDF we can apply accelerated techniques (Semi-Iterative, Second-Degree and Variable Extrapolation) following [20,19,14].

2.1 The Semi-iterative method

We now consider iterative schemes for further accelerating the convergence of EDF. It is known [20,23] that the convergence of (2) can be greatly accelerated if one uses the Semi-Iterative scheme

$$u^{(n+1)}=\rho_{n+1}(I-\tau_oL)u^{(n)}+(1-\rho_{n+1})u^{(n-1)}$$

with

$$\rho_1=1,\ \rho_2=\left(1-\frac{\sigma^2}{2}\right)^{-1},\ \rho_{n+1}=\left(1-\frac{\sigma^2}{4}\rho_n\right)^{-1},\ n=2,3,\ldots,$$

and

$$\sigma=\gamma_o(M) \tag{5}$$

It is worth noting that σ is equal to $\gamma_o(M)$, which is the minimum value of the convergence factor of EDF. In addition, $\gamma_o(M)$ and τ_o, for EDF, are given by the expressions of Table 1 for the corresponding values of N_1 and N_2. It can be shown [20,23] that

$$RR_\infty(SI-EDF)\simeq\frac{1}{\sqrt{2}}RR_\infty(SI-DF)$$

[1] $RR(.)=\frac{1}{R(.)}$.

which indicates that the number of iterations of SI-EDF will be approximately 30% less than the number of iterations of SI-DF in case of stretched torus.

3 The Atmospheric Model

As a case study we will consider the simulation of a simple atmospheric model. The atmospheric models of climate and weather use a 3-dimensional grid to represent the atmosphere's behavior. The involved computations in these grids are of two kinds: *dynamics* and *physics* [13,12]. The dynamics calculations correspond to the fluid dynamics of the atmosphere and are applied to the horizontal field. These calculations use explicit schemes to discretize the involved partial differential equations because they are inherently parallel. Alternatively, the physics calculations represent the natural procedures such as clouds, moist convection, the planetary boundary layer and surface processes and are applied to the vertical level. The computations of a vertical column are local, that is, they do not need data from the neighboring columns and are implicit in nature.

As far as our simulation study is concerned, we will model the computations in the horizontal plane by solving the Navier-Stokes equations and the computations in the vertical plane with random load values, corresponding to the implicit *physics* calculations.

3.1 Numerical Solution of the Navier-Stokes(NS) Equations

The model problem considered here is that of solving the 2-D incopressible Navier-Stokes (NS) equations. The equations in terms of vorticity Ω and stream function Ψ are

$$\Delta\Psi = -\Omega, \qquad u\frac{\partial\Omega}{\partial x} + v\frac{\partial\Omega}{\partial y} = \frac{1}{Re}\Delta\Omega \tag{6}$$

where Re is the Reynold number of the fluid flow and u, v are the velocity components in the y and x directions, respectively. The velocity components are given in terms of the stream function Ψ by $u = \frac{\partial\Psi}{\partial y}$ and $v = -\frac{\partial\Psi}{\partial x}$. If the computational domain is the unit square, then the boundary conditions for such flow are given by $\Psi = 0, \frac{\partial\Psi}{\partial x} = 0$ at $x = 0$ and $x = 1$, $\Psi = 0$, $\frac{\partial\Psi}{\partial y} = 0$ at $y = 0$ and $\Psi = 0$, $\frac{\partial\Psi}{\partial y} = -1$ at $y = 1$. The 5-point discretization of NS equations on a uniform grid of mesh size $h = 1/N$ leads to

$$\frac{1}{h^2}\left[-4\Psi_{ij} + \Psi_{i-1,j} + \Psi_{i+1,j} + \Psi_{i,j+1} + \Psi_{i,j-1}\right] = \Omega_{ij} \tag{7}$$

and

$$\Omega_{ij} = l_{ij}\Omega_{i-1,j} + r_{ij}\Omega_{i+1,j} + t_{ij}\Omega_{i,j+1} + b_{ij}\Omega_{i,j-1} \tag{8}$$

with $l_{ij} = 1/4 + 1/16Reu_{ij}$, $r_{ij} = 1/4 - 1/16Reu_{ij}$, $t_{ij} = 1/4 - 1/16Rev_{ij}$, $b_{ij} = 1/4 + 1/16Rev_{ij}$, where $u_{ij} = \Psi_{i,j+1} - \Psi_{i,j-1}$, $v_{ij} = \Psi_{i-1,j} - \Psi_{i+1,j}$. The boundary conditions are also given by $\Omega_{i,0} - \frac{2}{h^2}\Psi_{i,1} = 0, \Omega_{N,j} - \frac{2}{h^2}\Psi_{N-1,j} = 0$ and $\Omega_{0,j} - \frac{2}{h^2}\Psi_{1,j} = 0, \Omega_{i,N} - \frac{2}{h^2}(h - \Psi_{i,N-1}) = 0$

3.2 The Local Modified SOR Method

The local SOR method was introduced by Ehrlich [7,8] and Botta and Veldman [2] in an attempt to further increase the rate of convergence of SOR. The idea is based on letting the relaxation factor ω vary from equation to equation. Kuo et. al [18] combined local SOR with red black ordering and showed that is suitable for parallel implementation on mesh connected processor arrays. In the present study we generalize local SOR by letting two different sets of parameters $\omega_{ij}, \omega'_{ij}$ to be used for the red ($i+j$ even) and black ($i+j$ odd) points, respectively. An application of our method to (7) and (8) can be written as follows

$$\Omega_{ij}^{(n+1)} = (1-\omega_{ij})\Omega_{ij}^{(n)} + \omega_{ij} J_{ij} \Omega_{ij}^{(n)}, \qquad i+j \text{ even}$$

$$\Omega_{ij}^{(n+1)} = (1-\omega'_{ij})\Omega_{ij}^{(n)} + \omega'_{ij} J_{ij} \Omega_{ij}^{(n+1)}, \qquad i+j \text{ odd}$$

with

$$J_{ij}\Omega_{ij}^{(n)} = l_{ij}^{(n)}\Omega_{i-1,j}^{(n)} + r_{ij}^{(n)}\Omega_{i+1,j}^{(n)} + t_{ij}^{(n)}\Omega_{i,j+1}^{(n)} + b_{ij}^{(n)}\Omega_{i,j-1}^{(n)}$$

and $l_{ij}^{(n)} = 1/4 + 1/16Reu_{ij}^{(n)}$, $r_{ij}^{(n)} = 1/4 - 1/16Reu_{ij}^{(n)}$, $t_{ij}^{(n)} = 1/4 - 1/16Rev_{ij}^{(n)}$, $b_{ij}^{(n)} = 1/4 + 1/16Rev_{ij}^{(n)}$ where $u_{ij}^{(n)} = \Psi_{i,j+1}^{(n)} - \Psi_{i,j-1}^{(n)}$, $v_{ij} = \Psi_{i-1,j}^{(n)} - \Psi_{i+1,j}^{(n)}$. A similar iterative scheme holds also for Ψ_{ij}. If μ_{ij} are real, then the optimum values of the LMSOR parameters are given by [3]

$$\omega_{1,i,j} = \frac{2}{1 - \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1-\overline{\mu}_{ij}^2)(1-\underline{\mu}_{ij}^2)}}, \quad \omega_{2,i,j} = \frac{2}{1 + \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1-\overline{\mu}_{ij}^2)(1-\underline{\mu}_{ij}^2)}}$$

where $\overline{\mu}_{ij}$ and $\underline{\mu}_{ij}$ are computed by $\overline{\mu}_{ij} = 2\left(\sqrt{\ell_{ij}r_{ij}}\cos\pi h + \sqrt{t_{ij}b_{ij}}\cos\pi k\right)$, $\underline{\mu}_{ij} = 2\left(\sqrt{\ell_{ij}r_{ij}}\cos\frac{\pi(1-h)}{2} + \sqrt{t_{ij}b_{ij}}\cos\frac{\pi(1-k)}{2}\right)$, with $h = k = 1/\sqrt{N}$

If μ_{ij} are imaginary, then the optimum values of the LMSOR parameters are given by

$$\omega_{1,i,j} = \frac{2}{1 - \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1+\overline{\mu}_{ij}^2)(1+\underline{\mu}_{ij}^2)}}, \quad \omega_{2,i,j} = \frac{2}{1 + \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1+\overline{\mu}_{ij}^2)(1+\underline{\mu}_{ij}^2)}}$$

If $\mu_{ij} = \mu_{Rij} + i\mu_{Iij}$ are complex, we propose the following heuristic formulas

$$\omega_{1,i,j} = \frac{2}{1 + \overline{\mu}_R\underline{\mu}_R - \overline{\mu}_I\underline{\mu}_I(1 - (\underline{\mu}_R\overline{\mu}_R)^{2/3})^{-1} + \sqrt{M_{R,I}}}$$

$$\text{and} \qquad \omega_{2,i,j} = \frac{2}{1 - \overline{\mu}_R\underline{\mu}_R + \overline{\mu}_I\underline{\mu}_I(1 - (\underline{\mu}_R\overline{\mu}_R)^{2/3})^{-1} + \sqrt{M_{R,I}}}$$

where $\quad M_{R,I} = [1 - \overline{\mu}_R^2 + \overline{\mu}_I^2(1 - \overline{\mu}_R^{2/3})^{-1}][1 - \underline{\mu}_R^2 + \underline{\mu}_I^2(1 - \underline{\mu}_R^{2/3})^{-1}]$

4 Domain Decomposition and Load Transfer

Let us assume the domain for the solution of the NS equations is rectangular. Initially, we apply a domain decomposition technique which divides the original domain into p square subdomains, where p is the number of available processors(see Figure 1). This decomposition proved to be optimal, in case the load is the same on all mesh points [3], in the sense of minimizing the ratio communication over computation. Next, each subdomain is assigned to a processor in a mesh network. The parallel efficiency depends on two factors: an equal distribution of computational load on the processors and a small communication overhead achieved by minimizing the boundary length. If the latter is achieved by requiring the minimization of the number of cut edges i.e. the total interface length, the mesh partitioning problem turns out to be NP-complete. Fortunately, a number of graph partitioning heuristics have been developed (see e.g. [6,10]). Most of them try to minimize the cut size which is sufficient for many applications but this approach has also its limitations [11]. To avoid these limitations we apply a load balancing scheme such as to maintain the structure of the original decomposition. This is achieved by reducing or increasing, according to a simple averaging load rule, the width of each row. Although this approach reduces the effectiveness of the method, as will not achieve full balance, it proves to be efficient. Next, we consider a load balancing scheme, which employs the above feature. Let us assume the situation as illustrated on the left of Figure 1. The shaded area denotes the physics computations, i.e. the load distribution. In an attempt to balance the load among the processors we decompose the load area into smaller domains as this is illustrated on the right of Figure 1. We will refer to this partitioning as *nesting*. The advantage of nesting is that the structure of the domain decomposition graph remains unchanged, thus minimizing the interprocessor communication at the cost of imbalance. The problem now is to determine the width of each row and column. Let us consider the case presented in Figure 2, where we have four processors a, b, c, d, each one assigned initially a square with the same area. Further, we assume that the result of the EDF

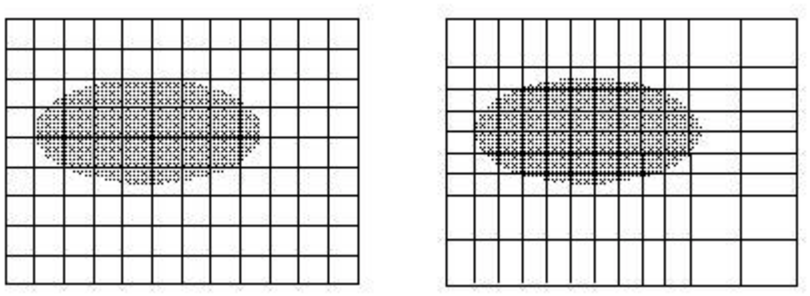


Fig. 1. Domain decomposition by *nesting*

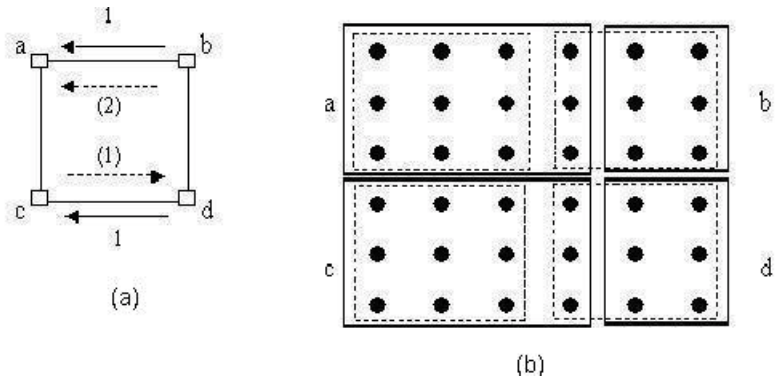


Fig. 2. Column transfer according to the *weighted average* method

algorithm is that processor a must receive two columns of mesh points from processor b and processor c must send one column to processor d (Figure 2(a) dotted arrows). But if these transfers are carried out, they will destroy the mesh structure of the domain decomposition graph as now processor d will have two neighbors a and b. In order to avoid this phenomenon we allow the *weighted average* number of columns to be transferred among the processors a, b and c, d, where *weighted average*= $\lceil \frac{2+(-1)}{2} \rceil = 1$. This means that processor a will receive one column(instead of two) and processor c will also receive one column(instead of sending one). After the load transfer is carried out, the domain graph remains a mesh as is depicted in Figure 2(b) with the solid lines. This process requires communication between processors along two successive rows of the mesh network of processors. A similar procedure for the processors along the columns will fix the width of each column. For a $\sqrt{p} \times \sqrt{p}$ mesh this process requires a total $O(\sqrt{p})$ communication.

5 Simulation Results

To evaluate the effectiveness of our load balancing algorithm, we ran some tests for the considered model. In these tests we examined two cases each time: the model running without the load balancing algorithm against to its periodical use for a specific time interval.The method used was the SI-EDF [17] for different mesh sizes. For the physics computations we assumed a normal distribution of loads superimposed on the given mesh for solving the NS equations. In fact, we used a scale factor M, which determines the imbalance of computational vertical load and obtained results for different values of this parameter. So, we were able to examine the behavior of our load balancing algorithm in different scenarios between the *physics* and *dynamics* calculations. Our results, summarized in Table 2, indicate that when the vertical load is small in relation to the horizontal, then load balancing should be avoided. On the other hand, as the load increases, we may reach an improvement of nearly 45% when using the SI-EDF load bal-

Table 2. Simulation Results. **L/B** =Load Balancing. Numbers in 2nd and 3nd columns are secs. The last column shows the percentage improvement.

mesh size	44 processors		
	Without L/B	**With L/B**	
	Total	*Total*	*Improvement*
M = 1			
20x20	46.932	50.551	−7.71%
40x40	149.88	117.306	21.73%
80x80	366.119	287.374	21.51%
100x100	499.268	394.327	21.02%
M = 100			
20x20	4689.879	4652.724	0.79%
40x40	15517.46	9654.508	37.78%
80x80	37866.911	20745.52	45.21%
100x100	48499.712	26434.305	45.50%
M = 10000			
20x20	489042.981	461727.89	5.59%
40x40	1499122.957	1112865.135	25.77%
80x80	3767658.518	2186112.869	41.98%
100x100	4700380.338	2834764.494	30.69%

ancing algorithm. Even though we kept the structure of the application graph unchanged with the cost of approximate balance, the gain is satisfactory.

Acknowledgement

The project is co-funded by the European Social Fund and National Resources (EPEAK II) Pythagoras, Grant No. 70/3/7418.

References

1. Boillat, J.E.: Load balancing and poisson equation in a graph. Concurrency: Practice and Experience 2, 289–313 (1990)
2. Botta, E.F., Veldman, A.E.P.: On local relaxation methods and their application to convection-diffusion equations. J. Comput. Phys. 48, 127–149 (1981)
3. Boukas, L.A., Missirlis, N.M.: The Parallel Local Modified SOR for nonsymmetric linear systems. Inter. J. Computer Math. 68, 153–174 (1998)
4. Cybenko, G.: Dynamic load balancing for distributed memory multi-processors. Journal of Parallel and Distributed Computing 7, 279–301 (1989)
5. Diekmann, R., Muthukrishnan, S., Nayakkankuppam, M.V.: Engineering diffusive load balancing algorithms using experiments. In: Lüling, R., Bilardi, G., Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1997. LNCS, vol. 1253, pp. 111–122. Springer, Heidelberg (1997)

6. Diekmann, R., Frommer, A., Monien, B.: Efficient schemes for nearest neighbour load balancing. Parallel Computing 25, 789–812 (1999)
7. Ehrlich, L.W.: An Ad-Hoc SOR Method. J. Comput. Phys. 42, 31–45 (1981)
8. Ehrlich, L.W.: The Ad-Hoc SOR method: A local relaxation scheme. In: Elliptic Problem Solvers II, pp. 257–269. Academic Press, New York (1984)
9. Elsasser, R., Monien, B., Schamberger, S., Rote, G.: Toward optimal diffusion matrices. In: International Parallel and Distributed Processing Symposium, IEEE Computer Society Press, Los Alamitos (2002)
10. Farhat, C., Maman, N., Brown, G.: Mesh partitioning for implicit computations via iterative domain decomposition: Impact and optimization of the subdomain aspect ratio. Int. J. Numer. Meth. in Eng. 38, 989–1000 (1995)
11. Hendrickson, B., Leland, R.: An improved spectral graph partitioning algorithm for mapping parallel computations. SIAM J. Sci. Comput. 16(2), 452–469 (1995)
12. Karagiorgos, G., Missirlis, N.M., Tzaferis, F.: Dynamic Load Balancing for Atmospheric Models. In: Zwieflhofer, W., Kreitz, N. (eds.) Proceedings of the Ninth ECMWF Workshop on the use of High Performance Computing in Meteorology, Developments in teracomputing, November 13-17, 2000, pp. 214–226. World Scientific, Singapore (2000)
13. Karagiorgos, G., Missirlis, N.M.: Iterative Load Balancing Schemes for Air Pollution Models in Large-Scale Scientific Computing. In: Margenov, S., Waśniewski, J., Yalamov, P. (eds.) LSSC 2001. LNCS, vol. 2179, pp. 291–298. Springer, Heidelberg (2001)
14. Karagiorgos, G., Missirlis, N.M.: Accelerated diffusion algorithms for dynamic load balancing. Inform. Proc. Letters 84, 61–67 (2002)
15. Karagiorgos, G., Missirlis, N.M.: Fourier analysis for solving the load balancing problem. Foundations of Computing and Decision Sciences 27(3) (2002)
16. Karagiorgos, G., Missirlis, N.M.: Optimal diffusion for load balancing in weighted torus (submitted)
17. Karagiorgos, G., Missirlis, N.M.: Fast diffusion load balancing algorithms on torus graphs. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, Springer, Heidelberg (2006)
18. Kuo, C.-C.J., Levy, B.C., Musicus, B.R.: A local relaxation method for solving elliptic PDE's on mesh-connected arrays. SIAM J. Sci. Statist. Comput. 8, 530–573 (1987)
19. Muthukrishnan, S., Ghosh, B., Schultz, M.H.: First and second order Diffusive methods for rapid, coarse, distributed load balancing. Theory of Comput. Systems 31, 331–354 (1998)
20. Varga, R.S.: Matrix Iterative Analysis. Prentice-Hall, Englewood Cliffs (1962)
21. Xu, C.Z., Lau, F.C.M.: Optimal parameters for load balancing the diffusion method in k-ary n-cube networks. Infor. Proc. Letters 4, 181–187 (1993)
22. Xu, C.Z., Lau, F.C.M.: Load balancing in parallel computers: Theory and Practice. Kluwer Academic Publishers, Dordrecht (1997)
23. Young, D.M.: Iterative Solution of Large Linear Systems. Academic Press, New York (1971)

A Method of Adaptive Coarsening for Compressing Scientific Datasets

Tallat M. Shafaat[1] and Scott B. Baden[2]

[1] Royal Institute of Technology (KTH),
School of Information and Communication, Stockholm, Sweden
`tallat@kth.se`
[2] Department of Computer Science and Engineering,
University of California, San Diego, La Jolla CA 92093-0404, USA
`baden@cs.ucsd.edu`
`http://www.cse.ucsd.edu/users/baden`

Abstract. We present *adaptive coarsening*, a multi-resolution lossy compression algorithm for scientific datasets. The algorithm provides guaranteed error bounds according to the user's requirements for subsequent post-processing. We demonstrate compression factors of up to an order of magnitude with datasets coming from solutions to time-dependent partial differential equations in one and two dimensions.

1 Introduction

A current challenge in large scale computing is how to compress, without losing valuable information, the prodigious output of simulations of ever increasing fidelity. Compression is traditionally applied to data visualization [11,5,9,6]. However, analysis and post-processing of scientific data often entails taking spatial derivatives, and the required tolerances usually exceed that of visualization itself. Not surprisingly, users are reluctant to compress data in the spatial domain. Instead, they often compress by sub-sampling in the time domain. While such sub-sampling introduces aliasing errors, users seem less concerned about the artifacts introduced in the temporal domain than in the spatial domain. An alternative is to use lossless compression [8,4]. But owing to the presence of noise in floating point data, compression is modest—often far less than a factor of two.

We present a lossy compression strategy called *adaptive coarsening* which coarsens data selectively and locally according to how the user intends to subsequently process the data. Our algorithm produces a compact representation providing guaranteed error bounds for the designated post-processing operations. An adaptively coarsened dataset includes geometric meta data describing the structure of the multi-resolution mesh. This information may be used to optimize subsequent data access and analysis.

We discuss preliminary results with two data sets coming from solutions to one and two dimensional partial differential equations. Using Adaptive Coarsening, we obtained compression factors of up to 11 and 15, respectively. Like wavelet compression [9], adaptive coarsening is a multi-resolution technique; however, it does not represent data progressively.

This paper makes two contributions: (1) it offers an adaptive sub-sampling procedure for scientific data represented as uniform arrays, and (2) a framework that takes into account the context in which the data is subsequently post-processed.

2 Adaptive Coarsening

Consider the numerical solution to a time-dependent partial differential equation in one dimension shown in Fig.1, which has been computed on a uniform mesh using a finite difference method. Owing to the irregularity of the solution, some portions of the mesh may be stored at a lower resolution than others without significant loss of accuracy. As a result we may approximate the uniform mesh with a mesh with variable spacing as shown in Fig.2.

Adaptive coarsening works by coarsening a mesh, re-constructing the result back to the original mesh's index domain, and coalescing (coarsening) those points where the re-constructed data approximates the original data with sufficient accuracy. This process is carried out recursively on the newly generated coarse mesh(es) until it is no longer possible to coarsen the data: either further coarsening would violate a specified error bound, or the resultant new coarse meshes fall below some minimal size threshold. As previously mentioned, the notion of "sufficient accuracy" is measured with respect to the designated post-processing operations. For example, if we are interested in accurately reconstructing second derivatives from compressed data, then we measure the accuracy in terms of the second derivative of the input, rather than the input itself.

Adaptive coarsening employs the following error estimation procedure to determine which points may be safely stored at a lower resolution. Let G^h be a set of points defined uniformly over a discrete index space, which is a set of contiguous integers in one dimension. (In general we have a regularly spaced set of points in d dimensions.) Let $u(G^h)$ or, by an abuse of notation, u^h, represent the data on G^h. Define the *sub-sampling* operator $\mathcal{R} : u^h \rightarrow u^{2h}$ which maps values from a mesh defined on G^h onto a coarsened index domain G^{2h}, which, in one dimension, has half as many points.[1] Define the *up-sampling* operator $\mathcal{P} : u^{2h} \rightarrow u^h$, that maps values from a coarse index domain G^{2h} onto the refined domain G^h.

As previously mentioned, compression will be carried out in the context of post-processing. Let Φ be the post-processing operator. We compress the data, post-process it, and then re-construct the original by up-sampling, i.e. interpolation. That is, we compute $\mathcal{P} \circ \Phi \circ \mathcal{R}\ (u^h)$. Noting that $\mathcal{P} \circ \mathcal{R}$ is not the identity operator, we define the *error* operator $\mathcal{E} = \Phi - \mathcal{P} \circ \Phi \circ \mathcal{R}$. Now, for a given relative error threshold $\epsilon << 1$, we may coarsen the mesh without a significant loss of accuracy where $|\mathcal{E}(u^h)/\Phi(u^h)| < \epsilon$, and $| \cdot |$ is the absolute value taken

[1] The coarsening factor may be greater than two.

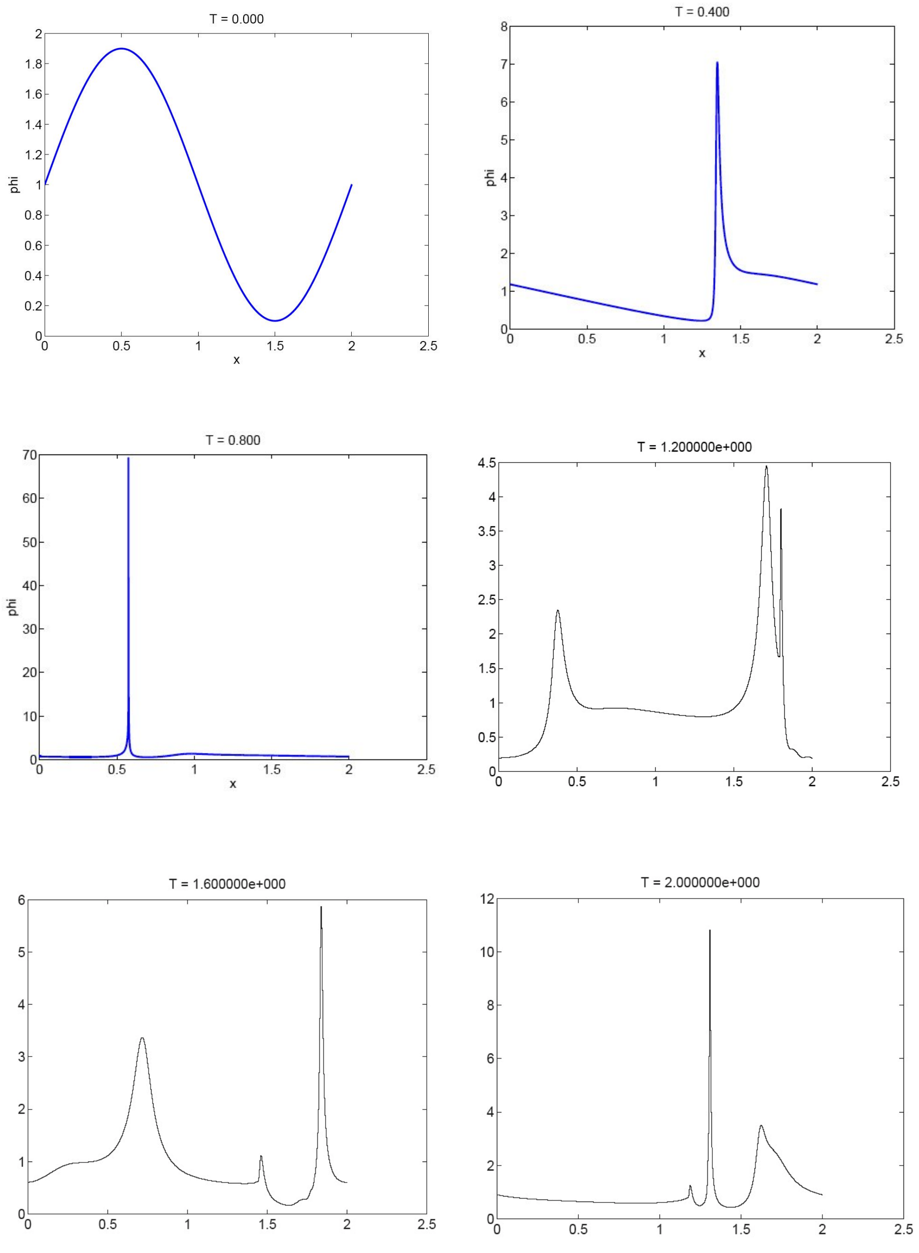


Fig. 1. The solution to a time dependent partial differential equation

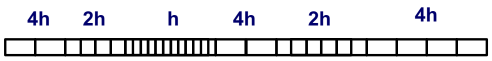


Fig. 2. An adaptively coarsened mesh

point-wise[2]. We carry out this procedure recursively on the newly coarsened portions of the mesh until it is no longer possible to coarsen the data.

The adaptive coarsening algorithm appears as pseudo code in Fig. 3. The algorithm proceeds one level at a time (line 2), starting from the initial (finest) level 0. Line (5) estimates the error and coarsens the mesh accordingly, generating a list of intervals (rectangles in 2D, etc.). Since there may be more than one subdomain at the parent's level, we augment the existing coarse grids at the current level with a union $\cup$ operation (6). We also remove from the previous finer level any points that have been coarsened, since each point may appear in at most one level.[3] We perform this operation using set difference (7).

Adaptive coarsening requires rules for coarsening and refining the data, that is, the operations $\mathcal{P}$ and $\mathcal{R}$. These rules may be defined by the user, in particular, to customize them to the data and the postprocessing operator Φ. At present we use simple sub-sampling for $\mathcal{R}$, that is, selecting every Cth point, where C is the sub-sampling rate. Our up-sampling operator $\mathcal{P}$ currently uses a piecewise cubic Hermite interpolating polynomial[4].

```
1. let level[0] = $u^h$
2. for i := 1 to maxLev
3.     let $h^i$ := discretization at level $i$
4.     foreach subdomain $u_j^{h^{i-1}}$ ∈ level[i-1]
5.         where $\mathcal{E}(u_j^{h^{i-1}}) < \epsilon$, create a new coarse domain $u^{h^i}$
6.         level[i] := level[i] ∪ $u^{h^i}$
7.         level[i-1] := level[i-1] \$\mathcal{P}u^{h^i}$
8.     end foreach
9. end for
```

Fig. 3. The adaptive coarsening algorithm. The operations \ and ∪ are set union and set difference operations.

3 Results

We present results for 1D and 2D datasets obtained by solving time-dependent partial differential equations.

[2] Clearly the significance of "loss of accuracy" depends on the choice of an appropriate definition of ϵ.

[3] By comparison, progressive methods would represent values at all levels, up to the coarsest one.

[4] Matlab `pchip`, www.mathworks.com/access/helpdesk/help/techdoc/ref/pchip.html

1D Dataset. We solved a variant of the Burger's equation on a mesh with 16,385 points over a series of 163,840 timesteps. The dataset consists of 81 snapshots spaced at equal intervals in time, including the initial data. We take the second derivative as the post-processing operation.

We applied adaptive coarsening to the 81 snapshots, using relative error thresholds of 10^{-3} and 10^{-4}. We obtained compression factors of 11.4 and 3.86, respectively.

If we *do not* take into account the context of post-postprocessing, i.e., we set Φ to the identity operation, compression increases significantly: in our case to 53.8 and 19.6, respectively. Thus, we reduce compression effectiveness when we take into account how the data will be subsequently analyzed. However, this tradeoff is essential: it gives us the assurance that we will incur only modest, i.e. acceptable, errors when we take second derivatives of the compressed data.

In order to compensate for the decreased spatial sampling rate in coarsened data, we computed the 2nd derivative post processing operation using a higher order (4th order) 5-point centered-difference formula. Compared with the 3-point 2nd order formula, the higher order formula boosts compression by about 20% [10]. The added cost of this higher order stencil is modest. While the number of floating point operations increases, the number of memory accesses stays the same, and these largely determine the cost of taking spatial derivatives. Moreover, the cost of the I/O dominates the cost of CPU processing,

Adaptive coarsening is similar to adaptive sub-sampling employed in High Definition TV signal processing [1], which splits the index domain into fixed size pieces and sub-samples each piece separately. This strategy results in a more regular structure, and lower software bookkeeping overheads, than adaptive coarsening. Owing to additional constraints on the sub-sampling process, we expect that adaptive sub-sampling will yield a lower compression factor than adaptive coarsening. Indeed, the simplicity of adaptive sub-sampling comes at a cost. Compared with adaptive coarsening, compression drops to 7.56 and 3.33, respectively. At a threshold of 10^{-3}, adaptive coarsening is about 40% more effective than adaptive sub-sampling. The added flexibility offered by adaptive coarsening fits the sub-sampling pattern more tightly to the measured error in the data, leading to a higher degree of compression.

2D Data Set. The 2D data set was obtained by solving the Navier Stokes equations.[5] There were 51 snapshots, each comprising an array of 1020×1020 floating point numbers. These data carry a differentiated quantity (vorticity), and hence have already been post-processed. We have preliminary results for just one threshold: 10^{-3}. We applied adaptive sub-sampling in two dimensions, obtaining a compression factor of 14.9.

We then selected the parts of the mesh that could not be compressed at all, unraveling them in row major order into 1D arrays and applying 1D adaptive coarsening to the result. Because the data are smoother in the X direction than the Y, this optimization increased compression by about 10% to 16.4. However,

[5] `http://www.math.ucla.edu/~anderson/270e.1.04f/Assignment8/Assign8.html`.

knowing which direction to compress along requires some user intervention. We are currently extending adaptive coarsening to multiple dimensions, and pursuing automated strategies for anisotropic compression.

4 Conclusions and Future Work

We have demonstrated a multi-resolution compression technique called adaptive coarsening. Our approach is based on the philosophy that technological factors present an opportunity to employ plentiful processing cycles to reduce the space required to store massive data sets. We have been able to achieve an order of magnitude in compression for uniform mesh data sets. The effect is to significantly reducing the time to transfer data from off-line storage, and to process the data in subsequent analysis. Moreover, this level of compression was obtained while retaining engineering precision. Compression drops rapidly as the number of digits increases. To this end we are investigating more sophisticated sampling and interpolation procedures. Our ultimate target is three dimensional data, and work is currently in progress.

Although Adaptive Coarsening was applied to data arranged on a regular array of points, the technique is applicable to irregular data sets, e.g. arising in finite element methods, so long as there exist appropriate sub-sampling and up-sampling procedures [7].

Adaptive coarsening parallelizes readily. Sub-sampling and up-sampling operations require nearest neighbor communication only. Some collective communication is required to ensure that the irregular mesh structure makes balanced use of parallel I/O, but the cost is modest compared with that of the I/O.

Multiresolution compression techniques have been employed for several years in computer graphics, signal processing, and efficient mesh generation [11,5,9,6,9]. A complete list of references is too lengthly to include here. Hierarchical triangulations have been applied to the visualization of geophysical data [6]. Adaptive coarsening is similar in spirit to this technique, though the goal is to conserve space rather than bandwidth. By conserving space, adaptive coarsening also conserves bandwidth.

Our approach permits the direct manipulation of the compressed data sets; that is, without the need to reconstruct the original data. This capability is desirable because it enables data analysis to proceed within the reduced footprint of the compressed data and promotes the reuse of existing analysis code bases, simplifying post-processing application development. A framework may be constructed to apply the analysis code to each mesh component. This frees the user from having to manage the details. In general, the user needs to provide application dependent coarsening and refinement modules along with appropriate metrics for estimating the error. While such operations are often application-dependent, strategies based on Richardsonian extrapolation, for example, are robust in Structured Adaptive Mesh Refinement [3,2]. Thus, there is hope that they may prove equally useful in adaptive coarsening. These issues await further study.

Acknowledgments

This research was carried out while the second author was on sabbatical leave at Kungliga Tekniska Högskolan (KTH) in Stockholm, Sweden. The authors gratefully acknowledge the assistance of Bjorn Sjögren (KTH) and Chris Anderson (UCLA) for providing code and advice for the 1D and 2D PDE solvers, respectively. We are also grateful to Jesper Oppelstrup for many lively discussions. Scott Baden dedicates his portion of the work done in this paper to the memory of Olav Beckmann (1972-2006). This work was supported in part by NSF contract ACI9619020 (National Partnership for Advanced Computational Infrastructure (NPACI)) and the University of California, San Diego.

References

1. Belfor, R.A.F., Hesp, M.P.A., Lagendijk, R.L., Biemond, J.: Spatially adaptive subsampling of image sequences. IEEE Trans. Image Proc. 3(5), 492–500 (1994)
2. Berger, M.J., Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. J. Comput. Phys. 82(1), 64–84 (1989)
3. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. J. Comput. Phys. 53(3), 484–512 (1984)
4. Engelson, E., Fritzson, D., Fritzson, P.: Lossless compression of high-volume numerical data from simulations. In: Proc. Data Compression Conf, pp. 574–586 (2000)
5. Freitag, L.A., Loy, R.M.: Adaptive, multiresolution visualization of large data sets using a distributed memory octree. In: Proc. SC 99 (November 1999)
6. Gerstner, T.: Multiresolution visualization and compression of global topographic data. GeoInformatica (2001)
7. Gregorski, B.F., Sigeti, D.E., Ambrosiano, J.J., Graham, G., Wolinsky, M., Duchaineau, M.A., Hamann, B., Joy, K.I.: Multiresolution representation of datasets with material interfaces. In: Farin, G., Hamann, B., Hagen, H. (eds.) Hierachical and Geometrical Methods in Scientific Visualization, pp. 99–117. Springer, Heidelberg (2003)
8. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast lossless compression of scientific floating-point data. In: Proc. Data Compression Conf. (2006)
9. Roerdink, J., Westenberg, M.: Wavelet-based volume visualization. Technical Report 98-9-06, Univeristy of Groningen (1998)
10. Shafaat, T.M.: Context dependent compression using adaptive subsampling of scientific data. M.S Thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden (2006)
11. Wilhelms, J., Van Gelder, A.: Octrees for faster isosurface generation. ACM Trans. Graph. 11(3), 201–227 (1992)

A Computational Framework for Topological Operations

Michael Spevak, René Heinzl, Philipp Schwaha, and Siegfried Selberherr

Institute for Microelectronics, TU Wien,
Gusshausstrasse 27-29, 1040 Wien, Austria
spevak@iue.tuwien.ac.at

Abstract. We present a complete topological framework that is able to provide incidence traversal operations for various topological elements. This enables one to perform the necessary topological operations for several discretization schemes. A combination of incidence information combined with an archetype concept enables one to optimize traversal operations of inter-dimensional objects without explicitly storing them. Access to topological structures is provided using a generalized iterator concept.

1 Introduction

The field of scientific computing often imposes highly complex formulae with quantities on different topological elements. For example, some discretization schemes require the scalar solution to reside on vertices, while the projections of the vector-valued fluxes are stored on edges. Many applications require such a discretization of partial differential equations (PDE) as well as interpolation mechanisms and thus strongly depend on the base traversal mechanisms provided by the environment. It is quite common for a discretization scheme to require quantities originally associated with a vertex on an edge and vice versa. The projections of fluxes on edges also need to be assembled to truly vector-valued quantities associated with a vertex and an edge. In order to accomplish this, the required information has to be collected by traversing the local neighborhood of a vertex or an edge. To this date, data structures and algorithms are implemented in a heavily application and discretization scheme specific way, making their reuse practically impossible.

We present a set of base traversal operations that is sufficient for many applications. This approach results in a rigorous implementation of topological structures, which covers all types of topological elements such as vertices, cells and general inter-dimensional elements called faces. The expressiveness of source code is increased, because we do not need to explicitly write traversal algorithms for each of the elements, such as edge-cell traversal, because this information can be derived from a subset of highly optimized operations automatically.

The iterator concept allows to formulate algorithms based on this interface independently of the actual implementation of the topological data structure including dimension and archetype. The consequent use of this interface leads to dimensionally and topologically independent formulations of algorithms, e.g. a finite volume discretization scheme can be formulated independently of the type of cell complex and the dimension.

2 Motivation and Related Work

The motivation for developing a topological framework is derived from the need for flexibility in high performance applications in the field of scientific computing, especially in Technology Computer-Aided Design (TCAD). With a growing number of different simulation tools [1,2,3,4,5] and various requirements on the underlying data structures the question arises which part of an application can be re-used, if it is properly implemented. The main aim of our work is to provide a library that offers the functionality common to many different kinds of simulation tools in the field of scientific computing. This is especially true for functions which are evaluated on a topological cell complex.

In the last decade many approaches towards implementing a general purpose simulation environment for the solution of partial differential equations have been taken. Most of the tools resulting from these attempts use topological structures which are specialized to a particular discretization scheme. This reduces resource use, but it comes at the cost of greatly diminishing the flexibility of topological traversal. As an example, the finite volume method does not require vertex-faces traversal. However, for some reasons it might be advantageous to implement discretization equations based on a mixed finite element/finite volume scheme which requires such traversal operations.

The major step towards a more flexible use of topological structures is presented in [6]. The grid algorithm library (GrAL) introduces the first generalized iterator concept [7] based on multi-dimensional data structures.

Most of the other environments completely veil the topological information by formalisms such as element matrices [8] and control functions [9]. Some commercial simulation tools, such as FEMLab, accept the input in the form of a final PDE. For this reason calculations which use non-standard traversal mechanisms are cumbersome or impossible to specify.

3 Framework and Interfaces

The main aim of the topological container interface is to provide mechanisms for construction, modification and traversal. The most important conceptual requirement for the topological data structure is the retrieval of incidence information. We define the incidence relation in the following manner:

$$\mathrm{inc}(a, b) \leftrightarrow a \subset b \vee a \supset b \tag{1}$$

where a and b denote different topological elements. In the following table (Fig. 1) we list all different methods of incidence which are possible between topological elements of different dimension. It can be seen easily that the incidence relation of elements of the same dimension can be modeled by the equality relation. The first row shows all edges, faces as well as cells which are incident with the same base vertex. The first column shows vertices which are incident with one base edge, faces or cell.

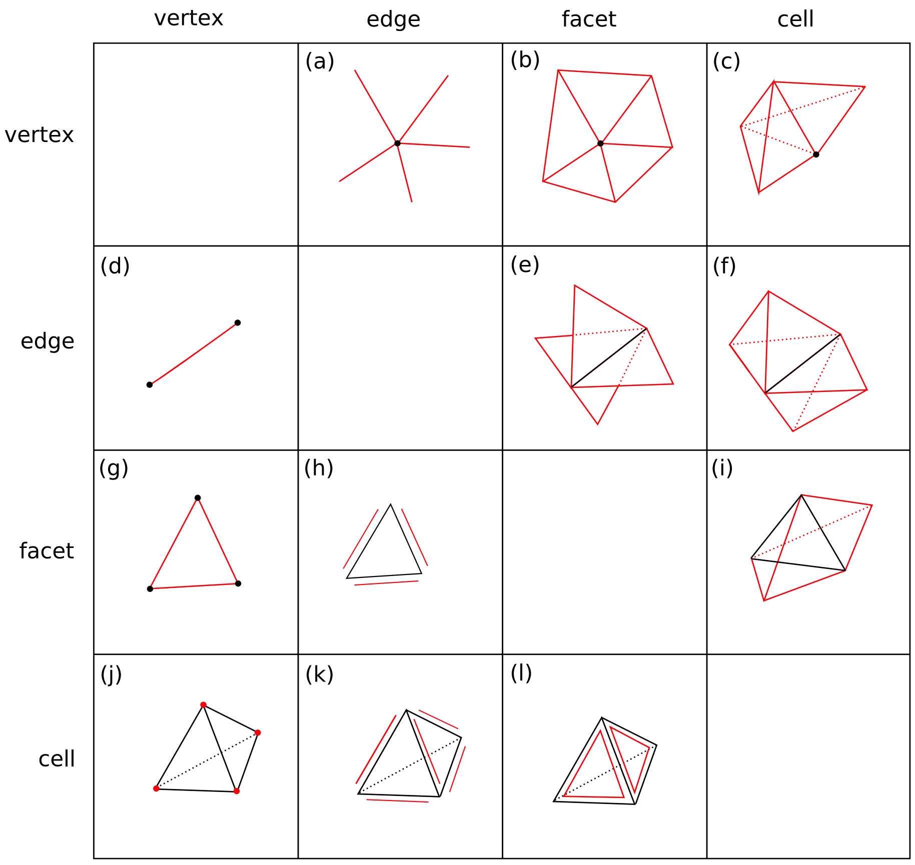


Fig. 1. Traversal methods induced by the incidence relation. horizontal: traversal schemes of the same base element. vertical: traversal scheme of the same traversal elements.

3.1 Topological Container Concept

The topological container covers the basic information of the topological cell complex. Our concept provides a subset of the required methods (Fig. 1) from which all further information can be obtained. According to the concepts of the standard template library (STL [10]), we provide iterators for the cells as well as the vertices of the cell complex. In analogy to the STL we use a formulation with `begin()` and `end()`. In order to obtain the base traversal mechanisms between

Name	Description	Requirements
vertex_iterator	iterator over vertices	iterator concept
vertex_begin(), vertex_end()	iterator range	const
cell_iterator	iterator over cells	iterator concept
cell_begin(), cell_end()	iterator range	const
cell_vertex	local traversal iterator	constructable with cell
vertex_cell	local traversal iterator	constructable with vertex

Fig. 2. Concepts for the topological base structure

vertices and cells we define traversal iterators. These traversal iterators perform the operations shown in Fig. 1, (c) and (j).

Due to the concept definition (Fig. 2) of the cell complex container we can formulate algorithms conveniently. As an example we present an algorithm which traverses all vertices as well as all cells.

Global Traversal

```
cell_complex_t cc;
cell_complex_t::vertex_iterator v_it;
for(v_it = cc.vertex_begin();
    v_it != cc.vertex_end(); ++v_it)
{
  // do something on vertices
}

cell_complex_t::cell_iterator c_it;
for(c_it = cc.cell_begin();
    c_it != cc.cell_end(); ++c_it)
{
  // do something on cells
}
```

A direct consequence of the use of the iterator ranges `vertex_begin()`, `vertex_end()` is that standard algorithms such as `for_each` are automatically supported.

3.2 Topological Elements and Handles

The data structures for single topological elements is kept to a bare minimum. In general, each topological data structure covers a so called handle in order to be distinguishable from other topological elements. Basically any type can be used for these handles, which allows to uniquely identify the element within all elements of the same dimension. The value of such handles itself does not have any semantic meaning apart from being equal. The only valid operation on handles is, therefore, the equality relation.

For inter-dimensional topological elements a unique identification can be found either via storing all the vertices or storing a cell and a local index which determines the element within the cell. We discuss vertex based indexing in the following. A handle of an inter-dimensional element is uniquely constructed by the vertex handles, e.g. $h_F = h_{v_1} + n \cdot h_{v_2}$ where the following abbreviations are used:

– h_F: the handle used by a faces
– h_{v_i}: the handle from each vertex
– n: the number of vertices

For this reason the reconstruction of vertex information from the handle is straightforward. This covers incidence operations (Fig. 1 (d) and (g)).

3.3 Archetypes

As we only store incidence information between cells and vertices (Fig. 1 (c) and (j)) the incidence relation between inter-dimensional elements is still undefined. There are many methods to specify this kind of information, for instance to use containers and explicitly storing this information. Even though this is possible and can be used in cases where high performance is required, such methods result in a high memory consumption, because most of the information has to be duplicated.

In the following we take advantage of the fact that all elements within the container have the same shape (e.g. tetrahedral elements). If this is not the case (e.g. we have a small number of elements of different shape), we have to perform a dispatch operation in order to obtain the correct shape.

The concept that provides the internal structure of the cells within the cell-complex is called an `archetype`. The archetype [6] introduces local inter-dimensional elements within a cell. A topological 2-simplex, for example, consists of three vertices and three 1-faces (edges). The archetype can be shown either as simple graph or Hasse diagram (Fig. 3).

Using the archetype concept as well as the vertex on cell relation (Fig. 1 (j)) we can derive further traversal mechanisms (Fig. 1 (k) and (l)) as each cell is aware of its covered vertices.

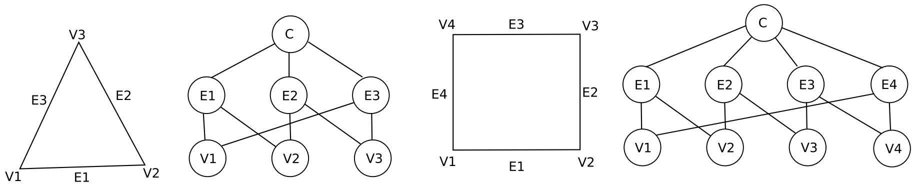


Fig. 3. The graph as well as the Hasse Diagram of the 2-simplex as well as a 2-cuboid archetype

3.4 Traversal Operations

Once the archetype as well as the vertex-cell incidence information is available, all incidence relations between arbitrary elements of the cell complex can be calculated. The topological framework stores vertex-cell incidence relations within the container which can be seen in Fig. 1 (c) and (j).

We present an example using the simplex archetype to introduce the computational operations that complete the traversal operations. All other relations ((a,b)(d,e,f,)(g,h,i)(k,l)) in Fig. 1 are taken care of by the framework. From topological handles of inter-dimensional elements *vertex on element* iteration can be obtained (Fig. 1 (d) and (g)). Due to the archetype information we obtain the element on cell traversal (Fig. 1 (k) and (l)). These base relations are sufficient for the construction of all other element-element incidence relations. The following algorithm gives a generic implementation for obtaining the traversal information via the base operations.

Incidence traversal

```
get all vertices which belong to the element (d) (g) (j)
get all cells which belong to these vertices (c)
get all n-dimensional sub-elements on those cells (j) (k) (l)
select the incident elements (by vertex-comparison)
```

We show the application of this algorithm using the example of faces on edge traversal (Fig. 4, Fig. 1 (e)). From the initial edge we obtain the vertices which are located on the edge using the basic traversal mechanism (d). Using the vertex-cell (c) information we obtain all cells which cover the vertices. We obtain all faces which are on these cells by method (k). From this set we select only the faces which are incident with the initial edge.

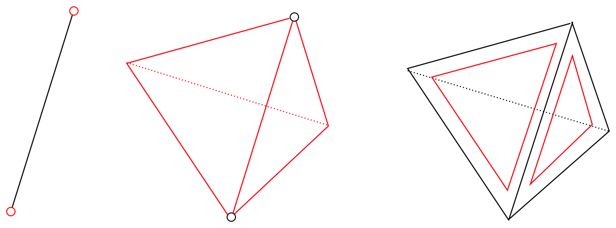


Fig. 4. Construction of the faces on edge traversal set. (left) The initial edge. (middle) The vertices on the initial edge. (right) Incident faces.

As all of these operations are local, we do not need to iterate all elements of a cell complex in order to perform an incidence traversal operation. This construction method is superior to an explicit search of topological elements within the complete cell complex for large meshes.

3.5 Generalized Iterator Concept

In order to allow an arbitrary number of nested traversal operations, which is often required in applications, it is necessary to have appropriate data structures for traversal in order to keep the program code as concise as possible. For this reason we use the random access iterator concept in order to provide access to the topological elements. We refine the iterator concept [7] for data structural convenience (Fig. 5). In contrast to the STL iterator concept, dereferentiation of this generalized iterator does not provide the data content but a handle, that is used as a key to access a property map or quantity [11] in order to obtain the contained data. Our iterator concept is a refinement of the random access iterator concept and introduces the following additional concepts.

Name	Description	Requirements
bool valid()	validity of iterator (not end)	const
iterator end()	past end of iterator validity	const
void reset()	set to the start point	const

Fig. 5. Concept refinement for the generalized iterator within the topological framework

To demonstrate the application of the `valid()` predicate we show a simple example of two nested iterations. We traverse from a base vertex to all incident edges and from these edges to incident vertices. (Fig. 1, (a) and (d))

The validity concept and its application

```
cell_vertex voc_it(cell);
while(voc_it.valid())
{
  vertex_edge eov_it(*voc_it);

  while(eov_it.valid())
  {
    //operations on edges
    ++eov_it;
  }
  ++voc_it;
}
```

In the first traversal state, a `cell` is used to perform an iteration over all its incident vertices. Each incident vertex `*voc_it` is available in the outer loop. The inner traversal loop is initiated using the actually traversed element of the outer loop. From this vertex we obtain all incident edges using the dereferentiation of the inner iterator.

In order to use standard algorithms of the STL, a data structure has to provide an initial as well as a terminal element. For this reason the `end()` function can be employed.

The iterator end functionality and its application

```
struct func_obj{
void operator()(cell_t cell) { cout << c << endl;}
};

cell_vertex voc_it(cell);
for_each(voc_it , voc_it.end(), func_obj());
```

As many of the iterators require a non-negligible time for construction, it is often more efficient to use an iterator twice rather than creating a second instance for the same purpose. For this reason we introduce the `reset` function that enables us to reset an iterator to the beginning of the iteration range.

Usage of the reset function

```
cell_vertex voc_it(cell);
for(; voc_it.valid(); ++voc_it)
{ // ... some operation
}
voc_it.reset();
for(; voc_it != voc_it.end(); ++voc_it)
{  // ... some operation
}
```

4 Generalized Data Access

The topological data structure itself, however, is not sufficient to perform complex calculations on such a structure. For this reason we provide means for the storage of values on the topological container. Each of the topological elements such as vertices or edges can be associated with one or more values.

In our approach, the *property map* concept [7] is adopted, which offers the possibility of accessing the quantities in a functional way by a mechanism called *quantity accessor*. The quantity accessor implementation also takes care of accessing quantities with different data locality, e.g., quantities on vertices, edges, faces, or cells. The quantity accessor is initialized with a domain. During initialization, the quantity accessor `quan` is bound to a specific domain with its quantity key. The `operator()` is evaluated with a vertex of the cell complex as argument and returns a reference to the stored value.

Quantity assignment

```
string key_quan  = "user_quantity";
quan_t quan = scalar_quan(domain, key_quan);

quan(vertex) = 1.0;
```

In the following code snippet a simple example of the generic use of this accessor is given, where a scalar value is assigned to each vertex in a domain. The quantity accessor creates an assignment which is passed to the `std::for_each` algorithm.

Quantity assignment

```
string key_quan = "user_quantity";
quan_t quan = scalar_quan(domain, key_quan);

for_each(cc.vertex_begin(), cc.vertex_end(), quan = 1.0);
```

5 Conclusion

A computational mechanism was introduced for completing traversal operations obtained from the archetype's structure and a minimum of explicitely stored information. As a result we presented a framework that provides a complete means of topological traversal operations based the concept of archetypes.

Our framework presented here integrates well into existing software components such as the STL.

The use of clean, well defined iterator interfaces alongside generalized standard routines makes it possible to develop orthogonal and modular software. Futhermore the provided means are not only sufficient to build a homoegenous interface but can also be used to implement several discretization schemes.

References

1. Selberherr, S., Schütz, A., Pötzl, H.: MINIMOS—A Two-Dimensional MOS Transistor Analyzer. IEEE Trans. Electron Dev. ED-27(8), 1540–1550 (1980)
2. Halama, S., Pichler, C., Rieger, G., Schrom, G., Simlinger, T., Selberherr, S.: VISTA — User Interface, Task Level, and Tool Integration. IEEE J. Techn. Comp. Aided Design 14(10), 1208–1222 (1995)
3. Sabelka, R., Selberherr, S.: A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. Microelectronics Journal 32(2), 163–171 (2001)
4. IμE: MINIMOS-NT 2.1 User's Guide. Institut für Mikroelektronik, Technische Universität Wien, Austria (2004), `http://www.iue.tuwien.ac.at/software/minimos-nt`
5. Binder, T., Hössinger, A., Selberherr, S.: Rigorous Integration of Semiconductor Process and Device Simulators. IEEE Trans. Comp.-Aided Design of Int. Circ. and Systems 22(9), 1204–1214 (2003)
6. Berti, G.: GrAL - The Grid Algorithms Library. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) Computational Science - ICCS 2002. LNCS, vol. 2331, pp. 745–754. Springer, Heidelberg (2002)
7. Abrahams, D., Siek, J., Witt, T.: New Iterator Concepts. Technical Report N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003)
8. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II – A General Purpose Object Oriented Finite Element Library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University (2006)

9. Rafferty, C.S., Smith, R.K.: Solving Partial Differential Equations with the Prophet Simulator (1996)
10. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
11. Heinzl, R., Spevak, M., Schwaha, P., Grasser, T.: Concepts for High Performance Generic Scientific Computing. In: Proc. of the 5th MATHMOD, vol. 1, Vienna, Austria (2006)

Applications of Grid Computing in Genetics and Proteomics

Jorge Andrade[1], Malin Andersen[1,2], Lisa Berglund[1], and Jacob Odeberg[1,2]

[1] Department of Biotechnology, Royal Institute of Technology (KTH), AlbaNova University Center, SE-106 91 Stockholm, Sweden
{jorge, jacob, malina}@biotech.kth.se, lisaber@kth.se
http://www.biotech.kth.se

[2] Department of Medicine, Atherosclerosis Research Unit, King Gustaf V Research Institute, Karolinska Institutet, Karolinska University Hospital, Stockholm, Sweden

Abstract. The potential for Grid technologies in applied bioinformatics is largely unexplored. We have developed a model for solving computationally demanding bioinformatics tasks in distributed Grid environments, designed to ease the usability for scientists unfamiliar with Grid computing. With a script-based implementation that uses a strategy of temporary installations of databases and existing executables on remote nodes at submission, we propose a generic solution that do not rely on predefined Grid runtime environments and that can easily be adapted to other bioinformatics tasks suitable for parallelization. This implementation has been successfully applied to whole proteome sequence similarity analyses and to genome-wide genotype simulations, where computation time was reduced from years to weeks. We conclude that computational Grid technology is a useful resource for solving high compute tasks in genetics and proteomics using existing algorithms.

1 Introduction

Bioinformatics is a relatively new field of biological research involving the integration of computers, software tools, and databases in an effort to address biological questions. Areas include human genome research, simulations of biological and biochemical processes, and proteomics (for example protein folding simulations). With an increasing amount and complexity of data in genomics and genetics generated by today's high-throughput screening technologies and the development of advanced algorithms for mining complex data, computational power now sometimes defines the practical limit. High performance computing or alternative solutions are required to undertake the intensive data processing and analysis. Grid computing [1], offers a model for solving massive computational problems by subdividing the computation in a set of small jobs, executed in parallel on geographically distributed resources.

However, the current job management process on Grid environments is relatively complex and non-automated. Biologists who want to take advantage of

Grid resources face a process of having to manually submit their jobs, periodically check the resource broker for the status of the jobs ("Submitted", "Ready", "Scheduled", "Running", or "Finished" status), and finally get the results with a raw file transfer from the remote storage area or remote worker to the local file system of their user interface. Different solutions for increasing the usability, scalability and stability in computational Grids have recently been proposed [2], [3].

The presented implementation represents a model by which access and utilization of Grid resources is greatly facilitated, allowing biologist and other non-Grid-experts to exploit the Grid power without necessarily having knowledge of Grid related details and procedures. The utility of this implementation is demonstrated by application to two computationally expensive bioinformatics tasks: Whole proteome sequence similarity analysis and genotype simulations for genome wide linkage analysis

2 Methods

In order to make the interaction with the complex computational environments on Grids more straightforward to the biologically oriented scientists, the following tasks were automated: Proxy setup handles the user authentication as a member of a Virtual Organization (VO) and grants the user access to the Grid resources. By default, twelve hours is the time for the proxy to be in effect. After the proxy expires, the task of re-creating new proxy is automatically scheduled in the local Grid client. Job submission involves the remote distribution of the split input data files or databases, as well as the executable binary files to the Grid workers. For each Grid job submitted, a Grid job specification is created using the Resource Specification Language (RSL). Processing. After job submission, a local temporary installation of datasets and executables in the allocated remote nodes is performed. After that, parallel execution is started in remote nodes, and a constant monitoring of the current job's status is performed. Job re-submission in case of job failure or excessive delay in Grid queue systems is also handled. Job collection. When specific Grid jobs are finished, partial results are downloaded from the remote Grid workers to the local computer. This module is also able to handle parallel retrieval of several finished jobs. The figure 1 shows a graphical description of the Grid framework configuration used for this implementation.

3 Implementation

A Perl script based Grid broker that ensure unique user authentication was implemented, allowing the user to remotely deploy and execute pre-existing algorithms or software across available Grid resources at submission time. The presented solution is adjusted to NorduGrid ARC [4], but can be easily adapted to any Globus based Grid middleware.

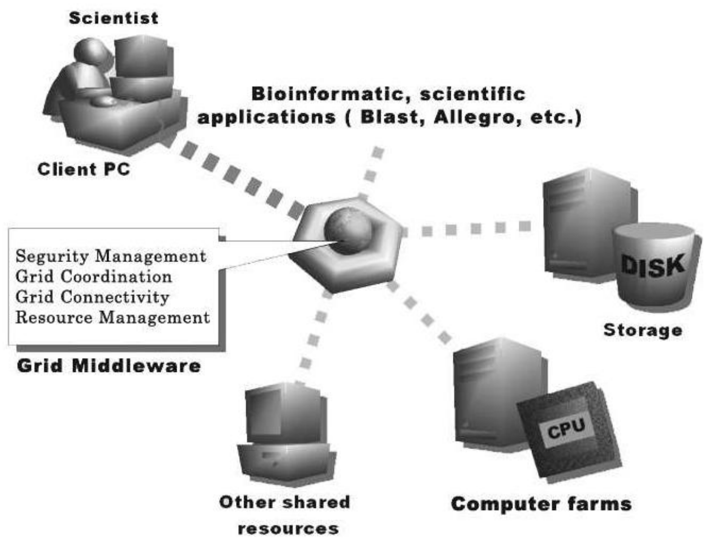


Fig. 1. Grid computing Framework for application in Bioinformatics

This implementation can be adapted to tasks suitable for parallelization where an existing Linux executable exists. The implementation consists of two Perl scripts:

gridjobsetup.pl. Manages two main tasks. Firstly, the "big" computationally expensive task is partitioned into a user-selected number of smaller equally sized atomistic jobs, each corresponding to a fraction of the total data. Secondly, for each datra fraction, a Grid job specification is created using the resource specification language (RSL).

gridbroker.pl. This is the Grid broker. Its function is to manage the submission, monitoring and collection of the Grid jobs. Following node allocation and job submission, gridbroker.pl performs temporary installations of the deployed executable on the Grid nodes/remote workers, and parallel execution of the Grid jobs is started. gridbroker.pl constantly monitors the parallel execution of the distributed tasks, and in the case of job failure or if a job or set of jobs are excessively delayed in the work-queue scheduler, gridbroker.pl manages the re-submission of this job or set of jobs to different available Grid workers. When jobs reach the status of "finished", forked download of specific job-results to the user local file system is performed. The partial Grid job results are finally concatenated to generate the output file. A fraction of the Perl implementation of the broker is shown below. The code shows a loop that manages the submission of a user defined number of Grid jobs; a vector of Grid job identifiers is created in memory and in an archive. This vector will then be used to mange the monitoring and downloading of the jobs. A log file that registers submission start and finish times is also created.

Fraction of the Algorithm that Manage the Submission of Grid Jobs

```
Input: XRSL-specification(s) of a number of Grid jobs;
for each Grid job,a set of specific input parameters.
Action: Submit the given number of Grid jobs.
Output: Vector of Job's id and file with timings.
1. Process XRSL-specification
2. Create a time-log-file and register the start of
   submission
3. Create and open a job-id-file
4. For each job
   (a) Select the cluster(s) to which the job will be
       Submitted
   (b)Submit the job
   (c)Collect the retrieved job-id
   (d)Push the collected job-id in a vector
   (e)Push the collected job-id in a job-id-file
5. Register in time-log-file the end of submission
6. Close time-log-file
7. Close job-id-file
```

Fraction of Algorithm that Manage the Monitoring and Downloading of Finished Grid Jobs

(The following algorithm shows the constantly monitoring of job's status using the previously created vector of jobs identifiers; in case of job "failure", re-submission of jobs is performed, jobs that have successfully reached the status of "finished" are downloaded.)

```
Input: job-id vector and job-id-file.
Action: Monitoring and collection of Grid jobs and resubmission if
"job-failure".
Output: Collection of finished Grid Jobs and time-log-file.

1. While number of downloaded jobs <= number of total
   Grid jobs submitted
2. For each job:
   (a)Monitoring status of vector job-id[i]
   (b)If status of job-id[i] is "FAILURE" then:
      i.   Re-submit job- id[i] to available Grid cluster
      ii.  Delete old and push new retrieved job-id
      iii. Delete old and push new job-id in job-id File
      iv.  Register re-submission time in the log-file
   (c)If satus of job-id[i] is "FINISHED" then:
i.  Collect job-id[i] and register time
ii. Push job-id[i] from vector of Job’s id
iii.    Push job-id[i] from file of Job’s id
iv. Increase the counter of downloaded jobs
3. Register end of job-collection and close log-file
```

4 Results

XWe have aimed to develop a generic Grid implementation for solving bioinformatics tasks suitable for parallelization where neither pre-selection of available Grid nodes nor pre-installation of software or databases will be necessary. Existing Linux-based executables can be used when scaling up tasks prohibitively time-consuming to perform in single work stations, as our solution will not require re-codification or programming modifications. The implementation is also applicable in situations where the source code is not available. To streamline the process we chose the strategy of making temporary installations of the executable and databases locally at each remote node at submission, followed by un-installation after download and collection of the results. By avoiding the need of predefined run-time environments, this implementation limits the interaction with Grid administrators for installation of applications/software and updates, thereby accommodating for dynamic Grid environments in which available nodes change between submissions. This strategy is however not applicable for instance in cases when a database management system (DBMS) is required, typical examples of DBMSs like Oracle, Microsoft SQL Server or MySQL, will necessary need the use of a specific run-time environments.

Our implementation was evaluated in two highly computer intensive real applications in proteomics and genetics:

The first application deals with whole proteome protein similarity analysis using a sliding window algorithm [5]. In contrast to ordinary blastp queries aligning full length query protein sequences, the sliding window approach results in a significantly higher number of blast searchers. Using a sliding window size of 51 amino acids, the number of blastp searches for a 1000 amino acid protein increase from 1 to 950. For the entire human Ensembl database [http://www.ensembl.org] of close to 34,000 human proteins, this corresponds to about 15,000,000 blastp searches. The time needed to run this number of blastp searches on a single computer was about eight weeks. As the Ensembl database is constantly evolving and being updated, where protein sequences are added, changed or deleted, frequent reprocessing of the database becomes necessary in the HPA program [http://www.proteinatlas.org] in order to work with the most accurate data at any one time. Once a new version of the database is released, the sequence similarity data on which the epitope design is based needs also to be updated. The computational requirements for this task were exceeding in-house resources if the processed results of a database update were to be delivered before it was already obsolete. With a Grid implementation where local installations of both the blastp executable and the entire Ensembl database was performed on each node (a total package of a size of 16 MB)[5], runtime was reduced from about eight weeks on one single up-to-date computer, to less than 24 hours using 300 Grid nodes in Swegrid [http://www.swegrid.se]. The absolute speed-up for this application was calculated as:

$$Sp = \frac{T_1^s}{T_p} \tag{1}$$

Where T_1^s is the sequential run-time, and T_p is the execution time in p Grid nodes. Using the complete human Ensembl database as input, speed-up of 56 fold was archived, this was calculated by dividing $T_1^s = 1344$ hours by $T_p = 24$ hours (the Grid run-time with same data as input in 300 Grid processors in Swegrid). The expected linear speed-up (300 fold in 300 nodes) was not archived, mainly due to Grid latency. By making a local installation of a database at each submission, the speed of running queries against a local database was obtained together with running against the most recent update. The alternative strategy of storing the database in one single "Grid storage resource" accessed by all the other nodes, proved to create an I/O overload in the Grid storage server, resulting in a significant increase of the total runtime.

The second application was facilitating computer simulations of genotypes using a HMM based software [6], in order to evaluate the significance of genome-wide linkage data. This was applied in a study aimed to identify novel genes involved in the pathogenesis of Alzheimers disease (AD) by performing a non-parametric multipoint linkage analysis on AD families from the relatively genetically homogeneous Swedish population. On a genome-wide scale, this task is extremely computationally intensive. In the absence of sufficient computational resources the number of simulations would therefore have to be limited, which could lead to the estimation of insufficient global significance levels and false positive linkage claims. We developed Grid-Allegro [7] which was used in the hypothesis testing to evaluate the statistical significance of the linkage data under the null hypothesis of no linkage using a set of 109 AD families. Serial execution time required to perform the minimum required 22000 genotype simulation analyses was reduced from the projected time, more that 3 years on a single up-to-date CPU, to less than 3 days when distributed computing was performed in 600 Grid workers in Swegrid [7].

5 Discussion

There are several computationally demanding algorithms and tasks in bioinformatics that may cause a computational overload when scaled up. To the researcher without access to expensive resources in-house such as dedicated clusters or computer farms, Grids represents a cost-effective and powerful resource. However, a current obstacle especially to the biologically oriented researcher is managing the middleware that is still raw and hardly accessible. For the non-computer scientist, more user-friendly alternative solutions are necessary. One alternative is to develop web-based user front-end services of underlying Grid implementations, which are accessed by third party users. This is the most accessible alternative of exploiting Grid resources, as it is associated with minimal complexity where no necessary previous knowledge of distributed computing is required by the user. Grid resource brokers and job submission services based on Grid and Web services have been previously proposed [8]. However, for our specific purposes, we decided to use a generic, script-based strategy for implementing Grid-aware applications of bioinformatics task that are suitable for parallelisation. Our major concerns were related with security, stability and usability. Although Grid security is based in public key infrastructure (PKI) and this architecture offers strong security levels for the Grid end-user, current PKI implementations suffer from serious usability issues, especially when applied to web-based Grid-services. [9] Strong efforts are required in searching for new mechanisms for increasing the usability of Grid security. [10]

Web-based implementations also confine the input submission format to those defined or envisioned by the provider/developer, which may reduce the flexibility for the third party user. Furthermore, Web-based Grid implementations may require re-codification of previously existing single CPU-oriented algorithm implementations. The developer assumes the administrator responsibility for maintaining the availability and updating of the resource. When web-based services are developed and provided through large initiatives [11], this indeed represents a transparent and user-friendly solution. However, new applications depend on continued development and implementation by these providers, and are hence not always available to meet the specific needs in individual third party projects. The alternative generic strategy, although requiring basic computer knowledge by the user, greatly increases the flexibility by enabling the implementation to be applied to similar distributable computation-demanding tasks.

In conclusion, our implementation facilitates the biologically oriented scientist's remote deployment and execution of pre-existing codifications of bioinformatics algorithms across multiple Grid resources. By applying this implementation in solving two data and CPU intensive tasks, we have demonstrated the potential utility of Grid technology for addressing highly computational demanding bioinformatics task.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. International Journal of High Performance Computing Applications 15(3), 200–222 (2001)
2. Ellert, M., Konstantinov, B., K'onya, J., Lindemann, J., Livenson, I., Nielsen, J., Smirnova, O., Wäänänen, A.: Advanced Resource Connector middleware for lightweight computational Grids. Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications 23, 219–240 (2007)

3. Elmroth, E., Tordsson, J.: Grid Resource Brokering Algorithms Enabling Advance Reservations and Resource Selection Based on Performance Predictions. Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications (2007)
4. Ellert, M., et al.: The NorduGrid project: using Globus toolkit for building GRID infrastructure. Nuclear Instruments & Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment 502(2-3), 407–410 (2003)
5. Andrade, J., et al.: Using Grid technology for computationally intensive applied bioinformatics analyses. Silico Biology, 6 (2006)
6. Gudbjartsson, D.F., et al.: Allegro, a new computer program for multipoint linkage analysis. Nat Genet 25(1), 12–13 (2000)
7. Andrade, J., et al.: The use of Grid computing to drive data-intensive genetic research. European Journal of Human Genetics (March 21, 2007)
8. Elmroth, E., Tordsson, J.: An interoperable, standards-based Grid resource broker and job submission service. In: First International Conference on e-Science and Grid Computing, IEEE Computer Society Press, Los Alamitos (2005)
9. Gui, X.L., et al.: A grid security infrastructure based on behaviors and trusts. In: Grid and Cooperative Computing Gcc 2004 Workshops, Proceedings, vol. 3252, pp. 482–489 (2004)
10. Beckles, B., Welch, V., Basney, J.: Mechanisms for increasing the usability of grid security. International Journal of Human-Computer Studies 63(1-2), 74–101 (2005)
11. Blanchet, C., et al.: GPS@ Bioinformatics Portal: from Network to EGEE Grd, vol. 2006, pp. 187–193. IOS Press, Amsterdam (2006)

Grid Enabling Your Data Resources with OGSA-DAI

Mario Antonioletti[1], Malcolm Atkinson[2], Neil P. Chue Hong[1], Bartosz Dobrzelecki[1], Alastair C. Hume[1], Mike Jackson[1], Kostas Karasavvas[2], Amy Krause[1], Jennifer M. Schopf[2,3], Tom Sugden[1], and Elias Theocharopoulos[2]

[1] EPCC, University of Edinburgh, JCMB, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
`N.ChueHong@epcc.ed.ac.uk`

[2] National e-Science Centre, University of Edinburgh & Glasgow, Edinburgh EH8 9AA, UK

[3] Distributed System Laboratory, Argonne National Laboratory, Argonne, IL, 60439 USA

Abstract. OGSA-DAI (Open Grid Services Architecture - Data Access and Integration) provides an extensible software framework allowing data resources, such as files, relational and XML databases, to be exposed through Web services acting within collaborative Grid environments or, more modestly, in stand-alone mode. OGSA-DAI may be deployed to WSRF-based platforms, such as the Globus Toolkit 4, as well as non-WSRF based ones, such as the UK OMII Server or standard versions of Tomcat and axis. Regardless of the platform, the core functionality provided remains the same. OGSA-DAI allows data resources to be accessed and integrated into the main infrastructures presently being used to construct Grids. OGSA-DAI provides a number of optimisations that reduce unnecessary data movement by shifting work to the Web service and encapsulating multiple client-Web service interactions into a single one, and allows for functionality to be added or customised based on the application. OGSA-DAI is widely used and is available from www.ogsadai.org.uk. It is also bundled with the OMII-UK and Globus Toolkit distributions. This paper gives an overview of what OGSA-DAI is, how it works, presents some usage scenarios, and outlines future enhancements.

Keywords: Data, Databases, Grid, OGSA-DAI.

1 Introduction

With current advances in technology and the decreasing cost of storage, increasingly large amounts of data are being produced, maintained, kept on-line, and shared within communities. For instance, astronomers are collecting data together, such as surveys of the sky made at different wavelengths and resolutions, and making it collectively available through *Virtual Observatories* [1]; biologist are gathering DNA and genomic data from different species and making this data available to biologists through data stores, providing a rich source of data to pursue insights into biological systems [2] and in the health sector, digital medical data are being collected and maintained by hospitals allowing experts to collaborate in patient diagnosis and providing case histories that can be used to inform a prognosis for patients suffering from similar maladies [3,4].

The need to access disparate data sources, often spanning multiple institutions, can lead to new insights and discoveries to be made. By combining different wavelength data for the same patch of sky, astronomers have been able to make new discoveries that would not have otherwise been possible from a single survey [5,6]. Biologists now have the capability of performing cross-species comparisons to determine new genes and their function [7]. Doctors can improve the diagnosis of breast cancer by comparing current mammographs with old mammographs in combination with the associated patient histories [8]. The advantage being able to share data and resources in a controlled manner within a collaborative environment is clear. The provision of generic middleware to facilitate this process is the ethos that is currently driving the evolution of the Grid and, in the data area, OGSA-DAI (Open Grid Services Architecture - Data Access and Integration) provides software which makes it easy to publish and share data across organisational boundaries, and develop applications which use both public and personal data resources, through a secure, extensible framework based on web service standards.

OGSA-DAI is not the only solution currently available for data in the Grid space. *Storage Resource Broker* (SRB) [9], developed by the San Diego Supercomputer Center, provides access to collections of data primarily using attributes or logical names rather than using the data's physical names or locations. SRB is primarily file oriented, although it can also work with various other data object types. OGSA-DAI on the other hand takes a database oriented approach to its access mechanisms. *WebSphere Information Integrator* (WSII), a commercial product from IBM, provides data searching capabilities spanning organisational boundaries, provides a means for federating and replicating data, as well as allowing for data transformations and data event publishing to take place [12]. A more detailed comparison between OGSA-DAI and WSII can be found in [10]. *Mobius* [11], developed at Ohio State University, provides a set of tools and services to facilitate the management and sharing of data and metadata in a Grid environment. To expose XML data in Mobius, the data must be described using an XML Schema, which is then shared via their Global Model Exchange. Data can then be accessed by querying the Schema using, for example, XPath. OGSA-DAI, in contrast, does not require an XML Schema to be created for each piece of data; rather, it directly exposes that information (data and metadata/schema) and relies on the resource's intrinsic querying mechanisms to query its data. These three products all provide mechanisms to share data across organisational boundaries, however they complement the functionality provided by OGSA-DAI.

In the remainder of this paper OGSA-DAI will be examined in more detail. Section 2 gives an overview of the current release of OGSA-DAI explaining the underlying components and how they operate. Section 3 describes some common patterns of use for OGSA-DAI, and Section 4 describes some of the future work planned for the next release. Finally conclusions are provided in Section 5.

2 An Overview of OGSA-DAI

The first thing to note about OGSA-DAI is that it is not targeted directly at the end user, but rather it gives service providers the base functionality which they can use to create their own services and clients to expose data tailored to their own communities. OGSA-DAI has been made extensible by design so that any missing functionality can be developed and grafted to work within the same framework. In addition, different security models may be employed, static metadata can be exposed via configuration files, and dynamic metadata can be created and exposed at the service via the use of call back functions. OGSA-DAI may also be extended to support new types of data resources that are not already supported by the OGSA-DAI distribution. For example, the WebDB project has extended OGSA-DAI to cater for RDF based data [13]. Use of OGSA-DAI allows service providers to develop and deploy their own Grid solutions much more quickly and effectively than might otherwise be the case.

OGSA-DAI is tested and operates well with two current Grid fabric providers, the Globus Toolkit[1] and the OMII-UK[2] and there are plans to port OGSA-DAI to work with UNICORE[3] and gLite[4] under the OMII-Europe project [14]. This ensures that, if any of the above toolkits is to be used, that the OGSA-DAI services will meet user and developer needs in a wide variety of environments.

In OGSA-DAI data resource and service capabilities are exposed through the use of *activities*, the basic unit of work within OGSA-DAI. At the server, an activity is described by a piece of XML Schema specifying the syntax of an XML fragment that is used to activate an associated Java implementation class that performs the desired task at the server. Different XML activity fragments may be composed together in a *perform document* which contains one or more activities linked together through a named set of inputs and outputs describing the data flow between them. For example, an XPath query activity can wrap an XPath expression which then acts on an XML database, the results of this can then be transformed using XSLT activity and finally the transformed results may be delivered to a specified third party using a delivery activity. It is this ability to encapsulate multiple interactions in a single Web service interaction, through the use of perform documents, which otherwise would require multiple distinct client-service interactions, coupled with the fact that activities provide a framework for moving computation close to the data that is seen as one of

[1] www.globus.org/toolkit
[2] www.omii.ac.uk
[3] www.unicore.org
[4] www.glite.org

the advantages of using OGSA-DAI. More complex behaviour may be obtained by composing OGSA-DAI services together and using these to provide more sophisticated capabilities such as *Distributed Query Processing* as provided by the OGSA-DQP project [16].

The OGSA-DAI *Client Toolkit* (CTk) provides a programmatic interface that facilitates programming interactions with OGSA-DAI services. The CTk has an activity representation for each of the server side activities – these representations are essentially used to produce the XML fragment, within the context of a perform document, to trigger the corresponding server side activity. The CTk also provides a programmatic means for composing the client-side activities together to construct the desired perform document – in this way the user does not need to have to deal with any of the underlying XML. In addition, the CTk also handles the interactions with the service and provides methods to add (or extract) data from the request (or result) messages, respectively. Moreover, the CTk is agnostic as to whether a WSRF or non-WSRF service is being accessed providing an additional abstraction layer hiding the particular flavour of OGSA-DAI service that is being contacted. The overall aim of the CTk is to facilitate the provision of clients to interact with OGSA-DAI services.

Putting the above into context a schematic representation of an OGSA-DAI service is shown in Figure 1. A client, built using the CTk, sends a perform

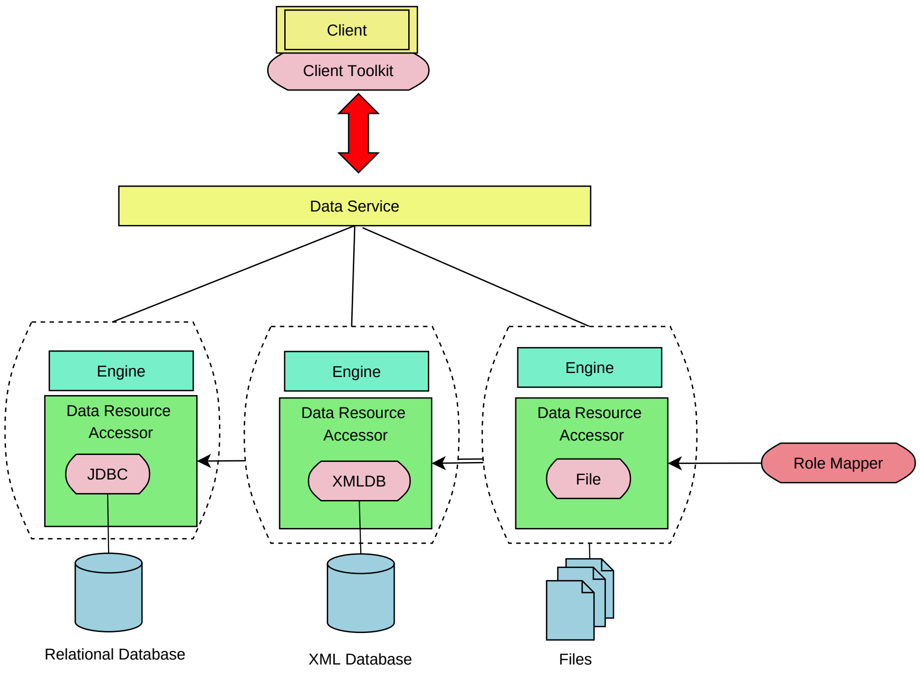


Fig. 1. A schematic representation of an OGSA-DAI service

document to an OGSA-DAI data service, which in the instance shown has three types of data resource associated with it. In the WSRF version of OGSA-DAI WS-Addressing *end point references* are used to specify the data resource being targeted by the client [17]. For the non-WSRF version the data resource name, specified at deployment time, is appended to the service URL, for example

http://myhost:8080/MyService

would become:

http://myhost:8080/MyService/MyDataResource.

Once a message is accepted by the service interface, the functionality for both flavours of OGSA-DAI is the same. A perform document and any Grid credentials are passed through the service layer to the *Engine* of the targeted data resource. The Engine coordinates the running of the activities in the perform document. A *Data Resource Accessor* (DRA) wraps the underlying *data resource*: this abstraction facilitates the addition of new types of data resources to OGSA-DAI. The Engine passes any Grid credentials to the DRA and, if these are valid, the DRA returns an open connection to the data resource that can then be used by any activity that interacts with the data resource. The DRA consults a *Role Mapper* that maps Grid credentials, essentially the *distinguished name*, to a database role that can be used to access the database. OGSA-DAI comes with a basic role mapper that attempts to match a database role (represented as simple username and password pairs) within an XML file for a given a set of Grid credentials, thus allowing database systems which do not use Grid credentials to be accessed, albeit not in a scalable fashion. This is another extensibility point where service providers would wish to develop their own role mapper and substitute the existing one: two groups have developed different solutions to this.

The OGSA-DAI Engine ensures that all activities run correctly and coordinates the passing of data from one to the other. Failure in one activity signifies failure in the execution of the whole perform document. As yet there is no transactional behaviour, including rollback mechanisms, although this is planned for a future release. Data may be piped in from a third party using a *delivery from* activity or sent to a third party using a *delivery to* activity, both of these can use other transport protocols to pipe data into or out of a service obviating the requirement for SOAP and using, for example, GridFTP, FTP or HTTP to fetch data or send it to a third party. If the processing completes successfully the data or status of the processing is sent back to the client in a response document.

From this brief overview we can see that OGSA-DAI is a sophisticated piece of middleware that provides a uniform access interface to various types of data. It partially virtualises data: intrinsic connection mechanisms to the underlying data resource are no longer a concern but a client still needs to know the underlying type of data model that is being used – for instance SQL queries need to be targeted at a relational data resource and will make no sense when targeted at an XML database. Moreover, query expressions targeted at a particular data resource are not inspected so any vendor specific language extensions must also be appropriate for the underlying data resource used. A client is able to determine the type of the underlying data resource via metadata available through the service interface that then allows it to direct the appropriate type of queries for that type of data resource. However, OGSA-DAI does provide the basis for providing data model integration, through the use of transformation activities which e.g. translate the results of queries to XML and relational data resources into WebRowSet before aggregating them. This then outlines the basics of the OGSA-DAI framework. The next section briefly outlines a couple of usage scenarios.

3 Deployment Scenarios

OGSA-DAI provides a versatile framework which can be used to provide data access capabilities within Grid infrastructures. Many projects already use OGSA-DAI, primarily in research areas such as GIS and bioinformatics. An up to date list can be found on the OGSA-DAI website[5].

Five basic common usage patterns are illustrated in Figure 2.

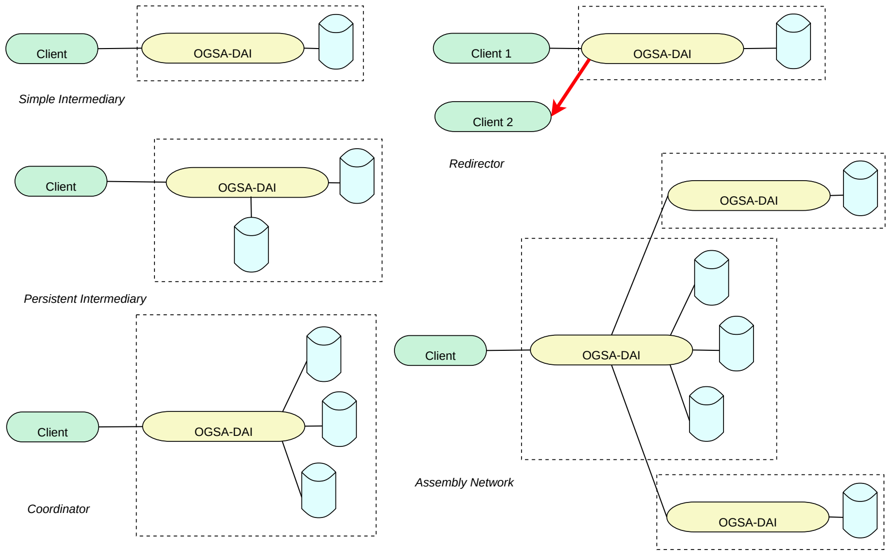


Fig. 2. OGSA-DAI scenarios

The *simple intermediary* is the simplest archetypal usage scenario supported by OGSA-DAI, and is the basis for many of the higher-level scenarios. This scenario consists of an OGSA-DAI service interposed between client applications

[5] See http://www.ogsadai.org.uk/about/projects.php

and a data resource providing a consistent interface for different kinds of data and supporting a rich, extensible set of operations that can be performed on that data. Using this base scenario one can envisage many discoverable OGSA-DAI services listed in third party registries and used by clients to retrieve data for their specific ends, all different types of data shared and made available through a common interface. In addition, examination of this basic usage pattern has also led to various optimisations being made for the 2.2 release of OGSA-DAI, see [18] for more details.

The *persistent intermediary* scenario illustrates the use of mechanisms for storing intermediate results which can then be used by subsequent requests. These intermediate results could be stored transparently in memory, a local database, the local file systems or some other suitable means on behalf of the OGSA-DAI service clients. This scenario currently is partially supported by using the OGSA-DAI *dataStore activity* which currently holds results in memory, although this can be extended to hold results in more permanent storage. It can also be implemented using OGSA-DAI by storing data temporarily in a scratch database accessible by the service. This functionality allows a coordinating service to hold temporary data to perform data joins from multiple data resources.

The *redirector* scenario allows data to be sent to a third party, including the originator, as opposed to embedding it in the response. Moreover the third party delivery protocol does not have to be SOAP based – data can be delivered using GridFTP, ftp, or some other delivery means. In this instance SOAP is effectively being used as the control channel while the data channel is done via a more efficient transport protocol. OGSA-DAI supports this scenario by allowing a number of alternative data transport mechanisms that can also be used to transfer data into OGSA-DAI services.

In the *coordinator* scenario, an OGSA-DAI service interfaces to an arbitrary number of data resources and presents them as a composite resource to its clients, producers, and consumers. This means that data can be routed between data resources or combined from those resources within a single request or session without routing data via the client. There is already some support for this type of scenario in OGSA-DAI as multiple data resources can be configured per data service and used with specialised query activities to provide resilient querying of a set of data resources sharing a common schema. This presents the set of data resources as a single virtualised data resource.

In the *network assembly* scenario an OGSA-DAI service uses an arbitrary number of other OGSA-DAI services as well as data resources already curated by the service in order to collect together data. This type of service coordination successively adds facilities that may be used in combination towards achieving a data-oriented workflow. The invoked services in this workflow do not have to be OGSA-DAI services. The multiple services may form a pipeline in order to draw on additional computation facilities or a tree in order to place parts of a total query close to the data sources. As this permits arbitrary fan out and arbitrary recursive composition, many architectures are possible: a simple example is shown above. OGSA-DQP provides an instance of the assembly network pattern using OGSA-DAI services as well as some of their own service types.

In general the documentation of scenarios like those described above is beneficial as a means of providing best practice and guidelines for using the features and components of OGSA-DAI. There is insufficient space here to go into more depth but best practice and guidelines are being documented in the OGSA-DAI Web pages[6]. These scenarios, as well as other inputs such as performance studies [19], are being used to motivate the future directions being taken by OGSA-DAI which are briefly outlined in the next section.

4 Future Directions

A number of architectural changes are about to be introduced into the next OGSA-DAI release and following releases. Some of the highlights are:

– *Improved scalability* by providing load balancing capabilities to dispatch incoming request to different JVMs, potentially running on different machines, to execute perform documents. Initial policies will be simple, eg. round-robin, but additional more complex policies will be enabled as well.
– *Improved robustness* by allowing requests to run on different JVMs so if that a request has aberrant behaviour it does not bring down the whole container and compromise other jobs.
– *Improved activity model* to make it easier to develop and maintain activities while at the same time providing more powerful mechanisms to dynamically configure activities.
– *Improved sessions handling* will allow activities to store and retrieve data from an existing session, which could span multiple requests.
– A *new resource model* will allow perform documents to contain activities that can access more than one resource exposed by an OGSA-DAI service (currently a perform document can only target a single data resource). This new model will thus allow more powerful user-driven data integration scenarios to be enacted by an OGSA-DAI service between multiple resources. It will also allow other OGSA-DAI components to be treated as WSRF-resources. For example, sessions, requests, and data sinks/sources (input/output streams) can be modelled as resources which then allows these to be endowed with mechanisms for lifetime management and authorisation as available in other resources.
– *New data integration activities* taking advantage of the new resource model a new set of data integration activities are being designed that should facilitate the enactment of data integration scenarios.
– *Distributed Query Processing* capabilities are being introduced through the absorption of the OGSA-DQP project, currently distributed separately from OGSA-DAI, into the OGSA-DAI product itself.

[6] See www.ogsadai.org.uk/documentation/scenarios for details.

- A *new tuple intermediate data format* for relational data, called an *ODTuple* (for OGSA-DAI Tuple), will provide a common way for connected activities to exchange data. This will minimise the amount of data conversion that is required take place between activities. This format is:
  - light-weight,
  - able to stream well within and between processes,
  - efficient for single types, elements, and tuples,
  - able to support base types plus String, File, BLOB, and NULL,
  - able to supports warnings, errors and exceptions, and
  - easily extensible.

  This relational structure can be used to represent the majority of the current data formats used within OGSA-DAI, such as WebRowSet and CSV (Comma Separated Values).

These additions to the next release will make OGSA-DAI a more powerful framework and increase the support for data integration as well as making the scenarios described in the previous section easier to implement and extend.

5 Conclusions

This paper has provided motivation for the production of middleware to facilitate the sharing of data within established communities to enable new insights and discoveries to be produced. The provision of middleware that facilitates this process is the underlying motivation for OGSA-DAI. OGSA-DAI is not targeted directly at the end-user but rather it provides a framework that has to be customised for a given user-community by its own developers. Through the use of OGSA-DAI the amount of effort required to produce these targeted data services and applications should be greatly reduced. A snapshot overview of OGSA-DAI has been given and some indicators of the future directions that are being taken to enhance the product and provide additional capabilities for those that rely on OGSA-DAI for their data access and integration base requirements. More information about OGSA-DAI and the software may be downloaded from the project Web site at www.ogsadai.org.uk.

Acknowledgements

This work is supported by the UK e-Science Grid Core Programme, through the Open Middleware Infrastructure Institute UK, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. Djorgovski, S.G.: Virtual astronomy, information technology, and the new scientific methodology. In: Proceedings of the Seventh International Workshop on Computer Architecture for Machine Perception, 2005. CAMP 2005, July 4-6, 2005, pp. 125–132 (2005)
2. Edgar, R., Domrachev, M., Lash, A.E.: Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. Nucleic Acids Research 30(1), 207–210 (2002)
3. Brady, J.M., Gavaghan, D.J., Simpson, A.C., Mulet-Parada, M., Highnam, R.P.: eDiaMoND: a Grid-enabled federated database of annotated mammograms. In: Berman, F., Fox, G.C., Hey, A.J.G. (eds.) Grid Computing: Making the Global Infrastructure a Reality. Wiley Series, pp. 923–943 (2003)
4. Buetow, K.H.: Cyberinfrastructure: Empowering a "Third Way" in Biomedical Research. Science 308(5723), 821–824 (2005)
5. Virtual observatory finds black holes in previous data. News in brief. Nature 429, 494–495 (June 2004)
6. Astronomers Detect New Category of Elusive 'Brown Dwarf'. The New York Times, (Tuesday June 1, 1999)
7. Wipat, A., Sun, Y., Pocock, M., Lee, P., Watson, P., Flanagan, K.: Developing Grid-based Systems for Microbial Genome. Comparisons: The Microbase Project. In: Proceedings of the UK e-Science All Hands Meeting (2004)
8. Solomonides, A., McClatchey, R., Odeh, M., Brady, M., Mulet-Parada, M., Schottlander, D., Amendolia, S.R: MammoGrid and eDiamond: Grids Applications in Mammogram Analysis. In: dos Reis, A.P., Isaias, P. (eds.) Proceedings of the IADIS International Conference: e-Society 2003, Lisbon, Portugal. June 2003, pp. 1032–1033 (2003)
9. Storage Resource Broker (SRB), `www.sdsc.edu/srb`
10. Sinnott, R.O., Houghton, D.: Comparison of Data Access and Integration Technologies in the Life Science Domain. In: Proceedings of the UK e-Science All Hands Meeting 2005 (September 2005)
11. Mobius, projectmobius.osu.edu
12. Web Sphere Information Integrator (WSII), `www.ibm.com/software/data/integration`
13. OGSA-DAI-RDF project, `www.dbgrid.org`
14. OMII-Europe project, `http://www.omii-europe.org`
15. InteliGrid project, `http://www.inteligrid.com`
16. Alpdemir, N., Mukherjee, A., Gounaris, A., Paton, N.W., Watson, P., Fernandes, A.A.A.: OGSA-DQP: A Grid service for distributed querying on the Grid. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 858–861. Springer, Heidelberg (2004)
17. Gudgin, M., Hadley, M., Rogers, T.: Web Services Addressing 1.0 - Core (WS-Addressing). W3C Recommendation (May 9, 2006)
18. Dobrzelecki, B., Antonioletti, M., Schopf, J.M., Hume, A.C., Atkinson, M., Chue Hong, N.P., Jackson, M., Karasavvas, K., Krause, A., Parsons, M., Sugden, T., Theocharopoulos, E.: Profiling OGSA-DAI Performance for Common Use Patterns. In: Proceedings of the UK e-Science All Hands Meeting 2006 (2006)
19. Kottha, S., Abhinav, K., Muller-Pfefferkorn, R., Mix, H.: Accessing Bio-Databases with OGSA-DAI - A Performance Analysis. In: Dubitzky, W., Schuster, A., Sloot, P.M.A., Schroeder, M., Romberg, M. (eds.) GCCB 2006. LNCS (LNBI), vol. 4360, Springer, Heidelberg (2007)

UniGrids Streaming Framework: Enabling Streaming for the New Generation of Grids

Krzysztof Benedyczak[1], Aleksander Nowiński[2], Krzysztof Nowiński[2], and Piotr Bała[1,2]

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, PL-87-100 Toruń, Poland
golbi@mat.uni.torun.pl

[2] Interdisciplinary Center for Mathematical and Computational Modelling, Warsaw University, Pawińskiego 5a, PL-02-106 Warsaw, Poland

Abstract. We present a new infrastructure for high performance streaming in OGSA/WSRF compliant grid. The UniGrids Streaming Framework (UGSF) works with UnicoreGS as WSRF hosting environment. The paper discusses the advantages of mixed SOAP based control with highly efficient streaming. The UGSF components, streaming server and WSRF web service are described along with a detailed performance analysis including comparison to standard solutions. Some applications based on the UGSF are also presented.

1 Introduction

Current trends in grid technology is clearly focused on OGSA (Open Grid Services Architecture) [1] which implies usage of the web services. The detailed guidelines on how to build grid services are given by the WSRF specifications [2]. The consensus about the importance of such approach was motivated by many reasons. Here we can point to an interoperability as the most significant one. The WSRF as well as other specifications allow developers to easily create grid software compatible with other WSRF implementations. Moreover, as web services technology is widely adopted in B2B applications, one can make use of existing experiences and adopt available solutions. A good example is the BPEL [3] specification defined for business processes, which is now being used as the tool to define grid workflows. These reasons form a solid base for OGSA which plays a vital role in grids nowadays. But we can not forget that web services also have disadvantages. Here we will focus on two of them that are crucial for data streaming.

The first and the most important drawback of web services is efficiency. Web services technology is based on the SOAP protocol. This results in extensive usage of XML. The obvious consequence is a large overhead for even a simple operation: the SOAP engine has to perform a lot of XML parsing or encoding. Moreover, XML data encoding is very verbose, thus ineffective. In the most streaming applications such data overhead is undesirable. The other problem is data streaming: SOAP is message driven and XML to be parsed must be fully read.[1]

Presented disadvantages cause that web services technology can not be seen as suitable for any interactive, real-time application. It is hard to imagine a scientist steering an interactive device with latency of every operation measured in seconds. Of course, various XML technologies like binary encodings, aforementioned streaming processing of XML, MTOM [6] or TCP bindings can be used to boost the performance of web services. We are sure that in some cases it is possible to build streaming technology on top of such optimised web services. An obvious advantage of such approach, is better integration with existing web services agents (like new UniGrids gateway) and not much more. It is also clear that such solutions can be useful only for less demanding streaming applications.

To solve the problem we have developed a hybrid system which is a platform to build any type of streaming services managed in WSRF compliant way on. The solution is highly responsive and efficient.

2 System Design

The UGSF system is based on the WSRF's compliant version of the well recognized UNICORE middleware [8]. The UnicoreGS [7] is used as the WSRF hosting environment.

The aim of the UniGrids Streaming Framework (UGSF) is to provide direct data streaming and steering for applications. The main part of UGSF is *UGSF core* which is a middleware that allows developers to create dedicated streaming solutions.[2] Substantial effort was made to prepare a system where the creation of a specialized solution is as easy and quick as possible. Every system based on the UGSF will use the *core* together with some application dependent code. The UGSF core provides basic functionality common for all streaming applications. This includes creation or shut-down of a connection. The system is designed in such a way that a group of versatile software pieces can be reused. A good example is the component to locate UnicoreGS job's working directory and access it's contents.

The *UGSF core* consists of a *UGSF Web Service* part, a *Streaming Server* part and a library to create clients. The usage of the last component is optional. The *UGSF Web Service* takes advantage of WSRF capabilities. It is used to control a set of available stream types, to create new streams and to manage already created ones. The *Streaming Server* part is managed by a *UGSF Web Service* and performs streaming. The cClient library is used to simplify the creation of the client-side software. Overall architecture is shown in Figure 1.

[1] It is worth to note that currently there are intensive efforts to eliminate this issue and hopefully new generations of SOAP engines (as AXIS 2 [4] or XFire [5] with support for StaX) will solve it.

[2] We will use this therm whenever we will refer to basic framework, without actual stream implementations which are also included in UGSF distribution.

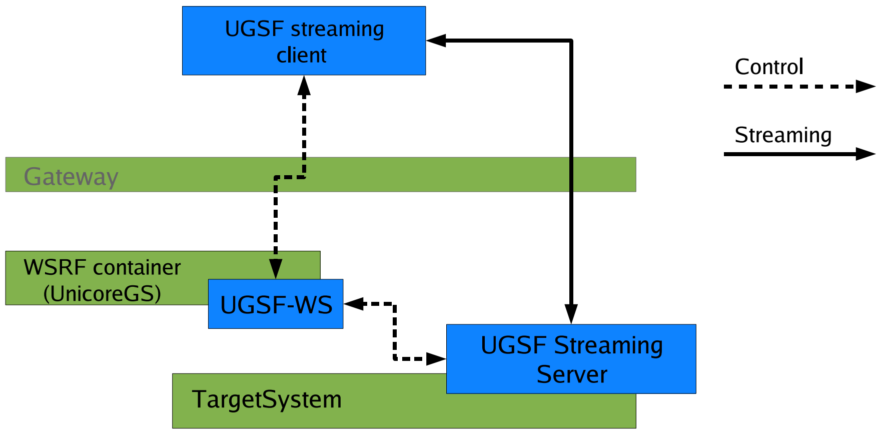


Fig. 1. The general architecture of UGSF

The *UGSF core* is complemented with stream implementations. These consists of two parts: the streaming server and the web service modules. The web service module implements control operations specific to the stream implementation. The streaming server module deals with a wire streaming protocol and data consumption/acquisition.

The general pattern of UGSF usage is as follows:

- The UGSF installation is configured by the system administrator, who defines so called *stream types*. Every *stream type* is one stream implementation with some configuration parameters (which can't be modified by users).
- The user chooses the *stream type* and creates its instance. If the implementation stipulates that some user's parameters are needed, then the user must supply them. As a result a reference to the newly created stream management WS-Resource is returned.
- The user can invoke any common (provided by *UGSF core*) or special (stream implementation defined) operation on the WS-Resource assigned to the created stream. The resource properties contain (among others) information about how to connect to the *UGSF streaming server* to start streaming.
- The user connects to the *UGSF streaming server* and starts the data transfer. It is possible to control the connection via the web service interface.

The *Streaming Server* and the clients built for UGSF are grid-enabled. Therefore, the UGSF can be used to let legacy applications benefit from the grid technology (e. g. grid authorization), using already developed stream implementations.

To accomplish general overview of the UGSF we present details about the underlying transport level protocol. In principle, the UGSF is highly flexible and can be used for any application level protocol. Currently there is no possibility to use any protocol than TCP. This decision was motivated by multiple factors. Our first aim was to support tunneling of streamed data with the UNICORE gateway, which can operate only on TCP connections. Another reason is that the usual use of grid middleware requires high security and reliability of connections (e.g. scientific applications which stream video must not loose any frames contrary to typical multimedia situations when such loss is acceptable). This is much easier to implement in general framework based on TCP/TLS. Nevertheless, in the future versions of the UGSF the UDP entry points can be added. This will involve some redesign of the UGSF *Streaming Server*.

2.1 UGSF Web Service

The *UGSF Web Service* component consists of two kinds of web services. A base one (called *StreamingFrameworkService*) is responsible for the connection authorisation, the creation of stream and its setup. During this process the new WS-Resource (called *StreamManagementService*) is created with a dedicated web service interface. This WS-Resource acts as a controller of an active streaming connection.

The *StreamingFrameworkService* is a WS-Resource which maintains lists of *StreamManagementServices*. It can be argued that this is a perfect case for the WSRF Service Group which is a federation of WS-Resources. Unfortunately, the Service Group can't be used here due to the security restrictions. The WSRF specification doesn't permit filtering the Service Group's content. As a result every user would have the possibility to see other users streams.

The *StreamingFrameworkService* allows users to get a list of available streaming services and setup a connection to the specific streaming service. The list of both owned and accessible streams is available (see section 2.3 for details). In addition, the *StreamingFrameworkService* has an administrative interface, which empowers a system administrator to enable and disable particular stream types on the fly. The service reconfiguration such as addition or removal of stream types is also possible.

For each created stream, an instance of the *StreamManagementService* allows the user to perform universal operations for all streams. This includes shutting the stream down (by means of WS-Lifetime interface), getting status and statistics of the connection, as well as pausing or resuming streaming. This functionality can be easily enriched by the developer. He can extend *StreamManagementService* with additional operations. The enriched implementations are free to consume any special XML configuration supplied to the *StreamingFrameworkService* and required for service setup and creation.

We have also developed an additional service called *StreamingFrameworkFactory*, which allows site administrators to create base UGSF services and configure them initially. The developed service follows the pattern of the UniGrids atomic services [7].

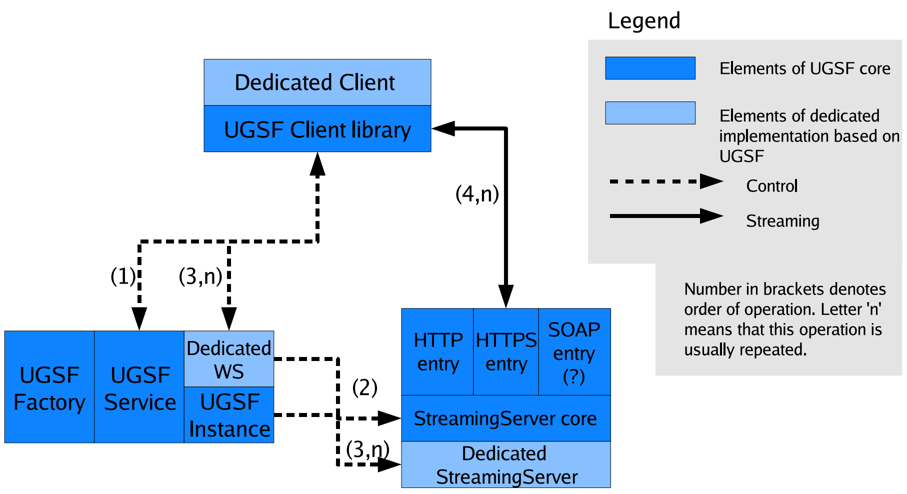


Fig. 2. Services and modules of UGSF components

2.2 UGSF Streaming Server

The UGSF *Streaming Server* is a stand-alone, modular application, which performs streaming to and from the target system. The server is tightly connected to the *UGSF Web Service* which maintains stream definitions. The communication is done with Java RMI. The server is modular, and highly configurable. The dedicated modules were created to access the actual streaming data source. Such a module also gives access to the running job's outcome. It can also provide it with input, if required. On the other hand there are stream modules that don't need any job to cooperate with. A module which gives access to physical resources like a video camera is a good example. Another one is a module which enables grid usage of the legacy TCP or UDP servers. There is also a whole class of auxiliary modules which acts without any external resources. These modules, for example, convert input data from one format to another.

For a particular site, there can be more than one *Streaming Server*, operated by only one *UGSF Web Service*. Each server is able to handle multiple stream modules. There is also the possibility to configure another setup: one single *Streaming Server* can be managed by more than one *UGSF Web Service*. However, such a scenario is of little practical value.

The access to the *UGSF Streaming Server* is accomplished with a special protocol. Currently the protocol is trivial but it may be developed to a more complicated one when new features are needed. The access to the *Streaming Server* is done by means of exchangeable *entry modules*. More than one entry module can be turned on simultaneously. Every stream can be accessed by many entry protocols and the application can choose the one it prefers or understands. Currently there are available HTTP and HTTPS entry points with simple POST based protocols (in fact there is one entry point which can be configured to use or not to use TLS). The system is ready to use the other protocols as well.

2.3 Advanced Features and Security

In addition to the basic infrastructure for streaming connection creation, the UGSF provides a set of advanced stream related operations. These operations focus on a sophisticated data flow creation. By the term *data flow* we mean the here composition of one or more streams between servers and/or clients created for one application.

Every stream implementation can contain more than one *flow*. Flow is a synonym to a connection, e.g. if one stream maintains three flows then it is possible to open three concurrent connections to this stream. This provides an opportunity to create more advanced streams with a clear separation of logical "flows" of data, including separation of input and output.

The UGSF streams have metadata attached. For every flow there are defined, among others, supported formats. It is possible to specify more than one format for a single flow, as well as express the only supported format combinations. Any flow can have two-way traffic, but it is suggested that a flow should only use input or output whenever possible (so to be one directional). When two-way traffic is required, two flows are preferred. Streams designed this way are much more effortlessly integrated into data flows.

In order to enable composition of other than trivial data flows (i.e. client $\leftrightarrow$ server), UGSF offers a connect operation. It instructs an already created flow of one stream to exit its passive state and to actively initiate connection to another flow.

There is also a possibility to create a flow with "cloning" ability. Such a flow can be used to dynamically create new flows in a stream implementation. A good example of the cloning feature is a multiplexer, which basically manages two flows, the input and output. The output flow has cloning ability and the user can clone the output flow multiple times. As a result he can fork the input into arbitrary number of outputs.

Up to now, we haven't covered one significant aspect of the UGSF system: security. The main question here is: What are the requirements to open connection to the *Streaming Server*? The simplest approach is to enable access to the stream only to its owner. Unfortunately, such a method is not sufficient for more complicated scenarios, such as server $\leftrightarrow$ server connections. To give an example; Let's consider a data flow where server A is the source of data. This data should be processed by a server B and finally the output should be received by the user U. If U creates appropriate stream instances on A and B, B will not be allowed to access A's stream - only U will be.

To solve this problem, every flow is assigned a token, which is the identity of its owner and its access policy. The token is a large unique number. The access policy is defined by the creator of the stream and describes who is authorised to contact the flow. The default policy is "owner-only". In this case only users with a certificate matching the flow owner's certificate can open a connection. He has also to present the flow token for identification purposes. Please note that the token value is not sensitive as it is valid only after a connection is established using a valid certificate. Policy can allow public (non restricted) access and also an explicitly specified entity to access the stream.

In the matter of UGSF security we still have some work to do. We would like to provide XACML [9] support for policy description. Also some trust delegation should be supported to achieve better integration with standard grid trust delegation (but this is a matter of better system cohesion).

3 Applications

The UGSF system includes several basic stream implementations.

The first one is *TCPStream*, which can be seen as a grid version of the SSH tunnel. It has a similar functionality to such a tunnel and an obvious advantage is that users don't need shell access to the grid site. Moreover, they are authorised in the same way as for any other UniGrid services.

Using available client software for creating TCP tunnels in the UGSF package, we have used UGSF to steer an Advantech's ADAM/5000TCP device with an existing application. The ADAM/5000TCP is a Modbus [10] Ethernet device. The UGSF is very useful because Modbus Ethernet devices are in general insecure and must be protected by firewalls. This example shows that by using UGSF, the whole range of Ethernet devices can be secured and grid enabled only by putting in a few lines in the UGSF configuration file.

The *TCPStream* is accompanied by *UDPStream* which does a task not widely available by any other software. It tunnels UDP datagrams over the TCP protocol, maintaining UDP "sessions" in a manner similar to that used by firewalls (packets are scanned for changes of destination ports).

In the UGSF there is also a *FileStream* implemented. It serves as a streaming version of a file access service. The *FileStream* has the ability to detect file growth, and allows to stream file content as new data is put in. Clearly, this solution is targeted to receive results of arbitrary simulation in real time. To monitor grid job results, there is another stream called *IVisStream* which is a simple extension of *FileStream*. It supports, in addition to *FileStream* features, location of files outputted by a given grid job.

Currently we are working on more universal stream types, which will support data flow creation, (as e.g., the Multiplexer stream for splitting arbitrary flow into multiple ones) and to add generic support for video streaming which is a necessity for many streaming applications. We chose Theora [12] as our "native" codec. Streams to compress raw video (and decompress it) will be available shortly.

4 Performance

During the design of the UGSF, our aim was not to introduce *any* penalty on throughput (except enforcement of TCP and use of SSL in most cases). It was achieved, as after stream setup, the developer can use an arbitrary protocol on open socket connection. The *UGSF core* does not add any extra data to the

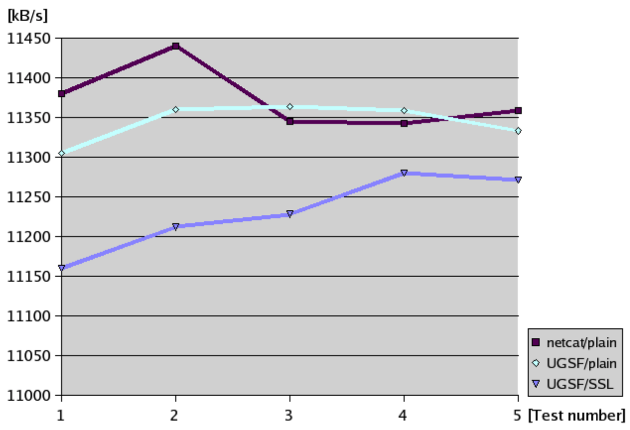


Fig. 3. The performance of plain netcat stream versus netcat over UGSF TCP tunnel (over the 100Mbit network)

opened stream. To check the performance and see how tunneling over a particular stream will impact it, we have run performance tests on the *TCPStream*. For the tests we have used netcat TCP session. We have compared a direct connection to one tunneled via *TCPStream*. Two machines running Fedora Core 4 operating system (kernel version 2.6.13-1.1532_FC4) were used. The systems were interconnected with 100Mbit Ethernet. Server machine (B) was equipped with Intel Xeon 2,4GHz CPU and the client was AMD Athlon 64 3400+ CPU (A).

As it is shown in the figure 3, the plain UGSF tunnel performs nearly the same as the plain netcat connection. The SSL version is, of course, slightly slower, but still the difference is tiny and acceptable in most usage scenarios.

We have also looked at the CPU usage reported by the *Streaming Server*. The server consumed less than 15% of the CPU time at A and about 2–3% more CPU

Table 1. Performance comparison of the UGSF and web service based implementation. The data was sent in small chunks in two directions. RQ stands for "request" and RE for "response". The third column contains the total number of full message exchanges (i.e. sending request and receiving response) per second.

Messages sizes	**Implementation**	**(RQ + RS) per second**	**Relative speedup**
RQ 16B/RS 10kB	web service/UnicoreGS 1.4.1	0.96	1
RQ 10kB/RS 10kB	web service/UnicoreGS 1.4.1	0.48	1
RQ 16B/RS 10kB	UGSF/Java DataStream	23.38	24
RQ 10kB/RS 10kB	UGSF/Java DataStream	20.76	43

time at B machine. When SSL is turned on, the CPU intensive encryption caused the increase of the systems utilization to about 50 and 55 percents respectively.

To summarize, the results are promising: practically there is nothing to optimize. The CPU operations on recent hardware does not impact throughput of 100MBit streams and there is still some CPU power left. Moreover, the CPU intensive operations are mostly those coming from the SSL sockets implementation of the Java toolkit.

The most interesting topic is the comparison of operations invoked by means of standard web service calls, with an analogous system based on the UGSF. The web services operate exclusively on messages. The AXIS 1.x environment used in the UniGrids project limits it to complete message exchanges instead of real streaming. It can be theoretically proven that one-way web services can resolve this issue. However, it is problematic from the server side when a client is behind the firewall/NAT. Some progress can be made by using HTTP 1.1 persistent connections [11] but currently this (along with other needed functionality) is not available in AXIS 1.x.

In order to run comparison tests we have developed a trivial UnicoreGS service with only one operation, which consumes and returns a configurable amount of raw data (Base64 encoded). The results of running series of operations on this service are given in the first two rows of tab. 1. Also small client-server application was prepared to test UGSF version. It was used through UGSF TCPStream, and as internal protocol we used Java DataStream. As it can be noted from the last column of table 1, the speed up is more than 20. In fact, this is the minimal performance gain. In reality UGSF can be used to operate much more effectively: by implementing specialised UGSF stream type, the two extra data hops introduced by generic TCPStream and it's client can be eliminated. Moreover, in many cases streaming applications can benefit from parallel streaming, while in the tests we were using synchronised message exchanges. Test results were obtained on the same machines as above.

We would like to mention that there is a lot to improve in the web service version too. A Better SOAP engine (e.g. AXIS 2), and usage of its features, can give a substantial performance boost. Also UnicoreGS currently is still in development and there were no optimisations made.

5 Summary

The presented development is focused on various applications where UGSF will have a possibility to prove its value. We consider a device access and remote steering, video transmission and scientific image processing. Of course, visualisation and real-time monitoring of computation are also of interest, as it has already been presented for the UNICORE middleware [13]. The UGSF includes two stream implementations that allow for tunneling connections to both TCP and UDP legacy servers on the grid site. Services to stream changing content of grid jobs in real-time are also ready to be used. Support for data flow creations encourages to use UGSF in a component driven way, where already created stream implementations are reused in larger applications. In general, the developed infrastructure opens a field to numerous applications, which require on-line data streaming and steering.

This work was supported by European Commission under IST grant UniGrids (No. 004279).

References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project (2002), http://www.globus.org/alliance/publications/papers/ogsa.pdf
2. Czajkowski, K., Ferguson, F.D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The WS-Resource Framework, Version 1.2. OASIS (Organization for the Advancement of Structured Information Standards) (April 2006), http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf
3. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S. (eds.): Trickovic, I., Weerawarana, S. Business Process Execution Language for Web Services, Version 1.1. OASIS (2003)
4. Axis 2 project (August 2006), http://ws.apache.org/axis2
5. XFire project (August 2006), http://xfire.codehaus.org
6. Gudgin, M., Mendelsohn, N., Nottingham, M., Ruellan, H.: SOAP Message Transmission Optimization Mechanism, W3C Recommendation (January 25, 2005), http://www.w3.org/TR/2005/REC-soap12-mtom-20050125
7. UniGrids project website (August 2006), http://www.unigrids.org
8. UNICORE at SorceForge (August 2006), http://sourceforge.net/projects/unicore
9. Godik, S., Moses, T.: eXtensible Access Control Markup Language, Version 1.1. OASIS Committee Specification (August 07, 2003), http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf
10. Modbus Organization, Inc. and Schneider Automation Inc.: MODBUS Application Protocol Specification, vol. 1.1 (August 2006), http://www.modbus.org/specs.php
11. Fielding, R., et al.: RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1, Section 8.1: Persistent Connections. The Internet Society (1999), http://www.ietf.org/rfc/rfc2616.txt
12. Theora I Specification. Xiph.org Foundation (March 7, 2006), http://www.theora.org/doc/Theora_I_spec.pdf
13. Bała, P., Benedyczak, K., Nowiński, A., Nowiński, K.S.: Real-time visualisation for Unicore middleware. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 608–615. Springer, Heidelberg (2006)

Dynamic Clusters Available Under Clusterix Grid

Jan Kwiatkowski[1], Marcin Pawlik[1], Gerard Frankowski[2], Kazimierz Balos[3], Roman Wyrzykowski[4], and Konrad Karczewski[4]

[1] Institute of Applied Informatics, Wroclaw University of Technology, Wybrzeze Wyspianskiego 27, PL-50-370 Wroclaw, Poland
jan.kwiatkowski@pwr.wroc.pl
[2] Poznan Supercomputing and Networking Center, ul. Noskowskiego 10, PL-61-704 Poznan, Poland
[3] Institute of Computer Science, AGH, ul. Mickiewicza 30, PL-30-059 Krakow, Poland
[4] Institute of Computer and Information Science, Czestochowa University of Technology, Dabrowskiego 73, PL-42-200 Czestochowa, Poland

Abstract. The increase of the speed of computer networks paired with the ubiquity of inexpensive, yet fast and generously equipped hardware offers many organizations an affordable way to increase the available processing power. Clusters, hyperclusters and even grids, not so long ago seen only in huge datacenters, can now be found helping many small organizations in solving their computational needs. Clusterix is a truly distributed national computing infrastructure with 12 sites (static Linux clusters) located across Poland. The computing power of Clusterix can be increased by connecting additional clusters. These clusters are called dynamic because it is assumed that they will be connected to the core infrastructure in a dynamic manner, using an automated procedure. In the paper we present the design foundation of the Cumulus grid deployed at Wroclaw University of Technology, together with the method for its integration as a dynamic component in the Clusterix grid.

1 Introduction

In many fields of science, the increase of computational requirements has made access to powerful computational installations a necessity. The creation of a dedicated Beowulf type cluster, composed of cheap commodity hardware is currently in the reach of many organizations. Although this standard procedure has many advantages, its creation is not always the best solution. In the paper we propose an alternative approach - the creation of a dynamic cluster ready to be utilized as a dynamic subsystem of the Clusterix grid [2,5]. This scheme is illustrated by the example of the grid installation located at the Wroclaw University of Technology.

The main objective of the Clusterix national grid project, coordinated by Czestochowa University of Technology, is to develop mechanisms and tools that allow for deployment of a production grid with the core infrastructure consisting of static clusters based on 64-bit Linux machines [10]. Local PC-clusters are placed across Poland in independent centers connected by the Polish Optical Network PIONIER. Currently, the core infrastructure of the Clusterix comprises 250 Itanium2 CPUs located in 12 sites. The computing power of the Clusterix environment can be increased by connecting additional dynamic clusters. The automated procedure is prepared to assist their attachment and detachment.

The term dynamic cluster (as opposed to static cluster) can be used to describe local clusters that are dynamically and frequently attaching and detaching themselves to a grid environment. It can also be used to describe the clusters created from the machines operating part-time as the computational nodes and part-time detached from their cluster to perform other duties. In the Cumulus grid the clusters are dynamic in both of the above meanings, but in the context of the Clusterix grid only the first one is used. The paper presents the design, the hardware and software configuration of the Cumulus dynamic grid and details of the method of its integration into the Clusterix grid environment.

The structure of the paper is as follows: In section two, the architecture of the Cumulus grid and its dynamic clusters is described. Section three outlines the elements of the Clusterix architecture utilized during dynamic cluster attachment. Section four describes the dynamic cluster connection procedure. In the final section the conclusions and future work plans are presented.

2 Cumulus Grid Design Overview

The Cumulus grid [6] is composed of three clusters named Calvus, Humilis and Nimbus. The clusters are located in two separate buildings of the Faculty of Computer Science and Management, Wroclaw University of Technology (Fig. 1). The Calvus and Humilis clusters are composed of dynamic and dedicated nodes, while in the Nimbus test cluster only two dedicated nodes are available. The dynamic nodes - located in the University computer laboratories - are logically connected to the cluster only in the predefined hours, when they are not needed for other purposes. The dedicated nodes form the static parts - full-time available in the clusters. They have the whole of their hard disk space devoted to the scratch and swap space utilized by the cluster system. The Calvus cluster consists of 40 dynamic nodes and 2 dedicated ones. The Humilis cluster is built of 16 dynamic nodes and 2 dedicated ones.

One of the most important Cumulus grid design requirements was that its deployment would not interfere with a typical role of the computers used for university courses. During office-hours the dynamic nodes are detached from their cluster. When the dynamic nodes are not available, only the dedicated ones are operational - the cluster can not be efficiently utilized for computations, but it is still available for testing purposes.

Utilization of the network-based startup procedure makes the clusters easy to control and modify by centralizing their configuration and operating system data in one place - the cluster server. It is assumed that the nodes will be used

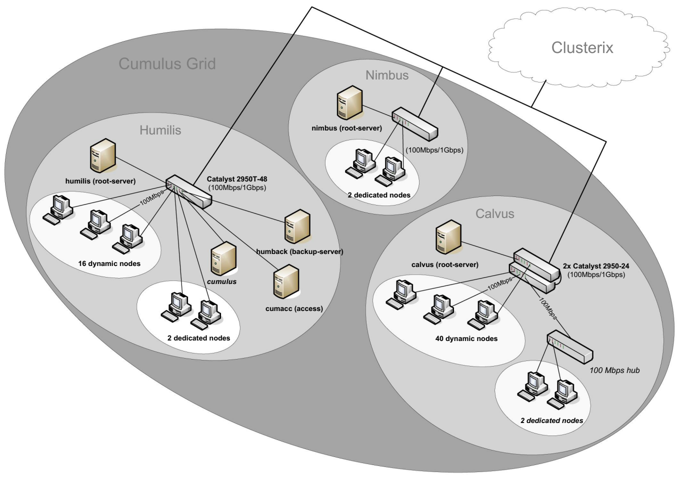


Fig. 1. Cumulus grid deployment structure

for CPU-intensive operations. If frequent data access is required, the necessary files can be placed in the scratch space available locally. A high server load can be observed only if a large number of nodes is simultaneously started, and only during the time of their bootup. The node startup time is slightly prolonged (in the Calvus cluster for 40 nodes it takes approximately 3 minutes) but as this procedure is typically initialized only once a day, this small inconvenience seems to be completely acceptable and justified. If the need to increase the computational power arises, a simple reconfiguration of the queuing systems makes it possible to logically move the chosen number of nodes from one cluster to another.

The computational power available to the users of the Cumulus grid is extended by its incorporation as a dynamic component of the Clusterix grid. The Cumulus clusters can be dynamically connected to the Clusterix infrastructure as separate entities via their root-servers. The procedure to connect the whole Cumulus grid to the Clusterix environment would require reconfiguration of the routing rules on all the clusters and is not currently automated by the Clusterix software.

3 Architectures of Static and Dynamic Clusters

Several conditions should be satisfied to provide the attractiveness of the connection procedure for its potential users [4]. First of all, the attachment and detachment procedures should be simple and automated. After the initial fulfillment of conditions necessary to provide integration of a dynamic cluster (installed software and initial verification of the cluster, performed only once), its operator should be able to attach and detach the cluster automatically by calling a single command. The attachment and detachment procedure was developed to implement the necessary steps.

The unified architecture of static clusters in the core has been tailored to implement this functionality in an efficient and secure manner. In particular, each dynamic cluster can be connected to a selected firewall/router (FW/R), whose public network interface is the only access point to the external network (Internet in particular). This solution allows for a balanced implementation of the attachment procedure, giving the possibility to choose the most appropriate static cluster to establish connection. The firewall/router controls the network traffic both inside the static cluster (computational nodes, access node), and on the edge with the external network. This allows for the concentration of the network traffic control to one place. But also, the lack of other services running on this machine makes it more difficult for compromising. Such a solution can create a performance bottleneck, so this architecture may be modified in the future if it becomes necessary.

As for now, in order to avoid situations where the FW/R throughput limits the whole Clusterix grid efficiency, an additional test was performed on assuring the optimal TCP/IP stack configuration for machines within the grid under real network conditions. It is - among other things - recommended to increase the minimum, default, and maximum TCP read and write buffers size. Also to disable caching ssthresh variable, to allow increases of the TCP buffer size by the receiver, and to assure that the system uses the BIC-TCP optimization algorithm. As a result of the optimizations above, the transmission speed between selected static clusters within the Clusterix network increased up to 500-600%. This order of magnitude refers to single connections - in case of multiple connections (the tests were performed for 20 connections) the throughput improvement is less considerable or even unchanged due to the link saturation or limitations introduced by the internal architecture of switches used within the grid infrastructure [12].

The internal architecture of dynamic clusters is not defined by the Clusterix project. The only requirement is the existence of the machine acting as the cluster access point for the connections to and from the Clusterix network. This machine is called a dynamic cluster firewall (DCF). In the case of the Cumulus grid the clusters' root-servers act as DCFs,

4 Dynamic Cluster Attachment

The attachment procedure is composed of two main parts. The first one - initial configuration - is performed once on the dynamic cluster firewall (DCF) and once on the firewall/router (FW/R) in the Clusterix network to prepare the environment and install software necessary for the attachment procedure. The second part - attachment and detachment procedure - is performed whenever the dynamic cluster is to be attached or detached from the Clusterix environment.

There are two methods developed for establishing tunnels within the Clusterix grid. For real purposes it is strictly recommended to build a virtual private network (VPN) between local networks of static and dynamic clusters, using the IPSec protocol. This protocol, despite its well known disadvantages, is said to be "the best IP Security protocol available at the moment" [13]. IPSec was also chosen because it operates on the OSI network layer level and therefore may be configured to protect all network traffic matching specific criteria (e.g. source and destination network address), regardless of the application that generated the traffic, or the protocol that is associated with the traffic. Unfortunately, the IPSec protocol is well known because of its high level of complexity as well. Thus unexpected errors may occur during configuration, or due to authentication issues. Having this in mind, it was decided to implement the possibility of establishing standard IP-over-IP tunnels. These tunnels do not offer any built-in security features like data encryption and therefore should not be used for any real purposes (the FW/R administrator is able to configure the system to reject tunnel establishing requests of this kind). However, they are a useful solution to determine whether an error is independent on the protocol used (e.g. occurs due to specific network conditions), or is strictly IPSec-specific.

During the attachment procedure, an encrypted tunnel is built to allow secure communication between the nodes in the dynamic cluster and in the Clusterix core. The encryption is necessary when the communication is performed utilizing untrusted links. In the Clusterix project, static and dynamic clusters use addresses from the 10.0.0.0/8 address pool. Packets encapsulation in the tunnel is thus required to connect dynamic clusters to the Clusterix core if the public internet connections are utilized.

4.1 Initial Configuration

The DCF communicates with the FW/R using the SSH protocol. To be able to execute the whole procedure automatically, passwordless authentication is utilized (instead of the password, a public and private key pair will be used). The protection level of the FW/R and DCF private keys becomes crucial - it is assumed that they are protected appropriately. On the FW/R for every dynamic cluster a pair of public and private keys should be generated. Using different keys for each dynamic cluster increases the whole system security level. Similarly on the DCF, a different key pair should be generated for every FW/R that can be utilized to establish connection to the Clusterix environment. An empty passphrase should be passed during the generating process, which will disable the password based authorization. The DCF should be able to log on to the dedicated account via ssh with no password provided.

This solution allows attaching and detaching a dynamic cluster in a fully automated way (except the initial, one-time, configuration procedure). On the other hand, it significantly decreases the system security level. One has to realize that if the DCF's private SSH key is compromised, the attacker is able to log on to the FW/R. In order to mitigate this risk, operating systems on both the FW/R and the DCF should be configured as introduced below.

On both the DCF and the FW/R, a new account should be added. This account is dedicated only to the dynamic clusters attachment. The account name should be (for administrative purposes) identical on every FW/R (e.g. "dynamic"). For security purposes a new group should be established, containing only this user. No shell should be available for the "dynamic" user. Instead in the /etc/passwd file, the full path to the dclctl configuration script should be given. This full path should also be appended to the /etc/shells file. Additionally, the password based login for the "dynamic" account should be disabled.

Preparing a tunnel and changing the local FW/R configuration requires invoking commands that should be available only for the superuser (like ip, ifconfig, iptables). It is absolutely undesired that the "dynamic" user has such privileges. The most convenient way is to use the sudo facility that allows us to give the specified user some higher privileges, but only for strictly defined activities within the system. However, it should be emphasized that this still reduces the overall system security level. Therefore other measures should be taken to secure the FW/R as strongly as possible. To enable no-password logging from the DCF, the appropriate public key should be copied from the DCF to every FW/R that should offer the DCF access to the Clusterix grid environment. Finally the dclctl configuration scripts should be installed in the "dynamic" account's home directory. On the FW/R side, all calls to the dclctl scripts are logged together with the parameters passed, and the calling host's IP address. The IP address is read from SSH environment variables, not from the network packet. The script is able to reconfigure IPSec parameters only for this specific IP address.

After the DCF SSH private key is compromised, the attacker is still able to send requests to the FW/R. Primarily, he or she may detach the dynamic cluster from the grid. However, if the private key is compromised, the DCF system is probably already in a significantly higher danger than disabling the possibility of operating together with Clusterix. It should be analyzed more thoroughly, what threats may arise from that for the FW/R.

Sending non-authorized requests to the FW/R should not harm that system, unless some additional conditions are fulfilled - e.g. the attacker discovers a serious vulnerability of an SSH application, which could be exploited, or someone (e.g. the superuser) logs on to the FW/R from the DCF, which is rather improbable.

Logging on to the "dynamic" account does not invoke any shell, but only causes running the dclctl script with passed parameters. Strings that do not match any script options are ignored. A successful attack on the FW/R would require breaking the FW/R security system itself. The attacker may try to introduce a DOS attack - however, a proper configuration of the FW/R operating system will disable a large number of consecutive logon attempts. The attacker might try to craft packets using the stolen private key from the attacked DCF and attempt to spoof the IP of other dynamic clusters in order to detach them from the grid in an unauthorized way. He or she would however have to force the SSH daemon on the FW/R to recognize the SSH connection as coming from another host than that it really originates from (e.g. to successfully attack the DNS Server of the FW/R). This is still possible, however it makes the whole attack more complicated and consumes more of the attacker's resources.

Finally, IPSec protocol properties have to be configured properly. It was decided to use IPSec-tools and racoon packages, originating from the KAME project - basically because of the easy configuration and the shell script administering facilities. The packages contain - among other things - two of the most important tools: setkey (for manipulating Security Policy Database and Security Association Database) and racoon daemon (for automatic key distribution). The racoon daemon is installed in "direct-configuration" mode in order to enable shell script administration.

A key issue to take under consideration is the authentication method. Within the Clusterix project, X.509-certificate based authentication is strongly preferred to the preshared key method. The latter is significantly less complicated to configure-and-use, but has got two meaningful disadvantages. First, it requires storing the keys in plain text in a file and second, it is unscalable - it requires one to have separate pairs of keys for each pair of communicating hosts - unless all keys are the same, which is in turn highly insecure.

The certificate based authentication avoids these disadvantages. Moreover, it is not necessary to implement a separate Public Key Infrastructure as the hosts running under Clusterix use Globus software, which requires obtaining host certificates from a trusted CA. Clusterix uses mainly Polish Grid CA for that purpose (although other CAs are acceptable as well). Both the DCF and FW/R have to obtain their host certificates as described on the Polish Grid CA web page. Unfortunately, in order to automatically authenticate the peer by the racoon daemon, the host's private key has to be generated with an empty password. Therefore the key must be protected on the operating system level as strongly as possible (only read permissions for the user). The private key file and the certificate should be put into a path defined within the racoon daemon configuration file (together with appropriately renamed files containing trusted CA certificate(s) and appropriate CRL(s)) [14]. Other important files, like configuration files of setkey and racoon tools etc., are given as restrictive permissions as possible, too.

4.2 Attachment and Detachment Procedure

Every dynamic cluster can be connected to one of the 12 static clusters in the Clusterix core. According to principles of IP addressing adopted in the Clusterix project [1], each static cluster possesses a certain subclass of private addresses from the 10.0.0.0/8 pool. Inside this subclass, 16 separate subclasses for dynamic clusters are distinguished. This means that a maximum of 16 dynamic clusters may be integrated with a given static cluster at the same time.

The attachment and detachment procedures are initialized by the DCF using SSH connection to the "dynamic" account on the FW/R (Fig. 2).

Logging in to the account results in the dclctl script invocation. The "up" or "down" parameter is passed to the script to determine if the connection should be established or closed. The communication between the FW/R and the DCF is performed using the standard input and error descriptors. The standard input is used to transfer necessary data (e.g. the address range allocated by

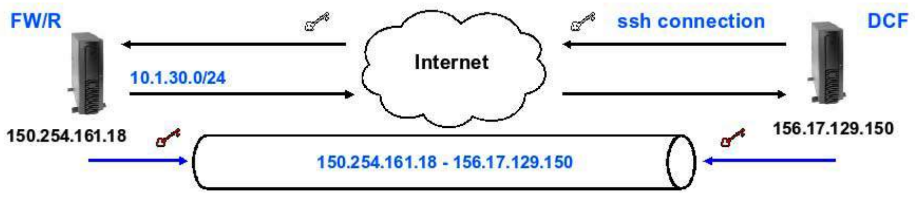


Fig. 2. DCF attachment

the FW/R to the DCF) and information if a particular part of the script was successfully accomplished on the remote side. The standard error is used to transfer additional debugging information. The only parameter that needs to be set up on the DCF side, is the address of the FW/R used for connection. On the FW/R side, the only configuration step that is required is the allocation of appropriate address range for every cluster that is to be connected.

In the first step of the attachment procedure (Fig. 3), the DCF connects to the selected FW/R by logging in via ssh to the "dynamic" account with an "up" parameter to indicate that the connection should be created. An additional parameter, "secure", determines which type of tunnel is to be created (however, this is not necessary provided that the administrators properly define their configuration files). If the DCF is allowed to log in, the dclctl script is invoked on the FW/R. The script determines the DCF's IP and creates a local end of the tunnel. The address range allocated on the FW/R (based on the entry in the dclctl configuration file associated with the determined caller's IP) for the particular DCF is written to the standard output. The FW/R creates IPSec security policy structures (SP) for the assigned address range, sets appropriate routing rules for that and finally opens suitable ports on the local firewall in order to be able to communicate via IPSec.

On the DCF, the assigned private class range is read by the dclctl script and the local end of the tunnel is prepared. Then the DCF performs similar activities (configuration of SP structures, routing and local firewall rules) on its side.

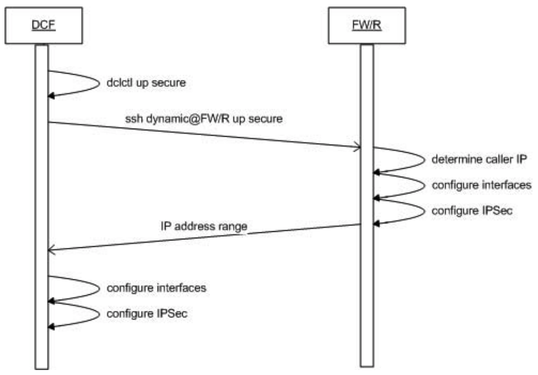


Fig. 3. Attachment procedure

The detachment procedure is also initialized by the DCF which logs in to the FW/R passing a "down" parameter. After the SSH connection is established, both ends of the tunnel are destroyed. So are the IPSec security policies, security associations (if any), routing and firewall rules. As a result, the dynamic cluster is detached from the Clusterix infrastructure.

The availability of dynamic clusters is monitored by the Grid Resource Management System [9]. Agents of the JMX-based Infrastructure Monitoring System JIMS [2], running on dynamic clusters, are responsible for reporting their availability when a connection succeeds. When the cluster is detached, the JIMS system recognizes the lack of this computational resource.

5 Possibilities of Future Improvements

The Clusterix is still an ongoing project but as the first prototype phase is already finished, it is possible to recapitulate obtained results and discuss the usability of the created environment. Some of the problems encountered are mentioned in the previous sections and the three most important ones will be shortly presented below.

The procedure of attaching dynamic clusters to the Clusterix core is suitable for connection of individual clusters. As local grid installations are becoming common [11], there also exists a need to offer easy and reliable ways of attaching this kind of installations. The cluster connection procedure developed in the project can be extended to accommodate the steps needed for a grid attachment. The required additional procedure logic developments are planned to part of the future Clusterix project phases.

Using the firewall/router for attaching static and dynamic clusters, as described in the section 3, can create performance bottlenecks. It causes a latency raise and lowers the throughput available. The impact of these problems on the performance depends on the particular application implementation. The choice of solution depends on the evaluation of the results of the ongoing tests. One of the proposed improvements is the utilization of hardware firewalls (instead of the FW/R).

The security considerations can lead to that the SSH based connection procedure is replaced by one utilizing a dedicated service for attaching and detaching dynamic clusters. It can be based for example on SSL/TLS protocol and the same X.509 certificates that are used for IPSec authentication. This issue will undergo further considerations.

6 Conclusions

The possibility to connect dynamic clusters to the Clusterix backbone opens the access to a shared environment with extraordinary computational power. The Cumulus grid created at Wroclaw University of Technology has already proved to be a beneficial scientific tool for research and educational purposes. Together with the access to the Clusterix environment it can be utilized both as a traditional cluster and also as a grid installation. Cumulus joins extendable computational power of dynamic machines, advantages of the persistent presence of available dedicated nodes and the computational power, delivered by the Polish national computational grid to execute applications and meta-applications. The developed connection procedure does not require complicated installation nor configuration changes. Utilizing only the SSH and IPSec protocols it is easy to configure, yet robust and secure. The installation was used in many projects and experiments. It served as a testbed for performance tests of the dependable information system developed in the European Community Framework Programme 6, project DeDiSys [3]. Cumulus was employed for performance evaluation of Kohonen network parallelization techniques [7], analysis of parallel program execution anomalies [8], analysis of virtualization techniques suitable for utilization in grid environments and in various other scientific projects.

Acknowledgments. This research has been partially supported by the Polish Ministry of Science and Information Society Technologies under grant 6T11 2003C/06098 "ClusteriX - National Cluster of Linux Systems" and the European Community Framework Programme 6, project DeDiSys, contract No 004152.

References

1. Belter, M., Kaminski, M., Krzywania, R.: Development of the IPv4 addressing scheme in the CLUSTERIX Project. Tech. Report, Poznan Supercomputing and Networking Center (2004)
2. Clusterix Project Homepage, `http://www.clusterix.pl`
3. The DeDiSys 6FP Project Homepage: `http://www.dedisys.org`
4. Frankowski, G., Balos, K., Wyrzykowski, R., Karczewski, K., Krzywania, R., Kosinski, J.: Intergrating Dynamic Clusters in Clusterix Environment. In: Proc. CGW'05, pp. 280–292, Cracow (November 2005)
5. Kopta, P., Kuczynski, T., Wyrzykowski, R.: Grid Access and User Interface in Clusterix Project. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 249–257. Springer, Heidelberg (2006)
6. Kwiatkowski, J., Pawlik, M., Wyrzykowski, R., Karczewski, K.: Cumulus - Dynamic Cluster Available under Clusterix. In: Proc. CGW'05, pp. 82–87, Cracow (November 2005)
7. Kwiatkowski, J., Pawlik, M., Markowska-Kaczmar, U., Konieczny, D.: Performance Evaluation of Different Kohenen Network Parallelization Techniques. In: IEEE Proc. PARELEC'06, pp. 331–336, Bialystok (2006)
8. Kwiatkowski, J., Pawlik, M., Konieczny, D.: Parallel Program Execution Anomalies. In: Proc. LaSCoG 2006, Wisla (November 2006)
9. Lawenda, M., Okon, M., Oleksiak, A., Ludwiczak, B., Piontek, T., Pukacki, J., Meyer, N., Nabrzyski, J., Stroinski, M.: Running Interactive Jobs in the Grid Environment. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 758–765. Springer, Heidelberg (2006)
10. Wyrzykowski, R., Meyer, N., Stroinski, M.: Concept and Implementation of Clusterix: National Cluster of Linux Systems. In: Proc. LCI Int. Conf. on Linux Clusters: The HPC Revolution 2005, Chapel-Hill, NC (2005)

11. Kauhaus, C., Fey, D.: Building Mini-Grid Environments with Virtual Private Networks: A Pragmatic Approach. In: IEEE Proc. PARELEC'06, pp. 111–115, Bialystok (2006)
12. Ferguson, N., Schneier, B.: A Cryptographic Evaluation of IPSec, `http://www.schneier.com/paper-ipsec.pdf`
13. Binczewski, M., Bogacki, W., Krzywania, R., Weinert, A.: TCP protocol tuning in the Clusterix network. Tech. Report, Poznan Supercomputing and Networking Center (2006)
14. Spenneberg, R.: IPSec HOWTO, `http://www.ipsec-howto.org/ipsec-howto.pdf`

Agent-Based Societies for the Sharing, Brokerage and Allocation of Grid Resources

Gabriele Pierantoni, Brian Coghlan, and Eamonn Kenny

Trinity College Dublin,
Computer Science Department, Rm-005, CALab1, INS building, TCD,
Dublin 2, Ireland
pierantg@cs.tcd.ie

Abstract. One of the definitions of economy as *"the administration of the concerns and resources of any community or establishment with a view to orderly conduct and productiveness"*[2] appears almost identical to the definition of the problem of resource allocation in grid computing. From an economic perspective, the grid itself coincided with the creation of an economy in the very moment that it enabled the exchange or sharing of resources between different owners. This consideration led us to envisage a very high-level resource brokerage architecture based on grid agents capable of implementing different social and economic interactions in grids.

1 Introduction

The complexity of the design of a grid resource brokerage system lies in the functionalities that such a system should implement and in the characteristics that it should have. These are:

– Ability to cope with the different ownerships of resources and jobs.
– Ability to define what optimal resource selection means for each actor (resource owners, job owners, global welfare of the grid, optimal occupation of the resources, etc) and to offer different solutions to different definitions of optimality.
– Ability to cope with the highly dynamic characteristics of grid computing, offering flexibility, scalability and fault tolerance.
– Ability to interface with existing and foreseen grid solutions.
– Ability to cope with different scenarios, ranging from pure cooperation to pure competition.
– Capability to be implemented in small steps, allowing the incremental composition of simple brokerage services into more complex ones.

The various solutions[19],[12] envisaged by the scientific community to tackle the complexities of resource brokerage and allocation in grids encompass genetic algorithms, simulated annealing and hybrid solutions[5]. In addition to these, there are economy-based approaches that range from pure competitive liberalism to pure cooperation, from uncontrolled markets[18][11], to centralized controlled economies[10][9]. Nonetheless, all of these enable in effect low-level (the majority) or mid-level (the minority) resource brokerage.

Below, we describe a very high-level resource brokerage architecture based on grid agents capable of implementing different social and economic interactions in grids.

2 The Socio-economic Paradigm

As in human societies and economies, grids are a way to allow actors to cooperate and compete to try to achieve their goals. These goals may be personal or collective. The practice of using an economic and social approach is justified by the fact that grids are a human invention that recreates in the digital environment what has been achieved over many centuries with economies and societies: the possibility to cooperate and compete among and with each other to pursue collective and personal goals. In reality humans arrange in fractal structures called societies that are characterized by sets of rules that all the participants should follow (but often don't).

The rules that characterize an association may involve the exchange of goods (these we will call economic societies) or may bind together actors willing to cooperate for a certain goal voluntarily and without expecting an economic revenue (we will call these empathy-based societies). Actors may expose one or both behaviors depending on the circumstances. A resource owner may be willing to offer part of his resource for free if they are used for certain goals, while he may ask for payment if they are used for other goals, or, finally, he may definitely not like to have his resource used to pursue goals he dislikes or condemns. These relations, driven by a sense of empathy, cannot be modeled in economic terms. On the other hand we can find purely economic behaviors where actors relate to each other to buy and sell goods and services in the pursuit of their own interest.

2.1 Grid Services as Production

We describe the resource allocation using the *microeconomic*[13] concept of a *production chain* that goes from the basic goods, or *factors* to *intermediate commodities* and, in the end, to the final job, or *produced commodity.* This conception of grid jobs as supply chains is illustrated in Fig. 1.

We divide the factors into three main categories: primary factors having no semantic value such as CPU time, memory storage and network connection offered directly (as an example through a scheduler), those having semantic value (i.e. data), and finally existing grid resources (job submission, information systems and file management services). These are, in reality, produced commodities but we model them as factors as their internal production chain is not modelled in economic terms, although the price at which they are sold may reflect the consumptions of primary factors.

The distinction between data and primary factors, even though not directly used in this first approach to the problem, may turn out to be quite important

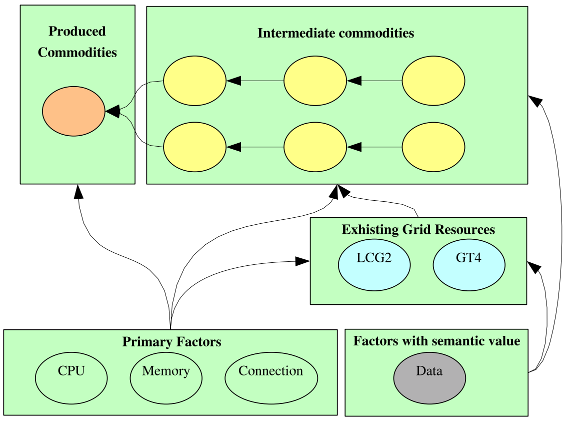


Fig. 1. Representation of a grid job as a microeconomic supply chain

as the real economic value of a data set becomes represented by its meaning or semantic.

2.2 Endowments, Demands and Supplies

At each of the production steps an actor (either a human or a program) assembles factors and, possibly, intermediate commodities to create a produced commodity. If at any given time an actor has not enough factors to produce the commodities he wants, he may engage in social or economic transactions to obtain the commodities he needs. To model this behaviour we borrow once more concepts and terminology from microeconomics[13] with which we define the *Endowment* as the set of commodities owned by the actor, the *Needs* as the set of commodities needed by the actor, the *Supplies* as the set of commodities that the actor has in abundance and can be traded or donated, the *Demands* as the set of commodities that the actor cannot cover with its endowment and the *Satisfied Needs* as the set of commodities that the actor can cover with its endowment.

An actor owns an Endowment set of commodities and faces a Needs set of commodities. Demands and Supplies are obtained from Endowment and Needs as illustrated in Fig. 2. It is worth noting that the assessment of the factors needed to complete a certain job is a crucial part of the process and that it is a topic of active research in the grid community. In particular, the assessment of the computational power required can be achieved in different ways, such as:

– User specification: In this simplest way, the user has the duty to provide a description of the minimum set of resources necessary.
– Simple statistical analysis: Particular types of jobs, such as parameter sweep applications, allow a relatively easy way to assess their needs[8],[7],[17], [18],[11]

– Complex statistical analysis: Research, such as that described in[4] is being undertaken to assess the computational requirements of a broader variety of jobs.

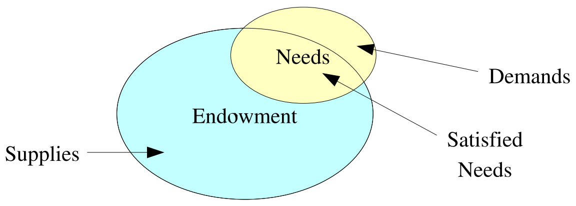


Fig. 2. Demands and Supplies

In this case, the Endowment and the Needs sets overlap partially. Where they overlap, that part of the Needs has been satisfied by the Endowment. The actor will be able to trade or donate the remaining part of its Endowment, and will also need to trade for the unsatisfied part of the set of its Needs. Formally: Supplies = Endowment $\cap$ $\overline{\text{Needs}}$, Demands = Needs $\cap$ $\overline{\text{Endowment}}$ and Satisfied Needs = Endowment $\cap$ Needs.

2.3 Exchange and Other Social Interactions

Thus if an actor owns an Endowment set of resources and faces a Needs set of resources, the result is two disjoint sets: Demands and Supplies, that will be traded, donated or acquired in the societies that the actor is joining. In order to do this, actors engage in relations, both competitive and empathic, to fulfill their Needs sets. These types of relations can be:

– Co-operative relations, such relations consist of the access to resources without the need of a balancing transaction representing a payment. These relations can be of the following nature: *donations*, *lending and borrowing*, *Keynesian investments*[1] and *common goods.*
– Non-monetary economic relations such as bartering.
– Monetary economic relations that involve payments and monetary transactions such as posted price models, auctions and calls for tender.

3 Architecture

Our resource brokerage architecture[3] follows naturally the social and economic paradigm used to analyze the problem, and it is based on the ongoing research

[1] We term *Keynesian Investment*, in honour of the economist John Maynard Keynes, a relationship where an Institution invests in Grid resources binding their use, partially or fully, to a certain user or use

on grid interoperability[16][15] and a concept of *Social Grid Agents* [14] to tackle the issues of resource brokering. The design architecture is based on the following assumptions:

- Grid resources can be accessed only through a service.
- Resources and services have a owner.
- Owners are implemented by grid agents.
- Grid agents are identifiable by their grid certificates.
- The mechanism of delegation allows access to resources by actors that do not own them.
- Agents can communicate with each other using one or more communication layers.

Another insight underlying the architectural design is that these grid agents are at the convergence of three flows of information: the production process, the policies information and the access rights or ownership. As said previously, the production process produces commodities under the rules described by the policies to which the grid agent is bound by using the factors to which it is granted access through the ownership information flow. As production is bound to the other two information flows, it seems natural to divide the architecture into two layers: a production layer implemented by the *Production Grid Agent*, and a surrounding social layer implemented by the *Social Grid Agent*.

3.1 Production Grid Agents

Production Grid Agents are composed of two main entities. Firstly the *Grid Body* consists of existing grid services surrounded by a GT4 service that enables it to be used by the agents. The link between the GT4[1] service and the existing Grid Service usually consists of API invocations, but can also be performed with system calls if an API is lacking. Secondly a *Java Brain* implements the production behaviour with which the agent controls the existing grid service. Production Grid Agents have the following characteristics:

- They hold a description of the service they produce.
- They always consume factors.
- They produce commodities.
- They have a *purse* of factors that they are allowed to use by their social layer.
- They hold the information on their consumption of factors.
- They comply with the rules imposed by their social layer.

In order to comply with these requirement we represent the Production Grid Agent as in Fig. 3. Among the processes that compose the Java Brain of a Production Grid Agent, the most important are the following:

- A Policy Interface that acquires the information about the rules that are to be followed.

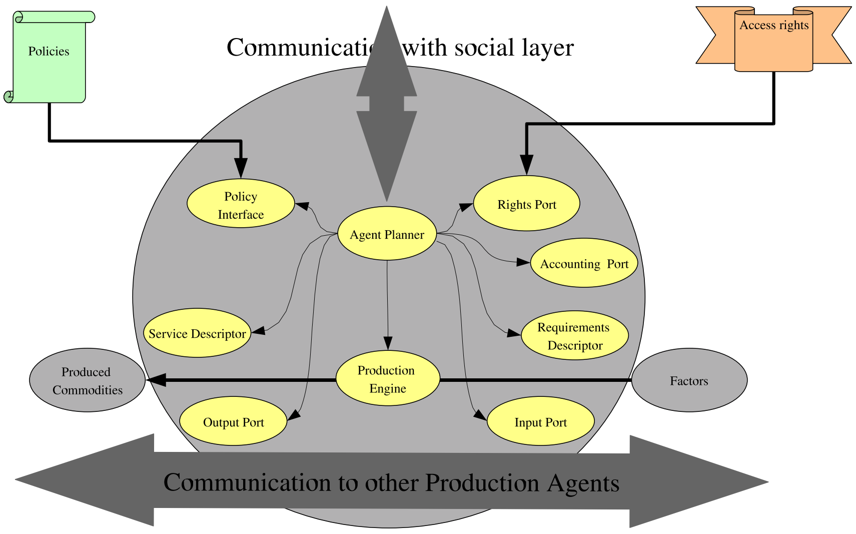


Fig. 3. The Production Agent

- A Rights Port that acquires the information about the factors that the Production Agent can access.
- An Output Port through which the completed service will be delivered to other production agents.
- A Service Descriptor that holds the description of the service.
- A Requirements Descriptor that holds information regarding factor needs.
- An Accounting Port that holds the information regarding factor usage.
- A Production Engine that actually performs the production process by controlling the various Grid Bodies.
- An Agent Planner that will coordinate the work of the Production Engine to provide the Produced Service.

The agent we have described, is capable of producing a service if it is given a description and access to enough commodities. They can be arranged in production chains where the factors of one are the produced commodities of another. Nevertheless, from a social perspective, they still are *islands in the ocean* as they lack the ability to negotiate the process necessary to interface with each other. To use a human comparison, Production Grid Agents are workers that are not capable of self organization. To implement this functionality, an additional layer is built around the Production Grid Agent that provides social capabilities. This layer transforms a Production Grid Agent into a Social Grid Agent.

3.2 Social Grid Agents

This layer is responsible for the social and economic behaviour of the agents. Here Supplies and Demands are computed and social and economic transaction

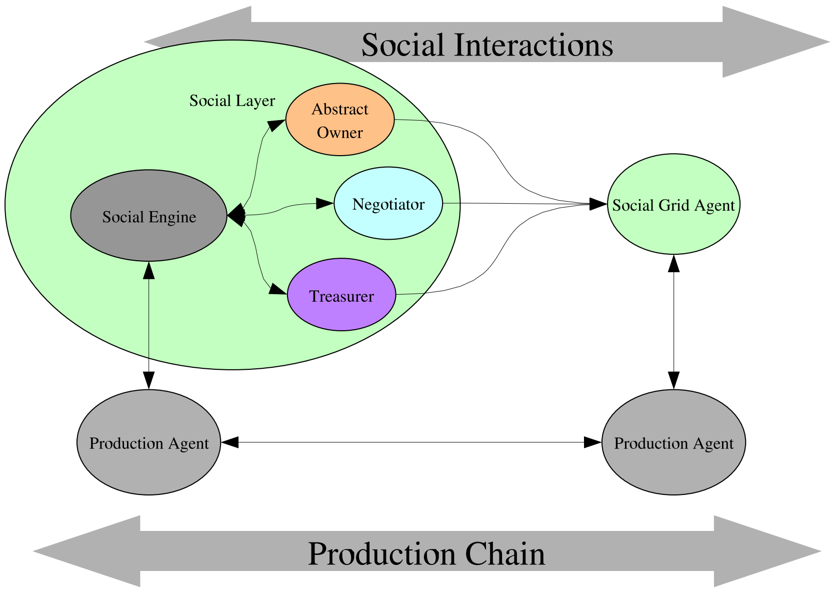


Fig. 4. The Social Grid Agent

are managed to meet the Needs of the agent. The functionalities offered by this additional layer are: social capabilities, ownership management, negotiating capabilities and treasurership.

As stated above, Social Grid Agents are Production Grid Agents with a social layer built around them. Their architecture is presented in Fig. 4. This social layer allows the creation of agents capable of engaging in social and economic transactions regarding the production agent's activities and the primary factors they have ownership of. This approach allows a more homogeneous description of every commodity as they are all accessed through services owned by a Social Grid Agent. As an example an agent that offers raw computational power will offer access to its server factory as a service owned by a Social Grid Agent that may not be the real owner of the resources. This approach is inspired by the concept of abstract ownership[6]. On the other hand, a more complex service, such as that offered by an entire grid with its own resource broker, information system and pool of resources, might be offered through a different service owned by a Social Grid Agent that is (or is entitled by) the real grid owner, to offer the grid resources.

3.3 Grid Societies

The social capabilities of the Social Grid Agents can also be used to build Grid Societies that, although encompassing many grid agents, offer the same *"social* *interface"* to other grid agents. As shown in Fig. 5, Grid Societies have a topology of services that is similar to the Social Grid Agents. It is worth noticing that the Production Agent may or may not be present, depending on whether the society is a production society such as a company, or a gathering society such as a tribe or a market.

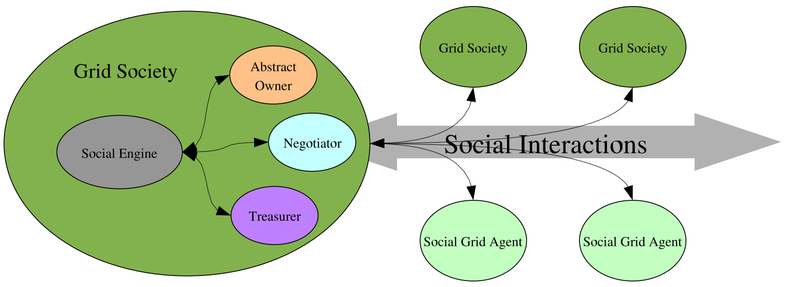


Fig. 5. A Grid Society

Remember that the purpose of the social layer is to bring together different grid agents to build composite agents capable of offering more complex services. The resulting entities are called Grid Societies and can be economy-based or empathy-based, the difference residing in the presence or absence of economic exchanges. Grid Societies will expose the same interfaces as any other Social Grid Agent to allow the creation of complex architectures of societies and agents.

The social capabilities of the Social Grid Agents allow for a great variety of social structures. Fig. 6 illustrates three different ways in which Social Grid Agents can interact.

- Individual Social Interactions (a), where Social Grid Agents interact among each other, joining in a production process or engaging in exchange of their endowments.
- Exchange-based societies (b), where Social Grid Agents gather in a society to engage in exchanging relations. These social structures can model markets, tribes and volunteer-based sharing structures.
- Production-based societies (c), where Social Grid Agents gather in a society to engage in production relations. These social structures can model companies.

4 Conclusion

Why create such a resource brokerage architecture? Because it can incorporate many kinds of realistic market actors and their interactions, because these may coexist just as they do in the real world, and because they may be composed and decomposed in the same bewildering array of combinations as is evident in the

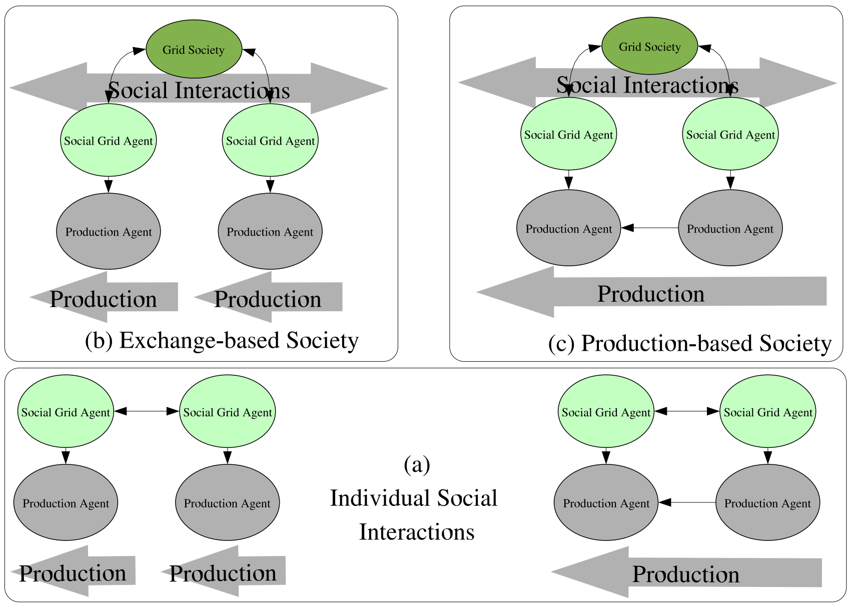


Fig. 6. Different Grid Societies

real world. In this way, true grid resource markets may be constructed, where birth, death, marriage, profits, losses, theft and charity may all be represented. Clearly this is an ambitious, and possibly even unattainable, goal, requiring considerable further research. Currently this research is being conducted as two parallel activities: A top-down approach involving a more detailed analysis of the functionalities offered by the current actors in grid computing. And a bottom-up approach involving the implementation of experimental prototypes of the key parts of the proposed architecture, in order to test weaknesses in the hypothesis. Eventually, this is intended to yield a first prototype with economic and empathic functionalities that will serve both as a brokerage agent and as a testbed for further analysis of grid economics.

References

1. Globus toolkit web page, `http://www.globus.org/toolkit`
2. Online Oxford English Dictionary, `http://dictionary.oed.com`
3. Social Grid Agents Sourceforge project page, `http://sourceforge.net/projects/socialgridagent/`
4. Bunn, J., Cavanaugh, R., van Lingen, F., McClatchey, R., Atif Mehmood, M., Newman, H., Steenberg, C., Thomas, M., Ali, A., Anjum, A., Willers, I.: Predicting the resource requirements of a job submission. In: Proceedings of Computing for High Energy Physics, Interlaken, Switzerland (2004)

5. Buyya, R., Abraham, A., Nath, B.: Nature heuristics for scheduling jobs on computational grids. In: The 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000), Cochin, India (December 2000)
6. Buyya, R.: Economic-based Distributed Resource Management and Scheduling for Grid Computing (May 2002)
7. Buyya, R.: Economic-based Distributed Resource Management and Scheduling for Grid Computing. PhD thesis
8. Buyya, R., Abramson, D., Giddy, J.: An economy driven resource management architecture for global computational power grids. In: The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, USA (June 26-29, 2000)
9. Elmroth, E., Gardfjäll, P.: Design and Evaluation of a Decentralized System for Grid-wide Fairshare Scheduling. In: First IEEE Conference on e-Science and Grid Computing (May 2005)
10. Gardfjäll, P., Elmroth, E., Johnsson, L., Mulmo, O., Sandholm, T.: Scalable Grid-wide Capacity Allocation with the SweGrid Accounting System (sgas). Draft version (2006) (Final version to be submitted for journal publication)
11. Lotia, N., Hayat, Z., Sherwani, J., Ali, N., Buyya, R.: Libra: An economy-driven job scheduling system for clusters. Softw. Pract. Exper. 34(6), 573–590 (2004)
12. Lee, C., Lindell, B., Nahrsted, K., Foster, J., Kasselman, C., Roy, A.: A distributed resource management architecture that supports advanced reservations and co-allocation. In: Seventh International Workshop on Quality of Service (1999)
13. Mukherji, A.: Walrasian and Non-Walrasian Equilibria: An Introduction to General Equilibrium Analysis. Oxford University Press, Oxford (1990)
14. Pierantoni, G., Kenny, E., Coghlan, B.: An Architecture Based on a Social-Economic Approach for Flexible Grid Resource Allocation. In: Cracow Grid Workshop (CGW05), Cracow, Poland (November 2005)
15. Pierantoni, G., Kenny, E., Coghlan, B., Lyttleton, O., O'Callaghan, D., Quigley, G.: Interoperability using a Metagrid Architecture. In: HDPC 06, Paris, France (June 2006)
16. Pierantoni, G., Lyttleton, O., O'Callaghan, D., Quigley, G., Kenny, E., Coghlan, B.: Multi-Grid and Multi-VO Job Submission based on a Unified Computational Model. In: Cracow Grid Workshop (CGW05), Kracow, Poland (November 2005)
17. Abramson, D., Buyya, R., Venogopal, S.: The grid economy. Special Issue on Grid Computing Environments, The Journal of Concurrency and Computation: Practice and Experience (CCPE) 93, 698–714 (2005)
18. Giddy, J., Buyya, R., Abramson, D., Stockinger, H.: Economic models for management of resources in peer-to-peer and grid computing. Technical Track on Commercial Applications for High-Performance Computing, SPIE International Symposium on The Convergence of Information Technologies and Communications (ITCom 2001), Denver, Colorado, USA (August 2001)
19. Chapin, S., Buyya, R., DiNucci, D.: Architectural models for resource management in the grid. In: Buyya, R., Baker, M. (eds.) GRID 2000. LNCS, vol. 1971, Springer, Heidelberg (2000)

Opus^{IB} – Grid Enabled Opteron Cluster with InfiniBand Interconnect

Olaf Schneider, Frank Schmitz, Ivan Kondov, and Thomas Brandel

Forschungszentrum Karlsruhe,
Institut für Wissenschaftliches Rechnen,
Herrmann-von-Helmholtz-Platz 1,
D-76344 Eggenstein-Leopoldshafen, Germany
{thomas.brandel,ivan.kondov,frank.schmitz, olaf.schneider}@iwr.fzk.de

Abstract. Opus^{IB} is an Opteron based cluster system with InfiniBand interconnect. Grid middleware provide the integration into CampusGrid and D-Grid projects. Mentionable details of hardware and software equipment as well as configuration of the cluster will be introduced. Performance measurements show that InfiniBand is not only well suited for message-passing based parallel applications but also competitive as transport layer for data access in shared cluster file systems or high throughput computing.

Keywords: Cluster, Grid, Middleware, File system, SAN, InfiniBand.

1 Introduction

Cluster systems with fast interconnects like Myrinet, Quadrics or InfiniBand become more and more important in the realm of high performance computing (HPC). The Institute for Scientific Computing at the Forschungszentrum Karlsruhe was active in adopting and testing InfiniBand technology very early. We started with a small test system in 2002, followed by a Xeon based system with 13 nodes called IWarp in 2003. The next generation of InfiniBand cluster is Opus^{IB}, which we describe in this paper.

In the following section we briefly look at the project CampusGrid in which most activities reported here are embedded. The key facts about Opus^{IB}'s hardware and software are collected in several subsections of Sect. 3. Thereafter, in Sect. 4, we comment on some measurements which prove the achievable performance with InfiniBand. We shall conclude the paper with a short survey on Opus^{IB} as part of the D-Grid infrastructure.

2 The CampusGrid Project

The R&D project CampusGrid [1,2] was initiated at the Forschungszentrum 2004 with the aim to construct and build a heterogeneous network of resources for computing, data and storage. Additionally, the project gives users the opportunity to run their applications in such a heterogeneous environment. Grid technologies were selected as a state-of-the-art method to achieve these goals. The use of standard Grid middleware in our local infrastructure is advantageous, because we enable the scientists of the Forschungszentrum to smoothly enter the global Grid.

The project started with a testbed for evaluation of middleware and other components. While the initial testbed was small, it already comprised all kinds of resources in our heterogeneous IT environment: clusters, SMP servers, and vector processors as well as managed storage (SAN). During project progress more and more production systems shall be integrated in the CampusGrid environment. In order to do so we need a clear and smooth migration path from our classical HPC environment into the new Grid-based infrastructure. Thus, in the design of the CampusGrid architecture we need to take care of many boundary conditions we can not (easily) change in our project, e. g. central user administration via Active Directory Services (ADS).

The cluster Opus^{IB} started as part of the CampusGrid testbed and it is now growing into a fully productive system.

3 Hardware and Software of Opus^{IB}

3.1 Overview

The name Opus^{IB} is an abbreviation for Opteron cluster with InfiniBand. As the name implies, the cluster is assembled of dual processor nodes with Opteron 248 processors and the high-performance networking fabric is an InfiniBand switch (InfinIO9000 by SilverStorm). All worker nodes and most cluster nodes run CERN Scientific Linux as operating system (64bit version).

At the time of writing this, there are 64 worker nodes with 128 CPUs in total and an aggregated memory of about 350 GB. All worker nodes and the switch fabric are build into water cooled cabinets by Knürr. This technology was originally developed for the GridKa [3] cluster.

3.2 InfiniBand

InfiniBand (IB) is a general purpose network and protocol usable for different higher level protocols (TCP/IP, FibreChannel/SCSI, MPI, RFIO/IB) [4]. In contrast to existing interconnect devices that employ a shared-bus I/O architecture, InfiniBand is channel-based, i. e., there is a dedicated path from one communication partner to the other. Links can be aggregated, which is standardized for 4 and 12 links called 4X and 12X. We use 4X in our installation, that means 1 GB/s usable bandwidth (in each direction). FibreChannel (FC) bridges plugged into the IB switch enable us to directly connect storage devices in the Storage Area Network (SAN) to the cluster nodes. Thus it is not necessary to equip each node with a FC host bus adapter.

As an off-the-shelf high-speed interconnect InfiniBand is a direct competitor of technologies like Myrinet and Quadrics. Our decision to use InfiniBand in the cluster was mainly due to the positive experiences in recent projects (cf. [5]).

3.3 Running Two Batch Schedulers Concurrently

A peculiarity of the Opus^{IB} cluster is that all worker nodes are managed by two different job schedulers concurrently. At one hand, we have the OpenPBS successor TORQUE [6] together with the MAUI scheduler [7]. On the other hand, there is a mixed LoadLeveler cluster which consists of the Opus^{IB} nodes, several PowerPC blades and some other single machines. Recently we added our AIX production system (pSeries 655 and 630) to this mixed cluster.

The reasons for running these two batch systems concurrently are numerous. First the history: As we started assembling the cluster we chose TORQUE for a couple of reasons – it is Open Source, compatible to OpenPBS, but more stable. At the point when IBM provided first LoadLeveler with the mixed cluster option, we decided to try it – because of curiosity. Shortly before we had got some PowerPC blades which could serve as AIX nodes in our testing environment. The Linux part of the testbed was just Opus^{IB}. At that point, the cluster was still in a quite experimental mode of operation. Thus, two batch systems did not cause any problems but were sometimes useful for tests. This configuration survived the gradual change of the cluster into productive operation. Currently, LoadLeveler works very well and is used for the majority of user jobs submitted in the classical way on the command line. On the other hand, Grid middleware supports more often TORQUE/PBS than LoadLeveler. Moreover, the combination with MAUI scheduler is quite popular in the Grid community. Thus, TORQUE serves as a kind of reference system when using Grid Middleware. A third reason is, that we want to stay independent of commercial software vendors as far as possible. That means, an Open Source solution should be available at least as fall-back.

Running two job managers concurrently without noting of each other of course holds the danger to overload nodes with too many jobs. In practice, however, we noticed that such problems occur less often than expected. The reason was probably that both schedulers take the actual workload on a node into account, when making the scheduling decision. For Maui scheduler this behavior is triggered by setting the configuration parameter MAXLOAD. Hence, Maui marks a node busy if the load exceeds MAXLOAD. The exact value needs some tuning – we used values between 1.1 and 2.5. LoadLeveler prefers the node with the lowest load by default.

If overcommitment occurs, it is always very harmful, especially if it affects the workload balance of a parallel job (since a single task is slowed down compared to all other tasks). Recently we tried to solve this kind of problems by adding prolog and epilog scripts to each job. After submission a job waits in a queue until a matching resource is available. Right before job startup the scheduler, say LoadLeveler, runs a prolog script, to which the list of processors (nodes) occupied by the job is passed (via the variable LOADL_PROCESSOR_LIST). The prolog script utilize this information to decrease the number of job slots in the list of available resources at the other scheduler (i. e. TORQUE). After the job has finished, the slot number is increased in the epilog script. Thus, we reconfigure dynamically the resources of the second scheduler if a job is started by the fist scheduler, and vice versa.

3.4 Cluster Management Using Quattor

Automated installation and management of the cluster nodes is one of the key requirements for operating a cluster economically. At Opus^{IB} this is done using Quattor, a software developed at CERN [8]. Some features of Quattor are:

– automated installation and configuration
– software repository (access via http)
– Configuration Data Base (CDB) server
– template language to describe setup
– Node Configuration Manager (NCM) using information from CDB
– templates to describe software setup and configuration

The open standards, on which the Quattor components are based, allow easy customization and addition of new functionality. For instance, creating new node configuration components is essentially writing a Perl module. In addition, the hierarchical CDB structure provides a good overview of cluster and node properties. Addition of new hardware or changing installed software on existing hardware is facilitated tremendously by Quattor – the process takes as long as several minutes.

3.5 Kerberos Authentication and Active Directory

For the CampusGrid project it was decided to use Kerberos 5 authentication with the Active Directory Server as Key Distribution Center (KDC). Thus all Opus^{IB} nodes are equipped with Kerberos clients and a Kerberos enabled version of OpenSSH. As a work-around for the missing Kerberos support in the job scheduling systems (PBS, LoadLeveler) we use our own modified version of PSR [9], which incorporates Kerberos 5 support.

While Kerberos is responsible for authentication, the identity information stored in the passwd file still needs to be transferred to each node. For this purpose we use a newly developed Quattor component, which retrieves the necessary data via LDAP from the Active Directory and then distributes it to the cluster by the usual Quattor update mechanism.

3.6 StorNext File System

SNFS is a commercial product by ADIC [10]. It has several features which support our goal to provide seamless access to a heterogeneous collection of HPC and other resources:

- Native clients are available for many operating systems (Windows, AIX, Linux, Solaris, IRIX).
- The metadata server does not require proprietary hardware.
- Active Directory integration is part of the current version.
- We get very good performance results in our evaluation (cf. Sect. 4).
- Installation procedure and management is simpler than in competing products.

A drawback is that file system volumes can not be enlarged during normal operation without a maintenance period.

3.7 Globus Toolkit 4

In the project CampusGrid we decided to use Globus Toolkit 4 (GT4) as basic middleware. For an overview of features and concepts of GT4 we refer to Forster [11] and the documentation of the software [12]. The current configuration for the Opus^{IB} cluster is depicted in Fig. 1. We use the usual Grid Security Infrastructure (GSI), the only extension is a component to update the grid mapfile with data from the Active Directory.

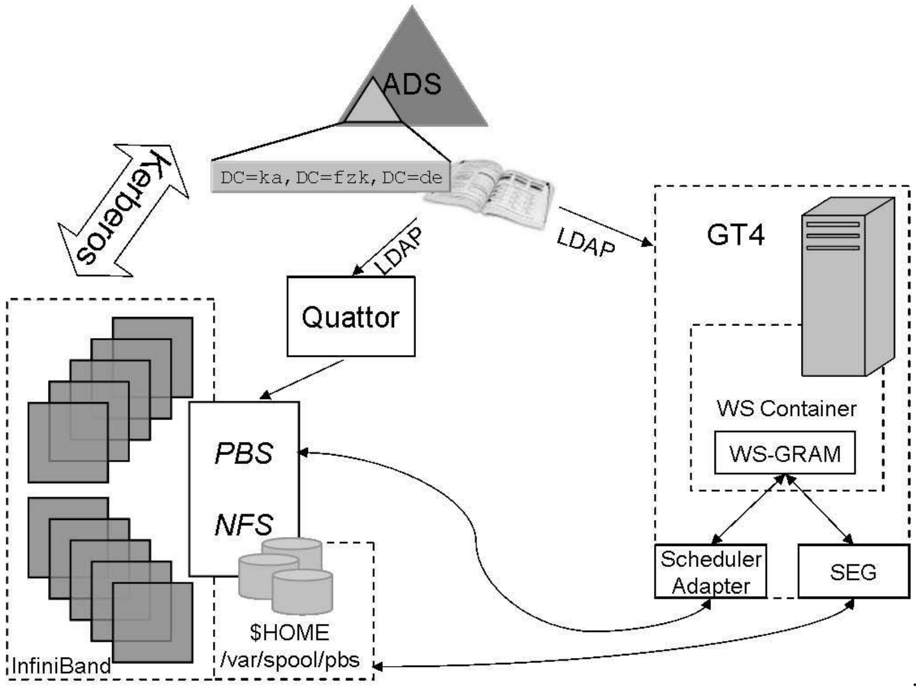


Fig. 1. WS-GRAM for job submission on Opus^{IB}, with identity management using Active Directory Server (ADS)

For LoadLeveler jobs there is an additional WS-GRAM adapter and a scheduler event generator (SEG). Actually there are two of them – one running on the GT4 server mentioned above and a second running on a PowerPC machine with AIX. The latter was installed to test GT4 with AIX.

The cluster monitoring data gathered by Ganglia [13] are published in MDS4, the GT4 monitoring and resource discovery system (using the Ganglia Information Provider shipped with Globus).

4 Benchmarks

4.1 Data Throughput

Right from the start our objective in evaluating InfiniBand technology was not only the fast inter-node connection for message-passing based parallel applications. We also focused on the data throughput in the whole cluster. The large data rates achievable with InfiniBand are appropriate for accessing large amounts of data from each node concurrently via file system or other protocols like RFIO. Preliminary results of our studies can be found in [5]. Later on we successfully tested the access to storage devices via an InfiniBand-FibreChannel bridge together with a Cisco MDS9506 SAN director. These tests were part of a comprehensive evaluation process with the aim to find a SAN based shared file system solution with native clients for different architectures. Such a heterogeneous HPC file system is intended as a core component of the CampusGrid infrastructure. A comparison of StorNextFS (cf. Sect. 3.6) with two competitors (SAN-FS by IBM and CXFS by SGI) is given in Table 1.

Table 1. Write throughput in MB/s for different file systems and client hardware, varying file sizes and fixed record size of 8 MB

	SunFire with IB		p630 (AIX)		
	SNFS	CXFS	SNFS	CXFS	SAN-FS
64 MB	177	93	70	50	59
1 GB	176	91	70	49	53
4 GB	175	96	73	49	52

The measurements are done using the benchmark software IOzone [14]. Write performance was always measured such that the client waits for the controller to confirm the data transfer before the next block is written. This corresponds to the behavior of NFS mounted with option 'sync'. Due to compatibility issues it was not possible to install the SAN-FS client on our SunFire nodes with InfiniBand and Opteron processors. The reported values rely on sequentially written files of various size – 128 kB to 4 GB, doubling size in each step – while the record size goes from 64 kB to 8 MB. Typically a monotonic increase of data throughput

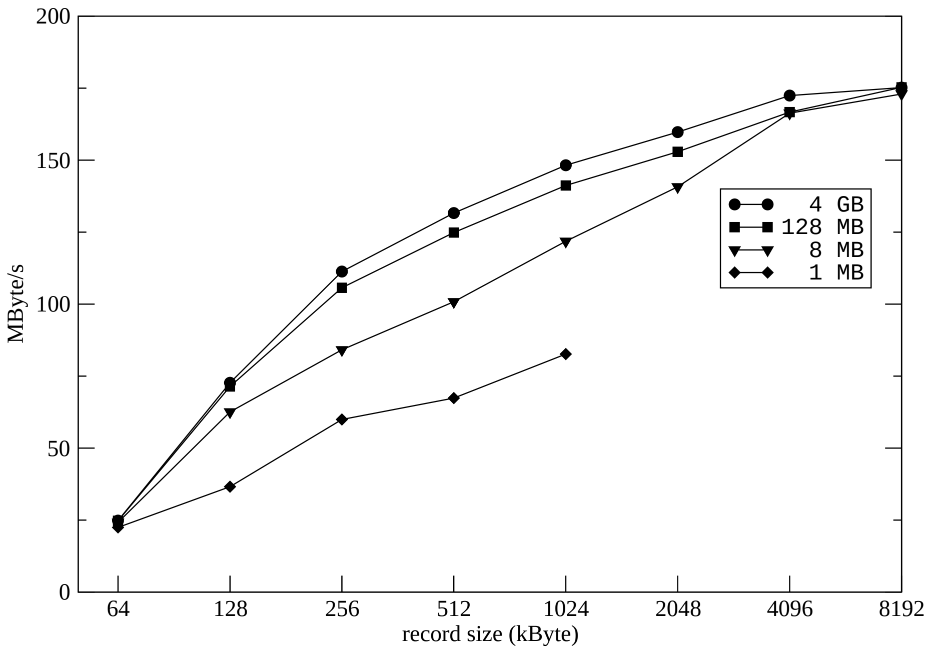


Fig. 2. Write performance of SNFS on SunFire with InfiniBand using IB-FC bridge

with growing file and record size can be observed. This behavior is depicted in Fig. 2.

All measurements are done using a disk-storage system by Data Direct Networks (S2A 8500). The connection between SAN fabric and IB-FC bridge or, accordingly, the p630, was a 2 Gigabit FibreChannel link. Thus, the overall bandwidth from the cluster to the storage is limited to the capacity of this link. For file system access with considerable overhead we can not expect more than about 180 MB/s (or, correspondingly, 1.5 GBit/s). Measurements [15] show that SNFS behave well if more than one client access the same file system.

4.2 Parallel Computing

Beside serial applications with high data throughput, the application mix on Opus^{IB} contains typical MPI applications from several scientific domains (climate simulation, CFD, solid state physics and quantum chemistry). Parallel floating-point performance which is relevant for the latter applications was benchmarked using HPL [16]. It was compiled and linked on Opus^{IB} using all available C compilers (GCC, Intel, PGI) and the libraries MVAPICH [17] and ATLAS [18]. The latter was compiled with the architecture defaults from the vendor. The tests were performed on up to 18 nodes.

Figure 3 shows the measured performance. It scales linearly with the number of processors. The performance per node is quite constant – between 3.7 and 3.3 Gflops, which corresponds to about 80% of the peak performance.

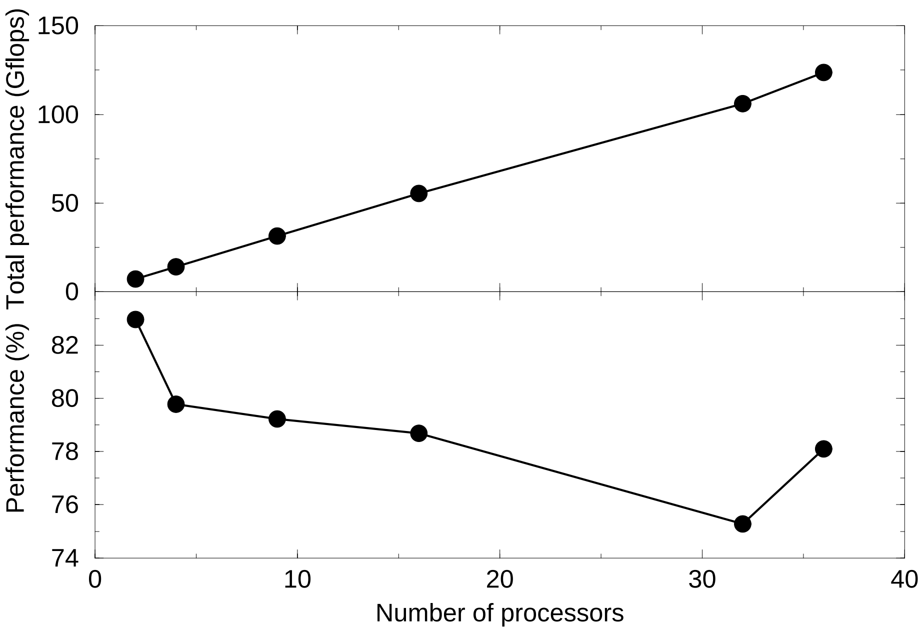


Fig. 3. Total performance of the HPL test RR00L2L2 for maximal problem size. The HPL benchmark was compiled with the GNU C Compiler 3.2.

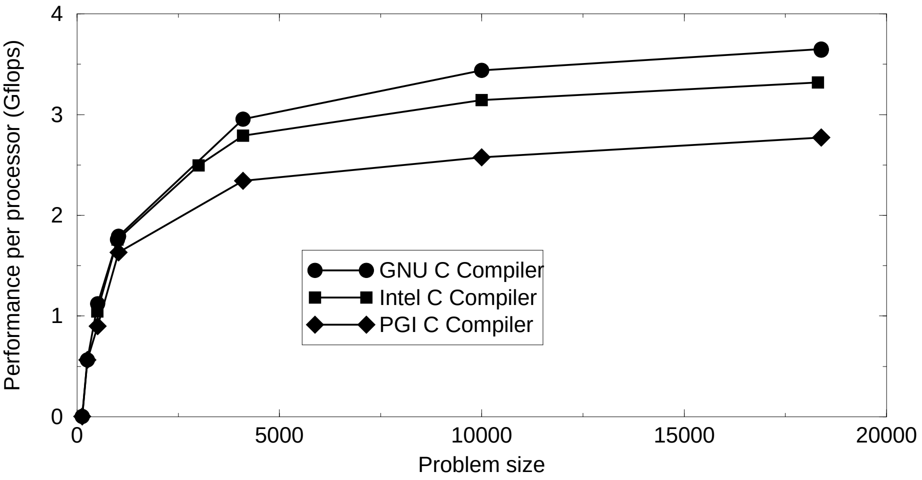


Fig. 4. Performance comparison of the HPL test WR00L2L2 on two processors with three different compilers

Comparing the compilers, we see that the GNU C compiler performs best in our tests (cf. Fig. 4). However, for small problems (up to size 1000) the actual choice of the compiler does not matter. For larger problems (size 18000) the PGI code is about 25% slower while the lag of the Intel compiler is moderate.

These results should not be misconstrued in the sense that GCC produces always better code than the commercial competitors. Firstly, real applications do not behave exactly as the benchmark. Secondly, according to our experiences, PGI performs much better with Fortran code (see also [19]).

5 D-Grid

The D-Grid initiative aims at design, building and operating a network of distributed, integrated and virtualized high-performance resources and related services which allow processing of large amounts of scientific data and information. D-Grid currently consists of the integration project (DGI) and six community projects in several scientific domains [20]. One work package of the integration project is to build an infrastructure called Core D-Grid. As part of this core infrastructure Opus^{IB} should be accessible via the three middleware layers GT4, Unicore [21] and gLite [22]. The GT4 services are the same as for CampusGrid plus an OGSA-DAI interface to an 1 TB mySQL database (cf. [23]) and integration into the MDS4 monitoring hierarchy of D-Grid.

The integration with gLite middleware suffers from a yet missing 64bit port of gLite. Thus, 32bit versions of the LCG tools must be used. The jobs submitted via gLite are scheduled on the cluster via TORQUE. Unicore also supports TORQUE, so we can use one local job scheduler on the cluster for all three middlewares. Each middleware should run on a separate (virtual) machine which is a submitting host for the local batch system.

So far, only few nodes are equipped with the gLite worker node software. A complete roll-out of the system with access via all three middlewares (GT4, gLite, and Unicore) is scheduled for the first quarter of 2007. At that time we will be productive with an extended D-Grid infrastructure (for example, we are adding 32 nodes to Opus^{IB} with two dual core processors each). A detailed report about configuration details and experiences will be subject of a separate publication.

Acknowledgments. The authors thank all colleagues for support in daily management Opus^{IB} and other systems and for helpful suggestions regarding our research activities. Great thanks are due to the developers of Open Source software tools we use in our projects.

References

1. Institut für Wissenschaftliches Rechnen, Forschungszentrum Karlsruhe: CampusGrid (2005), `http://www.campusgrid.de`
2. Schneider, O.: The project CampusGrid (NUG-XVI General Meeting, Kiel (May 24-27, 2004)

3. Institut für Wissenschaftliches Rechnen, Forschungszentrum Karlsruhe: Grid Computing Centre Karlsruhe (GridKa) (2005), `http://www.gridka.de`
4. InfiniBand Trade Association: InfiniBand Architecture (2006), `http://www.infinibandta.org`
5. Schwickerath, U., Heiss, A.: First experiences with the InfiniBand interconnect. Nuclear Instruments and Methods in Physics Research A 534, 130–134 (2004)
6. Cluster Resources Inc.: TORQUE Resource Manager (2006), `http://old.clusterresources.com/products/torque`
7. Cluster Resources Inc.: Maui cluster scheduler (2006), `http://old.clusterresources.com/products/maui`
8. Quattor development team: Quattor. System administration toolsuite (2006), `http://quattor.web.cern.ch`
9. The LAM team: Password Storage and Retrieval System (2006), `http://www.lam-mpi.org/software/psr`
10. Advanced Digital Information Corporation (ADIC): StorNext File System (2006), `http://www.adic.com`
11. Foster, I.T.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: Jin, H., Reed, D., Jiang, W. (eds.) NPC 2005. LNCS, vol. 3779, pp. 2–13. Springer, Heidelberg (2005)
12. The Globus Alliance: Globus Toolkit 4.0 Release Manuals (2006), `http://www.globus.org/toolkit/docs/4.0`
13. The Ganglia Development Team: Ganglia (2006), `http://ganglia.info/`
14. Capps, D.: IOzone Filesystem Benchmark (2006), `http://www.iozone.org`
15. Schmitz, F., Schneider, O.: The CampusGrid test bed at Forschungszentrum Karlsruhe (NUG-XVII General Meeting, Exeter, GB, May 25-27, 2005)
16. Petitet, A., Whaley, R.C., Dongarra, J., Cleary, A.: HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers (2004), `http://www.netlib.org/benchmark/hpl`
17. Network-Based Computing Laboratory, Dept. of Computer Science and Engg., The Ohio State University. MVAPICH: MPI over InfiniBand project (2006), `http://nowlab.cse.ohio-state.edu/projects/mpi-iba`
18. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing 27(1-2), 3–35 (2001), see also `http://www.netlib.org/atlas`
19. Research Computing, Information Technology Services, University of North Carolina: Compiler Benchmark for LINUX Cluster (2006), `http://its.unc.edu/hpc/performance/compiler_bench.html`
20. The D-Grid Initiative: D-Grid, an e-Science-Framework for Germany (2006), `http://www.d-grid.de`
21. The Unicore Project: UNICORE (Uniform Interface to Computing Resources) (2005), `http://www.unicore.eu`
22. The EGEE Project: gLite. Lightweight Middleware for Grid Computing (2006), `http://www.glite.org`
23. Jejkal, T.: Nutzung der OGSA-DAI Installation auf dem Kern-D-Grid (in German) (2006), `http://fuzzy.fzk.de/~GRID/DGrid/Kern-D-Grid.html`

Extending the HPC-ICTM Geographical Categorization Model for Grid Computing

Rafael K.S. Silva[1], Marilton S. de Aguiar[2], César A.F. De Rose[1], and Graçaliz P. Dimuro[2]

[1] PPGCC, Pontifícia Universidade Católica do Rio Grande do Sul, Av. Ipiranga 6681, 90619-900 Porto Alegre, Brazil
{rksilva, derose}@cpad.pucrs.br
[2] PPGInf, Universidade Católica de Pelotas, Rua Felix da Cunha 412, 96010-000 Pelotas, Brazil
{marilton, liz}@ucpel.tche.br

Abstract. This paper presents the results obtained with an extension of the HPC-ICTM model for grid computing. The HPC-ICTM is a multi-layered interval tessellation-based model for the categorization of geographic regions for high performance computing. The ICTM model is executed as a *Bag-of-Tasks* on the *OurGrid* environment. For each layer, the categorizer process is applied separately, without communication between the layers. An analysis of the performance of the model is presented.

1 Introduction

The ICTM (*Interval Categorizer Tessellation Model*) is a multi-layered and multi-dimensional tessellation model for the simultaneous categorization of geographic regions considering several different characteristics (relief, vegetation, climate, land use etc.) of such regions, which uses interval techniques [10,11] for the modeling of uncertain data and the control of discretization errors.

To perform a *simultaneous categorization*, the ICTM proceeds (in parallel) to individual categorizations considering one characteristic per layer, thus generating different subdivisions of the analyzed region. Each layer represents a tesselation for one determined property of the same analyzed region. An appropriate projection procedure of the categorizations performed in each layer into a basis layer provides the final categorization that allows the combined analysis of all characteristics that are taken into consideration by the specialists in the considered application, allowing interesting analyzes about their mutual dependency.

The HPC-ICTM – the multi-layered interval categorizer tessellation-based model for high performance computing – was introduced in [2], whereas the formalization of the ICTM, the single-layered model for the relief categorization of geographic regions, called TOPO-ICTM (*Interval Categorizer Tessellation Model for Reliable Topographic Segmentation*), was first presented in [1].

In this paper, we present some aspects about extension of the HPC-ICTM for grids. Moreover, we present also some features about first parallel implementation of the ICTM model on clusters, that generated de first version of the

HPC-ICTM model. We used the Message Passing Interface (MPI) to implement the HPC-ICTM on clusters because it is the standard for programming in such environments [8,13].

The computational grids have important characteristics that must be considered when developing a parallel application. One of most important characteristics is the high geographic dispersion [4,9]. Thus, applications that possess tasks that produce much communication, are not adjusted for this platform.

A class of applications that fits easily in the grid environment is the *BOT* (Bag-of-Tasks), because they are characterized by having independent tasks that do not communicate among themselves [5]. The ICTM model can be executed as a *BOT* application. For each layer of the model, the categorizer process is applied separately, without communication between the layers. Thus, despite the fact of the partition in layers, the ICTM presents the typical behavior of a *BOT* application. We implemented the HPC-ICTM grid extension on the *OurGrid* environment [3].

2 Geographical Categorizations Using the HPC-ICTM Model

The use of ICTM model for analysis of several properties of large geographic regions requires high computational power. A solution for this problem was to parallel the ICTM model for clusters execution. Thus appeared the first version of the HPC-ICTM model, through MPI library. However, sometimes the cluster resources are not enough to categorize these large areas. Moreover, to get different geographic data is a task sufficiently complicated. The main reason is the high cost of the generating devices.

Therefore, we proposed an extension of HPC-ICTM model for Grid Computing, through the OurGrid environment. The main objective is to get computational resources enough for the categorizations and make use of geographic data distributed in grid machines located in specialized research centers, of simple and efficient way.

The HPC-ICTM is a multi-layered and multi-dimensional tessellation model for the simultaneous categorization of geographic regions considering several different characteristics (relief, vegetation, climate, land use etc.). In order to control and minimize the errors coming from the discretization of the region into tessellation cells and from numerical computations, we employ interval techniques [11].

The input data are extracted from satellite images, where the heights are given in certain points referenced by their latitude and longitude coordinates. The geographic region is represented by a regular tessellation that is determined by subdividing the total area into sufficiently small rectangular subareas, each one represented by one cell of the tessellation. This subdivision is done according to a cell size established by the geophysics or ecology analyst and it is directly associated to the refinement degree of the tessellation.

Applications in Ecology were found, where an adequate subdivision of geographic areas into segments presenting similar topographic characteristics is often convenient (see, e.g: [6,7]).

3 The Parallel Implementation of the `ICTM`

The parallel version of `ICTM` explores four possibilities for problem decomposition: *Layers* - each parallel process calculates a determined layer of the model; *Functions* - each parallel process calculates an independent function of the model; *Domains* - each parallel process calculates part of the region that will be analyzed; *Cells* - each parallel process calculates individual cells of the model.

Since in clusters architectures the interconnection network can be considered bottleneck (the cost of local computation is much cheaper then communication with neighbors nodes) we are interested in a problem decomposition that results in big chunks of work with low communication overhead (coarse grain [14]).

Considering this granularity issue, decomposition in layers is the most simple and direct way to implement a parallel version of the `ICTM`. Being each process responsible for a determined layer of the model, these processes perform the `ICTM` calculations for each property in parallel.

The `HPC-ICTM` was implemented on clusters, using the MPI (*Message Passing Interface*) [8] library, a master-slave scheme [14] (Fig. 1). The master process is responsible for loading the input files and parameters (the data and the radius), dividing total work in *nl* tasks (*nl* is the number of layers that will be processed), sending the radius value and the tasks for all slave processes to start the categorization process, and keeping control of the tasks.

The slave processes receive the information sent by the master process, execute their tasks and generate their own outputs. The directory with input and output files is in the same file system, being accessible by all the cluster nodes. After that, they ask the master for more work. Until there is work to do, the master keep sending tasks to the slaves.

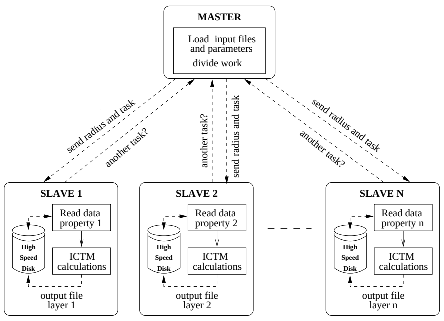


Fig. 1. Master-slave scheme to solve the problem of decomposition in layers

Decomposition in layers follows the rule $np = nl + 1$, where np indicates the number of processes and nl indicates the number of layers or properties that need to be processed, considering each slave process running in a different node. Thus, each layer is analyzed by a different slave process and the remnant process is the master process.

Communication occurs between master and slaves (not among slaves) and only (i) when the master process sends the radius value and a layer for a slave to compute, and (ii) when a slave notifies the master that the computation of a layer has finished.

Each slave allocates only the amount of memory needed to calculate one layer/property of the model. However, if the main memory size of a cluster node is insufficient to execute one layer of the model, the application will access the hard disk (swap) or, in the worst case, it will abort. In these cases, the problem decomposition in layers is not recommended.

The analysis of the `HPC-ICTM` performance and some results concerning its application to the relief/land-use categorization was presented in [12].

4 The `HPC-ICTM` Model on the *OurGrid* Environment

OurGrid (presented in [3]) is an environment for grid computing developed by the Federal University of Campina Grande (UFCG, Brazil) in partnership with the HP Brazil (Hewlett Packard Brazil). It is an environment for global execution, with *sites* for parallel applications based on Bag-of-Tasks. This computational power is provided by the idle resources of all participants, and is shared in a way that makes those that contribute get more when they need.

Currently, the platform can be used to run any application whose tasks (i.e., parts that run on a single machine) do not communicate among themselves during execution, as, e.g., most simulations, data mining and searching.

To execute a parallel application in *OurGrid*, the user located at a given *site* must have the *MyGrid* package installed in his computer for the local scheduling of the applications and to communicate with the remote machines (grid machines). The user also needs to know the address of a *peer*. It is through this *peer* that the addresses of the available machines in the grid will be acquired. The idle resources of one determined domain are announced to the local *peer* by *user agents*, or the *peer* maintains a list of the available machines that is constantly updated.

After communicating with the *MyGrid–Peer*, the resources are supplied to the scheduler of the user, and then the application submission can be scheduled by the *MyGrid–User Agents*. In the cases where the *peer* does not obtain all the resources that the user asks, it can search them in the community by asking other *peers*. The way that the solicitation of a remote *peer* is replied is based on the model of exchange favors[3]. Basically, a peer will prefer the one that had previously yielded more resources.

In the *OurGrid*, each job is performed in three phases: i) *init*: this phase is responsible for the information sending from home machine to grid machine; ii) *remote*: this phase is responsible for running the application at the grid machine; and iii) *final*: responsible for sending results from the grid machine to the home machine.

Hence, the multi-layered `ICTM` can be modelled as one job with n tasks (see Fig. 2), where n is the number of layers that will be analyzed. In each phase, the job sends instructions or information as follows: i) in the *init* phase, the data on each layer and the sequential implementation of the ICTM; ii) in the *remote* phase, the *start* instruction which executes the `ICTM` for the input data supplied; and iii) in the final phase, the instruction that returns the results from the analyzed layer to the user.

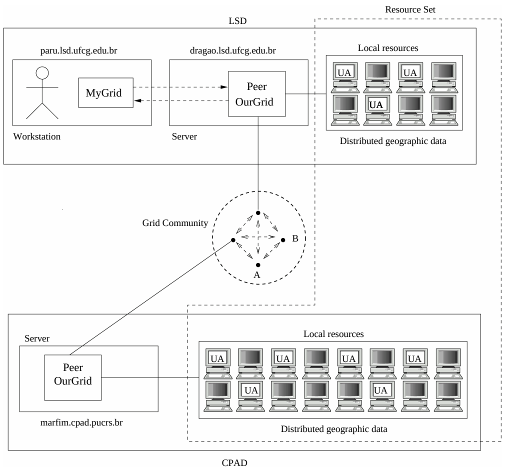


Fig. 2. The `HPC-ICTM` on the *OurGrid* Environment

5 Performance Analysis and Final Remarks

Images from the LANDSAT[1] satellite and SRTM[2] radar were used to validate the `HPC-ICTM` model for grid computing.

All the geographic data extracted from the images are altimetric data. Thus, for the accomplishment of tests with various layers, the same data of topography was replicated in each layer. However, it does not affect the obtained results since the dimension of each layer is accurately the same.

Categorizations were obtained for the region surrounded the lagoon *Lagoa Pequena* (Rio Grande do Sul, Brazil). The objective of this characterization was the extraction of information to be used for the management and preservation of the area. The analyzed region was divided in five subareas (sets of input data) to allow tests with different sizes of data sets (quad. A: 58.081 cells, quad. B: 471.409 cells, quad. C: 2.310.385, quad. D: 3.763.196 cells and quad. E: 18.782.740 cells).

In order to verify the behavior and the performance of all the parallel models of the `ICTM` model, three different tests for the clusters and two tests for the computational grids were performed. Moreover, a test of the sequential version of `ICTM` model for all quadrants was performed in order to calculate the reference

[1] Land Remote Sensing Satellite.
[2] Shuttle Radar Topography Mission.

Table 1. Results from the sequential implementation - T_{seq} (l is the number of layers)

Quadrant	**l**	**E800 Server**	***Workstation***	**P4**	**IA64-dual**
$A_{(241\times241)}$	1	0.632s	0.593s	0.257s	0.452s
	3	1.838s	1.627s	0.885s	1.119s
	5	3.024s	2.663s	1.449s	1.827s
	7	4.471s	3.763s	2.804s	2.516s
$B_{(577\times817)}$	1	5.483s	4.132s	1.884s	3.234s
	3	15.282s	12.373s	5.681s	7.188s
	5	25.606s	20.514s	9.464s	11.979s
	7	37.045s	28.682s	14.247s	16.818s
$C_{(1309\times1765)}$	1	25.276s	19.971s	8.450s	12.098s
	3	*Swap*	*Swap*	26.711s	33.652s
	5	–	*Swap*	45.693s	56.226s
	7	–	–	–	79.568s
$D_{(1739\times2164)}$	1	42.282s	33.242s	15.734s	21.340s
	3	–	*Swap*	44.982s	58.135s
	5	–	–	–	116.649s
	7	–	–	–	–
$E_{(4022\times4670)}$	1	–	–	*Swap*	100.364s
	3	–	–	–	–
	5	–	–	–	–
	7	–	–	–	–

time to be used in the comparisons with the parallel versions (see Table 1). Note that the best sequential times were obtained in the 64 bits processors (IA64-dual). These times were used as reference for the parallel executions.

The following tests were executed in the clusters: i) the layer parallelization of the ICTM model, i.e., each layer in one processor; the quadrants A, B and C had been analyzed with 3, 5 and 7 layers (see Table 2); ii) the functional parallelization of the ICTM model, .i.e., each task of the method is executed in one processor, using the quadrants C and D (see Table 3) ; iii) the ICTM parallelization with domain decomposition, using the quadrant E (see Table 4).

The tests executed in grids were: i) using centralized geographic data for all quadrants and 3, 5 and 7 layers (see Table 5)[3]; and ii) using distributed geographic data for the quadrants C and D, using 3, 5 and 7 layers (see Table 6). In this case, the data was locally stored in each machine of the grid.

These tests was performed in machines located at CPAD (High Performance Center, PUCRS) and LSD (Distributed Systems Laboratory, UFCG, Federal University of the Campina Grande, Brazil). We concluded that the partitioning in layers for clusters must be used in the analysis of large geographic regions that present different properties. The larger number of layers, the greater will be the gain obtained with the parallel version. The parallelization of the ICTM operations in clusters must be used in the analysis of large geographic regions that present

[3] WQR and Storage Affinity scheduler algorithms was used.

Table 2. Results from the layer parallelization implementation (l is the number of layers; T_{seq} is the reference sequential time; T_{par} is the parallel time; Speedup = T_{seq}/T_{par}; Effic = percentage of efficiency)

Quadrant	**l**	**T**$_{seq}$	**E800**			**IA64-dual**		
			T$_{par}$	**Speedup**	**Effic.**	**T**$_{par}$	**Speedup**	**Effic.**
$A_{(241\times241)}$	3	1.119s	2.085s	0.54	13%	1.711s	0.65	16%
	5	1.827s	2.285s	0.80	13%	2.378s	0.77	13%
	7	2.516s	2.840s	0.86	11%	2.630s	0.96	12%
$B_{(577\times817)}$	3	7.188s	7.119s	1.00	25%	4.013s	1.79	45%
	5	11.979s	7.437s	1.61	27%	4.557s	2.63	44%
	7	16.818s	8.055s	2.09	26%	5.568s	3.02	38%
$C_{(1309\times1765)}$	3	33.652s	28.621s	1.18	29%	13.564s	2.48	62%
	5	56.226s	30.627s	1.84	31%	14.014s	4.01	67%
	7	79.568s	31.005s	2.57	32%	14.433s	5.51	69%

Table 3. Results from the functional parallelization implementation (radius = neighborhood cells; T_{seq} is the reference sequential time; T_{par} is the parallel time; Speedup = T_{seq}/T_{par}; Effic = percentage of efficiency)

Quadrant	**radius**	**P4**				**IA64-dual**			
		T$_{seq}$	**T**$_{par}$	**Speedup**	**Effic.**	**T**$_{seq}$	**T**$_{par}$	**Speedup**	**Effic.**
$C_{(1309\times1765)}$	10	11.784s	9.945s	1.18	24%	11.637s	9.645s	1.21	24%
	20	15.721s	13.002s	1.20	24%	12.347s	10.134s	1.22	24%
	30	24.249s	20.278s	1.20	24%	13.069s	10.715s	1.22	24%
	40	27.731s	22.678s	1.22	24%	20.735s	14.967s	1.39	28%
	50	31.952s	23.765s	1.34	27%	34.118s	24.122s	1.42	28%
	60	36.730s	25.456s	1.44	29%	38.816s	27.356s	1.42	28%
	70	41.918s	28.330s	1.48	30%	43.287s	30.506s	1.42	28%
	100	75.210s	39.423s	1.91	38%	79.325s	44.331s	1.79	36%
	200	248.321s	110.098s	2.25	45%	256.567s	115.123s	2.23	45%
$D_{(1739\times2164)}$	10	18.216s	12.110s	1.50	30%	18.118s	12.002s	1.51	30%
	20	24.439s	16.315s	1.50	30%	20.277s	13.221s	1.53	31%
	30	37.578s	25.080s	1.50	30%	22.340s	14.620s	1.53	31%
	40	43.946s	26.648s	1.66	33%	47.858s	29.345s	1.63	33%
	50	51.121s	28.950s	1.77	35%	55.387s	31.210s	1.77	35%
	60	59.166s	30.008s	1.97	39%	62.655s	33.001s	1.90	38%
	70	68.224s	33.987s	2.01	40%	69.668s	35.301s	1.97	39%
	100	155.345s	62.345s	2.51	50%	159.333s	67.441s	2.36	47%
	200	399.178s	132.234s	3.02	60%	405.212s	136.200s	2.98	60%

a level of great complexity, either for the size of the radius of neighborhood cells or for the data absolute values.

The computational environment used in this test was:

– `HP E800 Server`: Dual Intel PIII 1GHz, 512MB RAM and 256KB *cache*;
– *Workstation*: Intel Pentium 4 1.6GHz, 512MB RAM and 256KB *cache*;

Table 4. Results from the domain decomposition implementation (T$_{seq}$ is the reference sequential time; #blocks is the number of blocks cells; T$_{par}$ is the parallel time; Speedup = T$_{seq}$/T$_{par}$)

Quadrant	**T$_{seq}$**	**#blocks**	**Amazônia - 32 *bits***	
			T$_{par}$	**Speedup**
$E_{(4022\times4670)}$	100.364s	3	105.332s	0.95
		7	70.450s	1.42
		15	50.340s	1.99
		31	41.455s	2.42
		39	33.211s	3.02
		63	69.420s	1.45
		79	115.367s	0.87

Table 5. Results from the centralized geographic data implementation (l is the number of layers; T$_{seq}$ is the reference sequential time)

Quadrant	**l**	**T$_{seq}$**	**OurGrid**	
			WQR	**S. Affinity**
$A_{(241\times241)}$ File: 540K	1	0.452s	18.789s	16.066s
	3	1.119s	24.512s	21.569s
	5	1.827s	55.319s	49.156s
	7	2.516s	59.441s	55.345s
$B_{(577\times817)}$ File: 4.8M	1	3.234s	81.921s	72.844s
	3	7.188s	89.666s	82.898s
	5	11.979s	100.701s	86.737s
	7	16.818s	116.201s	91.345s
$C_{(1309\times1765)}$ File: 21M	1	12.098s	204.400s	192.382s
	3	33.652s	222.556s	210.293s
	5	56.226s	232.875s	216.667s
	7	79.568s	250.213s	225.312s
$D_{(1739\times2164)}$ File: 38M	1	21.340s	274.231s	264.112s
	3	58.135s	290.389s	278.236s
	5	116.649s	313.222s	284.561s
	7	-	325.667s	296.212s
$E_{(4022\times4670)}$ File: 178M	1	100.364s	470.476s	456.980s
	3	-	496.557s	465.180s
	5	-	503.112s	477.213s
	7	-	525.987s	503.098s

- `HP Compaq dc5000 MT`: Intel Pentium 4 2.8GHz, 1GB RAM and 1024KB *cache* (denotated by `P4`);
- `HP Integrity rx2600 IA64`: Dual Itanium 64 *bits* 1.5GHz, 2GB RAM and 1024KB *cache* (denotated by `IA64-dual`);
- 8 nodes `HP-E60`: Dual Pentium III 550MHz, 256MB RAM and 512KB *cache*;
- 8 nodes `HP-E800`: Dual Pentium III 1GHz, 256MB RAM and 256KB *cache*;

Table 6. Results from the distributed geographics data implementation (l is the number of layers; T_{seq} is the reference sequential time)

Quadrant	**l**	$\mathbf{T}_{seq}$	**Distributed**		**OurGrid**
			LSD	**CPAD**	
$C_{(1309\times1765)}$	3	33.652s	2	1	21.915s
	5	56.226s	3	2	23.011s
	7	79.568s	4	3	24.110s
$D_{(1739\times2164)}$	3	58.135s	1	2	47.760s
	5	116.649s	2	3	50.234s
	7	-	3	4	53.314s

– 8 nodes `HP Compaq dc5000 MT`: Pentium 4 2.8GHz, 1GB RAM and 1024KB *cache* (denotated by `P4`);
– 5 nodes `HP Integrity rx2600 IA64`: Dual Itanium 64 *bits* 1.5GHz, 2GB RAM and 1024KB *cache* (denotated by `IA64-dual`).

The partitioning in layers for computational grids must be used when the properties are locally stored in the nodes that will make its processing. We also advise the use of grids mainly when the data will be distributed. The transference of large archives by the Internet becomes the bottleneck of the grid implementation. On the other hand, if the network that establishes connection between the resources is dedicated, then the transfer of archives may not cause much impact in the time of execution of the application.

As future work will be developed a new version of the `HPC-ICTM` model for Grid Computing with communication between tasks, since the last version of the OurGrid environment adds support will be parallel applications developed on MPI. This new characteristic will allow the implementation of other possibilities for problem decomposition.

References

1. Aguiar, M.S., Costa, A.C.R., Dimuro, G.P: ICTM: an interval tesselation-based model for reliable topographic segmentation. Numerical Algorithms 37(1-4), 3–11 (2004)
2. Aguiar, M.S., Dimuro, G.P., Costa, F.A., Silva, R.K.S., De Rose, C.A.F., Costa, A.C.R., Kreinovich, V.: HPC-ICTM: The Interval Categorizer Tessellation-Based Model for High Performance Computing. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 83–92. Springer, Heidelberg (2006)
3. Andrade, N., et al.: OurGrid: An approach to easily assemble grids with equitable resource sharing. In: Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing, pp. 61–86. Springer, Heidelberg (2003)
4. Baker, M., Buyya, R., Laforenza, D.: Grids an Grid Technologies for Wide-area Distributed Computing. Software: Practice and Experience 32(15), 1437–1466 (2002)

5. Cirne, W.: Running Bag-of-Tasks Applications on Computational Grids: The My-grid Approach. In: Proceedings of the 2003 International Conference on Parallel Processing, pp. 407–416. IEEE Computer Society Press, Kaohsiung (2003)
6. Coblentz, D., Kreinovich, V., Penn, B., Starks, S.: Towards reliable sub-division of geological areas: Interval approach. In: Reznik, L., Kreinovich, V. (eds.) Soft Computing in Measurements and Information Acquisition, pp. 223–233. Springer, Heidelberg (2003)
7. Forman, R.T.T.: Land Mosaics: the ecology of landscapes and regions. Cambridge University Press, Cambridge (1995)
8. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, p. 371. MIT Press, Cambridge, Mass (1999)
9. Nemeth, Z., Sunderam, V.: Characterizing Grids: Attributes, Definitions and Formalisms. Journal of Grid Computing 1, 9–23 (2003)
10. Kearfort, R.B., Kreinovich, V. (eds.): Applications of Interval Computations. Kluwer, Dordrecht (1996)
11. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM, Philadelphia, PA, USA (1979)
12. Silva, R.K.S., De Rose, C.A.F., Aguiar, M.A., Dimuro, G.P., Costa, A.C.R.: In: IEEE 2006 International Symposium on Modern Computing. Proceedings. IEEE Computer Society Press, Sofia, Bulgaria (to appear, 2006)
13. Walker, D.W.: The Design of a Standard Message Passing Interface for Distributed Concurrent Computers. Parallel Computing 20(4), 657–673 (1994)
14. Wilkinson, B., Allen, M.: Parallel Programming: techniques and applications using networked workstations and parallel computers. Prentice-Hall, Upper Saddle River, New Jersey (1999)

Distributed SILC: An Easy-to-Use Interface for MPI-Based Parallel Matrix Computation Libraries

Tamito Kajiyama[1,2], Akira Nukada[1,2], Reiji Suda[1,2], Hidehiko Hasegawa[3], and Akira Nishida[1,4]

[1] CREST, Japan Science and Technology Agency, Saitama 332–0012, Japan
[2] The University of Tokyo, Tokyo 113–8656, Japan
kajiyama@is.s.u-tokyo.ac.jp
[3] University of Tsukuba, Tsukuba 305–8550, Japan
[4] 21st Century COE Program, Chuo University, Tokyo 112–8551, Japan

Abstract. The present paper describes the design and implementation of distributed SILC (Simple Interface for Library Collections) that gives users access to a variety of MPI-based parallel matrix computation libraries in a flexible and environment-independent manner. Distributed SILC allows users to make use of MPI-based parallel matrix computation libraries not only in MPI-based parallel user programs but also in sequential user programs. Since user programs for SILC are free of a source-level dependency on particular libraries and computing environments, users can easily utilize alternative libraries and computing environments without any modification in the user programs. The experimental results of two test problems showed that the implemented SILC system achieved speedups of 2.69 and 7.54 using MPI-based parallel matrix computation libraries with 16 processes.

1 Introduction

The traditional way of using matrix computation libraries based directly on library-specific application programming interfaces usually leads to a source-level dependency on the libraries in use. This source-level dependency is the primary reason why users (i.e., application programmers) are often required to make a considerable amount of modifications to their user programs, for example when porting them to other computing environments or when trying out other libraries having different sets of solvers, matrix storage formats, arithmetic precisions, and so on. To address this issue inherent in the traditional programming style, we have been proposing an easy-to-use application framework named Simple Interface for Library Collections (SILC) [1,2]. A user program in the SILC framework first deposits data such as matrices and vectors into a separate memory space. Next, the user program makes requests for computation by means of mathematical expressions in the form of text. These requests are translated into calls for appropriate library functions, which are carried out in the separate

```
double *A, *B;
int desc_A[9], desc_B[9], *ipiv, info;
/* create matrix A and vector B */
PDGESV(N, NRHS, A, IA, JA, desc_A, ipiv, B,
       IB, JB, desc_B, &info);
/* solution X is stored in B */
```

(a)

```
silc_envelope_t A, b, x;
/* create matrix A and vector B */
SILC_PUT("A", &A);
SILC_PUT("b", &b);
SILC_EXEC("x = A \\ b"); /* call PDGESV() */
SILC_GET(&x, "x");
```

(b)

Fig. 1. A comparison of two C programs (a) in the traditional programming style and (b) in the SILC framework, both making use of ScaLAPACK to solve a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$

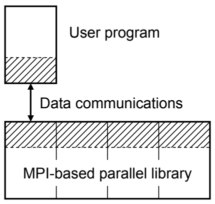
memory space independently of the user program. Finally, the user program fetches the results of computation from the separate memory space.

Figure 1 shows two user programs written in C, one in the traditional programming style and the other in the SILC framework. The traditional user program shown in Fig. 1 (a) prepares matrix A and vector $\boldsymbol{b}$ using library-specific data structures and makes a call for a library function in ScaLAPACK [3] to solve a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$. The user program for SILC shown in Fig. 1 (b) realizes the same computation using the following three routines: `SILC_PUT` to deposit A and $\boldsymbol{b}$ into a separate memory space, `SILC_EXEC` to issue a request for solution of the linear system by means of a mathematical expression in the form of text, and `SILC_GET` to retrieve the solution $\boldsymbol{x}$. The mathematical expression specified as the argument of `SILC_EXEC` is translated into a call for the library function in ScaLAPACK for example, and carried out in the separate memory space.

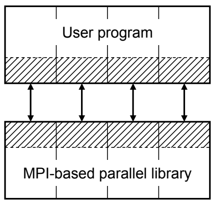
We have developed a SILC system for sequential and shared-memory parallel computing environments [1,2]. The current implementation of SILC is based on a client-server architecture, in which a user program is a client of a SILC server running in a remote computing environment. Since a user program for SILC does not contain any library-specific code, no modification to the user program is required to utilize alternative matrix computation libraries. Moreover, users can automatically gain the advantages of parallel computation by using a SILC server that runs in a parallel computing environment. The main overhead in using SILC, on the other hand, is the cost of data communications between a user program and a SILC server. However, it is not difficult to reduce the relative amount of communication overhead, since the time complexities of matrix computations tend to be larger than their space complexities. For instance, solving a dense linear system with N unknowns takes $O(N^3)$ time, while the time necessary for data communications is of $O(N^2)$. Consequently, in many cases the use of a faster matrix computation library and computing environment results in good speedups even at the cost of data communications.

2 SILC for Distributed Parallel Computing Environments

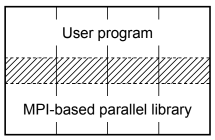
We have been developing a SILC system for distributed parallel computing environments that allows users to make use of MPI-based matrix computation



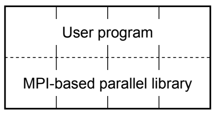
Configuration (A).



Configuration (B).



Configuration (C).



The traditional configuration.

Fig. 2. Three system configurations of distributed SILC, compared with the configuration of an MPI-based parallel program in the traditional programming style

libraries in a flexible and computing environment-independent manner. The primary goal in the design of distributed SILC is to support as many MPI-based parallel matrix computation libraries and computing environments as possible, since SILC is a piece of middleware placed between user programs and matrix computation libraries, serving as an abstraction layer that hides the details of the libraries and underlying computing environments. Having this design goal in mind, we consider three system configurations shown in Fig. 2. The shaded parts in the figure show the components that SILC provides. For comparison, the figure also shows the configuration of a user program in the traditional programming style. The traditional user program in this example consists of four MPI processes.

Configurations (A) and (B) are based on a client-server architecture in which a user program is a client of an MPI-based parallel SILC server. The user program establishes a TCP connection to the SILC server and makes use of MPI-based parallel matrix computation libraries managed by the server. The user program in (A) is sequential, while the user program in (B) is an MPI-based parallel program. Both the server and the user program in (B) shown in Fig. 2 consist of four MPI processes.

In Configuration (A), the user program makes a connection to one of the server processes through which both data and requests for computation are transferred. The server distributes the received data among the server processes by means of a data redistribution mechanism, keeping the data among the server processes in a distributed manner. The data redistribution mechanism is also utilized to make a change in data distributions in the following two situations. One situation is when the server handles requests for computation, where the data is passed to a library function as an argument in a different data distribution the library function accepts. The other situation is when the user program fetches the results of preceding computation requests, where the data is transferred in the data distribution that the user program requires. Computation requests by means of textual mathematical expressions are handled by the library interface in the server. The interface incorporates an interpreter that translates the expressions into calls for appropriate library functions, which are carried out within the server processes.

In Configuration (B), each process of the MPI-based parallel user program makes a separate connection to one of the server processes; that is, multiple connections are established between the user program and the server. Data is retained in a distributed manner in both the user program and the SILC server, and parallel data transfer is performed between the user program and the server through the multiple connections. The data redistribution mechanism is employed in the same manner as Configuration (A), when the server needs to change distributions of data. Requests for computation, on the other hand, are sent to the server from one process on behalf of the user program. Since a user program in Configuration (B) is an ordinary MPI-based program, it can be executed, for example, as follows:

```
mpirun -np n ./my_silc_application
```

where n is the number of processes on which the program runs. At the moment, the number of processes of the user program must be smaller than or equal to the number of the server's processes.

Configuration (C) is prepared for some restrictive computing environments in which the client-server architecture cannot be adopted. In this configuration, the data redistribution mechanism and library interface of the SILC server are implemented as a library, which is linked to MPI-based parallel user programs together with MPI-based parallel matrix computation libraries. There is no source-level difference between a user program in Configuration (B) and another program in (C); that is, the source code of the two programs is the same, so that these configurations can be exchanged without any modification to the source code. Unlike Configurations (A) and (B), on the other hand, library functions in this configuration are carried out within the processes of a user program.

3 Experiments

To determine whether the implemented SILC system is capable of achieving speedups when compared with the traditional programming style, we conducted experiments with regard to the following two test problems.

Problem 1. A dense linear system $A\boldsymbol{x} = \boldsymbol{b}$.
Problem 2. An initial value problem of a partial differential equation (PDE).

Table 1 is a summary of the computing environments used in the experiments. These computing environments are in the same Gigabit Ethernet (GbE) LAN. Both Xeon4 and Xeon8 consist of a disjoint set of nodes in the same GbE-based PC cluster. Only one core of each node was used. Computation was done in double precision real throughout the experiments.

Table 1. The computing environments used in the experiments

Host name	Specifications
Xeon4	IBM eServer xSeries 335 (dual Intel Xeon 2.8 GHz, L2 cache 512 KB, Memory 1 GB) × 4, Red Hat Linux 8.0, LAM/MPI 7.0
Xeon8	Different 8 nodes in the same PC cluster as Xeon4
Altix	SGI Altix 3700 (Intel Itanium2 1.3 GHz × 32, L2 cache 256 KB, Memory 32 GB), Red Hat Linux Advanced Server 2.1, SGI MPI 4.4 (MPT 1.9.1)

3.1 Problem 1: Solution of a Dense Linear System

Consider solving a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ using the `PDGESV` routine in ScaLAPACK, where A is an $N \times N$ dense matrix and $\boldsymbol{b}$ and $\boldsymbol{x}$ are N-vectors. We prepared the following two user programs, both of which are MPI-based parallel programs written in C.

Program P_1 in the traditional programming style. The program first prepares A and $\boldsymbol{b}$ in the two-dimensional block-cyclic distribution [3]. The elements of A are random numbers, while those of $\boldsymbol{b}$ are given so that all elements of solution $\boldsymbol{x}$ will be 1. Then, the program makes a call for `PDGESV` to solve the linear system. The time elapsed in the ScaLAPACK routine was measured as the execution time of the program.

Program P_2 in the SILC framework. The program also prepares A and $\boldsymbol{b}$ in the same way as P_1. Next, the program makes two calls for `SILC_PUT` to deposit A and $\boldsymbol{b}$ into a SILC server, respectively, and another call for `SILC_EXEC` to request the solution of the linear system. This request is translated into a call for the `PDGESV` routine, which is carried out on the server side. Finally, the program calls for `SILC_GET` to retrieve the solution $\boldsymbol{x}$ from the server. We prepared three SILC servers running in Xeon4, in Xeon8, and in Altix. The elapsed time from the connection to a server to the data transfer of $\boldsymbol{x}$ was measured as the execution time of the program.

Table 2 summarizes the computing environments used for Problem 1. Both P_1 and P_2 were executed with 4 processes in Xeon4, whereas the SILC servers used by P_2 were executed with 4 processes in Xeon4, with 8 processes in Xeon8, and with 16 processes in Altix. Since P_2 is an MPI-based parallel program, this configuration corresponds to Configuration (B) shown in Fig. 2. Timing was done by the `gettimeofday` system call.

Table 3 shows the experimental results, where T is execution time in seconds, S is speedup (i.e., a ratio of the execution time of P_1 to that of P_2), and C is a proportion of communication overhead to the execution time of P_2 (we assumed $C = (T - T_{comp})/T$, where T_{comp} is a computation time measured on the server side). The execution time of P_2 includes the time for the data transfer and distribution of A and $\boldsymbol{b}$, as well as the time for the collection and data transfer of $\boldsymbol{x}$. These data communications constitute the major overhead

Table 2. The computing environments used for Problem 1

Label	User program	SILC server	Configuration
Trad.	P_1 in Xeon4 (4 PEs)	–	–
SILC (local)	P_2 in Xeon4 (4 PEs)	Xeon8 (4 PEs)	(B)
SILC (remote #1)	P_2 in Xeon4 (4 PEs)	Xeon8 (8 PEs)	(B)
SILC (remote #2)	P_2 in Xeon4 (4 PEs)	Altix (16 PEs)	(B)

Table 3. The results of Problem 1 (solution of a dense linear system $A\boldsymbol{x} = \boldsymbol{b}$). T is execution time in seconds, S is speedup, and C is communication overhead.

	Trad.	SILC (local)		SILC (remote #1)		SILC (remote #2)	
N	T	T (S)	C	T (S)	C	T (S)	C
1,000	1.592	1.417 (1.12)	7.8%	2.179 (0.73)	9.5%	0.790 (2.02)	16.0%
2,000	5.403	5.453 (0.99)	6.5%	5.789 (0.93)	11.1%	2.827 (1.91)	13.8%
4,000	27.153	30.145 (0.90)	4.0%	22.626 (1.20)	9.8%	14.235 (1.91)	10.2%
8,000	186.991	208.880 (0.90)	2.3%	130.892 (1.43)	6.5%	69.481 (2.69)	8.3%

in using SILC. However, as indicated by the proportion C in Table 3, the cost of data communications becomes relatively smaller as dimension N increases, because the solution of the dense linear system requires a computation time on the order of $O(N^3)$, while the data communications take only $O(N^2)$ time. Since the computation time can be significantly reduced by using a faster computing environment via SILC, some speedups are expected to be achieved even at the cost of data communications when N is large. This holds true for the experimental results shown in Table 3 – the speedups in the case of $N = 8,000$ were 1.43 with the SILC server in Xeon8 and 2.69 with the server in Altix.

3.2 Problem 2: Solution of an Initial Value Problem of a PDE

We solve the two-dimensional time-dependent diffusion equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ ($t \geq 0$) in the region $0 \leq x \leq 1$ and $0 \leq y \leq 1$ subject to the initial condition

$$u(x,\, y,\, 0) = \begin{cases} 1 & \text{if } |x - 0.5| < 0.1 \text{ and } |y - 0.5| < 0.1, \\ 0 & \text{otherwise,} \end{cases}$$

and boundary conditions $u(0,\, y,\, t) = u(1,\, y,\, t) = u(x,\, 0,\, t) = u(x,\, 1,\, t) = 0$ for $t > 0$, using the Crank-Nicolson method [4]. Suppose $t_0 = 0$ is the initial time and $\Delta t > 0$ is a constant time interval, and define the k-th time step as $t_k = t_{k-1} + \Delta t$. In using the Crank-Nicolson method, we have to solve a system of linear equations $A\boldsymbol{x}_k = \boldsymbol{b}_k$ for each time step, where A is an $N \times N$ sparse matrix and $\boldsymbol{b}_k$ and $\boldsymbol{x}_k$ are N-vectors. $\boldsymbol{b}_k$ is defined as $\boldsymbol{b}_k = C\boldsymbol{x}_{k-1}$, i.e. the matrix-vector product of another $N \times N$ sparse matrix C and the solution $\boldsymbol{x}_{k-1}$ at t_{k-1}. We prepared the following two user programs, both of which are sequential programs written in C.

Table 4. The computing environments used for Problem 2

Label	User program	SILC server	Configuration
Trad.	P_1 in Xeon4 (1 PE)	–	–
SILC (local)	P_2 in Xeon4 (1 PE)	Xeon4 (4 PEs)	(A)
SILC (remote #1)	P_2 in Xeon4 (1 PE)	Xeon8 (8 PEs)	(A)
SILC (remote #2)	P_2 in Xeon4 (1 PE)	Altix (16 PEs)	(A)

Table 5. The results of Problem 2 (solution of an initial value problem of a PDE). T is execution time in seconds, S is speedup, and C is communication overhead.

	Trad.	SILC (local)		SILC (remote #1)		SILC (remote #2)	
N	T	T (S)	C	T (S)	C	T (S)	C
10,000	0.432	0.693 (0.62)	46.54%	1.040 (0.42)	41.34%	0.423 (1.02)	56.8%
40,000	5.019	3.164 (1.59)	33.55%	3.756 (1.34)	39.32%	1.209 (4.15)	38.2%
90,000	19.206	8.587 (2.24)	28.83%	7.402 (2.59)	24.08%	2.981 (6.44)	31.1%
160,000	43.118	17.617 (2.45)	22.72%	13.850 (3.11)	22.69%	6.078 (7.09)	27.1%
250,000	82.798	30.627 (2.70)	17.78%	22.505 (3.68)	19.77%	10.987 (7.54)	23.1%

Program P_1 in the traditional programming style.

1. Prepare matrices A and C and the initial solution $\boldsymbol{x}_0$ at t_0. The matrices are stored in the Compressed Row Storage (CRS) format [5].
2. For each time step t_k ($k = 1, 2, 3, \ldots$):
   (a) Compute $\boldsymbol{b}_k = C\boldsymbol{x}_{k-1}$ using the sparse matrix-vector product routine in the sequential version of an iterative solvers library Lis [6].
   (b) Solve $A\boldsymbol{x}_k = \boldsymbol{b}_k$ using the Conjugate Gradient (CG) method [5] in Lis with a zero initial guess.

Program P_2 in the SILC framework.

1. Prepare matrices A and C and vector $\boldsymbol{x}_0$ in the same way as P_1.
2. Send A, C, and $\boldsymbol{x}_0$ to a SILC server using `SILC_PUT`. In the server, the data is distributed among the server processes.
3. For each time step t_k ($k = 1, 2, 3, \ldots$):
   (a) Send a request for computation by `SILC_EXEC` to compute $\boldsymbol{b}_k = C\boldsymbol{x}_{k-1}$ using a parallel sparse matrix-vector product routine.
   (b) Send another request with `SILC_EXEC` to solve $A\boldsymbol{x}_k = \boldsymbol{b}_k$ using the CG method in the MPI-based parallel version of Lis with a zero initial guess.
   (c) Receive $\boldsymbol{x}_k$ from the SILC server using `SILC_GET`.

The library routines used by P_1 are sequential, whereas those carried out by the SILC server are MPI-based parallel routines. Table 4 shows the computing environments used for Problem 2. We executed both P_1 and P_2 in a node of Xeon4 using a single processor, and measured their execution times for the first 40 time steps using the `gettimeofday` system call. We used the same SILC servers as those in Problem 1 to run P_2. Since P_2 is a sequential program, this configuration corresponds to Configuration (A) shown in Fig. 2.

Table 5 shows the experimental results. In comparison with Program P_2 for Problem 1, P_2 for this problem consumed a relatively large proportion of the execution time in depositing A, C, and $\boldsymbol{x}_0$ into the server and fetching $\boldsymbol{x}_k$ for each time step. Suppose $K = 40$ is the number of time steps, α is the iteration count of the CG method, and $\beta = 5N - 4\sqrt{N}$ is the number of non-zero elements in A and C. Then, the number of floating-point operations for sparse matrix-vector product $C\boldsymbol{x}_{k-1}$ is 2β, while that for solving $A\boldsymbol{x}_k = \boldsymbol{b}_k$ with the CG method is $4N + \alpha(2\beta + 12N + 3)$. Therefore, the matrix computations in P_1 and P_2 require a computation time on the order of $O(\alpha KN)$. On the other hand, data communications between P_2 and a SILC server require a communication time on the order of $O(KN)$. That is, the ratio of the computation time to the communication time is almost proportional to α, which is small compared to N in this test problem. In other words, this problem is somewhat disadvantageous to SILC in the sense that P_2 can hardly yield a speedup in the first place. However, in the experiments we observed speedups of 3.68 using the SILC server in Xeon8 and 7.54 using the server in Altix in the case of $N = 250,000$, by means of faster matrix computations in these remote computing environments.

3.3 Observations

The experimental results of the two test problems showed that the implemented SILC system is capable of achieving speedups when it deals with large problems. Although SILC imposes some communication overhead due to the data transfer between a user program and a SILC server, the overhead can be offset by speedups through the faster matrix computation libraries and computing environment that SILC makes available. The overhead can also be reduced by means of a faster interconnect between the user program and the server.

In addition to the quantitative benefit of speedups, SILC also provides a qualitative benefit in that it enables MPI-based parallel matrix computation libraries to be used not only in MPI-based parallel user programs but also in sequential user programs. In fact, P_2 in Problem 1 was an MPI-based parallel program, while P_2 in Problem 2 was a sequential program. The former used ScaLAPACK and the latter employed Lis, both in a remote MPI-based parallel computing environment. Since the SILC server to be used by a user program can be specified outside the user program, various computing environments as well as the matrix computation libraries that are available in the computing environments can be evaluated one after another without any modification to the user program.

4 Related Work

Improving the utility of matrix computation libraries is a major research topic in the areas of high-performance computing and Grid computing.

The Trilinos project [7] has been proposing a framework for integrating matrix computation libraries into a C++ class library and developing a number of libraries for numerical linear algebra. The application programming interface (API) of each library is consistent with others' in terms of (1) common data structures of matrices and vectors, and (2) common abstract classes based on which users define solvers by inheritance. The libraries are also organized as Trilinos packages by means of (3) common directory structures and installation procedures. However, the libraries vary in the details of their APIs; for example, the API of a dense direct solvers library and that of an iterative solvers library, both developed in the Trilinos project, are not exactly the same, so that users are required to modify their user programs to utilize one library instead of the other in use. In SILC, requests for computation are issued by means of textual mathematical expressions through which any libraries (even having incompatible APIs) can be utilized in the same way in any programming languages.

Amesos [8] is a C++ class library which gives access to various direct linear solvers through a common API. The library provides good support for many existing libraries based on different parallelization techniques, including sequential libraries such as LAPACK and parallel libraries such as ScaLAPACK. Amesos focuses on direct solvers for dense matrices, whereas SILC provides support for a wider range of matrix computations in a language-independent manner.

Since SILC is a piece of middleware based on a client-server architecture, our framework shares some functionalities with Grid computing middleware such as Ninf-G [9] and NetSolve [10]. Ninf-G is a middleware system for realizing Remote Procedure Call (RPC) in Grid computing environments. Ninf-G allows user programs to carry out MPI-based parallel matrix computation libraries in remote distributed parallel computing environments. In Ninf-G, a particular call for a remote procedure takes place in one process; that is, RPC is carried out sequentially either in a sequential user program or in a process of an MPI-based parallel user program [11]. All input and output data is once gathered to one of the remote processes by which the remote procedure is carried out in parallel. In addition, users are required to specify the ways of distributing the input data to the other processes as well as of collecting output data to the sending process, both by means of Ninf-G's interface description language. In contrast, SILC enables data transfer between a user program and a SILC server to be performed in parallel by means of Configuration (B) shown in Fig. 2, allowing the user program and the server to avoid the data redistribution to/from one process before the data transfer. Moreover, users do not have to care about the details of the data redistribution on the server side as long as supported matrix storage formats are in use.

5 Concluding Remarks

This paper described the design and implementation of a SILC system for distributed parallel computing environments. By using this system, MPI-based parallel matrix computation libraries can be utilized not only in MPI-based parallel user programs but also in sequential user programs. Moreover, no modification to the user programs is required to make use of different computing environments. The experimental results of two test problems showed that some speedups are feasible by using faster matrix computation libraries in distributed parallel computing environments via SILC, provided that the amount of matrix computations is large enough to reduce the relative amount of communication overhead due to data transfer between a user program and a SILC server. In the experiments, the implemented SILC system achieved speedups of 2.69 in Problem 1 using ScaLAPACK and 7.54 in Problem 2 using an iterative solvers library Lis, both through a remote SILC server that runs on 16 processes.

The primary subjects of our future study include an implementation of Configuration (C) shown in Fig. 2, a quantitative analysis concerning the cost of data communications, a proposal of a performance evaluation model for distributed SILC with emphasis on the communication overhead [12], and the development of plug-in modules for integrating various existing matrix computation libraries into the SILC framework.

Acknowledgments. The authors would like to thank the session chairperson Dr. Xiaoye Sherry Li and anonymous reviewers for insightful remarks. This research was supported by a grant [13] from the Core Research for Evolutional Science and Technology (CREST) of the Japan Science and Technology Agency.

References

1. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: SILC: Flexible and environment independent interface for matrix computation libraries. In: Wyrzykowski, R., Dongarra, J.J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 928–935. Springer, Heidelberg (2006), `http://ssi.is.s.u-tokyo.ac.jp/silc/`
2. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: LAPACK in SILC: Use of a flexible application framework for matrix computation libraries. In: Proc. HPC Asia 2005, pp. 205–212 (2005)
3. Blackford, L.S., et al.: ScaLAPACK Users' Guide. SIAM (1997)
4. Smith, G.D.: Numerical Solution of Partial Differential Equations. Oxford University Press, Oxford (1965)
5. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM (1994)
6. SSI Project: User's Manual for Lis 1.0.2 (2006), `http://ssi.is.s.u-tokyo.ac.jp/lis/`
7. Heroux, M.A., et al.: An overview of the Trilinos project. ACM Transactions on Mathematical Software 31, 397–423 (2005)
8. Sala, M.: On the design of interfaces to serial and parallel direct solver libraries. Technical Report SAND–2005–4239, Sandia National Laboratories (2005)
9. Ninf Project, `http://ninf.apgrid.org/`
10. NetSolve, `http://icl.cs.utk.edu/netsolve/`

11. Takemiya, H., Tanaka, Y., Nakada, H., Sekiguchi, S.: Development and execution of large scale grid applications using MPI and GridRPC: Hybrid QM/MD simulation. IPSJ Trans. on Advanced Computing Systems (in Japanese) 46, 384–395 (2005)
12. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: A performance evaluation model for the SILC matrix computation framework. In: Proc. IFIP Intl. Conf. on Network and Parallel Computing, pp. 93–103 (2006)
13. Nishida, A., Kotakemori, H., Kajiyama, T., Nukada, A.: Scalable software infrastructure project. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, Springer, Heidelberg (2006)

A Study of Real World I/O Performance in Parallel Scientific Computing

Dries Kimpe[1,2], Andrea Lani[3], Tiago Quintino[1,3], Stefan Vandewalle[1], Stefaan Poedts[2], and Herman Deconinck[3]

[1] Technisch-Wetenschappelijk Rekenen, K.U.Leuven, Celestijnenlaan 200A, BE-3001 Leuven, België
Dries.Kimpe@cs.kuleuven.be
[2] Centrum voor Plasma-Astrofysica, K.U.Leuven, Celestijnenlaan 200B, BE-3001 Leuven, België
[3] Von Karman Instituut, Waterloosesteenweg 72, BE-1640 Sint-Genesius-Rode, België

Abstract. Parallel computing is indisputably present in the future of high performance computing. For distributed memory systems, MPI is widely accepted as a de facto standard. However, I/O is often neglected when considering parallel performance. In this article, a number of I/O strategies for distributed memory systems will be examined. These will be evaluated in the context of COOLFluiD, a framework for object oriented computational fluid dynamics. The influence of the system and software architecture on performance will be studied. Benchmark results will be provided, enabling a comparison between some commonly used parallel file systems.

1 Motivation and Problem Description

1.1 Parallel Programming

Numerical simulation and other computationally intensive problems are often successfully tackled using parallel computing. Frequently these problems are too large to solve on a single system or the time needed to complete them makes single-CPU calculation unpractical.

Successful parallelisation is usually measured by the problem "speedup". This quantity indicates how much faster a given problem is solved on multiple processors, compared to the solution time on one processor. More often than not, this speedup is only based on the computationally intensive part of the code, and phases as program startup or data loading and saving elude the test. Also, when the ratio of computation to the input data is high enough, I/O time is negligible in the total execution time.

However, when scaling to larger problem sizes (and consequently more processors), one often sees that I/O is becoming an increasingly large bottleneck. The main reason for this is that without parallel I/O, the I/O and calculation potential of a cluster quickly becomes unbalanced. This is visible both in hardware and in software; often there is but a single file server managing data for the whole cluster. Moreover, traditional I/O semantics do not offer enough expressional power to coordinate requests, leading to file server congestion, reducing the already limited I/O bandwidth even further.

1.2 Computational Fluid Dynamics and COOLFluiD

Computational fluid dynamics (CFD) deals with the solution of a system of partial differential equations describing the motion of a fluid. This is commonly done by discretizing these equations on a mesh. Depending on the numerical algorithm, a set of unknowns is associated with either nodes or cells of the mesh. The amount of computational work is proportional to the number of cells. For realistic problems this quickly leads to simulations larger than a single system can handle.

COOLFluiD[4] is an object oriented framework for computational fluid dynamics, written in C++. It supports distributed memory parallelisation through MPI, but still allows optimized compilation without MPI for single-processor systems. COOLFluiD utilises parallel I/O for two reasons. One is to guarantee scalability of the code. The other is to hide parallelisation from the end user. During development, a goal was set to mask the differences between serial and parallel builds of COOLFluiD as much as possible. This, among other things, requires that the data files used and generated by the parallel version do not differ from those in the serial version. This depends on parallel I/O, as opening a remote file for writing on multiple processors using posix semantics is ill defined and often leads to corrupted files.

1.3 I/O in a Parallel Simulation

There has been much research on the optimal parallel solution of a system of PDEs. However, relatively little study has been devoted to creating scalable I/O algorithms for this class of problems.

Generally speaking, there are three reasons for performing I/O during a simulation. At the start of the program, the mesh (its geometric description and an initial value for each of the associated unknowns) needs to be loaded into memory. During the computation, snapshots of the current solution state are stored. Before ending the program, the final solution is saved.

In a distributed memory machine, the mesh is divided between the nodes. Consequently each CPU requires a different portion of the mesh to operate. This offers opportunities for parallel I/O, since every processor only accesses distinct parts of the mesh.

Figure 1 shows an example of a typical decomposition, and the resulting I/O access pattern. On the left, the partitioned mesh is shown. On the right, the file layout (row-major ordering) can be seen. Color indicates which states are accessed by a given CPU.

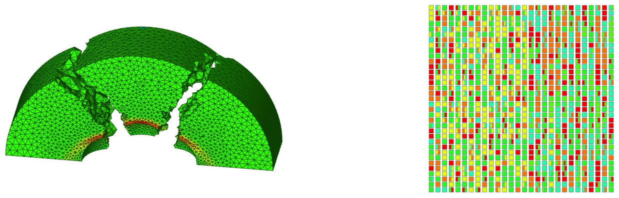


Fig. 1. Decomposition and file access pattern of a 3D sphere

2 I/O Strategies

Within COOLFluiD, I/O is fully abstracted. This simplifies supporting multiple file formats and access APIs, and allows run-time selection of the desired format. Mesh input and output is provided by *file plugins*. A file plugin offers a well defined, format independent interface to the stored mesh, and can implement any of the following access strategies:

Parallel Random Access: This strategy has the potential to offer the highest performance. It allows every processor to read and write arbitrary regions of the file. If the system architecture has multiple pathways to the file this can be exploited. File plugins implementing this interface enable all CPUs to concurrently access those portions of the mesh required for their calculations.

Non-Parallel Random Access: In this model, the underlying file format (or access API) does not support parallel access to the file. Only a single CPU is allowed to open the file, which will be random accessible. This strategy can be used with data present on non-shared resources, for example local disks.

Non-Parallel Sequential Access: Sometimes the way data is stored prohibits meaningful true parallel access. For example, within an ASCII based file format, it is not possible to read a specific mesh element without first reading all the previous elements. This is due to the varying stride between the elements. As such, even when the OS and API allow parallel writing to the file, for mesh based applications, this cannot be done without corrupting the file structure. Note that applications that do not care about the relative *ordering* of the entries in the file can still use parallel I/O to read and write from this file (using shared file pointer techniques). However, as this article studies I/O patterns for mesh based applications this is not taken into consideration.

3 Performance Testing

Currently, obtaining good parallel I/O performance is still somewhat of a black art. By making use of the flexibility COOLFluiD offers concerning mesh I/O, an attempt is made to explore and analyse the many different combinations of file system, API and interconnect that can be found in modern clusters.

3.1 Test Description

We will concentrate on the parallel random access pattern, since the other two access strategies are inherently non-scalable (when considering I/O bandwidth). Although COOLFluiD supports them, they are offered as a convenience. For large simulations, converting the mesh to one of the formats supporting true parallel random access is recommended.

Figure 2 shows the software invoked during mesh transfers. COOLFluiD has file plugins that utilise a storage library (HDF5[3] or PnetCDF[7]) or that directly employ MPI-IO to access the mesh. Internally, these storage libraries rely on the I/O functions of MPI to access raw files. ROMIO[6] is an implementation of these I/O functions, and is used in almost all research or open source MPI implementations. ROMIO has a number of ADIO (abstract-device interface for I/O) drivers providing optimized access to a certain file system. While PVFS2[2] and NFS have specific ADIO implementations, Lustre[1], aiming for full POSIX compliance, is accessed through the generic "UFS" driver.

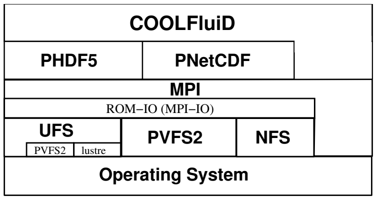


Fig. 2. Software stack for parallel mesh I/O

For testing, the time needed to access a set of unknowns (of a given dimension) will be measured. These unknowns are stored as a linear sequence of "states" (figure 3), each state consisting of a number of doubles. The storage library (HDF5, PnetCDF) is responsible for the mapping between the virtual layout ($n \times d$ doubles) and the file layout (a linear byte sequence). In general, each state is only accessed by one CPU. However, states on the border of a partition will be accessed by multiple CPUs. States are loaded or stored in groups, where the group size is determined by the buffer size. Since MPI-IO requires file datatypes to have positive type displacements, states need to be addressed in increasing order (for a given CPU). This means that in each access round, a CPU will access $\frac{buffer\_size}{sizeof(double) \times d}$ states. Because the state partitions are balanced to evenly distribute the computational cost between the CPUs, this also causes the I/O load to be balanced.

3.2 Test Hardware

All tests were conducted on VIC, a 862 CPU cluster located at K.U.Leuven. The cluster has a number of different interconnect fabrics. All nodes possess a gigabit

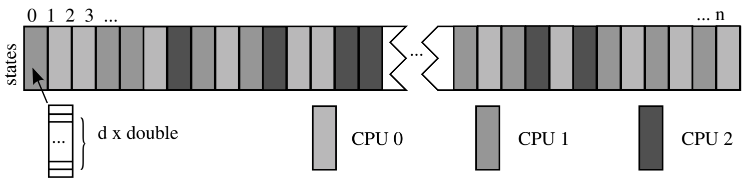


Fig. 3. (Virtual) file layout of the unknowns

ethernet connection. Two 144 port infiniband (4X) switches provide infiniband connections to most of the nodes.

For PVFS2[2], 4 I/O servers were employed, each server having 2 opteron CPUs. Data is stored locally on a SATA disk attached to the node. The disk has a raw read bandwidth of approx. 50 Mb/s. One of the I/O servers doubles as metadata server. Connections between the servers and clients were made using native Infiniband. PVFS2 version 1.5.1 was used.

The Lustre[1] file system used for testing ran on the same 4 servers. Here too, one of them performed both metadata (MDS) and storage (OST) functions, while the others only served as storage servers. Connections were made using IPoIB, an IP emulation mode running over the infiniband network. All files were striped over the available I/O servers. The Lustre version was 1.4.6.

A dedicated server (of the same type) was installed to export the NFS file system, also using IPoIB. The `async` option was enabled, allowing the server to cache writes in order to increase performance.

Eight nodes were reserved as I/O clients. Only one of the two opteron CPUs from every node was used, avoiding contention for the network ports.

All nodes were installed for the purpose of this article, in order to exclude any interference from other jobs running on the cluster.

3.3 MPI-IO File Hints

The MPI-2 file interface enables the user to specify implementation specific hints. These hints can be used to communicate additional information to the underlying software layers. However, an implementation is free to ignore hints. Both PnetCDF and HDF5 allow the user to specify hints, which are subsequently passed unmodified to MPI-IO during file access.

Unfortunately, most hints are useless in combination with high level storage libraries. In order to specify meaningful hints, an application needs to know intimate details of the underlying file layout. However, the goal of a storage library is to abstract this underlying file layout and to present a higher level interface to the application. Because of this, and also considering the fact that an implementation can ignore hints, the influence of file hints on performance was not studied.

3.4 MPI-IO on PVFS2

Since both HDF5 and PnetCDF rely on MPI-IO for actual file access, ROMIO performance can help explain their test results, therefore pure MPI-IO performance will be discussed first. PVFS2 will be accessed through the user-space PVFS2 ROMIO driver, which does not do client-side caching. As such, OS file caches do not influence the performance.

Figure 4 shows the read and write performance of a number of MPI access methods. (test dataset: 400000 x 8 doubles) For MPI-IO, two completely different access methods were studied. For the first (the left graph), no file datatypes were used. This is referred to as “level 0” for independent accesses, and “level 1” for collective accesses[8]. The second method does use file datatypes, which make it possible to address full non-contiguous access pattern in one operation. This method is known as “level 2” for independent and “level 3” for collective accesses.

Additionally, each method was tested using a combination of the following optimizations:

optimize: Try to group adjacent requests into larger ones. This is done by the *client* application, before passing the request to MPI-IO or the storage library.

typed: Use a one-dimensional array of an array type (with base type double) instead of a two-dimensional array of doubles. Since there was no releveant performance difference between typed and non-typed tests, this data was omitted from the graphs.

collective: Use `MPI_File_write_all` instead of `MPI_File_write`.

Selecting the right I/O method can make a huge difference in I/O performance. Level 0 and 1 lead to unuseable performance (less than 1 Mb/s!). Although level 2 performs a little bit better, the graph already indicates a scaling problem, even with eight nodes for four I/O servers! Only level 3 I/O leads to acceptable performance.

3.5 PnetCDF and HDF5 on PVFS2

PnetCDF currently doesn’t offer a suitable API for unstructured dataset access. Because of this, client software is forced to repeatedly call the library to access a small number of elements, leading to many small read or write operations (level 0/1). Although ROMIO is capable of aggregating and grouping some of these requests, the data volume is still too small to really benefit from this. For the same reason, collective I/O (which has some additional overhead) performs even worse.

Figure 4 shows the results. Because of the slow I/O speed, a small test dataset (50000×8) was choosen. This dataset was actually smaller than the buffer size. On one CPU, when optimization of the access pattern was enabled, this resulted in one large read/write request of the full dataset. Therefore, speed measurements with only one client node were omitted from the graph when optimization was enabled. Because of the huge difference between the performance of the

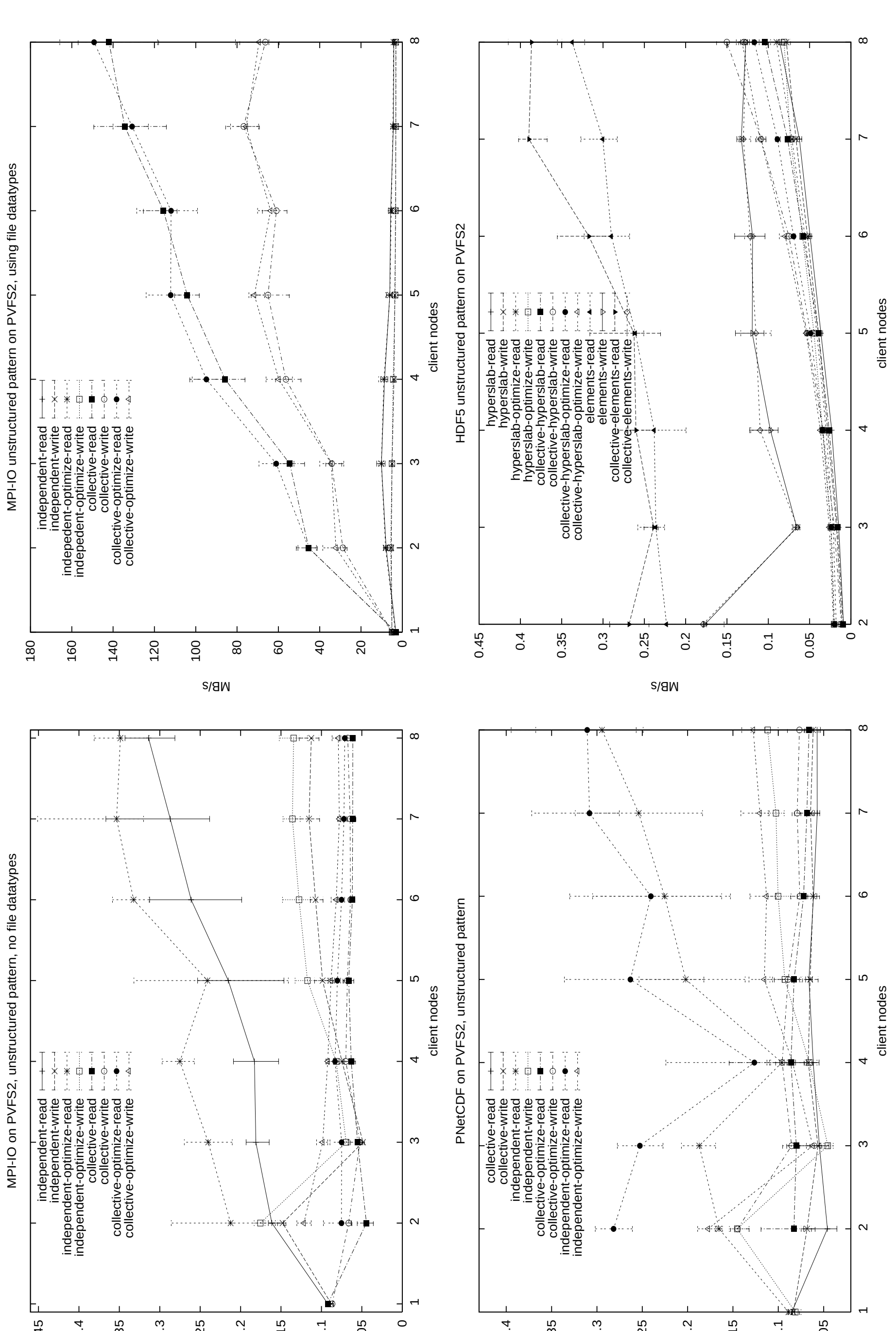


Fig. 4. unstructured access pattern performance on PVFS2

different I/O levels, the performance of the PnetCDF library (for this access pattern) ultimately is determined by the access pattern presented by PnetCDF to MPI-IO. The graph clearly resembles the MPI-IO graph (without file datatypes).

Figure 4 also demonstrates that HDF5 also has serious problems dealing with unstructured access patterns. Although the API offers two ways to setup the access pattern, both have problems. A first way is to use a union of hyperslabs. The `H5S_select_hyperslab` call allows extending an existing dataset selection with a specified hyperslab. By repeatedly calling this function, the full access pattern can be described. Unfortunately, every time the function is called, it loops over the old selection to make sure no duplicate selections exist. This causes the setup time of the access pattern to become unreasonably large, making it even slower than the actual data transfer.

The second method uses the `H5S_select_elements` call, which allows the user to specify an array of coordinates of points that have to be read. Although application-level optimization (by grouping adjacent points) is not possible with this call, the access pattern can be described in one fast function call. However, internally, HDF5 (currently) does not relay this access pattern to MPI-IO. Instead, it is broken up again in seperate one-element read operations, and read by independent I/O requests. Because of this, its performance is comparable to that of pure MPI-IO using independent accesses without file datatypes.

Results with only one client node were not obtained, as the latest stable HDF5 release (1.6.5) contains a bug preventing its use on only one CPU when using userspace ROMIO drivers.

3.6 Unstructured MPI-IO Access on NFS and Lustre

The previous tests demonstrated that PnetCDF performance closely follows that of MPI-IO with the same access pattern. For HDF5, at least when using `H5S_select_elements` style selections, this is also the case. The other HDF5 method, a hyperslab union, is limited not by I/O time but by the setup time. Therefore, only MPI-IO performance will be shown for Lustre and NFS.

In figure 5 can be seen that NFS and PVFS2 had different design goals. NFS was designed as a general purpose network file system, optimized to handle multiple small file requests. As such, for unstructured access without file datatypes (causing small requests), on average NFS performs better than PVF2 *if the number of clients is small.* For more than 4 clients, the NFS server cannot handle the load any more. Since NFS uses UDP – an unreliable transport protocol – lost packets triggering retransmission delays cause a serious performance drop. This can clearly be seen for collective operations, which by their nature increase network contention. For 5 client nodes and more, transfer speed approached zero as some transfers took multiple hours to complete. For this reason, no performance numbers were obtained for collective modes with more than 6 clients.

When using file datatypes, by aggregating data, collective calls are able to improve performance. Because of the small dataset size, from 6 or more clients on, performance becomes less predictable due to client side caching. If (part of) the data is in the client side cache, in addition to avoiding the data transfer, load

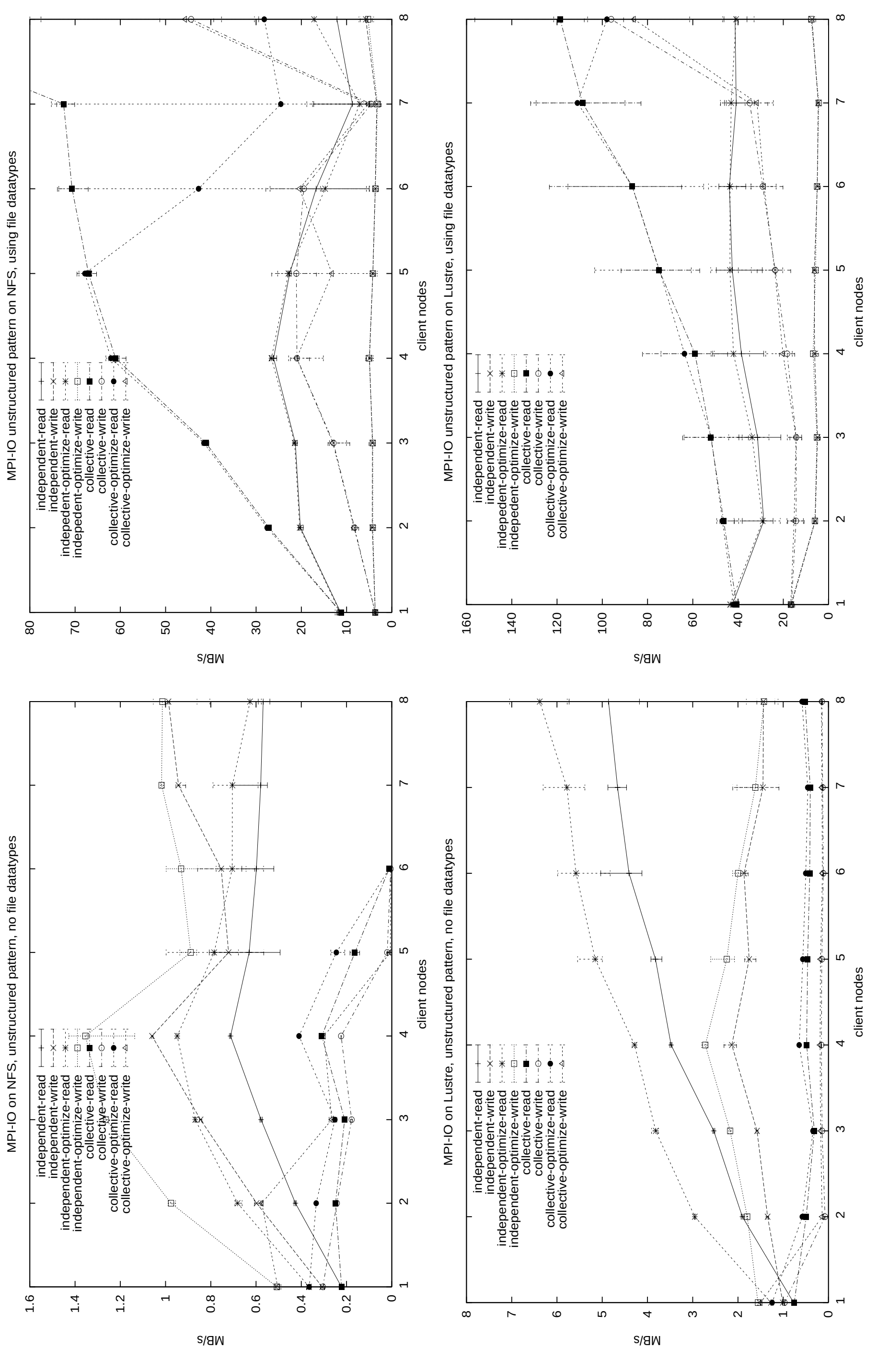


Fig. 5. MPI-IO performance on NFS and Lustre

on the NFS server is reduced, resulting in extra performance. Application-based merging of adjacent access requests is a delicate issue on NFS. When dealing with very small requests some benefit can be seen. However, when using level 3 I/O (collective mode and file datatypes), optimizing collective read patterns can cause a 10-fold performance drop. This is probably due to network or server congestion. The graph shows the average, minimum and maximum transfer speed obtained. For collective reading with 6 or more clients, the avarage is misleading; In reality, either very high or very low numbers were obtained.

The Lustre file system performs very well for level 0 I/O. This is partly due to the client side cache, write accumulation and read-ahead. By default, on the client nodes, Lustre utilized up to 1500 MB for client side caching. Also, a maximum read-ahead window of 40MB was automatically set. Writes are accumulated until at least 1MB of dirty pages is available.

When doing independent reads without file datatypes, these optimizations enable the best performance of all file systems tested. Independent write speed is still adequate, but performance drops when the number of clients increases. This is probably due to lock contention.

However, collective operations without file datatypes need to be avoided on Lustre. From two clients on, transfer speeds become unworkable. As is the case with NFS, collective operations cause the most lock contention and this translates into low performance. PVFS2, which does not have file locking, is not affected by this.

Looking at level 2 and 3 I/O (independent and collective using file datatypes), graph 5 shows nice results. In independent access modes, there is the usual performance drop going from one to two clients, due to the locking protocol. Collective access is less affected, because of its ability to avoid issuing small requests[9].

4 Conclusion

As a first conclusion one can state that the most important factor influencing performance is the access pattern. Using contiguous accesses results in much better performance, up to an order of magnitude.

The large gap in transfer speed between NFS, a traditional shared file system, and true parallel file systems such as Lustre and PVFS2 is clearly visible. Even with only 8 clients, the NFS server load became unreasonably high, demonstrating the need for scalable I/O solutions.

For non-contiguous access patterns (such as those resulting from unstructured meshes) performance using pure MPI-IO is adequate when using collective I/O and file datatypes. However, at this time, none of the tested storage libraries is ready for these kind of acccesses. Unless these libraries can accept a non-contiguous file access pattern, and pass this information information on to MPI-IO, they cannot be used. Until they do, if true non-contiguous file access is really needed, MPI-IO should be utilized.

However, if eventually all data needs to be accessed, utilizing an application level parallel cache will outperform any non-contiguous file access method. In such a scheme, the application would first read all data in contiguous chunks, store everything in a parallel cache, and serve all future (non-contiguous) requests from this cache. Also, although this could equally wel be done by MPI-IO (or the storage library), application-level grouping of read and write requests slightly increases performance.

PVFS2 is easy to install (no superuser access required), is well supported by research MPI implementations and performs very well compared to commercial file systems such as Lustre. PVFS2 enables end-users to easily setup and run I/O servers alongside their jobs, allowing scalable I/O on any cluster offering local disk access.

References

1. Lustre: A Scalable, High-Performance File System. white paper (November 2002), http://www.lustre.org/docs/whitepaper.pdf
2. Latham, R., Miller, N., Ross, R., Carns, P.: A Next-Generation Parallel File System for Linux Clusters, LinuxWorld, vol. 2 (January 2004)
3. HDF5: http://hdf.ncsa.uiuc.edu/HDF5/
4. Lani, A., Quintino, T., Kimpe, D., Deconinck, H., Vandewalle, S., Poedts, S.: The COOLFluiD Framework: Design Solutions for High-Performance Object Oriented Scientific Computing Software. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2005. LNCS, vol. 3514, pp. 281–286. Springer, Heidelberg (2005)
5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: PVM/MPI 2004, pp. 97–104 (2004)
6. Thakur, R., Gropp, W., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation, pp. 180–187 (October 1996)
7. Li, J., Liao, W.-k., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: Proceedings of SC2003, Phoenix, AZ (November 2003)
8. Thakur, R., Gropp, W., Lusk, E.: A case for using MPI's derived datatypes to improve I/O performance. In: Proc. of SC98: High Performance Networking and Computing (1998)
9. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proc. of the 7th symposium on the Frontiers of Massively Parallel Computation, pp. 182–189 (February 1999)

Epitaxial Surface Growth with Local Interaction, Parallel and Non-parallel Simulations

Carmen B. Navarrete[1,2], Susana Holgado[1], and Eloy Anguiano[1,2]

[1] Dpto. Ingeniería Informática, Escuela Politcnica Superior,
Universidad Autónoma de Madrid, ES-28049 Madrid, Spain
{carmen.navarrete, susana.holgado, eloy.anguiano}@uam.es
[2] Centro de Referencia Linux (CRL, UAM-IBM), Office B-206,
Escuela Politnica Superior, Universidad Autónoma de Madrid,
ES-28049 Madrid, Spain

Abstract. Usually, theories of surface growth are based on the study of global processes without taking into account the local behaviour of atoms. In this work we present two simulations making use of a parallel computing library. These two simulations are based on a simple model that allows us to simulate the surface growing process of a certain material. The first one is a quasi-static model whereas the second recreates the atomic interaction considering the free atoms in continuous movement along the surface. Both simulations make use of local principles of thermodynamic for atomic deposition, relaxation and diffusion of a growing surface. The obtained results agree with those that use global theories and with experimental results of Scanning Tunneling Microscopy (STM).

1 Introduction to the Physical Model

Surface growth of materials is a very well studied topic from the theoretical and experimental point of view [1] [2] [3] [4] [5] [6]. The surface structure of certain materials presents a granular morphology randomly distributed along the surface, with hills of various heights and lateral extensions (structure typically named as mounds) [7] [8] [9] [10]. This property as the roughness (with less significance) are properties which behaviour has been defined as fractal in several times [11] [12] [13].

Film growth involves two sciences: the thermodynamics and the kinetics. The kinetics appears because of the continuous movement of the atoms on the surfaces due to the presence of thermal, kinetics and chaotic fluctuations [14]. The thermodynamics gives the conditions of nonbalance due to the temperature of the substratum. By calculating these parameters we can grow a surface, initially layer by layer until completing the first ones and then grow the surface making mounds. In this case, the growth process is named epitaxial growth [15].

2 Simulation

We want to simulate the epitaxial surface growth of materials taking into account the atomic interaction between free atoms that are in continuous movement along the surface. This is a tightly coupled Laplacian problem with very high requirements of memory and computations.

The two simulation we have implemented use two Monte Carlo methods: one at the spatial-temporal dimension, determined by the deposition flux, and related to the temperature of the substratum. Another at the spatial dimension also related to the temperature.

These simulations [16] are both based on local behavior, calculating the partition function of the accessible states of the atom (Fig. 2). This allows each atom to know its probability of diffusion towards the next position. One by one, these atoms will randomly choose, based on these probabilities, the direction in which they finally will move. As it is a stochastic process, the direction of each movement is not always the most favorable for optimizing the total energy of the substratum. Once all the atoms have been diffused to their positions, the deposition process will start again, repeating the algorithm explained in Figure 1 a fixed number of times, according to the temperature of the growing interface.

```
begin
    foreach $layer \leftarrow 0$ to $MONOLAYERS$ do
        foreach $deposited\_atom \leftarrow 0$ to $DEPOSITIONS$ do
            $Random\_Deposition$

            foreach $diffunded\_atom \leftarrow 0$ to $ATOMS$ do
                $probability \leftarrow Calculate\_Diffusion\_Probability()$
                if $probability > VALUE$ then
                    /* the atom diffunds                                  */
                    $energy \leftarrow Calculate\_Energy()$
                    $Calculate\_Direction(energy)$
                else
                    $Move\_Atom()$
                end
            end
        end
    end
end
```

Fig. 1. Pseudocode of the epitaxial quasi-static surface growth

The parallel version uses the LAM-MPI API [17] [18] [19]. The main difference between this algorithm and the non-parallel one is that this one divides the computation into several processes. These processes can be of two different kinds: the slaves that simulate the deposition of atoms, the calculus of the energies contributed by the neighboring atoms, the diffusion of the atom on the image and the simulation of the interaction. The master only manages the synchronization of the different slave processes. This election has been adopted in order to clearly separate the communication between MPI processes, the synchronization with

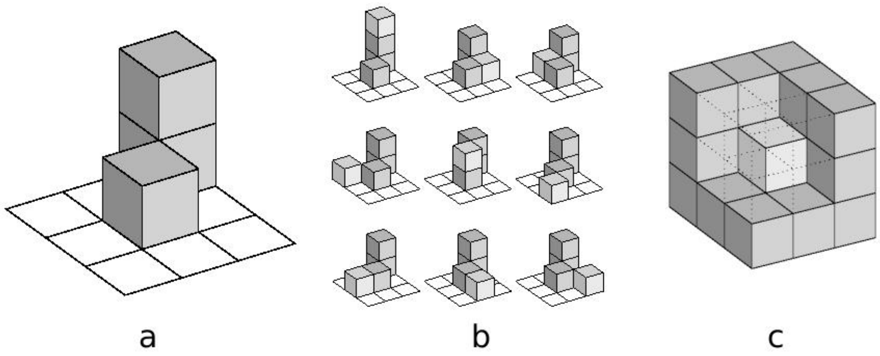


Fig. 2. a) Example of the surface in a certain moment of the simulation. b) Accessible states of one deposited atom. Elemental diffusions that can be done by an atom centered in a 3×3 matrix based on the previous surface. c) First neighbors area that contributes with energy to the central atom. Calculus of the global energy of an atom.

all the nodes of the cluster and the algorithm for the problem to be solved. To implement the communication between all processes we have studied several structures of queues that also permit us to study different models of atomic interactions in the growing surfaces.

3 Domain Decomposition

Different computations have been performed: the sequential (a quasi-static model with no possible atomic interaction) and the parallel simulations. For the parallel one, the domain decomposition depends so much, not in the physical result of the execution (since this one only depends on the material and on the distribution of randomness of positions) although it is related to the optimization and speed of processing. Communication between processes take time that physically can be considered as a variation of temperature. Thus it is very important to get a good domain decomposition that optimizes the communication between processes. The domain decomposition consists of a distributed or shared solution. We have studied diverse domain decompositions that can be resumed in these ones:

– **Partitioning the image in rectangles:** a distributed solution in which the image is partitioned in horizontal or vertical stripes depending on the number of slaves, value known as N. The most important advantage of this method is the simplicity of the management of the data from the master node to the slaves. The main disadvantage of this decomposition resides in that it implies different rates of growth between subimages due to the differences between the speed of the processors. Also we have to keep in mind that atoms are in continuous movement along the surface and nothing can assure us that an atom would not leave its subimage in its movement. If it was

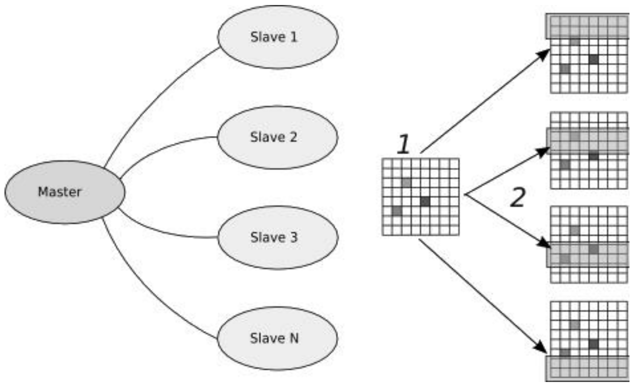


Fig. 3. (1) Image at the master process. (2) Domain decomposition for the problem with N slave processes.

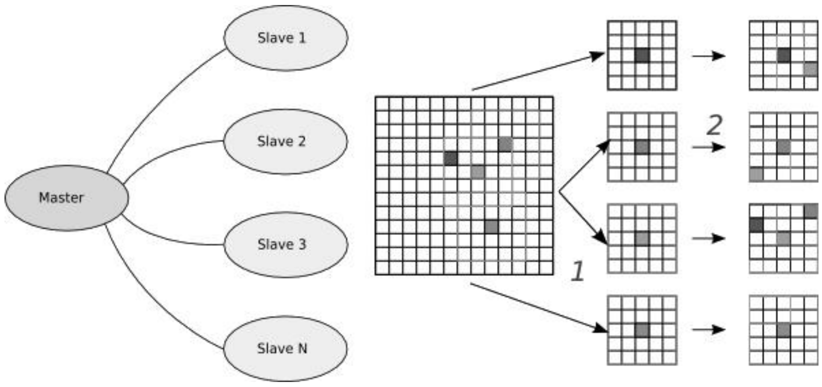


Fig. 4. (1) Image at the master and decomposition in environments. (2) 5×5 matrix and environments at the slaves.

deposited at another subimage, that could not be in the same slave. Due to this computation model the interaction among processes are too intensive, spending much more time in the construction of communication messages than in the simulation (Fig. 3).

– **Partitioning the image according to the environment of the atom:** we define the environment of an atom as a matrix of 5×5 elements that contains the information about the first neighbors and about the second ones (distance of 2 units in each direction) of a certain atom. In this, both distributed and shared solution, the environments of the atoms are calculated dynamically by the master process. This is the only information that the master sends to the slave. Thus, this method saves much memory in the slaves. The disadvantage resides in that this partition of the image takes too much time, and the real computation on each step is so small that it is hard to get good speedup (Fig. 4).

– **Replicating:** a shared solution in which all slaves have the image replicated from the master process. The master and the slaves have shared the data structure that represents the surface. This allows the slaves to calculate the bonding energy of an atom without knowing what any other node of the cluster have calculated. The main disadvantage is that all slaves have to store the image of the surface in their memory (Fig. 5).

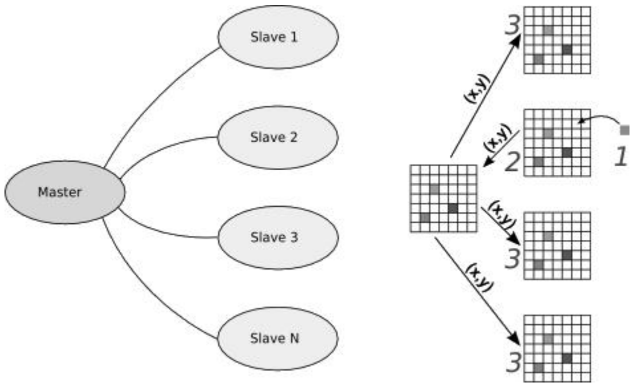


Fig. 5. (1) Calculation of the final position of the atom by the slave. (2) The slave send that position to the master. (3) The master notifies all slaves of the position by inserting the result in each queue.

4 Atomic Interaction

To simulate the atomic interaction between atoms in diffusion, we need to use a queue structure. The atomic interaction and the deposition flux depends on the queues and their treatment. As we have seen, the queues are essential to discouple the two Monte Carlo methods that are involved in the solution of this physical problem. We have performed several tests of random number generation with the SPRNG Library [20] to prove that the Monte Carlo Methods from the point of view of the random number generation were totally discoupled. The results shown no measurable difference between using this library or not. The main difference between these two methods is the execution time. This is why we use the less computationally expensive algorithm, that is the simple random number generation.

The queue structure (Fig. 6) consists of a double data array managed by the master process. The data structure allocated in the queue consists of two data structures. Each structure contains two coordinates to identify the position of one atom in the image. The two points represent the movement of the atom on its diffusion process, from the first position towards the second one. This

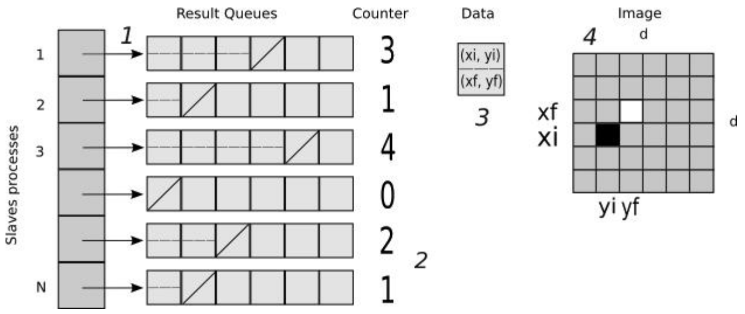


Fig. 6. Queue structure at the master process. (1) Double array data structure. (2) Entries counter for readjusting the tasks of the slaves. (3) Data structure allocated on the queue. (4) Representation of the movement of the atom according to the information shown in the data structure.

structure also contains a counter that defines the occupation of the queue. This counter is needed for readjusting the tasks for each process allowing us not to have to consider the speed of processing of each node of the cluster.

5 Results

Analyzing the results of these domain decompositions, we have chosen the pure replication solution, just because this is the one that better fits the problem of the atomic interaction, from the point of view of the optimization and speed of processing, due to the minimization of the communications between the master and the slave processes.

In order to verify the exactitude of the simulator, the results obtained with existing results were compared to results of STM experiments [7] [8]. This analysis verify that the results are similar to the ones obtained experimentally, as it can be seen in Figure 7.

From the point of view of the execution time we can observe that the non-parallel algorithm takes much time, even more if we try to use it to recreate the atomic interaction. The results of the parallel version have been obtained executing the application in a 8-MPI heterogeneous NUMA [21] LAM-MPI [17] [18] [19] cluster. 4 of those 8 processors were a simple Pentium–III Clamath and the other 4 were the processors of an IBM e-Server SMP–Xeon.

Different computations have been performed. We can not conclude that the parallel algorithm is better than the sequential only by considering the execution time. This is an unfair comparison since algorithm 2 is based on a slightly different model as it regards additional physical interactions (Fig. 8). But we can conclude that the parallel algorithm is better because it permits us to study the atomic interaction in the growing surfaces in a reasonable time.

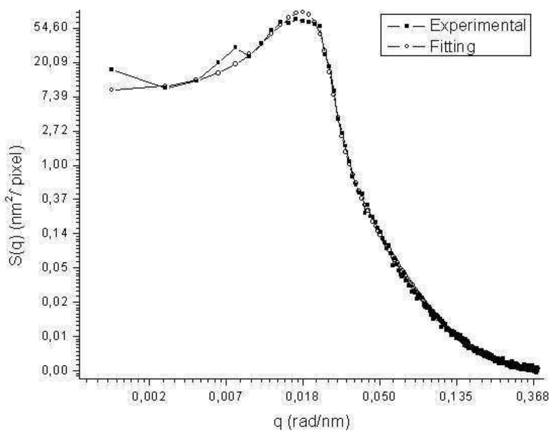


Fig. 7. Fitting between the experimental and the simulated results

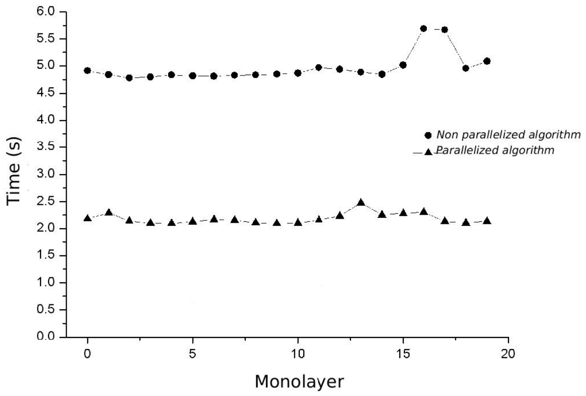


Fig. 8. Comparison of the execution time between the two developed simulators. The growth parameters were: energy $\epsilon = e/k_bT$ =0.5 and dimension $d = 32$ $pixels^2$.

References

1. Sander, L.M., Meakin, P., Ramanlal, P., Ball, R.C.: Ballistic deposition on surfaces. Physics Review A, 34 (1986)
2. Krug, J.: The columnar growth angle in obliquely evaportad thin films. Matterwissen Werkstofftech 26 (1995)
3. Viseck, T.: Fractal growth phenomena. World Scientific, Singapore (1989)
4. Hara, T.: A stochastic model and the moment dynamics of the growth of amorphous films. J. theor Biol. 109 (1984)
5. Meakin, P., Krug, J., Kassner, K., Family, F.: Laplacian needle growth. Europhys Letters 27 (1993)

6. Krug, J.: Origins if scale invariance in growth processes. Taylor & Francis Ltd (1997)
7. Oliva, A.I., Anguiano, E., Sacedón, J.L., Aguilar, M., Mendez, J.A., Aznarez, J.A.: Extended statistical analysis of rough growth fronts in gold films prepared by thermal evaporation. Physics Review B, 60 (1999)
8. Oliva, A.I., Sacedon, J.L., Anguiano, E., Aguilar, M., Aznarez, J.A., Mendez, J.A.: Surface Science 417 (1998)
9. Hoffmann, H., Vancea, J.: Thin Solid Films 85 (1981)
10. Peto, G., Norman, S., Anderson, T., Somogyi, S.: Thin Solid Films 77 (1981)
11. Mandelbrot, B.B.: The fractal geometry of Nature. W.H. Freeman, New York (1982)
12. Meakin, P.: Fractals, Scaling & Growth Far from Equilibrium. Cambridge University Press, Cambridge (1998)
13. Family, F., Viseck, T.: Dynamics of fractal surfaces. World Scientific, Singapore (1989)
14. Mullins, W.W.: Solid Surface morphologies governed by capillarity, metal surfaces structure, energetics and kinetics. Gjostein, N.A., Robertson, W.D. (1963)
15. Michely, T., Krug, J.: Islands, Mounds, and Atoms. Patterns and Processes in Crystal Growth Far from Equilibrium, Springer Series in Surface Sciences, vol. 42 (2004)
16. Holgado, S., Navarrete, C.B., Anguiano, E.: Surface growth modeled by ab-initio Monte Carlo simulation. Trends in NanoTechnology'04 Segovia
17. Requirements Specification Andrew. Reliability in LAM/MPI
18. Burns, G., Daoud, R., Vaigl, J.: LAM: An open cluster environment for MPI
19. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230 (1994)
20. Mascagni, M., Chi, H.: Parallel linear congruential generators with sophie-germain moduli. Parallel Comput. 30(11), 1217–1231 (2004)
21. Culler, D.E., Gupta, A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, San Francisco (1997)

Data Dependence Analysis for the Parallelization of Numerical Tree Codes

Gerhard Zumbusch

Friedrich-Schiller-Universität Jena,
Institut für Angewandte Mathematik,
Ernst-Abbe-Platz 2, 07743 Jena, Germany
zumbusch@mathe.uni-jena.de
http://cse.mathe.uni-jena.de

Abstract. Data dependence analysis for automatic parallelization of sequential tree codes is discussed. Hierarchical numerical algorithms often use tree data structures for unbalanced, adaptively and dynamically created trees. Moreover, such codes often do not follow a strict divide and conquer concept, but introduce some geometric neighborhood data dependence in addition to parent-children dependencies. Hence, recognition mechanisms and hierarchical partition strategies of trees are not sufficient for automatic parallelization. Generic tree traversal operators are proposed as a domain specific language. Additional geometric data dependence can be specified by code annotation. A code transformation system with data dependence analysis is implemented, which generates several versions of parallel codes for different programming models.

1 Introduction

Automatic parallelization of general sequential, imperative code is one of the ultimate goals of compiler construction. Numerical algorithms in scientific computing can often be parallelized efficiently with a data parallel approach. Basically there are three related problems to address: First the data partition problem, second the mapping problem where sets of partitions are mapped to a processor, and third the data dependence analysis where to add communication operations in a distributed memory environment and synchronization operations in a shared memory environment. Both partitioning and mapping can be hard problems depending on the data dependence graph. The data dependence analysis itself can be technically too complex for compilers. However, often there exist good solutions for the problems known in the specific area of application. For example additional geometric information along with a geometric partition of data may work very well and may break general NP-hard problems, but such a solution can be impossible to derive solely from a given sequential code. Hence, parallelization of codes written in a domain specific languages may be possible, while parallelization in general is not feasible.

In this paper, we restrict ourselves to hierarchical tree methods, such as fast summation algorithms for N-body simulations and the fast multipole method.

Given a large number of geometric entities, the numerical algorithms approximate the sum over expensive $N(N-1)/2$ pair wise interactions by combining the action of groups of entities with others distant away. This can be done hierarchically, which leads to an unbalanced k-ary tree. Algorithms work bottom-up for the computation of groups, top-down for the action of groups on other groups or entities and use local neighborhood data on each tree level for pair wise interactions. The overall complexity under reasonable assumptions is reduced to $\mathcal{O}(N\log N)$ or $\mathcal{O}(N)$, depending on the algorithm and the analysis.

There are a number of parallel implementations of such tree algorithms, among them shared [1] and distributed memory implementations [2,3] and references therein. The latter follow a data parallel style with a distributed tree data structures which contains all geometric data. Operations on the tree are subdivided into operations on a common coarse tree part including the tree root and into distributed operations on finer sub-trees exclusively performed and stored on a single processing element. Except for embarrassingly parallel communicationless algorithms, there are algorithms which require top-down or bottom-up data exchange, or some data geometrically close to the tree node. In many cases this can be assembled into a single global communication step and arranged as local tree traversal before or after the communication step. Subtle changes of the numerical algorithm however may cause a more elaborate communication scheme like a data exchange at each tree level or a dynamic process of request and serve processes.

2 Dependence Analysis

A major part for automatic code parallelization is data dependence analysis [4,5]. Efficient distributed memory parallelization however requires global data dependence analysis. There are partial solutions of this problem for loops and array languages like in HPF, parallel libraries [6] and parallel skeletons [7], which have been applied to divide-and-conquer operations [8]. For a detailed overview, see [9].

As an example of a parallel domain language, we consider numerical codes using data organized as one large tree. Algorithms doing so are fast summation techniques like the fast multipole method or some hierarchical grid solver for partial differential equations. Such structures currently cannot be handled automatically in high performance languages and compilers.

The computational atom of the tree algorithms to be discussed is a node data structure. In C++ this is typically a class with some data members and member functions. For tree traversal in the unbalanced tree there are methods to access the child nodes. Furthermore, there may be functions to access nodes in the geometric neighborhood, see Fig. 1. Numerical algorithms on the tree consist of a specification of the (partial) tree traversal along with some operations on the data members of each tree node visited. Of course, there are many different ways to express this [10]. However, for parallelization and for dependence analysis it is favorable to separate tree traversal from operations on the nodes. Even further, it is often possible to derive the type of tree traversal from the dependence analysis of the operations and to omit the tree traversal code. Hence, tree algorithms can be put into a generic tree library, while the user code specifies the operations on a single tree node. Note that tree creation requires data partitioning and thus requires further domain specific information. Often, some geometric data decomposition can be used, which can also be put into the library.

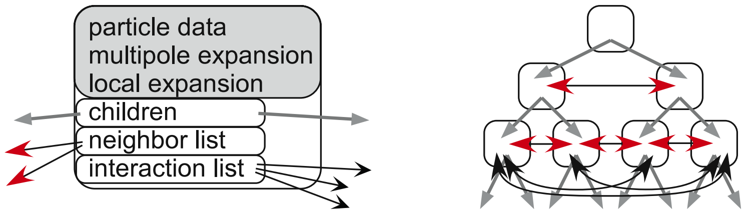


Fig. 1. A sample tree node data structure (left) and the related binary tree (right) for a fast multipole summation

We are left with the data dependence analysis of the operations on a single tree node. A fine scale dependence analysis done by optimizing compilers is not needed here, but solely the relation of data members of the current tree node with other tree nodes. Currently we use a set of m4 scripts to create code, the g++ compiler to process it, and a set of perl script to analyse the code's output. Based on a parallel tree library, this way a dependence analysis of the user code can be performed. The result is a parallel distributed memory message-passing code or a shared memory parallel pThread code or a hybrid message-passing/ pThread code for distributed memory systems with multiprocessor nodes.

A path matrix dependence analysis of the operations on a single tree node is performed, see [5]. The read and write operations on all data members of a node and on all nodes accessed relative to that node are recorded for different stages of the algorithm. A standard bottom-up tree traversal for example will read data of the node and its children and write data of the node, see Fig. 2. Some top-down tree traversal may only write data of the node's children. A detailed comparison of data members read and written reveals flow dependencies which cannot be parallelized this way. Hence, it is possible to verify that some algorithm can be parallelized and to create the actual code for send and receive of the necessary data members at some stage of a parallel tree traversal. Further, the dependence pattern can be used to determine the type of tree traversal and therefore the processor communication pattern. The same information is also used for synchronization operations of a shared memory implementation.

Data dependence analysis on geometric neighbor nodes like in Fig. 2 (right) can be done in the same way as for child/ parent dependencies. However, additional information is needed for the construction of the communication patterns. Given additional user code annotation, for example based on a geometric interpretation of the node relations, a transitive hull is set up, which contains at least

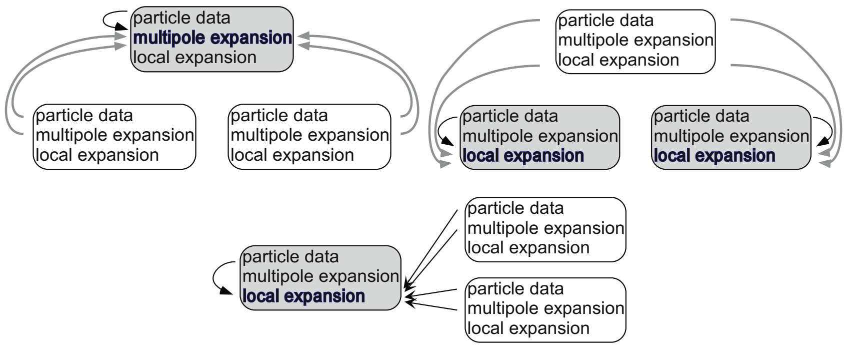


Fig. 2. Sample data dependencies of tree operations: Read/write parent data, read child data, bottom-up traversal (left). Read/write child data, read parent data, top-down traversal (right). Read/write node data, read nodes of interaction list (bottom).

all neighbor nodes involved. We currently use a user defined relation, which may or may not exclude complete sub-trees of interaction nodes. This relation is marked by the code annotation `REQUIRE`, see also the end of the next section.

3 A Fast-Multipole-Method Example Code

As an illustrative example, we provide and discuss some parts of an implementation of a two-dimensional fast-multipole method. The detailed algorithm, formulae and mathematical notation can be found for example in [11]. Here we concentrate on some of the algorithmic structure and features that need to be taken care of in the parallelization. The goal is to evaluate pair interaction forces of a large number of particles. The idea is to approximate long distances computations on coarser parts of a tree of particles, which is done in a bottom-up summation and a top-down evaluation pass.

First of all, we declare a `tree` node derived from a generic k-ary tree defined in the numerical tree library. Multipole and local expansion, particle and field data are added. Further, numerical parts of the algorithm are encapsulated as methods of the `tree` class.

```
class tree : public KAryTree<class tree,4> {
public:
  complex<double> x, field;
  TinyArray1<complex<double>, MAX_EXP_TERMS> mp_exp, local_exp;
  ...
};
```

The tree is created by successive insertion of particles. In order to separate particles from one another, the respective tree node is subdivided. The distributed memory implementation also needs to distribute the tree data structure. A straightforward way to do this is to start with a coarse tree replicated on all processors. The complete sub-tree of the coarse tree leaf is mapped to exactly one processor. Hence, the tree generation can be performed in parallel. In the case additional load-balancing is necessary, for example space-filling curves can be used [2,3].

```
tree *root = new tree;
```

The first part of the fast multipole summation method computes the far field multipole expansions in a bottom-up order. Children node expansions are shifted to the origin of the parent node expansion and summed up. The implementation consists of some methods of the `tree` classes. We show the declaration only. The comments indicate data dependencies for this presentation. Note that the automatic dependence analysis is based on the actual implementation, not on the comments.

```
  void InitMPExp();               // store mp_exp
  void ComputeMPExp();            // store mp_exp
  void ShiftMPExp(tree *cb);      // store mp_exp, load cb->mp_exp
```

The main code instantiates and uses a tree iterator and contains the actual algorithmic atom to be executed for each tree node. This is a generic tree operator and a first example of the proposed domain language.

```
BottomUpIterator<tree> iu(root);

ForEach(tree *b, iu, ‘
  if (b->isleaf())  b->ComputeMPExp();
  else {
    b->InitMPExp();
    for (int i=0; i<tree::dim; i++)
      if (b->child[i])
        b->ShiftMPExp(b->child[i]);
  }  ’)
```

The code transformation system converts this expression into an ordinary tree traversal in the sequential case. The system is able to determine the type of tree traversal and emits error messages in cases where an unsuitable iterator is specified. However, the construction of a parallel iterator implies that the operations on the children of a node are independent and can be executed in parallel. Hence, for the thread parallel version, sub trees are assigned to different threads, once the coarse tree provides enough sub trees to distribute the load evenly. However, there is some data dependence, namely the child to parent dependence in the array `mp_exp`. This translates into message passing at the level of sub tree to coarse tree on distributed memory machines. The presented code transformation system is able to detect this dependence, even as a inter-procedure code analysis, and to emit the correct message passing instructions.

The second stage of the fast multipole summation computes the interaction lists. For a balanced tree, the interactions are a set of siblings of a node. However, in the case of unbalanced trees, additional nodes on finer or coarser levels may be needed for the interactions. Nevertheless, the interaction lists can be computed in a top-down tree traversal.

```
TopDownIterator<tree> it(root);
```

For distributed memory machines, we replicate the operations on the already replicated coarse tree, such that no communication or message passing is actually needed in this step.

The final stage of the algorithm, which can also be executed in conjunction with the second stage, computes the local expansions and finally the fields. Here, the far field multipole expansions are evaluated directly or converted into near field local expansions, which need to be evaluated. This can be performed in a top-down tree traversal with a set of methods in the `tree` class,

```
  void VListInter(tree *src);   // store local_exp, load src->mp_exp
  void UListInter(tree *src);   // store field, load src->x
  void WListInter(tree *src);   // store field, load src->mp_exp
  void XListInter(tree *src);   // store local_exp, load src->x
  void ShiftLocalExp(tree *cb); // store cb->local_exp, load local_exp
```

which perform the four types of possible conversions and shift the local expansions for propagation to the children nodes.

```
ForEach(tree *b, it, `
  for (int i=0; i<tree::dim; i++)
    if (b->child[i]) {
      b->ShiftLocalExp(b->child[i]);
      b->child[i]->field = 0.0;
      if (b->child[i]->isleaf())
        for (list<tree*>::iterator n = b->child[i]->inter.begin();
             n != b->child[i]->inter.end(); n++) {
          if ((*n)->isleaf()) b->child[i]->UListInter(*n);
          else                b->child[i]->WListInter(*n);
        }
      else
        for (list<tree*>::iterator n = b->child[i]->inter.begin();
             n != b->child[i]->inter.end(); n++) {
          if ((*n)->isleaf()) b->child[i]->XListInter(*n);
          else                b->child[i]->VListInter(*n);
        }
      b->child[i]->EvaluateLocalExp();
    }  ')
```

Both sequential and thread parallel versions are relatively easy to generate, since there is only a parent to child data dependence in the `local_exp` arrays. However, in the distributed memory version of this code, the dependence on sibling nodes `mp_exp` arrays and particle data `x` turns out to be a severe problem. While it is easy to detect this as a possible dependence, only a global analysis of the tree and interaction list construction may lead to an efficient and correct result. Some message passing is needed for correctness, but too much may exhaust local memory and degrade parallel efficiency. At this point we clearly see the limitations of automatic parallelization.

The solution we chose here is an additional hint (code annotation) in the application program. The hint provides a criterion to select a set of nodes, which are needed at most. The transformed code initiates a message passing step to exchange the variables determined by the dependence analysis. Data of a node is sent to another processor, if and only if the given criterion may match for any of this processors nodes. Hence, the criterion has to be transitive such that it is fulfilled for a pair of nodes, if it is fulfilled for one of its children and the other node.

```
REQUIRE(list<tree*> neighbor, fetch);
REQUIRE(list<tree*> inter, fetch);
int fetch(tree *b) { return (distance(b) <= 2 * fmin(diam, b->diam)); }
```

Some more details of the code are presented in the following section.

4 Numerical Experiments

For illustration purposes of the concept of generative programming and automatic parallelization of codes written in a domain language we have to implement several systems: First of all, a data dependence analysis tool and a code generation system have to be created. In order to demonstrate its use a domain language and a parallel application library has to be written. Finally a sample application code has to be developed, which is written in the domain language and which is compiled by the code generation system.

The main part of data dependence analysis currently is implemented in a non-robust way leading to a speculative parallelization. The tree atoms of code are compiled and instantiated in different settings. A runtime system keeps track of all references to variables, which results in a dependence analysis capable of interprocedure analysis and recursive calls. However, this requires some programming discipline and the possibility of missing some data dependence. An alternative approach to be pursued in the future would substitute this step by a static code analysis within the optimization phase of a standard C++ compiler. The code generation system further includes multiple passes of the code by the macro preprocessor m4 to generate code, g++ to either compile the code or look for errors and some perl scripts to extract information from the compiler error messages or code instantiations. The results of the compilation process is a correct sequential or parallel code. Each parallel programming model of message passing, thread parallelism or mixed model leads to a different parallel code. The overall execution time of the code generation process is of the same order as standard compilation times of optimizing compilers and are substantially lower than compilation times of some cases of expression templates or self-tuning libraries, see Table 1.

Table 1. Execution times of the source-to-source transformation and compilation times, FMM example, wall clock in sec. Total times are measured and do not exactly match the sum of sub tasks mainly due to pipelining effects of the compilation stages

	sequential	pThreads	MPI	pThreads & MPI
find out of scope variables	9.0			
find local scope variables	1.7			
create instrumented code	–	4.3		
create src code	0.45	0.78	0.75	0.83
compile src, no flags / -O3	2.3/3.2	2.4/3.8	4.8/7.4	5.0/7.4
total, no flags / -O3	11.5/12.6	17.7/20.1	20.2/23.8	21.3/22.9
532 lines of code expand to	590	680	729	777

Now we are ready for the second implementation, namely a numerical tree algorithm to be parallelized and compiled by the code generation system. Again for illustration purposes we chose a well documented example. The fast multipole method in two spatial dimensions can be written for a log$r$ potential conveniently with complex numbers and arithmetic [11] using a Laurent- and a power-series. We chose the fast multipole code of the SPLASH-2 program collection [12] as an initial point. The number of coefficients is fixed both for the far field Laurent series and the near field power series. A quad-tree is constructed with at most one particle per node, each representing a square shaped cell. The particles are distributed uniform over the unit square $[0, 1]^2$ which leads to a slightly unbalanced tree, but good load balance for processor numbers of powers of two.

Wall clock times of a single fast multipole summation, including the computation of far field, near field and the interaction lists are reported on two computer platforms. First, we consider a shared memory computer with four dual-core AMD Opteron processors at 1.8GHz with 64bit Scientific Linux totaling eight processors. In Table 2 we report execution times for two different problem sizes for the sequential, the thread parallel (pThreads), the message-passing (MPI) and the mixed parallel code. Second, the same codes are compiled and run on a beowulf cluster of 32 PCs with a dual-core Intel D820 processor at 2.8GHz running 64bit Rocks Linux connected by gigabit ethernet totaling 64 processors. Three different problems sizes and four programming models are run, see Table 3. The all MPI version places two MPI processes on a computer, while the mixed programming model uses one MPI job, which initiates a second worker thread. On both platforms we use the Mpich MPI implementation with shared memory communication on a computer and p4 device over the network.

On both platforms and for all parallel programming models we observe good parallel speedup and efficiency. Further, four-times the number of particles, leading roughly to four-times the amount of work, also gives good parallel scaling. The most efficient parallel programming model on the shared memory machine is message-passing. The thread implementation is slightly slower. The parallelization is efficient up to eight processors, especially for the larger problem size.

Table 2. Execution times of the FMM example, wall clock in sec. SMP server with 4 dual core processors

no. of proc. cores	1	2	4	8
$0.36 \cdot 10^6$particles, MPI	21.56	10.80	6.47	3.73
2 threads per MPI node		11.60	6.93	3.73
4 threads per MPI node			7.13	4.34
8 threads per MPI node				4.19
$1.44 \cdot 10^6$particles, MPI	84.15	42.11	21.56	11.29
2 threads per MPI node		43.84	24.24	11.68
4 threads per MPI node			25.26	12.73
8 threads per MPI node				13.68

Table 3. Execution times of the FMM example, wall clock in sec. Beowulf cluster with dual core nodes and gigabit ethernet

no. of proc. cores	1	2	4	8	16	32	64
$0.36 \cdot 10^6$particles, MPI	28.42	14.81	7.14	3.45	2.14	2.00	2.73
2 threads per MPI node		15.84	7.64	3.76	2.01	1.30	1.22
$1.44 \cdot 10^6$particles, MPI				15.03	10.17	7.21	5.47
2 threads per MPI node					7.53	3.90	2.70
$5.76 \cdot 10^6$particles, MPI						15.6	8.26
2 threads per MPI node							8.24

For reasons of main memory size, the larger problems can be run on the cluster only for a certain number of computers and above. Up to eight processors, message passing is most efficient. On larger processor numbers the mixed programming model is faster, probably due a limitation of network speed (gigabit ethernet) and a fewer number of communication operations of each computer. Versions up to four processors are executed faster on the AMD processors, but the cluster is faster for eight processors (and more).

Hence all parallel programming models are efficient in general. The optimal choice depends on the platform and number of processors. Execution larger numbers of fast multipole summations, the compilation times (reported from a slower computer) can be neglected.

5 Conclusion

We have discussed some aspects of the automatic parallelization of tree codes. With fast multipole and related algorithms in mind, a programming style with a domain specific tree traversal library and some user code which defines data structures and operations on the tree nodes is briefly introduced. This style allows for a data dependence analysis of the tree algorithms and an efficient parallelization, in the sense of telescoping programming languages. Code annotation was used, when static dependence analysis was no longer sufficient. The library and preprocessor have been used so far among others for the parallelization of a fast multipole method.

We would like to thank the anonymous referees for their helpful comments.

References

1. Singh, J.P., Holt, C., Gupta, A., Hennessy, J.L.: A parallel adaptive fast multipole method. In: Proc. 1993 ACM/IEEE conf. Supercomputing, pp. 54–65. ACM, New York (1993)
2. Salmon, J.K., Warren, M.S., Winckelmans, G.S.: Fast parallel tree codes for gravitational and fluid dynamical N-body problems. Int. J. Supercomp. Appl. 8(2), 129–142 (1994)
3. Caglar, A., Griebel, M., Schweitzer, M.A., Zumbusch, G.: Dynamic load-balancing of hierarchical tree algorithms on a cluster of multiprocessor PCs and on the Cray T3E. In: Meuer, H.W. (ed.) Proc. 14th Supercomp. Conf., Mannheim, Mateo (1999)
4. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, San Francisco (2002)
5. Hummel, J., Hendren, L.J., Nicolau, A.: A framework for data dependence testing in the presence of pointers. In: Proc. 23rd annual int. conf. parallel processing, pp. 216–224 (1994)
6. Oldham, J.D.: POOMA. A C++ Toolkit for High-Performance Parallel Scientific Computing. CodeSourcery (2002)
7. Kuchen, H.: Optimizing sequences of skeleton calls. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 254–273. Springer, Heidelberg (2004)
8. Herrmann, C., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. Parallel Proc. Let. 10(2/3), 239–250 (2000)
9. Lengauer, C.: Program optimization in the domain of high-performance parallelism. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 73–91. Springer, Heidelberg (2004)
10. Ananiev, A.: Algorithm alley: A generic iterator for tree traversal. Dr. Dobb's J. 25(11), 149–154 (2000)
11. Beatson, R., Greengard, L.: A short course on fast multipole methods. In: Ainsworth, M., Levesley, J., Light, W., Marletta, M. (eds.) Wavelets, Multilevel Methods and Elliptic PDEs. Numerical Mathematics and Scientific Computation, pp. 1–37. Oxford University Press, Oxford (1997)
12. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. 22nd annual int. symp. computer architecture, pp. 24–36. ACM, New York (1995)

Efficient Assembly of Sparse Matrices Using Hashing

Mats Aspnäs, Artur Signell, and Jan Westerholm

Åbo Akademi University,
Faculty of Technology, Department of Information Technologies,
Joukahainengatan 3–5, FI-20520 Åbo, Finland
{mats, asignell, jawester}@abo.fi

Abstract. In certain applications the non-zero elements of large sparse matrices are formed by adding several smaller contributions in random order before the final values of the elements are known. For some sparse matrix representations this procedure is laborious. We present an efficient method for assembling large irregular sparse matrices where the non-zero elements have to be assembled by adding together contributions and updating the individual elements in random order. A sparse matrix is stored in a hash table, which allows an efficient method to search for an element. Measurements show that for a sparse matrix with random elements the hash-based representation performs almost 7 times faster than the compressed row format (CRS) used in the PETSc library. Once the sparse matrix has been assembled we transfer the matrix to e.g. CRS for matrix manipulations.

1 Introduction

Sparse matrices play a central role in many types of scientific computations. There exists a large number of different storage formats for sparse matrices [1], such as the compressed row format, compressed column format, block compressed row storage, compressed diagonal storage, jagged diagonal format, transposed jagged diagonal format [2] and skyline storage. The Sparskit [3] library supports 16 different sparse matrix formats. These storage formats are designed to take advantage of the structural properties of sparse matrices, both with respect to the amount of memory needed to store the matrix and the computing time to perform operations on it. The compressed row format, for instance, is a very efficient representation for sparse matrices where rows have identical structure and where elements are accessed in increasing row and column order, e.g. when multiplying a matrix with a vector.

However, in applications where a sparse matrix has to be assembled, by adding and updating individual elements in random order, before the matrix can be used, for example for inversion, these storage methods are inefficient. In this paper, we argue that hashing with long hash tables can be used to efficiently assemble individual matrix elements in sparse matrices, improving time performance of matrix element updating by up to a factor of 50 compared to the compressed row format.

We stress that the proposed sparse matrix format is not intended to be used as a general replacement for the traditional storage formats, and that it is not efficient for performing computations on sparse matrices, like computing matrix–vector products or solving linear systems. These algorithms access the matrix elements in a highly regular fashion, normally row-wise or column-wise, and do not exhibit any of the irregular behavior for which the hash-based representation is designed to be efficient. The hash-based representation is efficient for applications where the elements have to be accessed repeatedly and in random order. Thus, once the matrix has been assembled in the hash-based representation it is transferred to, e.g., CRS for matrix manipulations or matrix-vector calculations.

2 Background

The problem of assembling large irregular sparse matrices originated in a project where the goal was to improve the performance of a parallel fusion plasma simulation, Elmfire [4]. In fusion plasma simulations particles move along orbits circling a torus simultaneously making small Larmor circles along their main trajectory. Instead of explicitly calculating the long range particle–particle interactions, a short time average electrostatic potential is calculated on a grid spanning the torus, and the particle-particle interaction is replaced by a local interaction with the electrostatic potential. The grid points receive contributions from every particle within a given radius from the grid point, typically several grid units. The problem of calculating the electrostatic potential turns into the problem of collecting and inverting a linear sparse matrix, where the individual elements are assembled from the contributions of up to 100 particles at irregular moments in time as the particles are simulated forward in time. In a typical application the electrostatic potential was represented by a 28000×28000 sparse matrix of double-precision floating-point values with a 10% fill rate.

The plasma simulation program used a collection of sparse matrix routines from the PETSc library [5] to first assemble and then invert the distributed sparse matrix representing the electrostatic potential. Computing the inverse of the sparse matrix turned out to be very efficient, taking only a few seconds. However, the assembly of the matrix was very time consuming (of the order 100 seconds), and the decision was made to use another sparse matrix representation to first assemble the matrix and then construct the compressed row format used in the PETSc library to invert it.

Effectively, we are trading memory for speed, since the sparse matrix is assembled in one data structure (a hash table) and then converted to another data structure (the CRS format) before it is used in the computation. However, the improved efficiency of the program clearly outweighs the drawback of the increased memory consumption.

3 Sparse Matrix Representation Using Hash Tables

Our sparse matrix representation is essentially a coordinate storage format where the non-zero elements are stored in a hash table [6] to allow for an efficient method to update individual elements in random order. An element (x, y) in a two-dimensional sparse matrix with r rows and c columns, $x \in [0, r-1]$, $y \in [0, c-1]$, with value v will be stored in a hash table with the key $k = x*c+y$. The hash function $h(k)$ is simply the value $k \ mod \ s$. The size of the hash table, s, is chosen to be n_z/b, where n_z is the number of non-zeros in the sparse matrix and b is a small integer value, for instance 20. The number of non-zero elements in the sparse matrix is given as a parameter by the user. The idea is that, assuming that the non-zero elements are evenly distributed over the hash keys, on an average b elements are mapped to any hash key and we try to keep b reasonably small in order to be able to quickly locate an entry with this hash key value. The size of the hash table is actually chosen to be a prime number s' larger than or equal to n_z/b. This choice will typically map the non-zero elements evenly among the different hash values.

Encoding the (x, y) coordinates of non-zero matrix elements as a single integer value $k = x * c + y$ saves memory space. Using 32-bit integers, a non-zero value requires only 12 bytes of memory instead of 16 bytes if the (x, y)-coordinates and the value are stored as two integers and one double-precision floating point value. Using 64-bit integers we need 16 bytes for each non-zero element instead of 24 bytes. A minor drawback of this approach is that the row and column indices have to be computed with integer division and modulo-operations. The representation also puts an upper limit on the size of the sparse matrices that can be represented. The largest unsigned integer that can be represented with 32 bits (the constant `UINT_MAX` in C) is $2^{32} - 1$. Thus, the largest square matrix that can be stored has 65535 rows and columns. The largest unsigned 64-bit integer value (`ULONG_MAX` in C) is $2^{64} - 1$, which allows for square matrices with approximately 4.2 x 10^9 rows and columns.

Matrix elements which map to the same position in the hash table are stored in an array of tuples consisting of a key k and a value v. The number of elements in an array is stored in the first position (with index zero) of the array, together with the size of the array. All arrays are initialized to size n_z/s'. In other words, the arrays are initialized to the average number of non-zero elements per hash key. If the non-zero values are evenly distributed over all hash keys, no memory reallocation should be necessary.

Three operations can be performed on the sparse matrix: an **insert**-operation which assumes that the element does not already exist in the matrix, a **get**-operation which retrieves a value and an **add**-operation which adds a value to an already existing value in the matrix.

To insert a value v into position (x, y) in the matrix, we first compute its key k and the hash key $h(k)$ as described earlier. We check if there is space for the new element in the array given by the hash key $h(k)$ by comparing the number of elements stored in the array with the size of the array. If the array is full, it is enlarged by calling the `realloc` function and updating the size of the array. The value is inserted as a tuple (k, v) into the next free position in the array, and the number of elements in the array is incremented by one. Since memory reallocation is a time-consuming procedure, arrays are always enlarged by a factor that depends on the current size of the array (for instance by 10% of its current size). The minimum increase, however, is never less than a pre-defined constant, for instance 10 elements. This avoids small and frequent reallocations at the expense of some unused memory.

The get-operation searches for an element with a given (x, y)-position in the sparse matrix. We first compute the key k and the hash key $h(k)$ and perform a linear search in the array $h(k)$, looking for an element with the given key. If the key is found we return its value, otherwise the value 0.0 is returned. The linear search that is needed to locate an element will be short and efficient, since it goes through a contiguous array of elements with an average length of b stored in consecutive memory locations. This means that the search is efficient, even though it is an $O(N)$ operation. The key to the success of using long hash tables is thus manifested by the short length of the array needed for each hash value $h(k)$. An alternative approach would be to use a tree-based structure to resolve collisions among hash keys, for instance an AVL-tree [6]. This would make the search an $O(logN)$ operation, but the insert-operation would become much more complex. Another drawback would be the increase in memory to store the elements, since each element would then need two pointers for its left and right subtrees, in addition to the key and the value.

The add-operation adds a given value to the value stored in the sparse matrix at position (x, y). It first performs a get-operation, as described above, to locate the element. If the element is found, i.e. (x, y) already has a non-zero value stored in the sparse matrix representation, the given value is added to this. If the element is not found, it is inserted using the insert-operation.

3.1 Look-Up Table

In some cases there is a temporal locality when assembling the elements of a sparse matrix, which implies that we will be accessing the same element repeatedly at small time intervals. To speed up accesses to such matrix elements and avoid linear searches, we store the positions of the elements that have been accessed recently in a look-up table. The look-up table is also implemented as a hash table, but with a different hash function $l(k)$ than that used for storing the matrix elements. The look-up table contains the key of an element and the index of its position in the array. To search for an element (x, y) we compute the key k and the look-up table hash function $l(k)$ and compare the value at position $l(k)$ in the look-up table against the key k. If these are equal, the element can be found in the array associated with hash table $h(k)$ at the position given in the index in the look-up table. More than one element can be mapped to the same position in the look-up table. No collision handling is used in the look-up table, but only the last reference to an element will be stored. The size of the look-up table is given as a parameter by the user when the matrix is created. The size is a fraction $1/a$ of the hash table size s', and is also chosen to be a prime number.

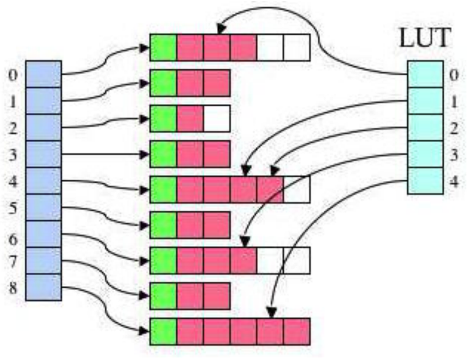


Fig. 1. Hash table structure

A typical size for the look-up table could be 1/10 of the hash table size. The data structure used to represent the sparse matrix is illustrated in Figure 1.

4 Performance

We have compared the times to assemble sparse matrices of size 10000 by 10000 with a fill degree of 10% using both the compressed row format of PETSc and our hash-based representation. The test matrices are chosen to exhibit both efficient and inefficient behavior for the two storage formats. In the test programs, 10 million non-zero elements are inserted into the sparse matrix storage either with an insert-operation or an add-operation, depending on whether the elements are all unique or not. The time to perform the insert- or add-operations are measured with the `time`-function in the C language.

The structures of the test matrices (for illustration purposes with 100 by 100 matrices) are shown in figure 2.

The first test inserts non-zero values into every 10*th* column in the matrix in order of increasing (x, y)-coordinates. Hence the matrix is perfectly regular

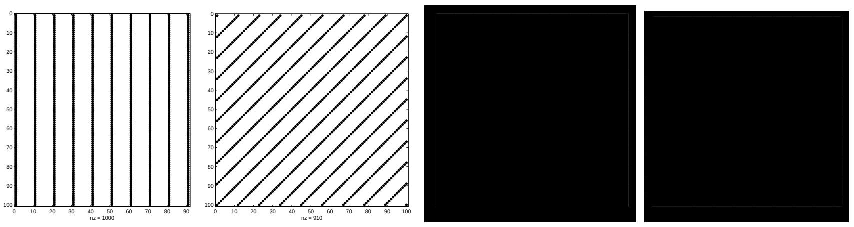


Fig. 2. Structure of test matrices: every 10*th* element, every 11*th* element, random and clustered

with identical row structures. The matrix elements are all unique and we need only perform an insert-operation. As the measurements show, the CRS format used in PETSc is very efficient for this case, clearly outperforming the hash-based representation. This is to be expected since in the CRS format all accesses go to consecutively located addresses in memory (stride one), thus making excellent use of cache memory.

The second test inserts the same elements as in the first case, i.e. a non-zero value in every $10th$ position, but in reverse order of (x, y)-coordinates, starting from the largest row and column index. The matrix elements are all unique and we need only perform an insert-operation. The measurements show that this case performs considerably slower using the CRS format in PETSc, while the hash-based representation is only slightly slower than the previous case.

The third test inserts non-zero elements in every $11th$ position in the matrix. Thus, consecutive rows will have different structure since the row length is not divisible by 11. The matrix elements are all unique and we need only perform an insert-operation. Surprisingly, the CRS format in PETSc is extremely inefficient for this case, while the performance of the hash-based representation is similar to the two previous cases.

The fourth test inserts 10 million randomly generated non-zero elements, which are inserted into the sparse matrix storage using an add-operation, since the elements are not necessarily all unique. We can see that the CRS format in PETSc is very inefficient for this case, while the hash-based representation shows a small performance penalty.

In the fifth test we first generate 100000 randomly located clusters centers, and then for each cluster we generate 100 random elements located within a square area of 9 by 9 around the cluster center. As in the fourth test, we must use an add-operation since the matrix elements are not all unique.

An example of the type of sparse matrices encountered in the fusion plasma simulation is illustrated in figure 3.

4.1 Execution Time

The results of the measurements of the execution times are summarized in table 1. The tests were run on a AMD Athlon XP 1800+ processor with 1 GB of memory. The operating system was Red Hat Enterprise Linux 3.4 with gcc 3.4.4. The test programs were compiled with the -O3 compiler optimization switch.

The measurements show that the compressed row format used in PETSc is very inefficient if the sparse matrix lacks structure, while the hash-based representation is insensitive to the structure of the matrix. In our application, the fusion simulation, the matrix structure was close to the fifth test case, consisting of randomly distributed clusters, each consisting of about 1 to 100 values. In our application a significant speed-up was indeed achieved using the sparse matrix representation with long hash tables. In addition, the logic for assembling the distributed matrix with contributions from all parallel processes was simplified, thanks to the restructuring of the code that had to be done at the same time.

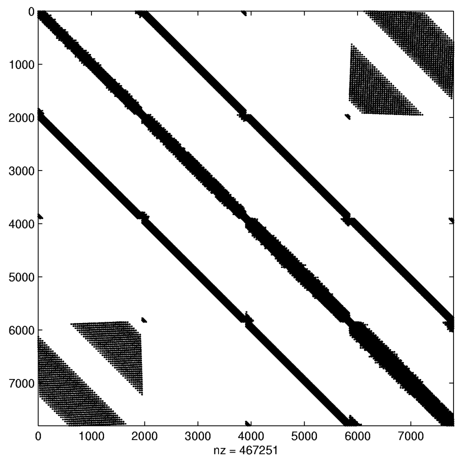


Fig. 3. Example of a sparse matrix in the fusion plasma simulation

Table 1. Measured execution times in seconds for the five testcases

Matrix structure	**PETSc**	**Hash**
Every 10*th* element, natural order	2.4	6.9
Every 10*th* element, reverse order	20.6	7.5
Every 11*th* element	377.8	7.1
Random	86.6	12.5
Clusters	37.8	12.0

4.2 Memory Usage

The sparse matrix representation based on hash tables uses slightly more memory than the compressed row format. The compressed row format uses $12n_z+4(r+1)$ bytes of memory to store a matrix with r rows and n_z non-zero entries. The hash-based representation uses $12n_z + 16n_z/b + 8n_z/ab$ bytes, where b is the average number of non-zero elements per hash key and a gives the size of the look-up table as a fraction of the hash table size, as explained in section 3.1

For the sparse matrices used in the test cases presented in section 4, all with 10000 rows and columns filled to 10%, that is, with 10 million non-zero elements, the compressed row format needs about 114 MB of memory, whereas the hash-based representation needs about 122 MB.

5 Conclusions

We have presented an efficient solution to the problem of assembling unstructured sparse matrices, where most of the non-zero elements have to be computed by adding together a number of contributions. The hash-based representation has been shown to be efficient for the particular operation of assembling and updating elements of sparse matrices, for which the compressed row storage format is inefficient. The reason for this inefficiency is that access to individual elements is slow, particularly if the accesses do not exhibit any spatial locality, but are randomly distributed. For these types of applications, a representation using a hash table provides a good solution that is insensitive to the spatial locality of the elements.

The hash-based representation is not intended to be used as a general purpose storage format for sparse matrices, for instance to be used in matrix computations. It is useful in applications where unstructured sparse matrices have to be assembled, as described in this paper, before they can be used for computations. Our experiences show that the performance of programs of this kind can be significantly improved by using two separate sparse matrix representations; one during matrix element assembly and another for matrix operations, e.g. inversion. The trade-off in memory consumption is outweighed by the increased performance when assembling the matrix elements using long hash tables.

References

1. Dongarra, J.: Sparse matrix storage formats. In: Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H. (eds.) Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM (2000), `http://www.cs.utk.edu/~dongarra/etemplates/node372.html`
2. Montagne, E., Ekambaram, A.: An optimal storage format for sparse matrixes. Information Processing Letters 90, 87–92 (2004)
3. Saad, Y.: SPARSKIT: a basic tool kit for sparse matrix computetions, version 2, University of Minnesota, Department of Computer Science and Engineering, `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`
4. Heikkinen, J., Henriksson, S., Janhunen, S., Kiviniemi, T., Ogando, F.: Gyrokinetic simulation of particle and heat transport in the presence of wide orbits and strong profile variations in the edge plasma. Contributions to Plasma Physics 46(7-9), 490–495 (2006)
5. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2004)
6. Goodrich, M.T., Tamassia, R.: Algorithm Design. Foundations, Analysis, and Internet Examples. John Wiley & Sons, Chichester (2001)

A Web-Site-Based Partitioning Technique for Reducing Preprocessing Overhead of Parallel PageRank Computation*

Ali Cevahir, Cevdet Aykanat, Ata Turk, and B. Barla Cambazoglu

Bilkent University, Department of Computer Engineering,
TR-06800 Bilkent, Ankara, Turkey
{acevahir, aykanat, atat, berkant}@cs.bilkent.edu.tr

Abstract. A power method formulation, which efficiently handles the problem of dangling pages, is investigated for parallelization of PageRank computation. Hypergraph-partitioning-based sparse matrix partitioning methods can be successfully used for efficient parallelization. However, the preprocessing overhead due to hypergraph partitioning, which must be repeated often due to the evolving nature of the Web, is quite significant compared to the duration of the PageRank computation. To alleviate this problem, we utilize the information that sites form a natural clustering on pages to propose a site-based hypergraph-partitioning technique, which does not degrade the quality of the parallelization. We also propose an efficient parallelization scheme for matrix-vector multiplies in order to avoid possible communication due to the pages without in-links. Experimental results on realistic datasets validate the effectiveness of the proposed models.

1 Introduction

PageRank is a popular algorithm used for ranking Web pages by utilizing the hyperlink structure among the pages. PageRank algorithm usually employs the random surfer model [21], which can be described as a Markov chain, where the PageRank values of pages can be computed by finding the stationary distribution of this chain. Traditionally, PageRank computation is formulated as finding the principal eigenvector of the Markov chain transition matrix and solved using the iterative power method. Recently, linear system formulations and associated iterative solution methods [3, 9, 18] are investigated for PageRank computation as well. In both types of formulations, PageRank computation can be accelerated via parallelization [9, 20] or increasing the convergence rate of the iterative methods [5, 12, 15, 16, 19].

The focus of this work is on reducing the per iteration time through parallelization. Among several formulations [13, 15, 17, 18] proposed for handling the dangling-page (pages without out-links) problem, widely used formulation

* This work is partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under project EEEAG-106E069.

of Kamvar et al. [15] is selected for parallelization. In this formulation, which is based on the power method, the kernel operations are sparse-matrix vector multiply and linear vector operations. The partitioning scheme adopted in our parallelization is based on rowwise partitioning of the transition matrix and conformable partitioning of the linear vectors used in the iterative power method.

Recently, the hypergraph-partitioning-based sparse matrix partitioning method of Catalyurek and Aykanat [6, 7] is applied by Bradley et al. [4] for efficient parallelization of the above-mentioned power method formulation. This parallelization technique successfully reduces the communication overhead while maintaining computational load balance. However, the preprocessing overhead due to hypergraph partitioning, which must be repeated often due to constantly evolving nature of the Web, is quite significant compared to the duration of the PageRank computation.

In this work, we investigate techniques for reducing the overhead of the preprocessing step before the PageRank computation without degrading the quality of the parallelization. To this end, we propose a site-based compression on the rows of the transition matrix relying on the expectation that sites form a natural clustering on pages. Then, the conventional hypergraph model [6, 7] is applied on the compressed site-to-page transition matrix to induce a partitioning on the original page-to-page transition matrix. We also propose an efficient parallelization scheme for matrix-vector multiplies in order to avoid possible communication due to the pages without in-links. Furthermore, we extend the hypergraph-partitioning model to encapsulate both this efficient parallelization scheme and the computational load balance over the whole iterative algorithm. Experimental results on realistic Web datasets verify the validity of the proposed models. The proposed site-based partitioning scheme reduces the preprocessing time drastically compared to the page-based scheme while producing better partitions in terms of communication volume. Our implementation for the proposed parallel PageRank algorithm shows that site-based partitioning scheme leads to better speedup values compared to the page-based scheme on a 32- node PC cluster.

The rest of the paper is organized as follows. Section 2 summarizes the PageRank algorithm. The proposed parallelization scheme is discussed in Section 3. Section 4 describes the proposed page-based and site-based partitioning schemes. Experimental results are presented in Section 5. Finally, concluding remarks are given in Section 6.

2 PageRank Algorithm

PageRank can be explained with a probabilistic model, called the random surfer model. Consider a Web user randomly visiting pages by following out-links within pages. Let the surfer visit page i at a particular time step. In the next time step, the surfer chooses to visit one of the pages pointed by the out-links of page i at random. If page i is a dangling page, then the surfer jumps to a random page. Even if page i is not a dangling page, the surfer may prefer to jump to a random page with a fix probability instead of following one of the out-links of page i.

In the random surfer model, the PageRank of page i can be considered as the (steady-state) probability that the surfer is at page i at some particular time step. In the Markov chain induced by the random walk on the Web containing n pages, states correspond to the pages in the Web and the $n \times n$ transition matrix $\mathbf{P} = (p_{ij})$ is defined as $p_{ij} = 1/deg(i)$, if page i contains out-link(s) to page j, and 0, otherwise. Here, $deg(i)$ denotes the number of out-links within page i.

A row-stochastic transition matrix $\mathbf{P}'$ is constructed from $\mathbf{P}$ as $\mathbf{P}' = \mathbf{P} + \mathbf{d}\mathbf{v}^T$ via handling of dangling pages according to the random surfer model. Here, $\mathbf{d} = (d_i)$ and $\mathbf{v} = (v_i)$ are column vectors of size n. $\mathbf{d}$ identifies dangling pages, i.e., $d_i = 1$ if row i of $\mathbf{P}$ corresponds to a dangling page, and 0, otherwise. $\mathbf{v}$ is the teleportation (personalization) vector which denotes the probability distribution of destination pages for a random jump. Uniform teleportation vector $\mathbf{v}$, where $v_i = 1/n$ for all i, is used for generic PageRank computation [14]. Non-uniform teleportation vectors can be used for achieving topical or personalized PageRank computation [11, 21], or preventing link spamming [10].

Although $\mathbf{P}'$ is row-stochastic, it may not be irreducible. For example, the Web contains many pages without in-links, which disturb irreducibility. An irreducible Markov matrix $\mathbf{P}''$ is constructed as $\mathbf{P}'' = \alpha\mathbf{P}' + (1-\alpha)\mathbf{e}\mathbf{v}^T$, where $\mathbf{e}$ is a column vector of size n containing all ones. Here, α represents the probability that the surfer chooses to follow one of the out-links of the current page, and $(1-\alpha)$ represents the probability that surfer makes a random jump instead of following the out-links.

Given $\mathbf{P}''$, PageRank vector $\mathbf{r}$ can be determined by computing the stationary distribution for the Markov chain, which satisfies the equation $(\mathbf{P}'')^T\mathbf{r} = \mathbf{r}$. This corresponds to finding the principal eigenvector of matrix $\mathbf{P}''$. Applying the power method directly for the solution of this eigenvector problem leads to a sequence of matrix-vector multiplies $\mathbf{p}^{k+1} = (\mathbf{P}'')^T\mathbf{p}^k$, where $\mathbf{p}^k$ is the kth iterate towards the PageRank vector $\mathbf{r}$. However, matrix $\mathbf{P}''$ is completely dense, whereas original $\mathbf{P}$ is sparse. Kamvar et al. [15] propose an efficient multiplication scheme by reformulating the multiplication with dense matrix $(\mathbf{P}'')^T$ in terms of sparse $\mathbf{P}^T$. This efficient PageRank algorithm is given in Fig. 1.

3 Parallel PageRank Algorithm

Two basic types of operations are performed repeatedly at each iteration of the PageRank algorithm given in Fig. 1. The first type is sparse-matrix vector multiply (i.e., $\mathbf{q} \leftarrow \alpha\mathbf{A}\mathbf{p}$), and the second type is linear vector operations, such as L_1 norm (e.g., $\|\mathbf{q}\|_1$), DAXPY (i.e., $\mathbf{q} \leftarrow \mathbf{q} + \gamma\mathbf{v}$) and vector subtraction (i.e., $\mathbf{q} - \mathbf{p}$). We consider the parallelization of the computations of the PageRank algorithm through rowwise partitioning of the $\mathbf{A}$ matrix as $\mathbf{A} = [\mathbf{A}_1^T \cdots \mathbf{A}_k^T \cdots \mathbf{A}_K^T]^T$, where processor P_k stores row stripe $\mathbf{A}_k$. All vectors (e.g., $\mathbf{p}$ and $\mathbf{q}$) used in the algorithm are partitioned conformably with the row partition of $\mathbf{A}$ to avoid communication of the vector components during linear vector operations. That is, the $\mathbf{p}$ and $\mathbf{q}$ vectors are partitioned as $[\mathbf{p}_1^T \cdots \mathbf{p}_K^T]^T$ and $[\mathbf{q}_1^T \cdots \mathbf{q}_K^T]^T$, respectively. Processor P_k is responsible for performing the local matrix-vector

```
PageRank($\mathbf{A}$, $\mathbf{v}$)
1. $\mathbf{p} \leftarrow \mathbf{v}$
2. repeat
3.     $\mathbf{q} \leftarrow \alpha \mathbf{A}\mathbf{p}$
4.     $\gamma \leftarrow ||\mathbf{p}||_1 - ||\mathbf{q}||_1$
5.     $\mathbf{q} \leftarrow \mathbf{q} + \gamma \mathbf{v}$
6.     $\delta \leftarrow ||\mathbf{q} - \mathbf{p}||_1$
7.     $\mathbf{p} \leftarrow \mathbf{q}$
8. until $\delta < \varepsilon$
9. return $\mathbf{p}$
```

Fig. 1. Efficient PageRank algorithm based on the power method: $\mathbf{A} = \mathbf{P}^T$ is the transition matrix, $\mathbf{v}$ is the teleportation vector, and ε is the convergence threshold

multiply $\mathbf{q}_k \leftarrow \alpha \mathbf{A}_k \mathbf{p}$ while holding $\mathbf{p}_k$. Processor P_k is also responsible for the linear vector operations on the kth blocks of the vectors.

In this scheme, the linear vector operations can be efficiently performed in parallel such that only the norm operations require global communication overhead. Fortunately, the volume of communication incurred due to this global communication does not increase with increasing n, and it is only $K-1$ words. On the other hand, depending on the way in which rows of $\mathbf{A}$ are partitioned among the processors, entries in $\mathbf{p}$ may need to be communicated among the processors before the local matrix-vector multiplies, hence this scheme can be considered as a pre-communication scheme. During the pre-communication phase, a processor P_k may be sending the same $\mathbf{p}_k$-vector entry to different processors according to the sparsity pattern of the respective column of $\mathbf{A}$. This multicast like operation is referred to here as *Expand* operation. Note that the communication requirement during the pre-communication may be as high as $(K-1)n$ words and $K(K-1)$ messages, and the communication occurs when each sub-matrix $\mathbf{A}_k$ has at least one nonzero in each column.

As seen in Fig. 1, PageRank algorithm requires two global communication operations in the form of all-to-all reduction due to the norm operations at steps 4 and 6 in Fig. 1. The global operations may incur high communication overhead in parallel architectures with high message latency. In this work, we propose a coarse-grain parallel PageRank algorithm, which reduces the number of global communication operations at each iteration from two to one by rearranging the computations as shown in Fig. 2. Here, two global norms are accumulated at all processors in a single all-to-all reduction operation performed at step 5(c) in Fig. 2. Hence, the proposed coarse-grain formulation halves the latency overhead while keeping the communication volume the same. The only drawback of this formulation is that it will perform an extra iteration compared to the power method formulation given in Fig. 1, because the convergence check is applied on the $\mathbf{p}$ vectors of the previous two iterations. In Fig. 2, a superscript k denotes the partial result computed by processor P_k, e.g., γ^k is the partial result for global scalar γ, where $\gamma = \sum_{k=1}^{K} \gamma^k$.

```
Parallel-PageRank($\mathbf{A}_k$, $\mathbf{v}_k$)
1.          $\mathbf{p}_k \leftarrow \mathbf{v}_k$
2.          $\mathbf{t}_k \leftarrow \mathbf{0}$
3.          repeat
4.   (a)        $\mathbf{p} \leftarrow \mathrm{Expand}(\mathbf{p}_k)$
     (b)        $\mathbf{q}_k \leftarrow \alpha\mathbf{A}_k\mathbf{p}$
5.   (a)        $\gamma^k \leftarrow \|\mathbf{p}_k\|_1 - \|\mathbf{q}_k\|_1$
     (b)        $\delta^k \leftarrow \|\mathbf{p}_k - \mathbf{t}_k\|_1$
     (c)        $\langle\gamma, \delta\rangle \leftarrow \mathrm{AllReduceSum}(\langle\gamma^k, \delta^k\rangle)$
6.              $\mathbf{t}_k \leftarrow \mathbf{p}_k$
7.              $\mathbf{p}_k \leftarrow \mathbf{q}_k + \gamma\mathbf{v}_k$
8.          until $\delta < \varepsilon$
9.          return $\mathbf{p}_k$
```

Fig. 2. Coarse-grain parallel PageRank algorithm (pseudocode for processor P_k)

Web data may contain many pages without in-links [3]. This property can be utilized to increase the efficiency of the parallel PageRank algorithm as follows. Since pages without in-links correspond to zero rows of matrix **A**, the matrix-vector multiply at step 4 of Fig. 2 results in zero values for the respective **q**-vector entries. Hence, for each page i without in-links, p_i iterate can be simply updated as γv_i instead of the DAXPY operation at step 7. Note that v_i is a constant throughout the iterations and γ is a global scalar computed and stored at all processors at each iteration of the algorithm. Hence, possible expand communications of p_i due to the sparsity pattern of column i of **A** can be totally avoided as follows: We replicate v_i among the processors that have at least one row with a non-zero at column i at the very beginning of the algorithm and then enforce each one of those processors to redundantly compute $p_i = \gamma v_i$ at each iteration of the algorithm.

4 Rowwise Partitioning

The objective in the proposed parallelization is to find a rowwise partition of **A** that minimizes the volume of communication during each sparse matrix-vector multiply while maintaining the computational load balance during each iteration.

4.1 Page-Based Partitioning

Rowwise partitioning of irregularly sparse matrices for the parallelization of matrix-vector multiplies is formulated using the hypergraph-partitioning model [6, 7]. In the column-net model proposed for rowwise partitioning [6, 7], a given matrix is represented as a hypergraph which contains a vertex for each row and a net for each column. The net corresponding to a column connects the vertices corresponding to the rows that have a non-zero at that column. The vertices connected by a net are said to be its pins. Vertices are associated with weights which are set equal to the number of non-zeros in the respective rows.

A K-way vertex partition on the hypergraph is decoded as assigning the rows corresponding to the vertices in each part of the partition to a distinct processor. Partitioning constraint on balancing the part weights corresponds to balancing the computational loads of processors, whereas partitioning objective of minimizing the cutsize corresponds to minimizing the total communication volume during a parallel matrix-vector multiply.

In this work, we adopt the hypergraph-partitioning model and extend it to encapsulate both the computational load balance over the whole iterative algorithm and the efficient parallelization scheme described in Section 3. For this purpose, we reorder the rows and columns of matrix $\mathbf{A}$ in such a way that rows and columns corresponding to the pages without in-links are permuted to the end. Then, we decompose the reordered transition matrix $\mathbf{A}$ as follows:

$$\mathbf{A} = \begin{array}{|c|c|}\hline \mathbf{C} & \mathbf{W} \\ \hline \multicolumn{2}{|c|}{\mathbf{Z}} \\ \hline \end{array}$$

Here, rows of sub-matrix $\mathbf{Z}$ and columns of sub-matrix $\mathbf{W}$ correspond to the pages without in-links. Note that $\mathbf{Z}$ is a sub-matrix containing all zeros. We compute a rowwise partition of $\mathbf{A}$ in two phases. The first and second phases respectively incorporate the partitioning of the PageRank computation for the pages with and without in-links among the processors.

In the first phase, we obtain a K-way row partition of sub-matrix $[\mathbf{C} \mid \mathbf{W}]$ by partitioning the column-net representation $\mathcal{H}(\mathbf{C})$ of the $\mathbf{C}$ sub-matrix. $\mathcal{H}(\mathbf{C})$ contains one vertex and one net for each row and non-zero column of sub-matrix $\mathbf{C}$, respectively. Note that no nets are introduced for zero-columns which correspond to dangling pages. Vertices of $\mathcal{H}(\mathbf{C})$ are weighted to incorporate the floating point operations (flops) associated with the non-zeros of both $\mathbf{C}$ and $\mathbf{W}$ sub-matrices as well as the flops associated with the linear-vector operations. That is the weight of vertex i is set equal to: $2 \times nnz(row\ i\ of\ [\mathbf{C} \mid \mathbf{W}]) + 7$. The first term accounts for the number of flops associated with row i during a matrix-vector multiply operation since each matrix non-zero incurs one multiply and one add operation. The second term accounts for the number of flops associated with the linear vector operations performed on the ith entries of the vectors.

In the second phase, rows of sub-matrix $\mathbf{Z}$ are distributed among the parts of the rowwise partition obtained at the end of the first phase. Although it may seem awkward to mention about distributing zero-rows across processors, recall that partitioning of rows also incur the assignment of the respective vector entries and the associated linear vector operations. The number of linear vector operations performed on the vector entries corresponding to the zero rows reduces from 7 to 4 flops since the respective $\mathbf{q}$-vector entries remain as zero. The only metric considered during this distribution is to improve the balance of the partition obtained in the first phase.

4.2 Site-Based Partitioning

Experimental results show that the page-based partitioning technique proposed in Section 4.1 is quite successful in minimizing the total volume of communication during the parallel PageRank computation. However, the preprocessing overhead incurred in the first phase of the page-based scheme is quite significant due to the partitioning of the hypergraph representing the page-to-page $\mathbf{C}$ sub-matrix, which is very large in practice. For example, the time elapsed for 16-way partitioning of the `Google` and `In-2004` datasets are as high as the time elapsed for 64 and 47 iterations of the PageRank computations performed for the respective datasets.

In this section, we propose a technique to reduce this overhead by partitioning a compressed version, $\bar{\mathbf{C}}$, of the page-to-page sub-matrix $\mathbf{C}$. We generate the site-to-page $\bar{\mathbf{C}}$ matrix exploiting the fact that Web sites form a natural clustering of pages. In matrix $\bar{\mathbf{C}}$, rows correspond to Web sites while columns corresponds to pages. The union of the non-zeros of the $\mathbf{C}$-rows that correspond to the pages residing in a site form the non-zeros of the $\bar{\mathbf{C}}$-row corresponding to that site. Then, we apply the column-net model on $\bar{\mathbf{C}}$ to obtain the $\mathcal{H}(\bar{\mathbf{C}})$, and partition $\mathcal{H}(\bar{\mathbf{C}})$ instead of $\mathcal{H}(\mathbf{C})$. The weight of a vertex j in $\mathcal{H}(\bar{\mathbf{C}})$ corresponding to site j is set equal to

$$\sum_{\mathrm{page\ i} \in \mathrm{site\,j}} (2 \times nnz(\mathrm{row\ i\ of\ } [\mathbf{C} \mid \mathbf{W}])) + 7 \times p_{\mathrm{in-links}}(\mathrm{site\,j}),$$

where $p_{\mathrm{in-links}}(\mathrm{site\,j})$ denotes the number of pages with at least one in-link in site j.

Although the compression of $\mathbf{C}$ reduces the number of vertices in $\mathcal{H}(\bar{\mathbf{C}})$ significantly, the compression does not reduce the number of nets. However, experimental results show that $\mathcal{H}(\bar{\mathbf{C}})$ contains many nets that connect a single vertex. These single-pin nets correspond to the pages that have out-links only to the sites they belong to. Since single-pin nets have no potential to incur cost to the cutsize, they can be discarded from $\mathcal{H}(\bar{\mathbf{C}})$ before the partitioning. Removal of single-pin nets significantly reduces the number of total nets in $\mathcal{H}(\bar{\mathbf{C}})$. The partitioning time of $\mathcal{H}(\bar{\mathbf{C}})$ is expected to be much less than that of $\mathcal{H}(\mathbf{C})$, since $\mathcal{H}(\bar{\mathbf{C}})$ contains significantly fewer vertices and nets.

5 Experimental Results

In our experiments, two datasets with different sizes are used. The `Google`[1] dataset is provided by Google and includes .edu domain pages in the US. The `In-2004`[2] dataset is crawled by UbiCrawler and includes pages from the Web of India. The properties of these datasets are given in Table 1. For the convergence threshold of $\epsilon = 10^{-8}$, the PageRank computations converge in 91 and 90

[1] `http://www.google.com/programming-contest/`

[2] `http://law.dsi.unimi.it/`

Table 1. Properties of datasets and column-net hypergraph representations of the respective **C** and $\bar{\mathbf{C}}$ matrices

		`Google`	`In-2004`
# of pages		913,569	1,347,446
# of pages w/o in-links		132,167	86
# of sites		15,819	4,376
# of links		4,480,218	13,416,945
% intra-site links		87.42	95.92
% inter-site links		12.58	4.08
Page-based hypergraph	**# of vertices**	781,402	1,347,446
	# of nets	608,769	1,065,161
	# of pins	4,741,970	14,482,106
Site-based hypergraph	**# of vertices**	15,819	4,376
	# of nets	214,659	205,106
	# of pins	600,952	555,195

iterations for the `Google` and `In-2004` datasets, respectively. In the PageRank computations, the damping factor α is set to 0.85, conforming with the usual practice [3].

Table 1 also shows the properties of the column-net hypergraphs $\mathcal{H}(\mathbf{C})$ and $\mathcal{H}(\bar{\mathbf{C}})$, which represent the **C** and $\bar{\mathbf{C}}$ matrices, respectively. As seen in Table 1, the proposed compression scheme leads to a significant decrease in the size of the hypergraphs. For example, in the `In-2004` dataset, approximately 99%, 80% and 96% reductions are obtained in the number of vertices, nets and pins, respectively. Direct K-way hypergraph partitioning tool kPaToH [2, 8] is used, with default parameters and an imbalance tolerance of 3%, for partitioning the hypergraphs. As kPaToH involves randomized heuristics, kPaToH is run ten times with different seed values for each partitioning instance and the averages of those results are reported in the following figures. The partitioning operations are performed on an Intel Pentium IV 3.0 GHz processor with 2 GB of RAM.

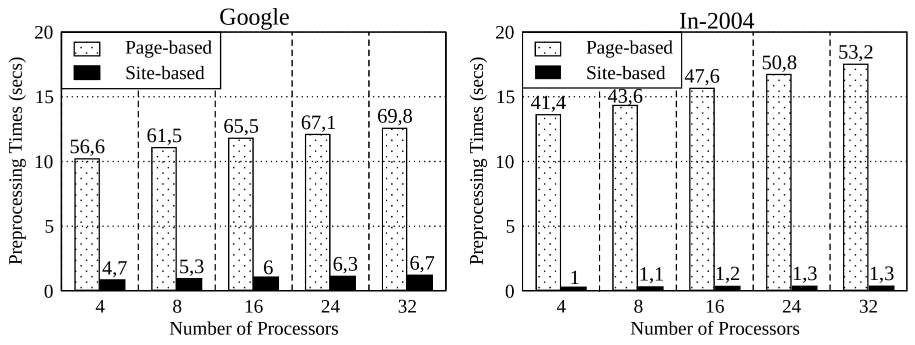


Fig. 3. Preprocessing times in seconds

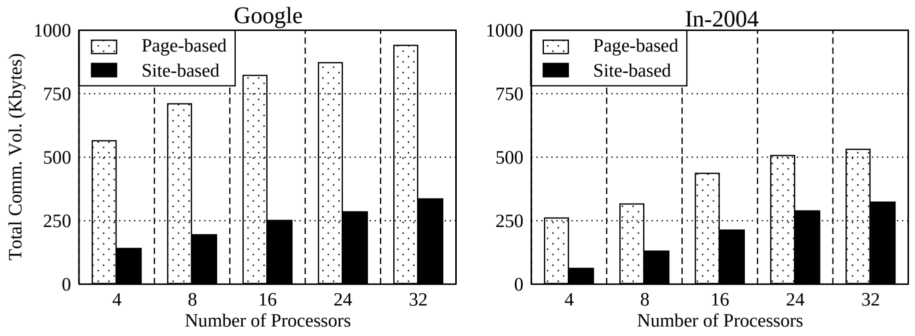


Fig. 4. Total communication volumes in Kbytes

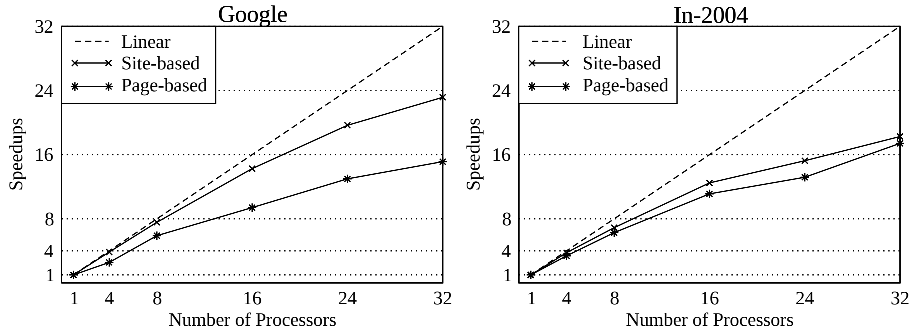


Fig. 5. Speedup curves

Fig. 3 displays the variation of the preprocessing times of page-based and site-based partitioning schemes with increasing number of processors. For the page-based scheme, the preprocessing time involves only the partitioning time, whereas for the site-based scheme, it involves both compression and partitioning times. As seen in Fig. 3, the proposed site-based partitioning scheme achieves a drastic reduction in the preprocessing time compared to the page-based scheme. For example, the site-based scheme performs the preprocessing approximately 11 and 40 times faster than the page-based scheme in partitioning the `Google` and `In-2004` datasets, on the overall average. In Fig. 3, the number annotated with each bar shows the ratio of preprocessing time to the sequential per iteration time. According to these values, in the `In-2004` dataset, the preprocessing time of the site-based scheme is approximately equal to a single iteration time of the sequential PageRank computation.

Fig. 4 displays the quality of the partitions obtained by the page-based and site-based partitioning schemes in terms of total communication volume. As seen in the figure, the partitions obtained by the site-based scheme incurs 70%

and 54% less communication volume than those of the page-based scheme for the Google and In-2004 datasets, respectively. These experimental findings verify the expectation that sites constitute natural clusters of pages.

In order to compare the speedup performances of the page- and site-based partitioning schemes, the parallel PageRank algorithm proposed in Section 3 is implemented using the ParMxvLib library [22]. The parallel PageRank algorithm is run on a 32-node PC cluster interconnected by a Fast Ethernet switch, where each node contains an Intel Pentium IV 3.0 GHz processor, 1 GB of RAM. The speedup curves are given in Fig. 5. As seen in the figure, the site-based partitioning scheme leads to higher speedup values than the page-based scheme, in accordance with the reduction in the communication volumes. For example, the site-based scheme leads to a speedup of approximately 24 on 32 processors, whereas the page-based scheme achieves a speedup of 16.

6 Conclusion

An efficient parallelization technique for PageRank computation was proposed and implemented. Experimental results show that, compared to a state-of-the-art parallelization scheme, the proposed technique not only reduces the preprocessing time drastically, but also reduces the parallel per iteration time. Although the proposed parallelization scheme is applied on a particular power method formulation, the underlying ideas can be easily and effectively applied to the parallelization of other iterative method formulations investigated in the literature for PageRank computation.

References

1. Aykanat, C., Pinar, A., Catalyurek, U.V.: Permuting sparse rectangular matrices into block-diagonal form. SIAM J. Scientific Computing 25(6), 1860–1879 (2004)
2. Aykanat, C., Cambazoglu, B.B., Ucar, B.: Multilevel hypergraph partitioning with multiple constraints and fixed vertices. J. Parallel and Distributed Computing. (submitted)
3. Berkhin, P.: A survey on PageRank computing. Internet Mathematics 2(1), 73–120 (2005)
4. Bradley, J.T., Jager, D.V., Knottenbelt, W.J., Trifunovic, A.: Hypergraph partitioning for faster parallel PageRank computation. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) Formal Techniques for Computer Systems and Business Processes. LNCS, vol. 3670, pp. 155–171. Springer, Heidelberg (2005)
5. Brezinski, C., Redivo-Zaglia, M., Serra Capizzano, S.: Extrapolation methods for PageRank computations. Comptes Rendus de l'Académie des Sciences de Paris, Series I 340, 393–397 (2005)
6. Catalyurek, U.V., Aykanat, C.: Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In: Saad, Y., Yang, T., Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1996. LNCS, vol. 1117, pp. 75–86. Springer, Heidelberg (1996)
7. Catalyurek, U.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on Parallel and Distributed Systems 10(7), 673–693 (1999)

8. Catalyurek, U.V., Aykanat, C.: A multilevel hypergraph partitioning tool, version 3.0. Tech. Rep., Bilkent University (1999)
9. Gleich, D., Zhukov, L., Berkhin, P.: Fast parallel PageRank: A linear system approach. Tech. Rep. YRL-2004-038, Yahoo! (2004)
10. Gyöngyi, Z., Garcia-Molina, H., Pedersen, J.: Combating Web spam with TrustRank. In: Proc. 30th Int'l Conf. on VLDB, pp. 576–587 (2004)
11. Haveliwala, T.: Topic sensitive PageRank. In: Proc. 11th Int'l WWW Conf., pp. 517–526 (2002)
12. Ipsen, I.C.F., Kirkland, S.: Convergence analysis of a PageRank updating algorithm by Langville and Meyer. SIAM J. Matrix Anal. Appl. 27, 952–967 (2006)
13. Ipsen, I.C.F., Selee, T.M.: PageRank computation, with special attention to dangling nodes. SIAM J. Matrix Anal. Appl. (2007) (submitted)
14. Ipsen, I.C.F., Wills, R.S.: Mathematical properties and analysis of Google's PageRank. Bol. Soc. Exp. May. Apl. 34, 191–196 (2006)
15. Kamvar, S., Haveliwala, T., Manning, C., Golub, G.: Extrapolation methods for accelerating PageRank computations. In: Proc. 12th Int'l WWW Conf., pp. 261–270 (2003)
16. Kamvar, S., Haveliwala, T., Golub, G.: Adaptive methods for computation of PageRank. In: Proc. Int'l Conf. on the Numerical Solution of Markov Chains (2003)
17. Kamvar, S., Haveliwala, T., Manning, C., Golub, G.: Exploiting the block structure of the Web for computing PageRank. Tech. Rep., Stanford Univ. (2003)
18. Langville, A.N., Meyer, C.D.: Deeper inside PageRank. Internet Mathematics 1(3), 335–380 (2005)
19. Langville, A.N., Meyer, C.D.: A reordering for the PageRank problem. SIAM J. Scientific Computing 27(6), 2112–2120 (2006)
20. Manaskasemsak, B., Rungsawang, A.: Parallel PageRank computation on a gigabit PC cluster. In: Proc. AINA'04, pp. 273–277 (2004)
21. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the Web. Tech. Rep. 1999-66, Stanford Univ. (1999)
22. Ucar, B., Aykanat, C.: A library for parallel sparse matrix-vector multiplies. Tech. Rep. BU-CE-0506, Department of Computer Engineering, Bilkent University, Ankara, Turkey (2005)
23. Ucar, B., Aykanat, C.: Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for matrix-vector multiplies. SIAM J. Scientific Computing. 25(6), 1837–1859 (2004)

Is Cache-Oblivious DGEMM Viable?

John A. Gunnels[1], Fred G. Gustavson[1], Keshav Pingali[2], and Kamen Yotov[2]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
{gunnels, fg2}@us.ibm.com
[2] Dept. of Computer Science, Cornell University, Ithaca, NY 14853, USA
{pingali, kyotov}@cs.cornell.edu

Abstract. We present a study of implementations of DGEMM using both the cache-oblivious and cache-conscious programming styles. The cache-oblivious programs use recursion and automatically block DGEMM operands A, B, C for the memory hierarchy. The cache-conscious programs use iteration and explicitly block A, B, C for register files, all caches and memory. Our study shows that the cache-oblivious programs achieve substantially less performance than the cache-conscious programs. We discuss why this is so and suggest approaches for improving the performance of cache-oblivious programs.

1 Introduction

One recommendation of the Algorithms and Architectures (AA) approach [1,6,10] is that researchers from Architecture, Compilers and Algorithms communicate. In this spirit, the authors of this paper, who are from the compilers and algorithms areas, have been collaborating in a research project to build BRILA (Block Recursive Implementation of Linear Algebra), a domain-specific compiler for Dense Linear Algebra (DLA) programs. This compiler takes recursive descriptions of linear algebra problems, and produces optimized iterative or recursive programs as output.

As part of this effort, we investigated the cache-oblivious (CO) and cache-conscious (CC) approaches to implementing programs for machines with memory hierarchies. CO research introduced recursion via the divide and conquer paradigm into DLA approximately ten years ago [9,4]. The work in [9] was inspired by earlier work [1] which first enunciated the AA approach; see also [10,6]. [9] additionally advocated the use of new data structures (NDS) and L1 cache blocking to improve the performance of DLA recursive algorithms such as the Level 3 BLAS [6].

The results described in this paper summarize and extend a companion paper directed towards the optimizing compiler community [15]. Our main findings can be summarized as follows. As is well-known, the performance of many programs such as DGEMM is limited by the performance of the memory system in two ways. First, the latency of memory accesses can be many hundreds of cycles, so the processor may be stalled most of the time, waiting for reads to complete. Second, the bandwidth from memory is usually far less than the rate at which the processor can consume data. Our study examined these limitations for highly optimized CO and CC DGEMM programs, generated by BRILA, on four modern architectures: IBM Power 5, Sun UltraSPARC IIIi, Intel Itanium 2, and Intel Pentium 4 Xeon. We found that there is a significant gap between the performance of CO and CC algorithms for DGEMM; we provide reasons why this gap exists and how it might be overcome. We are not aware of any similar study in the literature.

The rest of this paper is organized as follows. In Section 2 we give a quantitative analysis of how blocking can reduce the latency of memory accesses as well as the bandwidth required from memory. Using this analysis, one can tell when an architecture cannot deliver a peak performing DGEMM. This was the case for the IBM PC604 computer whose L1 cache was too small and whose bandwidth between L2 and memory was too small[1]. In Section 3 we discuss the performance of naïve iterative and recursive programs. Neither program performs well on any architecture but for different reasons: the iterative program performs poorly mainly because of poor memory hierarchy behavior, and the recursive one behaves poorly mainly because of recursive overhead. In Section 4 we evaluate approaches for reducing recursive overhead. Following [4] we introduce, in [15], a *recursive microkernel* which does instruction scheduling and register blocking for a small problem of size $R_U \times R_U \times R_U$. However, even after considerable effort, we are unable to produce a recursive microkernel that performs well. In Section 5 we explore *iterative* microkernels produced by the BRILA compiler using tiling and unrolling of the standard iterative programs. We show that these iterative microkernels perform much better than the recursive microkernels. A main finding of [3, 11, 15] is that prefetching is important to obtain better performance. While prefetching is easy if the outer control structure is iterative, it is not clear how to accomplish this if the outer control structure is recursive. In Section 6 we discuss the importance of two recent architectural innovations: streaming and novel floating-point units [3, 16]. We describe a new concept called the L0/L1 cache interface for reasoning about the impact of these innovations [11]. Lastly, in Section 7, we summarize our findings on recursive and iterative approaches to matrix multiplication.

2 Approximate Blocking

The cache-oblivious approach can be viewed as a way of performing approximate blocking for memory hierarchies. Each step of the recursive divide-and-conquer process generates sub-problems of smaller size, and when the working set of a sub-problem fits in some level of the memory hierarchy, that sub-problem can execute without capacity misses at that level. It is known that this recursive approach is *I/O optimal* for common problems like matrix-multiplication and FFT, which means intuitively that the volume of data transfers between different cache

[1] The IBM PC architecture introduced prefetching instructions (called touches), and using them and the AA approach, one of us introduced the concept of algorithmic prefetching to improve DGEMM performance on this platform.

levels is as small as it can be (to within constant factors) for any program implementing the same computation [7]. However, program performance is limited by both the latency of memory accesses and the bandwidth between different levels of the memory hierarchy. We argue in this section that minimizing the volume of data transfers between cache levels by approximate blocking may reduce bandwidth demands to an adequate level, but may not necessarily address the latency problem.

We consider blocked Matrix-Matrix Multiply (MMM) of $N \times N$ matrices in which each block computation multiplies matrices of size $N_B \times N_B$. We assume that there is no data reuse between block computations, which is a conservative assumption for both latency and bandwidth estimates. We find an upper bound on N_B by considering a simple two-level memory hierarchy model with a cache and main memory, and requiring the size of the working set of the block computation to be less than the capacity of the cache, C. Assume that the line size is L_C, and that the cache has an access latency l_C. Let the access latency of main memory be l_M. The working set is bounded above by the size of the sub-problem. Therefore, the following inequality is a conservative approximation:

$$3N_B^2 \leq C \ . \tag{1}$$

The total number of memory accesses each block computation makes is $4N_B^3$. Each block computation brings $3N_B^2$ data into the cache, which results in $\frac{3N_B^2}{L_C}$ cold misses. If the block size is chosen so that the working set fits in the cache and there are no conflict misses, the cache miss ratio of the complete block computation is $\frac{3}{4N_B \times L_C}$. Assuming that memory accesses are not overlapped, the expected memory access latency is as follows:

$$l = \left(1 - \frac{3}{4N_B \times L_C}\right) \times l_C + \frac{3}{4N_B \times L_C} \times l_M \ . \tag{2}$$

Equation (2) shows that the expected latency decreases with increasing N_B, so latency is minimized by choosing the largest N_B for which the working set fits in the cache. In practice, the expected memory latency computed from Equation (2) is somewhat pessimistic because loads can be overlapped with each other or with actual computations, reducing the effective values of l_C and l_M. These optimizations are extremely important in the generation of the micro-kernels, as is described in Section 4. Furthermore, hardware and software prefetching can also be used to reduce effective latency, as is discussed in Section 5 of [15] and Section 6.

Bandwidth considerations provide a lower bound for N_B. We start by considering the bandwidth between the registers and the L1 cache, which we will refer to as $B(L0, L1)$. Assume that the processor can perform one fused multiply-add per cycle. Therefore, the time to execute a matrix-multiply is bounded above by N^3. Without blocking for registers, the bandwidth required between registers and the L1 cache is therefore $4N^3 \div N^3 = 4$ doubles/cycle.

If the processor cannot sustain this bandwidth, it is necessary to perform register-blocking. If the size of the register block is N_B, we see that each block computation requires $4N_B^2$ data to be moved, so our simple memory model implies that the total data movement is $\left(\frac{N}{N_B}\right)^3 \times 4N_B^2 = \frac{4N^3}{N_B}$. The ideal execution time of the computation is still N^3, so the bandwidth required from memory is $\frac{4N^3}{N_B} \div N^3 = \frac{4}{N_B}$ doubles/cycle. Therefore, register-blocking by a factor of N_B reduces the bandwidth required from L1 cache by the same factor.

We can now write the following lower bound on the value of N_B, where $B(L0, L1)$ is the bandwidth between registers and the L1 cache:

$$\frac{4}{N_B} \leq B(L0, L1)\ . \tag{3}$$

Inequalities (1) and (3) imply the following inequality for N_B:

$$\frac{4}{B(L0, L1)} \leq N_B \leq \sqrt{\frac{C}{3}}\ . \tag{4}$$

This argument generalizes to a multi-level memory hierarchy. If $B(L_i, L_{i+1})$ is the bandwidth between levels i and $i+1$ in the memory hierarchy, $N_B\,(i)$ is the block size for the i^{th} cache level, and C_i is the capacity of this cache, we obtain the following inequality:

$$\frac{4}{B(L_i, L_{i+1})} \leq N_B\,(i) \leq \sqrt{\frac{C_i}{3}}\ . \tag{5}$$

In principle, there may be no values of $N_B\,(i)$ that satisfy the inequality. This can happen if the capacity of the cache as well as the bandwidth to the next level of the memory hierarchy are small. According to this model, the bandwidth problem for such problems cannot be solved by blocking. The IBM PC604 processor is an example of such a processor.

Equation (5) shows that in general, there is a range of block sizes for which bandwidth constraints can be satisfied. In particular, if the upper bound in Equation (5) is more than twice the lower bound, the divide-and-conquer process in the cache-oblivious approach will generate a block size that lies within these bounds, and bandwidth constraints will be satisfied. However, Equation (2) suggests that latency might still be a problem, and that it may be a bigger problem for the CO approach since blocking only approximate.

3 Naive Recursive and Iterative DGEMM Routines

As a baseline for performance comparisons, we considered naïve recursive routines that recursed down to 1×1 matrix multiplications, as well as unblocked iterative routines for MMM. We found that both performed very poorly but for different reasons. We show that recursive codes have low cache misses but high calling overhead whereas the opposite holds for iterative codes.

For these implementations we need a data structure for matrices and a control structure. We use standard Fortran and C two dimensional arrays to represent the matrix operands A, B, C of DGEMM. In [9,6] the *all-dimensions* (AD) stategy of divide and conquer is described. Here one divides the rows and columns of A, B, C equally and thereby generates eight sub-problems of half the size. For the iterative control, we used simple *jki* loop order, [5]. This was the worst choice for iterative control as our matrices were stored in row major order. Both programs performed very poorly on all four platforms obtaining about 1% of peak. Three reason emerged as the main cause of bad performance:

1. The overhead of recursive calling was in the hundreds of cycles whereas an independent FMA executed in one cycle.
2. Both programs made poor use of the floating point register files. Currently, compilers fail to track register values across procedure calls. For iterative codes compilers can do register blocking; however, none of our compilers were able to do so.
3. A remarkable fact emerged when we examined L2 cache misses on Itanium. Our iterative code suffered roughly *two* cache misses per FMA resulting in a miss ratio of 0.5! So, poor memory behavior limits the performance of our iterative code. For the recursive code the miss ratio was a tiny .002 misses per FMA, resulting in a miss ratio of 0.0005!. This low miss ratio is a practical manifestation of the theoretical I/O optimality of recursive programs. However, because of the calling overhead, I/O optimality alone did *not* guarantee good overall performance.

Many more details and experimental results about this Section are contained in Section 3 of [15].

4 Recursive Kernels for DGEMM and Reducing Recursive Overhead

A standard approach to reducing the recursive overhead is to cut the recursion off once the problem size is below some cut-off, and call a kernel routine to perform the sub-problem. For DGEMM one no longer performs single FMAs at the leaves of the recursion tree; instead, the program would call a kernel routine that would perform N_B^3 FMA's. This *microkernel* is a basic block obtained by fully unrolling the recursive code for a small problem of size $R_U \times R_U \times R_U$, as suggested by Frigo et al. [4]. This kernel routine has to be optimized carefully for the registers and the processor pipeline, so it is not "register-oblivious". We call this a *recursive* microkernel since the FMAs are performed in the same order as they were in the original recursive code.

The optimal R_U value is determined empirically for values between 1 and 15; *i.e.*, this microkernel must be done in a register file. The overhead of recursion is amortized over R_U^3 FMAs, rather than a single FMA. Furthermore, a compiler might be able to register allocate array elements used in the microkernel, which gives the effect of register blocking. Therefore, performance should improve significantly.

One also needs to worry about selecting a good data structure to represent the matrix operands A, B, C of the microkernel as the standard Fortran and C two dimensional arrays are often a poor choice [10]. Virtually all high performance BLAS libraries internally use a form of a contiguous blocked matrix, such as Row-Block-Row (RBR); *e.g.*, see [9, 10, 14, 6, 11]. An alternative is to use a recursive data layout, such as a space filling curve like Morton-Z [9, 6]. In [15] we compared the MMM performance using both these choices and we rarely saw any performance improvement using Morton-Z order over RBR. Thus [15] used RBR in all of its experiments and it chose the data block size to match the kernel block size.

In [15] we considered four different approaches to performing register allocation and scheduling for the recursive microkernel; see Sections 4.1,2,3,4 and Table 4 of [15]. The results of these four recursive microkernel experiments led us to the following four conclusions:

– The microkernel is critical to overall performance. Producing a high performance recursive microkernel is a non-trivial job, and requires substantial programming effort.
– The performance of the program obtained by following the canonical recipe (recursive outer control structure and recursive microkernel) is substantially lower than the near-peak performance of highly optimized iterative codes produced by ATLAS or in vendor BLAS. The best we were able to obtain was 63% of peak on the Itanium 2; on the UltraSPARC, performance was only 38% of peak.
– For generating the microkernel code, using Belady's algorithm [2] followed by scheduling may not be optimal. Belady's algorithm minimizes the number of loads, but minimizing loads does not necessarily maximize performance. An integrated register allocation and scheduling approach appears to perform better.
– Most compilers we used did not do a good job with register allocation and scheduling for long basic blocks. The situation is more muddied when processors perform register renaming and out-of-order instruction scheduling. The compiler research community needs to pay more attention to this problem.

5 Iterative Kernel Routines for DGEMM

Iterative microkernels are obtained by register-blocking the iterative code, and have three degrees of freedom called K_U, N_U, and M_U. The microkernel loads a block of the C matrix of size $M_U \times N_U$ into registers, and then accumulates in them the results of performing a sequence of outer products between column vectors of A of size M_U and row vectors of B of size N_U. Therefore, $M_U + N_U$ loads of A and B operands accomplish $M_U \times N_U$ FMA's. Clearly, with a large enough register file one can do cache blocking out of higher levels of cache than L1. This actually happens on Itanium and on IBM BG/L, see [3]. With this kernel, K_U can be very large.

The microkernels were generated by BRILA in [15] as follows:

1. Start with a simple *kji* triply-nested loop for performing an MMM with dimensions $\langle K_U, N_U, M_U \rangle$ and unroll it completely to produce a sequence of $M_U \times N_U \times K_U$ FMAs.
2. Use the algorithm described in Figure 5 of [15] for register allocation and scheduling, starting with the sequence of FMAs generated above. As in Section 4.4 of [15], Belady register allocation was used and [15] scheduled dependent instructions back-to-back on the Pentium 4 Xeon.

5.1 Iterative Minikernels

By blocking explicitly for L1 and other cache levels, we get iterative *minikernels*. These minikernels usually deal with square blocks, and invoke the iterative microkernel to perform the matrix multiplications. We also considered combining recursive outer control structures with the iterative microkernel [15].

Not surprisingly, we found that blocking the iterative code for all levels of cache gave roughly the same performance as using a recursive outer control structure with the iterative microkernel. However, the iterative outer control offers many more blocking choices as there are $3! =$ six choices for each higher cache level, although this large number of choices can be reduced to only four choices [8]. Furthermore, to get the highest level of performance, it is necessary to implement prefetching from the memory, as is done in hand-tuned libraries. Prefetching for iterative control structures is well understood, but appears to be more difficult for recursive control structures because they access the matrices in a relatively unstructured fashion. We believe more research into prefetching for recursive codes is needed.

6 Pipelined Streaming and Novel Floating Point Units

Finally, we discuss the impact of novel features of the processor pipeline.

Many processors now support streaming but there are a limited number of streaming units on a given platform [16]. In Section 5.1, we discussed minikernels that block explicitly for L1 and other cache levels [1, 17, 11]. In [15] the data structure of the minikernel was RBR, which is a special case of the Square Block Format (SBF) [9, 10, 14]. In [14] the authors show that this data format minimizes L1, L2, and TLB misses for common matrix operations that involve both row and column operations. Typically, the order NB of a SBF matrix is chosen so it will reside in L1 cache. Unfortunately, streaming makes the use of SBF suboptimal since the iterative microkernel requires $2N_U + M_U$ streams with RBR storage. Usually, M_U and N_U is around four so about twelve streams are required. Unfortunately, twelve is too large. With NDS, we show that these streams can be reduced to three, one each for the A, B, C operands of DGEMM. We note that three is minimal for DGEMM as one stream is required for each of DGEMM's operands; see [11].

Next, we discuss novel floating point units. These are SIMD vector like and several platforms now have them. For example, on Intel one has SSE units which are capable of delivering 4,2 multiplies or 4,2 adds for single, double precision operands in a single cycle. The peak MFlop rate is thus quadrupled/doubled with this floating point unit. On IBM Blue Gene, BG/L, there is a pair of double SIMD Floating point units with associated double SIMD floating point register files [3]. It turns out that these units are not fully integrated with the CPU. Thus, there is an interface problem that exists in getting data from the L1 cache into the register files of these novel units. In [11] we discuss the matters in more detail and in particular, we define a new concept called the L0/L1 cache interface; also see [3]. Note that in this context the L0 cache is the register file of a given processor.

Now we briefly describe how these problems can be handled. An answer lies in changing the internal data structure [11]. Instead of using a standard Fortran or C two dimensional array, it is necessary to use a four-dimensional array. The inner two dimensions constitute register blocks. These register blocks are transposes of the register blocks described in Section 5. However, data in this new format can be addressed with memory stride of one so that twelve streams mentioned above become three. Furthermore, the L0/L1 interface problem disappears as data will now enter L1 (or bypass it completely) in a way that is optimal for its entrance into the L0 cache; see [11].

7 Conclusion

We summarize the results of [15] for the recursive and iterative programming styles. Our recursive microkernel work led us to the following conclusions.

- The performance of the program obtained by following the canonical recipe (recursive outer control structure and recursive microkernel) is substantially lower than the near-peak performance of highly optimized iterative codes produced by ATLAS or in vendor BLAS. The best we were able to obtain was 63% of peak on the Itanium 2; on the UltraSPARC, performance was only 38% of peak.
- The microkernel is critical to overall performance. Producing a high performance recursive microkernel is a non-trivial job, and requires substantial programming effort.
- For generating code for the microkernel, using Belady's algorithm [2] followed by scheduling may not be optimal. Belady's algorithm minimizes the number of loads, but minimizing loads does not necessarily correlate to performance. An integrated register allocation and scheduling approach appears to perform better.
- Most compilers we used did not do a good job with register allocation and scheduling for long basic blocks. This problem has been investigated before. The situation is more muddied when processors perform register renaming and out-of-order instruction scheduling. The compiler research community needs to pay more attention to this problem.

Our iterative microkernel and blocking work led us to the following conclusions.

- Using the techniques in the BRILA compiler, we can generate iterative microkernels giving close to peak performance. They perform significantly better than recursive microkernels.
- Wrapping a recursive control structure around the iterative microkernel gives a program that performs reasonably well since it is able to block approximately for all levels of cache and block exactly for registers. If an iterative outer control structure is used, it is necessary to block for relevant levels of the memory hierarchy.
- To achieve performance competitive with hand-tuned kernels, minikernels need to do data prefetching. It is clear how to do this for an iterative outer control structure but we do not know how to do this for a recursive outer control structure.

References

1. Agarwal, R.C., Gustavson, F.G., Zubair, M.: Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. IBM Journal of Research and Development 38(5), 563–576 (1994)
2. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
3. Chatterjee, S., et al.: Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. IBM Journal of Research and Development 49(2-3), 377–391 (2005)
4. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache-oblivious Algorithms. In: FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, p. 285. IEEE Computer Society Press, Los Alamitos (1999)
5. Dongarra, J.J., Gustavson, F.G., Karp, A.: Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. SIAM Review 26(1), 91–112 (1984)
6. Elmroth, E., Gustavson, F.G., Kågström, B., Jonsson, I.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)
7. Hong, J.-W., Kung, H.T.: I/O complexity: The red-blue pebble game. In: Proc. of the thirteenth annual ACM symposium on Theory of computing, pp. 326–333 (1981)
8. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: A Family of High-Performance Matrix Multiplication Algorithms. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 256–265. Springer, Heidelberg (2006)
9. Gustavson, F.G.: Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. IBM Journal of Research and Development 41(6), 737–755 (1997)
10. Gustavson, F.G.: High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. IBM Journal of Research and Development 47(1), 31–55 (2003)

11. Gustavson, F.G., Gunnels, J.A., Sexton, J.C.: Minimal Data Copy for Dense Linear Algebra Factorization. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 540–549. Springer, Heidelberg (2007)
12. Gustavson, F.G., Henriksson, A., Jonsson, I., Kågström, B., Ling, P.: Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In: Kagström, B., Elmroth, E., Waśniewski, J., Dongarra, J.J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 195–206. Springer, Heidelberg (1998)
13. Gustavson, F.G., Henriksson, A., Jonsson, I., Kågström, B., Ling, P.: Superscalar GEMM-based level 3 BLAS—the on-going evolution of a portable and high-performance library. In: Kagström, B., Elmroth, E., Waśniewski, J., Dongarra, J.J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 207–215. Springer, Heidelberg (1998)
14. Park, N., Hong, B., Prasanna, V.K.: Tiling, Block Data Layout, and Memory Hierarchy Performance. IEEE Trans. Parallel and Distributed Systems 14(7), 640–654 (2003)
15. Roeder, T., Yotov, K., Pingali, K., Gunnels, J., Gustavson, F.: The Price of Cache Obliviousness. Department of Computer Science, University of Texas, Austin Technical Report CS-TR-06-43 (September 2006)
16. Sinharoy, B., Kalla, R.N., Tendler, J.M, Kovacs, R.G., Eickemeyer, R.J., Joyner, J.B.: POWER5 System Microarchitecture. IBM Journal of Research and Development 49(4/5), 505–521 (2005)
17. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing (1-2), 3–35 (2001)

Partitioning and Blocking Issues for a Parallel Incomplete Factorization

Pascal Hénon, Pierre Ramet, and Jean Roman

ScAlApplix Project, INRIA Futurs and LaBRI UMR 5800,
Université Bordeaux 1, FR-33405 Talence Cedex, France
{henon, ramet, roman}@labri.fr

Abstract. The purpose of this work is to provide a method which exploits the parallel blockwise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust and efficient preconditioners based on a parallel incomplete factorization.

1 Introduction

Over the past few years, parallel sparse direct solver have made significant progress. They are now able to solve efficiently real-life three-dimensional problems having in the order of several millions of equations (see for example [1,5,8]).

Nevertheless, the need of a large amount of memory is often a bottleneck in these methods. On the other hand, the iterative methods using a generic preconditioner like an ILU(k) factorization [21] require less memory, but they are often unsatisfactory when the simulation needs a solution with a good precision or when the systems are ill-conditioned. The incomplete factorization technique usually relies on a scalar implementation and thus does not benefit from the superscalar effects provided by the modern high performance architectures. Futhermore, these methods are difficult to parallelize efficiently, more particulary for high values of level-of-fill.

Some improvements to the classical scalar incomplete factorization have been studied to reduce the gap between the two classes of methods. In the context of domain decomposition, some algorithms that can be parallelized in an efficient way have been investigated in [14]. In [19], the authors proposed to couple incomplete factorization with a selective inversion to replace the triangular solutions (that are not as scalable as the factorization) by scalable matrix-vector multiplications. The multifrontal method has also been adapted for incomplete factorization with a threshold dropping in [10] or with a fill level dropping that measures the importance of an entry in terms of its updates [2]. In [3], the authors proposed a block ILU factorization technique for block tridiagonal matrices.

Our goal is to provide a method which exploits the parallel blockwise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust parallel incomplete factorization based preconditioners [21] for iterative solvers.

For direct methods, in order to achieve an efficient parallel factorization, solvers usually implement the following processing chain:

- the *ordering* phase, which computes a symmetric permutation of the initial matrix A such that factorization will exhibit as much concurrency as possible while incurring low fill-in. In this work, we use a tight coupling of the Nested Dissection and Approximate Minimum Degree algorithms [17];
- the *block symbolic factorization* phase, which determines the block data structure of the factored matrix L associated with the partition resulting from the ordering phase;
- the *block repartitioning and scheduling* phase, which refines the partition, by splitting large supernodes in order to exploit concurrency within dense block computations, and maps it onto the processors;
- the *parallel numerical factorization* and the *forward/backward elimination* phases, which are driven by the distribution and the scheduling of the previous step.

In our case, we propose to extend our direct solver PaStiX [8] to compute an incomplete *block factorization* that can be used as a preconditioner in a Krylov method. The main work will consist in replacing the *block symbolic factorization* step by some algorithms able to build a dense block structure in the incomplete factors. We keep the ordering computed by the direct factorization to exhibit parallelism. *Reverse Cuthill and McKee* techniques are known to be efficient for small values of level-of-fill (0 or 1), but, to obtain robust preconditioners, we have to considere higher values of level-of-fill. In addition, the *Reverse Cuthill and McKee* leads to an ordering that does not permit independent computation in the factorization and thus it is not adapted for parallelization. The extensions that are described also have to preserve the dependences in the elimination tree on which all direct solver algorithms rely.

2 Methodology

In the direct methods relying on a Cholesky factorization ($A = L.L^t$), the way to exhibit a dense block structure in the matrix L is directly linked to the ordering techniques based on the nested dissection algorithm (ex: METIS [15] or SCOTCH [16]). Indeed the columns of L can be grouped in sets such that all columns of a same set have a similar non zero pattern. These sets of columns, called supernodes, are then used to prune the block structure of L. The supernodes obtained with such orderings mostly correspond to the separators found in the nested dissection process of the adjacency graph $G(A)$ of matrix A. Another essential property of this kind of ordering is that it provides a block elimination tree that is well suited for parallelism [8].

An important result used in direct factorization is that the partition $\mathcal{P}$ of the unknowns induced by the supernodes can be found without knowning the non zero pattern of L. The partition $\mathcal{P}$ of the unknowns is then used to compute the block structure of the factorized matrix L during the so-called *block symbolic factorization.* This block symbolic factorization for direct method is a very low time and memory consuming step since it can be done on the quotient graph

$Q(G(A), \mathcal{P})$ with a complexity that is quasi-linear in respect to the number of edges in the quotient graph. We exploit the fact that:

$$Q(G(A), \mathcal{P})^* = Q(G^*(A), \mathcal{P})$$

where the exponent $*$ means "elimination graph". It is important to keep in mind that this property can be used to prune the block structure of the factor L because one can find the supernode partition from $G(A)$ [13]. For an incomplete ILU(k) factorization, these properties are not true any more in the general case. The incomplete symbolic ILU(k) factorization has a theorical complexity similar to the numerical factorization, but an efficient algorithm that leads to a practical implementation has been proposed [18]. The idea of this algorithm is to use searches of elimination paths of length $k+1$ in $G(A)$ in order to compute $G^k(A)$ which is the adjacency graph of the factor in ILU(k) factorization.

Another remark to reduce the cost of this step is that any set of unknowns in A that have the same row structure and column structure in the lower triangular part of A can be compressed as a single node in $G(A)$ in order to compute the symbolic ILU(k) factorization. Indeed the corresponding set of nodes in $G(A)$ will have the same set of neighbors and consequently the elimination paths of length $k+1$ will be the same for all the unknowns of such a set. In other words, if we consider the partition $\mathcal{P}_0$ constructed by grouping sets of unknowns that have the same row and column pattern in A then we have:

$$Q(G^k(A), \mathcal{P}_0) = Q(G(A), \mathcal{P}_0)^k.$$

This optimization is very valuable for matrices that come from finite element discretization since a node in the mesh graph represents a set of several unknowns (the degrees of freedoms) that forms a clique. Then the ILU(k) symbolic factorization can be devised with a significantly lower complexity than the numerical factorization algorithm.

Once the elimination graph G^k is computed, the problem is to find a block structure of the incomplete factors. For direct factorization, the supernode partition usually produces some blocks that have a sufficient size to obtain a good superscalar effect using the BLAS 3 subroutines. The exact supernodes (group of successive columns that have the same non-zeros pattern) that are exhibited from the incomplete factor non zero pattern are usually very small. A remedy to this problem is to merge supernodes that have nearly the same structure. This process induces some *extra fill-in* compared to the exact ILU(k) factors but the increase of the number of operations is largely compensated by the gain in time achieved thanks to BLAS subroutines. The principle of our heuristic to compute the new supernode partition is to iteratively merge supernodes whose non zero patterns are the most similar until we reach a desired extra fill-in tolerance.

To summarize, our incomplete block factorization consists of the following steps:

– find the partition $\mathcal{P}_0$ induced by the supernodes of A;
– compute the incomplete block symbolic factorization $Q(G(A, \mathcal{P}_0))^k$;

- find the exact supernode partition in the incomplete factors;
- given a *extra fill-in* tolerance α , construct an approximated supernode partition $\mathcal{P}_\alpha$ to improve the block structure of the factors;
- apply a block incomplete factorization using the parallelization techniques implemented in our direct solver PaStiX [8,9].

The incomplete factorization is then used as a preconditioner in an iterative method using the block structure of the factors.

3 Amalgamation Algorithm

The previous section shows that the symbolic factorization of ILU(k) method, though more costly than in the case of exact factorizations, is not a limitation in our approach. Another remark is that the ordering step can be more expensive, in terms of memory and time, than the ILU(k) factorization but parallel ordering softwares are now available [6,7]. Nevertheless, in this paper, we use a sequential version of Scotch [16]. What remains critical is to obtain dense blocks with a sufficient size in the factor in order to take advantage of the superscalar effects provided by the BLAS subroutines. The exact supernodes that can be exhibited from the symbolic ILU(k) factor are usually too small to allow good BLAS efficiency in the numerical factorization and in the triangular solves. To address this problem we propose an amalgamation algorithm which aims at grouping some supernodes that have almost similar non-zero patterns in order to get bigger supernodes. By construction, the exact supernode partition found in any ILU(k) factor is always a sub-partition of the *direct supernode partition* (i.e. corresponding to the direct factorization). We impose the amalgamation algorithm to merge only ILU(k) supernodes that belong to the same direct supernode. That is to say that we want this *approximated supernode partition* to remain a sub-partition of the direct supernode partition. The rational is that when this rule is respected, the additional fill entries induced by the approximated supernodes can correspond to fill-paths in the elimination graph $G^*(A)$ whereas merging supernodes from different supernodes will result in "useless" extra fill (zero terms that does not correspond to any fill-path in $G^*(A)$). Thus, the extra fill created when respecting this rule has a better chance of improving the convergence rate. Future works will investigate a generalized algorithm that releases this constraint.

As mentioned before, the amalgamation problem consists of merging as many supernodes as possible while adding the least extra fill. We propose a heuristic based on a greedy algorithm: given the set of all supernodes, it consists of iteratively merging the pair of succesive supernodes $(i, i+1)$ which creates the least extra fill in the factor (see Figure 1) until a tolerance α is reached. Each time a pair of supernodes is merged into a single one the total amount of extra fill is increased: the same operation is repeated until the amount of additional fill entries reaches the tolerance α (given as a percentage of the number of non-zero elements found by the ILU(k) symbolic factorization). This algorithm requires to know at each step which pair of supernodes will add the least fill-in in the factors when they are merged. This is achieved by maintaining a heap that contains all the pair of supernodes sorted by their cost (in terms of new entries) to merge them. As said before, we only consider pair of ILU(k) supernodes that belong to the same *direct supernode*). This means that, each time two supernodes are merged, the number of extra fill that would cost to merge the new supernode with its father or its son (it can only have one inside a direct supernode) has to be updated in the heap.

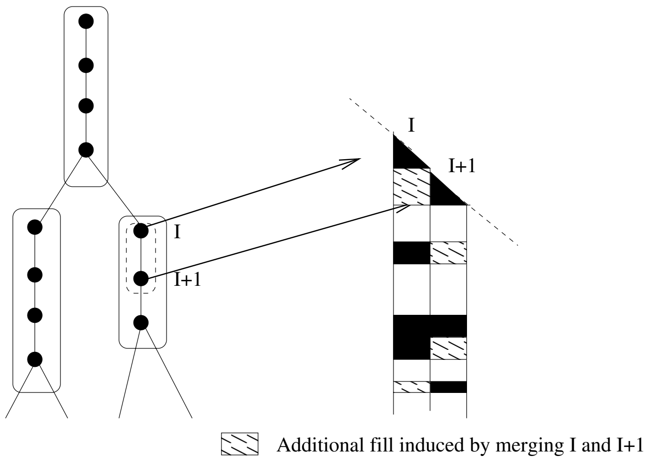


Fig. 1. Additional fill created when merging two supernodes I and I+1

The next section gives some results on the effect of the α parameter and a comparison to the classic scalar ILU(k) preconditioner.

4 Results

In this section, we consider 3 test cases from the PARASOL collection (see Table 1). $\mathrm{NNZ_A}$ is the number of off-diagonal terms in the triangular part of matrix A, $\mathrm{NNZ_L}$ is the number of off-diagonal terms in the factorized matrix L (for direct method) and OPC is the number of operations required for the factorization (for direct method).

Numerical experiments were performed on an IBM Power5 SMP node (16 processors per node) at the computing center of Bordeaux 1 university, France. We used a GMRES version without "restart". The stopping iteration criterion used in GMRES is the right-hand-side relative residual norm and is set to 10^{-7}.

For matrices that are symmetric definite positive, one could use a preconditioned conjugate gradient method; but at this time we only have implemented the GMRES method in order to treat unsymmetric matrices as well. The choice of the iterative accelerator is not in the scope of this study.

Table 1. Description of our test problems

Name	Columns	NNZ_A	NNZ_L	OPC
SHIPSEC5	179860	4966618	5.649801e+07	6.952086e+10
SHIP003	121728	3982153	5.872912e+07	8.008089e+10
AUDI	943695	39297771	1.214519e+09	5.376212e+12

Table 2 gives the influence of the amalgamation parameter α that is the percentage of extra entries in the factors allowed during the amalgamation algorithm in comparison to the ones created by the exact ILU(k) factorization. For the AUDI problem with several levels of fill (k), the table reports :

– the number of supernodes,
– the number of blocks,
– the number of non-zeros in the incomplete factors divided by the number of non-zeros in the initial matrix (`Fill-in`),
– the time of amalgamation in seconds (`Amalg.`),
– the time of sequential factorization in seconds (`Num. Fact.`),
– the time of triangular solve (forward and backward) in seconds (`Triang. Solve`)
– the number of iterations.

Table 2. Effect of amalgamation ratio α for AUDI problem

k	α	# Supernodes	# Blocks	Fill-in	Amalg.	Num. Fact.	Triang. Solve	Iterations
1	0%	299925	11836634	2.89	1.31	258	9.55	147
1	10%	198541	6332079	3.17	1.92	173	7.18	138
1	20%	163286	4672908	3.46	4.24	77	4.92	133
1	40%	127963	3146162	4.03	4.92	62	4.86	126
3	0%	291143	26811673	6.44	5.17	740	12.7	85
3	10%	172026	11110270	7.07	6.24	463	8.99	78
3	20%	136958	6450289	7.70	7.31	287	8.22	74
3	40%	108238	3371645	8.97	8.06	177	7.53	68
5	0%	274852	34966449	8.65	7.04	1567	14.3	69
5	10%	153979	12568698	9.50	7.72	908	10.7	63
5	20%	125188	6725165	10.35	8.84	483	9.44	59
5	40%	102740	3254063	12.08	9.87	276	8.65	52

We can see in Table 2 that our amalgamation algorithm reduces significantly the number of supernodes and the number of blocks in the dense block pattern of the matrix.

As a consequence, the superscalar effects are greatly improved as the amalgamation parameter grows: this is particulary true for the factorization which exploits BLAS-3 subroutines (matrix by matrix operations). The superscalar effects are less important on the triangular solves that require much less floating point operations and use only BLAS-2 subroutines (matrix by vector operations). We can also verify that the time to compute the amalgamation is negligible in comparison to the numerical factorization time. As expected the number of iterations decreases with the amalgamation fill parameter: this indicates that the

Table 3. Performances on 1 and on 16 processors PWR5 for 3 test cases

AUDI								
			1 processor			16 processors		
k	α	Iter.	Num. Fact.	Triang. Solve	Total	Num. Fact.	Triang. Solve	Total
1	10%	138	173	7.18	1163.84	26	0.65	115.70
1	20%	133	77	4.92	731.36	18	0.55	91.15
1	40%	126	62	4.86	674.36	11	0.47	70.22
3	10%	78	463	8.99	1164.22	58	1.25	155.50
3	20%	74	287	8.22	895.28	33	0.97	104.78
3	40%	68	177	7.53	689.04	17	0.70	64.60
5	10%	63	908	10.70	1582.10	89	1.59	189.17
5	20%	59	483	9.44	1039.96	47	1.26	121.34
5	40%	51	276	8.65	725.80	23	0.82	65.64
SHIP003								
			1 processor			16 processors		
k	α	Iter.	Num. Fact.	Triang. Solve	Total	Num. Fact.	Triang. Solve	Total
1	10%	–	1.41	0.28	–	0.32	0.05	–
1	20%	–	1.41	0.28	–	0.28	0.05	–
1	40%	–	1.58	0.29	–	0.28	0.04	–
3	10%	76	4.14	0.45	38.14	0.69	0.07	6.01
3	20%	75	4.05	0.45	37.80	0.62	0.05	4.37
3	40%	64	4.43	0.42	31.31	0.60	0.04	3.16
5	10%	49	7.81	0.55	34.76	1.13	0.07	4.56
5	20%	35	6.98	0.55	26.23	0.90	0.06	3.0
5	40%	34	7.24	0.49	23.9	0.98	0.06	3.02
SHIPSEC5								
			1 processor			16 processors		
k	α	Iter.	Num. Fact.	Triang. Solve	Total	Num. Fact.	Triang. Solve	Total
1	10%	121	1.28	0.32	40.0	0.28	0.03	3.91
1	20%	117	1.26	0.32	38.7	0.25	0.03	3.76
1	40%	111	1.44	0.33	38.07	0.24	0.03	3.57
3	10%	70	2.29	0.44	33.09	0.41	0.04	3.21
3	20%	66	2.29	0.43	30.67	0.38	0.04	3.02
3	40%	62	2.83	0.42	28.87	0.43	0.04	2.91
5	10%	54	3.32	0.51	30.86	0.54	0.05	3.24
5	20%	51	3.40	0.49	28.39	0.50	0.05	3.05
5	40%	47	4.11	0.47	26.2	0.59	0.05	2.94

extra fill allowed by the amalgamation corresponds to numerical non-zeros in the factors and are useful in the preconditioner.

Table 3 shows the results for the 3 problems both in sequential and in parallel for different levels-of-fill and different amalgamation fill parameter values. "–" indicates that GMRES did not converge in less than 200 iterations. As we can see the parallelization is quiet good since the speed-up is about 10 in most cases on 16 processors. This is particulary good considering the small amount of floating point operations required in the triangular solves.

The performance of a sequential scalar implementation of the columnwise ILU(k) algorithm are reported in Table 4. The "–" corresponds to cases where GMRES did not converged in less than 200 iterations. When compared to Tables 2 and 3 what can be noticed is that the scalar implementation is often better for a level-of-fill of 1 (really better for $\alpha = 0$) but is not competitive for higher level-of-fill values. The scalar implementation of the triangular solves is always the best compared to the blockwise implementation: we explain that by the fact that the blockwise implementation of the triangular solves suffers of the overcost paid to call the BLAS subroutines. It seems that this overcost is not

Table 4. Performances of a scalar implementation of the column-wise ILU(k) algorithm

AUDI					
k	Fill-in	Num. Fact.	Triang. Solve	Total	Iterations
1	2.85	75.0	2.63	482.65	155
3	6.45	466.9	4.95	922.3	92
5	8.72	1010.4	6.21	1488.57	77

SHIP03					
k	Fill-in	Num. Fact.	Triang. Solve	Total	Iterations
1	1.99	3.32	0.16	–	–
3	4.15	15.69	0.29	39.47	82
5	5.93	33.74	0.37	58.53	67

SHIPSEC5					
k	Fill-in	Num. Fact.	Triang. Solve	Total	Iterations
1	1.79	3.38	0.22	33.08	135
3	2.76	10.83	0.33	38.55	84
5	3.46	19.56	0.38	46.16	70

compensated by the acceleration provided by BLAS-2 subroutines compared to the scalar implementation. This is certainly due to the size of the block not being sufficiently for BLAS-2. On the contrary, a great difference is observed in the incomplete factorization between the scalar implementation and the blockwise implementation. In this case, the BLAS-3 subroutines offer a great improvement over the scalar implementation especially for the higher level-of-fill values that provide the bigger dense blocks and number of floating point operations in the factorization.

5 Conclusions

The main aims of this work have been reached. The blockwise algorithms presented in this work allow to significantly reduce the complete time to solve linear systems with incomplete factorization technique. High values of level-of-fill are manageable even in a parallel framework. Some future works will investigate a generalized algorithm that releases the constraint that imposes the amalgamation algorithm to merge only ILU(k) supernodes that belong to the same supernode. Furthermore, we will study a pratical way of setting automatically the extra fill-in tolerance α. We work on modifying the amalgamation algorithm such that it merges supernodes (and accept fill-in) while it can decreases the cost (in CPU time) of the preconditioner according to an estimation relying on a BLAS modelization.

References

1. Amestoy, P.R., Duff, I.S., Pralet, S., Vömel, C.: Adapting a parallel sparse direct solver to architectures with clusters of SMPs. Parallel Computing 29(11-12), 1645–1668 (2003)
2. Campbell, Y., Davis, T.A.: Incomplete LU factorization: A multifrontal approach. Tech Report TR95-024 (1995)

3. Chang, T.F., Vassilevski, P.S.: A framework for block ILU factorizations using block-size reduction. Math. Comput. 64 (1995)
4. Chapman, A., Saad, Y., Wigton, L.: High-order ILU preconditioners for CFD problems. Int. J. Numer. Meth. Fluids 33, 767–788 (2000)
5. Gupta, A.: Recent progress in general sparse direct solvers. In: Alexandrov, V.N., Dongarra, J.J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) Computational Science - ICCS 2001. LNCS, vol. 2073, pp. 823–840. Springer, Heidelberg (2001)
6. Chevalier, C., Pellegrini, F.: Improvement of the Efficiency of Genetic Algorithms for Scalable Parallel Graph Partitioning in a Multi-Level Framework. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 243–252. Springer, Heidelberg (2006)
7. Schloegel, K., Karypis, G., Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience 14(3), 219–240 (2002)
8. Hénon, P., Ramet, P., Roman, J.: PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. Parallel Computing 28(2), 301–321 (2002)
9. Hénon, P., Ramet, P., Roman, J.: Efficient algorithms for direct resolution of large sparse system on clusters of SMP nodes. In: SIAM Conference on Applied Linear Algebra, Williamsburg, Virginie, USA (July 2003)
10. Karypis, G., Kumar, V.: Parallel Threshold-based ILU Factorization. In: Proceedings of the IEEE/ACM SC97 Conference, ACM Press, New York (1997)
11. Li, X.S., Demmel, J.W.: A scalable sparse direct solver using static pivoting. In: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas (March 22-24, 1999)
12. Lipton, R.J., Rose, D.J., Tarjan, R.E.: Generalized nested dissection. SIAM Journal of Numerical Analysis 16(2), 346–358 (1979)
13. Liu, J.W.H., Ng, E.G., Peyton, B.W.: On finding supernodes for sparse matrix computations. SIAM J. Matrix Anal. Appl. 14(1), 242–252 (1993)
14. Magolu monga Made, M., Van der Vorst, A.: A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings. Future Generation Computer Systems 17(8), 925–932 (2001)
15. Karypis, G., Kumar, V.: A fast and high-quality multi-level scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20, 359–392 (1998)
16. Pellegrini, F.: SCOTCH 4.0 User's guide. Technical Report, INRIA Futurs (2005)
17. Pellegrini, F., Roman, J., Amestoy, P.: Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. Concurrency: Practice and Experience 12, 69–84 (2000)
18. Hysom, D., Pothen, A.: Level-based Incomplete LU factorization: Graph Model and Algorithms (pdf file) Tech Report UCRL-JC-150789, Lawrence Livermore National Labs, 19 (November 2002)
19. Raghavan, P., Teranishi, K., Ng, E.G.: A latency tolerant hybrid sparse solver using incomplete Cholesky factorization. Numer. Linear Algebra (2003)
20. Saad, Y.: ILUT: a dual threshold incomplete ILU factorization. Numerical Linear Algebra with Applications 1, 387–402 (1994)
21. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM (2003)
22. Watts III, J.W.: A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. Society of Petroleum Engineers Journal 21, 345–353 (1981)

Automatic Performance Tuning for the Multi-section with Multiple Eigenvalues Method for Symmetric Tridiagonal Eigenproblems

Takahiro Katagiri[1,2], Christof Vömel[1], and James W. Demmel[1]

[1] Computer Science Division, University of California, Berkeley,
387 Soda Hall, Berkeley, CA 94720-1776 U.S.A.
{voemel, demmel}@cs.berkeley.edu
[2] Graduate School of Information Systems,
The University of Electro-Communications
1-5-1 Chofu-gaoka, Choufu-shi, Tokyo 182-8585, Japan
katagiri@is.uec.ac.jp

Abstract. We propose multisection for the multiple eigenvalues (MME) method for determining the eigenvalues of symmetric tridiagonal matrices. We also propose a method using runtime optimization, and show how to optimize its performance by dynamically selecting the implementation parameters. Performance results using a Hitachi SR8000 supercomputer with eight processors per node yield (1) up to 6.3x speedup over a conventional multisection method, and (2) up to 1.47x speedup over a statically optimized MME method.

1 Introduction

The bisection method is widely used to compute the eigenvalues of a symmetric tridiagonal matrix T, especially when a subset of the spectrum is desired. The bisection method is based on repeated evaluation of the function $Count(x) =$ number of eigenvalues of T less than x. Evaluation of this function at two points a and b yields the number of eigenvalues in the interval $[a, b]$. There can be a lot of parallelism inherent in the bisection method. First, disjoint intervals may be searched independently, but the number of intervals available depends on the distribution of the eigenvalues of T. Second, one can divide an interval into more than two equal parts (i.e., multisection as opposed to bisection) but the efficiency depends on how much faster $Count(x(1 : k))$ can be evaluated at k points $x(1 : k)$ than at one point [5,8], which in turn depends on the computer being used to perform the calculation.

So to optimize the performance, an implementation of the bisection method could select the points at which to evaluate $Count(x)$ based on both the intervals that contain eigenvalues that have been found so far, and the relative speeds of $Count(x(1 : k))$ for different k. At one extreme, when the n-by-n matrix T has fairly uniformly distributed eigenvalues, multisection could be used initially until there were sufficient intervals to take advantage of the available parallelism. By doing this there would eventually be n disjoint intervals to each of which bisection could be applied. In the other extreme case in which all the eigenvalues of T lie in a few tight clusters so that there are never many intervals containing eigenvalues, multisection could be applied.

We propose a method for selecting parameters during runtime the optimal mix of multisection. Like other automatic tuning systems [9,1,4,2] at install-time a few benchmarks are run to determine the speed of $Count(x(1:k))$, and a simple performance model is used at runtime to decide what to do. We assume a shared memory parallel system in this paper but we also consider the use of vectorization. We call our method the multisection with multiple eigenvalues (MME) method. Our implementation is intended for inclusion in the forthcoming LAPACK and ScaLAPACK codes `xSTEGR` [3].

This paper is organized as follows. Section 2 explains the kernel of the MME method. Section 3 proposes a runtime optimization method for the MME method. Section 4 describes an evaluation of the proposed runtime method using a Hitachi SR8000 supercomputer. Finally, the conclusions from this study are given.

2 Kernel Derivation for the MME Method

2.1 LAPACK dlarrb Routine

The kernel of the bisection method explained in this section is based on the LAPACK `xSTEGR` implementation. Figure 1 shows the whole bisection kernel for the `dlarrb` routine in LAPACK `xSTEGR`. However, the MME method is not limited to the implementation in LAPACK `xSTEGR`.

The kernel in Fig. 1 returns the number of eigenvalues under the value σ. Note that the kernel in Fig. 1 is a bisection for one target eigenvalue. The total number of eigenvalues should be calculated in the `dlarrb` routine based on the relatively robust representation (RRR) [7]. The value of σ is determined using the interval for the target eigenvalue. This interval is given by the representation tree [7] in the multiple relatively robust representation (MRRR) algorithm.

The `dlarrb` routine performs "limited" bisection to refine the eigenvalues of $L\ D\ L^T$. IEEE-features [6], for example `NaN`, are used in the kernel to make it more rapid. The value of σ in Fig. 1 is the value of the bisection search for the target eigenvalue. It is calculated using the formula $a+(b-a)/2$ in the bisection method, where the current interval is $[a,b]$.

In the kernel shown in Fig. 1, there are three kinds of computational parts, namely: I) the upper part of $LDL^T - \sigma I = L_+\ D_+\ L_+^T$; II) the lower part of $LDL^T - \sigma I = U_-\ D_-\ U_-^T$; III) the twist index. This is because the MRRR algorithm uses a special data format of the target matrix, namely the twisted factorization of the target tridiagonal matrix T.

We consider that the condensed bisection kernel is represented by the line $\langle 2\rangle$–$\langle 7\rangle$ for part I) in Fig. 1, since the data dependency of part II) is the same as that of part I), while part III) is negligible. The kernel of part I can be regarded as the bisection kernel hereafter.

```
⟨0⟩ $S = 0$; $P = 0$; $NEG1 = 0$;  $NEG2 = 0$; $NEGCNT = 0$;
⟨1⟩ I) Upper Part
⟨2⟩ do $J = 1, R - 1$
⟨3⟩   $T = S - \sigma$
⟨4⟩   $DPLUS = D(J) + T$
⟨5⟩   $S = T*LLD(J) / DPLUS$
⟨6⟩   if ( $DPLUS$ .lt. $ZERO$ ) $NEG1 = NEG1 + 1$
⟨7⟩ enddo
⟨8⟩ if ($S$ .eq. NaN) Use a slower version the above loop;
⟨9⟩ $NEGCNT = NEGCNT + NEG1$

⟨10⟩ II) Lower Part
⟨11⟩ do $J = N - 1, R, -1$
⟨12⟩   $DMINUS = LLD(J) + P$
⟨13⟩   $P = P*D(J) / DMINUS - \sigma$
⟨14⟩   if ( $DMINUS$ .lt. $ZERO$ ) $NEG2 = NEG2 + 1$
⟨15⟩ enddo
⟨16⟩ if ($P$ .eq. NaN) Use a slower version the above loop;
⟨17⟩ $NEGCNT = NEGCNT + NEG2$

⟨18⟩ III) Twist Index
⟨19⟩ $GAMMA = S + P$
⟨20⟩ if ($GAMMA$ .lt. $ZERO$) $NEGCNT = NEGCNT + 1$
⟨21⟩ return ($NEGCNT$)
```

Fig. 1. The whole bisection dernel for the dlarrb routine in LAPACK xSTEGR. The variable R is a twist index for the twisted factorization of the tridiagonal matrix T.

2.2 The Bisection Kernel

Figure 2 shows the kernel of the bisection method for one target eigenvalue. The kernel of Fig. 2 cannot be parallelized, since it has a loop-carried flow-dependency for the variable S.

2.3 The Multisection Kernel

Fig. 3 shows the kernel for the multisection method with ML points for one target eigenvalue. The values of $\sigma(1 : ML)$ shown in Fig. 3 are calculated using $\sigma(i) = a + h \cdot i$, for $i = 1, 2, ..., ML$, where $h \equiv (b - a)/(ML + 1)$ and where the current interval is $[a,b)$.

The kernel of Fig. 3 can be parallelized for the outer loop of I, since none of the variables is dependent on I. However, there is a problem; namely, if we want to take a large vector length for I to reduce the parallel overhead, we should take a large number of points for multisection, given by the value of ML in Fig. 3. But the efficiency of searching decreases as ML increases due to the additional overhead of computing $NEG1(I)$. Hence, there is a trade-off between the parallel execution efficiency and the searching efficiency in the multisection kernel.

```
⟨0⟩ $S = 0$; $NEG1 = 0$;
⟨1⟩ do $J = 1, R - 1$
⟨2⟩   $T = S - \sigma$
⟨3⟩   $DPLUS = D(J) + T$
⟨4⟩   $S = T*LLD(J) / DPLUS$
⟨5⟩   if ( $DPLUS$ .lt. $ZERO$ ) $NEG1 = NEG1 + 1$
⟨6⟩ enddo
```

Fig. 2. The Bisection Kernel

```
⟨0⟩ $S(1:ML) = 0$; $NEG1(1:ML) = 0$;
⟨1⟩ do $I = 1, ML$
⟨2⟩   do $J = 1, R - 1$
⟨3⟩     $T(I) = S(I) - \sigma(I)$
⟨4⟩     $DPLUS(I) = D(J) + T(I)$
⟨5⟩     $S(I) = T(I)*LLD(J) / DPLUS(I)$
⟨6⟩     if ( $DPLUS(I)$ .lt. $ZERO$ ) $NEG1(I) = NEG1(I) + 1$
⟨7⟩   enddo; enddo
```

Fig. 3. The Multisection Kernel

2.4 The MME Kernel

Fig. 4 shows the kernel of MME. The values of σ ($1 : ML$, $1 : EL$) in Fig. 4 are calculated using $\sigma(i, k) = a_k + h_k \cdot i$, for $i = 1, 2, ..., ML$, where $h_k \equiv (b_k - a_k)/(ML + 1)$, if the current interval is $[a_k,b_k)$ for the k-th eigenvalue for $k = 1, 2, ..., EL$. The loops of K and I in Fig. 4 can be combined.

The MME kernel and the combined-loop MME kernel have the following advantages over the normal multisection method.

First, by using a long loop length for I, the parallel overhead can be reduced compared that of the normal multisection method. This is because we can make

```
⟨0⟩ $S(1:ML, 1:EL) = 0$; $NEG1(1:ML, 1:EL) = 0$;
⟨1⟩ do $K = 1, EL$
⟨2⟩   do $I = 1, ML$
⟨3⟩     do $J = 1, R - 1$
⟨4⟩       $T(I,K) = S(I,K) - \sigma(I,K)$
⟨5⟩       $DPLUS(I,K) = D(J) + T(I,K)$
⟨6⟩       $S(I,K) = T(I,K)*LLD(J) / DPLUS(I,K)$
⟨7⟩       if ( $DPLUS(I,K)$ .lt. $ZERO$ ) $NEG1(I,K) = NEG1(I,K) + 1$
⟨8⟩     enddo; enddo; enddo
```

Fig. 4. The multisection with multiple eigenvalues (MME) kernel

the length EL times the normal multisection length ML. Hence, the ratio is EL times greater, which is equal to the number of eigenvalues, than the normal multisection method.

Second, while we can select a small length for ML, the outer loop-length can be kept long by setting EL appropriately. This means that the searching efficiency can be kept high, since it is not necessary to use a long length for ML to obtain high parallelism.

One drawback of using the MME kernel is that if there is no multiple eigenvalues in the routine of `dlarrb`, the merit of the MME method will be the same as that of the normal multisection method.

2.5 Overall Process for the MME Method

Figure 5 shows the overall process for the MME method. Note that the number of eigenvalues in Fig. 5 cannot be known in advance even when computing all the eigenvalues for the input matrix. This is because the usage of the bisection routine strongly depends on the algorithm and the numerical characteristics of the input matrices. In the case of the MRRR algorithm, the number of eigenvalues is determined by the representation tree which is based on the numerical characteristics of the input matrix.

Hence, the parameters EL and ML cannot be optimized in advance, rather a runtime optimization of EL and ML has to be performed.

Figure 6 shows the details for the MME method in the LAPACK `dlarrb` routine. There are three computational parts for the eigenvalue computation in Fig. 6. They are: I) Computation of the left intervals for $[a_{J:J+EL-1}, a_{J:J+EL-1} + \delta_{J:J+EL-1}]$ in ⟨3⟩–⟨9⟩; II) Computation of the right intervals for $[b_{J:J+EL-1} - \delta_{J:J+EL-1}, b_{J:J+EL-1}]$ in ⟨10⟩–⟨16⟩; III) Improvement of the accuracy of the intervals $[a_{j_{uc}}, b_{j_{uc}}]$ in ⟨17⟩–⟨23⟩, for $j_{uc} \in [J, J+EL-1]$ for unconverged intervals in the previous two parts of I) and II), in Fig. 6. Where $\delta_{J:J+EL-1}$ are the previously calculated existence regions of the target eigenvalues previously calculated.

If the eigenvalues are strongly clustered, the computation for part III in ⟨17⟩–⟨23⟩ will be heavy.

```
⟨1⟩ if (#Eigenvalues .gt. 1) then
⟨2⟩   Call MME routine with $EL \equiv$ #Eigenvalues; $ML \equiv$ an appropriate value;
⟨3⟩ else
⟨4⟩   Call normal multisection routine with $ML \equiv$ an appropriate value;
⟨5⟩ endif
```

Fig. 5. Overall of MME method

```
⟨1⟩ do $J = 1$, #Eigenvalues, $EL$
⟨2⟩   Make sure that $[a_j,b_j]$ for $j$-th eigenvalue, where $j = J, ..., J + EL - 1$;
    ⟨3⟩ I) Compute $NEGCNT_j$ from $L_+D_+L_+^T = LDL^T - a_{J:J+EL-1}$.
    ⟨4⟩ while (all intervals are enough small)
    ⟨5⟩   Set the points of $\sigma(1 : ML, 1 : EL)$ in the left intervals of $a_{J:J+EL-1}$;
    ⟨6⟩   Call the MME kernel with $EL$ and $ML$;
    ⟨7⟩   Fix the interval of $a_{J:J+EL-1}$ using returned numbers on $\sigma(1 : ML, 1 : EL)$
    ⟨8⟩   Check all intervals;
    ⟨9⟩ end while
    ⟨10⟩ II) Compute $NEGCNT_j$ from $L_+D_+L_+^T = LDL^T - b_{J:J+EL-1}$.
    ⟨11⟩ while (all intervals are enough small)
    ⟨12⟩   Set the points of $\sigma(1 : ML, 1 : EL)$ in the right intervals of $b_{J:J+EL-1}$;
    ⟨13⟩   Call the MME kernel with $EL$ and $ML$;
    ⟨14⟩   Fix the interval of $b_{J:J+EL-1}$ using returned numbers on $\sigma(1 : ML, 1 : EL)$;
    ⟨15⟩   Check all intervals;
    ⟨16⟩ end while
    ⟨17⟩ III) There are unconverged intervals for $j_{uc} \in [J,J + EL - 1]$.
    ⟨18⟩ while ((all intervals are enough small) .or. (#iteration .gt. $MAXITER$))
    ⟨19⟩   Set the points of $\sigma(1 : ML, 1 : EL)$ in the current intervals of $[a_{j_{uc}}, b_{j_{uc}}]$;
    ⟨20⟩   Call the MME kernel with $EL$ and $ML$;
    ⟨21⟩   Fix the intervals of $[a_{j_{uc}},b_{j_{uc}}]$ using returned numbers on $\sigma(1 : ML, 1 : EL)$;
    ⟨22⟩   Check all intervals;
    ⟨23⟩ end while
⟨24⟩ enddo
⟨25⟩ if (#Eigenvalues is not divided by $EL$)
Call the Multisection kernel with $ML$ for the rest eigenvalue computations;
```

Fig. 6. The details of the MME Method. The details of the dlarrb routine for the MME method are also shown.

3 The Runtime Optimization Method

3.1 An Overview of the Process

We have already mentioned that runtime optimization is required to optimize the parameters EL and ML, which are the number of eigenvalues and multisection points for each target eigenvalue for the MME method, respectively. In this section we propose a method for runtime optimization.

Figure 7 shows an overview of the runtime optimization. This method has the following two phases: (1) At install time, the kernel is benchmarked using the procedure described above for many different values of EL and ML, and the time per point is determined and used to compute $Count(x)$ (the total time divided by $EL \cdot ML$). Then, the time is listed. The list is used to select the optimal parameters (EL, ML) at runtime. (2) At runtime, the optimal parameters (EL, ML) are selected, according to the runtime information on the number of disjoint eigenvalues.

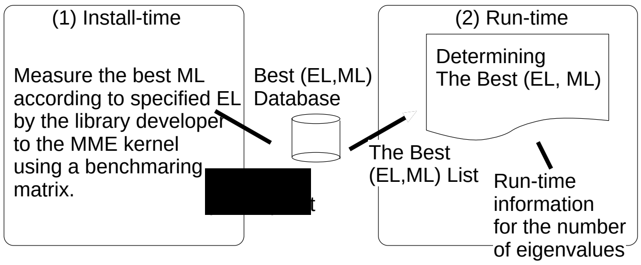

(1) Install-time
Measure the best ML according to specified EL by the library developer to the MME kernel using a benchmaring matrix.
[redacted]
Best (EL,ML) Database
(2) Run-time
Determining The Best (EL, ML)
The Best (EL,ML) List
Run-time information for the number of eigenvalues

Fig. 7. The runtime optimization method for MME

The details are given in the following. At install time, the kernel times were determined for all values of $EL \in [1, MAXEL]$ and $ML \in [1, MAXML]$, using a single test matrix. On a Hitachi SR8000 supercomputer, $MAXEL$ was set to 32, and $MAXML$ to 24, and a random test matrix of size 10,500 (large enough for accurate timing, while small enough to store in the cache) was used. Even though this computation only needs to be performed once per architecture, it can be sped up considerably by "pruning" large portions of the table where the time per point can be easily predicted to be quite large. On the Hitachi SR8000 supercomputer this was done using the restriction $EL \cdot ML \leq 16$ for $EL > 1$, where 16 is the value of ML that minimizes the time per point for $EL = 1$.

Next, still at install time, these timings were used to compute an optimal strategy (a set of (EL, ML) pairs) for any possible number m of disjoint intervals: for each m from 1 to $MAXEL$, the kernel was either run for $EL = m$ and the optimal value of ML determined, or else m was subdivided into two (or more) segments $m = m_1 + m_2$, and the kernel was run twice (or more), once with $EL = m_1$ (and the optimal value of ML) and once with $EL = m_2$ (and a possibly different value of ML). The result of this search process was stored in a list for use at runtime.

At runtime, at any point during the computation, there was a list of m not-yet-converged disjoint intervals each containing one or more eigenvalues. The table that had been constructed was used to rapidly select the optimal values of (EL, ML) to use. If $m > MAXEL$, m was broken into segments of size at most $MAXEL$.

3.2 Details of the Run-Time Optimization Method

Figure 8 shows the parameter optimization algorithm at install time.

In Fig. 8, there are two parts: I) Brute-force searching for a multisection in $\langle 1 \rangle$–$\langle 2 \rangle$; II) Heuristic searching for MME in $\langle 3 \rangle$–$\langle 13 \rangle$.

In part II), the ml is searched from 1 to MLE, such that $MLE \cdot el \leq ML^*$ to reduce the searching time on the line $\langle 4 \rangle$ in Fig. 8. This is based on an empirical fact that the optimal length of the outer loop for the MME kernel is constant for the SR8000, and the length can be measured in the "pure" multisection case,

```
⟨1⟩ Measure the kernel time for normal multisection, then find the best $ml$.
Thus, find the best for $ml \in [1,..,MAXML]$ with $el = 1$.
⟨2⟩ Let the best multisection point $ml$ be $ML^*$. Set $EL(1) = 1$ and $ML(1) = ML^*$.

⟨3⟩ do $el = 2,\ MAXEL$
⟨4⟩   Find the best $ml$ with the MME kernel with $EL = el$ and $ml \in [1,...,MLE]$,
      such that $(MLE \leq MAXML)$ and $(MLE \cdot el \leq ML^*)$.
⟨5⟩     if $(MLE$ .eq. $0)$ then $ml = 1$ is only measured.
⟨6⟩   Let the best $ml$ be $ML^*_{el}$ and the time be $T_{el}$.
  ⟨7⟩   do $co_{el} = 1,\ el - 1$ : Check The Time for Co-problems.
  ⟨8⟩     Find the best time of $(EL^*_{co\_best},\ ML^*_{co\_best})$ using the MME kernel
          with $EL = el$ for fixed $EL(co_{el})$ and $ML(co_{el})$.
  ⟨9⟩   enddo
⟨10⟩ Let the above best time be $T_{co\_best}$.
⟨11⟩ if $(T_{co\_best} < T_{el})$  $EL(el) = EL^*_{co\_best}$; $ML(el) = ML^*_{co\_best}$;
⟨12⟩ else  $EL(el) = el$; $ML(el) = ML^*_{el}$;
⟨13⟩ enddo
```

Fig. 8. The Parameter Optimization Algorithm at Install time

namely the case when $EL = 1$ on the line ⟨1⟩. This is a heuristic to reduce the searching time for ml. We call this heuristic "reduced search for ml."

The co-problems are solved in ⟨7⟩–⟨9⟩ using the table $(EL(i),\ ML(i)),i = 1,..,el$, which is the previously optimized table for parameter sets. The kernel is called twice or more when solving the co-problems. The table for the previously optimized parameter sets is also used to reduce the solving time for co-problems. This method thus sorted using a dynamic programming method. We call the table "the tuned table for co-problems."

When the heuristics of the reduced search is used for ml and the optimized table is used for co-problems, the frequency with which the kernel is called is reduced from $(1/2)MAXEL^2 \cdot MAXML + (1/2)MAXEL \cdot MAXML$ to $(1/2)MAXEL^2 + C \cdot MAXEL$, where C is a constant number. The algorithm in Fig. 8 is faster than the simple implementation by a factor of approximately $MAXML$. In our evaluation, we use $MAXML = 24$, hence the algorithm in Fig. 8 is faster by a factor of 24.

4 Performance Evaluation

4.1 Machine Environment

We used the Hitachi SR8000 supercomputer, with 8 PEs per node (corresponding to "job type E8E"), and the compiler used was the Hitachi OFORT90 version V01-04-/B with compiler options `-O4` and `-parallel=4`. FLOPS per node is 8 GFLOPS. This machine is installed at the Graduate School of Information Systems, The University of Electro-communications, Tokyo, Japan. The MME loops can be easily parallelized by using the Hitachi compiler.

4.2 Benchmarking Details

Our benchmark included four matrices: *Mat#1* with diagonals (-1,2,-1); *Mat#2* with uniform random entries in $[0, 1]$; *Mat#3,* the Wilkinson matrix W_n^+; *Mat#4* built from 100 "glued" copies of the Wilkinson matrix W_{21}^+. These matrices have very different eigenvalue distributions. The numbers for calling bisection routine depends on the matrix. For example, more than 3000 times is called in the *Mat#3*.

We used the dimension of $n = 2100$ for the Hitachi SR8000. We used the development version of `DSTEGR` [3] for LAPACK 4.0, which is known as the latest version 3.1, to compute all the eigenvalues and eigenvectors. It has two modes, DQDS mode and "bisection" mode. In DQDS mode the dqds algorithm computes the eigenvalues which are later refined using the bisection/multisection method. In "bisection" mode, `DSTEGR` uses only the bisection/multisection method. The latter mode is more appropriate for distributed memory parallelism.

Internally, we call the following different multisection variants:

– *Bisection:* The conventional algorithm ($EL = ML = 1$).
– *Basic Multisection:* $EL = 1$ and ML are selected to minimize the runtime per point.
– *Statically Tuned MME:* For each matrix, a fixed pair (EL, ML) was chosen from $(1 : 4, 8, 16, 32) \times (1, 2, 4, 8, 16, 32)$ to minimize the runtime. (EL, ML) was chosen by trying all $7 \cdot 6 = 42$ possibilities and choosing the best.
– *Run-Time Tuned MME:* This is described in Section 3.

4.3 The SR8000 Results

The tuning time in the parameter optimization at install-time for our runtime tuning method was about 120 seconds for the Hitachi SR8000.

Tables 1 and 2 show that in DQDS mode for the Hitachi SR8000, runtime tuned MME speeds up up to 6.8x over bisection, 6.3x over basic multisection, and

Table 1. Performance of Statically Tuned MME for the Hitachi SR8000

(a) Total Execution Time for `DSTEGR` Routine in DQDS mode.

Method / Benchmarking	Mat#1	Mat#2	Mat#3	Mat#4
bisection [s.]	0.347	1.83	15.2	8.20
Basic Multisection [s.], Best (ML)	0.323, (8)	1.45, (8)	5.90, (16)	3.30, (16)
Statically Tuned MME [s.], Best (EL,ML)	0.075, (16,2)	1.16, (2,8)	5.34, (2,8)	1.75, (16,1)

(b) Total Execution Time for the `DSTEGR` Routine in "bisection" mode.

Method / Benchmarking	Mat#1	Mat#2	Mat#3	Mat#4
bisection [s.]	3.94	7.11	23.9	17.2
Basic Multisection [s.], Best (ML)	2.11, (16)	3.69, (16)	8.70, (16)	6.19, (16)
Statically tuned MME [s.], Best (EL,ML)	0.51, (16,1)	2.93, (2,8)	7.89, (2,8)	2.75, (16,1)

Table 2. Performance of Run-Time Tuned MME for the Hitachi SR8000

(a) Total Execution Time for the DSTEGR Routine in DQDS mode.

Method / Benchmarking	Mat#1	Mat#2	Mat#3	Mat#4
Run-Time tuned MME [s.]	0.051	0.935	5.44	1.29
Speedup over bisection	<u>6.8x</u>	1.9x	2.7x	6.3x
Speedup over Basic Multisection	<u>6.3x</u>	1.5x	1.08x	2.5x
Speedup over Static Tuning	<u>1.47x</u>	1.2x	0.98x	1.3x

(b) Total Execution Time for DSTEGR Routine in "bisection" mode.

Method / Benchmarking	Mat#1	Mat#2	Mat#3	Mat#4
MME with Run-Time Fixing [s.]	0.53	2.08	7.21	2.33
Speedup over bisection	<u>7.4x</u>	3.4x	3.3x	7.3x
Speedup over Basic Multisection	<u>3.9x</u>	1.7x	1.2x	2.6x
Speedup over Static Tuning	0.96x	<u>1.40x</u>	1.09x	1.1x

1.47x over statically tuned MME. In one case it was 2% slower than statically tuned MME, but in all other cases it was always faster.

In bisection mode, the corresponding maximum speedups are 7.4x, 3.9x, and 1.4x. Again, in just one case the runtime tuned MME was 4% slower than the statically tune MME, but in all other cases it was always faster.

In Table 2, our run-time method especially can optimize the Mat#1. This is because, the benchmark is using a random matrix. Improving the efficiency of optimization is one of future work.

5 Conclusion

In this paper, we proposed a runtime auto-tuning method for the MME kernel for symmetric tridiagonal eigenproblems.

In the performance evaluation using a Hitachi SR8000 supercomputer we obtained considerable speedup using the runtime method compared to using the conventional bisection method. In addition, several speedups were obtained relative to the statically tuned case for the MME kernel.

The efficiency of the method using statically tuning for MME depends on the eigenvalue distribution for the input matrix. Hence, it is difficult to specify the best parameter before the routine runs because the input matrix is determined at runtime. The runtime tuning method can be used to obtain a tuning database, which is created in install-time. The database archives the auto-tuning according to the characteristics of input matrix without using manual-tuning. Thus, the runtime tuning method proposed in this paper is very useful for actual numerical libraries.

Acknowledgment

This research is partially supported by the Kayamori Foundation of Informational Science Advancement, and Grant-in-Aid for Scientific Research (C)

No.18500018, and Grant-in-Aid for Scientific Research on Priority Areas (Cyber Infrastructure for the Information-explosion Era) No.18049014.

References

1. Bilmes, J., Asanović, K., Chin, C.-W., Demmel, J.: Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In: Proceedings of International Conference on Supercomputing 97, pp. 340–347 (1997)
2. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R.C., Yelick, K.: Self-adapting linear algebra algorithms and software. In: Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation, 93(2) (2005)
3. Dhillon, I.S., Parlett, B.N., Vömel, C.: LAPACK working note 162: The design and implementation of the MRRR algorithm. Technical Report UCBCSD-04-1346, University of California, Berkeley (2004)
4. Frigo, M.: A fast Fourier transform compiler. In: Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 169–180, Atlanta, Georgia (May 1999)
5. Lo, S.-S., Philippe, B., Sameh, A.: A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. SIAM J. Sci. Stat. Comput. 8(2), s155–s165 (1987)
6. Marques, O.A., Riedy, E.J., Vömel, C.: LAPACK working note 172: Benefits of IEEE-754 features in modern symmetric tridiagonal eigensolvers. Technical Report UCBCSD-05-1414, University of California, Berkeley (2005)
7. Parlett, B.N., Dhillon, I.S.: Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices. Linear Algebra and Appl. 387, 1–28 (2004)
8. Simon, H.D.: Bisection is not optimal on vector processors. SIAM J. Sci. Stat. Comput. 10(1), 205–209 (1989)
9. Whaley, R., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27, 3–35 (2001)

Optimizing a Parallel Self-verified Method for Solving Linear Systems

Mariana Kolberg, Lucas Baldo, Pedro Velho,
Luiz Gustavo Fernandes, and Dalcidio Claudio

Faculdade de Informática, PUCRS,
Avenida Ipiranga, 6681 Prédio 16 - Porto Alegre, Brazil
{mkolberg,lbaldo,pedro,gustavo,dalcidio}@inf.pucrs.br

Abstract. Solvers for linear equation systems are commonly used in many different kinds of real applications, which deal with large matrices. Nevertheless, two key problems appear to limit the use of linear system solvers to a more extensive range of real applications: computing power and solution correctness. In a previous work, we proposed a method that employs high performance computing techniques together with verified computing techniques in order to eliminate the problems mentioned above. This paper presents an optimization of a previously proposed parallel self-verified method for solving dense linear systems of equations. Basically, improvements are related to the way communication primitives were employed and to the identification of the points in the algorithm in which mathematical accuracy is needed to achieve reliable results.

1 Introduction

Many real problems need numerical methods for their simulation and modeling. The result of such methods must be of high accuracy to achieve a good simulation. For a given problem, self-verified methods compute a highly accurate inclusion of the solution and automatically prove the existence and uniqueness of a true result within the given enclosure interval [5]. Finding the verified result often increases dramatically the execution times [8]. The use of these methods can increase the quality of the result, but also the execution time. It is known that interval arithmetic is more time consuming than real arithmetic, since the computation must be performed on two bounding values in each step. The use of parallel computing is a typical approach to improve the computational power and thus to solve large problems in reasonable time. The use of clusters plays an important role in such scenario as one of the most effective manner to improve the computational power without increasing costs to prohibitive values.

However, in some numerical problems, accuracy is essential. For instance, even in a simple linear system like presented in [1], the solution achieved with IEEE double precision arithmetic is completely wrong indicating the need for verified computation concepts. One solution for this problem can be found in [2], where the verified method for solving linear system using the C-XSC library is based on the enclosure theory described in [9].

There are some other options for achieving verified results. Among them INTLAB [4] and other algorithms described in recent papers. The algorithms proposed by Rump and Ogita [8,7] seem to be very promising. However there is no publicly available implemented version. Therefore we could not compare with our implementation. C-XSC seems to the authors the best option due to its exact scalar product, which could make a difference in the result accuracy in critical problems.

C-XSC [5], C for eXtended Scientific Computation, is a programming tool for the development of numerical algorithms which provide highly accurate and automatically verified results. C-XSC is not an extension of the standard language, but a class library which is written in C++. Therefore, no special compiler is needed. With its abstract data structures, predefined operators and functions, C-XSC provides an interface between scientific computing and the programming language C++. Besides, C-XSC supports the programming of algorithms which automatically enclose the solution of given mathematical problems in verified bounds. Such algorithms deliver a precise mathematical statement about the true solution.

Enclosure methods characteristics can be observed through the analysis of the Newton-like iteration (Equation 1):

$$x_{k+1} = Rb + (I - RA)x_k, k = 0, 1, ... \qquad (1)$$

This equation is used to find a zero of $f(x) = Ax - b$ with an arbitrary starting value x_0 and an approximate inverse $R \approx A^{-1}$ of A. If there is an index k with $[x]_{k+1} \overset{\circ}{\subset} [x]_k$ (the $\overset{\circ}{\subset}$ operator denotes that $[x]_{k+1}$ is included in the interior of $[x]_k$), then the matrices R and A are regular, and there is a unique solution x of the system $Ax = b$ with $x \in [x]_{k+1}$. We assume that $Ax = b$ is a dense square linear system and we do not consider any special structure of the elements of A.

A parallel self-verified linear equation solver proposed in [6] uses as base the algorithm 1. This algorithm describes the implementation of the enclosure methods theory explained before.

As shown in Algorithm 1, the result of this method is a vector, where each element is an interval which contains the correct result. These intervals are of high accuracy, in the sense that for most problems, when these intervals are rounded, they differs at most 1 unit in the last place from the exact result.

This paper presents an optimization of the parallel version of this algorithm presented in [6], which uses technologies as MPI communications functions [10] and C-XSC library to improve the precision [7]. Two points of optimization were focused: the parts in which mathematical accuracy is relevant and the communication cost. Our main contribution is to speed-up the performance of the previous parallel self-verified linear solver. We also point out the advantages and drawbacks of self-verified methods usage in clusters of computers.

This paper is organized as follows: next section briefly describes some implementation issues. Section 3 shows the optimization strategies for the parallel

Algorithm 1. Compute an enclosure for the solution of the square linear system $Ax = b$.

```
 1: $R \approx (A)^{-1}$ {compute an approximate inverse using LU-Decomposition algorithm}
 2: $\tilde{x} \approx R \cdot b$ {compute the approximation of the solution}
 3: $[z] \supseteq R(b - A\tilde{x})$ {compute enclosure for the residuum (without rounding error)}
 4: $[C] \supseteq (I - RA)$ {compute enclosure for the iteration matrix (without rounding
    error)}
 5: $[w] := [z]$, $k := 0$ {initialize machine interval vector}
 6: do
 7:    $[y] := [w]$
 8:    $[w] := [z] + [C][y]$
 9:    $k{+}{+}$
10: while $[w] \subseteq int[y]$ or $k > 10$
11: if $[w] \subseteq int[y]$ then
12:    $\Sigma([A], [b]) \subseteq \tilde{x} + [w]$
13: else
14:    "no verification"
15: end if
```

self-verified method. The analysis of the results obtained through those optimizations is presented in section 4. Finally, conclusions and some future works are highlighted in the last section.

2 Implementation Issues

We carried out some tests to find the most time-consuming steps of the algorithm. We found out that the steps 1 and 4 consume together more than 90% of the total processing time. The computation of an approximate inverse matrix of A (matrix R on step 1) takes more than 50% of the total time, due to the evaluation with high accuracy. The computation of the interval matrix [C] (parallel preconditioning) takes more than 40% of the total time, since matrix multiplication requires $O(n^3)$ execution time, and the other operations are mostly vector or matrix-vector operations which require at most $O(n^2)$.

We assume the coefficient matrix A to be dense, i.e. in a C-XSC program, we use a square matrix of type rmatrix, to store A and we do not consider any special structure of the elements of A. Our goal is to make a parallel version of the C-XSC algorithm that verifies the existence of a solution and computes an enclosure for the solution of the system $Ax = b$ for a square $n \times n$ matrix A with a better performance as the sequential version.

Our parallel approach involves two slightly different techniques to solve the main bottlenecks of the original algorithm. First, we used a parallel phases approach to achieve speedup in the core of the computation bottleneck: the computation of the inverse matrix. The second technique is a worker/manager approach to achieve parallel preconditioning.

The load balancing approach is made using a simple yet effective algorithm that accomplishes two main constraints. (i) the load balancing must be distributed, in such a way that each process computes its own workload eliminating communication costs due to information exchanges; (ii) It also must be fast in order to avoid long interruptions on the target computation. The algorithm representing the load balancing strategy for both parallelization is shown in algorithm 2.

Algorithm 2. Workload distribution algorithm.

```
1: if $pr \leq (N\%P)$ then
2:     $lb_i = (\frac{N}{P} \times pr) + (pr)$
3:     $ub_i = lb + \frac{N}{P} + 1$
4: else
5:     $lb_i = (\frac{N}{P} \times pr) + (N\%P)$
6:     $ub_i = lb + \frac{N}{P}$
7: end if
```

Let N be the dimension of a matrix and P be the number of processes used on computation. Also, pr represents the identification of a process (starting from 1 to P), lb_i is the lower bound and ub_i is the upper bound of the i^{th} process, *i.e.*, the first and the last row/column that a process must compute. In this algorithm, % is the remind of the integer division $\frac{N}{P}$. With this load balancing approach, it is clear that processes receive continuous blocks of rows/columns. This choice was considered the best by the authors, since it simplifies the implementation of the communication step . However, other load balancing schemes could be used without loss or gain of performance, once the computational cost to process a row/column is the same.

Readers interested in a more detailed explanation of the parallel version of the algorithm should read [6].

3 Optimization Strategies

After the analysis of the original work, two possible optimization points were detected: the use of the mathematical accuracy primitives and the use of the communication primitives.

The first optimization was related to the use of high accuracy on the computation of R (see algorithm 1 line 1). Since R is an approximation of the inverse matrix of A, which does not necessarily need exact results, it can be computed without high accuracy. The elimination of high accuracy on the computation of R does not compromise the results obtained and decreases the overall executing time (detailed results are presented in section 4).

On the other hand, the data types of the C-XSC library, which allow the self-verification and were used on the parallel approach, do not fit with the MPI library data types. In order to solve this problem, the parallel previous version used a simple, yet time consuming, solution. It is based on a pack and unpack step before sending and receiving data. In the current work, we used an interface for the MPI library that already implements the pack and unpack procedures in a low level [2]. This improvement makes the data exchange among processes faster, decreasing the communication cost of the parallel solution.

4 Results

In order to verify the benefits of these optimizations, two different experiments have been performed. The first test is related to the correctness of the result. Once we make a mathematical optimization, we need to verify that it did not change the accuracy of the result. The second experiment was performed to evaluate the speed-up improvement brought by the proposed optimizations.

The results presented in this section were obtained on the parallel environment ALiCEnext (Advanced Linux Cluster Engine, next generation) installed at the University of Wuppertal (Germany). This cluster is composed of 1024 1.8 GHz AMD Opteron processors (64 bit architecture) on 512 nodes connected by Gigabit Ethernet. ALiCEnext processors employ Linux as operating system and MPI as communication interface for parallel communication.

Performance analysis of the optimized parallel solver were carried out varying the order of input matrix A in three different grains: 500×500, $1,000 \times 1,000$ and $2,500 \times 2,500$. Matrix A and vector b were generated by two distinct forms: using the Boothroyd/Dekker formula [3] and random numbers, to evaluate the correctness and the performance of the new implementation respectively. Boothroyd/Dekker matrices have the following structure:

$$A_{ij} = \binom{n+i-1}{i-1} \times \binom{n-1}{n-j} \times \tfrac{n}{i+j-1}, \forall i,j = 1..N$$

$$b = 1, 2, 3, ..., N.$$

Figure 1(a) shows the comparison between the old parallel version (without R optimization) and the new parallel implementation (with R optimization) both using matrix A as $1,000 \times 1,000$.

Figure 1 shows two graphs about the the performance of the new parallel implementation: (i) figure 1(a) presents a comparison between the results obtained by the old and the new (with R optimization) parallel implementations both using as input matrix A size 1,000 x 1,000; (ii) figure 1(b) introduces experiments varying the system of equations size.

In figure 1(a), it can be seen that the parallel version with optimization reached lower execution times than the parallel version without optimization. For dimension 1,000, the execution time with 2 processors is, however, a bit large: around 350 seconds. Based on this information, we can project that the execution time for dimension 10,000 would be around 4 days ($350 * 10^3$). Therefore we did not test larger dimensions.

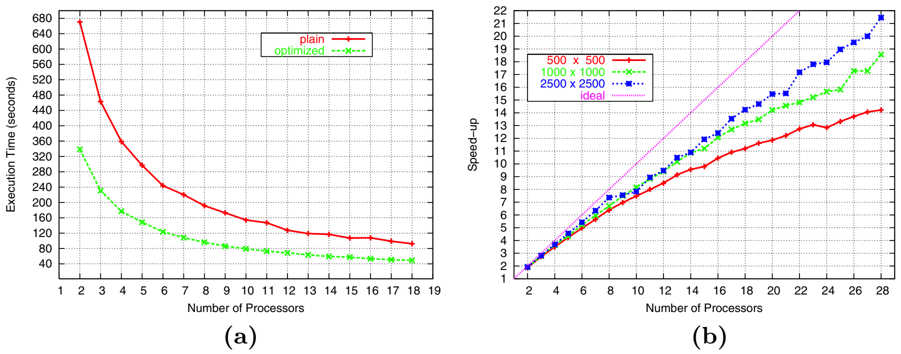


Fig. 1. Experimental Results

Looking at figure 1(b) it is possible to remark that, using the matrices orders mentioned above, the proposed solution presents a good scalability. It is also important to highlight that for the largest input matrix, the speed-up achieved was around 21 for 28 processors which is a representative result for cluster platforms.

Many executions were tackled using the Boothroyd/Dekker formula with a matrix 10×10 with different number of processes $(1..P)$. The tests generated by the Boothroyd/Dekker formula presented the same accuracy on both versions (sequential and parallel) and indicate that the parallelization did not modify the accuracy of the results.

Finally, we only could carry out experiments up to 28 processors due to cluster nodes availability for our experiments. For all experiments, this number of processors was not enough to achieve the inflexion point in the speed-up curves, indicating that more processors could be used to improve even more the speed-ups.

5 Conclusion

An optimized parallel implementation for the self-verified method for solving dense linear systems of equations was discussed in this paper. The main objective is to allow the use of this new algorithm in real life situations, which deal with large matrices. The self-verification provides reliability but also decreases the performance. Therefore, we optimized an implementation which was parallellized aiming at a better performance. Two optimizations were carried out: the use of the mathematical accuracy primitives (replacement of the exact scalar product in the computation of R) and the use of the communication primitives.

Two different input matrices with three different sizes were used in several experiments aiming to evaluate the speed-up achieved in this new implementation. All three granularities increased the performance with significant gain. The correctness tests also point out a good implementation, where the results were obtained without any loss of accuracy. Thus, the gains provided by the self-verified computation could be kept with a significant decrease in the execution time through its parallelization. The load balancing strategy seems to be a good choice according to the results found in all tested input cases.

Our main contribution is to increase the use of self-verified computation trough its parallelization, once without parallel techniques it becomes the bottleneck of an application. We can notice rather interesting speed-ups for the self-verified computation, reinforcing the statement related to the good parallelization choices. However, more experiments with a higher amount of processors and different matrices would be interesting to verify and validate this implementation to a larger range of input matrices.

Finally, it is the opinion of the authors that the results obtained are promising and the implementation allowed a quite good understanding of the problem, leading to new directions for further investigations. Also, other possible optimizations in the use of verified operations must be mathematically investigated in order to guarantee that no unnecessary time consuming operations are being executed.

References

1. Bohlender, G.: What Do We Need Beyond IEEE Arithmetic? In: Computer Arithmetic and Self-validating Numerical Methods, pp. 1–32. Academic Press, San Diego (1990)
2. Grimmer, M.: An MPI Extension for the Use of C-XSC in Parallel Environments. Technical report, Wissenschaftliches Rechnen / Softwaretechnologie, Wuppertal, DE (2005), `http://www.math.uni-wuppertal.de/wrswt/literatur.html`
3. Hammer, R., Ratz, D., Kulisch, U., Hocks, M.: C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems. Springer, Heidelberg (1997)
4. INTLAB. INTerval LABoratory, `http://www.ti3.tu-harburg.de/~rump/intlab/`
5. Klatte, R., Kulisch, U., Lawo, C., Rauch, R., Wiethoff, A.: C-XSC- A C++ Class Library for Extended Scientific Computing. Springer, Heidelberg (1993)
6. Kolberg, M., Baldo, L., Velho, P., Webber, T., Fernandes, L.F., Fernandes, P., Claudio, D.: Parallel Selfverified Method for Solving Linear Systems. In: 7^{th} VECPAR - International Meeting on High Performance Computing for Computational Science (to appear, 2006)
7. Ogita, T., Rump, S.M., Oishi, S.: Accurate Sum and Dot Product with Applications. In: 2004 IEEE International Symposium on Computer Aided Control Systems Design, Taipei, Taiwan, September 2004. LNCS, pp. 152–155. IEEE Press, Los Alamitos (2004)
8. Ogita, T., Rump, S.M., Oishi, S.: Accurate Sum and Dot Product. SIAM Journal on Scientific Computing 26(6), 1955–1988 (2005)
9. Rump, S.M.: Kleine Fehlerschranken bei Matrixproblemen. PhD thesis, University of Karlsruhe, Germany (1980)
10. Snir, M., Otto, S., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge, MA (1996)

A Parallel Block Iterative Method for Interactive Contacting Rigid Multibody Simulations on Multicore PCs

Claude Lacoursière

HPC2N/VRlab and Department of Computing Science,
Umeå University, SE-901 87 Umeå, Sweden
claude@cs.umu.se
http://www.cs.umu.se/~claude

Abstract. A hybrid, asynchronous, block parallel method to approximately solve complementarity problems (CPs) in real-time on multicore CPUs is described. These problems arise from interactive real-time simulations of systems of constrained, contacting rigid bodies, which are useful in virtual operator training systems for instance. A graph analysis phase identifies components which are weakly coupled using simple heuristics. Each component is then solved in parallel using either a block principal pivot or a projected block Gauss-Seidel method running in separate threads. Couplings which generate forces between the subsystems are handled iteratively using a Gauss-Seidel process which communicates updates between the interacting subsystems asynchronously. Preliminary results show that this approach delivers good performance while keeping overhead small.

1 Introduction

Interactive real-time simulation of systems of constrained rigid bodies and particles is an essential component of several commercially relevant applications, such as virtual environment heavy machinery operator training systems. Numerical integration of multibody systems needed for these simulations involves solution of systems of equations with several hundred variables. The best known strategy to simulate frictional contacts—ubiquitous in real-life systems—requires solving mixed linear complementarity problems (MLCP)s as described below. In the real-time interactive simulation context, solutions of these MLCPs must be delivered in less than 10ms to keep screen refresh rates above 60 frames per second. Direct solution methods for MLCP are slow for large systems because they rely on single pivot operations [3] for which sparsity exploitation is difficult. For this reason, iterative methods based on pairwise interatcions—which are fast but much less accurate—have dominated the 3D graphics and interactive games literature [8] [4]. Parallelization has also been generally overlooked.

The present proposes a parallel hybrid solution method which uses either a direct or an iterative solver on blocks of strongly coupled equations and Gauss-Seidel (GS) iterations to approximate the coupling forces between those blocks.

This scheme is easily implemented on multicore PCs using simple threads and synchronization. The test application consists of a wheeled vehicle toting a long tether. Such vehicles are used for various remote operations from de-mining to underwater inspections. Virtual training systems have proved useful and commercially relevant. The proposed hybrid scheme delivers good performance on this example.

The rest of the article is organized as follows. Section 2 briefly introduces the problem being solved in the context of interactive simulations. Section 3 presents the splitting strategy as well as the iterative algorithm which computes approximate coupling forces between subsystems. Section 4 covers heuristics used to split a physical system into interacting blocks, and Section 5 describes the asynchronous execution framework implementing the GS scheme to compute the approximate coupling forces. Results and conclusions are presented in Section 6 in which a simple system is analyzed to provide preliminary data. The present report is part of work in progress.

2 Problem Formulation

Consider a general mechanical system with n-dimensional generalized velocity vector $v \in \mathbb{R}^n$. The system has a square, real, positive definite and block diagonal $n \times n$ mass matrix M, with easily computed inverse $U = M^{-1}$, and is subject to a variety of constraints which have Jacobian matrix G of size $m \times n$. The essential computation of a large family of stepping schemes involves the solution of a *mixed linear complementarity problem* (MLCP):

$$\begin{gathered} Sy + q = w = w^{(+)} - w^{(-)} \\ 0 \leq y - l \perp w^{(+)} \geq 0, \qquad 0 \leq u - y \perp w^{(-)} \geq 0, \end{gathered} \tag{1}$$

at each time step. The real, square $m \times m$ matrix S has the form $S = GUG^T$. In addition, $q \in \mathbb{R}^m$ is a real vector, $w^{(+)}, w^{(-)}$ are the real positive and negative components of the residual vector $w \in \mathbb{R}^m$, respectively. The lower and upper bound vectors $l, u \in \bar{\mathbb{R}}^m$, with $\bar{\mathbb{R}} = \mathbb{R} + \{\pm\infty\}$, are extended real vectors. The inequality and orthogonality signs in (1) are understood componentwise so that $a, b \in \mathbb{R}^m$, $a \geq b$ means $a_i \geq b_i, i \in \{1, 2, \ldots, m\}$, $a \leq b$ means $a_i \leq b_i, i \in \{1, 2, \ldots, m\}$, and $a \perp b$ means $a_i b_i = 0$ for all $i \in \{1, 2, \ldots, m\}$, whenever $a, b \geq 0$. This problem of solving (1) is abbreviated as MLCP(S, q, l, u). The solution of this MLCP is the vector $y \in \mathbb{R}^m, y = \text{SOL}(\text{MLCP}(S, q, l, u))$, which is unique as long as S is symmetric and positive definite [2]. The solution vector y produces the constraint force vector $G^T y$ from which the updated velocities and coordinates can be computed directly given a time-integration strategy.

For jointed mechanical systems, Jacobians have the simple block structure

$$G^T = \left[G^{(1)^T} \; G^{(2)^T} \ldots G^{(n_c)^T} \right], \tag{2}$$

where each block row $G^{(i)}$ is of size $n_i \times n$, with $\sum_i n_i = m$. The integer n_c is the number of constraints. Each block row $G^{(i)}$ contains only a few nonzero column blocks, $c_i(1), c_i(2), \ldots, c_i(p_i)$ where p_i is usually 1 or 2, as is the case for common joints and contact constraints. In this format, each column block usually corresponds to a single physical body. This produces the first level of partitioning of the system where the velocity vector v is decomposed into blocks $v^{(i)}, i = 1, 2, \ldots, n_b$, where n_b is the number of *generalized bodies* in the system, which may or may not have the same dimensionality. Each block here corresponds to a given physical body.

The connectivity structure at this level of partitioning is a bipartite graph, where nodes are either physical bodies or constraints. Any common graph traversal algorithm can identify the connected components. Processing these is an embarrassingly parallel problem.

The solution of MLCP (1) for each connected component can be performed using, e.g., a block pivot [3] (BP) method, which is equivalent to applying the Newton-Raphson method on the nonsmooth formulation, $\text{mid}((y - l)_i, (Sy + q)_i, (y - u)_i) = 0$, where $\text{mid}(a, b, c)$ is the midpoint function [6], defined component-wise as:

$$\text{mid}(a, b, c) = a + c - \sqrt{(a - b)^2} + \sqrt{(b - c)^2}. \tag{3}$$

This is called a direct solver since it can produce the exact solution with appropriate smoothing strategy [6]. Other possibilities include iterative solvers such as projected Gauss-Seidel (PGS) methods or projected Successive Overrelaxation (PSOR) [3] methods which are extremely simple to implement and quickly yield answers of moderate accuracy. The BP, PGS and PSOR can all be warm started from an approximate solution and they typically decrease the infeasibility at each stage. Warm starting is not possible for direct pivoting methods such as the Lemke, Cottle-Dantzig, or Keller algorithms [3]. In addition, direct pivoting methods do not reduce infeasibility monotonically and cannot be stopped early to yield an approximate solution. Both features are key element for splitting methods in which slightly different MLCPs (1) with the same matrix S but slightly different vectors q, l and u, are solved numerous times.

The BP method without smoothing performs well on average but it can cycle over a set of candidates, none of which exactly solve the problem, when the problem is nearly degenerate or infeasible which is common in practice due to the use of a vertex based definition of contacts, large integration step, and *a posteriori* collision detection [5]. Degeneracy problems are not considered further here.

3 Splitting Strategy and MLCP Iterative Method

To parallelize further, we consider one of the connected components described in the previous section and assume a partitioning of the bodies into two groups for example, labeled as 1, 2, having block velocity vectors and block inverse mass matrix $v^{(i)}, U_{ii}, i = 1, 2$, respectively. Any given constraint Jacobian can then be split into blocks: $G^{(i)} = [G_1^{(i)}, G_2^{(i)}]$, where block $G_j^{(i)}$ acts only on the block coordinates v_j, $j = 1, 2$. The natural separation of constraints produces three groups, namely, those acting only on the first group, those acting only on the second group, and those acting on both groups of bodies

$$G = \begin{bmatrix} G_{11} & 0 \\ 0 & G_{22} \\ G_{31} & G_{32} \end{bmatrix}. \tag{4}$$

After splitting the mass matrix correspondingly, with $U = \mathrm{diag}(U_{ii})$, matrix S in (1) has the form

$$S = \begin{bmatrix} S_{11} & 0 & S_{31}^T \\ 0 & S_{22} & S_{32}^T \\ S_{31} & S_{32} & S_{33} \end{bmatrix}, \tag{5}$$

where $S_{3i} = G_{3i}U_{ii}G_{ii}^T, i = 1, 2$, and $S_{33} = \sum_{i=1}^{2} G_{3i}U_{ii}G_{3i}^T$. Obviously, if the constraints in group 3 are chosen to be mass orthogonal to those in groups 1 and 2, there are no off-diagonal terms. Minimizing these coupling terms is likely to improve the convergence rate and is subject of future work.

To formulate an iterative method for solving this problem, we need a definition for the infeasibility error $e \in \mathbb{R}^m$. Consider a candidate solution vector x and residual $w = Sx + q$. For each component i, first check that it is within range, i.e., $l_i < x_i < u_i$. If so, the residual should vanish so we set $e_i = w_i$. If the bounds are finite and either $x_i = l_i$ or $x_i = u_i$, we use $e_i = \min(x_i - l_i, w_i)$ or $e_i = \min(u_i - x_i, w_i)$, respectively. If x_i is out of bound, we use the midpoint function $e_i = \max(x_i - u_i, \min(w_i, z_i - l_i))$.

A projected block GS solution of MLCP(S, q, l, u) is formulated in Algorithm 1.

Algorithm 1. Serial block projected Gauss-Seidel for solving partitioned MLCP(S, q, l, u).

```
1: Set $\nu \leftarrow 1$, choose $\tau > 0$ and $\nu_m > 0$
2: Initialize $y_1^{(1)}, y_2^{(1)}, y_3^{(1)}$.
3: **repeat**
4:     Solve: $y_1^{(\nu+1)} \leftarrow \mathrm{SOL}(\mathrm{MLCP}(S_{11}, q_1 + S_{31}^T y_3^{(\nu)}, l_1, u_1))$
5:     Solve: $y_2^{(\nu+1)} \leftarrow \mathrm{SOL}(\mathrm{MLCP}(S_{22}, q_2 + S_{32}^T y_3^{(\nu)}, l_2, u_2)))$
6:     Solve: $y_3^{(\nu+1)} \leftarrow \mathrm{SOL}(\mathrm{MLCP}(S_{33}, q_3 + S_{31} y_1^{(\nu+1)} + S_{32} y_2^{(\nu+1)}, l_3, u_3))$
7:     $\nu \leftarrow \nu + 1$
8:     Compute infeasibility $e^{(\nu)}$
9: **until** $\|e^{(\nu)}\| < \tau$ OR $\nu > \nu_m$
```

This GS process reduces the infeasibility monotonically for a positive definite matrix, even in the case of MLCP [3] but the convergence is linear at best and can be stationary in case the matrix S is degenerate or nearly so. This algorithm can be parallelized with or without synchronization (see [1] for descriptions of chaotic asynchronous schemes) by using threads so that solutions to each of the subsystems are computed in parallel. Overhead can be kept small by only having mutex and condition variables to control block reading from and block writing to the shared data, namely, the kinematic variables in subsystems $1, 2$.

4 Heuristics for Splitting of Connected Components

Splitting of connected components into primary groups of bodies and constraints and secondary groups of constraints should be done so as to minimize the coupling between the primary groups. The theoretical and algorithmic treatment of this minimization problem merits further investigation. Only a simple heuristic has been implemented which is now described.

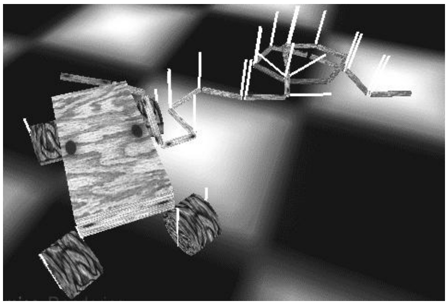
Each body is first labeled with a *grouping* identifier. It is expected that bodies with the same grouping index are interacting strongly with each other, at least when they are collected into the same connected component. This grouping index also carries a priority so that bodies in grouping g_1 are expected to be more important than those in grouping g_2 if $g_1 < g_2$.

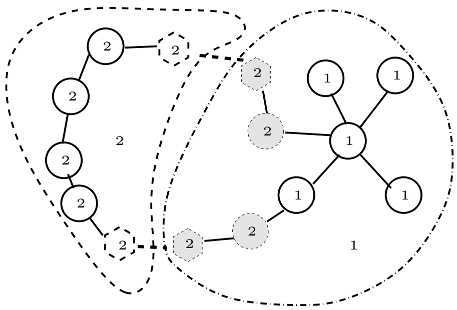
The heuristic is to collect all connected components belonging to the same grouping id, g, as well as all bodies and constraints which are connected to bodies in g, have a grouping id higher than g, and are connected within depth k of a body with grouping index g. An illustration of this process is found in Fig. 1(b).

Connected components are constructed by performing a breadth first search in the rigid body system. After sorting the bodies in increasing order of grouping index, the first untouched body in that list is selected and from it, connected bodies are found by first listing attached constraints and then listing the bodies found for each constraint. If one of these bodies has a different grouping index, breadth first search is re-initiated at that body, going down to depth k. The bodies found during this search necessarily have a higher grouping index than current. Once this search terminates at a body with depth k, all the uncounted constraints attached to this body are added to the secondary partition pool containing all secondary constraints. This corresponds to the G_{3i} blocks in the previous section.

When this search terminates, we have a forest of primary partitions and a secondary partition with all coupling constraints. This secondary partition could be refined further but that is left for future work.

Consider the example of a tethered vehicle illustrated in Fig. 1. First assign grouping index one for the vehicle and two for the tether, and set the maximum depth to two. The partitioner will collect the main vehicle and the first two cable segments into the first partition. The imported bodies are filled in circles and octagons in Fig. 1(b). If the tether does not otherwise touch the vehicle or its wheels, the tether is in a primary partition by itself and the secondary partition contains a single constraint. If the vehicle rolls over the cable though, the first primary partition contains the main vehicle, the first two segments of the tether, and at least two more cable segments near each contact, symmetrically distributed about the contact location. This is illustrated in Fig. 1(b) where the end of the cable touches one of the vehicle parts. This introduces two more imported bodies in primary partition 1 illustrated with the gray bodies. This leads


(a) A still from a simulation of a tethered vehicle. Vertical white lines indicate normals at contact points.



(b) A sample graph and its partitioning.

Fig. 1. An example application involving a tethered vehicle. A still from the simulation is shown in (a) and a simplified graph analysis is shown in (b).

to four bodies, drawn with octagons instead of circles, which are influenced both by their primary partition and the forces computed by the secondary partition which contains the dashed constraints. The rest of the tether is then segmented and cut into two or more primary partitions, and the secondary partition contains two or more coupling constraints.

5 Asynchronous Execution

The shared variables in the system are the constraint forces applied on the rigid bodies. Indeed, some of the bodies are listed both in a primary partition (once only) and in secondary partitions (potentially several times).

When working on any given partition, a solver must first read the current net forces from the global data, solve for updated forces, and add the updated contributions back to the global data. To do this efficiently, the rigid body force data is segmented along the primary partitions. One mutex and one global time stamp is created for each of these, the stamp being incremented each time the data is updated by one of the parallel solver. Secondary partitions must acquire locks for all the primary partitions they are connected with but primary ones only acquire one lock.

In order to know if a solver should compute a new solution, local time stamps are kept for all the data sets used in a given partition. Reference counting the global time stamps allows to know if a data set is coupled to others.

Partitions and associated local data are packaged as a work unit and put in a priority queue so that partitions with lower iteration count have higher priority. A thread pool, designed much as in Ref [7], Ch. 3, is started during program initialization and each of the threads waits after a semaphore on which a thread manager broadcast when there is fresh data. Threads report upon completion to the manager via another semaphore by signaling.

The thread manager first configures all the data for a new solve process, fills the priority queue with the work units, and broadcast the workers in the thread pool. Each work thread then picks up a work item from the top of the priority queue and executes the solve operations on it. When this is done, a check is made to see whether or not that work unit needs more computations done to it and it is requeued accordingly. The work thread then picks up another work item from the top of the queue or goes idling if that is empty. Meanwhile, the thread manager listens for signals from the work threads and marshals them back to work as long as processing is not completed.

For each work unit, the program first checks if it is coupled to others and whether there is fresh data to consider by measuring the difference between local and global time stamps. If there is fresh data, locks are acquired, data is cached locally, locks are then released, and computations are performed. When the results are ready, the global data set is locked while the local data is added to it. Time stamps are then incremented and all locks are released. Local iteration count is then increased and the work unit is requeued. If there are no couplings when a work unit corresponding to a primary partition is processed, the final integration stage is performed using the available global data.

Work stops when a maximum iteration count is reached though a strategy based on global error would presumably be more accurate and potentially slower. The latter is future work.

6 Results and Conclusions

An implementation of Algorithm (1) was realized using Vortex, a commercial rigid multibody dynamics simulation library not originally designed to handle couplings between partitions (see `http://www.cm-labs.com`). The reference platform was Linux and the standard pthreads library. As a preliminary test, a simple example of a tethered car was constructed. This consists of a simplified vehicle with a chassis and four wheels, dragging a long cable made of rigid body sections attached with ball joints. A still of that simulation is shown in Fig. 1(a). The graphics rendering is minimal here because the main goal was to test the performance of the splitting scheme but a similar setup is used in a commercial trainer application for remotely operated tethered vehicles. Such an application demands hundreds of rigid bodies for the tether simulation because of the combination of long cable length and tortuous paths in typical scenarios.

The natural partitioning here is to explicitly assign the rigid bodies composing the vehicle in group 1 and those composing the tether in group 2. Further partitioning can be realized by segmenting the cable so that each segment contains 10 rigid bodies, say. With this strategy, we can then use a direct BP or iterative PGS solver on each primary partition, making the overall algorithm hybrid. Since the main vehicle becomes unstable when using the PGS solver, the direct solver is always used on that group. Therefore, without splitting, it is necessary to use a direct solver for the entire system and this gives the reference baseline for performance.

Simulations were run on an Athlon 64 X2 Dual Core Processor 3800+ with 512MB secondary cache, 2.0 GHz clock speed, and 2GB of RAM. The vehicle was set on a circular path so that it would eventually roll over the tether. Simulations ran for 10000 steps which corresponds to nearly 3 minutes of real time. The time step was set to $h = 1/60 = 0.01667$ which produces real-time for refresh rates of 60 frames per second, with one integration step per frame. For cases where splitting is used, we performed five (5) coupling iterations based on visual inspection of the simulation. Global error estimate was not computed and not used as a stopping criterion for these runs. This is left for future work. When the iterative PGS was used, the stopping criteria was an absolute local error of less than 10^{-3} or an iteration count exceeding 20—parameters selected by trial and error. The direct solver used here is a BP method [3] with protection against cycling and restart and the iterative solver is a PGS method. Both are part of the Vortex toolkits.

Results are collected in Table 1 where the direct solver was used on each subgroup, and Table 2 where the PGS solver was used for cable segments and the BP solver for the main vehicle. In both these tables, the first column contains the number of rigid body segment in the tether of the example illustrated in Fig. 1(a). The second column is the number of parallel threads. Only cases with 1 or 4 threads were constructed. The third column indicates how many different groups where used in the simulation. This number is two (2) if we only split the tether in a group and the vehicle in the other but we also show results of segmenting the cable into many more groups using segmenting. The next column is the percentage CPU utilization which goes near 100% on single CPU and remains less than 160% when both cores are used with four (4) parallel threads. This is because other sections of the software are not multithreaded. The fifth column is the average time used for solving the MLCP, isolated from the other computations and the sixth column is the speedup of the solver time as compared to the single thread, single group setting. The last two column contain the total time taken by the simulation for one frame and the speedup, also with respect to the single group, single thread configuration.

The first striking observation is that splitting alone produces a speedup of a factor of 10 to 30. This is explained by the fact that we are now only computing an approximation of the MLCP (1). In other words, time is saved by decreasing the accuracy of the result. That would not be possible using a direct method on the full problem for instance. Using four separate threads of execution and enough groupings to saturate the thread pool, the speedup is up to 40 for small systems, and 120 for large ones. This speedup is sufficient to make the application truly interactive, both in speed and stability.

Since using the iterative method on the entire system produces unsatisfactory results for the vehicle dynamics, the baseline in Table 2 is computed with two groups, namely, one for the tether and one for the vehicle, both in the same thread. A speedup of 1.5 is observed on average when using four (4) parallel threads of execution as opposed to just one. This is close to what one would expect since other components of the software take much time.

Table 1. Timing results using a direct solver for each group. The baseline refers to a single thread simulation with a simple group containing all bodies and constraints.

Length	Threads	Groups	% CPU	Solver Time	Solver Speedup	Total Time	Speedup
60	1	1	99	98.3	98.9	1.0	1.0
	1	2	99	7.8	8.2	12.6	12.1
	1	7	100	6.7	7.0	14.7	14.0
	4	7	137	2.1	2.3	46.2	42.4
80	1	1	99	67.7	68.2	1.0	1.0
	1	2	100	6.8	7.1	9.9	9.6
	1	9	100	5.7	6.0	11.9	11.3
	4	9	147	2.9	3.2	23.4	21.4
100	1	1	99	186.9	187.3	1.0	1.0
	1	2	99	7.6	8.0	24.6	23.4
	1	11	99	8.7	9.1	21.5	20.6
	4	11	157	6.9	7.4	26.9	25.3
120	1	1	99	393.2	393.8	1.0	1.0
	1	2	99	11.7	12.2	33.7	32.2
	1	13	99	11.7	12.2	33.7	32.2
	4	13	151	3.2	4.7	121.3	83.2
140	1	1	97	513.9	514.5	1.0	1.0
	1	15	100	11.2	11.8	46.1	43.8
	4	15	152	3.8	4.3	134.7	120.5

Table 2. Timing results using an iterative solver for each group. The baseline here refers to a single thread simulation with two groups, the cable being processed with the PGS and the vehicle with BP solver respectively.

Length	Threads	Groups	% CPU	Solver Time	Solver Speedup	Total Time	Speedup
60	1	2	99	5.6	5.8	1.0	1.0
	4	7	149	3.5	3.7	1.6	1.5
80	1	2	100	7.8	8.2	1.0	1.0
	4	9	158	5.2	5.5	1.5	1.5
100	1	2	99	7.9	8.3	1.0	1.0
	4	11	160	5.7	6.2	1.4	1.3
120	1	2	99	9.3	9.8	1.0	1.0
	4	13	158	5.2	5.6	1.8	1.7
140	1	2	100	10.6	11.0	1.0	1.0
	4	15	163	8.7	9.3	1.2	1.2

The conclusion we draw from the data is that combining problem splitting and a direct solver is attractive in comparison to the far less accurate GS method which, unsurprisingly, does not benefit significantly from the parallelization. Further work is necessary to assess the error margin of this strategy and is work in progress. Also part of future work is the investigation of more sophisticated partitioning schemes based on spectral analysis as well as conjugate gradient-type methods for computing inter-partition couplings.

Acknowledgments

Constructive comments by Bo Kågström, Daniel Kressner and Kenneth Bodin on earlier versions of this paper, as well as cooperation from CMLabs is gratefully acknowledged. This research was conducted using the resources of High Performance Computing Center North (HPC2N), and supported in part by the "Objective 1 Norra Norrlands" EU grant VISTA awarded to VRlab at Umeå University, and by the *Swedish Foundation for Strategic Research* under the frame program grant SSF-A3 02:128.

References

1. Bai, Z., Huang, Y.: A class of asynchronous parallel multisplitting relaxation methods for the large sparse linear complementarity problems. Journal of Computational Mathematics 21(6), 773–790 (2003)
2. Cottle, R.W., Pang, J.-S., Stone, R.E.: The Linear Complementarity Problem. In: Computer Science and Scientific Computing, Academic Press, New York (1992)
3. Júdice, J.J.: Algorithms for linear complementarity problems. In: Spedicato, E. (ed.) Algorithms for Continuous Optimization. NATO ASI Series C, Mathematical and Physical Sciences, Advanced Study Institute, vol. 434, pp. 435–475. Kluwer Academic Publishers, Dordrecht (1994)
4. Kaufman, D.M., Edmunds, T., Pai, D.K.: Fast frictional dynamics for rigid bodies. ACM Trans. Graph. 24(3), 946–956 (2005)
5. Lacoursière, C.: Splitting methods for dry frictional contact problems in rigid multibody systems: Preliminary performance results. In: Ollila, M. (ed.) Conference Proceedings from SIGRAD2003, November 20–21, 2003, Umeå Unversity, Umeå, Sweden, pp. 11–16. SIGRAD (2003)
6. Li, D., Fukushima, M.: Smoothing Newton and quasi-Newton methods for mixed complementarity problems. Comp. Opt. and Appl. 17, 203–230 (2000)
7. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming. O'Reilly and Associates, Inc., Sebastopol, CA 95472 (1996)
8. Weinstein, R., Teran, J., Fedkiw, R.: Dynamic simulation of articulated rigid bodies with contact and collision. IEEE TVCG 12(3), 365–374 (2006)

PyTrilinos: High-Performance Distributed-Memory Solvers for Python

Marzio Sala[1], William F. Spotz[2], and Michael A. Heroux[2]

[1] Department of Computer Science, ETH Zurich, CH-8092 Zurich
marzio@inf.ethz.ch
[2] Sandia National Laboratories, PO Box 5800 MS 1110,
Albuquerque, NM 87185-1110*

Abstract. PyTrilinos is a collection of Python modules targeting serial and parallel sparse linear algebra, direct and iterative linear solution techniques, domain decomposition and multilevel preconditioners, nonlinear solvers and continuation algorithms. Also included are a variety of related utility functions and classes, including distributed I/O, coloring algorithms and matrix generation. PyTrilinos vector objects are integrated with the popular NumPy module, gathering together a variety of high-level distributed computing operations with serial vector operations.

PyTrilinos uses a hybrid development approach, with a front-end in Python, and a back-end, computational engine in compiled libraries. As such, PyTrilinos makes it easy to take advantage of both the flexibility and ease of use of Python, and the efficiency of the underlying C++, C and FORTRAN numerical kernels. The presented numerical results show that, for many important problem classes, the overhead required by the Python interpreter is negligible.

1 Introduction

The choice of the programming language for the development of large-scale, high-performance numerical algorithms is often a thorny issue. It is difficult for a single programming language to simultaneously support ease-of-use, rapid development, and optimized executables. Indeed, the goals of efficiency and flexibility often conflict. The key observation to approaching this problem is that the time-critical portion of code requiring a compiled language is typically a small set of self-contained functions or classes. Therefore, one can adopt an interpreted (and possibly interactive) language, without a big performance degradation, provided there is a robust interface between the interpreted and compiled code.

This article describes a collection of numerical linear algebra and solver libraries, called PyTrilinos, built on top of the Trilinos [3] project. The main

* ASCI program and the DOE Office of Science MICS program at Sandia National Laboratory. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

goal of PyTrilinos is to port to a scripting language most of the Trilinos capabilities, for faster development of novel numerical algorithms.

Among the available scripting languages, we decided to adopt Python. Python is an interpreted, interactive, object-oriented programming language, which combines remarkable power with very clean syntax (it is often observed that well-written Python code reads like pseudo code). Perhaps most importantly, it can be easily extended by using a variety of open source tools such as SWIG [1], f2py or pyfort to create wrappers to modules written in C, C++ or FORTRAN for all performance critical tasks.

PyTrilinos adds significant power to the interactive Python session by providing high-level commands and classes for the creation and usage of serial and distributed, dense and sparse linear algebra objects. Using PyTrilinos, an interactive Python session becomes a powerful data-processing and system-prototyping environment that can be used to test, validate, use and extend serial and parallel numerical algorithms. In our opinion, Python naturally complements languages like C, C++ and FORTRAN as opposed to competing with them. Similarly, PyTrilinos complements Trilinos by offering interactive and rapid development.

This paper is organized as follows. Section 2 reports more details on the choice of Python and the interface generator tool, SWIG [1]. Section 3 gives an overview of the structure of PyTrilinos, which closely reflects the structure of the Trilinos framework. The organization of PyTrilinos is briefly outlined in Section 4, while Section 5 describes the strategies adopted in parallel environments. Comparisons with MATLAB and Trilinos are reported in Section 6. Finally, Section 7 draws the conclusions.

2 Why Python and SWIG

Python has emerged as an excellent choice for scientific computing because of its simple syntax, ease of use, object-oriented support and elegant multi-dimensional array arithmetic. Its interpreted evaluation allows it to serve as both the development language and the command line environment in which to explore data. Python also excels as a “glue” language between a large and diverse collection of software packages—a common need in the scientific arena.

The Simple Wrapper and Interface Generator (SWIG) [1] is a utility that facilitates access to C and C++ code from Python. SWIG will automatically generate complete Python interfaces (and other languages) for existing C and C++ code. It also supports multiple inheritance and flexible extensions of the generated Python interfaces. Using these features, we can construct Python classes that derive from two or more disjoint classes and we can provide custom methods in the Python interface that were not part of the original C++ interface.

Python combines broad capabilities with very clean syntax. It has modules, namespaces, classes, exceptions, high-level dynamic data types, automatic memory management that frees the user from most hassles of memory allocation, and much more. Python also has some features that make it possible to write large programs, even though it lacks most forms of compile-time checking: a program can be constructed out of modules, each of which defines its own namespace. Exception handling makes it possible to catch errors where required without cluttering the code with error checking.

Python's development cycle is typically much shorter than that of traditional tools. In Python, there are no compile or link steps—Python programs simply import modules at runtime and use the objects they contain. Because of this, Python programs run immediately after changes are made. Python integration tools make it usable in hybrid, multi-component applications. As a consequence, systems can simultaneously utilize the strengths of Python for rapid development, and of traditional languages such as C for efficient execution. This flexibility is crucial in realistic development environments.

3 Multilevel Organization of PyTrilinos

PyTrilinos is designed as a modular multilevel framework, and it takes advantage of several programming languages at different levels. The key component is **Trilinos** [3], a set of numerical solver packages in active development at Sandia National Laboratories that allows high-performance scalable linear algebra operations for large systems of equations. The source code of the current Trilinos public release accounts for about 300,000 code lines, divided in about 67,000 code lines for distributed linear algebra objects and utilities, 20,000 code lines for direct solvers and interfaces to third-party direct solvers, 128,000 code lines for multilevel preconditioners, and 76,000 code lines for other algebraic preconditioners and Krylov accelerators. This count does not include BLAS, LAPACK, MPI, and several other libraries used by Trilinos. This explains why a "pure" Python approach is of no interest; instead, we generate interfaces between the language in which most of Trilinos packages are written, C++, and Python.

Although C++/Python interfaces can be generated by hand, it is more convenient to adopt code generators to automate the process. We have adopted **SWIG**, the Simplified Wrapper and Interface Generator, which is a preprocessor that turns ANSI C/C++ declarations into scripting language interfaces, and produces a fully working Python extension module.

It is important to note that PyTrilinos is well-connected to other scientific projects developed within the Python community. Indeed, PyTrilinos vectors inherit from the **NumPy** vector class. NumPy is a well-established Python module to handle multi-dimensional arrays including vectors and matrices. A large number of scientific packages and tools have been written in or wrapped for Python that utilize NumPy for representing fundamental linear algebra objects. By integrating with NumPy, PyTrilinos also integrates with this sizeable collection of packages. It also adopts **Distutils**, a Python module with utilities aimed at the portable distribution of both pure Python modules and compiled extension modules. Distutils has been a part of the standard Python distribution since Python version 2.2.

4 PyTrilinos Organization

PyTrilinos reflects the Trilinos organization by presenting a series of *modules*, each of which wraps a given Trilinos *package*, where a package is an integral unit usually developed by a small team of experts in a particular area. Trilinos packages that support namespaces have a Python submodule for each namespace. Algorithmic capabilities are defined within independent packages; packages can easily interoperate since generic adaptors are available (in the form of pure virtual classes) to define distributed vectors and matrices. The modules currently included are briefly described below.

The first and most important PyTrilinos module is **Epetra** [2], a collection of concrete classes to support the construction and use of vectors, sparse distributed graphs, and dense and distributed sparse matrices. It provides serial, parallel and distributed memory linear algebra objects. Epetra supports double-precision floating point data only, and uses BLAS and LAPACK where possible, and as a result has good performance characteristics. All the other PyTrilinos modules depend on Epetra.

EpetraExt offers a variety of extension capabilities to the Epetra package, such as input/output and coloring algorithms. The I/O capabilities make it possible to read and write generic Epetra objects (like maps, matrices and vectors) or import and export data from and to other formats, such as MATLAB, the Harwell/Boeing or Matrix Market format, therefore accessing a large variety of well-recognized test cases for dense and sparse linear algebra.

Galeri allows the creation of several matrices, like the MATLAB's `gallery` function, and it can be useful for examples and testing.

Amesos [4] contains a set of clear and consistent interfaces to the following third-party serial and parallel sparse direct solvers: UMFPACK, PARDISO, TAUCS, SuperLU and SuperLU_DIST, DSCPACK, MUMPS, and ScaLAPACK. As such, PyTrilinos makes it possible to access state-of-the-art direct solver algorithms developed by groups of specialists, and written in different languages (C, FORTRAN77, FORTRAN90), in both serial and parallel environments. By using Amesos, more than 350,000 code lines (without considering BLAS, LAPACK, and ScaLAPACK) can be easily accessed from any code based on Trilinos (and therefore PyTrilinos).

AztecOO [7] provides object-oriented access to preconditioned Krylov accelerators, like CG, GMRES and several others, based on the popular Aztec library. One-level domain decomposition preconditioners based on incomplete factorizations are available.

IFPACK [5] contains object-oriented algebraic preconditioners, compatible with Epetra and AztecOO. It supports construction and use of parallel distributed memory preconditioners such as overlapping Schwarz domain decomposition with several local solvers. IFPACK can take advantage of SPARSKIT, a widely used software package.

ML [6] contains a set of multilevel preconditioners based on aggregation procedures for serial and vector problems compatible with Epetra and AztecOO. ML can use the METIS and ParMETIS libraries to create the aggregates.

NOX is a collection of nonlinear solver algorithms. NOX is written at a high level with low level details such as data storage and residual computations left to the user. This is facilitated by interface base classes which users can inherit from and define concrete methods for residual fills, Jacobian matrix computation, etc. NOX also provides some concrete classes which interface to Epetra, LAPACK, PETSc and others.

LOCA is the library of continuation algorithms. It is based on NOX and provides stepping algorithms for one or more nonlinear problem parameters.

New_Package is a parallel "Hello World" code whose primary function is to serve as a template for Trilinos developers for how to establish package interoperability and apply standard utilities such as auto-tooling and automatic documentation to their own packages. For the purposes of PyTrilinos, it provides an example of how to wrap a Trilinos package.

5 Serial and Parallel Environments

Although testing and development of high-performance algorithms can be done in serial environments, parallel environments still constitute the most important field of application for most Trilinos algorithms. However, Python itself does not provide any parallel support. Because of this, several projects have been developed independently to fill the gap between Python and MPI. All of these projects allow the use of Python through the interactive prompt, but additional overhead is introduced. Also, none of these projects define a well-recognized standard, since they are still under active development.

The PyTrilinos approach is somewhat complementary to the efforts of these projects. We decided to use a standard, out-of-the-box, Python interpreter, then wrap only the very basics of MPI: MPI_Init(), MPI_Finalize(), and MPI_COMM_WORLD. By wrapping these three objects, we can define an MPI-based Epetra communicator (derived from the pure virtual class Epetra_Comm class), on which all wrapped Trilinos packages are already based. This reflects the philosophy of all the considered Trilinos packages, that have no explicit dependency on MPI communicators, and accept the pure virtual class Epetra_Comm instead.

The major disadvantage of our approach is that Python cannot be run interactively if more than one processor is used. Although all the most important MPI calls are available through Epetra.Comm objects (for example, the rank of a process is returned by method `comm.MyPID()` and the number of processes involved in the computation by method `comm.NumProc()`), not all the functions specified by the MPI forum are readily available through this object. For example, at the moment there are no point-to-point communications, or non-blocking functions (though they could be easily added in the future).

In our opinion, these are only minor drawbacks, and the list of advantages is much longer. First, since all calls are handled by Epetra, no major overhead occurs, other than that of parsing a Python instruction. Second, all PyTrilinos modules that require direct MPI calls can dynamically cast the Epetra.Comm object, retrieve the MPI communicator object, then use direct C/C++ MPI calls. As such, the entire set of MPI functions is available to developers with no additional overhead. Third, a standard Python interpreter is used. Finally, serial and parallel scripts can be identical, and PyTrilinos scripts can be run in parallel from the shell in the typical way.

6 Numerical Results

We now present some numerical results that compare the CPU time required by PyTrilinos and MATLAB 7.0 (R14) to create a serial sparse matrix. The test creates a sparse diagonal matrix, setting one element at a time. The MATLAB code reads:

```
A = spalloc(n, n, n);
for i=1:n
  A(i,i) = 1;
end
```

while the PyTrilinos code contains the instructions:

```
A = Epetra.CrsMatrix(Epetra.Copy, Map, 1)
for i in xrange(n):
  A.InsertGlobalValues(i, [1.0], [i])
A.FillComplete()
```

Clearly, other techniques exist to create in MATLAB and in PyTrilinos sparse diagonal matrices. However, the presented example is representative of several real applications, which often have the need of setting the elements of a matrix one (or a few) at a time. Numerical results for this test case are reported in Table 1, left. Even for mid-sized sparse matrices, PyTrilinos is much faster than MATLAB's built-in sparse matrix capabilities. Table 1, right, reports the CPU required for a matrix-vector product. The sparse matrices arise from a 5-pt discretization of a Laplacian on a 2D Cartesian grid (as produced in MATLAB by the command `gallery('poisson', n)`). Note that PyTrilinos is up to 50% faster than MATLAB for this very important computational kernel.

Since PyTrilinos is intended largely for sparse matrices, these results confirm the achievement of project goals compared to MATLAB, especially because the set of algorithms available in PyTrilinos to handle and solve sparse linear systems is superior to that available in MATLAB.

We now report numerical comparisong for a matrix-vector product performed in PyTrilinos and in Trilinos, using the Epetra package. The results are given in Figure 1. Experiments were conducted on a 16-node Linux/GCC/LAM-MPI cluster where each node has a single AMD Athlon (Barton 2600) processor and

Table 1. On the left, CPU time (in seconds) required by MATLAB and PyTrilinos to set the elements of a sparse diagonal matrix of size n. On the right, CPU time (in seconds) required by MATLAB and PyTrilinos to perform 100 matrix-vector products. The sparse matrices, of size $n \times n$, correspond to a 5-pt discretization of a 2D Laplacian on a rectangular Cartesian grid.

n	MATLAB	PyTrilinos
1,000	0.00397	0.0059
10,000	0.449	0.060
50,000	11.05	0.313
100,000	50.98	0.603

n	MATLAB	PyTrilinos
50	0.02	0.0053
100	0.110	0.0288
500	3.130	1.782
1,000	12.720	7.150

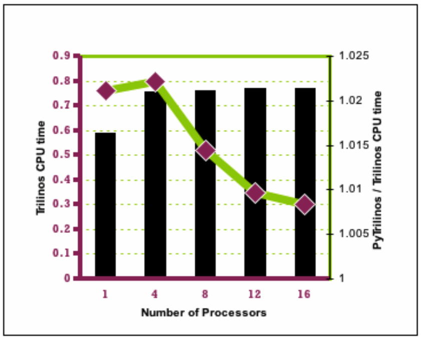


Fig. 1. Wall-clock time (in seconds) on 16-node Linux/GCC cluster. The bars report the time required by Trilinos (scale on the left), while the line reports the ratio between the time required by PyTrilinos and the time required by Trilinos (scale on the right).

the cluster has a dedicated Fast Ethernet switch for inter-node communication. The timing results show that there is essentially no difference in timing results for either version, and that parallel scalability is excellent, as it should be for this type of problem.

7 Conclusions

In this paper we have presented an overview of the PyTrilinos project, an effort to facilitate the design, integration and ongoing support of Python access to a large collection of mathematical software libraries. PyTrilinos provides a simple but powerful rapid development environment, along with the integration tools needed to apply it in realistic environments. In our opinion, the most significant impact of PyTrilinos is in the following areas:

- **Rapid Prototyping.** Because Python is a simple language, coding is much faster than in other languages. For example, its dynamic typing, built-in containers, and garbage collection eliminate much of the manual bookkeeping code typically required in languages like C or C++. As most bookkeeping code is missing, Python programs are easier to understand and more closely reflect the actual problem they're intended to address. Often, well-written Python code looks like pseudo code, and as such it is easier to write, read, and maintain.

– **Brevity.** Python codes can be short and concise. Since things like type declaration, memory management, and common data structure implementations are absent, Python programs are typically a fraction of their C or C++ equivalents. Brevity is also promoted by the object-oriented design of both PyTrilinos and Trilinos itself. Python scripts are short, generally with few jump statements, and therefore have good software metrics in terms of code coverage.
– **Modularity and Reusability.** Python allows the code to be organized in reusable, self-contained modules. This also reflects the natural structure of Trilinos itself. Since Python supports both procedural and object-oriented design, users can adopt their preferred way of writing code.
– **Explorative Computation.** Since Python is an interpreted and interactive scripting language, the user can undertake computations in an explorative and dynamic manner. Intermediate results can be examined and taken into account before the next computational step, without the compile-link-run cycle typical of C or C++.
– **Integration.** Python was designed to be a "glue" language and PyTrilinos relies on the ability to mix components written in different languages. Python lends itself to experimental, interactive program development, and encourages developing systems incrementally by testing components in isolation and putting them together later. By themselves, neither C nor Python is adequate to address typical development bottlenecks; together, they can do much more. The model we are using splits the work effort into *front-end* components that can benefit from Python's easy-of-use and *back-end* modules that require the efficiency of compiled languages like C, C++, or FORTRAN.
– **Software Quality.** Software quality is of vital importance in the development of numerical libraries. If the quality of the software used to produce a new computation is questionable, then the result must be treated with caution as well. If, however, the quality of the software is high, it can reliably be made available to other research groups.

Producing high quality software for state-of-the-art algorithms is a challenging goal. Therefore, the production of high quality software requires a comprehensive set of testing programs. A way to do that without influencing the rapid development of prototype code, is to write tests in Python. By helping to detect defects, PyTrilinos can become an important testing tool for Trilinos itself. (Clearly, PyTrilinos tests require a bug-free interface between Trilinos and PyTrilinos.) Using PyTrilinos in the Trilinos test harness, one can experiment with the code to detect and manage dynamic errors, while static errors (like argument checking) must be detected by other types of testing.

– **Stability.** The only Python module on which PyTrilinos depends is NumPy, for both serial and parallel applications. Since NumPy is a well-supported and stable module, users can develop their applications based on PyTrilinos with no need to change or update them in the near future.
– **Data Input.** All scientific applications require data to be passed into the code. Typically, this data is read from one or more files and often the input logic becomes extensive in order to make the code more flexible. In other words, the scientific code developers often find themselves implementing a rudimentary scripting language to control their application. We have found that applications developed in Python avoid this distraction from more scientific work because the Python scripts themselves become high-level "input files," complete with variable definitions, looping capabilities and every other Python feature.

Of course, Python (and by extension PyTrilinos) is not the perfect language or environment for all problems. The most important problems we have encountered are:

– **Portability.** PyTrilinos is developed concurrently on both Linux and Mac OS X, and it should port successfully to most other platforms where Trilinos, Python, NumPy and SWIG are available. However, configuring Trilinos, and thus PyTrilinos, for a new platform can be a non-trivial exercise.
– **Shared Libraries on Massively Parallel Computers.** Another problem is related to the shared library approach, the easiest way of integrating third-party libraries in Python. Most massively parallel computers do not support shared libraries, making Python scripts unusable for very large scale computations.
– **Lack of Compile-time Checks.** In Python all checks must be performed at run-time. Furthermore, Python does not support strong typing of variables, so user mistakes related to incorrect variable types can be a challenge to find and correct, where these types of mistakes would be caught quickly by a strongly-typed language and compiling system such as C++ and Java.
– **Performance Considerations.** By using a Python wrapper, a performance penalty is introduced due to decoding of Python code, the execution of wrapped code, and returning the results in a Python-compliant format. These tasks may require thousands of CPU cycles, therefore it is important to recognize this situation when it occurs. The performance penalty is small if the C/C++ function does a lot of work. Therefore, for rarely called functions, this penalty is negligible. All performance critical kernels should be written in C, C++, or Fortran, and everything else can be in Python.
– **Management of C/C++ Arrays.** Although SWIG makes it easy to wrap Python's lists as C and C++ arrays (and vice-versa), this process still requires the programmer to define wrappers in the interface file, that converts the array into a list, or a list into an array. Without an explicit wrapper, the proper handling of arrays can result in non-intuitive code, or memory leaks.

The most important feature of Python is its powerful but simple programming environment designed for development speed and for situations where the complexity of compiled languages can be a liability. Of course, Python enthusiasts will point out several other strengths of the language; our aim was to show that Python can be successfully used to develop and access state-of-the-art numerical solver algorithms, in both serial and parallel environments.

We believe that PyTrilinos is a unique effort. For the first time a large number of high-performance algorithms for distributed sparse linear algebra is easily available from a scripting language. None of the previously reported projects for scientific computing with Python handles sparse and distributed matrices, or the diversity of solver algorithms. We hope that PyTrilinos can help to make the development cycle of high-performance numerical algorithms more efficient and productive.

References

1. Beazley, D.M.: Automated scientific software scripting with SWIG. Future Gener. Comput. Syst. 19(5), 599–609 (2003)
2. Heroux, M.A.: Epetra Reference Manual, 2nd edn (2002), `http://software.sandia.gov/trilinos/packages/epetra/doxygen/latex/EpetraRefere-nceManual.pdf`
3. Heroux, M., Bartlett, R., Hoekstra, V.H.R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories (2003)
4. Sala, M.: Amesos 2.0 reference guide. Technical Report SAND-4820, Sandia National Laboratories (September 2004)
5. Sala, M., Heroux, M.: Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories (February 2005)
6. Sala, M., Hu, J., Tuminaro, R.: ML 3.1 smoothed aggregation user's guide. Technical Report SAND-4819, Sandia National Laboratories (September 2004)
7. Tuminaro, R., Heroux, M., Hutchinson, S., Shadid, J.: Official Aztec user's guide: Version 2.1. Technical Report Sand99-8801J, Sandia National Laboratories, Albuquerque NM, 87185 (November 1999)

Amesos: A Set of General Interfaces to Sparse Direct Solver Libraries

Marzio Sala[1], Ken Stanley[2], and Michael A. Heroux[3]

[1] Department of Computer Science, ETH Zurich, CH-8092 Zurich
marzio@inf.ethz.ch
[2] Department of Computer Science, Oberlin College, Oberlin, Ohio, USA
[3] PO Box 5800 MS 1110, Albuquerque, NM 87185-1110, USA⋆

Abstract. We present the Amesos project, which aims to define a set of general, flexible, consistent, reusable and efficient interfaces to direct solution software libraries for systems of linear equations on both serial and distributed memory architectures. Amesos is composed of a collection of pure virtual classes, as well as several concrete implementations in the C++ language. These classes allow access to the linear system matrix and vector elements and their distribution, and control the solution of the linear system. We report numerical results that show that the overhead induced by the object-oriented design is negligible under typical conditions of usage. We include examples of applications, and we comment on the advantages and limitations of the approach.

1 Introduction

This paper describes the design and the implementation of interfaces to serial and parallel direct solution libraries for linear systems of type

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a real and sparse square matrix, and $x, b \in \mathbb{R}^n$ are the solution and the right-hand side vectors, respectively.

Generally speaking, a direct solution algorithm for (1) is any technique that computes three matrices, L, D and U, such that $P\,A\,Q = L\,D\,U$, where P and Q are permutation matrices and the linear systems with matrices L, D and U are easy to solve. The process of computing these three matrices is called *factorization*. Generally, L is a lower triangular matrix, U is an upper triangular matrix, D is a diagonal matrix (or possibly the identity matrix), and the algorithm adopted for their computation is some variant of the Gaussian elimination method.

⋆ ASCI program and the DOE Office of Science MICS program at Sandia National Laboratory. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The point of view of this paper is the one of application developers, interested in *using* direct solver libraries for a specific problem. As such, we will not consider the *development* of direct solvers; instead, we suppose that software libraries offering algorithms for the direct solution of (1) are already available. The objective is *interfacing* one or more of these libraries with the application code of interest. This usually involves the following steps. First, a library is chosen because of its robustness, performances in terms of CPU time and memory, documentation, usage, availability. Then, an interface between the application code and the library is written, and this may require storing the linear system matrix A using the storage format required by the library. Finally, the calling sequence is hard-wired in the final application code. The outlined process is typically repeated every time a new application is developed, and every time a new solver is under consideration.

Note that, for both serial and distributed sparse solvers, no "gold standard" exists. Therefore, application developers aiming to solve (1) generally have to look for a suitable library that meets their needs. The choice is affected by several criteria, like the ease of use, availability, quality of documentation and support, and, of course, the set of capabilities, the solution time, and the memory usage. The relative importance of these aspects is usually subjective. For relatively simple problems, the performance is usually the key point, followed by memory usage. For complicated problems, and especially in industrial applications, reliability is of paramount importance.

In our opinion, writing a custom interface between the selected library and the application is sub-optimal for both application and library developers because it:

1. **Offers partial coverage.** Writing a custom-made interface means that only the targeted library will be used. This is inconvenient because of the already mentioned difficulty to choose *a-priori* the best library for a given application. In some cases, a theoretical analysis of the problem at hand can suggest the right algorithm. Alternatively, one can consider numerical comparisons on test matrices available in the literature. Often, however, one has to validate a given library on the application, architecture, and data set of interest, and this can be done only if the interface is already available;
2. **Produces maintenance problems.** Including the interfaces within the application code requires the application developers to manipulate the matrix format, the memory management, and the calling conventions, that can vary from one library to the following. Although not necessary difficult, these activities are usually time-consuming;
3. **Delays the usage of new libraries.** Large application codes usually have a long life. The sheer size of the codes and a natural reluctance to change successful projects discourage any effort of rewriting unless absolutely necessary. Since a new library (or a new version of a given library) may require a new matrix format or distribution, or new calling conventions, application developers may simply decide to continue using the interface already developed.

In this paper, we present a software project, called Amesos which aims to address these problems by using object-oriented (OO) design and programming. Amesos is composed of a set of clean, consistent and easy-to-use interfaces between the application and the direct solver libraries. Each interface takes care of dealing with the direct solver library, in a manner that is transparent to the application developers, and automatically manages matrix formats, data layout, and calling conventions.

The paper is organized as follows. Section 2 describes the requirements and the design of the Amesos project. The basic classes are outlined in Section 3, and the list of supported solvers is given in Section 4. Section 5 reports an example of usage and some numerical results that quantify the overhead required by the generality of the approach. Finally, Section 6 outlines the conclusions.

2 Project Design

The Amesos design is based on the following requirements:

1. **Simplicity of usage.** Solving linear system (1) in a language like MATLAB is very easy, i.e. one just writes `x = A \ b`. It should not be much more difficult in a non-MATLAB code;
2. **Flexibility.** More than one algorithm/library must be available, for both serial and parallel architectures;
3. **Efficiency.** The overhead due to the framework must be minimal.

The basic ideas of this design are indeed quite old, and can be traced back to almost 30 years ago [8,11]. More recently, articles [10,6] discussed the usage of abstract interfaces and OO design for the direct solution of sparse linear systems. We extend these ideas by abstracting the concepts to a higher level, and making the interfaces independent of the supported libraries. The interfaces are presented and implemented as a set of C++ classes using several well-known design patterns. C++ supports object-oriented programming, and it is relatively easy to interface Fortran 77, Fortran 90 and C libraries with C++ code. The C++ language supports abstraction through classes, inheritance and polymorphism. For application developers, abstraction is important because it brings simplicity, by allowing components with a minimal interface. It also ensures flexibility because it decouples the different algorithmic phases from the data structures. Finally, abstraction allows extensibility in the sense that new (yet to be developed) libraries can be easily added, at almost no cost to the application developer. Another candidate language could have been Fortran 90, but it does not have inheritance and polymorphism.

Regarding the parallel computing mode, we consider parallel architectures with distributed memory, and we suppose that the message passing interface MPI is adopted. This approach is the de-facto standard in scientific computing. As a result, the presented design can be easily interfaced with the aforementioned projects.

3 Basic Classes

The key point of our design is to manage all direct solver libraries by a workflow structured as follows:

1. Definition of the sparsity pattern of the linear system matrix;
2. Computation of the symbolic factorization, which includes preordering and analysis. The symbolic factorization refers to all operations that can be performed by accessing the matrix structure only (without touching the values of the matrix entries);
3. Definition of the values of the linear system matrix;
4. Computation of the numeric factorization, that is, the computation of the entries of the factored terms;
5. Definition of the right-hand side b;
6. Solution of the linear system, that is, computation of x.

Steps 1, 3 and 5 are application-dependent and will not be discussed here; instead, our aim is to standardized steps 2, 4 and 6 by adding an intermediate layer between the application and the direct solver libraries. The design discussed below uses several common design patterns [9]. The first is the *Builder Pattern*, by which we define an abstract class whose methods are steps 2, 4 and 6. Because each of these steps could in principle be replaced with an alternate algorithm, our design also represents the *Strategy Pattern*. Finally, we use the *Factory Pattern* as a means of selecting a specific concrete solver instance by use of a string argument.

Presently we introduce a set of abstract classes, which will be used to define interfaces to the data layout of distributed objects, vectors, matrices, linear system, and the solver. The design is reported here as a set of C++ classes, but the concepts are more general and the discussion is largely language-independent.

Our design is based on a set of C++ pure virtual classes, which will be used to define interfaces to the data layout of distributed objects, vectors, matrices, linear system, and the solver. These classes are: a `Map` class that specifies the layout of matrix rows; a `Vector` class that defines distributed vectors; a `Matrix` class which offers a standard calling sequence to access matrix elements. We suppose that each matrix row is assigned to exactly one processor, and it is easy to access all nonzero elements of any locally owned row by calling a `GetRow()` method which returns the nonzero indices and values for the specified (local) row[1]; a `LinearProblem` class which contains the linear system matrix, the solution and the right-hand side vectors; a `Solver` class that manages the underlying solver library.

The pure virtual class `Solver` deserves more details. This class contains the following methods:

– `void SetLinearProblem()` sets the linear problem to solve;

[1] A row-based approach is adopted or supported by most parallel linear algebra libraries, like PETSc, AztecOO, Epetra and HYPRE.

– `void SetParameters(List)` specifies all the parameters for the solver by using a generic container (hash table);
– `int SymbolicFactorization()` performs the symbolic factorization, that is, all the operations that do only require the matrix graph and not the actual matrix values;
– `int NumericFactorization()` performs the numeric factorization, that is, it computes the matrices L, D and U by accessing the matrix values. Both the solution and the right-hand side vectors are not required in this phase;
– `int Solve()` solves the linear system. This phase requires the solution and the right-hand side vectors.

Note that a concrete implementation may decide to skip the symbolic factorization (that is, the method returns without doing nothing) and perform the entire factorization within `NumericFactorization()`. Equivalently, one can implement both factorization and solution within `Solve()`, and let both factorization phases are no-operation methods.

The design is organized as follows: each interface is defined by a class, derived from the pure virtual `Solver` class, and it internally allocates and manages all the objects required by the underlying solver. This insulates the user from lower-level details.

4 Supported Libraries

Amesos offers interfaces to the following direct solvers: LAPACK, ScaLAPACK, KLU[2] [4], UMFPACK [3], SuperLU and SuperLU_DIST [5] DSCPACK [14], MUMPS [1], TAUCS [13], and PARDISO [17,18]. Other interfaces are under development.

Amesos takes advantage of the Epetra package [12] to specify the `Map, Vector, Matrix` and `LinearProblem` interfaces. The Standard Template Library (STL) is used to increase performances whenever possible.

To increase portability, Amesos is configured using Autoconf and Automake; each interface can be enabled or disabled at configure time. Users can take advantage of the bug-tracking tool Bugzilla to provide feedback or request improvements. Amesos can be downloaded as part of the Trilinos framework at the web page `http://software.sandia.gov/trilinos/packages/amesos`. More details on the usage of Amesos can found in [15] and the on-line documentation, while the design is presented in greater details in [16].

5 Example of Usage and Numerical Results

An example of usage of Amesos is reported in Figure 1. Although this particular example requires MPI, Amesos can be compiled with or without support for MPI. (Clearly, distributed solvers are available only when compiled with MPI

[2] The sources of KLU are distributed within Amesos.

support.) The only AMESOS include file is Amesos.h, which does not include the header files of the supported library. The required interface is created by using the factory class Amesos. Factory classes are a programming tool that implements a sort of "virtual constructor," in the sense that one can instantiate a derived (concrete) object while still working with abstract data types. The example reported in Figure 1 adopts UMFPACK as a solver; however by using the factory class, other interfaces can be created by simply changing the input string. Note that the supported solver can be serial or parallel, dense or sparse: the user code still remains the same, except for the name of the solver; AMESOS will take care of data redistribution if required by the selected solver.

```
#include "Amesos.h"
#include "mpi.h"
#include "Epetra_MpiComm.h"
...
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  Epetra_MpiComm Comm(MPI_COMM_WORLD);

  <define Map, x, b, A>

  Epetra_LinearProblem Problem(A, x, b); // Interface 3.4

  Amesos Factory;                    // create a factory class
  Amesos_BaseSolver* Solver = Factory.Create("Umfpack", Problem);

  Teuchos::ParameterList List;       // allocate container for params,
  List.set("PrintTiming", true);     // set one in the container, then
  Solver->SetParameters(List);       // pass the container to the solver

  Solver->SymbolicFactorization(); // symbolic factorization
  Solver->NumericFactorization();  // numeric factorization
  Solver->Solve();                   // linear system solution
  delete Solver;

  MPI_Finalize();
  return(EXIT_SUCCESS);
} // end of main()
```

Fig. 1. Example of code using AMESOS. The parallel code uses the AMESOS interface to the serial UMFPACK library to solve the linear system. AMESOS takes care of gathering the matrix and vectors on processor 0 and scattering the solution as required. The creation of the matrix, solution and right-hand side are not reported.

The example code of Figure 1 has shown the facility of usage of Amesos. We now address the effectiveness of the interface. Table 1 reports the percentage of CPU time required by the Amesos interface with respect to the time required by the underlying library. We have considered the SuperLU interface for the solution of some of the matrices in the FIDAP collection, available at [2]. The matrix sizes range from 1159 (`FIDAP032`) to 22294 (`FIDAPM11`). All problems are solved using a 1.67 GHz G4 processor with 1024 Mbytes of RAM, running MAC OS X 10.4 and gcc 4.0.0. The table reports the percentage of the CPU time required by the interface with respect to the time required by the considered solver, and quantifies the overhead required by Amesos. We have used the default set of parameters for both solvers.

As expected, for small matrices (for example `FIDAP005` or `FIDAPM05`, of size 27 and 42, respectively) the overhead is considerable. When considering bigger matrices ($n > 3000$), then the overhead is always below 5%. All the overhead is spent in converting the matrix from the abstract matrix format to the format required by the library, and performing additional safety checks. Note that this overhead can indeed be reduced by adding specialized classes, that satisfies the interface and internally store the matrix in the format required by a given solver library, then perform a `dynamic_cast` to obtain the already allocated data structure containing the matrix. This solution is inelegant and requires knowledge of derived classes in the solver interface, but could greatly increase performance.

Table 1. Additional time required by the Amesos interface with respect to the time required by the solver for different matrices. n represents the size of the matrix, nnz the total number of nonzeros, and δ is defined as (additional Amesos time)/(time spent within SuperLU)* 100. The results were obtained on a G4 1.67 GHz with 1 GByte of RAM.

Name	n	nnz	nnz/n	δ
`FIDAP015`	6867	96421	14.04	2.89
`FIDAP018`	5773	69335	12.01	2.14
`FIDAP019`	12005	259863	21.64	3.34
`FIDAP020`	2203	69579	31.58	1.77
`FIDAP031`	3909	115299	29.49	3.51
`FIDAP032`	1159	11343	9.786	4.84
`FIDAP033`	1733	20315	11.72	13.1
`FIDAP035`	19716	218308	11.07	2.53
`FIDAPM03`	2532	50380	19.89	1.61
`FIDAPM11`	22294	623554	27.96	0.403
`FIDAPM29`	13668	186294	13.62	1.57

6 Concluding Remarks

In this paper, we have presented the Amesos project, which defines a model to access direct solver libraries. The advantages of this model are the following:

– The actual data storage format of the linear system matrix becomes largely unimportant. Each concrete implementation will take care, if necessary, to convert the input matrix to the required data format. This means that the application can choose *any* matrix format that can be wrapped by the abstract matrix interface.
– Issues like diagonal perturbations, dropping, reordering or fill-reducing algorithms can be easily introduced within the abstract matrix interface. For example, a dropping strategy or a modification of the diagonals simply requires a new `GetMyRow()` method, without touching the actual matrix storage. Also, reordering techniques can be implemented and tested independently of the library used to perform the factorization.
– The actual calling sequence required by each library to factor the matrix and solve the linear system is no longer exposed to the user, who only has to call methods `SymbolicFactorization()`, `NumericFactorization()` and `Solve()`.
– Interfaces can be tested more easily because they are all located within the same library and not spread out into several application codes. The framework is also quite easy to learn and use, since a basic usage requires about 20 code lines (see the example of Figure 1).
– It is easy to compare different solvers on a given set of problems. The AMESOS distribution contains a (working) template that reads a linear system from file in the popular Harwell/Boeing format [7] and solves it with all the enabled solvers. Users can easily modify this template to numerically evaluate the optimal library for *their* problems.
– The framework can serve users with different levels of expertise, from the usage of libraries as black-box tools, to a fine-tuning of each library's parameters.
– The framework can be easily extended to incorporate libraries for the resolution of over-determined and under-determined systems. Solving such systems may involve algorithms other than Gaussian eliminations; nevertheless, the interfaces will remain almost untouched.

The generality of the proposed model comes at a price. The presented model has the following limitations:

– Overhead may be introduced when converting or redistributing the matrix and/or the vectors into the library's format. For very large matrices, this can constitute a problem especially in terms of memory consumption, but is often not a first-order concern.
– Fine-tuning of solver's parameters can be difficult. Also, we offer no "intelligent" way of setting these parameters.
– There is no standard way to convert MPI communicators defined in C to MPI communicators defined in Fortran 90. On some architectures it is difficult or even impossible to perform such a task. Some hacks may be required.
– No support is offered for matrices in elemental format.
– It is almost impossible to support different releases of a given software library, because function names usually do not change from one version to the next,

making it impossible for the linker to select the appropriate version of the library.
- Some libraries offer a one-solve routine without storing any data after the solution is completed; this option is not supported by the presented design, but it could be easily added.
- There are no capabilities to obtain the L, D and U factors, or the reorderings P and Q. This is because each supported package uses a different storage format and distribution. Reordering and scaling can be made library-independent by extending the presented abstract interfaces.
- Problematic user-package communications. Because of the high-level view, the code is safer: it is more difficult to make errors or call the solver with the wrong set of parameters. AMESOS classes automatically perform safety checks, and return an error code when something goes wrong. However, it is often difficult to abstract the error messages from all supported libraries and report them in a uniform fashion. Users still need to consult the library's manual to decode the error messages.
- Finally, adding support for new direct solvers, or updating to newer versions of already supported libraries depends on the reactivity of the AMESOS developers. The AMESOS web page reports several mailing lists that can be used to communicate with developers.

Despite these issues, we find that the presented set of interfaces brings its users the well-known benefits of reusable libraries. Thanks to their generality, these interfaces (and the corresponding codes) can be used to easily connect intricate applications with state-of-the-art linear solver libraries, in a simple and easy-to-maintain way. From the point of view of application developers, the small amount of required code makes it very convenient to adopt a project like AMESOS. For the developers of linear solver libraries, writing one interface for their own solver can help to make it applicable and testable to a vast range of applications.

One of our goal in the design of AMESOS was to reduce the intellectual effort required to use direct solver libraries. We feel that this objective has been achieved, and the performance penalty is very limited in most cases. In our opinion, the main limitation of AMESOS is that it supports double precision only, while most direct solvers allows the solution in single precision and complex arithmetics.

References

1. Amestoy, P.R., Duff, I.S., L'Excellent, J.-Y., Koster, J.: MUltifrontal Massively Parallel Solver (MUMPS Versions 4.3.1) Users' Guide (2003)
2. Boisvert, R.F., Pozo, R., Remington, K., Barrett, R.F., Dongarra, J.J.: Matrix market: a web resource for test matrix collections. In: Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software, pp. 125–137. Chapman & Hall, Ltd., London (1997)
3. Davis, T.A.: UMFPACK home page (2003), `http://www.cise.ufl.edu/research/sparse/umfpack`

4. Davis, T.A., Palamadai, E.: KLU: a sparse LU factorization for circuit simulation matrices. Technical report, Technical report, Univ. of Florida (2005) (in preparation)
5. Demmel, J.W., Gilbert, J.R., Li, X.S.: SuperLU Users' Guide (2003)
6. Dobrian, F., Kumfert, G., Pothen, A.: The design of sparse direct solvers using object-oriented techniques. Technical report (1999)
7. Duff, I.S., Grimes, R.G., Lewis, J.G.: Sparse matrix test problems. ACM Trans. Math. Softw. 15(1), 1–14 (1989)
8. Duff, I.S., Reid, J.K.: Performance evaluation of codes for sparse matrix problems. In: Performance evaluation of numerical software, pp. 121–135. North-Holland, New York (1979)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc, Boston (1995)
10. George, A., Liu, J.: An object-oriented approach to the design of a user interface for a sparse matrix package. SIAM Journal on Matrix Analysis and Applications 20(4), 953–969 (1999)
11. George, A., Liu, J.W.H.: The design of a user interface for a sparse matrix package. ACM Trans. Math. Softw. 5(2), 139–162 (1979)
12. Heroux, M.A.: Epetra Reference Manual, 2.0 edn. (2002), `http://software.sandia.gov/trilinos/packages/epetra/doxygen/latex/Epetr aRefer-enceManual.pdf`
13. Irony, D., Shklarski, G., Toledo, S.: Parallel and fully recursive multifrontal supernodal sparse cholesky. Future Generation Computer Systems 20(3), 425–440 (2004)
14. Raghavan, P.: Domain-separator codes for the parallel solution of sparse linear systems. Technical Report CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University (2002)
15. Sala, M.: Amesos 2.0 reference guide. Technical Report SAND-4820, Sandia National Laboratories (September 2004)
16. Sala, M., Stanley, K.S., Heroux, M.A.: On the design of interfaces to sparse direct solvers 2006 (submitted)
17. Schenk, O., Gärtner, K.: On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report, Department of Computer Science, University of Basel 2004 (submitted)
18. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with PARDISO. Journal of Future Generation Computer Systems 20(3), 475–487 (2004)

d-Spline Based Incremental Parameter Estimation in Automatic Performance Tuning

Teruo Tanaka, Takahiro Katagiri, and Toshitsugu Yuba

Graduate School of Information Systems,
The University of Electro-Communications,
1-5-1 Chofu-gaoka, Choufu-shi, Tokyo 182-8585, Japan
{teru, katagiri, yuba}@hpc.is.uec.ac.jp

Abstract. In this paper, we introduce a new *d-Spline* based Incremental Performance Parameter Estimation method (*IPPE*). We first define a fitting function *d-Spline*, which has high flexibility to adapt given data and can be easily computed. The complexity of *d-Spline* is $O(n)$. We introduce a procedure for incremental performance parameter estimation and an example of data fitting using *d-Spline*. We applied the *IPPE method* to automatic performance tuning and ran some experiments. The experimental results illustrate of the advantages of this method, such as high accuracy with a relatively small estimation time and high efficiency for large problem sizes.

Keywords: automatic performance tuning, performance parameter estimation, Givens method, mathematical library.

1 Introduction

Automatic performance tuning is an optimization of performance parameters suitable for a certain computational environment in ordinary mathematical libraries [1,2,3]. The target computers include supercomputers, PC servers, PC clusters and MPPs. Viewpoints of the optimization include the number of ALUs, cache size, communication latency and/or bandwidth. Performance parameters such as unrolling depth and block size for cache are estimated for these computational environments.

In conventional performance parameter estimation the following procedure is applied to get the optimal values in a mathematical library:

step1. Choose static sampling points from values of the performance parameter,
step2. Run the target mathematical library to obtain execution time (executed value) at each sampling point,
step3. Define a fitting function and fit it to the executed values,
step4. Search a minimum value point of the fitting function which corresponds to the optimal value of the performance parameter.

In conventional methods the optimal performance parameters are estimated using a previously fixed number of (such as static) sampling points. When the number of fixed sampling points is large, the accuracy for estimation is high, but the computational efficiency is low and when the number of fixed sampling points is small, *vice versa*. This is a problem for the fixed number of sampling points.

In general, conventional performance parameter estimation has been used in automatic performance tuning.

In the case of automatic performance tuning, the possibility to execute the target mathematical library at any sampling point exists. It is possible to dynamically increment sampling points. In this paper we introduce a new method, named the Incremental Performance Parameter Estimation (*IPPE*), in which the estimation is started from the least sampling points and incremented dynamically to improve accuracy.

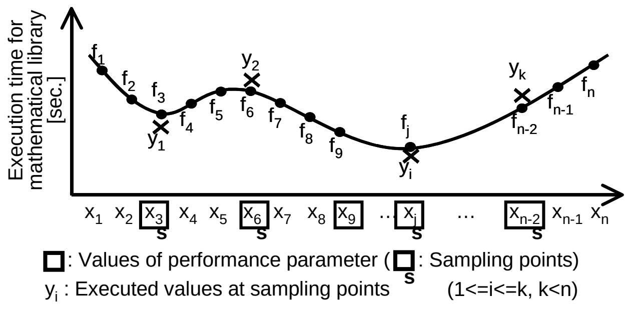
The *IPPE method* has two major issues. The first issue is the selection of a fitting function under the following conditions: 1) it should be calculated by a small number of sampling points because the estimation is started from the least number of sampling points and 2) the calculation cost should be sufficiently small because it is necessary to calculate a fitting function for every sampling point incremented dynamically. The second issue is the criteria for incrementing sampling points; 1) terminating and 2) selecting sampling points.

Section 2 of this paper describes *d-Spline* based incremental performance parameter estimation. Section 3 presents the procedure for *IPPE*. Section 4 illustrates an example of data fitting using *d-Spline*. Section 5 gives an application to automatic performance tuning. Section 6 provides summary, conclusions and future work.

2 *d-Spline* Based Incremental Performance Parameter Estimation

2.1 Fitting Function *d-Spline*

Fig.1 illustrates the fitting function *d-Spline* with sampling points and executed values. Regarding the first issue of the *IPPE method*, we define a fitting function



Execution time for mathematical library [sec.]
f_1
f_2
f_3
f_4
f_5
f_6
f_7
f_8
f_9
f_j
f_{n-2}
f_{n-1}
f_n
y_1
y_2
y_i
y_k
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 … x_j … x_{n-2} x_{n-1} x_n
s
□: Values of performance parameter (□ s: Sampling points)
y_i : Executed values at sampling points (1<=i<=k, k<n)

Fig. 1. Fitting Function *d-Spline*

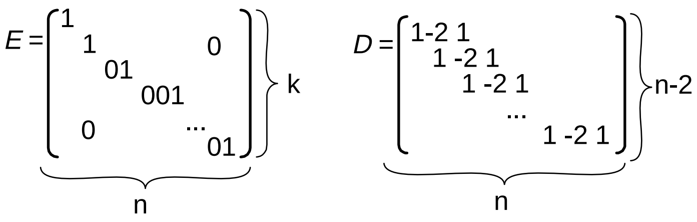


Fig. 2. Structure of Matrices E and D

"*d-Spline*" (discrete spline function), which has high flexibility to adapt to a given data set and which can be easily computed. *d-Spline* is presented as the value of discrete point $f_j = f(x_j)$, $1 \leq j \leq n$. *i.e.* $\boldsymbol{f} = (f_1, f_2, f_3, \ldots, f_j, \ldots, f_n)^t$, where t means transposition (Fig.1). Each x_j has the same interval. Parameter values are a portion of x. For smoothness, the number of x_j (n) must be sufficiently greater than the number of parameter values (N). The executed value y_i $(1 \leq i \leq k, k \leq N)$ for k sampling points out of N is the execution time of the target mathematical program. Executed values at sampling points $\boldsymbol{y} = (y_1, y_2, y_3, \ldots, y_i, \ldots, y_k)^t$. The smoothness of $\boldsymbol{f}$ presents $|f_{j-1} - 2f_j + f_{j+1}|$, $2 \leq j \leq n-1$. $\boldsymbol{f}$ is selected to minimize the following expression(1).

$$\min_f (\| \boldsymbol{y} - E\boldsymbol{f} \|^2 + \alpha^2 \| D\boldsymbol{f} \|^2), \tag{1}$$

where α is sufficiently small to adapt well to the executed values $\boldsymbol{y}$. The first term of expression(1) denotes distance between executed values $\boldsymbol{y}$ and $\boldsymbol{f}$. The second term of expression(1) denotes the smoothness of $\boldsymbol{f}$. Fig.2 depicts matrices E and D each of the size is $k \times n$ and $(n-2) \times n$, respectively. *d-Spline* is not a spline function because the continuity of the derivatives is not guaranteed, hence it is named *d-Spline* for discrete spline.

2.2 Analysis of *d-Spline* Using Givens Method

To solve expression(1), following least square problem is solved [4].

$$\min_f \| \boldsymbol{b} - Z\boldsymbol{f} \|^2. \tag{2}$$

Fig.3 shows the structure of matrix Z and vector $\boldsymbol{b}$. To suppress *fill-in*, changing zero elements to non-zero elements, $\| \boldsymbol{y} - E\boldsymbol{f} \|$ is transferred to $\| E^t\boldsymbol{y} - E^tE\boldsymbol{f} \|$. E^tE has non-zero data on some of the diagonal points. All non-diagonal elements are zero. To solve the least square problem, the Householder method is commonly used. However, for a sparse matrix, Givens method is better than Householder method [5]. Moreover, in our parameter estimation method, Givens method is better for suppressing *fill-in* because of the special structure of the matrix Z.

By applying Givens method, Z is transferred to R, which is a tri-diagonal matrix (half bandwidth is 3), whose computational complexity is $O(n)$, where n

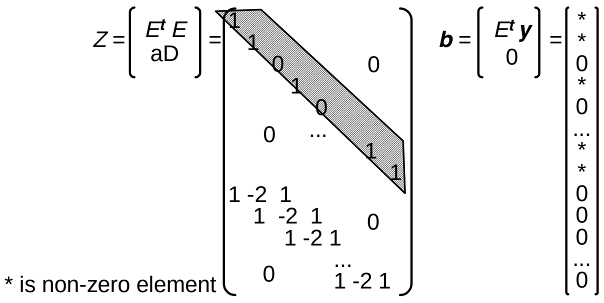
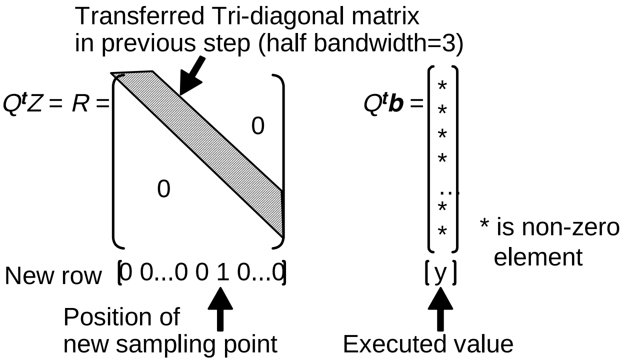


Fig. 3. Structure of Matrix Z and Vector $\boldsymbol{b}$



Transferred Tri-diagonal matrix in previous step (half bandwidth=3)
Q^tZ = R =
0
0
Q^tb =
* * * * ... * *
* is non-zero element
New row [0 0...0 0 1 0...0]
[y]
Position of new sampling point
Executed value

Fig. 4. Adding a New Row at Incrementing Sampling Point

is the number of the parameter value. The computational complexity of mathematical program is $O(L^2)$ or $O(L^3)$, where L is matrix size. Therefore, the computational complexity of solving *d-Spline* can be ignored. Fig.4 illustrates adding a new row at incrementing sampling point. When a new sampling point x_s is incremented, Givens transformation is performed to make R (obtained by existing points, Fig.4) plus one row. It is not necessary to apply Givens transformation for Z. Therefore, the same $O(n)$ requires less computation (approximately 1/5) in the increment phase. This shows that the method of adding sampling points and the fitting function *d-Spline* makes an efficient combination.

3 Procedure for Incremental Performance Parameter Estimation

This section describes a procedure for estimating the optimal parameter value by applying *d-Spline*. The second issue of the *IPPE* criteria for incrementing sampling points, 1) terminating and 2) selecting sampling points are considered here. Fig.5 shows the procedure for *IPPE*. The procedure is outlined below:

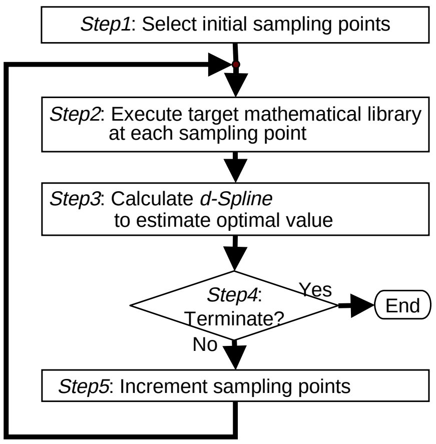

Step1: Select initial sampling points
Step2: Execute target mathematical library at each sampling point
Step3: Calculate d-Spline to estimate optimal value
Step4: Terminate?
Yes
End
No
Step5: Increment sampling points

Fig. 5. Procedure for the *IPPE method*

step1. Select $four$ initial sampling points from x, including both end points,
step2. Execute the target mathematical library for each point x_j to obtain $\boldsymbol{y}$,
step3. Compute *d-Spline* $\boldsymbol{f}$ adaptable to executed values $\boldsymbol{y}$, and estimate x_s which corresponds to f_s, the smallest of $\boldsymbol{f}$,
step4. If x_s is the same for successive p times(p=2,3,4,5), then exit(criterion 1), otherwise go to the next, *step5*,
step5. If estimated x_s is not included in the existing sampling points, then add the x_s as a new sampling point, otherwise select an other x_s to make f_s, such that $\max_s |f_{s-1} - 2f_s + f_{s+1}|$ (criterion 2) and go to *step2*.

In *step4*, the number of succession p is defined considering the balance between the accuracy for estimation and the computational efficiency.

4 An Example of Data Fitting Using *d-Spline*

This section introduces one example for data fitting using *d-Spline*. The target computer is a PC cluster and the target mathematical library is a Househoder tri-digitalization routing using the QR method, and the problem size is 800. The performance parameter of unrolling depth, from 1 to 16, will be estimated.

Fig.6a shows execution times for all of the performance parameter values. In Fig.6a, the X-axis contains values of performance parameter and Y-axis is the execution time for the target mathematical library. The dotted line shows various tops and bottoms, where data fitting is not easy.

Fig.6b shows the phase 1 execution times for all of the performance parameter values. In phase 1, we first select the initial four sampling points with equivalent intervals. The four *bold* points are initially executed values for a target mathematical library corresponding to the four initial sampling points. We will fit the executed values using *d-Spline*, and find its minimum point, which corresponds

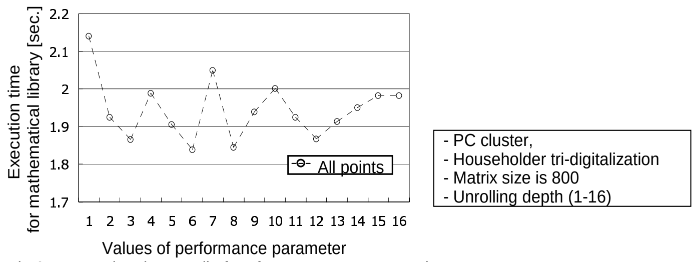


Fig.6a Execution time at all of performance parameter values.

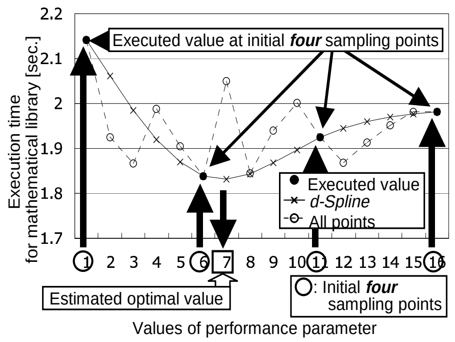


Fig.6b Phase 1.

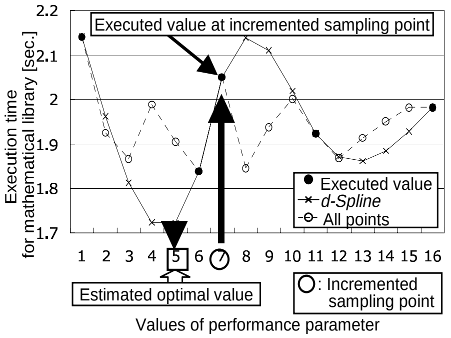


Fig.6c Phase 2.

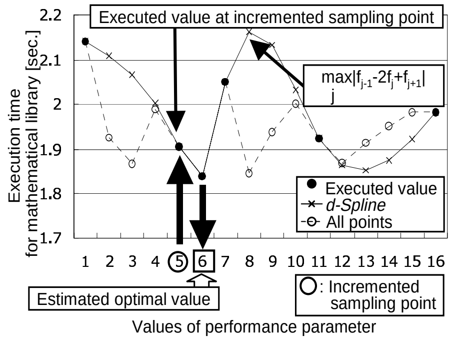


Fig.6d Phase 3.

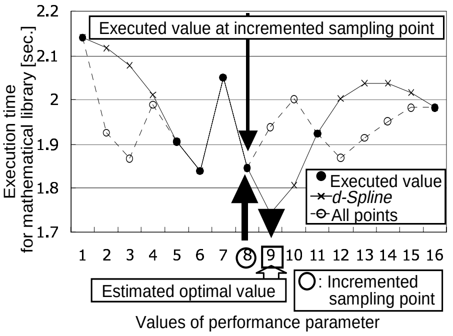


Fig.6e Phase 4.

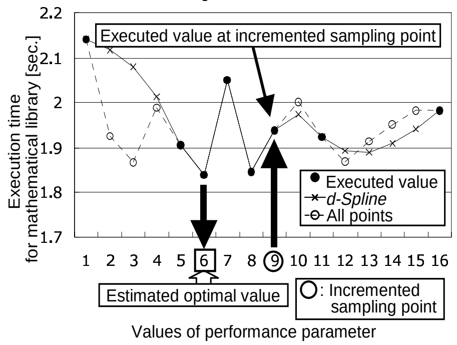


Fig.6f Phase 5.

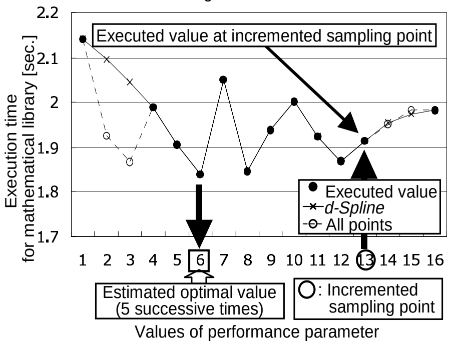


Fig.6g Phase 9.

Fig. 6. Example for Data Fitting Using *d-Spline*

to #7 for values of performance parameter. So we estimate #7 for optimal value. Next, we select a new sampling point #7, because the estimated optimal value of #7 was not included in the initial sampling points.

Fig.6c shows the phase 2 execution times for all of the performance parameter values. In phase 2 #7 is incremented in the sampling points. We fit *d-Spline* including the new executed value. Now the estimated optimal value is #5 corresponding to the minimum point on the *d-Spline*. #5 is a new sampling point.

Fig.6d, Phase 3, shows a new *d-Spline* including the new sampling point #5. We find the minimum point on the *d-Spline* and estimate the optimal value #6. This point is already included in the existing sampling points. In this case we select #8, where the difference equation of the second order has the largest value.

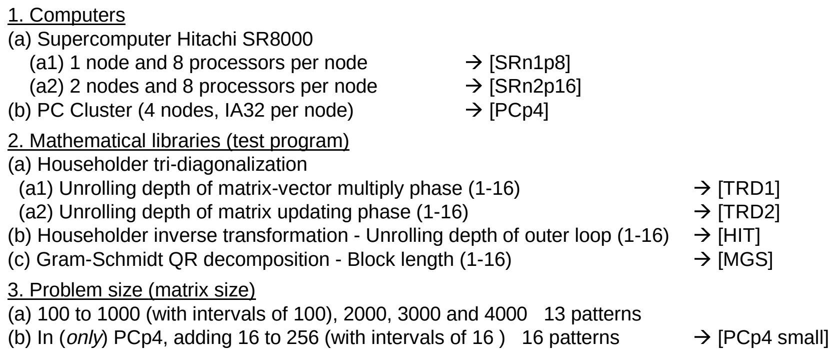
Fig.6e, Phase 4, includes the new sampling point #8. Now the estimated optimal value is #9, which is a new sampling point.

Fig.6f, Phase 5, includes the new sampling point #9. By the same procedure, the new sampling point selected is #6. The figure shows that fitting by *d-Spline* is similar to the dotted line for a set of all executed values. But, because the dotted line (the actual answer) is unknown, we continue to increment new sampling points.

Fig.6g shows that a new sampling point #13 is included in Phase 9. Twelve sampling points out of sixteen have already been selected. The estimated optimal value is #6, which has been the same for five successive times. The optimal value #6 was the same for the remaining four phases.

5 Application to Automatic Performance Tuning

Fig.7 shows the experiment parameters and their notations. Some experiments to evaluate the performance of our method are made on the Hitachi SR8000 and our PC cluster. The Hitachi SR8000 has 2 nodes with 8 processors per node and the compiler is the Hitachi OFORT90 version V01-04-/B with a compiler option



1. Computers	
(a) Supercomputer Hitachi SR8000	
(a1) 1 node and 8 processors per node	→ [SRn1p8]
(a2) 2 nodes and 8 processors per node	→ [SRn2p16]
(b) PC Cluster (4 nodes, IA32 per node)	→ [PCp4]
2. Mathematical libraries (test program)	
(a) Householder tri-diagonalization	
(a1) Unrolling depth of matrix-vector multiply phase (1-16)	→ [TRD1]
(a2) Unrolling depth of matrix updating phase (1-16)	→ [TRD2]
(b) Householder inverse transformation - Unrolling depth of outer loop (1-16)	→ [HIT]
(c) Gram-Schmidt QR decomposition - Block length (1-16)	→ [MGS]
3. Problem size (matrix size)	
(a) 100 to 1000 (with intervals of 100), 2000, 3000 and 4000 13 patterns	
(b) In (*only*) PCp4, adding 16 to 256 (with intervals of 16) 16 patterns	→ [PCp4 small]

Fig. 7. Experiment Parameters and their Notations

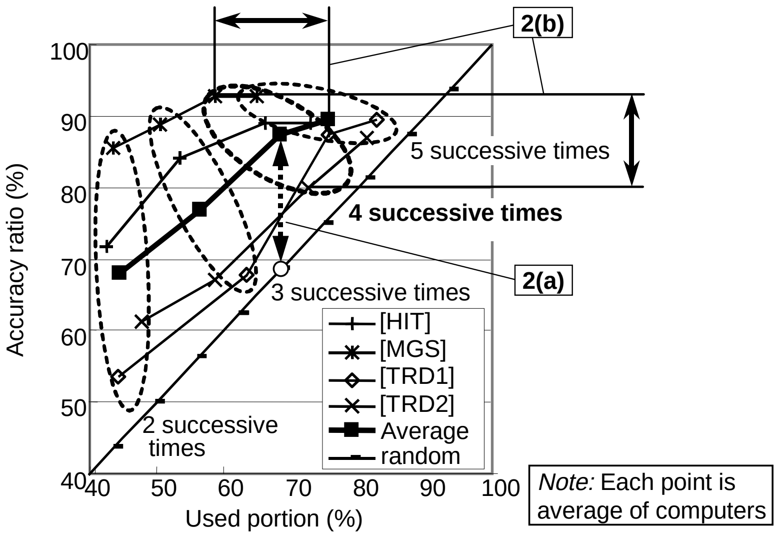


Fig. 8. Experimental Results–Mathematical Libraries

of -O4. Our PC cluster has 4 nodes with a single IA32 processor per node and the compiler is the PGI Fortran90 version 4.0-2 with a compiler option of -Fast.

The test programs are Householder tridiagonalization (*TRD1* and *TRD2* with different performance parameters), Householder inverse transformation (*HIT*) and Modified Gram-Schmidt QR decomposition (*MGS*). The problem sizes are from 100 to 1000 (strides of 100), 2000, 3000, and 4000. The total number of test patterns for our evaluation amounted to 216. The ABCLib [3] is used.

Fig.8 shows the results of our experiments in terms of the mathematical libraries. In X-axis the used portion is a ratio of the number of sampling points *per* number of parameter values. In Y-axis the accuracy ratio represents the correctly selected optimal value. Each line is the result of each mathematical library. Each point is the average of the results from four tested computers. The bold line is the average of four mathematical library lines. Each ellipse is a group with the same criteria of termination, which is indicated by the number of succession. The diagonal line represents sampling points selected at random. The following are our observations of the Fig.8:

1. All five lines are above the random line, which shows the usefulness of the *IPPE method*,
2. In the ellipse where the criterion of termination is four successive times:
   (a) The square point is the average of four mathematical library points, whose used portion is 68%, and the accuracy ratio was higher by 20% compared with the circle point on the random line. This shows the effectiveness of *IPPE method*,
   (b) The range of accuracy ratio increased from 80% to 92% while the range of used portion increased from 58% to 72%. To achieve an accuracy ratio

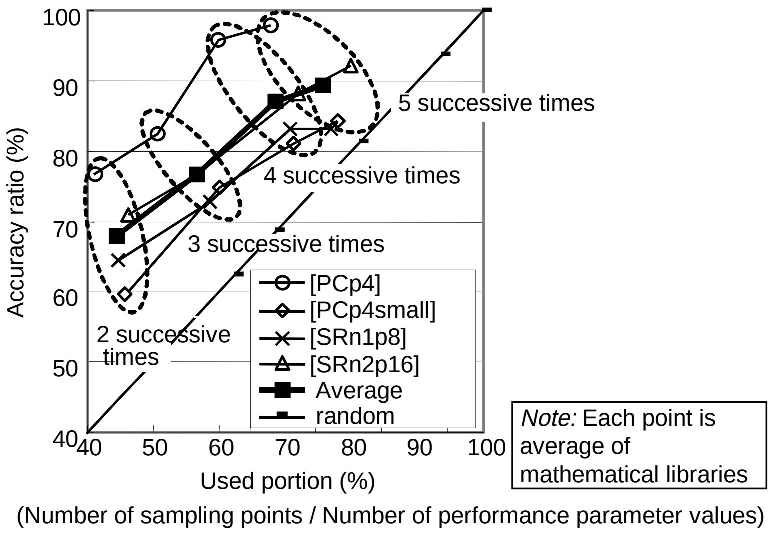


Fig. 9. Experimental Results–Computers

of 90%, in the case of *MGS*, the used portion was less than 60%. On the other hand, in case of *TRD1* and *TRD2*, it was more than 70%. If the used portion is fixed at 68%, the used portion is more than necessary for *MGS*, while *TRD1/TRD2* get less accuracy. Therefore, using the number of succession for the criteria of termination is more efficient than fixing used portion, or fixing the number of sampling points.

Fig.9 shows the results of our experiments in terms of computers. Each line is the average of mathematical libraries. We found:

1. All five lines were above the random line, which shows the usefulness of *IPPE method*,
2. The problem size or matrix size can be compared between *PCp4* and *PCp4 small*. The results of *PCp4* was better than *PCp4small*. This means the *IPPE method* is more useful in the case of large problem sizes.

Fig.10 is the histogram result when the selected optimal values were not correct in our experiments. When the criterion of termination was four successive times, the selected optimal values were incorrect for 28 patterns out of 216 patterns. The X-axis is the ratio of increased execution time for incorrectly estimated values *vs* the execution time for correct optimal values. The Y-axis is the number of patterns. The ratio of the increased execution time was 4.0% on average and 12.6% at the maximum, which were not considerable.

To summarize, the results of our experiments show the following advantages of the *IPPE method*:

1) Achieves high accuracy with relatively low usage portion,

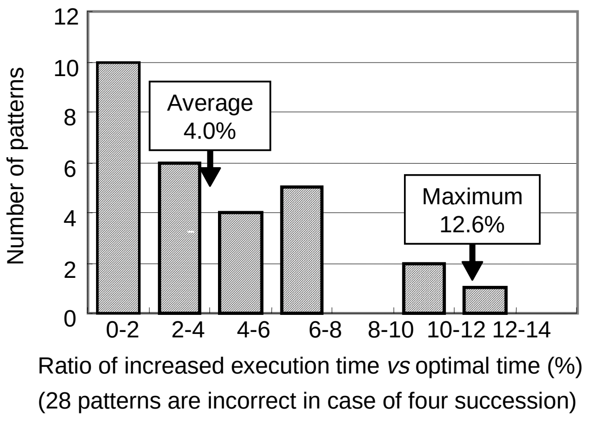


Fig. 10. Experimental Results–False Selection

2) Achieves high accuracy in all cases,
3) High efficiency for large problem sizes.

6 Conclusion

This paper presented the results of our study on performance parameter estimation in automatic parameter tuning. First, we proposed the Incremental Performance Parameter Estimation (*IPPE*) method. Then we introduced the fitting function *d-Spline*. Finally, the evaluation of the *IPPE method* by 216 patterns of mathematical libraries showed the effectiveness of the criteria for termination and the selection of sampling points.

Our future work will include: 1) Enhancement of *d-Spline* for multi-dimensional simultaneous evaluation of plural performance parameters, and 2) Application of *d-Spline* to estimate performance parameters for the interpolation of matrix sizes or problem sizes.

References

1. Bilmes, J., Asanovi, K., Chin, C.-W., Demmel, J.: Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In: Proceedings of International Conference on Supercomputing, vol. 97, pp. 340–347 (1997)
2. Whaley, R., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27, 3–35 (2001)
3. Katagiri, T., Kise, K., Honda, H., Yuba, T.: FIBER: A general framework for autotuning software. In: Veidenbaum, A., Joe, K., Amano, H., Aiso, H. (eds.) ISHPC 2003. LNCS, vol. 2858, pp. 146–159. Springer, Heidelberg (2003)
4. Tanaka, T., Tanabe, K.: A data fitting applying Bayes method, Kokyuroku, Research Institute for Mathematical Sciences, Kyoto Univ. 483 (in Japanese) (1983)
5. Tanaka, T.: Givens method and Householder method of solution for sparse least squares problem. The Research Report of the Institute of Statistical Mathematics (in Japanese) pp. 30–32 (1983)

A High Performance Generic Scientific Simulation Environment

René Heinzl, Michael Spevak, Philipp Schwaha, and Siegfried Selberherr

Institute for Microelectronics, TU Wien,
Gusshausstrasse 27-29, 1040 Wien, Austria
heinzl@iue.tuwien.ac.at

Abstract. A generic scientific simulation environment is presented which imposes minimal restriction regarding topological, dimensional, and functional issues. Therewith complete discretization schemes based on finite volumes or finite elements can be expressed directly in C++. This work presents our multi-paradigm approach, our generic libraries, some applications based on these libraries, and performance aspects.

1 Introduction

In the last decades numerous software environments and libraries have been developed to handle the many areas in the field of scientific computing. Due to the diversity of the mathematical structures, combined with efficiency considerations, the development of high performance simulation software is quite challenging. These challenges are becoming more difficult to meet, when the purpose of the software is to validate novel algorithms and complex methods, or to investigate physical phenomena that have not yet been fully understood. High performance computations have turned the attention especially to C++, since Blitz++ has shown that the run-time behavior is comparable to Fortran [1], the traditional language for scientific computing. In addition, distinct programming paradigms and their respective advantages can be combined in a multi-paradigm language, such as C++.

2 Motivation

Many library approaches [1,2,3] focus on topics such as expression templates, high performance matrix operations and calculations, and discretization of differential operators. The nature of dealing with different types of partial differential equations (PDEs) with the inherent coupling of topological traversion and functional description complicates the use of these libraries.

Our main area of work is focused on Technology Computer-Aided Design (TCAD), which deals with the assembly of large equation systems by utilizing discretized partial differential equations from different fields of physics. All types of PDEs (parabolic, elliptic, hyperbolic) and their discretization schemes such as the finite element method [4] or the finite volume method [5] have to be considered for the diverse types of problems. Types of topological cell complexes, different dimensions, and solving strategies have to be considered during application development [5,6,7]. Most of these applications use data structures such as `list` and `array` as well as triangles, quadrilaterals, tetrahedra, cuboids, each with their own access mechanisms, traversal operations, and data storage.

These issues demand great care to ensure high software quality while also addressing performance issues, when source code is being written. This is the primary motivation to develop generic libraries for high performance applications in the field of scientific computing. Generic library design deals with the conceptual categorization of computational domains, the reduction of algorithms to their minimal conceptual requirements, and strict performance guarantees. The benefits of this approach are the re-usability and the orthogonality of the resulting software.

3 Related Work

Various research groups have put a lot of effort into the development of libraries for sub-problems occurring in scientific computing. We briefly review the most important library approaches suitable for application design:

– The Boost Graph Library (BGL [2]) is a generic approach to the topic of graph handling and traversal with a standardized generic interface.
– The Computational Geometry Algorithms Library (CGAL [8]) implements generic classes and procedures for geometric computing with generic programming techniques.
– The Grid Algorithms Library (GrAL [9]) is a generic library for grid and mesh data structures and algorithms operating on them.
– deal.II [3] provides a framework for finite element methods and adaptive refinement for finite elements.
– ExPDE [10] collects efficient high-performance libraries for PDEs using the C++ technique of expression templates

Our analysis has revealed that, up to now, no related work can be used directly. All of these libraries have not been developed with emphasis on interoperability. This issue complicates the transition from one library to another. Therefore, our approach (Section 4.1) introduces a common layer with data structure definition and access routines, where all of these libraries can be used. With the generic programming paradigm and the implementation with templates in C++ the abstraction penalty can be minimized (Section 6).

Deal.II and the ExPDE library collection offer support for application design in the field of scientific computing. These libraries are an important step into library centric application design. But, as mentioned before, none of these libraries were developed with interoperability as a necessary constraint. As a consequence, additional code has to be introduced which slows the development process down and impedes the execution speed of the final application.

4 The GSSE

Based on the experience of developing high performance applications a *generic scientific simulation environment (GSSE)* with an overall high performance was developed, which does not impose restrictions on topological treatment or functional description. Different programming paradigms were used for the non-trivial and highly complex scenario of scientific computing. The generic programming paradigm realized by the template mechanism offers homogeneous interfaces between algorithms and data structures by the iterator/cursor pattern. Functional programming enables an efficient means to specify equations and offers extensible expressions.

Investigations of the applications developed at our institute have shown that the topological structures can be abstracted and generalized into a generic topology library (GTL), which is presented in more detail in Section 4.1. This new approach of a generalized topology enables a new functional description for the discretization of PDEs without sacrificing any performance. This is accomplished in our generic discretization library (GDL, Section 4.2), which fulfills the requirements for scientific computing, especially TCAD [11].

To introduce the base libraries of the GSSE, we examine different topological traversal operations without any assumption about the dimension. This dimension independent programming eases software development considerably and reduces the probability of errors.

4.1 Generic Topology Library: GTL

Topological functionality is mostly needed to fulfill the requirements of discretization schemes. All these schemes need a set of neighboring elements based on the topological property of **incidence**. Therefore the GTL [12] provides comprehensive incidence traversal and orientation operations with a generic interface similar to the C++ STL [13] and is based on GrAL [9]. Problems which can be formulated only with difficulty using existing libraries, can thereby be handled easily.

The following example presents an incidence traversal mechanism starting with an arbitrary cell iterator which is evaluated on a cell complex (grid). Then a vertex on cell iterator is initialized with a cell of the cell complex (cell container). The topological traversal is started with the `for` loop. During this loop an edge on the vertex iterator is created and initialized with the evaluated vertex. This edge iterator starts the next topological traversal. The `valid()` mechanism is used, because there is no `end()` iterator on inter dimensional objects.

```
cell_iterator  c_it = cell_complex.cell_begin();
for(cell_vertex  voc_it(*c_it); voc_it.valid(); ++voc_it)   {
  for(vertex_edge eov_it(*voc_it); eov_it.valid(); ++eov_it)  {
     //operations on edges
  }
}
```

4.2 Generic Discretization Library: GDL

One of the base operations in the context of PDEs is the assembly procedure of discretized equations into a matrix. To ease this procedure the GDL was developed. This library implements the ability to specify whole equations in a concise yet expressive way. The availability of a topology or traversal library, e.g. the GTL, is a fundamental requirement to specify equations in a functional way. Together these two libraries offer mechanisms to separate discrete mathematics into topological and numerical operations. To present the application of the GDL, we examine a simple equation:

$$\sum_{\mathrm{v} \to \mathrm{e}} (\Delta_{\mathrm{e} \to \mathrm{v}} \text{ quan}) = 0$$

where $\mathrm{v} \to \mathrm{e}$ denotes the traversal of all edges incident to the vertex, $\mathrm{e} \to \mathrm{v}$ denotes the traversal of the vertices incident to the edge, quan denotes the quantity located on a vertex, and Δ denotes the difference of this quantity.

The implementation with the GTL and the GDL without any dependence on the dimension or the type of cell complex is presented in the next code snippet. The addition over all edges incident to the given vertex are traversed by the `vertex_edge` expression with the `sum` functor. All elements have a standard local orientation, e.g. an edge provides a source and a sink vertex. We define an orientation function $\mathcal{O}(a, b)$ between an edge and a vertex, which returns $+1$ if a vertex coincides with the source and -1 if the vertex coincides with the sink (Figure 1).

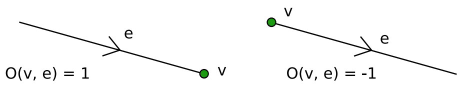


Fig. 1. Orientation of an edge. The orientation function returns either +1 or −1 depending on whether the vertex is the sink or the source of the edge.

Finally, the `_e` and `_1` are local variables (placeholders). The `_e` variable passes the edge into the next context `[ quan * orient(_1, _e) ]`, and `_1` stands for the local vertex.

```
for (v_it  = cell_complex.vertex_begin();
     v_it != cell_complex.vertex_end(); ++v_it)  {
  equation = sum<vertex_edge>
  [
    sum<edge_vertex>(0.0, _e) [ quan * orient(_1, _e) ]
  ](*v_it)
 // .. evaluate the equation object
}
```

The complex resulting from this mapping is completed by specifying the current vertex object `*v_it` at run-time, which clearly demonstrates the compile-time and run-time border. The datatype `equation` is explained in more detail in Section 4.4.

4.3 Matrix Assembly

Differential equations are discretized on elements of the topological structure and entered in a matrix by assembly methods. In general there exist two main methods of equation assembly which differ in the type of sub-matrices assembled in one step.

Element-wise assembly (Figure 2) is typically used in the finite element method. All cells (finite elements) are traversed and for each of the cells a local matrix is calculated. This matrix introduces coupling factors between the various shape functions. As each shape function is mapped to an element of the underlying cell complex, couplings between functions can be seen as couplings between values on elements. The local matrix entries are written into the global matrix according to a global vertex/cell numbering scheme. The finite volume scheme uses a different technique of matrix assembly (Figure 3). The differential equation is discretized on a vertex of the simulation domain. Couplings to other values are described by sums over topological elements. One of the major advantages of this method is that each degree of freedom causing a matrix entry has its own governing equation. This also implies that the matrix regions of assembly are disjoint, which allows a larger degree of parallelization, because each assembling element has exclusive access to matrix lines.

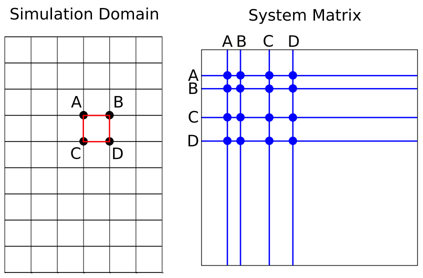


Fig. 2. Element-wise assembly. All cells are traversed and sub-matrices are calculated. The sub-matrices are inserted into the system matrix.

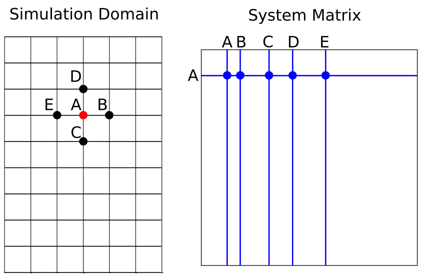


Fig. 3. Line-wise assembly. All vertices are traversed and linear equations are assembled. Linear equations are inserted into the system matrix.

4.4 Linear Functions

In order to simplify the line-wise assembly method, e.g. for finite volumes, we introduce the notion of a linear function data type. We consider some differential operator $\mathcal{L}(\psi)$ and the differential equation $\mathcal{L}(\psi) = 0$. Figure 4 shows a one-dimensional simulation domain with a quantity distribution. While the depicted values are not the solution of the considered equation, the residual can be determined by using the finite volume formulation. We consider not only the residual value but also the effects of linear changes of single values on the residuum. In

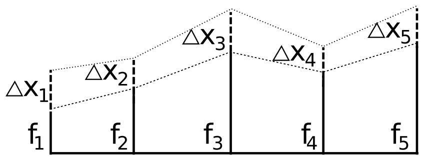


Fig. 4. Linear equation. The linear equation data structure stores the value as well as the dependence on the unknown variables Δx_i.

order to provide an environment which is able to handle small variations, we assign an index i to all variables. The small variations are called Δx_i and the function value is denoted by ψ_i. A function value including dependencies thus yields $\psi_i + \Delta x_i$. Based on these considerations we introduce addition.

$$(\psi_i + \Delta x_i) + (\psi_j + \Delta x_j) = (\psi_i + \psi_j) + (\Delta x_i + \Delta x_j)\ . \tag{1}$$

In order to define a closed algebraic structure, we use the concept of a linear function. This function can be written in the following manner

$$\sum_i a_i \cdot \Delta x_i + c_i\ . \tag{2}$$

The relations described in Equation 1 can be generalized to algebraic expressions. A data structure which implements these operations consequently, provides the following property: specifying the residuum and replacing the function value ψ_i by $\psi_i + \Delta x_i$ results in a linear equation for the values of Δx_i.

The following example uses finite volume schemes in order to assemble a simple Laplace equation. We use the linear equation data type instead of the standard numerical types (e.g. `double`) and replace the function values by linear equations. The constructor of the linear equation contains the value ψ_i as well as the index i and returns an equation object with the meaning $\psi_i + \Delta x_i$. The following code snippet presents an application of the introduced linear function concept to obtain the formulation of the Laplace equation:

```
linear_equation laplace_eqn;
vertex_edge eov_it(vertex)
for(; eov_it.valid(); ++eov_it) {
 linear_equation  equation;
 edge_vertex  voe_it(*eov_it);

 for(; v_it.valid(); ++v_it)
 {
   equation += linear_equation(f(*v_it), i(*v_it)) *
                                  orient(*v_it, *eov_it);
 }
 equation *= A(*eov_it) / d(*eov_it);
 laplace_equ += equation;
}
```

The linear equation can also be specified using functional programming.

```
laplace_equ = sum<vertex_edge>
[ sum<edge_vertex>(_e)
  [ lineqn(psi(_1), i(_1)) * orient(_1, _e)
  ] *  A(_1) / d(_1)
](vertex);
```

5 Application Design Using the GSSE

We demonstrate the development of simple applications which solve partial differential equations based on the facilities of the GTL and the GDL. These equations can be divided into elliptic equations, such as the Laplace or Poisson equation, parabolic equations which describe diffusive processes, and hyperbolic equations, such as wave equations. Examples for each of these equations are often found in TCAD. The first example presents the capability of reducing complex discretization to simple topological iterations, mostly based on the GTL. The second example utilizes the GDL based on a piecewise construction of functional parts to form a complex equation.

5.1 Maxwell's Equation

We present an implementation using Yee's algorithm for Maxwell's equations [14] in the following. While the Yee formulation makes use of staggered grids, the application on structured topologies based on the GTL and inter-dimensional iterators causes an enormous simplification. Instead of special grids, we employ different dimensional elements such as edges and faces for the representation of the electrical field strength and magnetic field. It turns out that the tensorial character of the quantities fits into the dimensionality concept of the topological elements. We present the special case of a transversal magnetic mode only. The following formulation can be derived by applying the Yee discretization scheme.

$$\begin{aligned} E_z^{n+1}(i,j) = E_z^n(i,j) & + \Delta t \, \frac{l}{\Delta x}[H_y^{n+1/2}(i+\frac{1}{2},j) - H_y^{n+1/2}(i-\frac{1}{2},j) \\ & - \Delta t \, \frac{l}{\Delta y}[H_x^{n+1/2}(i,j+\frac{1}{2}) - H_x^{n+1/2}(i,j-\frac{1}{2}) \end{aligned}$$

With the transfer of all index calculations to topological iteration and traversal mechanisms, e.g. the electric field quantity to edges E_e and the magnetic field quantity to facets H_f the formula can be rewritten as:

$$E_\mathrm{e} - E_\mathrm{e}^\mathrm{old} = \Delta t \, l_\mathrm{e} \Delta_{\mathrm{e}\to\mathrm{f}} [\frac{H_\mathrm{f}}{A_\mathrm{f}}],$$

where $\mathrm{e} \to \mathrm{f}$ denotes the traversal of all facets incident to the edges, and A_f represents the area of the corresponding facet. The evaluation of all quantities on their corresponding dimension and topological objects is completed automatically. The final source code is presented in the following code snippet. The minimal requirement to specify such complex equations can be seen clearly.

```
equation_E += dt * l * sum<edge_facet >(0.0, _e)
  [
     H / A * orient(_e, _1)
  ]
```

5.2 Applications for Device Simulation

Good understanding of electron transport in semiconductors is one key requirement for the progress in microelectronic devices. For device simulation we present the drift-diffusion model which reads in a simple form for electrons [5]:

$$\begin{aligned} \mathrm{div}\,(\varepsilon\,\mathrm{grad}\,(\Psi)) &= \mathrm{q}\,(n - N_{\mathrm{D}}) \\ \mathrm{div}\,(\mathbf{J}_{\mathrm{n}}) - \mathrm{q}\,\partial_t n &= \mathrm{q}R \\ \mathbf{J}_{\mathrm{n}} &= \mathrm{q}\,n\,\mu_{\mathrm{n}}\,\mathrm{grad}\,(\Psi) + \mathrm{q}\,D_{\mathrm{n}}\,\mathrm{grad}\,(n) \end{aligned} \tag{3}$$

Discretizing the continuity relation (as presented in Equation 3) using the Scharfetter-Gummel scheme [15] results in:

$$\sum_{\mathrm{v}\to\mathrm{e}} \mathrm{q}\,\mu_{\mathrm{n}} U_{\mathrm{t}} \frac{A}{d} \sum_{\mathrm{e}\to\mathrm{v}} n\,\mathrm{Bern}[\sum_{\mathrm{e}\to\mathrm{v}} \frac{\psi}{U_{\mathrm{t}}}] \tag{4}$$

$\mathrm{Bern}(x) = x/(e^x - 1)$ represents the Bernoulli function and U_{t} describes the thermal voltage. This discretized form can be transformed into C++ code using our formalism to yield:

```
// Poisson equation
equation_poiss=
sum<vertex_edge>
[ A / d * eps *
  sum<edge_vertex >(0.0, _e) [pot*orient(_1, _e)]
] - q*(n-nD)

// Continuity equation for electrons
equation_n = sum<vertex_edge>
[ q * mu_n * U_t * A / d  *
  sum<edge_vertex >(0.0,_e)
  [
    orient(_e, _1)*n(_1)*
    Bern(locate(_e) [sum<edge_vertex >[pot*orient(_1,_e)]/U_t])
]]
```

Due to the functional specification, a special mechanism has to be introduced `locate(_e)` to obtain the edge information in the innermost loop. The main problem with this type of programming in C++ is that the expression between `[]` opens up a new scope and new local variables. The `locate` function passes the edge information from the second `sum` to the `Bern` function.

5.3 Number of Code Lines

To implement a complete application, one line of code is used to import a given cell complex (mesh) from a file and another line is used to assemble the linearized functions into a generic matrix interface. The overall number of lines of code to implement an application can thereby be greatly reduced. This results in a minimization of maintenance of source code as well as the learning time for new developers.

6 Performance

The basic parts of how to achieve high performance in C++ are based on the usage of templates. Therewith the compiler's data-type-based function selection at compile time leading to a global optimization with inlined function blocks is possible. Additionally lightweight object optimization [16] with frequent allocation to registers is made available.

To circumvent the problems with benchmarking different techniques, we restrict the performance analysis on a simple but often used detail, namely the addition of several small matrices which occur in the discretization schemes described before. This simple expression can be compared to other benchmark studies [17]. The test is performed using the vector addition $A_f = A_b + A_c + A_d$, evaluated with different vector sizes on a Pentium 4 (2.4 GHz) with the GCC 4.1.0. Several approaches, a naive C++ implementation with `std::vector<T>`, Blitz++, a simple version of expression templates [18] , the C++ `std::valarray`, and finally the GDL approach are compared to a hand-optimized Fortran 77 implementation (F77) as can be seen in Figure 5 . Although functional and generic programming

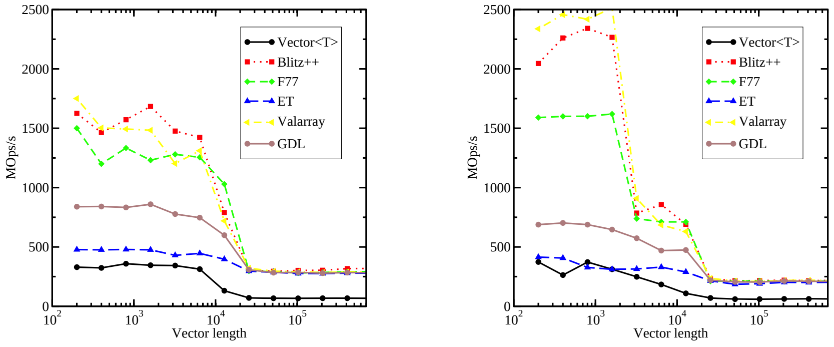


Fig. 5. Performance of the evaluated expression on a P4 (left) and an AMD64 (right)

support the parallelization significantly, the currently used computer technology restricts this effort due to their architecture. For vector lengths smaller than 10^4, cache hits reveal the full computation power of the CPU, longer vectors show the limits imposed by memory band width.

7 Conclusion

The application of several modern programming paradigms solves the problem of portability while insuring high performance by providing orthogonal means of optimization. Currently no language other than C++ offers sufficient support for all the necessary programming paradigms to enable this high-level abstraction at unmatched performance. As we have demonstrated in the complex field of TCAD, applications can be developed with a reasonable amount of effort.

References

1. Veldhuizen, T.L.: Using C++ Template Metaprograms. C++ Report 7(4), 36–43 (1995) (Reprinted in C++ Gems, S. Lippman (ed.))
2. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, London (2002)
3. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II Differential Equations Analysis Library, Technical Reference, `http://www.dealii.org/`
4. Zienkiewicz, O.C., Taylor, R.L.: The Finite Element Method. McGraw-Hill, Berkshire, England (1987)
5. Selberherr, S.: Analysis and Simulation of Semiconductor Devices. Springer, Heidelberg (1984)
6. Sabelka, R., Selberherr, S.: A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. Microelectronics Journal 32(2), 163–171 (2001)
7. IμE: MINIMOS-NT 2.1 User's Guide. Institut für Mikroelektronik, Technische Universität Wien, Austria (2004), `http://www.iue.tuwien.ac.at/software/minimos-nt`
8. Fabri, A.: CGAL- The Computational Geometry Algorithm Library (2001), `http://citeseer.ist.psu.edu/fabri01cgal.html`
9. Berti, G.: GrAL - The Grid Algorithms Library. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) Computational Science - ICCS 2002. LNCS, vol. 2331, pp. 745–754. Springer, Heidelberg (2002)
10. Härdtlein, J., Linke, A., Pflaum, C.: Fast Expression Templates - Object Oriented High Performance Programming. Eng. with Comput. 3515, 1055–1063 (2005)
11. Heinzl, R., Spevak, M., Schwaha, P., Grasser, T.: Concepts for High Performance Generic Scientific Computing. In: Proc. of the 5th MATHMOD, Vienna, Austria (2006)
12. Heinzl, R., Spevak, M., Schwaha, P., Selberherr, S.: A Generic Topology Library. In: Library Centric Sofware Design, OOPSLA, Portland, OR, USA, pp. 85–93 (2006)
13. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
14. Yee, K.S.: Numerical Solution of Initial Boundary Value Problems involving Maxwell's Equations in Isotropic Media. IEEE Trans. Antennas and Propagation 14(1), 302–307 (1966)
15. Scharfetter, D., Gummel, H.: Large-Signal Analysis of a Silicon Read Diode Oscillator. IEEE Trans. Electron Dev. 16(1), 64–77 (1969)
16. Siek, J., Lumsdaine, A.: Mayfly: A Pattern for Lightweight Generic Interfaces. In: Pattern Languages of Programs (1999)
17. Heinzl, R., Schwaha, P., Spevak, M., Grasser, T.: Performance Aspects of a DSEL for Scientific Computing with C++. In: Proc. of the POOSC Conf., Nantes, France, pp. 37–41 (2006)
18. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley, London (2004)

NestStepModelica – Mathematical Modeling and Bulk-Synchronous Parallel Simulation

Christoph Kessler, Peter Fritzson, and Mattias Eriksson

PELAB Programming Environments Lab, Dept. of Computer Science,
Linköping University, SE-581 83 Linköping, Sweden
{chrke, petfr, mater}@ida.liu.se

Abstract. Many parallel computing applications are used for simulation of complex engineering applications and/or for visualization. To handle their complexity, there is a need for raising the level of abstraction in specifying such applications using high level mathematical modeling techniques, such as the Modelica language and technology. However, with the increased complexity of modeled systems, it becomes increasingly important to use today's and tomorrow's parallel hardware efficiently. Automatic parallelization is convenient, but may need to be combined with easy-to-use methods for parallel programming.

In this context, we propose to combine the abstraction power of Modelica with support for shared memory bulk-synchronous parallel programming including nested parallelism (NestStepModelica), which is both flexible (can be mapped to many different parallel architectures) and simple (offers a shared address space, structured parallelism, deterministic computation, and is deadlock-free). We describe NestStepModelica and report on first results obtained with a prototype implementation.

1 Introduction to Mathematical Modeling and Modelica

Modelica is a modern language for equation-based object-oriented mathematical modeling which is being developed through an international effort [4,9]. It allows defining simulation models in a declarative manner, modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism. The multidomain capability of Modelica allows combining electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model.

To summarize, Modelica has improvements in several important areas:

- *Object-oriented mathematical modeling.* This technique makes it possible to create model components, supporting hierarchical structuring, re-use, and evolution of large and complex models covering multiple technology domains.
- *Acausal modeling.* Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in which they are used. However, for interfacing with traditional software, algorithm sections with assignments as well as external functions/procedures are also available in Modelica.

– *Physical modeling of multiple application domains.* Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. In Modelica the structure of the model becomes more natural in contrast to block-oriented modeling tools. For application engineers, such "physical" components are particularly easy to combine into simulation models using a graphical editor.

2 Integrating Parallelism and Mathematical Models

There are several levels of parallelism to be exploited in simulation code generated from high-level mathematical models:

– *Parallelism over the method.* One approach is to adapt the numerical solver for parallel computation, *i.e.*, to exploit parallelism over the method, for example, by using a parallel ordinary differential equation (ODE) solver (see e.g. Korch and Rauber [7]) that allows computation of several intermediate results in parallel. However, at least for ODE solvers, only limited parallelism is available. Also, the numerical stability can decrease by such parallelization.
– *Parallelism over time.* A second alternative is to parallelize the simulation over the simulated time. This is however best suited for discrete event simulations, since solutions to continuous time dependent equation systems develop sequentially over time, where each new solution step depends on the immediately preceding steps.
– *Parallelism over the system.* This means that the computations given in the model equations derived from the high-level model specification are distributed across several processors. For an ODE or DAE system, this usually means parallelization of the right-hand sides of such equation systems which are available in explicit form. Moreover, in many cases implicit equations can automatically be symbolically transformed into explicit form.
– *Parallelism over the model.* Under certain conditions, the model of a system can be split into loosely coupled submodels that are simulated in parallel on different processor subsets. This model distribution technique is known as *transmission line modeling, TLM.* This approach is justified if the connectors between submodels correspond to physical interactions that incur effect delays, which are included in the TLM coupling equations in a way that communication delays in exchanging data between submodel simulations at simulation time approximately match similar delays in simulated time, leading to numerically stable simulations without increased inaccuracy [10].

The following approaches are being investigated in the context of parallel simulation of Modelica models.

(1) *Automatic parallelization.* One obstacle to parallelization of traditional computational codes is the prevalence of low-level implementation details in such codes, which also makes automatic parallelization hard. Instead, it would be attractive to directly extract parallelism from the high-level mathematical

model, or from the numerical method(s) used for solving the problem. Aronsson and Fritzson [1] investigated the automatic parallelization of computations in the right-hand side of the generated model equation system, using techniques for fine-grained task scheduling.

As automatic parallelization methods have their limits, the environment should enable the programmer to explicitly expose (additional) parallelism.

(2) *Model distribution with TLM.* A natural idea is to let the programmer structure the application—if possible, already at the model level—into components representing submodels, using strongly-typed communication interfaces for loose couplings between submodels (TLM connectors). Ongoing work using this approach for Modelica is reported by Nyström *et al.* [10].

(3) *GridNestStep.* Mattsson and Kessler [8] proposed an explicit parallel programming language that provides bulk-synchronous parallel computing with a global address space on computational grids. In the Modelica domain, this framework is mainly intended to support the implementation of parallel solvers on a grid platform, thus exploiting parallelism over the method.

(4) *NestStepModelica.* Our fourth approach to soliciting parallelism is providing general easy-to-use explicit parallel programming constructs within the algorithmic part of the modeling language, thus also exploiting parallelism over the system that is embedded in `algorithm` sections and called functions. This is the approach we explore in this paper, with the NestStepModelica language embedded into the algorithmic part of the Modelica language.

3 Introduction to NestStepModelica

In this section we give a short overview of the most characteristic features of NestStepModelica and show an example program. For the interested reader we provide a survey of the language constructs in Section 4.

NestStepModelica is conceptually based on *NestStep* [5,6], which in turn is based on the BSP (Bulk-Synchronous Parallel) computation model [13]. The BSP model is an abstraction of a restricted message passing architecture and charges a cost for communication. It requires that the execution of parallel programs be organized as a sequence of *supersteps*. Each superstep consists of a phase of local computation where only process-local data can be accessed, followed by a global communication phase where processes first send and then receive values that will be needed later. Finally, a (conceptual) barrier synchronization marks the end of the superstep and the beginning of the next one.

NestStep is defined as a set of language extensions that may be added, with minor modifications, to any imperative programming language, be it procedural or object oriented. The sequential aspect of computation is inherited from the base language, which here is the algorithmic (non-equational) part of Modelica. Such `algorithm` sections of Modelica models encapsulate imperative specifications of model subcomputations that would be too complicated or too inefficient if coded in equation form. An `algorithm` section comprising the body of a `function` is executed when the `function` it belongs to is called. Such calls occur in conjunction with the solution of the equation system that the `function` belongs to. From the solver of the overall equation system, the `function` is viewed as a subsystem and is thus executed when needed, as given by data dependencies from/to the rest of the system of equations.

The new NestStepModelica language constructs provide shared variables and process coordination. NestStepModelica processes run, in general, on different processors (possibly on different machines) that are coupled by the NestStepModelica runtime system to a virtual parallel computer. Each processor executes one process with the same imperative fragment of a simulation program (SPMD), and the number of processes remains constant throughout the entire execution of that program fragment. Initially, all processes assigned for the execution of the program fragment form a single group. Processes are ranked (indexed) consecutively within a group from 0 upwards.

NestStepModelica enforces the superstep structure of the BSP model. A superstep is framed by the keywords `step` and `endstep`:

```
Integer k ( mem="shared");
...
step
   ...
   k = ... ;
   ...
endstep;
```

At any point of time, all processes of a process group must be working within the same superstep.

Variables declared by processes are private (process-local) by default, *i.e.*, each executing process creates its own copy and has exclusive access to it. Variables explicitly declared `shared`, such as `k` in the example above, generally exist in several copies, one per executing process (replication); write accesses during the local computation phase in the superstep may, of course, cause these copies to vary in contents, as each process only can access its local copy, but NestStepModelica guarantees that their contents is kept consistent at superstep boundaries. This means that, after passing the barrier corresponding to `endstep`, all copies of `k` on all processes will again have the same value—which one, can be programmed individually for each shared variable and for each superstep. For instance, an arbitrary written value may be committed to all copies, or the global sum or global maximum of all values written by the processes in the superstep could be computed and committed, as in the following code

```
Real maxerr ( mem="shared" );
...
step
   ...
   err = local_error( ... );
   ...
   endstepReduce ( result=err, op=Operators.max );
endstep;
```

where the global maximum of all values written to `err` is to be committed to all copies at the end of the superstep.

Moreover, even prefix sums computations can be performed as a side effect of a global sum computation. Technically, such reduction and prefix combinings come at virtually no extra cost, as the run-time system manages them pick-a-back on the interprocessor communication messages that are needed anyway for barrier synchronization and for restoring consistency of shared variables [5,6].

Example. We use the *parallel computation of prefix sums* as an example because it is simple and short enough to demonstrate the main ideas within a single page of code.

Figure 1 shows NestStepModelica code computing parallel prefix sums. The function `parPrefix` takes a distributed shared parameter array `x` of reals, which is distributed block-wise over the executing group of processes, and the maximum local problem size as an input parameter, and shall return a likewise block-wise distributed shared array `y` of the same extent as `x`. The function defines several (local) variables, all of which are private (exist once on each participating process) except `sum`, which is declared shared by all executing processes.

`parPrefix` consists of three supersteps. The first one contains a parallel loop that simply copies the contents of `x` to `y`; only locally accessible array elements are read and written by each process.

In the second superstep, each process computes prefix sums for the local partition of array `y` in the private array `prefix` and accumulates the sum of all local elements in the process-local copy of the shared variable `sum`. By `endstepReduce` the programmer dictates making the various write accesses to `sum` consistent by a plus-reduction where, in addition, each process gets the partial sum over the contributions by all processes with a rank less than the executing process written in its local variable `myPrefixSum`.

The third superstep sweeps once more over the local partition of array `y` and adds the offset value `myPrefixSum` to the process-relative values in `prefix` to derive the final prefix sums vector in `y`.

We notice that even in the imperative `algorithm` sections some restrictions of the functional programming core of Modelica apply. For instance, there are no global variables in Modelica. Functions must be free of side-effects, which makes it impossible to use naive approaches to compute e.g. the parallel prefix sums in place (*i.e.*, in the same array that contained the input data). However, an advanced optimizing compiler could actually perform such optimizations on function calls such as `vec := func(vec)` or similar, provided that analysis of the body of `func` shows that the semantics of its body is not dependent on the input vector and that the result vector is not aliased to the same memory.

```
function parPrefix  "Compute prefix sums in parallel"
  input  Real[:] x ( mem="shared", distr="block" );
               // x is a block-wise distributed shared array parameter
  input  Integer ndp;   // Process-private parameter
  output Real[size(x,1)] y ( mem="shared", distr="block" );
               // y is block-wise distributed shared array return value
protected
  Real[ndp] prefix;       // Process-private prefix array
  Real myPrefixSum;       // Prefix offset for this process
  Real sum(mem="shared"); // Shared variable
  Integer i,j;

algorithm

  step();   // First BSP superstep:
    // Iterate over local elements of y and copy contents of x to y:
    for i in localStart(y):localStride(y):localEnd(y) loop
      y[i] := x[i];
    end for;
    sum := 0;
  endstep();

  j := 1; // In Modelica lowest index for arrays is 1

  step();   // Second BSP superstep:
    for i in localStart(y):localStride(y):localEnd(y) loop
      prefix[j] := sum;
      sum := sum + y[i];
      j := j + 1;
    end for;
    endstepReduce( result=sum, op=Operators.plus, prefixVar=myPrefixSum );
  endstep();

  j := 1;

  step();   // Third BSP superstep:
    for i in localStart(y):localStride(y):localEnd(y) loop
      y[i] := prefix[j] + myPrefixSum;
      j := j + 1;
    end for;
  endstep();

end parPrefix;
```

Fig. 1. NestStepModelica code computing parallel prefix sums in parallel. (Note that in the current prototype implementation we simulate NestStepModelica keywords such as `step` and `endstep` by standard library functions instead; this workaround will be resolved in a final frontend implementation.)

4 Survey of NestStepModelica Language Constructs

4.1 Reflective Functions

`noProcessors()` gives the total number of processes allocated to this fragment.
`thisgroup()` returns a reference to an object providing information about the current group of the executing process. In particular,
`thisgroup().rank()` gives the rank of the executing process within its current group;
`thisgroup().gid()` gives the relative group ID (*i.e.*, its rank among all child groups of its parent group);
`thisgroup().size()` returns the number of processes in the current group;
`thisgroup().parent()` provides a reference to the parent group's group information object.

4.2 BSP Supersteps and Nested Parallelism

`step()` and `endstep()` denote the start resp. end of a superstep. These should be keywords but are currently realized as function calls, for pragmatic reasons.

Nested parallelism is desirable where massive parallelism should be exploited or where sufficient parallelism only can be solicited if parallelism-generating constructs are nested. Statically nested parallelism occurs *e.g.* for nested parallel loops, while dynamically nested parallelism is created *e.g.* by recursive parallel divide-and-conquer computations. In NestStepModelica, this is achieved by splitting a process group into disjoint subgroups that behave like disjoint BSP submachines—they execute the superstep(s) framed by `neststep` and `endneststep` independently of each other and in parallel. In particular, different subgroups may execute a different number of supersteps; processes restore consistency and barrier-synchronize only within their subgroup.
`neststep ( nsubgroups=`*intexpr* `)` denotes the start of a nested superstep, where the current process group is split into `nsubgroups` child groups of approximately equal size that execute the sub-superstep between `neststep` and `endneststep` independently and rejoin at `endneststep`. In the variant
`neststep ( nsubgroups=`*intexpr*`, subgrouptojoin=`*intexpr* `)` a process can dynamically determine the index of the subgroup it will join.
`endneststep()` denotes the end of a nested superstep where all processes barrier-synchronize and the original group is restored. Additionally, pending consistency-restoring operations to variables shared by all processes of the parent group are committed here.

4.3 Data Sharing and Array Distribution

`mem="shared"` is a shared memory allocation specifier for a variable or array that is to be shared between the processes of a group.
`distr="block"` is a type qualifier for block distribution of a shared array.
`distr="cyclic"` is a type qualifier for cyclic distribution of a shared array.

`localStart()` and `localEnd()` return the first and last locally available index in a distributed shared array, respectively;
`localStride()` returns the number of steps between two consecutive indices in a distributed shared array (this is always 1 for blockwise distributed arrays).

With `localStart()`, `localStride()`, and `localEnd()` available it becomes easy to express parallel loops as ordinary Modelica `for` loops, see Figure 1.

BSP-compliant mechanisms to access remote elements of distributed shared arrays, similar as in NestStep [6], are provided but omitted here for lack of space.

4.4 Combining

At the end of a BSP superstep, *i.e.*, at the conceptual barrier synchronization, NestStepModelica enforces that all copies of shared variables on the processes of a group contain the same value. How the consistency of possibly different values in these copies (which are caused by write operations within the superstep) should be made consistent again can be programmed individually for each shared variable and superstep, according to the following predefined policies for resolving (concurrent) write operations:

- *Arbitrary Concurrent Write BSP* A value written by any of the processes will be chosen and committed to all copies of that shared variable. This is the default policy.
- *Reduction-Combine Concurrent Write BSP*
- *Prefix-Combine Concurrent Write BSP*

`endstepReduce ( result, op )` specifies for individual shared variables or array sections how concurrent write conflicts should be resolved by use of a binary associative reduction operator `op` at the end of this superstep. Such operators are predefined in the Modelica standard library.
`endstepReduce ( result, op, prefixVar )` specifies that, as a side effect of the reduction computation, prefix-`op` results are returned to the private variable `prefixVar` supplied as output parameter, where the process with rank i obtains the global `op` over the `result` values written by processes ranked 0, ..., $i-1$.

5 Some Experimental Results

At the time of writing, the run-time system of NestStepModelica [12] is operational; it is implemented on top of MPI and the Tlib library which enhances process group management for nested SPMD parallelism in message passing programs [11]. A NestStepModelica front end (compiler) is under development as an extension of the Modelica language within the OpenModelica framework [3]. Currently, flat supersteps containing parallel loops over distributed shared arrays, such as in Figure 1, can be compiled; frontend support for nested parallelism is a subject of future work.

Figure 2 shows execution times and relative speedups on a Linux cluster with Xeon processors: for the parallel prefix sums program of Figure 1, run with $N = 100000$ and $N = 9,999,800$ array elements, and for Gaussian elimination without pivoting for a matrix of 5000×5000 elements.

Number of Processors	1	2	4	8	16	32
Parallel Prefix Sums, $N = 10^5$, Time [ms]	24.76	12.7	6.61	3.46	1.90	1.36
Parallel Prefix Sums, $N = 10^7$, Time [ms]	2470	1261	632	318	162	83
Gaussian Elimination, $N = 5000$, Time [ms]	2725	2788	1422	729	389	258

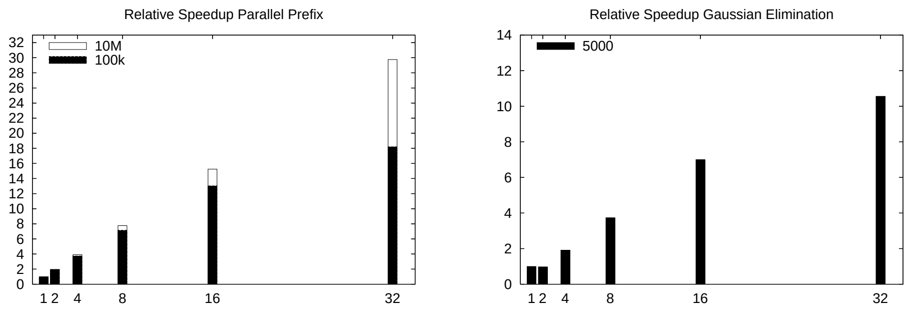


Fig. 2. Execution times and relative speedup on a Linux cluster for the parallel prefix program with 10^5 and 10^7 floats, and for Gaussian elimination with a 5000×5000 matrix without pivoting

6 Conclusion

Parallelization allows to simulate larger models in reasonable time. Moreover, the effective use of contemporary and future processor architectures makes parallelization a necessity anyway. Simulation code derived from equation-based mathematical models offers parallelism at several levels (method, time, system, and model), albeit each of these levels usually exhibits only very limited parallelism. Soliciting a sufficient amount of parallelism is therefore only possible if parallelization is done at several or all levels.

In this paper we described the NestStepModelica language and system that allows explicit parallel programming of heavy subcomputations in the model equation system, typically occurring on the right-hand side of the system. In addition, NestStepModelica could be used to program stand-alone imperative parallel computations within the Modelica environment. By marking parallelism explicitly with BSP supersteps, parallel loops, data sharing classifiers, and array distributions, automatic parallelization and scheduling methods at the equation system level, such as [1], can be complemented.

NestStepModelica offers several important features that allow for convenient expression of parallel computations: It offers a shared address space instead of message passing. Its simple but programmable memory consistency model is compliant with the superstep structure of the BSP model and thereby also coincides with the synchronization. Moreover, the BSP structure guarantees deadlock-free programs. Together this results in deterministic parallel program execution.

The explicitly parallel extensions in NestStepModelica should nevertheless be used with care, because each parallel computation incurs some overhead in the form of startup time and communication times. Therefore, only sufficiently heavy computations should be parallelized.

Finally, the parallel simulation program resulting from NestStepModelica can and should be complemented with other techniques for parallelization *e.g.* at the model or solver level to solicit more parallelism, which will result in a nested parallel simulation program.

Acknowledgments. This work was supported by Vinnova, project GRIDModelica, and the CUGS graduate school. We thank the National Supercomputing Center (NSC) at Linköping university for giving us access to their Linux cluster Monolith.

References

1. Aronsson, P., Fritzson, P.: Automatic Parallelization in OpenModelica. In: Proc. of 5th EUROSIM Congress on Modeling and Simulation, Paris (September 2004)
2. Bisseling, R.H.: Parallel Scientific Computation – A structured approach using BSP and MPI. Oxford University Press, Oxford (2004)
3. Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L., Broman, D.: The OpenModelica Modeling, Simulation, and Software Development Environment. Sim. News Europe 44/45 (December 2005), `www.ida.liu.se/projects/OpenModelica`
4. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press (2004), See also `www.mathcore.com/drModelica`
5. Keßler, C.: NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. The Journal of Supercomputing 17, 245–262 (2000)
6. Kessler, C.: Managing Distributed Shared Arrays in a Bulk-Synchronous Parallel Environment. Concurrency and Computation Pract. Exp. 16, 133–153 (2004)
7. Korch, M., Rauber, T.: Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. J. Parallel and Distr. Comput. 66, 444–468 (2006)
8. Mattsson, H., Kessler, C.: Towards a Bulk-Synchronous Distributed Shared Memory Programming Environment for Grids. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 519–526. Springer, Heidelberg (2006)
9. The Modelica Association. The Modelica Language Specification Version 2.2 (March 2005), `http://www.modelica.org`
10. Nyström, K., Fritzson, P.: Parallel Simulation with Transmission Lines in Modelica. In: Proc. 5th Int. Modelica Conf. (Modelica2006), Vienna, Austria (September 2006)
11. Rauber, T., Rünger, G.: Tlib: a library to support programming with hierarchical multi-processor tasks. J. Parallel and Distr. Comput. 65(3), 347–360 (2005)
12. Sohl, J.: A Scalable Run-time System for NestStep on Cluster Supercomputers. Master thesis LITH-IDA-EX-06/011-SE, Department of Computer Science, Linköping University, Linköping, Sweden (March 2006)
13. Valiant, L.: A Bridging Model for Parallel Computation. Comm. ACM 33(8) (August 1990)

Flecs, a Flexible Coupling Shell Application to Fluid-Structure Interaction*

Margreet Nool[1], Erik Jan Lingen[2], Aukje de Boer[3], and Hester Bijl[3]

[1] CWI, Department of Computing and Control,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`Margreet.Nool@cwi.nl`
[2] Habanera Software company, Delft, The Netherlands
[3] TU Delft, Faculty of Aerospace Engineering, Delft, The Netherlands

Abstract. Numerical simulations involving multiple, physically different domains can be solved effectively by coupling simulation programs, or *solvers*. The coordination of the different solvers is commonly handled by a *coupling shell*. A coupling shell synchronizes the execution of the solvers and handles the transfer of data from one physical domain to another. In this paper, we introduce FLECS, a flexible coupling shell, designed for implementing and applying an interface for multidisciplinary simulations with superior accuracy. The aim is not to achieve the best possible efficiency or to support a large feature set, but to provide a flexible platform for developing new data transfer algorithms and coupling schemes.

1 Introduction

Fluid-Structure Interaction (FSI) considers coupled fluid-solid problems, characterized by the interaction of fluid forces and structural deformations, which occur in many applications in industry and science. Nowadays, the simulation of FSI becomes more and more important, since future structures become lighter and more flexible and can be applied, e.g., to reduce the load on turbine blades, or, to reduce the noise on cars. Such applications require a real interdisciplinary approach, that can deal with complex physical models and very different scales.

The Faculty of Aerospace Engineering of the Delft University of Technology has started a project to develop a generic, open-source coupling shell, named FLECS [2], that can be used to join two or more arbitrary solvers. FLECS should provide an innovative combination of high order coupling in space and time. Moreover, to improve the accuracy and the efficiency of the computation, multilevel acceleration techniques for the coupling process [6], and fast prototyping and parallelization techniques will be supported.

The majority of coupling shells are embedded subprograms that have been developed for coupling two specific solvers. One exception is the coupling library

* Funding for this work was provided by the National Computing Facilities Foundation (NCF), under project numbers NRG-2005.03.

MpCCI (Mesh based Parallel Code Coupling) [4], which can be used as a separate program. Although MpCCI is relatively easy to use and provides many advanced features, it is less suitable for a scientific research community that is aimed at developing new data transfer algorithms. Numerical acceleration algorithms, like Krylov and multilevel methods - urgently required for efficiency - are not incorporated. Moreover, since MpCCI only provides the binary code, the user can not modify the implementation schedule of MpCCI.

In addition to accurate coupling in space, we want to reduce the partitioning errors in time by using specially designed high-order time integration methods [5]. It is our goal for Flecs to support solvers that run on parallel computers, in order to make Flecs suitable for large applications. In particular, Flecs must be able to deal with data sets that have been distributed over multiple parallel processes. In addition, Flecs should support the implementation of parallel data transfer algorithms. At the present stage of the project, experiments are limited to sequential solvers; each solver, running on its own processing unit, deals with one particular physical domain. The coupling server, running on a third processing unit, takes care of the exchange of data between the solvers. Since Flecs has to deal with separate solver processes, that have been started independently, it is not possible to use `MPI-1` for exchanging data between those processes. For that reason, Flecs is based on `MPI-2` [3].

The remainder of the paper is structured as follows. In Sect. 2, we describe the importance of coupling two solvers properly to solve interdisciplinary problems in an efficient and accurate way. Sect. 3 gives an overview of the design of Flecs. Through the help of a test problem, Sect. 4 illustrates how Flecs can be used. And finally, Sect. 5 contains some conclusions and future plans.

2 Coupling Methods

Interdisciplinary problems can be solved in two ways. In the first way, the so-called *monolithic* approach, a new dedicated solver is developed that solves the whole system at once. Major advantage is that the solver can be optimized for the specific problem. Development of such a complex, entirely new solver, will take an enormous effort, while there already exists many highly efficient and accurate solvers for the separate domains. The other way, called the *partitioned* approach, is to reuse monodisciplinary solvers, that have been developed and tuned for tens of years. In that case, each physical system is solved individually and interaction effects are treated as external conditions. A disadvantage of this approach is that the coupling algorithm is not as straightforward as it looks. Without much care the accuracy of the coupled problem easily reduces to first-order in time, irrespective of the order of the separate solvers.

Flecs provides an efficient coupling interface for partitioned computation of multidisciplinary problems. The design of Flecs allows all kinds of data transfer algorithms to couple different domains in space and time. Numerical acceleration techniques, like multigrid, can be incorporated, too.

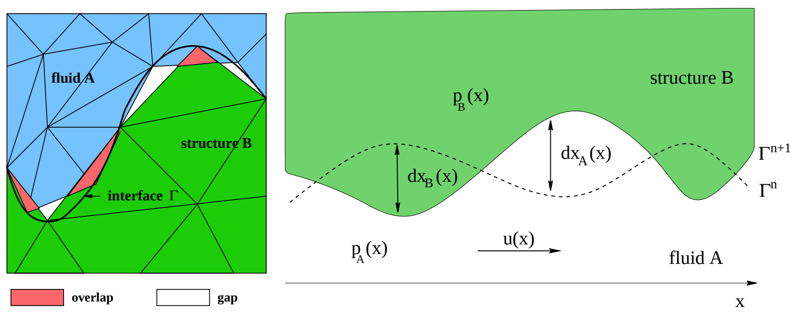


Fig. 1. (a) Non-matching grids in 2D and (b) Configuration of the quasi-1D test problem

2.1 Non-matching Grids

If different grid generators are used for both domains, the mesh interface may not only be non-conforming (nodes at the interfaces do not match, different discretization and/or interpolation order on both discrete interfaces), but also non-matching (cf. Fig. 1(a)) in the way that there are gaps and/or overlaps between the meshes. We remark, that generating matching grids is most of the time not desirable, because, in general, the simulation on one physical domain requires a much finer grid than the simulation on the other one. In the remainder of this paper, we consider FSI rather than some arbitrary physical domain interaction.

De Boer et al. [1] gives a detailed study of data transfer methods, and FSI simulations are performed on non-matching meshes. As coupling method in this paper, we take the radial basis function method (RBF) with large compact support. This method, favored by [1], because of its accuracy and efficiency, does not need orthogonal projection or search algorithms. The coupling between fluid and structure equations arises from the dynamic and kinematic boundary conditions (BC) at the fluid-structure interface. The BC for the displacement on the continuous fluid-structure interface Γ, given by $d\mathbf{x}_A(x) = d\mathbf{x}_B(x)$, where $d\mathbf{x}$ denotes the displacement on either the flow (A) or structure (B) side of the interface. The displacements of the flow points have to be predicted once the displacements of the structure points have been determined. The discrete version of the BC can be formulated as

$$\mathrm{d}\mathbf{x}_A = \mathbf{H}_{AB}\mathrm{d}\mathbf{x}_B,$$

where $\mathbf{H}_{AB} \in \mathbb{R}^{N_A \times N_B}$ is the transformation matrix prescribed by the RBF method. The numbers N_A, the number of flow, and N_B, the number of structure points on the fluid-structure interface, are usually very small compared to the total number of structure and flow points. Analogously, the discrete version of the BC for the pressure forces leads to

$$\mathbf{p}_B = \mathbf{H}_{BA}\mathbf{p}_A,$$

where the transformation matrix $\mathbf{H}_{BA}$ is of size $N_B \times N_A$. The computation of the matrices $\mathbf{H}_{AB}$ and $\mathbf{H}_{BA}$ involves the inversion of a small matrix. The matrices $\mathbf{H_{AB}}$ and $\mathbf{H_{BA}}$ depend on the coordinates of the interface points. If the positions of the interface points have been moved, these matrices are recalculated.

2.2 Coupling Algorithm

We consider, as an example of use, a quasi 1-D channel with a flexible curved wall as shown in Fig. 1(b). The main velocity, u, of the compressible flow is in the x-direction and the structure is modeled as a membrane. The diameter of the tube may vary due to a pressure difference between the pressure in the flow and in the wall. For more details on this test problem, we refer to [1].

The simulation of the compressible flow and the membrane is solved effectively by coupling two solvers. The solvers exchange data to take into account the effects on the other domain. Starting at time t^n, each solver computes the solution at time $t^n + \Delta t^n$ on its own particular domain. In general, Δt^n will be determined by the flow solver. The following steps are carried out to obtain the solution at t^{n+1} from the solution at t^n :

step 1. compute transformation matrices $\mathbf{H}^n_{AB}$ and $\mathbf{H}^n_{BA}$
step 2. obtain the pressure on the structure interface points $\mathbf{p}^n_{B,\gamma} = \mathbf{H}^n_{BA}\mathbf{p}^n_{A,\gamma}$
step 3. calculate the displacements of the structure $\mathrm{d}\mathbf{x}^{n+1}_B$ from the structure equations using the old value of the pressure $\mathbf{p}^n_B$
step 4. use the coupling method to compute the displacements of the tube wall $\mathrm{d}\mathbf{x}^{n+1}_{A,\gamma} = \mathbf{H}^n_{AB}\mathrm{d}\mathbf{x}^{n+1}_{B,\gamma}$
step 5. calculate the new pressure $\mathbf{p}^{n+1}_{A,\gamma}$ from the fluid equations with the new displacements of the tube $\mathrm{d}\mathbf{x}^{n+1}_A$.

The subscript γ denotes that the operations are only performed on data at the discrete interface points. The steps to gather and scatter the data on the interface points have been omitted. The computation of $\mathbf{H}^n_{AB}$ and $\mathbf{H}^n_{BA}$ requires the coordinates on the same time t^n. In Sect. 4, we will return to this example of use.

3 Design Overview

FLECS is decomposed into a *client library* that is to be called from the solver programs, and a *coupling server*, in short *server*, that coordinates the execution of the solvers, takes care of the coupling of the domains and handles the transfer of data between the solvers. Both the client library and the server have been implemented in `C`, so that it is relatively simple to use FLECS in solver programs that have been written in different programming languages like `C++` and `Fortran 90`. In the simplest case the server comprises a single process (as in Fig. 2) that executes the transfer algorithm sequentially.

To limit the complexity of the server, it can only couple two solvers at a time. However, one can couple a solver to two other solver processes by starting a second server.

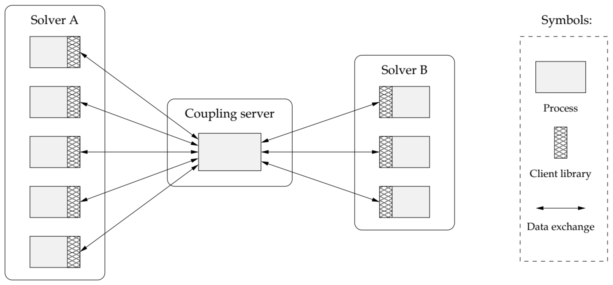


Fig. 2. Schematic representation of FLECS

3.1 The Client Library and Its Usage

The client library provides subroutines for establishing a connection with the server; for describing the geometry of the coupling interface; for describing the data that are to be transferred to and from the server; for sending data to the server; and for receiving data back from the server. If a solver program comprises multiple parallel processes, then each process will contain its own copy of the client library (see Fig. 2), and will establish a separate connection with the server.

Each solver program can be started independently, using its standard start-up procedure; there is no need to change the structure of the solver program. In fact, one only needs to extend the solver program with a small number of subroutine calls to the client library. For an overview of the functions exported by the FLECS client library, see Table 1.

To start a coupled simulation, both solvers set up a connection to the server by calling the client function `FLECS_Connect`. Then the solvers inform the server about the data to be transferred between them. Therefore, one or more *point sets* and *data sets* are created at each side of the interface, and, transferred to the server by calling the functions `FLECS_NewPointSet` and `FLECS_New- DataSet`. The solvers exchange data via the server by calling the functions `FLECS_SendDataSet`, `FLECS_RecvDataSet`, `FLECS_Send` and `FLECS_Recv`. The first pair transfers a data set from one solver to the other, and typically invokes a transformation algorithm on the server. The second pair transfers an arbitrary data array between the solvers, and is particularly used to communicate convergence and time stepping information between both solvers. Both pairs of functions require that a send operation *matches* a corresponding receive operation. The description of a single iteration step of the test problem of Sect. 2.2 can be found in Sect. 4.

3.2 The Coupling Server

The server consists of two parts, as shown in Fig. 3: a *communication and coordination layer*, and a *transfer algorithm*. The communication and coordination layer handles the initialization and finalization of the server; exchanges data

Table 1. An overview of the functions exported by the FLECS client library

Initialization functions	
FLECS_Init	Initializes `MPI`, parses the program names, opens the `MPI` port, publish the name of the server, save the server address
FLECS_Connect	Connects solver and coupling server
FLECS_NewPointSet	Registers point set on coupling interface with coupling server
FLECS_NewCoupling	Defines coupling between point set on this solver to point set on another solver
FLECS_NewElemSet	Defines element set on coupling interface with coupling server
FLECS_NewDataSet	Defines data set associated with point set
Finalization functions	
FLECS_Disconnect	Disconnects solver from coupling server
FLECS_Shutdown	Calls `MPI_finalize` and cleans up the allocated memory
FLECS_DelPointSet	Deletes registered point set
FLECS_DelCoupling	Deletes created coupling
FLECS_DelElemSet	Deletes created element set
FLECS_DelDataSet	Deletes defined data set
Data exchange functions	
FLECS_SendDataSet	Sends data set to other solver via coupling server
FLECS_RecvDataSet	Receives *transformed* data set from other solver
FLECS_Send	Sends *arbitrary* data array to other solver
FLECS_Recv	Receives *arbitrary* data array from other solver
Miscellaneous functions	
FLECS_SetCoords	Updates coordinates of registered point set
FLECS_ErrorString	Converts error code to human-readable error message
FLECS_UseElemSet	Use element set

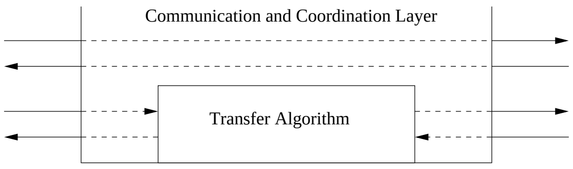


Fig. 3. The coupling server, consisting of a communication and coordination layer and a transfer algorithm. The arrows indicate the flow of data between the coupling server and two solver programs.

between the server and the two solver programs; manages the data structure – including point sets, couplings, and data sets – that have been created by the client library on behalf of the solver programs; and manages the coupling-specific data structures that have been created by a transfer algorithm. Obviously, more than one data set can be associated with a point set. To associate a data set to a particular point set, an integer value `pset`, which uniquely identifies a point set, must be involved in the data set message.

The transfer algorithm handles the conversion of a data set from one point set to another point set. This part of the server is based on a plug-in architecture, that makes it easy to implement new transfer algorithms, see also Sect. 4.1. The transfer algorithm itself can be implemented in any programming language. Since the transfer algorithm is a self-contained module of the coupling server, one can experiment with different types of transfer algorithms without having to worry about non-essential details such as communication between the server and the solver programs.

4 Execution of Test Problem Using Flecs Routines

Again, we consider the quasi-1D test problem described in Sect. 2.2. For simplicity, we assume that flow solver **A** and structure solver **B** have not been parallelized and that each solver process runs on a single processing unit. The server $\mathbf{\Gamma}$ becomes a third process which takes care of the communication and the interpolation of the meshes.

Fig. 4 represents a single FSI iteration outlined in nine (parallel) *stages*. The various steps of the coupling algorithm, as listed in Sect. 2.2, are shown, and, on the solvers **A** and **B** the calls to the client library of FLECS are inserted. We choose, that the flow solver determines the progress of the integration process, i.e., the time step is calculated by the flow solver. In addition, the flow residual $\mathbf{R}_{\mathbf{A}}^{n+1}$ controls the complete system. When the solver **A** has computed the next time step Δt^n, (*stage i*) its value must be sent via the server $\mathbf{\Gamma}$ to the solver **B** using the pair (`FLECS_Send`, `FLECS_Recv`, *stage a*).

Figure 4 illustrates that by breaking the computation of the transformation matrices $\mathbf{H}_{AB}^n$ and $\mathbf{H}_{BA}^n$ (**step** 1) into two separate parts, the server can compute these matrices simultaneously with other operations performed by the solvers (*stages c* and *g*). More precisely, no extra wall clock time is needed to compute these matrices. Let us assume that $\mathbf{H}_{BA}^n$ has been calculated in a previous integration step (*stage g*), then the force on the structure can be updated (**step** 2, *stage b*). As a result, we obtain the vector $\mathbf{p}_{B,\gamma}^n$ asked by solver **B**, by means of a call of `FLECS_RecvDataSet`. Next, solver **B** computes the solution at the structure domain (**step** 3, *stage b*) at time $t^n + \Delta t^n$. We observe that the computation of $\mathbf{H}_{AB}^n$ can be postponed, allowing the solver **B** to start the computation of **step** 3 earlier. As a consequence, the computation of $\mathbf{H}_{AB}^n$ can be carried out simultaneously with **step** 3 (*stage c*). The vector $\mathrm{d}\mathbf{x}_{B,\gamma}^{n+1}$ is needed to carry out **step** 4, and corresponds to calls of `FLECS_SendDataSet` and `FLECS_RecvDataSet` on the solvers **B** and **A**, respectively (*stage d*).

Again, by transferring $\mathrm{d}\mathbf{x}_{B,\gamma}^{n+1}$ (calling `FLECS_SendDataSet`) first and then the new point set $\mathbf{x}_{B,\gamma}^{n+1}$ (calling `FLECS_SetCoords`, *stage f*) more parallelism can be obtained. However, as stated above, the transformation must be applied on updated values of $\mathrm{d}\mathbf{x}_{B,\gamma}^{n+1}$. The transformation operation delivers new values $\mathrm{d}\mathbf{x}_{A,\gamma}^{n+1}$ to be transferred to solver **A** (calling `FLECS_RecvDataSet`, *stage d*). Next, the

Flow Solver **A**	Coupling Server $\mathbf{\Gamma}$	Structure Solver **B**	*Stage*
FLECS_Send(Δt^n)	Transfer(Δt^n)	FLECS_Recv(Δt^n)	*a*

FLECS_SendDataSet ($\mathbf{p}^n_{A,\gamma}$)	Receive($\mathbf{p}^n_{A,\gamma}$)
	$\mathbf{p}^n_{B,\gamma}$ =
	`Update`($\mathbf{H}^n_{AB}, \mathbf{p}^n_{A,\gamma}$)
	Send($\mathbf{p}^n_{B,\gamma}$)
		FLECS_RecvDataSet ($\mathbf{p}^n_{B,\gamma}$)	*b*

	$\mathbf{H}^n_{AB}$ = `Coupling`($\mathbf{x}^n_{A,\gamma}, \mathbf{x}^n_{B,\gamma}$)	`SolveStructure` $\Rightarrow \mathrm{d}\mathbf{x}^{n+1}_B$	*c*

FLECS_RecvDataSet ($\mathrm{d}\mathbf{x}^{n+1}_{A,\gamma}$)	Receive($\mathrm{d}\mathbf{x}^{n+1}_{B,\gamma}$)
	$\mathrm{d}\mathbf{x}^{n+1}_{A,\gamma}$ =
	`Update`($\mathbf{H}^n_{AB}, \mathrm{d}\mathbf{x}^{n+1}_{B,\gamma}$)
	Send($\mathrm{d}\mathbf{x}^{n+1}_{A,\gamma}$)
		FLECS_SendDataSet ($\mathrm{d}\mathbf{x}^{n+1}_{B,\gamma}$)	*d*

$\mathbf{x}^{n+1}_{A,\gamma} = \mathbf{x}^n_{A,\gamma} + \mathrm{d}\mathbf{x}^{n+1}_{A,\gamma}$			*e*

FLECS_SetCoords ($\mathbf{x}^{n+1}_{A,\gamma}$)	Receive($\mathbf{x}^{n+1}_{B,\gamma}$)
	Receive($\mathbf{x}^{n+1}_{A,\gamma}$)
		FLECS_SetCoords ($\mathbf{x}^{n+1}_{B,\gamma}$)	*f*

`SolveFlow` $\Rightarrow \mathbf{p}^{n+1}_A$ Compute $\mathbb{R}^{n+1}_A$ **finish** = $\mathbf{R}^{n+1}_A < \varepsilon$ `.or.` $t^{n+1} > t_e$	$\mathbf{H}^{n+1}_{AB}$ = `Coupling`($\mathbf{x}^{n+1}_{A,\gamma}, \mathbf{x}^{n+1}_{B,\gamma}$)		*g*
FLECS_Send(**finish**)	Transfer(**finish**)	FLECS_Recv(**finish**)	*h*

if **finish**
FLECS_Disconnect
FLECS_ShutDown
else
Compute Δt^{n+1}
end
	if **finish**
	Break both connections
	end
		if **finish**
		FLECS_Disconnect
		FLECS_ShutDown
		end
			i

Fig. 4. The $n+1$-th iteration step of the FSI process expressed in FLECS routines (see Table 1). The superscript n stands for the time step, whereas the subscript A or B indicate that the values belong to the domain of the flow solver **A** or the structure solver **B**. A vector refers to interface values in case the underscore γ is present. Here, t^n and Δt^n denote the current time and time step, $\mathbf{x}$ and $\mathrm{d}\mathbf{x}$ the position and displacement of the coordinates, $\mathbb{R}^n_A$ the residual, ε indicates the required accuracy of $\mathbf{x}$ for FSI iteration. `SolveFlow` performs a single iteration with the flow solver, resulting in, among others, the updated value $\mathbf{p}^{n+1}_A$, whereas `SolveStructure` carries out a single iteration with the structure solver, producing, among others, the updated value $\mathrm{d}\mathbf{x}^{n+1}_B$. The interface points of $\mathbf{p}^{n+1}_A$ and $\mathrm{d}\mathbf{x}^{n+1}_B$ are input for the transformation matrices. Finally, the `boolean` **finish** determines whether the program terminates.

flow solver carries out an integration step from t^n till t^{n+1} (**step** 5). Simultaneously, the matrix $\mathbf{H}_{AB}^{n+1}$ for the flow-structure interface can be calculated using the new point sets $\mathbf{x}_{A,\gamma}^{n+1}$ and $\mathbf{x}_{B,\gamma}^{n+1}$ (**step** 1, *stage g*).

Let ε be some given tolerance, and, let t_e be the end time, then if

$$\mathbf{finish} = R_A^{n+1} < \varepsilon \mid t_n + \Delta t^n \geq t_e$$

is not true, a new iteration step $n + 1$ can be started after calculating the new time step Δt^{n+1}. In case of convergence, the process terminates, and solver **A** sends messages to server $\mathbf{\Gamma}$ and solver **B**. This can be carried out by calls to the routines FLECS_Send and FLECS_Recv (*stage h*) to notify solver **B** to terminate the calculation, followed by calls to FLECS_Disconnect and FLECS_Shutdown to disconnect the connections and to clean up all MPI states.

4.1 The Server Program

The main loop of the server program can look like

```
int main ( int argc, char** argv )
{ flecs_transfer_t  transf;
  int               result;

  FLECS_InitTransfer ( &transf );
  transf.NewCoupling  = NewCoupling;
  transf.DelCoupling  = DelCoupling;
  transf.InitPoints   = InitPoints;
  transf.SetCoords    = SetCoords;
  transf.TransferData = TransferData;

  FLECS_SetTransfer  ( &transf );
  FLECS_SetErrorMode ( FLECS_ERRORS_ABORT );
  FLECS_Init         ( &argc, &argv );
  FLECS_Connect      ( );

  do
  { result = FLECS_MainLoop ( ); }
  while ( ! result );

  FLECS_Shutdown (); return 0;
}
```

The do while loop is executed repeatedly until result becomes FALSE. The subroutine FLECS_MainLoop '*listens*' whether there is a message to be received, where after the server copies the data out of the send buffer and will act accordingly. A *message* can be one of the functions listed in Table 1. Assume that the application demands a multigrid approach, and, assume that FLECS_MainLoop receives a message from solver **A** for a new coupling (i.e., solver **A** has called FLECS_NewCoupling). Then the server $\mathbf{\Gamma}$ may expect a similar request by solver **B**, followed by new point sets and associated data sets of both solvers. We remark that as a consequence of this simple approach, illustrated by the above program listing, the FLECS' user does not have to implement the server program, but only adds some simple programs `NewCoupling`, `DelCoupling`, `InitPoints`, `SetCoords` and `TransferData`. The latter must include the coupling algorithm. Moreover, the user has to include in his/her solver programs some calls to FLECS routines. Such calls must *correspond,* e.g., `FLECS_ Send` and `FLECS_Recv` are appearing in pairs, otherwise an error will be generated.

5 Conclusions and Future Plans

In this paper, we have introduced FLECS, a coupling shell, which can be used as an interface for multidisciplinary simulations, e.g., for fluid-structure interaction computations. A very simple quasi one-dimensional test problem is used to show the usage of an preliminary implementation of FLECS, and an overview of the available routines is given. More investigations are needed to prove its functionality, to experiment with different kind of coupling methods, such as nearest neighbor or Gauss interpolation. We would like to extend the parallel capabilities of FLECS to be able to simulate realistic cases on parallel computer platforms.

References

1. de Boer, A., van Zuijlen, A.H., Bijl, H.: Review of coupling methods for non-matching meshes. Comput. Methods in Appl. Mech. and Eng. 196(8), 1515–1525 (2007)
2. Lingen, E.J., Nool, M., de Boer, A., Bijl, H.: Design of Flecs, a flexible coupling shell (2006), see `http://www.aero.lr.tudelft.nl/FLECS`
3. Gropp, W., Lusk, E., Thurs, R.: Using MPI-2. In: Advanced Features of the Message Passing Interface, The MIT Press, Cambridge (1999)
4. MpCCI, The Fraunhofer-Institute for Algorithms and Scientific Computing (SCAI). Multidisciplinary Simulation through Code Coupling, see `http://www.scai.fraunhofer.de/mpcci.html`
5. Bosscher, S., van Zuijlen, A.H., Bijl, H.: Two level algorithms for partitioned fluid-structure interaction computations. Comput. Methods Appl. Mech. Eng. 196(8), 1458–1470 (2007)
6. van Zuijlen, A.H., Bijl, H.: Implicit and explicit higher order time integration schemes for structural dynamics and fluid-structure interaction computations. Comput. Struct. 83, 93–105 (2005)

New Scalability Frontiers in *Ab Initio* Electronic Structure Calculations Using the BG/L Supercomputer

Constantine Bekas, Alessandro Curioni, and Wanda Andreoni

IBM Research, Zurich Research Laboratory,
Säumerstrasse 4, Rüschlikon, CH-8803, Switzerland
{bek, cur, and}@zurich.ibm.com

Abstract. In recent years the BlueGene/L supercomputer (BG/L) has significantly extended the computational frontiers in several important problems of science and engineering. In this paper we focus on *ab initio* electronic structure calculations, which have enabled accurate modeling and prediction of the properties of materials by using first principle quantum-mechanical calculations. We report our progress in scaling the `CPV` code on EPFL's recently installed BG/L system using a Task Group (TG) hierarchical parallelization strategy for 3D `FFTs`. We illustrate that TG permits scaling of the 3D `FFTs` to the extent that the dense linear algebra kernels dominate in the overall cost.

1 Introduction

Ab initio electronic structure calculations, in the framework of Density Functional Theory (DFT) [5,7], have proven remarkably accurate in providing a wealth of information concerning several important physical properties of complex materials. However, DFT calculations are extremely demanding and have stretched our computational capabilities to their very limits. Therefore, advances in better simulation techniques and algorithms as well as advanced supercomputer architectures receive much of attention in this very active field of research.

The advent of the BG/L supercomputer has triggered a lot of excitement in the electronic structure calculation community, mainly caused by the prospect of highly increased scalability (see for example [4] where excellent scalability results are presented for up to 32000 cpus). In this paper, we report our progress in scaling the `CPV` code [2,8], which is part of the `quantum Espresso` package[1], on the recently installed BG/L supercomputer at EPFL, Lausanne, Switzerland[2].

The core problem in DFT calculations is the solution of the Kohn-Sham equations

$$H_\rho \Psi_\rho = E\Psi_\rho, \tag{1}$$

[1] `http://www.pwscf.org`

[2] `http://bgl.epfl.ch`

where ρ is the charge density of the electrons distribution, H_ρ is the Kohn-Sham Hamiltonian operator, Ψ_ρ are the wavefunctions and E is the energy of the system. Observe that this is a nonlinear eigenvalue problem, since the Hamiltonian and the wavefunctions depend upon each other through the charge density ρ. The last decades have seen many methods that attempt to efficiently solve equation (1). All of them utilize some short of iteration which aims at improving some initially selected wavefunctions so that at the end of the iteration the approximate energy E is as small as possible, or in other words the solution of equation (1) is self-consistent. The computational complexity of practical algorithms for this problem stems mainly from two factors:

Discretization. The Hamiltonian operator and thus the wavefunctions have to be discretized in some suitably selected basis. This typically results in a very large discretized Hamiltonian (in the order of millions of degrees of freedom). Consequently, the eigensolvers for dense matrices, such as the ones in LAPACK[3], cannot be applied in this case. Thus, one has to utilize instead iterative techniques that only use the application of the discretized Hamiltonian operator on a likewise discretized function (i.e. vector). In plane wave implementations, such as in the present case, this results in a sequence of inverse and forward 3D FFT transforms, the efficient implementation of which is crucial for overall performance.

Orthogonality. The wavefunctions Ψ_ρ are orthogonal among each other. Thus, the approximate discretized wavefunctions must likewise form an orthogonal basis. This has the consequence that all iterative methods that target to solve the nonlinear eigenproblem (1) have to maintain a set of mutually orthogonal approximate wavefunctions at each iteration. Maintaining orthogonality will of course come at a cost, which scales as the cube of the total number of valence electrons in the system.

2 Task Groups Strategy for 3D Parallel FFTs

The CPV code uses a plane wave basis to discretize the Schrödinger equation and thus relies on heavy use of 3D Fast Fourier transforms. We have investigated the efficient parallelization of 3D FFTs. In particular, we have adopted a scheme based on a Task Groups parallelization strategy that concurrently performs several parallel 3D FFTs, one per each group of processors. Similar to the approach in [6] for the CPMD code[4], we exploit opportunities for hierarchical parallelism in CPV.

In plane wave codes, the wavefunctions $\Psi_\rho = [\psi_1, \ldots, \psi_{occ}]$ are expanded in Fourier space, where *occ* is the total number of valence electrons, which in turn depends on the type and number of the involved atoms. In medium size simulations *occ* is in the order of several hundreds, while large simulations will

[3] http://www.netlib.org/lapack

[4] http://www.cpmd.org

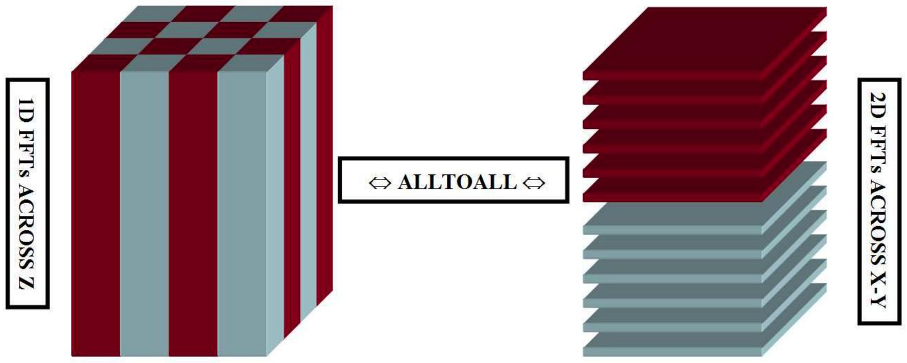


Fig. 1. Structure of the standard parallel 3D FFT

push *occ* to thousands or even tens of thousands. The charge density $\rho(r)$ at position r in real space is given as

$$\rho(r) = \sum_{i=1}^{occ} |\psi_i(r)|^2. \tag{2}$$

Observe that since the wavefunctions are expanded in Fourier space, computation of charge density in Fourier space would entail doubly nested summations. Instead, it is performed in real space. To this end, the wavefunctions are transformed back to real space by means of inverse 3D `FFTs`. In the case of P available processors, all of them can be devoted to a single parallel 3D `FFT`. On the other hand, we can do G, $(G < occ)$ parallel 3D `FFTs` concurrently. We define G groups of processors, each of which works on a single parallel 3D `FFT`. Thus, the number of loops in computing the charge density is $\lceil occ/G \rceil$ (special handling of the last loop takes care of the case that occ is not divided exactly by G).

Performing one parallel 3D `FFT` at a time, as is the common practise in electronic structure calculations, limits scalability. Here is why: The Fourier coefficients of the wavefunctions are organized in a $x-y-z$ 3D mesh in Fourier space. For all wavefunctions, each processor is assigned a number of pencils across the z (vertical) direction. Figure 1 (left cube) illustrates the case for two processors and one wavefunction. The 3D inverse `FFT` is performed as follows:

1. 1D inverse `FFTs` across the z (vertical) direction are computed independently.
2. An all-to-all global communication distributes the results to all processors, so that each processor ends up with a number of complete $x-y$ planes (see right cube of Fig 1).
3. Then, 2D inverse `FFTs` are performed independently by each processor without the need for further communication.

It is clear that if the number P of available processors is larger than the number of $x-y$ planes, which is the mesh dimension across the z direction, some processors will get no planes at all. In general, the scalability of this scheme is limited by the largest dimension of the FFT mesh. For parallel architectures with a moderate number of available processors this limitation is not severe as practical runs of *ab initio* codes use hundreds of $x - y$ planes. However, on the BG/L we need to utilize thousands of processors and thus we need a different parallel 3D FFT scheme. Our solution is to exploit opportunities for hierarchical parallelism.

Observe that in order to calculate the charge density $\rho(r)$ by means of (2) we need to iterate through a loop of 3D FFTs, equal to the number *occ* of valence electrons. The Task Groups (TG) strategy will assign different groups of processors to different wavefunctions. Suppose that a processor p_e is empty, in the sense that no $x - y$ planes would be assigned to it if all P processors were to participate in an inverse 3D FFT. Then, since the number of its peers in the group will be P/G we can choose the number of groups G so that a processor will be never empty. We outline the TG scheme in Table 1.

The concurrent implementation of G 3D FFTs is organized on a 2D mesh of processors. Each processor belongs to its row group as well as to its column group. Global communications are restrained within these groups. Iteration `k` performs the 3D FFTs needed for wavefunctions `(k-1)*G+i, i=1,...,G`. Remember that each processor holds only part of the Fourier coefficients for each wavefunction. Thus, the `all-to-all` within the row group (line 2) brings to each column group all the Fourier coefficients for the wavefunction assigned to it. For example, at iteration `k` the $j-th$ processor of the first column group will send its parts of the `(k-1)*G+i, i=2,...G` wavefunctions, to its row group peers $\mathtt{i} = 2, \ldots, \mathtt{G}$, respectively, while it will receive from them all needed parts for the $(\mathtt{k}-1) * \mathtt{G} + 1$ wavefunction. Then, all processors in each column group can perform a parallel 3D FFT (line 3). Finally, the charge density ρ can be accumulated by means of a global reduction across processors in each row group (line 4).

The Task Groups scheme requires additional memory. Remember that each processor holds a part of the wavefunctions coefficients for all eigenvalues. Thus, in order for a column group to work exclusively on a single eigenvalue each processor needs to receive additional wavefunction coefficients from its row group peers. The amount of the extra memory depends upon the number G of Task Groups. There is a tradeoff between the number of available processors P and the number of Task Groups. In order to exploit a large number of available processors we need many Task Groups. On the other hand, this will increase the amount of additional local memory as well as the traffic on the interconnect for the initial `all-to-all`. However, the 3D FFTs within each column group will also require less communication, since only P/G processors are involved in each column group.

Similar to the calculation of charge density, forces contribution to the orthogonality constraints for the wavefunctions requires a loop of forward 3D FFTs across the occupied states. A parallel 3D FFT is implemented following the same steps as in the inverse transformation in exactly the opposite order: i) Each processor holds a number of complete $x - y$ planes on which it performs 2D FFTs,

Table 1. The TG parallel 3D `FFT` scheme for the calculation of charge density $\rho(r)$

```
Define a 2D processor array
  The number of columns is equal to the number $G$ of Task Groups
  The number of rows is equal to the number of processors in each Task Group

1. DO $k = 1,\ occ/G$
    2. all-to-all communication in row group: brings all needed Fourier
       coefficients for 3D FFT
    3. parallel 3D FFT within column group
    4. allreduce to accumulate charge density within row group
5. ENDDO
```

ii) a global `all-to-all` assigns to each processor a number of z sticks on which independent 1D `FFTs` are performed. The Task Groups strategy is analogously adopted, so that each column group works on different wavefunctions.

We note that very good scaling for parallel 3D `FFTs` has been achieved by means of the Volumetric `FFT` algorithm [3], which employs distribution of the `FFT` mesh across all three $x-y-z$ directions. However, employing this scheme in `CPV` would require a major redesigning of the data organization and the corresponding data structures. The Task Groups scheme allows for very good scalability while requiring only minimal changes to the underlying electronic structures code.

Customizing for the BG/L. The decisive parameters in order to select the optimal G involve i) the amount of memory available to each processor core, 512 MB in coprocessor mode and 256 MB in virtual mode, and ii) the unsurpassed latency and bandwidth of the dedicated collective communication tree interconnect: latency of tree traversal 2.5 μs, 2.5 GB/s bandwidth per link, 23TB/s total binary tree bandwidth (64k machine). It is typical in our practical applications to use 8-32 Task Groups.

3 Orthogonalization

`CPV` implements the Car-Parrinello method that casts the nonlinear eigenproblem (1) in a constrained optimization framework using a suitable Lagrangian. The constraints ensure the orthogonality of the approximate wavefunctions. In particular, the Lagrange multipliers involved are given as solutions of a non-linear equation [8]:

$$A + \Lambda B + B^\top \Lambda + \Lambda C \Lambda^\top = I, \tag{3}$$

where, A, B and C are given $occ \times occ$ matrices and Λ is the matrix of Lagrange multipliers. All of these matrices change at each Molecular Dynamics step. This equation is typically solved iteratively where the first iterate $\Lambda^{(0)}$ is determined by solving the

$$\Lambda^{(0)} B_h + B_h \Lambda^{(0)} = I - A, \tag{4}$$

where $B_h = \frac{B+B^\top}{2}$ is the Hermitian part of matrix B. This equation is solved exactly if we consider the unitary matrix U that diagonalizes matrix B_h. Then, the next iterate $\Lambda^{(k+1)}$ is calculated by solving the following equation

$$\Lambda^{(k+1)}B_h + B_h\Lambda^{(k+1)} = I - A - \Lambda^{(k)}B_a + B_a\Lambda^{(k)} - \Lambda^{(k)}C\Lambda^{(k)}, \qquad (5)$$

where $B_a = \frac{B-B^\top}{2}$ is the skew-symmetric part of matrix B, which is also solved similarly to equation (4). Observe that we need to diagonalize matrix B only once, in order to calculate the first iterate $\Lambda^{(0)}$. However, this matrix changes at each MD step. This incurs a cubic cost $O(occ^3)$ in terms of the valence electrons, which is the well known cubic cost of DFT electronic structure calculations.

Observe that the solution of equation (5), in particular its right hand side, requires matrix-matrix multiplications. These operations are easy to distribute across the available processors. Furthermore, an extremely efficient `DGEMM` library, which utilizes both floating point units (FPUs) of both cores on each computing node, is available for the BG/L nodes and has been used to achieve unprecedented levels of performance for *ab initio* simulations on the BG/L [4] (see also `www.cpmd.org`).

Traditionally, the major cost of plane wave codes has been the Fourier transforms. However, as it will be illustrated in the experiments section, the Task Groups strategy allows for excellent scaling of the `FFTs` on massively parallel architectures. Thus, efficiently scaling dense eigensolvers become crucial in order to exploit the full potential of the BG/L. Currently, we are not aware of dense eigensolvers able to adequately scale to thousands or even hundreds of processors. Although very good existing such software, i.e. SCALAPACK[5], PLAPACK[6] can help, our goal is to prepare for large scale simulations that inherently will require thousands of processors.

3.1 Improved Orthogonalization Using Parallel Jacobi

We now briefly sketch an alternative approach, which is the subject of a forthcoming report [1]. It is well known that the Jacobi method for eigenvalues is in general much slower than the standard approach for computing eigenvalues and eigenvectors of dense matrices (see LAPACK), which utilizes reduction to Hessenberg form (or tridiagonal for symmetric matrices). However, parallel Jacobi schemes, such as the BFG library[7] have been shown to exhibit much better scaling than the standard approach. What is particular to the current application is that the iterative orthogonalization scheme is repeated at each step of the molecular dynamics simulation. That is, matrix $B_h^{(i)}$, which we need to diagonalize at the $i-th$ molecular dynamics step, is usually a small perturbation of matrix $B_h^{(i-1)}$ that was diagonalized at the previous step. The strategy is now straightforward to describe:

[5] http://www.netlib.org/scalapack/
[6] http://www.cs.utexas.edu/∼plapack/
[7] http://www.cse.clrc.ac.uk/arc/bfg.shtml

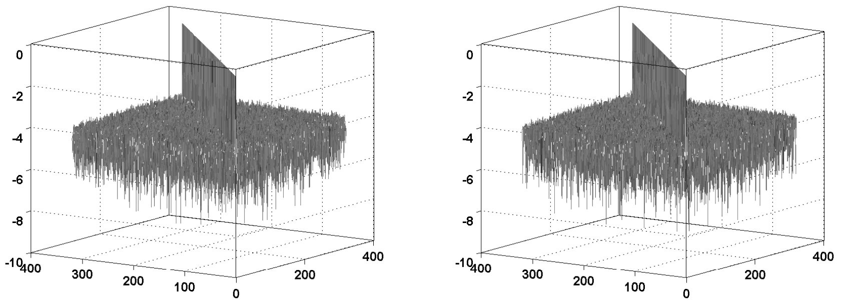


Fig. 2. Approximately diagonal matrices $\hat{D}_2$ and $\hat{D}_6$ for steps 2 and 6 (left and right respectively) of wavefunction optimization for a 32 water molecule simulation using `CPV`. The size of the matrices is 320×320, and we plot the absolute value of matrix elements in logarithmic scale.

1. Diagonalize matrix $B_h^{(1)} = U_0 D_0 U_0^\top$ at the first MD step using the fastest available parallel dense eigensolver (i.e. SCALAPACK).
2. At each step: approximately diagonalize matrix $B_h^{(i)}$ as $\hat{D}_i = U_{i-1}^\top B_h^{(i)} U_{i-1}$.
3. At that step: Use very few passes of the parallel Jacobi iteration to diagonalize matrix $\hat{D}_i$ and compute D_i, U_i.

It is expected that matrix $\hat{D}_i$ will be almost diagonal and thus very few passes (perhaps even one) will be needed for the Jacobi iteration to converge, rendering it much faster than the traditional reduce-diagonalize approach (see Figure 2). This, combined with the good scaling of the parallel Jacobi method will yield a highly efficient scheme for the calculation of the orthogonality constraints. In addition, the calculation of $\hat{D}_i$ is easy to parallelize and again uses the excellent double FPU - dual core `DGEMM` library available for the BG/L nodes.

4 Scalability Experiments

We experimented with a molecular system comprised of 80 water molecules, that represents a problem of intermediate size (240 atoms). The size of the FFT mesh used was 128^3. The number of occupied electrons of the system is $occ = 320$, which is the size of the square matrices that need to be diagonalized for the calculation of the orthogonality constraints.

The left plot of Figure 3 illustrates scalability results (total run time) for the calculation of the charge density and the forces contribution to the orthogonality constraints with and without the TG strategy. There are 128 $x-y$ planes across the z direction. Thus, the standard parallel 3D `FFT` implementation scales only up to 128 computing nodes. Note that for the FFT, only a single computing core is used on each node, as the remaining is exclusively devoted to communication. On the other hand, the TG implementation continues to scale, where we have

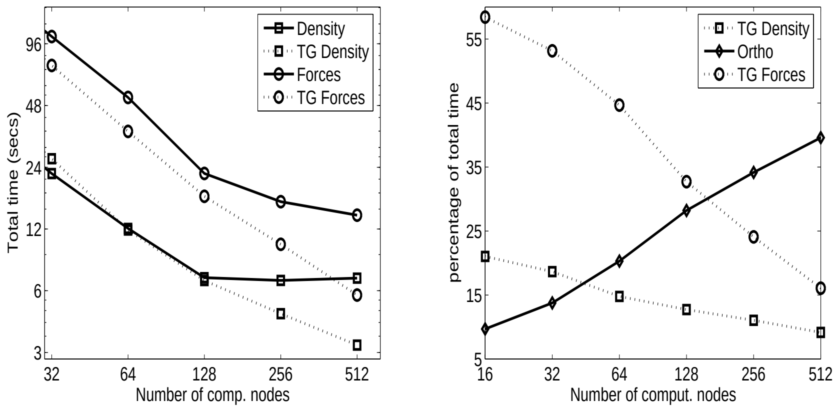


Fig. 3. Left: Scalability experiments for charge density and force contribution to the orthogonality constraints. Right: Comparison of total percentages for charge density and force contribution against orthogonalization. The test case is a system of 80 water molecules.

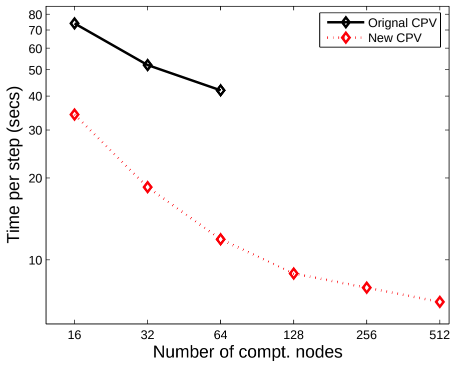


Fig. 4. Comparison of run times per step of old and new `CPV` implementations on the `BG/L`. The test case is a system of 80 water molecules.

used 2 Task Groups in the case of 256 computing nodes and 4 Task Groups in the case of 512 computing nodes.

The right plot of Figure 3 illustrates the percentage, in terms of run time, of the FFT related computation (with TG) compared with the percentage of the orthogonalization related computations. We stress that the latter is dominated by the diagonalization of the dense matrix B_h at each MD step. These constitute the main computational kernels of the application. What remains involves input-output operations and other tasks whose relative load reduces drastically as the size of the simulation increases. It is evident that the improved scaling of the 3D `FFTs` causes the diagonalization to become dominant in terms of cost: 40% for the orthogonalization while 25% for the 3D `FFTs` in the case of 512 computing nodes.

It is important to note that diagonalization is based on a BLAS 3 implementation that uses the dual-core, dual-fpu DGEMM library available for the BG/L nodes.

Figure 4 illustrates a comparison of our current BG/L implementation of CPV with the original porting to the machine. Introducing the TG parallel 3D FFT and using the dual-core, dual-fpu DGEMM library yields a five-fold decrease in time per step for the CPV code on BG/L. It is also very interesting to point out that in the latest version scaling continues up to 512 computing nodes. After that point and on most of the run time per step is spent in diagonalizing dense matrices for the orthogonalization, which as we showed does not scale in the current version of the code.

5 Conclusion

The TG strategy permits the efficient scaling of 3D FFTs. However, in codes such as CPV, that can use a lesser number of plane waves in the expansion of the wanefunctions, dense linear algebra kernels quickly dominate as the number of processors increases and impair scalability. Thus, efficient scaling in such kernels, even for small matrices, are of crucial importance in order to exploit current as well as emerging massively parallel architectures such as the BG/L. We have outlined a strategy that permits scalable diagonalization in the context of orthogonalization of the wave functions. Preliminary results are encouraging and suggest that the proposed approach can effectively push the scalability frontier of such codes. The Task Groups strategy is already implemented in the latest release of the CPV code (available through the quantum Espresso software package), exhibiting very satisfactory performance in real-world simulations.

Acknowledgements. The first author would like to thank Z. Drmač for useful discussions concerning the parallel Jacobi iteration.

References

1. Bekas, C., Curioni, A., Andreoni, W.: Improved orthogonalization in ab initio molecular dynamics simulations using a parallel Jacobi eigensolver. Manuscript in preparation (2006)
2. Car, R., Parrinello, M.: Unified approach for molecular dynamics and density-functional theory. Phys. Rev. Lett. 55(22), 2471–2474 (1985)
3. Eleftheriou, M., Moreira, J.E., Fitch, B.G., Germain, R.S.: A Volumetric FFT for BlueGene/L. In: Pinkston, T.M., Prasanna, V.K. (eds.) HiPC 2003. LNCS (LNAI), vol. 2913, pp. 194–203. Springer, Heidelberg (2003)
4. Gygi, F., Yates, R.K., Lorenz, J., Draeger, E.W., Franchetti, F., Ueberhuber, C.W., Sexton, J.C., de Supinski, B.R., Kral, S., Gunnels, J.A.: Large-scale first-principles molecular dynamics simulations on the bluegene/l platform using the qbox code. In: Proceedings of Supercomputing 05 (2005)
5. Hohenberg, P., Kohn, W.: Inomogeneous electron gas. Phys. Rev. 136(3B), B864–B871 (1965)

6. Hutter, J., Curioni, A.: Car-Parrinello molecular dynamics on massively parallel computers. Chem. Phys. Chem. 6, 1788–1793 (2005)
7. Kohn, W., Sham, L.J.: Self-consistent equations including exchange and correlation effects. Phys. Rev. 140(4A), A1133–A1138 (1965)
8. Laasonen, K., Pasquarello, A., Car, R., Lee, C., Vanderbilt, D.: Car-Parrinello molecular dynamics with Vanderbilt ultrasoft pseudopotentials. Phys. Rev. B 47(16), 10142–10153 (1993)

A PAPI Implementation for BlueGene

Nils Smeds

IBM, Isafjordsg 1, SE-164 92 Stockholm, Sweden
Nils.Smeds@se.ibm.com

Abstract. The IBM BlueGene/L (BG/L) super-computer holds 3 of the top 10 rankings on the 26th TOP500 list of LINPACK performance. The system is novel in its design in many aspects when compared to other more traditional high-performance computing systems.

When developing system libraries as well as when tuning application code for BG/L it is essential to be able to measure the impact of code modifications and algorithmic choices. PAPI is a platform neutral user level library to accomodate programmers' need to access on-chip performance counters. This paper describes the implementation of the low-level kernel interface for hardware performance counter access on BG/L and the accompanying PAPI implementation.

1 Introduction

1.1 Hardware Performance Counters

The central processing unit (CPU) found in many appliances today is a complex system in itself consisting of several cooperating functional units. The design of the modern CPU is the result of an evolutionary process where the experience of previous designs is combined with simulation of possible future designs to decide upon the next version of a particular CPU.

The simulation process is augmented by having probes (hardware performance monitors) available in the CPU that can be used to measure events visible to the CPU. These measurements can be used to validate the accuracy of the simulator and to diagnose performance bottlenecks in the current CPU design to guide future CPU enhancements.

Over the years computer systems vendors have become more open to publicize the implementation of these hardware performance monitors. This has allowed advanced application programmers to measure the behaviour of their codes on the particular CPU which in turn can be used as information for code tuning and guidance to further development of the application to enhance performance on that particular CPU.

1.2 PAPI

Even in the case where public information about the hardware performance monitors on the chip was available from the vendor, their use has been restricted to a select group of highly skilled programming experts. The reason is that the information generally available have most often been sparse or incomplete and the implementation is specific for every CPU and may even vary between CPU versions.

To alleviate the work of adapting an application code that has been modified to take measurements using the probes available on a particular CPU to another CPU, a hardware independent application programming interface (API) has been designed as described by Brown et al. [1]. This interface is known as the hardware performance counter API, PAPI.

PAPI allows access to hardware counters using high-level programming languages such as e.g. C and Fortran. Access to the counters is simplified by providing a set of predefined commonly available counter events, while architecture specific events still can be accessed by means of so called “native” events. The interface provides for simple construction of event “bundles” – event sets – that can be started, stopped and read independently. Counter conflicts, availability of the user selected events and similar hardware dependent aspects is handled by the PAPI library and underlying substrate and reported back to user level by return codes.

1.3 IBM BlueGene/L

The IBM BlueGene/L supercomputer is targeted at creating a highly scalable system capable to be used in the most demanding computational tasks. The design described by Adiga et al. [2] uses a system-on-a-chip (SOC) design to obtain high-reliability, relatively low power consumption and cooling requirements while still maintaining high-bandwidth, low-latency communication and a large capacity in floating point operations. Its design has also introduced new concepts in the field of hardware performance monitoring as the SOC design has enabled a hardware performance counter architecture that is capable of registering events not only in the memory hierarchy, but also in the network controllers.

2 Implementation

2.1 The IBM BlueGene/L Hardware Performance Counters

The IBM BlueGene/L compute node CPU is a number of functional units collected in a single chip. The functional units include 2 PowerPC 440 cores, 2 Gbit ethernet controllers, 6 torus network controllers, 3 tree network controllers, a barrier network controller, a system management JTAG (IEEE 1149.1-1990) network, 2 floating point units and a unified performance counter (UPC). The CPU cores and floating point units run at a clock frequency of 700MHz while the remaining functional units, including the UPC, run at 350MHz.

The UPC contains 48 counter registers, each register being 32 bit wide. The registers can be programmed to register counts from a total of 311 available events on the chip [3]. The events correspond to activities in the different functional units whose events tracing mechanisms are routed to the UPC. The chip also has access to a 64 bit wide clock cycle counter or time-stamp counter (TSC) that counts the number of core CPU cycles that pass.

In addition to the TSC and UPC registers, the floating point units (FPU) each have 2 32 bit wide counter registers capable of registering events in the respective FPU. There are 8 FPU events available. All in all this amounts to 52 counter registers and 319 events, not counting the TSC.

All counters are accessible from the running thread by reading special CPU registers. To enable a lower latency access mode the UPC counters can be mapped into user memory. The hardware counters are also accessible from the management network, JTAG, which allows for remote non-intrusive system wide counter monitoring.

2.2 `BGLperfctr` API

To allow for user level access to the UPC a low-level API was designed that abstracts some of the hardware specific aspects of the IBM BlueGene/L performance counter infrastructure. The implemented `BGLperfctr` interface defines access to the hardware counters by abstract names of the events as illustrated in table 1. The API also hides hardware implementation details such as possible mappings of counter events on to physical counters, free vs. occupied counters and the generation of control words for the UPC and the FPU counters.

Table 1. Example of IBM BlueGene/L specific events

`BGLperfctr` name	Description
BGL_FPU_ARITH_OEDIPUS_OP	All symmetric, asymmetric, and complex Oedipus multiply-add instructions
BGL_UPC_L3_CACHE_HIT	Hit in level 3 cache
BGL_UPC_TR_RCV_2_VC0_DPKT_RCV	Data packet arrived in tree receiver 2, channel 0
BGL_UPC_TS_YM_32B_CHUNKS	32 byte chunk sent in torus channel Y-

Counters are controlled by the user to either count the rising or falling edge of the condition, or alternatively to count the elapsed number of UPC clock cycles where the condition is one of either true or false. The counter infrastructure has a clock cycle that is twice the cycle of the CPU cores (350MHz vs. 700MHz).

The counters are virtualized by `BGLperfctr` into 64 bit unsigned counters to prevent counter overflow. These virtual counters are updated from the physical counters when a `BGLperfctr` read instruction is issued. In theory there is a risk of alias errors unless the virtual counters are updated within a 6.135s period since the UPC and FPU physical counters are only 32 bit wide and there can potentially occur up to one count per core clock cycle in a counter. The `BGLperfctr` library can set up a timer based interrupt to update the virtual counters at a predefined interval 0.1s short of the undersample limit. In a large scale parallel run there is a potential performance problem if these timer based interrupts are not synchronized among all the computational nodes participating in a run, c.f. Petrini et al. [4]. For this reason the `BGLperfctr` library optionally can initiate the timer in a synchronized fashion across the nodes by issuing a hardware assisted barrier at library initiation.

In the IBM BlueGene/L compute node chip the counter hardware is a single resource in the SOC design. This is reflected in the `BGLperfctr` library as its datastructure is a single instantiation located in a part of SRAM memory that is shared between the two PPC440d cores. Any modification to the counter

Table 2. IBM BlueGene/L-specific PAPI events. The PAPI event name is listed with a short description and a list of the hardware events used for the metric.

`BGLperfctr` Name	Description
Events used	
PAPI_BGL_OED	Oedipus instructions executed in FPU 0
BGL_FPU_ARITH_OEDIPUS_OP	
PAPI_BGL_TS_32B	32 byte data chunks sent in any direction
BGL_UPC_TS_XM_32B_CHUNKS BGL_UPC_TS_XP_32B_CHUNKS BGL_UPC_TS_YM_32B_CHUNKS BGL_UPC_TS_YP_32B_CHUNKS BGL_UPC_TS_ZM_32B_CHUNKS BGL_UPC_TS_ZP_32B_CHUNKS	
PAPI_BGL_TS_FULL	Cycles where any torus channel is waiting for tokens
BGL_UPC_TS_XM_LINK_AVAIL_NO_VCD0_VCD_VCBN_TOKENS BGL_UPC_TS_XP_LINK_AVAIL_NO_VCD0_VCD_VCBN_TOKENS BGL_UPC_TS_YM_LINK_AVAIL_NO_VCD0_VCD_VCBN_TOKENS BGL_UPC_TS_YP_LINK_AVAIL_NO_VCD0_VCD_VCBN_TOKENS BGL_UPC_TS_ZM_LINK_AVAIL_NO_VCD0_VCD_VCBN_TOKENS BGL_UPC_TS_ZP_LINK_AVAIL_NO_VCD0_VCD_VCBN_TOKENS	
PAPI_BGL_TR_DPKT	Number of data packets sent in any tree channel
BGL_UPC_TR_SNDR_2_VC1_DPKTS_SENT BGL_UPC_TR_SNDR_2_VC0_DPKTS_SENT BGL_UPC_TR_SNDR_1_VC1_DPKTS_SENT BGL_UPC_TR_SNDR_1_VC0_DPKTS_SENT BGL_UPC_TR_SNDR_0_VC1_DPKTS_SENT BGL_UPC_TR_SNDR_0_VC0_DPKTS_SENT	
PAPI_BGL_TR_FULL	Cycles with any tree channel full
BGL_UPC_TR_RCV_0_VC0_FULL BGL_UPC_TR_RCV_0_VC1_FULL BGL_UPC_TR_RCV_1_VC0_FULL BGL_UPC_TR_RCV_1_VC1_FULL BGL_UPC_TR_RCV_2_VC0_FULL BGL_UPC_TR_RCV_2_VC1_FULL	

set-up made using `BGLperfctr` in either of the cores is immediately visible to the other core. Efficient locking routines are used to ensure data integrity of the `BGLperfctr` state.

2.3 IBM BlueGene/L PAPI

The IBM BlueGene/L PAPI implementation is derived from the standard PAPI distribution with some minor modifications. The implementation fully supports the PAPI event set concept. The user defines what events are to be in a set and operations such as start, stop and read are performed on these event sets.

PAPI is divided into a platform independent library and a platform dependent substrate. Porting PAPI to a new platform is equivalent to implementing a new PAPI substrate for the computer platform at hand.

In the IBM BlueGene/L PAPI port some small modifications are made to the platform independent library: The PAPI library initiation is a synchronizing operation on IBM BlueGene/L and there are five new PAPI predefined events as shown in table 2.

3 Results

Results presented in this article are based on PAPI 2.3.4.3.bgl [5] on a IBM BlueGene/L system running the V1R3 system software release.

3.1 PAPI Timings

Timings of PAPI operations on various event sets are reported in table 3. Measurements were made using the `cost.c` program of the PAPI verification suite and are based on the elapsed time of $50,000$ repeats of the operation. Due to the single thread architecture of IBM BlueGene/L variance within the individual repeated operations is minimal, but was not measured. Event sets were selected to utilize various number of PAPI events (both predefined and native) that each utilizes one or more hardware events.

The listed results were matched against a simple cost model

$$T = T_0 + N_{UPC} \cdot T_{UPC} + N_{FPU} \cdot T_{FPU} \tag{1}$$

using linear regression to establish an approximate cost break-down in PAPI internal costs and the cost of operating on UPC and FPU counters respectively. An ANOVA analysis was performed to establish the confidence in the obtained parameters. The resulting estimates are tabulated in table 4. Although based on a small number of event set configurations it is clearly seen that the time to read an UPC counter through the memory mapped access method is significantly lower than the time to read an FPU counter. A model including the number of actual PAPI events in addition to the terms in (1) to try to differentiate with PAPI event handling cost and `BGLperfctr` cost was rejected due to a high p-value (0.195) for this added parameter.

Table 3. Cycle counts for operations on various event set configurations. The number of PAPI events in the event set as well as the number of underlying hardware events (UPC and FPU) are listed with the corresponding average time for a `PAPI_start()/PAPI_stop()` sequence and a `PAPI_read()` respectively.

Events			cyc/call	
PAPI	UPC	FPU	start/stop	read
1	1	0	9845.34	2100.01
2	2	0	14644.49	2558.01
1	6	0	31755.34	3458.01
2	12	0	62963.75	6202.01
2	1	1	17975.87	4187.56
2	6	1	40847.13	5613.78
2	0	2	19386.43	5678.92
4	0	4	33746.24	9563.47

Table 4. Approximate cost break-down for PAPI operations according to the model Eqn. (1) on the measured data in Table 3. Cost T_x is measured in the unit of core CPU cycles and listed with the associated ANOVA confidence value p.

	start/stop [1/cyc]	p	read [1/cyc]	p
T_0	4939	$7.8 \cdot 10^{-4}$	1727	$1.2 \cdot 10^{-4}$
T_{UPC}	4771	$9.2 \cdot 10^{-8}$	352	$3.1 \cdot 10^{-5}$
T_{FPU}	7258	$2.6 \cdot 10^{-6}$	1960	$1.4 \cdot 10^{-6}$

3.2 Compute Partition Wide Synchronization

On a large scale parallel system, asynchronous activity may be detrimental to high performance [4]. On the IBM BlueGene/L system care has been taken to eliminate disturbances by having the compute nodes run a light-weight kernel with one single thread per CPU core. To prevent overflow of the performance counters each running counter is sampled into a 64 bit wide virtual counter. To minimize disturbance in the system this sampling is made synchronously on all compute nodes using a timer interrupt based on a local timer. The timer is started after a global synchronization point. In Figure 1 the global and local spread of this operation is shown. The experiment was run on a 32 node compute partition. The figure shows the spread in the TSC among the nodes at arrival in the interrupt handler (global spread). Spread is defined as

$$S^i_{global} = \max_{p=0}^{N_p-1}(T^i_p) - \min_{p=0}^{N_p-1}(T^i_p) \tag{2}$$

$$S^i_{local} = \max_{p=0}^{N_p-1}(T^i_p - T^0_p) - \min_{p=0}^{N_p-1}(T^i_p - T^0_p), \tag{3}$$

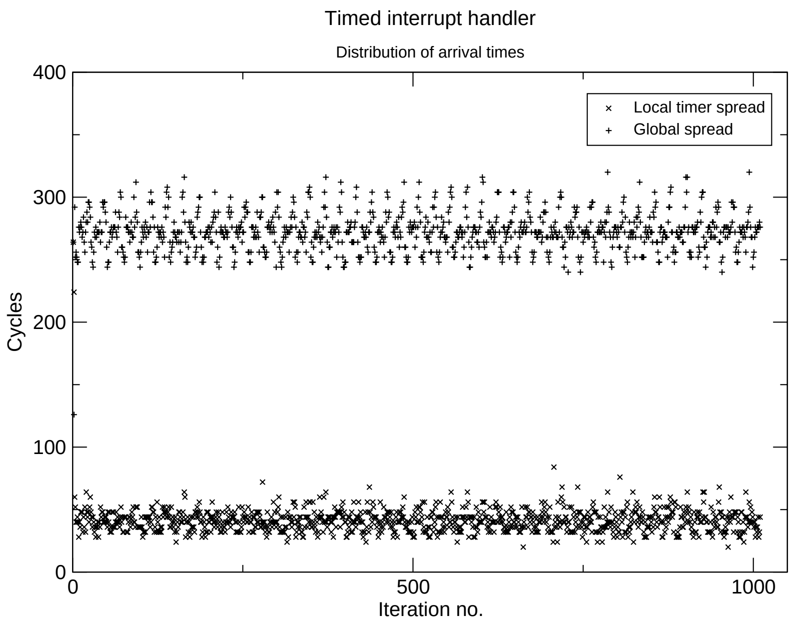


Fig. 1. Value of the TSC at arrival of the interrupt handler routine for 1000 interrupts using a 6.1s interval timed interupt on a 32 node run. See text for details.

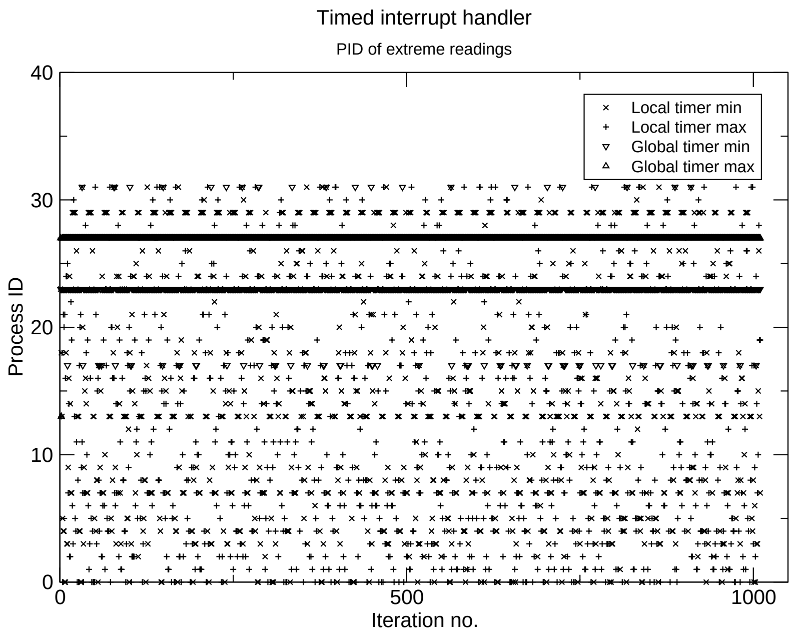


Fig. 2. Process id of the processor contributing to equations (2-3)

where T_p^i denotes the value of the TSC in node p at arrival in the interrupt handler in iteration i. The TSC is started at 0 after a global barrier during the compute node boot. $i = 0$ refers to an explicit call made to the interrupt handler just after setting up the timed interrupt.

Also, to illustrate the influence of the synchronization itself, a local spread is shown. The local spread is defined in the same way as the global spread, but using the difference of the TSC in the interrupt handler and the value of the TSC after setting the interrupt timer.

The experiment shows that the spread is very small, in global as well as local context. In particular the spread is much smaller than the time it takes to read values from the hardware counters and update the virtual counters. In Figure 2 the processor ID of the processors contributing to the spread definition (2-3) is plotted for each interrupt. This figure emphasizes the accuracy of the timer interrupt as it shows a variation in local spread whereas the global spread shows a systematic offset.

4 Conclusion

We implemented a low-level kernel-specific API that exposed the UPC and FPU counters as well as the TS counter to the user. Using this `BGLperfctr` API we successfully implemented a PAPI substrate for IBM BlueGene/L that is able to take full advantage of the rich set of counters of many different types on IBM BlueGene/L. The open design and flexibility of the native event interface in PAPI did not restrict the use of platform-specific counters, such as network utilization counters. Further, the IBM BlueGene/L implementation of PAPI allows the user to use the full set of `BGLperfctr` predefined names for such events in the creation of PAPI event groups.

Acknowledgement

The IBM BlueGene/L PAPI port was made as a part of an Academic Visitor's program to IBM TJ Watson Laboratories, Yorktown Heights, NY while the author was on leave from the Center for Parallel Computers at the Royal Institute of Technology, Stockholm, Sweden. The author wishes to thank IBM TJ Watson for the opportunity to work in the BlueGene development team and numerous colleagues and friends within IBM for invaluable support while performing this work. The author is also thankful for being granted leave from the Center for Parallel Computers to conduct this development work.

References

1. Browne, S., Dongarra, S., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications 14(3), 189–204 (2000)

2. Adiga, N.R., Almasi, G., Almasi, G.S., Aridor, Y., Barik, R., Beece, D., Bellofatto, R., Bhanot, G., Bickford, R., Blumrich, M., Bright, A.A., Brunheroto, J., Caşcaval, C., Castaños, J, Chan, W., Ceze, L., Coteus, P., Chatterjee, S., Chen, D., Chiu, G., Cipolla, T.M., Crumley, P., Desai, K.M., Deutsch, A., Domany, T., Dombrowa, M.B., Donath, W., Eleftheriou, M., Erway, C., Esch, J., Fitch, B., Gagliano, J., Gara, A., Garg, R., Germain, R., Giampapa, M.E., Gopalsamy, B., Gunnels, J., Gupta, M., Gustavson, F., Hall, S., Haring, R.A., Heidel, D., Heidelberger, P., Herger, L.M., Hoenicke, D., Jackson, R.D., Jamal-Eddine, T., Kopcsay, G.V., Krevat, E., Kurhekar, M.P., Lanzetta, A.P., Lieber, D., Liu, L.K., Lu, M., Mendell, M., Misra, A., Moatti, Y., Mok, L., Moreira, J.E., Nathanson, B.J., Newton, M., Ohmacht, M., Oliner, A., Pandit, V., Pudota, R.B., Rand, R., Regan, R., Rubin, B., Ruehli, A., Rus, S., Sahoo, R.K., Sanomiya, A., Schenfeld, E., Sharma, M., Shmueli, E., Singh, S., Song, P., Srinivasan, V., Steinmacher-Burow, B.D., Strauss, K., Surovic, C., Swetz, R., Takken, T., Tremaine, R.B., Tsao, M., Umamaheshwaran, A.R., Verma, P., Vranas, P., Ward, T.J.C., Wazlowski, M.: An Overview of the IBM BlueGene/L Supercomputer. In: Proc. ACM/IEEE conf supercomputing, pp. 1–22 (2002), `http://acm.supercomputing.org/sc2002/`
3. Martorell, X., Smeds, N., Walkup, R., Brunheroto, J.R., Almasi, G., Gunnels, J.A., DeRose, L., Labarta, J., Escale, F., Gimenez, J., Servat, H., Moreira, J.E.: Blue Gene/L performance tools. IBM J Res. Devel. 49, 407–424 (2005)
4. Petrini, F., Kerbyson, K., Pakin, S.: The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In: Proc. ACM/IEEE conf. supercomputing, pp. 1–17 (2003), `http://acm.supercomputing.org/sc2003/`
5. PAPI homepage and download site, `http://icl.cs.utk.edu/papi/`

Numerical Simulation of 3D Acoustic Logging*

Victor I. Kostin[1], Dmitry V. Pissarenko[2], Galina V. Reshetova[3], and Vladimir A. Tcheverda[4]

[1] ZAO "Intel A/O"
victor.i.kostin@intel.com
[2] Schlumberger Moscow Research Center, Russia
DPissarenko@moscow.oilfield.slb.com
[3] Institute of Numerical Mathematics and Mathematical geophysics SD RAS, Russia
kgv@nmsf.sscc.ru
[4] Institute of Geophysics SD RAS, 3, prosp. Koptyug, 630090, Novosibirsk, Russia
chev@uiggm.nsc.ru

Abstract. A Finite-difference method for simulation of sonic waves propagating in a vicinity of a borehole filled with fluid and surrounded by a 3D heterogeneous elastic medium is developed. The method utilizes an explicit second order of approximation FD scheme on staggered grids that approximates the elastodynamic system of equations in cylindrical coordinates. A computational domain is surrounded by a special Perfectly Matched Layer for cylindrical coordinate system designed to attenuate waves reflected from outer boundaries. Parallelization is based on a domain decomposition approach and implemented with the help of MPI library. Results of numerical experiments are presented.

1 Introduction

Full wave acoustic logs are very important for borehole measurements, providing information about physical properties of surrounding rocks. Historically these methods were based on the use of axially symmetric, or monopole, wave phenomena in a fluid-filled borehole ([2], [4], [12]). However, real media themselves are 3D heterogeneous; so, axial symmetry is somehow a restricting assumption. Recently, logging tools based on excitation and reception of non-axially symmetric wave phenomena have been developed and used in order to explore near borehole media ([10]). This implies the actuality of developing specific software destinated to numerical experimentation for a variety of 3D models.

Previous 3D FD studies that were made for Cartesian coordinates (see, for example, [5] and [11]) are not very useful for the problem under consideration. Indeed, if saw-like approximation of the interface between the fluid-filled borehole and the enclosing rocks was used, the interface in its turn would provoke the generation of a rather strong artifact known as "numerical scattering". An

* This research was partially supported by Russian Foundation of Basic Research, grants 05-05-64277, 06-05-64748 and 07-05-00538.

attempt to reduce the artifact by use of smaller spatial grid steps may conceivably lead to huge RAM demands and dramatic (or even catastrophic) slowing down of computations.

We suggest instead the use of a cylindrical coordinate system which is chosen to be co-axial with the borehole. Our approach possesses two key items advantageously distinguishing it from others that also utilize a cylindrical coordinate system for 3D numerical simulation of acoustic logs (see, for example, [3], [9]):

– periodical azimuthal refinement of spatial grid cells in order to avoid their inflation with radius increase;
– the implementation of Perfectly Matched Layer (PML) without azimuth splitting.

2 Statement of the Problem

The geometry of the problem under consideration is represented by a cylindrical tube of radius R, which is filled with some liquid and embedded in a heterogeneous elastic medium. Most interesting are waves propagating in some vicinity of the tube. This implies choosing a cylinder as a computational domain and cylindrical coordinates as an appropriate coordinate system.

Propagation of sonic waves in heterogeneous elastic media is governed by a t-hyperbolic system of partial differential equations for velocity vector $\boldsymbol{u} = (u_r, u_\phi, u_z)^T$ and stress "vector" $\boldsymbol{\sigma} = (\sigma_{rr}, \sigma_{\phi\phi}, \sigma_{zz}, \sigma_{r\phi}, \sigma_{\phi z}, \sigma_{rz})^T$. In a cylindrical coordinate system, the system of equations looks as follows:

$$\varrho \frac{\partial \boldsymbol{u}}{\partial t} = A \frac{\partial \boldsymbol{\sigma}}{\partial r} + \frac{1}{r} B \frac{\partial \boldsymbol{\sigma}}{\partial \phi} + C \frac{\partial \boldsymbol{\sigma}}{\partial z} + \frac{1}{r}(A - D)\boldsymbol{\sigma} \tag{1}$$

$$M \frac{\partial \boldsymbol{\sigma}}{\partial t} = A^T \frac{\partial \boldsymbol{u}}{\partial r} + \frac{1}{r} B^T \frac{\partial \boldsymbol{u}}{\partial \phi} + C^T \frac{\partial \boldsymbol{u}}{\partial z} + \frac{1}{r} D^T \boldsymbol{u} + \boldsymbol{F}(r, \phi, z; t) \tag{2}$$

For completeness, respective initial values for these vector functions should be assigned at $t = 0$ and appropriate boundary conditions posed on the outer boundary. The same system of partial differential equations is used also for modelling acoustic waves in a liquid. This makes the FD scheme and its respective computations simpler though it takes some extra time and memory. In order to correspond to fluid, Lame parameters and stress tensor components should be treated respectively. This is a well known common trick and we don't pay more attention to this specificity. Matrices A, B, C, D and M are the following:

$$A = \begin{pmatrix} 1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0\,1 \end{pmatrix}; \quad B = \begin{pmatrix} 0\,0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0 \end{pmatrix}; \quad C = \begin{pmatrix} 0\,0\,0\,0\,0\,1 \\ 0\,0\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0 \end{pmatrix};$$

$$D = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}^T \quad M = \frac{1}{\mu} \begin{pmatrix} \frac{\lambda+\mu}{3\lambda+2\mu} & -\frac{\lambda}{2(3\lambda+2\mu)} & -\frac{\lambda}{2(3\lambda+2\mu)} & 0\,0\,0 \\ -\frac{\lambda}{2(3\lambda+2\mu)} & \frac{\lambda+\mu}{3\lambda+2\mu} & -\frac{\lambda}{2(3\lambda+2\mu)} & 0\,0\,0 \\ -\frac{\lambda}{2(3\lambda+2\mu)} & -\frac{\lambda}{2(3\lambda+2\mu)} & \frac{\lambda+\mu}{3\lambda+2\mu} & 0\,0\,0 \\ 0 & 0 & 0 & 1\,0\,0 \\ 0 & 0 & 0 & 0\,1\,0 \\ 0 & 0 & 0 & 0\,0\,1 \end{pmatrix}$$

Let a sonic source be simulated as the volumetric point source

$$\boldsymbol{F}(r,\phi,z;t) = \frac{(\lambda+\mu)f(t)}{\mu(3\lambda+2\mu)} \frac{\delta(r-r_0,\phi-\phi_0,z-z_0)}{2\pi r}(1,1,1,0,0,0)^T$$

located at some point (r_0,ϕ_0,z_0) within the liquid or elastic medium, but not on their interface.

In our further experiments we assume zero initial data:

$$\boldsymbol{u}\Big|_{t=0} = 0; \quad \boldsymbol{\sigma}\Big|_{t=0} = 0 \tag{3}$$

though nothing prevents the use of other data. On the interface between fluid and solid media, i.e. on the tube inner surface, one should claim continuity of the normal component of the velocity vector $\boldsymbol{u}$ and the vector of normal stresses $\boldsymbol{\sigma}_r = (\sigma_{rr}, \sigma_{r\phi}, \sigma_{rz})^T$:

$$u_r\Big|_{r=R-0} = u_r\Big|_{r=R+0}; \quad \boldsymbol{\sigma}_r\Big|_{r=R-0} = \boldsymbol{\sigma}_r\Big|_{r=R+0} \tag{4}$$

In the FD computations, the system of partial differential equations (1) - (2) is approximated by a centered finite difference scheme on staggered grids. This leads to a well studied explicit finite-difference scheme ([13]). One of its advantages is the second order of approximation.

The cylindrical coordinate system has the peculiarity of inflating azimuthal grid steps with an increasing radius. In order to adjust this inflation, we perform periodical refinement of azimuth sampling: the step with respect to the azimuth is halved as soon as the radius is doubled (Fig.1). In Fig.1 one can see the mutual disposition of coarse (A) and fine (B) grids. To couple these grids, one should fill up the gaps at points marked as diamonds. At these points, the values of velocities and stress tensor components can not be computed on the coarse grid, but they are needed in order to keep computations on the fine grid.

We take into account the 2π-periodicity of wavefields with respect to azimuth and interpolate the components into the missed points. This interpolation is implemented by the use of a Discrete Fourier Transform with respect to the azimuth variable, and consequently keeps the second order of approximation. Actually, accuracy of such interpolation is much better because of the smoothness of our functions. Moreover, overhead of DFT can be neglected if FFT algorithms are used.

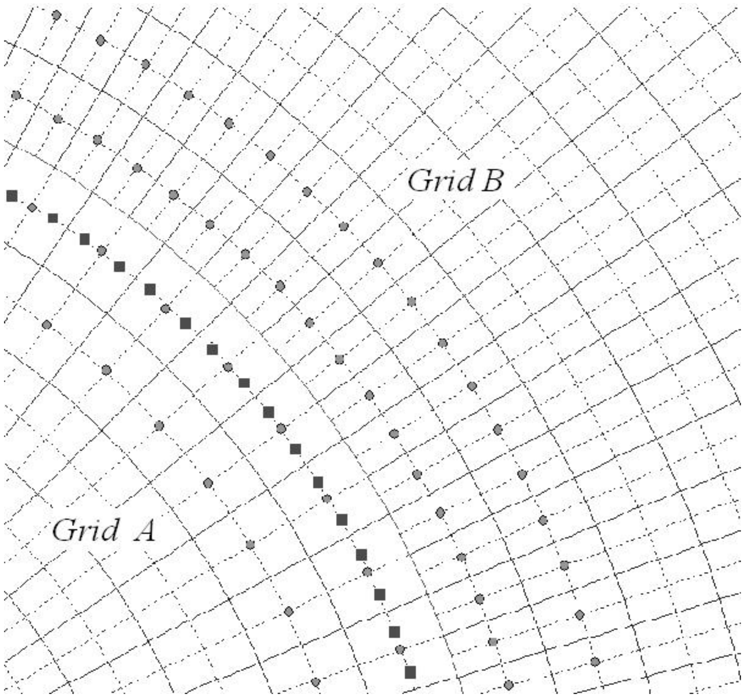


Fig. 1. Coarse (A) and fine (B) grid coupling. In order to move from coarse to fine grid, the velocity and stress vectors at diamonds are to be interpolated. This is done by means of DFT based interpolation.

3 Perfectly Matched Layers (PML) for Cylindrical Coordinates

Waves initiated somewhere in a bounded computational domain while propagating through the domain will reach the outer boundary and produce some reflected waves. If numerical simulation in the unbounded domain is the goal, those reflected waves are artifacts and their impact should be diminished. For these purposes we use the PML technique originally introduced for Maxwell equations in [1] and later generalized for elasticity ([6], [9]). The technique gets its name due to usage of a special layer (called Perfectly Matched Layer) which separates the computational domain from the outer boundary. Within the PML, some equations are constructed which are specially designed to attenuate waves propagating along specific axes. These artificial equations are to be coupled with the governing equations on the interfaces between the physical domain and the PML, so that no reflection arises on the interface.

Let us note, however, that strictly speaking the equations within the PML do not correspond to any physical medium and their usage is justified by its ability to attenuate waves propagating within the PML. The price for this is overhead due to the augmentation of the computational domain with PMLs, where the numbers of unknown functions and equations increase. Depending on the geometry, the overhead can reach 50% or more. In some configurations the use of PMLs may lead to instability. That means that the method should be applied with care.

Recall that we are interested in processes in some vicinity of the tube. In other words, we assume the cylindrical computational domain to be rather thin. In such a case the directions of outgoing waves on the interface between the physical domain and the PML will be mostly orthogonal to the azimuthal direction. This is the justification of our choice of the PML developed for the case with axial symmetry. Our PMLs are described in more detail below.

The attenuation with respect to z is introduced in exactly the same way as for cartesian coordinates. So, let us focus on the attenuation with respect to r. In the PML we introduce ([7]) the new unknown vector functions $\boldsymbol{u}^{\perp}, \boldsymbol{u}^{\|}, \boldsymbol{\sigma}^{\perp}, \boldsymbol{\sigma}^{\|}$ which should satisfy the following system of equations:

$$\varrho \frac{\partial \boldsymbol{u}^{\|}}{\partial t} = C \frac{\partial \boldsymbol{\sigma}}{\partial z}; \tag{5}$$

$$\left(\frac{\partial}{\partial t} + \alpha(r)\right) \varrho \boldsymbol{u}^{\perp} = A \frac{\partial \boldsymbol{\sigma}}{\partial r} + \frac{1}{r} B \frac{\partial \tilde{\boldsymbol{\sigma}}}{\partial \phi} + (A - D) \frac{\tilde{\boldsymbol{\sigma}}}{r}; \tag{6}$$

$$M \frac{\partial \boldsymbol{\sigma}^{\|}}{\partial t} = C^T \frac{\partial \boldsymbol{u}}{\partial z}; \tag{7}$$

$$\left(\frac{\partial}{\partial t} + \alpha(r)\right) M \boldsymbol{\sigma}^{\perp} = A^T \frac{\partial \boldsymbol{u}}{\partial r} + \frac{1}{r} B^T \frac{\partial \tilde{\boldsymbol{u}}}{\partial \phi} + D^T \frac{\tilde{\boldsymbol{u}}}{r}. \tag{8}$$

Here we used denotations

$$\boldsymbol{u} = \boldsymbol{u}^{\perp} + \boldsymbol{u}^{\|}; \quad \boldsymbol{\sigma} = \boldsymbol{\sigma}^{\perp} + \boldsymbol{\sigma}^{\|}.$$

Vector functions with tildes $\tilde{\boldsymbol{u}}$ and $\tilde{\boldsymbol{\sigma}}$ satisfy to ordinary (indeed, there are no differentiations with respect to spatial directions) differential equations with respect to t at each point of the PML:

$$\frac{\partial \tilde{\boldsymbol{u}}}{\partial t} + \frac{\beta(r)}{r} \tilde{\boldsymbol{u}} = \frac{\partial \boldsymbol{u}}{\partial t} + \alpha(r) \boldsymbol{u}; \quad \frac{\partial \tilde{\boldsymbol{\sigma}}}{\partial t} + \frac{\beta(r)}{r} \tilde{\boldsymbol{\sigma}} = \frac{\partial \boldsymbol{\sigma}}{\partial t} + \alpha(r) \boldsymbol{\sigma}. \tag{9}$$

Here $\beta(r) = \int\limits_{R_0}^{r} \alpha(\xi) d\xi$ with $\alpha(r) > 0$ for $r > R_0$ being a so called "damping" function that provides an exponential decay of waves within the PML.

For completeness the following gluing conditions on the interface $r = R_0$

$$\boldsymbol{u}\Big|_{r=R_0-0} = \left(\boldsymbol{u}^{\perp} + \boldsymbol{u}^{\|}\right)\Big|_{r=R_0+0}; \quad \boldsymbol{\sigma}\Big|_{r=R_0-0} = \left(\boldsymbol{\sigma}^{\perp} + \boldsymbol{\sigma}^{\|}\right)\Big|_{r=R_0+0} \tag{10}$$

and some correct boundary conditions on the outer boundary $r = R_1$ should be posed. We use

$$\boldsymbol{u}^{\perp}\Big|_{r=R_1} = 0; \quad \boldsymbol{\sigma}^{\perp}\Big|_{r=R_1} = 0. \tag{11}$$

In order to figure out an effect of the waves attenuation with respect to the r provided by (5) - (11) we apply a Fourier transform with respect to time and sum up the equations for perpendicular and parallel components of velocity and stress vectors. The result is presented as the following system of partial differential equations

$$-i\omega\varrho\hat{\boldsymbol{u}} = A\frac{\partial\hat{\boldsymbol{\sigma}}}{\partial\tilde{r}} + \frac{1}{\tilde{r}}B\frac{\partial\hat{\boldsymbol{\sigma}}}{\partial\phi} + C\frac{\partial\hat{\boldsymbol{\sigma}}}{\partial z} + \frac{1}{\tilde{r}}(A-D)\boldsymbol{\sigma} \qquad (12)$$

$$-i\omega M\hat{\boldsymbol{\sigma}} = A^T\frac{\partial\hat{\boldsymbol{u}}}{\partial\tilde{r}} + \frac{1}{\tilde{r}}B^T\frac{\partial\hat{\boldsymbol{u}}}{\partial\phi} + C^T\frac{\partial\hat{\boldsymbol{u}}}{\partial z} + \frac{1}{\tilde{r}}D^T\hat{\boldsymbol{u}} \qquad (13)$$

with complex-valued radius $\tilde{r} = r + \frac{i}{\omega}\int\limits_{r_0}^{r}\alpha(\xi)d\xi$. This variable is complex valued and this is the only difference with respect to the regular elastic wave equation in homogeneous media. It is well known that any wave in homogeneous elastic media in a cylindrical coordinate system can be represented as a series of Hankel's functions $H_j^{(1)}(kr)$. Such outgoing waves will generate the same series (i.e. the series with the same coefficients) in the PML zone except for the functions $H_j^{(1)}(k\tilde{r})$. No reflection arises on the interface between the elastic medium and the PML, or, in other words, the PML "perfectly matches" the elastic medium. The functions $H_j^{(1)}(k\tilde{r})$ exponentially decay as r increases due to the positiveness of the imaginary part of the variable $\tilde{r}$. The rate of their decrease depends on the choice of damping function $\alpha(r)$. However, it should be noted that though for continuous equations, the faster $\alpha(r)$ grows the better "quality" of the PML, it is no longer true for FD approximation. This is a known peculiarity in the theory of the PMLs that is confirmed also in our case. When the wave initiated within the elastic medium and passed through interface between the medium and PML reaches the outer boundary it generates a reflected wave which propagates inwards the computational domain. This reflected wave is expressed as a series of functions $H_j^{(2)}(k\tilde{r})$ with respective reflection coefficients and going through the PML will reach the target elastic zone as artifacts. Though we have no general proof justifying the smallness of artifacts, the numerical examples below demonstrate their good quality.

4 Parallel Implementation

For the implementation of parallel computations we use an approach based on domain decomposition of the target area. The total 3D model is sliced into a number of disc-like subdomains Ω_i. Each of these subdomains is assigned to a separate Processor Units (PU). A finite difference scheme assumes communication between neighboring processors (Fig.2), requiring them to exchange function values on the interfaces between slices. This communication is arranged with the help of a Message Passing Interface library. The same approach, but for simulation of 2D elastic waves propagation, was presented in [8]. The very important

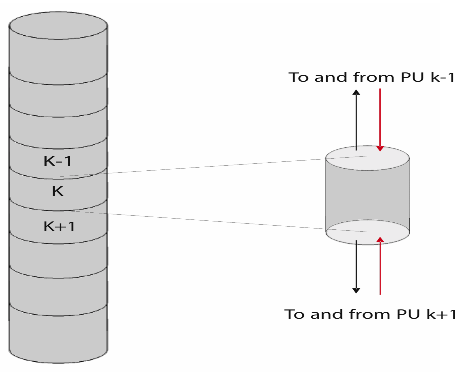


Fig. 2. Domain decomposition and exchange between neighboring PUs

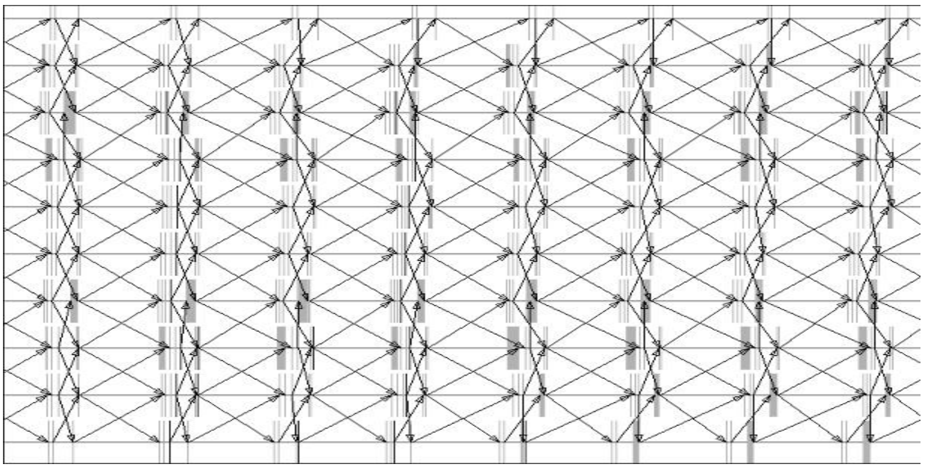


Fig. 3. Diagram of productive (horizontal line) and idle (vertical bar) times for a cluster made of ten processor units. The arrows point out the directions of data exchange between the PUs.

peculiarity of the problem under consideration, besides its essential 3D nature, is the comparatively small amount of data for the processors to exchange.

In order to visualize the balance of productive and idle times for each processor unit we use Jumpshot-4 (http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm). For the example described in the next section, the result is represented in Fig.3. In this diagram each horizontal line corresponds to the state of a single PU at each instant of time. Its waiting time is indicated by the vertical grey bar, all other time is productive. Arrows indicate the directions of data exchange between the PUs.

One can estimate from this diagram that overall efficiency is around 90%, which is a very good result.

5 Numerical Experiments

The series of numerical experiments were performed for a range of source frequencies, source positions and models of surrounding elastic media. For illustration let us consider the model that represents realistic structure of a casing well:

- Background - homogeneous elastic medium with wave propagation velocities $V_p = 3500$ m/s, $V_s = 2000$ m/s and density $\varrho = 2300\,kg/m^3$;
- Vertical borehole of diameter 0.2 m is filled with a mud with parameters $V_f = 1500$ m/s and $\varrho_f = 1000\,kg/m^3$;
- There is a steel tube encircling borehole; its wall thickness is equal to 0.015 m, wave propagation velocities $V_p = 5600$ m/s, $V_s = 3270$ m/s and density $\varrho = 7830\,kg/m^3$;
- There is a casing around the steel tube; its thickness is equal to 0.055 m, its elastic parameters are the following: $V_p = 4200$ m/s, $V_s = 2425$ m/s and density $\varrho = 2400\,kg/m^3$

All simulations were done with the source function $f(t)$ being the Ricker impulse

$$f(t) = \left[1 - 2\nu_0^2\pi^2\left(1 - \frac{1}{\nu_0^2}\right)\right] \exp\left\{-\nu_0^2\pi^2(t - \frac{1}{\nu_0^2})\right\}$$

with dominant frequency $\nu_0 = 10000$ Hz.

The target area is a cylinder with radius $R = 0.75$ meters and length $L = 3$ meters. Spatial approximation of the problem was performed on the grid with steps around 0.0025 meters. Simple calculations gives the total number of grid points within the target area to about $2.8 * 10^8$, that means that the RAM demand for the target area is about 15 Gb (one should store 12 values at each grid point - three elastic parameters, three components of the velocity vector and six components of the stress vector). The total RAM demand is higher because of presence of PML area and comes to 20 Gb. The time step needs to obey Courant's condition and happens to be around $4.5 * 10^{-7}$ sec. So, in order to perform simulation up to 0.003 sec one has to do 6000 steps.

Computations were done on a 20 nodes 2 way Itanium2 based cluster with Infiniband network. Each simulation took about 10 - 12 hours. In Fig.4, we present a series of snapshots for the vertical velocity u_z for 3 different source positions: axial source position (a), eccentric source, but still within borehole (b) and, finally, eccentric source placed outside of the borehole (c). One can observe great differences in the wavefields presented in the snapshots and also the wonderful quality of PMLs. The snapshots include the grid points belonging to the PML and, so, respective overhead due to use of these zones can be estimated.

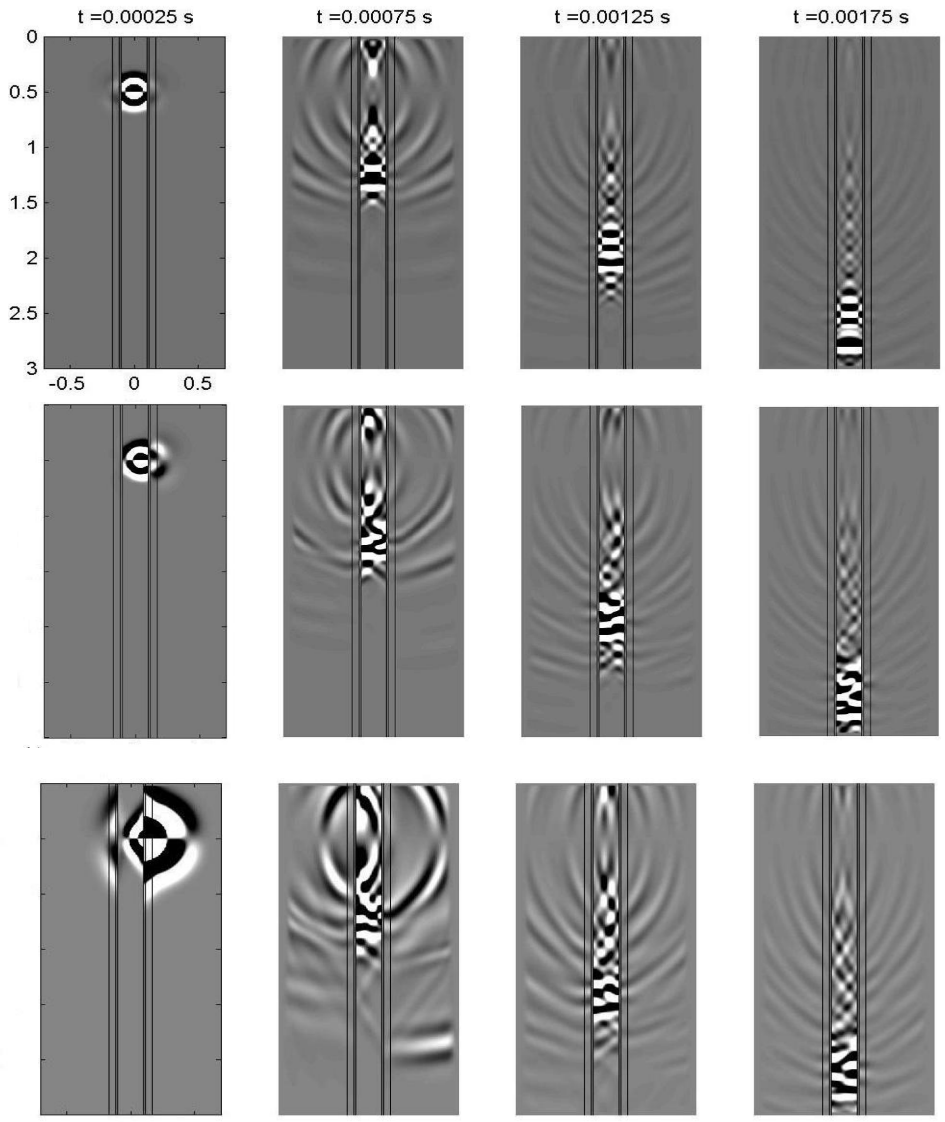


Fig. 4. Synthetic sonic logging: vertical velocity for three source positions. From top to bottom: axial, eccentric within borehole, eccentric within casing. Vertical lines present borehole/steel and steel/casing interfaces.

6 Conclusion

To conclude, we would like to emphasize that only a combination of all approaches described above provides success in the numerical simulation of the problem:

– The geometry of the problem allows slicing, which leads to straightforward and efficient parallelization on clusters;
– Our version of PML effectively attenuates waves reflected from the outer artificial boundaries;
– Periodical halving azimuth grid steps reduced the problem size to acceptable values.

Modern powerful computational systems like clusters sufficiently extend the set of solvable application problems. We should notice, however, that further complicating the problem, which seems to be of current importance for modern industry, may require additional approaches to be developed.

References

1. Berenger, J.P.: A perfectly matched layer for the absorption of electromagnetic waves. Journal of Computational Physics 114, 185–200 (1994)
2. Biot, M.A.: Propagation of elastic waves in a cylindrical bore containing a fluid. Journal of Applied Physics 23, 997–1005 (1952)
3. Chen, Y.-H., Chew, W.C., Liu, Q.-H.: A three-dimensional finite difference code for the modeling of sonic logging tools. Journal of Acoustical Society of America 2, 702–712 (1997)
4. Cheng, C.H., Toksoz, M.N.: Elastic wave propagation in a fluid-filled borehole and synthetic acoustic logs. Geophysics 46(7), 1042–1053 (1981)
5. Cheng, N., Cheng, C.H., Toksoz, M.N.: Borehole wave propagation in three dimensions. Journal of Acoustical Society of America 97(6), 3483–3493 (1995)
6. Collino, F., Tsogka, C.: Application of PML absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media. Geophysics 66, 294–307 (2001)
7. Kostin, V.I., et al.: Radiation of seismic waves by source located within fluid-filled borehole. Fizicheskaja Mezomehanika (in Russian) 5, 85–92 (2002)
8. Korneev, V.I., et al.: Modelling of Seismic waves propagation for 2D media (Direct and inverse problems). In: Malyshkin, V. (ed.) Parallel Computing Technologies. LNCS, vol. 1277, pp. 350–357. Springer, Heidelberg (1997)
9. Liu, Q.H., Sinha Bikash, K.: A 3D cylindrical PML/FDTD method for elastic waves in fluid-filled pressurized boreholes in triaxially stresed formations. Geophysics 68(5), 1731–1743 (2003)
10. Pistre, V., Plona, T., Sinha, B., et al.: Estimation of 3D borehole acoustic rock properties using a modular sonic tool. In: Extended Abstracts of EAGE 67-th Conference and Exhibition, Madrid, Spain, June 13-16, 2005, vol. I019 (2005)
11. Liu, Q.-H., Schoen, E., Daube, F., et al.: A three-dimensional finite-difference simulation of sonic logging. Journal of Acoustical Society of America 100(1), 72–79 (1996)
12. Stephen, R.A., Cardo-Casas, F., Cheng, C.H.: Finite-difference synthetic acoustic logs. Geophysics 50(10), 1588–1609 (1985)
13. Virieux, J.: P-SV propagation in heterogeneous media:velocity-stress finite-difference method. Geophysics 51, 889–901 (1986)

An Improved Interval Global Optimization Method and Its Application to Price Management Problem

Bartlomiej Jacek Kubica[1] and Ewa Niewiadomska-Szynkiewicz[2]

[1] Institute of Control and Computation Engineering,
Warsaw University of Technology, Nowowiejska 15/19, PL-00-665 Warsaw, Poland
bkubica@elka.pw.edu.pl
http://www.ia.pw.edu.pl
[2] Research and Academic Computer Network (NASK), Wawozowa 18,
PL-02-796 Warsaw, Poland
ewan@nask.pl
http://www.nask.pl

Abstract. We present an interval global optimization algorithm using a modified monotonicity test. The improvement applies to constrained problems and can result in significant speedup, when constraints are sparse, i.e. they “bind” a few of the variables, not all of them. A theorem that ensures the correctness of the new tool, is given and proved. The improved method is applied to an economic problem of setting optimal prices on a couple of products.

1 Introduction

Many issues related to the practical problems require the solution of the following very general optimization problem:

$$\min_{x \in X} f(x) \ , \tag{1}$$

where the function $f(x)$ can be nonconvex and X is defined by the set of (possibly nonconvex) constraints: $g_j(x) \leq 0$, $j = 1, \ldots, m$ and bounds on each component of x. The problem of designing algorithms to compute global solutions is very difficult. In general there are no local criteria in deciding whether the local solution is the global one. Many of methods apply heuristics able to find an approximate solution only, see [2], [8].

Interval methods aim to find the global solution(s). Unfortunately, they are usually slow and memory demanding so, the acceleration is worthwhile. The general framework of the optimization algorithm is the branch–and–bound schema. Many different tools are used to reject or at least narrow subboxes that do not contain the global optimum, e.g. [3], [4], [9], [2]. The objective of this paper is to identify some of the advantages and disadvantages of existing algorithms and provide suitable modifications to increase their efficiency. We propose a modified monotonicity test for constrained optimization problems. It allows to remove from further considerations far more boxes than its traditional relative. The presented algorithm was applied to a practical marketing problem.

2 Interval Branch–and–Bound Method

The general scheme of interval global optimization methods is branch–and–bound (see e.g. [1], [3], [4]). It can be expressed by the following pseudocode:

```
IBB ($\boldsymbol{x}^{(0)}$; f, $\nabla$f, $\nabla^2$f; g$_1$, $\nabla$g$_1$, $\nabla^2$g$_1, \ldots,$ g$_m$, $\nabla$g$_m$, $\nabla^2$g$_m, \ldots$)
// $\boldsymbol{x}^{(0)}$ is the initial box
// f$(\cdot)$ is the interval extension of the objective function $f(\cdot)$
// $\nabla$f$(\cdot)$ and $\nabla^2$f$(\cdot)$ are interval extensions of gradient and Hessian of $f(\cdot)$
// g$_i(\cdot)$ are interval extensions of the constraints, etc.
// $L_{sol}$ is the list of solutions
$[\underline{y}^{(0)}, \overline{y}^{(0)}] =$ f$(\boldsymbol{x}^{(0)})$ ;
compute $f_{min} =$ the upper bound on the global minimum (e.g. objective value
in a feasible point)
$L = \{(\boldsymbol{x}^{(0)}, \underline{y}^{(0)})\}$ ; // the list of boxes
$L_{sol} = \emptyset$ ;
while ($L \neq \emptyset$) do
    $\boldsymbol{x} =$ the element of $L$ with the lowest function value underestimation ;
    compute the values of interval extensions of the constraint functions ;
    if ($\boldsymbol{x}$ is infeasible) then discard $\boldsymbol{x}$ ;
    update $f_{min}$ if possible ;
    perform other rejection/reduction tests on $\boldsymbol{x}$ ;
    if ($\boldsymbol{x}$ is verified to contain a unique critical point or
        $\boldsymbol{x}$ is small and not infeasible) then
        add $\boldsymbol{x}$ to $L_{sol}$ ;
    else
        bisect $\boldsymbol{x}$ to subboxes $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ ;
        compute lower bounds $\underline{y}^{(1)}$ and $\underline{y}^{(2)}$ on the function value in
        the obtained boxes ;
        delete $\boldsymbol{x}$ from $L$ ;
        for $i = 1, 2$ do
            put $(\boldsymbol{x}^{(i)}, \underline{y}^{(i)})$ on the list $L$ preserving the increasing order of
            the lower bounds ;
        end for
        delete from $L$ boxes with $\underline{y}^{(i)} > f_{min}$ ;
    end if
end while
delete from $L_{sol}$ the boxes with $\underline{y}^{(i)} > f_{min}$ ;
return $L_{sol}$ ;
end IBB
```

The above pseudocode mentions some “rejection/reduction tests” that may be used in conjunction with the IBB algorithm. There are several such tests. Most important of them are: several kinds of Newton operators, constraint propagation steps and, probably oldest of them all, monotonicity tests. We do not describe them all, as they are widely available in literature, e.g. [1], [4], etc.

Most of the current research concentrate on several types of interval Newton operators and tools related to them e.g. [6]. It is reasonable – interval Newton methods are one of the most powerful interval techniques, indeed. They can not only reject or narrow subboxes during the branch–and–bound process, but they can also verify that a subbox contains a root (or a feasible point) certainly; sometimes, they can even verify the uniqueness of this root.

Nevertheless, in this paper, we shall concentrate on monotonicity tests. This tool is less investigated recently, but – as we shall see – it can also be improved, resulting in a significant speedup of the overall algorithm.

3 The Classical Monotonicity Test

Monotonicity test ([2]) allows to discover boxes where the objective function is strictly monotone – such boxes cannot contain a global minimum. In the case of a general constrained optimization problem, monotonicity test can be applied only to boxes where all constraints are satisfied, otherwise it does not guarantee the non-existence of an optimum. By $\boldsymbol{x} \subseteq X$ we denote the box on X and the inclusion functions of f, ∇f and g, respectively – by $\mathsf{f}(\boldsymbol{x})$, $\nabla\mathsf{f}(\boldsymbol{x})$ and $\mathsf{g}(\boldsymbol{x})$.

The classical monotonicity test (CMT) can be formulated by the following pseudocode:

```
for $i = 1, \ldots, n$ do
    compute $[\underline{y}, \overline{y}] = \nabla\mathsf{f}_i(\boldsymbol{x})$ ;
    if $(\overline{y} < 0)$ then
        // the function is decreasing
        compute $[\underline{g}_j, \overline{g}_j] = \mathsf{g}_j(\boldsymbol{x})$ for all $j = 1, \ldots, m$ ;
        if $(\overline{g}_j < 0 \quad \forall j = 1, \ldots, m)$ then discard $\boldsymbol{x}$ ;
        else if $(\overline{g}_j \le 0 \quad \forall j = 1, \ldots, m)$ then update $\underline{x}_i$ to $\underline{x}_i = \overline{x}_i$ ;
        end if
    end if
    if $(\underline{y} > 0)$ then
        // the function is increasing
        ...
    end if
end for
```

The simplest way to remove infeasible solutions is the flag system, proposed by Ratschek and Rokne [2]. The idea is as follows. We associate with each box $\boldsymbol{x}$ the binary vector $r = (r_1, \ldots, r_m)$, where m is the number of constraints. So, $r_j = 1$ indicates that the j-th constraint is satisfied in the box $\boldsymbol{x}$, otherwise $r_j = 0$. It is obvious that if a considered constraint is satisfied in a box $\boldsymbol{x}$, it would also be satisfied in all subboxes, resulting from its bisections. The algorithm, determining the flags for a new box $\boldsymbol{x}^k$, obtained from the bisection of $\boldsymbol{x}$, may be expressed as follows:

```
for $j = 1, \ldots, m$ do
    if $(r_j == 1)$ then $r_j^k = 1$ ;
    else
        compute $[\underline{g}, \overline{g}] = \mathsf{g}_j(\boldsymbol{x}^k)$ ;
        if $(\overline{g} \le 0)$ then $r_j^k = 1$ ;
        else if $(\underline{g} > 0)$ then discard $\boldsymbol{x}^k$ ;
        else $r_j^k = 0$ ;
    end if
end for
```

In the case of problems in which the constraints bind only a group of variables the efficiency of the monotonicity test can be increased. Next section proposes a suitable modification.

4 Main Results

The proposed modified version of the monotonicity test allows to discard a box $\boldsymbol{x}$ in the case when the constraint $g(x) \le 0$, $x \in \boldsymbol{x}$ is not satisfied but does not "bind" the variable, with respect to which the objective is monotonous. We say that $g_j(x_1, \ldots, x_n)$ does not "bind" the variable x_i when the variable x_i is *not present* in $g_j(x)$ formulation. The modified algorithm (MMT) is as follows:

```
for $i = 1, \ldots, n$ do
    compute $[\underline{y}, \overline{y}] = \nabla \mathsf{f}_i(\boldsymbol{x})$ ;
    if $(\overline{y} < 0)$ then
        // the function is decreasing
        compute $[\underline{g}_j, \overline{g}_j] = \mathsf{g}_j(\boldsymbol{x})$ for all $j$ such that $\frac{\partial \mathsf{g}_j(\boldsymbol{x})}{\partial x_i} \neq [0, 0]$ and $j = 1, \ldots, m$ ;
        if (all computed $\overline{g}_j$'s satisfy $\overline{g}_j < 0$) then discard $\boldsymbol{x}$ ;
        else if (all computed $\overline{g}_j$'s satisfy $\overline{g}_j \le 0$) then update $\underline{x}_i$ to $\underline{x}_i = \overline{x}_i$ ;
        end if
    end if
    if $(\underline{y} > 0)$ then
        // the function is increasing
        ...
    end if
end for
```

It can be proved that the modified test preserves all the solutions (see Theorem 1, below). A minor improvement, associated with the modified monotonicity test is a new variant of the flag vector. We can distinguish four possibilities: a) the j-th constraint is satisfied and inactive in the box $\boldsymbol{x}$ $(\overline{g}_j < 0)$, b) the constraint is satisfied, but may be active $(\overline{g}_j \le 0)$, c) the constraint may be violated for some points of the box $(0 \in [\underline{g}_j, \overline{g}_j])$, d) the constraint is violated in the whole box $(\underline{g}_j > 0)$ and the box should be discarded from further processing. Well then, to be consistent with the monotonicity test it is better to use "flags" of three possible values, instead of binary ones:

```
for $j = 1, \ldots, m$ do
    if $(r_j == 2)$ then $r_j^k = 2$ ;
    else
        compute $[\underline{g}, \overline{g}] = \mathsf{g}_j(\boldsymbol{x}^k)$ ;
        if ($\overline{g} < 0$) then $r_j^k = 2$ ;
        else if ($\overline{g} \leq 0$) then $r_j^k = 1$ ;
        else if ($\underline{g} > 0$) then discard $\boldsymbol{x}^k$ ;
        else $r_j^k = 0$ ;
    end if
end for
```

And now let us present the main theorem:

Theorem 1. *Consider the optimization problem (1). Consider a box $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)^T$, contained in the interior of the feasible set X. Assume for some $i \in \{1, \ldots, n\}$ the following two conditions are both fulfilled:*

- *$\forall x \in \boldsymbol{x}$ either $\frac{\partial f(x)}{\partial x_i} < 0$ or $\frac{\partial f(x)}{\partial x_i} > 0$,*
- *$\forall j = 1, \ldots, m$ either we have $\forall x \in \boldsymbol{x} \quad g_j(x) < 0$ or $\frac{\partial g_j(x)}{\partial x_i} \equiv 0$.*

Then there is no optimum in the box $\boldsymbol{x}$.

Proof
Assume a point $x_0 \in \boldsymbol{x}$ is the (at least local) optimum of the problem (1). The suppositions of Theorem 1 say that $\forall x \in \boldsymbol{x}$ either $\frac{\partial f(x)}{\partial x_i} < 0$ or $\frac{\partial f(x)}{\partial x_i} > 0$, i.e. the objective $f(\cdot)$ is monotone on the box $\boldsymbol{x}$.

If $f(\cdot)$ is monotone at x_0 that the only case when x_0 can be the optimum of the problem (1) is when some constraints are active at this point and all improvement directions are infeasible. Nevertheless, we can simply show that at least one improvement direction is feasible at x_0. Let us consider the direction pointed by the vector $d = (d_1, \ldots, d_n)^T$, where:

$$d_k = \begin{cases} 0, & \text{for } k \neq i \\ 1, & \text{for } k = i \end{cases} \qquad \text{if } f(\cdot) \text{ is decreasing, or}$$

$$d_k = \begin{cases} 0, & \text{for } k \neq i \\ -1, & \text{for } k = i \end{cases} \qquad \text{if } f(\cdot) \text{ is increasing.}$$

From now on let us assume that $f(\cdot)$ is decreasing, i.e. $\frac{\partial f(x)}{\partial x_i} < 0$. The proof for the second case would be analogous. Obviously, d is an improvement direction, because:

$$\nabla f(x_0)^T \cdot d = \frac{\partial f(x)}{\partial x_i} < 0 \ . \tag{2}$$

Let us show it is feasible. It is enough to prove that:

$$\exists \sigma > 0 \qquad \forall \alpha \quad 0 < \alpha < \sigma \rightarrow (x_0 + \alpha d) \text{ is feasible.} \tag{3}$$

Indeed, consider a constraint $g_j(x) < 0$, $j \in \{1, \ldots, m\}$. From the Taylor expansion we get:

$$g_j(x_0 + \alpha d) = g_j(x_0) + \nabla g_j(\xi) \cdot (\alpha d) = g_j(x_0) = \alpha \frac{\partial g_j(\xi)}{\partial x_i} \ , \tag{4}$$

where $\xi \in (x_0, x_0 + \alpha \cdot d) \subseteq \boldsymbol{x}$. But this implies that $\frac{\partial g_j(\xi)}{\partial x_i} = 0$ and (4) reduces to:

$$g_j(x_0 + \alpha \cdot d) = g_j(x_0) \ . \tag{5}$$

But x_0 is feasible by supposition, so $g_j(x_0 + \alpha \cdot d) = g_j(x_0) < 0$. Consequently, the direction d is feasible. Thus, there is a feasible improvement direction in any $x_0 \in \boldsymbol{x}$, so no optimum can be in $\boldsymbol{x}$.

Interpretation. The essence of the assumptions in Theorem 1 is quite simple, actually. The first supposition says that the objective is monotone with respect to the variable x_i. The second one says that all of the constraints are either inactive or do not contain the variable x_i in their formulae. This simply means that all of the active constraints are invariant with respect to x_i and the feasible set has the form of some kind of a pipe along x_i. An example of such a set is shown on Figure 1. It should be obvious that in such case the constraints cannot block the direction along x_i, which is an improvement direction.

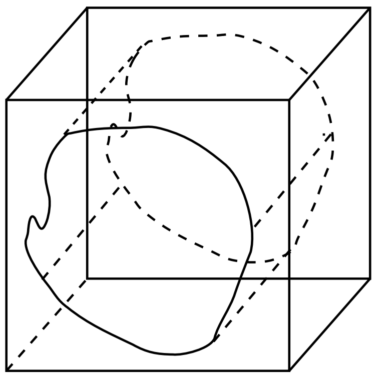


Fig. 1. When can the modified monotonicity test be applied ?

5 How to Treat Bound Constraints ? Peeling Process

To investigate the usefulness of the developed rejection/reduction test it is reasonable to check how it cooperates with other tools, like Newton methods, etc. We cannot present all of the results here. Investigated variants of the IBB algorithm use the interval Newton step (precisely: interval Gauss-Seidel step).

Whereas we examine how does the modified monotonicity test cooperate with the "peeling" process (see e.g. [3], [4]), so we apply four variants of the algorithm: with classical or modified monotonicity test and with or without peeling.

What is the peeling process ? It is one of the ways to deal with bound constraints. Actually there are two main approaches to solve bound–constrained problems:

– treat bound constraints as any other inequality constraints (or, roughly speaking, similarly to them),
– consider the “boundary boxes” and “interior boxes” separately.

In this second variant, we use reduced gradients and Hessians in boundary boxes (because they have a reduced dimension) and in interior boxes bounds constraints are ignored (in monotonicity tests or Newton steps; only other constraints are used then). The boundary boxes are created at the beginning of the IBB algorithm, by the peeling process.

Let us formulate the peeling procedure in a pseudocode. We present it in a slightly simpler form than the one in [3] or [4]:

```
peeling ($\boldsymbol{x}$, $i$)
if ($i == n$) then
    let $\boldsymbol{y} = \boldsymbol{x}$ ;
    let $y_i = [\underline{x}_i, \underline{x}_i]$ ; enqueue ($\boldsymbol{y}$) ;
    let $y_i = [\overline{x}_i, \overline{x}_i]$ ; enqueue ($\boldsymbol{y}$) ;
    let $y_i = x_i$ ; enqueue ($\boldsymbol{y}$) ;
else
    let $\boldsymbol{y} = \boldsymbol{x}$ ;
    let $y_i = [\underline{x}_i, \underline{x}_i]$ ; peeling ($\boldsymbol{y}$, $i + 1$) ;
    let $y_i = [\overline{x}_i, \overline{x}_i]$ ; peeling ($\boldsymbol{y}$, $i + 1$) ;
    let $y_i = x_i$ ; peeling ($\boldsymbol{y}$, $i + 1$) ;
end if
end peeling
```

According to [3], complexity of the peeling process is of the degree 3^k, where k is the number of bound constraints. This leads to inefficiency of this process if the number of bound constraints is large. All the same, not all of the bounds may actually be “bound constraints” – some of them may be known to be redundant or non–significant. Unfortunately, that is not the case for the problem we are trying to solve.

6 Numerical Experiments

Numerical experiments were performed for both testing problems and real-life applications. Results obtained for the selected test problem defined as:

$$\min_x \sum_{i=1}^{n} (-1)^i \cdot x_i^2 \ , \tag{6}$$

s.t. $\sum_{i=1}^n x_i \leq 1$, $x_{i-1} + x_i \leq 0.5$, $-1 \leq x_i \leq 2$, $i = 1, \ldots, n$.
are given in Fig. 2.

One of the considered case study was concerned with computing of the optimal prices for products that are sold in a shop. The goal was to maximize the total profit defined as: $PR = \sum_{i=1}^n \left(\frac{x_i}{1+v_i} - d_i \right) \cdot S_i$, where n denotes the number of

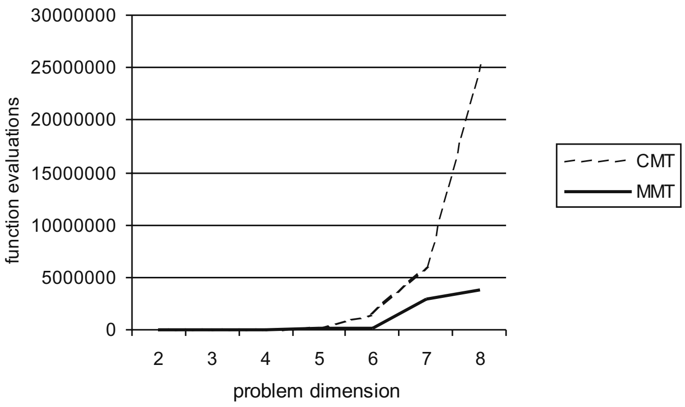


Fig. 2. Number of function evaluations; CMT – classical and MMT – modified monotonicity tests

products, v_i and d_i given constants, corresponding to the market entities of VAT and cost per product i, S_i are expected sales of products within the considered period and x_i prices, we are trying to set. Two models describing the market response S_i on the price were considered:

Cobb-Douglas model described in [10]. This model implements the cross-effects with other substitute or complementary own products: $S_i = \alpha_i \cdot \prod_{j=1}^{n} x_j^{\beta^{ij}}$, where β^{ij} is the elasticity of sales of the i–th product with respect to the price of the j–th product, β^{ii} is referred to as the direct elasticity and β^{ij}, where $i \neq j$ – the cross elasticity. This function is widely used, but it does not capture some important effects, such as different market sensitivities to small and large price changes. These features are expressed by the so-called *s-shape* models.

Hybrid model formulated in [7] that exhibits an *s-shape* and includes cross-effects: $S_i = a_i + \alpha_i \cdot \prod_{j=1}^{n} x_j^{\beta^{ij}} + c_{1i} \cdot \sinh\left(c_{2i} \cdot (x_i - x_i^{co})\right)$, where x_i^{co} is the average competitive price and a, c_1 and c_2 are model parameters. The model combines descriptions proposed by Cobb-Douglas and Gutenberg (see [10]). The following constraints for price, sale and cash of each product were considered: bounds on prices and sales, bounds on the cash flows; constraints for total sale and cash: $ST^{\min} \leq \sum_{i=1}^{n} S_i \leq ST^{\max}$, $CT^{\min} \leq \sum_{i=1}^{n} x_i \cdot S_i \leq CT^{\max}$, linear constraints for price differences of substitute or complementary products $d_{ij}^{\min} \leq x_i - x_j \leq d_{ij}^{\max}, \quad (i,j) \in D$, where D is the set of pairs of correlated products.

The overall optimization problem was as follows:

$$\max_x \left(PR = \sum_{i=1}^{n} \left(\frac{x_i}{1+v_i} - d_i \right) \cdot S_i(x) \right) , \tag{7}$$

s.t. $x_i^{\min} \leq x_i \leq x_i^{\max}$, $S_i^{\min} \leq S_i(x) \leq S_i^{\max}$, $C_i^{\min} \leq x_i \cdot S_i(x) \leq C_i^{\max}$, for $i = 1, \ldots, n$ and $ST^{\min} \leq \sum_{i=1}^{n} S_i(x) \leq ST^{\max}$, $CT^{\min} \leq \sum_{i=1}^{n} x_i \cdot S_i(x) \leq CT^{\max}$, $d_{ij}^{\min} \leq x_i - x_j \leq d_{ij}^{\max}, \quad (i,j) \in D$.

Tables 1 and 2 present the results of application of the flag-based IBB algorithm with classical (CMT) and modified (MMT) monotonicity tests to price management problem formulated for 8 products. The first one applies to the case of Cobb-Douglas model and the second one – to Hybrid model. The accuracy for all eight cases was set to: $\varepsilon = 0.05$ and there were two pairs of correlated products in the set D.

Meaning of the columns is as follows: the first one describes type of the monotonicity test, the other ones – the number of bisections, objective evaluations, objective's gradient and Hessian evaluations, constraints evaluations, respectively, and the last one – the number of boxes, enclosing the global optimum, resulting form the IBB algorithm.

Table 1. Price management problem – results for Cobb-Douglas model

IBB without peeling							
mon. test	exec.time	bisec.	f.evals.	grad.evals.	Hess.evals.	constr.evals.	res.boxes
CMT	25917.8 s	262430	776208	516851	514803	777351	252373
MMT	28.92 s	940	2285	1848	1377	2435	437
IBB with peeling							
mon. test	exec.time	bisec.	f.evals.	grad.evals.	Hess.evals.	constr.evals.	res.boxes
CMT	41561.7 s	317514	856786	574011	564052	1684852	223292
MMT	273.48 s	2603	9587	8356	3217	271166	263

Table 2. Price management problem – results for Hybrid model

IBB without peeling							
mon. test	exec.time	bisec.	f.evals.	grad.evals.	Hess.evals.	constr.evals.	res.boxes
CMT	1772.78 s	33810	100203	66848	66578	100554	32768
MMT	11.75 s	435	850	836	447	1040	12
IBB with peeling							
mon. test	exec.time	bisec.	f.evals.	grad.evals.	Hess.evals.	constr.evals.	res.boxes
CMT	8289.94 s	109380	249401	184489	175911	699398	30205
MMT	395.64 s	4023	11890	8879	4757	286847	7

Results. It can be observed that the modified monotonicity test results – at least for investigated problems – in dramatical speedup. The improvement is significant in both cases – when we use peeling process and when we do not.

Actually, for the price management problem, the variant of the IBB algorithm using peeling was much slower. This was caused by the fact that we had significant bound constraints on all of the eight decision variables. Peeling is usually inefficient in such cases. Nevertheless, the monotonicity test gave a large speedup also for this variant of the algorithm.

Obviously, the improvement was that large, because the examined problem had several constraints biding only few variables. Otherwise the speedup would not be achieved.

7 Conclusions

As a final conclusion we can say that the proposed modifications to the flag-based IBB algorithm and monotonicity test increase the speed of convergence with respect to classical ones. It seems to be a powerful accelerating device, which gives approximately exponential improvement for constraints binding few variables.

References

1. Hansen, E.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (1992)
2. Horst, R., Pardalos, P.M. (eds.): Handbook of Global Optimization. Kluwer, Dordrecht (1995)
3. Kearfott, R.B.: A Review of Techniques in the Verified Solution of Constrained Global Optimization Problems. In: Kearfott, R.B., Kreinovich, V. (eds.) Applications of Interval Computations, Kluwer, Dordrecht (1996)
4. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
5. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis, available at `http://www.mat.univie.ac.at/~neum/software/int/notation.ps.gz`
6. Kubica, B.J., Malinowski, K.: An Interval Global Optimization Algorithm Combining Symbolic Rewriting and Componentwise Newton Method Applied to Control a Class of Queueing Systems. Reliable Computing 11(5), 393–411 (2005)
7. Malinowski, K.: PriceStrat 4.0 Initial Research Paper. KSS Internal Document, Manchester (2000)
8. Michalewicz, Z., Fogel, D.B.: How to Solve It: Modern Heuristics. Springer, Heidelberg (2004)
9. Neumaier, A.: Complete Search in Continuous Global Optimization and Constraint Satisfaction. In: Acta Numerica, pp. 271–369. Cambridge University Press, Cambridge (2004)
10. Simon, H.: Price Management. North-Holland (1989)

Optimizing Neural Network Classifiers with ROOT on a Rocks Linux Cluster

Tomas Lindén, Francisco García, Aatos Heikkinen, and Sami Lehti

Helsinki Institute of Physics, University of Helsinki, POB 64, FIN-00014, Finland
`tlinden@cc.helsinki.fi`
`http://www.physics.helsinki.fi/~tlinden/`

Abstract. We present a study to optimize multi-layer perceptron (MLP) classification power with a Rocks Linux cluster [1]. Simulated data from a future high energy physics experiment at the Large Hadron Collider (LHC) is used to teach a neural network to separate the Higgs particle signal from a dominant background [2].

The MLP classifiers have been implemented using the ROOT data analysis framework [3]. Our aim is to reach a stable physics signal recognition for new physics and a well understood background rejection. We report on the physics performance of new neural classifiers developed in this study. We have used the benchmarking capabilities of ROOT and of the Parallel ROOT facility (PROOF) [4] to compare the performance of the Linux clusters at our campus.

1 Introduction

B-tagging is an important tool for separating the Higgs events with associated b-jets from the Drell-Yan background $Z, \gamma^* \rightarrow \tau\tau$, for which the associated jets are mostly light quark and gluon jets. The most simple algorithm used for b-tagging is a track counting algorithm. The track counting algorithm counts tracks with high enough track impact parameter significance (value divided by the estimated error), and if enough such tracks, usually 2 or 3, are found, the jet is b-tagged. The track impact parameter has an upper limit to suppress the background from long lived K and Λ^0 with impact parameters significantly larger than those from the b-decays. The track counting algorithm has given a 35% b-tagging efficiency for b-jets associated with the Higgs boson $\mathrm{gg} \rightarrow \mathrm{b\bar{b}H}$ with about 1% mistagging rate for light quark and gluon jets in Drell-Yan events [5]. Probabilistic algorithms[6] using track impact parameter and secondary vertex information give slightly better b-tagging efficiencies [7].

A neural network (NN) approach has been shown to be applicable to the problem of Higgs boson detection at the Large Hadron Collider. We study the use of NNs in the problem of tagging b-jets in $\mathrm{pp} \rightarrow$, $\mathrm{b\bar{b}H_{SUSY}}$, $\mathrm{H_{SUSY}} \rightarrow \tau\tau$ in the Compact Muon Solenoid experiment. For teaching the neural network we have selected b-jets from $\mathrm{t\bar{t}}$ events and light quark and gluon jets from the $\mathrm{Z}, \gamma^* \rightarrow \mathrm{ee}$ and W+jet events, all backgrounds to $\mathrm{H_{SUSY}} \rightarrow \tau\tau \rightarrow \mathrm{e} + \mathrm{jet} + \mathrm{X}$ [8]. The b-jets associated with the Higgs events are used for testing and verifying the results.

The events are simulated using PYTHIA [9] with CTEQ5 structure functions [10]. The CMS detector response is simulated using full GEANT simulation within the ORCA framework [11]. A low luminosity of $2\times10^{33}\text{cm}^{-2}\text{s}^{-1}$ with on average 3.4 minimum bias events superimposed per crossing was assumed. The simulated events are required to pass the trigger (single e or e+tau trigger)[12] and to have a successfully reconstructed primary vertex. The jet reconstruction is done using an Iterative Cone Algorithm [13], and tracks within a cone of $\Delta R = \sqrt{(\Delta\phi)^2 + (\Delta\eta)^2} = 0.7$ around the jet axis are reconstructed using a Combinatorial Trajectory Builder [14]. The generator level Monte-Carlo truth is used to identify the genuine b,c and light quark and gluon jets.

The simulated samples consist of 46000 b-jets associated with the Higgs boson, 401 000 b-jets from the $t\bar{t}$ events, 313 000 light quark and gluon jets from the Drell-Yan events and 297 000 light quark and gluon jets from the W+jet background.

2 Computational Environment

NPACI Rocks Cluster Distribution is a cluster management software for scientific computation based on Red Hat Linux, supporting cluster installation, configuration, monitoring and maintenance [15]. Several Rocks based clusters have made it to the Top500 list [16], for a current list of the installed Rocks clusters, see the Rocks website [17]. Rocks comes preconfigured with the Sun Grid Engine (SGE) batch queue system, which supports advanced features like back filling, fair share usage and array jobs.

There are two Rocks production clusters available at our institute. The larger one is a 64-bit 1.8/2.2 GHz AMD Opteron cluster called *ametisti*, which has 132 CPUs in 66 computational nodes with 2/4 GB RAM. Ametisti is shown schematically in Fig. 1. Initially it ran Rocks version 3.2, but it was upgraded to Rocks 4.1. On ametisti there is one dedicated Gb/s network for communication

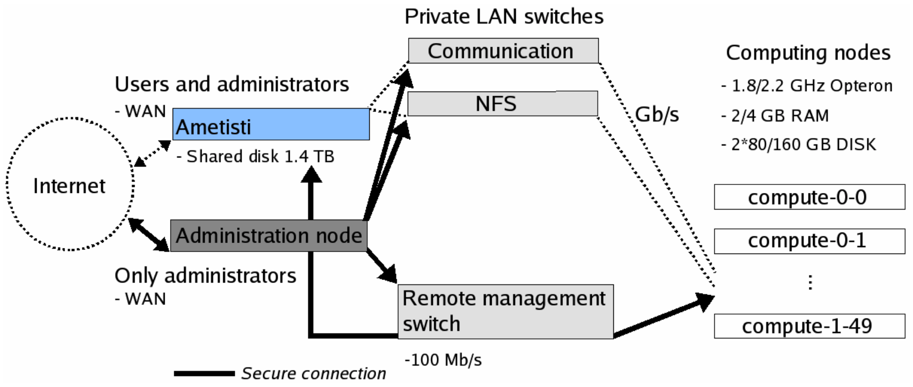


Fig. 1. A schematic picture of the 66+2 node dual AMD Opteron Rocks 1U rack cluster *ametisti*

and another dedicated Gb/s network for NFS-traffic to enhance the performance of the shared NFS disk system. In addition to this there is also a fast Ethernet network used for remote management.

The smaller one is a 32-bit 2.13 GHz AMD Athlon cluster called *mill*, running Rocks version 3.3. It has 64 CPUs in 32 computational nodes with 1 GB RAM connected to a fast Ethernet network to the Gb/s network interface of the cluster frontend.

3 Neural Networks in the ROOT Data Analysis Framework

In typical High Energy Physics (HEP) experiments, various background events hide weak signal events coming from the physics processes of interest. Efficient analysis of large data sets produced by HEP experiments requires, indexing, preselection of data using meta data tags and processing very large data samples of $\mathcal{O}$(TB)–$\mathcal{O}$(PB). The traditional method of separating signal and background is to make cuts in a multidimensional parameter space. A more recent data analysis method is to teach neural networks (NN) to distinguish between signal and background events and then use the taught NNs as classifiers.

The problem of teaching a neural network (NN) that performs well is quite a subtle issue, especially when only a limited number of training samples are available. Thus specific techniques are needed to avoid over fitting.

For this study we have selected the data analysis tool ROOT [3] developed at CERN and widely used in HEP. The benefits of this framework include: optimized treatment of large and complex data sets, support for C++ scripting and compiled code and PROOF - parallelized ROOT version supporting also Grid usage [4]. ROOT provides a flexible object oriented implementation of multi-layer perceptrons (MLPs). This is the most commonly used and accepted neural network type in HEP Higgs data analysis, because there is no significantly superior neural network. ROOT MLP provides various learning methods such as Steepest descent algorithm, Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS) and variants of conjugate gradients. Visualization of the network architecture and learning process is supported. The networks can be exported as a standalone C++ code. Input and output data is organized using the TTree class, which is the key ROOT data structure. The TChain class, which is a collection of files containing TTree objects, can be used for performing parallel processing of MLP related data. In our neural network approach to the b-tagging problem, we feed the networks with information on the number of tracks in the jet cone, leading track impact parameters (for a detailed description see [18]) and impact parameter significances. A SUSY event in the CMS detector and a closeup of secondary vertexes from another event are shown in Fig. 2. Finding an optimal set of variables for teaching an optimal MLP configuration is a multidimensional, computationally demanding optimization task suitable for solving with a Linux cluster.

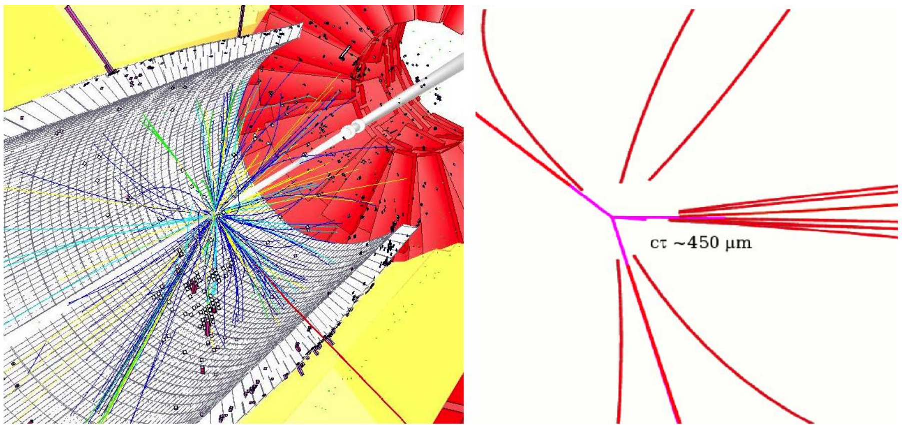


Fig. 2. *Left:* Geant4 based simulation of a SUSY event in the CMS detector with missing transverse energy, jets and several leptons in the barrel detector. (Picture: IguanaCMS.). *Right:* Successfully reconstructed jets can be identified as b-jets using a lifetime based tagging algorithm, which relies on displaced secondary vertices's.

Benchmarking tools are included in the ROOT Stress test suite. A standard mixture of CPU and I/O tasks is performed by running **stress -b**. The performance number ROOTMARK has been normalized to 800 on a 2,8 GHz Pentium 4 CPU running Linux and gcc version 3.2.3. On the 32-bit *mill* cluster we get 609 ROOTMARKS on average from three runs of **stress -b**. The 64-bit *ametisti* cluster achieves 988 ROOTMARKS on the 1,8 GHz nodes.

4 PROOF

PROOF [4] is an extension to ROOT which allows the parallel analysis of large ROOT trees. The large data sets produced by present and future high energy physics and other experiments makes it a very interesting and useful data analysis tool. PROOF can be used on a set of inhomogeneous workstations or in a homogeneous cluster environment. It uses a master slave architecture with possible layers of sub masters and load balancing through pull mode task distribution. It has been designed to be used on systems ranging from single or multi core workstations to large clusters with $\mathcal{O}(100)$ CPUs or even collections of clusters connected with grid middleware. The best performance is obtained when the data set to be analyzed is distributed over the local disks of the cluster nodes. Good scaling has been reported for over 70 CPUs [19].

PROOF is distributed with all standard ROOT releases. The configuration instructions are found in the file README.PROOF in the local ROOT installation area and on the TWiki WebHome-page [20]. To PROOF enable a cluster two daemons need to be started and configured with *xinetd*, for version 5.10 of

ROOT used in this work. On more recent ROOT releases PROOF can be setup so that only one daemon, xrootd [21], is needed on the computational nodes.

ProofBench is a benchmarking tool distributed together with PROOF for performance studies [22] in the *test* directory. ProofBench can also be used as a PROOF example to test the setup. The more recent 64-bit 1,8 GHz ametisti Opteron cluster nodes processes eight million events with four nodes (eight slaves) with one file per slave (14 GB of data in total) 1,8 times faster than the 32-bit 2,133 MHz *mill* Athlon nodes according to ProofBench. We have used mainly *ametisti* for the computations in this work, since it is faster than *mill*. The integration of PROOF with SGE to simplify our calculations remains the topic of further work.

5 MLP Results

We have used extensively a TMLPAnalyzer utility class, which contains a set of useful tests developing optimal neural network layouts. This class allows the user to check for unneeded variables, and to control the network structure.

With the DrawDInputs method we can study what kind of impact a small variation of input variables has on the network output. This kind of sensitivity analysis has been useful when selecting an optimal set of NN input parameters, as shown in Fig. 4. Already in our previous simulation impact parameter significances were found to have more classification power [2] when compared to impact parameters themselves. The ROOT MLP provides also a DrawTruthDeviation method, which is mainly useful for doing regression analysis. For training

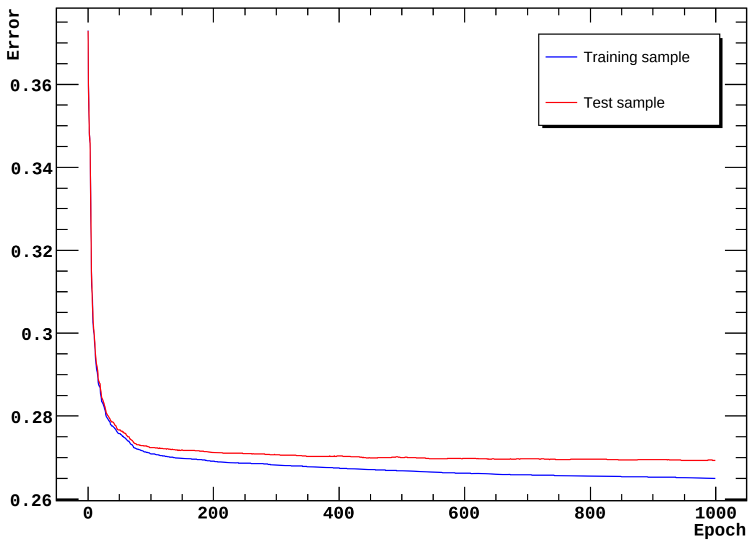
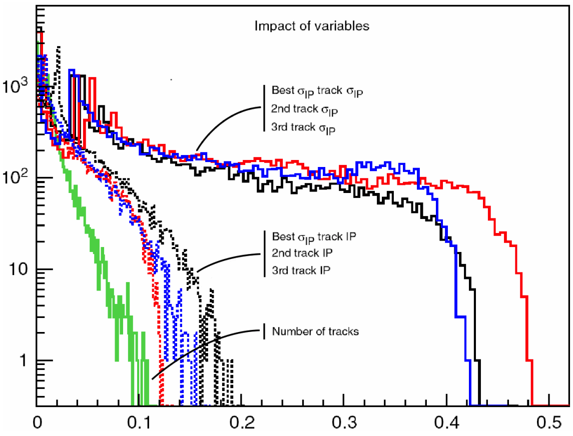
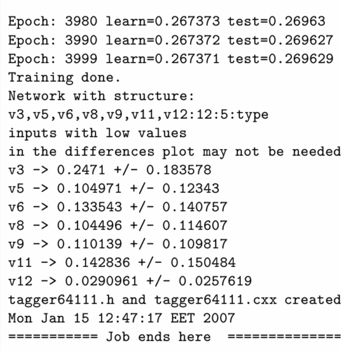


Fig. 3. Typical behaviour of training and validation error for 1000 epochs





```
Epoch: 3980 learn=0.267373 test=0.26963
Epoch: 3990 learn=0.267372 test=0.269627
Epoch: 3999 learn=0.267371 test=0.269629
Training done.
Network with structure:
v3,v5,v6,v8,v9,v11,v12:12:5:type
inputs with low values
in the differences plot may not be needed
v3 -> 0.2471 +/- 0.183578
v5 -> 0.104971 +/- 0.12343
v6 -> 0.133543 +/- 0.140757
v8 -> 0.104496 +/- 0.114607
v9 -> 0.110139 +/- 0.109817
v11 -> 0.142836 +/- 0.150484
v12 -> 0.0290961 +/- 0.0257619
tagger64111.h and tagger64111.cxx created
Mon Jan 15 12:47:17 EET 2007
=========== Job ends here  ==============
```

Fig. 4. *Left:* Visualization of ROOT MLP tool [3] shows that the track impact parameter significances have the best classification power. This technique was used to select suitable input variables. (More detailed description of variable selection can be found from [2]). *Right:* Exerpt from job 64111 log showing epoch data and significance of various input parameters for MLP 7-12-5-1 configuration.

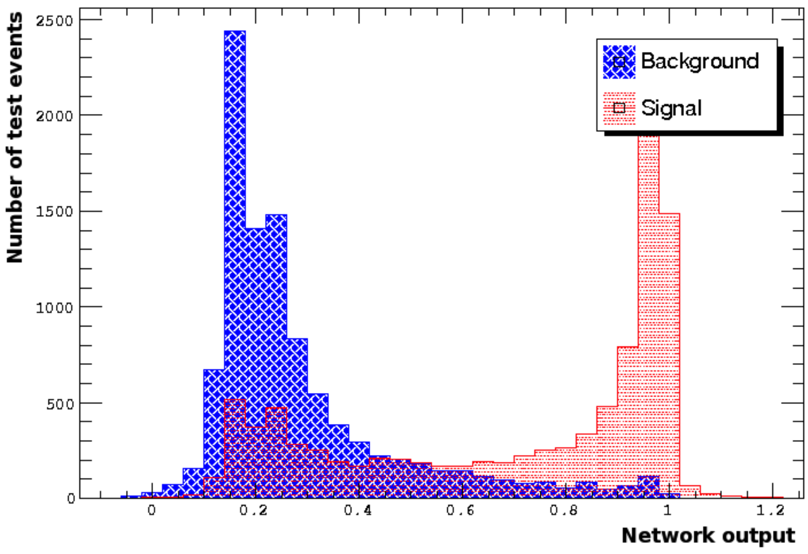


Fig. 5. A typical example of the classification power achieved with an optimized 10-15-5-1 network

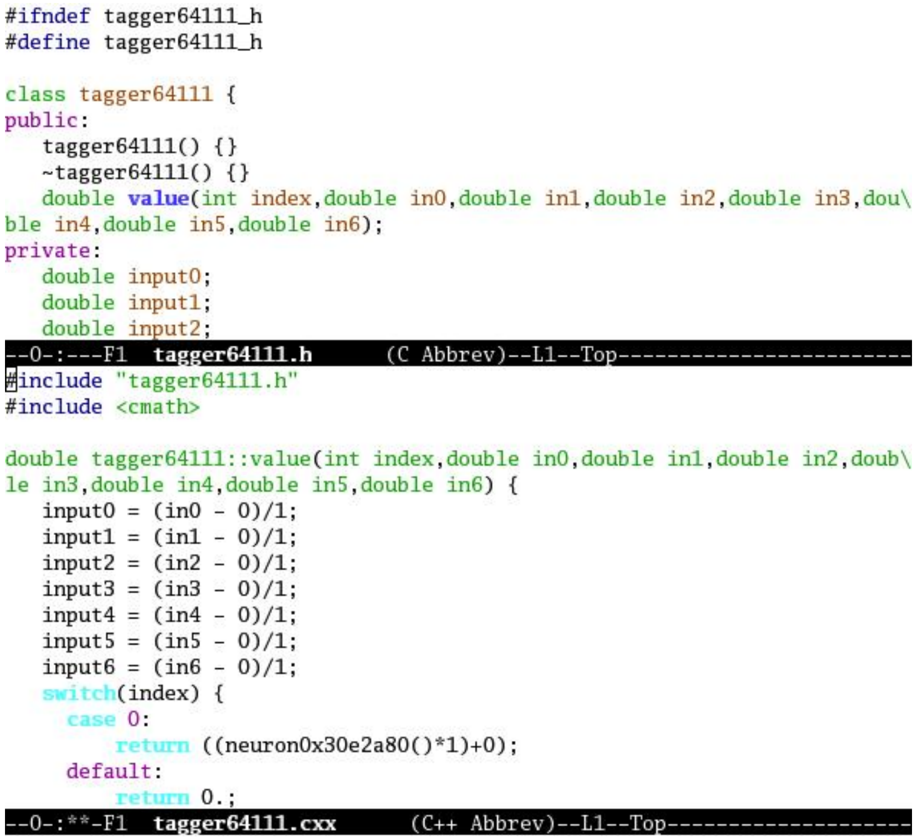
and testing a typical sample size was 10 000 events. In order to estimate the statistical significance of the classification power for the most promising MLP configuration, repeated experiments were made using additional 20 000 datasets.

We sampled a large number of NN architectures while examining the parameter space iteratively and running nearly 1000 jobs on the *ametisti* cluster and using approximately 500 CPU hours.

Typically the training lasted 1000-5000 epochs. The behavior of training and validation error for 1000 epochs is shown in Fig.3. The optimal generalisation value was found to be 1000-3000 epochs for the data used in [2] and for the new data sets. The larger data sets of CMS Monte Carlo data made possible improved testing (always independent event samples were used) and optimization on NN configurations. Networks were trained with the BFGS training algorithm which was found to be the most effective among the available methods.

Sensitivity analysis of promising neural classifiers were done by running analysis repeatedly with fresh random number seeds and by changing the input parameters minimally.

In our previous study [2] we found an optimal classifier configuration 7-10-5-1, (7 input nodes, 10 and 5 nodes in hidden layers and one output node) performing



```
#ifndef tagger64111_h
#define tagger64111_h

class tagger64111 {
public:
   tagger64111() {}
   ~tagger64111() {}
   double value(int index,double in0,double in1,double in2,double in3,dou\
ble in4,double in5,double in6);
private:
   double input0;
   double input1;
   double input2;
--0-:---F1  tagger64111.h       (C Abbrev)--L1--Top-----------------------
#include "tagger64111.h"
#include <cmath>

double tagger64111::value(int index,double in0,double in1,double in2,doub\
le in3,double in4,double in5,double in6) {
   input0 = (in0 - 0)/1;
   input1 = (in1 - 0)/1;
   input2 = (in2 - 0)/1;
   input3 = (in3 - 0)/1;
   input4 = (in4 - 0)/1;
   input5 = (in5 - 0)/1;
   input6 = (in6 - 0)/1;
   switch(index) {
     case 0:
         return ((neuron0x30e2a80()*1)+0);
     default:
         return 0.;
--0-:**-F1  tagger64111.cxx      (C++ Abbrev)--L1--Top-------------------
```

Fig. 6. Example of cluster job 64111 output: ROOT MLP generated standalone C++ code defining an optimized event-classifier

b-tagging with 42 % efficiency and 1 % mistagging probability. Starting from this configuration, using newly generated Monte Carlo data sets and careful teaching with better statistics, we were able to find a more optimal setup 7-12-5-1, with slightly better performance 44 $\pm$ 1 %. Also we found a new promising configuration 10-15-5-1, where three additional variables were introduced describing the particle momentum. Reference [2] gives a detailed description of the variables. Figure 5 shows an example of the classification power of this network.

We also found that a simple configuration with only one hidden layer (7-15-1) performed quite well, with b-tagging efficiency of 39 $\pm$ 1 % . The practical outcome of this study include ROOT generated C++ code defining optimized event-classifiers, shown in Fig. 6.

These results can be compared to an independent NN study to improve Higgs physics b-tagging reported in [23].

Our results indicate, that semi-automatic optimization for neural networks using the ROOT MLP implementation is feasible even in interpreted CINT mode. The usage of a cluster environment has enabled us to tune neural classifiers with high accuracy. First tests on a hybrid approach combining traditional cuts method, MLP and AdaBoost show promise of improved classification power compared to standard results.

6 Conclusions

ROOT and its parallel version PROOF provide very interesting possibilities for data analysis in many fields of science. The relative performance of two clusters have been measured with the benchmark tools provided. Using ROOT on the Rocks cluster *ametisti* has enabled us to study the physics performance of the optimized neural network classifiers. A promising NN configuration with previously unused variables was found and optimized neural classifiers were produced. The b-tagging efficiency was shown to improve from 42% to 44% with the more optimal neural network setup.

Comparing the traditional cuts method, with the MLP method and support vector machines and boosting is planned to be the topic of future work. Vector machines and boosting, rarely used in high energy physics, could prove to be promising tools to be used for LHC-era data analysis. The meta-algorithm AdaBoost, found to be less susceptible to the over fitting problem than most learning algorithms, is especially interesting to study.

References

1. Heikkinen, A., Lindén, T.: Validation of GEANT4 Bertini cascade nuclide production using parallel Root facility. In: Proc. of Computing in High Energy and Nuclear Physics, February 13–17, 2006, Mumbai, India (in press, 2006)
2. Heikkinen, A., Lehti, S.: Tagging b jets associated with heavy neutral MSSM Higgs bosons. NIM A 559, 195–198 (2006)
3. Rademakers, F., Goto, M., Canal, P., Brun, R.: ROOT Status and Future Developments. arXiv: cs.SE/0306078, `http://root.cern.ch/`

4. Ganis, G., et al.: PROOF – The Parallel ROOT Facility. In: Proc. of Computing in High Energy and Nuclear Physics 2006, 13 – 17 February 2006, Mumbai, India (In press)
5. Lehti, S.: Tagging b-Jets in $b\bar{b}H_{SUSY} \rightarrow \tau\tau$. CMS NOTE 2001/019
6. Weiser, C.: A Combined Secondary Vertex Based B-Tagging Algorithm in CMS. CMS NOTE 2006/014
7. Lehti, S.: Study of MSSM H/A$\rightarrow \tau\tau \rightarrow e\mu + X$ in CMS. CMS NOTE 2006/101
8. Kinnunen, R., Lehti, S.: Search for the heavy neutral MSSM Higgs bosons with the H/A$\rightarrow \boldsymbol{\tau}^{+}\boldsymbol{\tau}^{-} \rightarrow$ electron + jet decay mode. CMS NOTE 2006/075
9. Sjostrand, T., Lönnblad, L., Mrenna, S.: PYTHIA 6.2 Physics and Manual, hep-ph/010826, LU TP 01/21, 3rd edn. (April 2003)
10. Lai, H.L., Huston, J., Kuhlmann, S., Morfin, J., Olness, F., Owens, J.F., Pumplin, J., Tung, W.K.: Global QCD Analysis of Parton Structure of the Nucleon: CTEQ5 Parton Distributions, hep-ph/9903282. Eur. Phys. J. C12, 375–392 (2000)
11. ORCA, Object-oriented Reconstruction for CMS Analysis, `http://cmsdoc.cern.ch/orca`
12. CMS Collaboration: Data Acquisition & High-Level Trigger Technical Design Report, CERN/LHCC 2002-26, CMS TDR 6.2 (December 2002)
13. Chekanov, S.V.: Jet algorithms: A mini review, hep-ph/0211298
14. Adam, W., Mangano, B., Speer, Th., Todorov, T.: Track reconstruction in the CMS tracker, CMS NOTE 2006/041
15. Papadopoulos, P., Katz, M., Bruno, G.: NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. Concurrency Computat: Pract. Exper. 00. 1–20 (2002)
16. The list of the 500 fastest computers according to the Linpack benchmark, `http://www.top500.org/`
17. The homepage of NPACI Rocks cluster distribution, `http:// www.rocksclusters.org/`
18. Segneri, G., Palla, F.: Lifetime based b-tagging with CMS. CMS NOTE 2002/046
19. Caballero, I.G., Cano, D., Marco, R., Cuevas, J.: Prototype of a parallel analysis system for CMS using PROOF. In: Proc. of Computing in High Energy and Nuclear Physics, February 13-17, 2006, Mumbai, India (in press) (2006)
20. The Twiki page of PROOF, `http://root.cern.ch/twiki/bin/view/ROOT/PROOF`
21. The home page of xrootd, `http://xrootd.slac.stanford.edu`
22. Ballintijn, M., Roland, G., Gulbrandsen, K., Brun, R., Rademakers, F., Canal, P.: Super scaling PROOF to very large clusters. In: Proc. of Computing in High Energy and Nuclear Physics 2004, September 27 - October 1, 2004, Interlaken, Switzerland, CERN-2005-002, vol. 1 (2005)
23. Tentindo-Repond, S., Prosper, H.B., Bhat, P.C.: Neural Networks for Higgs Physics. In: Proceedings of Advanced Computing and Analysis Techniques in Physics Research, VII International Workshop ACAT 2000, Batavia, Illinois (2000)

A Model of Small-Scale Turbulence for Use in the PPM Gas Dynamics Scheme

Paul R. Woodward and David H. Porter

University of Minnesota, Laboratory for Computational Science & Engineering, 499 Walter Library, 117 Pleasant St. S. E., Minneapolis, Minnesota 55455, U.S.A
{paul, dhp}@lcse.umn.edu
http:www.lcse.umn.edu

Abstract. A model for unresolved, small-scale turbulent motions has been developed and implemented in the PPM gas dynamics code. The motivation for the model, the use of very large-scale simulation data in determining the model formulae and parameters, and validation tests of the model in the PPM code are described.

Keywords: turbulence, computational fluid dynamics.

1 Introduction

Our team at the University of Minnesota's Laboratory for Computational Science & Engineering (LCSE) has been simulating turbulent astrophysical fluid flows for many years using our PPM family of codes for the Euler equations of inviscid flow [1-4]. PPM uses interpolated parabolae to represent the subcell structures of Riemann invariants, an approximate nonlinear Riemann solver to obtain fluxes of conserved quantities, and directional operator splitting. The numerical dissipation of the scheme stabilizes the computation and, on scales of just a few grid cells, dissipates kinetic energy into heat in both shocks and in very small-scale turbulent eddies. The turbulent cascade on well resolved scales is treated faithfully by PPM, since it is an inviscid phenomenon. In our astrophysical flows, the details of the dissipation affect only scales that are far below our ability to represent them on the grid, while the amount of heat that is generated by the dissipation is controlled by the rate of energy transfer established by the resolved inviscid cascade together with the strict total energy conservation of the numerical scheme. In all these simulations (see [5] and references therein), we find that an enhancement of the velocity power spectrum just before the dissipation range is eventually produced. In comparisons with simulations of the Navier-Stokes equations, we found in [6] that this enhancement is a physical effect that is produced by a Navier-Stokes dissipation in much the way it is produced by our PPM scheme's numerical dissipation, although the Navier-Stokes terms cause enhancement of the power spectrum across a broader range of wavenumbers. This conclusion was later reinforced by the more finely resolved Navier-Stokes simulations of Yokokawa et al. [7] on the Earth Simulator. In the present work, we have designed an SGS model for turbulent motions that, when added to our PPM Euler scheme, or, we believe, to any similar Euler scheme, eliminates this enhancement of the near dissipation range spectrum.

2 Rate of Energy Transfer to Turbulence

Our initial focus in this work was on the rate, F, of energy transfer from large-scale to small-scale turbulent motions. As in the standard approach to large eddy simulation, we will distinguish these two scales by using a filter. We will apply this filter to results of PPM simulations of turbulent flows carried out on extremely fine grids, so that convergence studies indicate that the filtered results give an accurate representation of the large-scale flow, at least in appropriate statistical measures that apply in turbulent flows. We will use the detailed simulation results to evaluate the statistical averages below the scale of the filter that enter the fluid equations for the filtered quantities. Our convergence studies (see, for example, [6] or [8]) indicate that filter widths of 32 grid cell widths produce very accurate filtered flows and also quite accurate estimates of the sub-filter-scale statistical averages. We have found it important to choose a filter that does not tend to produce false signals, such as significant positive measures for the turbulent kinetic energy when a visual inspection of the flow clearly indicates that these measurements are inappropriate. The standard Gaussian filters fail on this criterion. We have chosen instead to construct within a filter box of 32^3 grid cells the first 10 lower-order spatial moments of the function to be filtered and to use these to construct a 10-coefficient polynomial representation of the function within the filter box which is then identified with the filtered result. Differences within the filter box between the actual function values and those of this quadratic polynomial fit are then identified with the fluctuating part of the function. This filtering process corresponds closely to the action of a standard numerical scheme. Like such a scheme, it produces an excellent result when the function within the filter box can be well represented by a strongly convergent Taylor series approximation. In a turbulent region in the limit of infinite Reynolds number that our turbulence model will seek to describe, such a Taylor series will not provide an adequate description of the function's behavior, and this is why we will need to supplement our numerical scheme with an SGS turbulence model.

We used data from an sPPM simulation of the Richtmyer-Meshkov instability of a shock-accelerated interface between denser and lighter gas to develop our initial model for F [9,10]. (sPPM is a simplified version of our PPM code.) We use DE_{turb}/Dt to denote the rate of change of the turbulent kinetic energy below the filter scale in the frame of reference moving with the filtered velocity. The standard filtering of the fluid equations gives the following equation for this quantity:

$$\begin{aligned}\frac{\partial E_{turb}}{\partial t} + \partial_j(\tilde{u}_j E_{turb}) &= \frac{DE_{turb}}{Dt} + E_{turb}\,\partial_j(\tilde{u}_j) \\ &= \left(\overline{p\,\partial_i u_i} - \bar{p}\,\partial_i\tilde{u}_i\right) - \tau_{ij}\,\partial_j\tilde{u}_i - \\ &\quad \partial_j\left(\overline{u_j p} - \tilde{u}_j\bar{p} - \tilde{u}_i\tau_{ij} + \tfrac{1}{2}\overline{\rho\, u_i^2\, u_j} - \tfrac{1}{2}\overline{\rho\, u_i^2}\,\tilde{u}_j\right)\end{aligned} \tag{1}$$

Here an overbar represents a volume-weighted and a tilde represents a mass-weighted average. The sub-filter-scale stress tensor, τ_{ij}, is defined by $\tau_{ij} = \overline{\rho u_i u_j} - \bar{\rho}\tilde{u}_i\tilde{u}_j = \bar{\rho}\,(\widetilde{u_i u_j} - \tilde{u}_i\tilde{u}_j)$, and the turbulent kinetic energy of sub-filter-scale motion, E_{turb}, is defined by $2E_{turb} = \overline{\rho u_i u_i} - \bar{\rho}\tilde{u}_i\tilde{u}_i = \tau_{ii}$. We identify the second term on the right above, $-\tau_{ij}\,\partial_j\tilde{u}_i$, as the rate of forward transfer of energy, F, from motions above the scale of the filter to those below it. Our analysis of the Richtmyer-Meshkov flow simulation data on its 8 billion cell grid led us to the following model for this term: $F_{Model} = A\,L_f^2\,\bar{\rho}\ \det(S_D) + C\,E_{turb}\ \nabla\cdot\tilde{u}$, where $A = -\,0.75$ and $C = -\,0.67$. Here L_f is the scale of the filter and $\det(S_D)$ is the determinant of the deviatoric rate of strain tensor, defined by $(S_D)_{ij} = \frac{1}{2}\left(\frac{\partial\tilde{u}_i}{\partial x_j} + \frac{\partial\tilde{u}_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\,\nabla\cdot\tilde{u}\right)$. We can combine the term $C\,E_{turb}\,\nabla\cdot\tilde{u}$ in this expression for F_{Model} with the similar term $E_{turb}\,\partial_j(\tilde{u}_j)$ on the left in our equation for DE_{turb}/Dt . If we define an effective sub-filter-scale turbulent pressure, q, by $q = (2/3)\ E_{turb}$, then, using the equation of continuity, we see (cf. [17]) that our two terms in the divergence of the filtered velocity combine to describe a variation under compression of this turbulent pressure with the 5/3 power of the density: $q \sim (\bar{\rho})^{5/3}$. As we pointed out in [9], the first term in F_{Model} proportional to $\det(S_D)$ relates the transfer of energy to small-scale turbulence to the topology of the local solenoidal flow field. This determinant is a rotational invariant, and since S_D is traceless and symmetric, in a frame in which it is diagonalized this determinant equals the product of its 3 eigenvalues, which are each positive or negative according to whether the flow is expanding or compressing in that dimension. If the flow is compressing in one dimension and expanding in the other two (remember that the trace must vanish), our model for F says that energy will be transferred from the larger scales of motion to small-scale turbulence, and E_{turb} will be increasing. This is the type of flow that occurs when you clap your hands. In a nearly inviscid gas, it tends to produce thin shear layers which quickly become unstable and produce turbulence. If the flow is compressing in two dimensions and expanding only in one, then our model for F predicts that energy in small-scale turbulent motions will be transferred to larger scales. This is the type of flow that results when you squeeze a tube of toothpaste. It is also like the flow in a tornado funnel. In such a flow, small vortices are stretched and tend to become aligned, and hence they tend to merge, forming larger structures and transferring their energy to larger scales. We have presented evidence, particularly in [12] and [13], that indeed this model for F correctly identifies the locations and strengths of regions of energy transfer to turbulence and in the reverse direction to the larger scales of the flow.

The first term on the right in our equation (1) for DE_{turb}/Dt above, the pdV work term $\left(\overline{p\,\partial_i u_i} - \bar{p}\,\partial_i\tilde{u}_i\right)$, we find from analysis of data from our PPM simulation of decaying Mach 1 turbulence [11] can be neglected relative to the second and third terms on the right, namely F and the final divergence term. Analysis of this same decaying turbulence simulation data shows that the final divergence term, $-\ \partial_j(\overline{u_j p} - \tilde{u}_j\bar{p} - \tilde{u}_i\tau_{ij} + \frac{1}{2}\overline{\rho\ u_i^2\ u_j} - \frac{1}{2}\overline{\rho\ u_i^2}\ \tilde{u}_j)$, can be modeled by a simple diffusion of E_{turb} as in: $-\,C_{diffuse}\,L_f\ \sqrt{2E_{turb}/\bar{\rho}}\ \ \nabla^2 E_{turb}$, with $C_{diffuse} = 0.07$ (see correlation plots in [17]). There is no term in our equation for DE_{turb}/Dt above that describes dissipation of the turbulent kinetic energy into heat, because we derived this equation from the inviscid Euler equations. However, such a term must exist, and it must have the same magnitude independent of the size of the physical (as opposed to numerical) viscosity, which we assume is extremely small. The long-time behavior of our simulation of decaying Mach 1 turbulence provides us with a measurement of this turbulence dissipation term. We carried such a PPM simulation on a 1000^3 grid out to a very long time. Fitting the long-time behavior to a model in which E_{turb} decays at a rate proportional to the eddy turn-over time of the principal energy containing modes, and realizing that this scale size is changing as the turbulence decays, we find that we may model the viscous dissipation of our sub-filter-scale turbulence by adding a term on the right in our equation for DE_{turb}/Dt that is $-\,C_{decay}(E_{turb}/L_f)\ \sqrt{2E_{turb}/\bar{\rho}}$, with $C_{decay} = 0.51$.

Modeling all these terms in equation (1) as described above gives us a turbulence model that we can add into our PPM gas dynamics code. Upon doing this, we find that the production of turbulent kinetic energy, E_{turb}, which we treat as a new fluid state variable described by an additional partial differential equation, occurs in about the right places and in about the right amounts. However, we also find that the enhancement of the near dissipation range velocity power spectrum is not eliminated. Therefore this model gets us part of the way to our goal, but not all the way. Our equation (1) involves a contraction, $-\tau_{ij}\ \partial_j \tilde{u}_i$, of the sub-filter-scale stress tensor with the velocity gradient tensor to produce F, the energy transfer rate that we have modeled. From the deficiency, just stated, of the model that results, we conclude that it is not enough to model this tensor contraction and that we must model the "shape" of the tensor τ_{ij}, that is, the ratios of its 6 independent components, as well. In our simple model, we used an eddy viscosity proportional to $\det(S_D)$, which has the effect of assigning the shape of the tensor S_{ij} to the tensor τ_{ij}. The analysis of Leonard [14] produces a model for τ_{ij} whose leading terms produce a contribution to the energy transfer rate $F = -\tau_{ij}\ \partial_j \tilde{u}_i$ that includes a term in $\det(S_D)$. We therefore chose this as our starting point for modeling the shape of the tensor τ_{ij}.

3 A More Complete Model of Subgrid-Scale Turbulence

The motivation for Leonard's model for τ_{ij} can be seen from a Taylor series analysis of the local flow. First we define a filter that gives us the simple average of a quantity within the filter volume. Then, to estimate the behavior of a velocity inside the filter volume, we expand each filtered velocity component in a Taylor series about its local average value over the filter scale and keep only the first-order terms. We can then derive the expression $\tau_{ij}/\bar{\rho} \;=\; \widetilde{u_i u_j} \;-\; \tilde{u}_i \tilde{u}_j \;\approx\; \frac{1}{12}\ (\partial \tilde{u}_i/\partial \tilde{x}_k)\ (\partial \tilde{u}_j/\partial \tilde{x}_k) \;\equiv\; \frac{1}{12}\ T_{ij}$, where we have introduced scaled coordinates $\tilde{x}_k$ centered on the filter box and in which the filter box width is unity. Note that only the terms involving the square of one of these scaled and centered coordinates contribute to the average. We have denoted Leonard's velocity gradient product tensor $(\partial \tilde{u}_i/\partial \tilde{x}_k)\ \ (\partial \tilde{u}_j/\partial \tilde{x}_k)$ by the symbol T_{ij} . It is important to realize that the first-order terms in the Taylor series that give rise to Leonard's expression are of course captured in our 10-coefficient polynomial representation of the filtered velocity field. In our view, they are not sub-filter-scale terms and thus should not appear in τ_{ij}. In fact, we would expect any numerical scheme worth discussing to incorporate these terms, which arise already in the complete absence of turbulence. All these terms and many higher-order ones as well are incorporated in our version of $\tilde{u}_i\tilde{u}_j$, and therefore they cancel these terms in $\widetilde{u_i u_j}$. However, using the data from our PPM simulation of decaying Mach 1 turbulence on the 2048^3 grid, we find that indeed $\tau_{ij}/\bar{\rho}$ is well correlated with T_{ij} in turbulent regions, as has been noted by others working with much coarser simulation data [19], although the coefficient of proportionality changes very significantly over time in a developing turbulent flow (see [17]). We therefore conclude that T_{ij} gives a good representation of the shape of the tensor τ_{ij} but not of its magnitude. In our analysis of the decaying turbulence data, we found that the model for the shape of τ_{ij} can be improved a bit by adding in a component proportional to S_{ij} (see [17]). However, we need not include this component in our turbulence model, because it has a purely dissipative effect, and our PPM scheme, like any other modern Euler method, already has a carefully tuned amount of dissipation for numerical stability both in smooth flow and in the presence of shocks.

We will use the shape of T_{ij} to model that of τ_{ij} for only the deviatoric (traceless) parts of both these tensors, which we denote by a subscript D. We will treat the compressional part of τ_{ij} through the turbulent pressure, q. The overall magnitude of $(\tau_D)_{ij}$ should be proportional to $\tau_{ii} = 2E_{turb}$, and we find that this expectation is verified by our data from the decaying turbulence run. If we can solve a partial differential equation for E_{turb}, then we would be able to get the magnitude of $(\tau_D)_{ij}$ from that solution and its shape from T_{ij}. This realization led us to the model:

$$(\tau_D)_{ij} = \alpha\, E_{turb}\, \frac{(T_D)_{ij}}{tr(T_{ij})} \quad ; \qquad T_{ij} = L_f^2 \frac{\partial \tilde{u}_i}{\partial x_k} \frac{\partial \tilde{u}_j}{\partial x_k} \tag{2}$$

where $L_f = 4\Delta x$ and $\alpha = 4.5$. The factors of L_f from T_{ij} cancel out, of course, in $(\tau_D)_{ij}$, but we have inserted them here to emphasize the dependence upon the assumed filter box width, which will enter the partial differential equation for E_{turb} non-trivially. The constant α is chosen so that the enhancement of the velocity power spectrum in the near dissipation range is just eliminated for coarse grid runs of the decaying Mach 1 turbulence problem using this SGS turbulence model. For a different numerical scheme than PPM, or for a different choice of the filter scale, L_f, we would expect α to assume a different value. However, for our model to be successful, this same value and choice of filter scale should work well for PPM in many turbulent flow applications, at all times during the development of those flows, and on all choices of grid resolution.

We discussed earlier how we might build a model partial differential equation, starting from equation (1), to solve for the time development of the turbulent kinetic energy, E_{turb}. We will re-express this here in terms of the equivalent turbulent pressure q:

$$\frac{\partial q}{\partial t} + u_i \frac{\partial q}{\partial x_i} = -\frac{5}{3} q \frac{\partial u_i}{\partial x_i} - \frac{3}{2} (\tau_D)_{ij} \frac{\partial u_i}{\partial x_j} - \sqrt{\frac{3q}{\rho}} \left(C_{diffuse} L_f \nabla^2 q + C_{decay} \frac{q}{L_f} \right) \tag{3}$$

$$(\tau_D)_{ij} = \frac{3}{2} \alpha \, q \frac{(T_D)_{ij}}{T_{kk}} + \beta L_f^2 \rho \det(S_D) \frac{(S_D)_{ij}}{(S_D)_{lm} (S_D)_{lm}} \tag{4}$$

$C_{diffuse} = 0.51$; $C_{decay} = 0.07$; $\alpha = 4.5$; $\beta = 0.01$, and $\beta = 0$ in smooth flow.

We relate the width of the filter box, L_f, to the grid cell width, Δx, by $L_f = 4\, \Delta x$. Note that we have added a second term, with coefficient β, in our model for $(\tau_D)_{ij}$. This is a turbulent kinetic energy generation term. We have set it proportional to $\det(S_D)$, because our earlier model analysis indicates that this will cause turbulent kinetic energy to be generated in roughly the right places in the flow. The amounts that will be generated will be controlled by the large-scale flow, as long as we choose the constant β small enough. The idea is that this term in $\det(S_D)$ will seed the flow with small amounts of turbulent kinetic energy. Once these "seeds" are "planted," the first term on the right in equation (4) will tend to produce exponential growth of turbulent kinetic energy until that growth becomes limited by the availability of the energy that is feeding this growth. That energy is the kinetic energy arriving continuously through the turbulent cascade in motions on scales just above the filter width. The presumption is that this cascade and hence this amount of energy is being accurately computed by the numerical scheme. Once E_{turb} grows substantially from its seed value, it should evolve independently from the way in which we seeded it. To prevent seeding this growth in smooth regions of the flow, we set β to zero where a standard PPM test (see [4]) indicates that the velocity field is smooth. In this model, we thus need only to set β so small that the size of the seed term in equation (4) is always small compared to the first term there, proportional to the constant α, once the local growth of E_{turb} (hence of q) saturates. We find that the choice $\beta = 0.01$ works well and that $\beta = 0.001$ also works, but produces a short delay in the generation of the turbulent energy. We find that $\beta = 0.1$ is too large, because it makes the second term on the right in equation (4) begin to compete with the first term and thus to produce too much dissipation in the near dissipation range of the velocity power spectrum.

To perform a fluid flow simulation with this model, we must add to equations (3) and (4) the fluid equations:

$$\frac{\partial \rho}{\partial t} = -\frac{\partial \rho u_i}{\partial x_i} \quad ; \quad \frac{\partial \rho u_i}{\partial t} = -\frac{\partial}{\partial x_j} \left((p+q)\, \delta_{ij} + \rho u_i u_j + (\tau_D)_{ij} \right) \tag{5}$$

$$\frac{\partial E}{\partial t} = -\frac{\partial}{\partial x_j} \left(u_j (E + p + q) + u_i (\tau_D)_{ij} \right) ; \quad E = \frac{p}{(\gamma - 1)} + \frac{3}{2} q + \frac{1}{2} \rho u_j u_j \tag{6}$$

Our PPM gas dynamics code uses directional operator splitting. We decompose each 1-D pass of the numerical algorithm into two steps. In the first step, we solve equations (5) and (6) above in the usual way, except for the complication of the turbulent pressure, q. In this first step, we set $(\tau_D)_{ij}$ to zero, since we will handle its contribution in the second step. We treat the gas, with its effective pressure of

$p+q$ as a fluid with an effective gamma-law equation of state, where $\gamma_{eff} = (\gamma\ p + 5q/3)\ /\ (p+q)$. The effective Eulerian sound speed is then given by $c^2_{eff} = (\gamma\ p + 5q/3)\ /\ \rho$. We follow the standard PPM procedure described in [4] to arrive at estimates of the time-averaged values of ρ and $(p+q)$ at the cell interfaces during the time step. From the density change along the averaged streamline crossing the cell interface, we find that we get a reasonable estimate for the change or jump in q by using the shock formula with γ_{eff}. From the jump in q, knowing the jump in $(p+q)$, we can then get the jump in p. This is not an obvious procedure. We have devised it with reference to a very detailed study of the propagation of a Mach 5 shock through a periodic brick of fully developed turbulence. This brick of turbulence was produced in our PPM simulation of decaying Mach 1 turbulence on the 2048^3 grid. We plan to refine our present treatment of shocks in our code through further shock-turbulence interaction studies. Nevertheless, the present technique appears to work rather well. It accounts in an approximate way for the large and sudden loss of turbulent kinetic energy to heat immediately following the viscous shock as the turbulence reestablishes local isotropy. At the end of the first step of this 1-D PPM pass, we need to update the individual pressures p and q. We have the updated total energy from solving equation (6), with $(\tau_D)_{ij}$ set to zero, as well as the updated kinetic energy from solving equation (5), but this is not sufficient information. To complete the system, we add equation (3) in which we set $(\tau_D)_{ij}$ to zero and also set both $C_{diffuse}$ and C_{decay} to zero, since we will handle these terms in the second step of this 1-D pass.

In the second step of our 1-D pass of the numerical algorithm, we do not allow the fluid to move. At the outset in this second step, we evaluate a standard PPM test (see [4]) for the smoothness of the velocity field, producing a measure of the lack of smoothness at each grid cell that ranges from 0 to 1. We also generate a smoother velocity field by taking for each velocity component half the present value plus a quarter of each neighbor value, doing this in the 3 grid directions in succession. Where the flow is not smooth, but not in shocks (which we detect as a standard part of the PPM scheme), we modify q by applying the term involving $(\tau_D)_{ij}$ in equation (3), using equation (4), and we also modify the velocity components by applying the terms involving $(\tau_D)_{ij}$ in equation (5). After this is done, the terms in $C_{diffuse}$ and C_{decay} in equation (3) are applied. The smoothed velocity components are used to evaluate the tensors T_{ij} and $(S_D)_{ij}$ in these formulae. It is a very important feature of our method that we limit the amount of energy that is transferred from the kinetic energy of small-scale turbulent motions, $E_{turb} = 3q/2$, to the large-scale flow in order that no more energy can be transferred than is available in this source. This constraint is applied locally at each cell interface. By solving our evolution equation for q, we know how much energy is available locally. Our scheme therefore cannot go into a runaway behavior in which extraction of turbulent energy results in conditions that cause estimates of the local amount of this energy to increase. This benefit alone, the prevention of unphysical runaway behavior, is well worth the cost of integrating the evolution equation for q. Other investigators working with the velocity gradient product tensor to model the SGS stress tensor τ_{ij} have inserted additional dissipative terms or introduced limiters that completely eliminate backscatter in order to address this problem [18,19].

4 Initial Results

We have of course used our PPM simulation of decaying Mach 1 turbulence on 2048^3 and 1000^3 grids ([11,15,16], the latter has a more complete data set) to test our modeling ideas and to set values for the constants α, $C_{diffuse}$, and C_{decay}. Nevertheless, it is still significant that a single value for α causes the enhancement of the velocity power spectrum in the near dissipation range to disappear for multiple grid resolutions, as shown in Figure 1. The agreement of the velocity power spectra in Figure 1 gives no information about the phases of these various modes. Although the turbulence is well developed and quite chaotic by time 1.86 in this problem, phase agreement can nevertheless be observed in renderings of the vorticity structures in a slice of the volume, as shown in Figure 2. There is quite a good correspondence between the results of PPM running with the turbulence model on a grid of 256^3 cells and a blending of the results of the run on the 2048^3 grid over boxes of 24^3 of its cells, or of 3^3 cells of the 256^3 grid. We have also tested the model successfully in a strongly anisotropic shear flow in which small-scale turbulence develops within large-scale breaking waves, and this will be reported in detail in a future article.

This work has been supported by grant DE-FG02-03ER25569 from the MICS program of the DoE Office of Science, through NSF CISE RR (CNS-0224424) and MRI (CNS-0421423) grants, through a donation of an ES-7000 computer from Unisys Corp., and by local support to the Laboratory for Computational

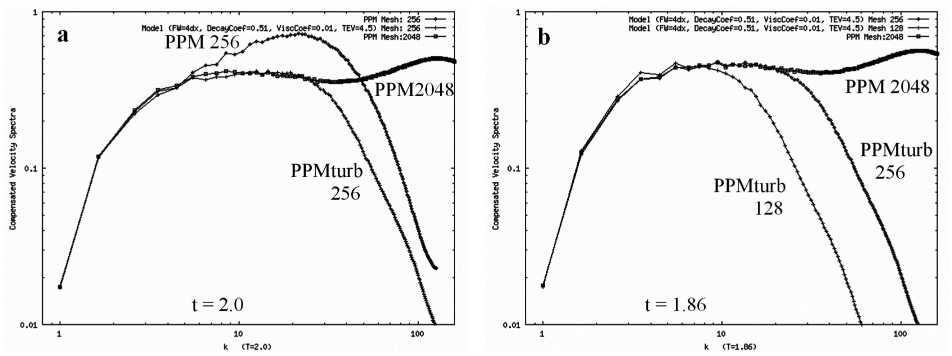


Fig. 1. Velocity power spectra of decaying Mach 1 turbulence are shown at late times, when the turbulence is fully developed. Spectra from the PPM run without a turbulence model on a grid of 2048^3 cells are compared with spectra from PPM runs with the turbulence model on 256^3 and 128^3 grids. A spectrum from a PPM run on a 256^3 grid without the model shows that the model eliminates the false enhancement of the spectrum in the near dissipation range of wavenumbers.

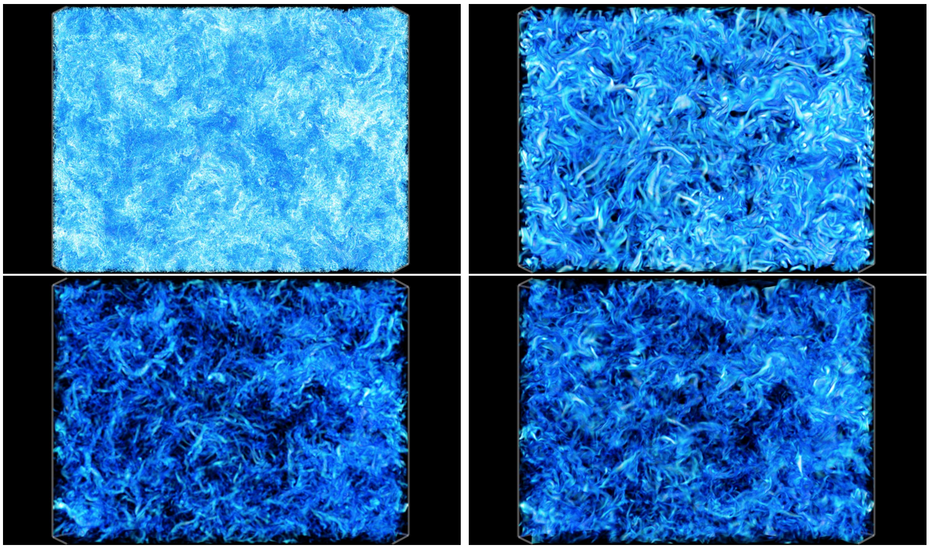


Fig. 2. Distributions of the magnitude of the vorticity in a slice of the volume in the decaying Mach 1 turbulence problem at time 1.86. All renderings are performed in the same way to enable direct comparisons. At the top are the PPM runs without a turbulence model on grids of 2048^3 (left) and 256^3 (right) cells. At the bottom left the data from the 2048^3 run has been blended over volumes of 3^3 cells of a $256\ ^3$ grid for comparison with the results, at bottom right, of a PPM run on a 256^3 grid using the turbulence model.

Science & Engineering (LCSE) from the University of Minnesota's Digital Technology Center and Minnesota Supercomputer Institute. We also acknowledge grants of time on the NSF TeraGrid cluster at NCSA and at the Pittsburgh Supercomputing Center.

References

1. Woodward, P.R., Colella, P.: J. Comput. Phys. 54, 115 (1984)
2. Colella, P., Woodward, P.R.: J. Comput. Phys. 54, 174 (1984)
3. Woodward, P.R.: In: Winkler, K.-H., Norman, M. L. (eds.) Astrophysical Radiation Hydrodynamics, p. 245. Reidel (1986)
4. Woodward, P.R.: A Complete Description of the PPM Compressible Gas Dynamics Scheme. In: Grinstein, F., Margolin, L., Rider, W. (eds.) Implicit Large Eddy Simulation: Computing Turbulent Fluid Dynamics (shorter version), Cambridge University Press, Cambridge (2006) LCSE internal report available from the main LCSE page at `www.lcse.umn.edu`
5. Woodward, P.R., Porter, D.H., Jacobs, M.: 3-D Simulations of Turbulent, Compressible Stellar Convection. In: Proc. 3-D Stellar Evolution Workshop, Univ. of Calif. Davis I.G.P.P. (July 2002), also available at `www.lcse.umn.edu/3Dstars`

6. Sytine, I.V., Porter, D.H., Woodward, P.R., Hodson, S.W., Winkler, K.-H.: J. Comput. Phys. 158, 225 (2000)
7. Yokokawa, M., Itakura, K., Uno, A., Ishihara, T., Kaneda, Y.: 16.4 Tflops Direct Numerical Simulation of Turbulence by a Fourier Spectral Method on the Earth Simulator. In: Proc. Supercomputing 2002, Baltimore (November 2002)
8. Porter, D.H., Woodward, P.R.: Using PPM to Model Turbulent Stellar Convection. In: Grinstein, F., Margolin, L., Rider, W. (eds.) Implicit Large Eddy Simulation: Computing Turbulent Fluid Dynamics, Cambridge University Press, Cambridge (2006), also available at `www.lcse.umn.edu/ILES`
9. Woodward, P.R., Porter, D.H., Sytine, I., Anderson, S.E., Mirin, A.A., Curtis, B.C., Cohen, R.H., Dannevik, W.P., Dimits, A.M., Eliason, D.E., Winkler, K.-H., Hodson, S.W.: Computational Fluid Dynamics. In: Ramos, E., Cisneros, G., Fernández-Flores, R., Santillan-González, A. (eds.) Proc. of the 4^{th} UNAM Supercomputing Conference, Mexico City, June 2000, World Scientific, Singapore (2001), available at `www.lcse.umn.edu/mexico`
10. Cohen, R.H., Dannevik, W.P., Dimits, A.M., Eliason, D.E., Mirin, A.A., Zhou, Y., Porter, D.H., Woodward, P.R.: Physics of Fluids 14, 3692 (2002)
11. Woodward, P.R., Anderson, S.E., Porter, D.H., Iyer, A.: Cluster Computing in the SHMOD Framework on the NSF TeraGrid. LCSE internal report (April 2004), available on the Web at `http://www.lcse.umn.edu/turb2048`
12. Porter, D.H., Woodward, P.R.: Simulating Compressible Turbulent Flow with the PPM Gas Dynamics Scheme. In: Grinstein, F., Margolin, L., Rider, W. (eds.) Implicit Large Eddy Simulation: Computing Turbulent Fluid Dynamics, Cambridge University Press, Cambridge (2006), also available at `www.lcse.umn.edu/ILES`
13. Woodward, P.R., Porter, D.H., Anderson, S.E., Edgar, B.K., Puthenveetil, A., Fuchs, T.: Parallel Computation of Turbulent Fluid Flows with the Piecewise-Parabolic Method. In: Proc. Parallel CFD 2005 Conf., Univ. Maryland, May, 2005 (2005), available at `http://www.lcse.umn.edu/parcfd`
14. Leonard, A.: Adv. Geophys. 18, 237 (1974)
15. LCSE movies from a variety of turbulent fluid flow simulations can be downloaded and/or viewed at, `www.lcse.umn.edu/MOVIES`
16. Porter, D.H., Woodward, P.R., Iyer, A.: Initial experiences with grid-based volume visualization of fluid flow simulations on PC clusters. In: Proc. Visualization and Data Analysis (VDA2005), San Jose, CA, January 2005 (accepted for publ.) (2005), available at `http://www.lcse.umn.edu/VDA2005`
17. Woodward, P.R., Porter, D.H., Anderson, S.E., Fuchs, T.: Large-Scale Simulations of Turbulent Stellar Convection and the Outlook for Petascale Computation. In: the SciDAC, conference, June 2006, Denver (2006), available at `http://www.scidac.gov/Conference2006/presentations/p_woodward_pres.pdf`
18. Vreman, B., Geurts, B., Kuerten, H.: Large-eddy simulation of the turbulent mixing layer. J. Fluid Mech. 339, 357–390 (1997)
19. Winckelmans, G.S., Wray, A.A., Vasilyev, O.V., Jeanmart, H.: Explicit-filtering large-eddy simulation using the tensor-diffusivity model supplemented by a dynamic Smagorinsky term. Physics of Fluids 13, 1385–1403 (2001)

Mapping in Heterogeneous Systems with Heuristic Methods

Juan-Pedro Martínez-Gallar[1], Francisco Almeida[2], and Domingo Giménez[3]

[1] Departamento de Estadística, Matemáticas e Informática, Universidad Miguel Hernández, ES-03202, Alicante, Spain
jp.martinez@uhm.es

[2] EIOC-ULL, Edificio Física-Matemáticas, ES-38271, La Laguna, Spain
falmeida@ull.es

[3] Universidad de Murcia, Departamento de Informática y Sistemas, ES-30071 Murcia, Spain
domingo@dif.um.es

Abstract. During recent years a large number of parallel routines and libraries have been developed. These routines have been conceived for homogeneous systems. Thanks to the evolution of technology, now it is quite usual to have heterogeneous systems. These routines and libraries need to be adapted to the new environment. There are at least two options. The routines could be rewritten, but this would be excessively costly in terms of time and money. Alternatively, the processes of a homogeneous routine can be mapped into the processors in the heterogeneous system. To do this, the development of efficient mapping techniques is necessary. Our approach to satisfactory mappings consists of modelling the execution time of the parallel routine, and obtaining the mapping that gives the minimum modelled execution time. Exact solutions to this problem are very time consumming. As an alternative, we are researching the application of heuristic techniques to solve this problem. This paper analyzes how Scatter Search can be applied to parallel iterative schemes.

1 Introduction

In the field of parallel computing, mapping problems must be solved to assign processes to processors in order to obtain a mapping with which a reduced execution time is obtained. The problem considered here is the one in which homogeneous processes are assigned to a parallel system, which may be homogeneous or heterogenous. In the case of a heterogeneous system, the type of algorithms used is often called HeHo [1] (Heterogeneous assignment of processes which have homogeneous distribution of data). The general mapping problem is widely known to be NP-complete [2,3]. Thus, efficient and optimal solutions have been found only for some particular mapping problems [4,5,6]. For other problems, approximate solutions are obtained with heuristic methods [2,7,8,9].

In previous works, the assignment of processes of a homogeneous program to a heterogeneous system was studied for a parallel dynamic programming scheme [10] and for dense linear algebra factorizations [11]. The problem has been solved by searching through a tree which includes all the possible processes to processors mappings. In this paper we propose to search through the tree with Scatter Search [12,13].

The paper is organized as follows. In Section 2 the tree used for the search is explained. Section 3 analyzes the scheme of a Scatter Search and different ways of applying the technique to the mapping problem. In Section 4 some experimental results are shown. Finally, Section 5 summarizes the conclusions and indicates future research directions.

2 The Mapping Tree

In figure 1, level number i represents the possible assignments of process number i to the corresponding processors in $[1, \ldots, P]$, where P is the number of processors in the system. The height of the tree depends on the number of processes used in the solution.

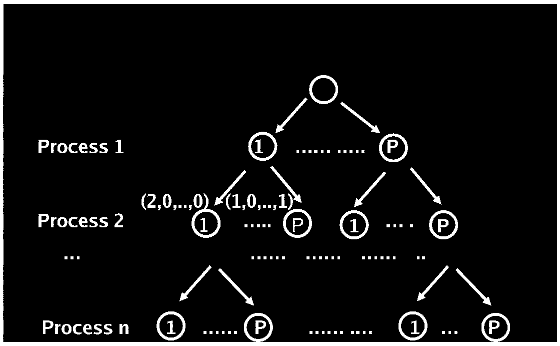


Fig. 1. Mapping processes $(1, \ldots, n)$ to processors $(1, \ldots, P)$

To obtain the mapping, an exact model of the execution time of the routine must be developed, and a search is performed in a tree representing the mappings. Each node in the tree represents one of the possible assignments, and it has the corresponding modelled execution time associated [10]. As an example, the figure has two nodes at level 2 labeled by their mapping array. The first has the value (2,0,...,0) because processes numbers 1 and 2 have been assigned to processor number 1 and no processes are assigned to the other processors. The second has the array (1,0,...,1) because process number 1 has been assigned to processor number 1, and process number 2 to processor number P.

The search through the tree is made with backtracking or branch and bound techniques, which consume a large amount of time, even when strategies to prune nodes in the tree are applied. Thus, the method does not scale well and it is only valid for small systems. For large systems, approximation methods (greedy) can be used to obtain the mapping in a reduced time, but the result is far from the optimum in some cases. This can be seen in table 1, where the times to decide the mapping (mapping time) using an exact (backtracking with pruning) and a greedy method are compared. The table also shows the execution time (exec. time) obtained with these mappings, in a system with 20 processors, and in a simulated configuration with 40 processors. In the paper, because the greedy method gives in some cases solutions far from the optimum, the proposed Scatter Search strategy is compared with a backtracking with pruning.

Table 1. Comparison of the execution time and the mapping time (in seconds) of greedy and backtracking methods when mapping in two heterogeneous systems (real and simulated)

method	System, 20 proc.		Simulation, 40 proc.	
	mapping time	exec. time	mapping time	exec. time
backtracking	0.122	2.96	111.369	5.07
greedy	0.001	5.09	0.009	5.72

The theoretical model requires a precise knowledge of the system parameters like basic arithmetic operations time, or sending and start-up times of a communication. Frequently it is very difficult to find an equation to represent the problem.

We consider parallel iterative schemes, which can be used in the solution of a large variety of problems [14,15]: dynamic programming, graph algorithms, genetic algorithms, iterative methods for the solution of linear systems, Fox and Cannon methods for matrix multiplication, Jacobi's relaxation, ... Additionally, the method can be applied to other problems and schemes, for example, linear algebra factorizations [11].

The case is considered in which the data to work with are assigned to the different processes in blocks of equal size. Each iteration consists of a computational part, and a communication step appears between two consecutive iterations. The total execution time is the addition of the execution times of the different iterations. Thus, it suffices to estimate the execution time of only one iteration that has a theoretical execution time of the form:

$$t(s, D) = t_c t_{comp}(s, D) + t_{comm}(s, D) \tag{1}$$

where s represents the problem size, D the number of processes used to solve the problem, t_c the cost of a basic arithmetic operation, t_{comp} the number of basic arithmetic operations, and t_{comm} the communication cost (it includes a term with t_s, start-up time, and another with t_w, word-sending time).

When estimating the computation and the communication part the type of system must be considered. The computation part is estimated by performing several sequential executions with different problem sizes, and using a least square adjustment. In a homogenous system, the cost of a basic arithmetic operation, t_c, is the same in the different processors, but in a heterogenous system the cost in each processor is different (the cost in processor i is t_{c_i}), and must be calculated for each one of the processors in the system.

Similarly, to estimate the cost of the communications, in a homogenous system t_s and t_w can be calculated between two processors, for example by running a ping-pong. In a heterogeneous system the cost between two processors depends on the two particular processors which perform the communications. To simplify the estimation there are several options. Adjustment by least square considering the system as homogeneous can be used [16]. Or the values of t_s and t_w between each pair of processors (using ping-pong) can be obtained, and the highest values would be included in the formula. The results have been obtained by least square adjustment considering the system as homogenous.

3 A Scatter Search Scheme

The previous techniques are either not fast enough or not efficient enough to get a satisfactory mapping. As an alternative, heuristic techniques which have been sucessfully applied to solve other hard optimization problems [13,17,18] can be applied. We have studied the application of the Scatter Search method.

Scatter Search is a very aggressive search method that attempts to find high quality solutions fast. It operates on a set of solutions by combining these solutions to create new ones [12,13]. From an initial reference set of solutions, new solutions are obtained by combining other solutions. Unlike a "population" in genetic algorithms, the reference set of solutions tends to be small, leading to a reduced execution time, which is essential for the goal we pursue. In genetic algorithms, two solutions are randomly chosen from the population and a "crossover" is used to generate a new solution. A typical population size in a genetic algorithm could consist of 100 elements, which are randomly sampled to create combinations. In contrast, typically the reference set in Scatter Search has 20 solutions or fewer, which are combined in a systematic way to create new solutions. The combination process could consider all pairs of solutions in the reference set, and it is necessary to keep the cardinality of the set small. Randomness is an important factor in this technique.

To apply the technique to our problem we consider that each element of the reference set of solutions represents a possible solution of the problem, that is, a possible mapping of processes to processors in the system (a node in the tree).

The elements with the lowest modelled execution time are included in the reference set, and those with the greatest distance function to the best elements are also included (e.g. the addition of the Euclidean distances could be used). The reason is that if only the best elements of the reference set are selected, the method could quickly converge to a local optimum, obtaining a solution which is far from the global optimum. The inclusion of elements which are far from the best elements will contribute to exploring the complete search space and to convergence to the global optimum.

The structure to represent the reference set is an array $D = (d_1, d_2, \ldots, d_P)$, with P being the number of processors in the system and d_i the number of processes assigned to processor i (figure 1). Fixing the number of processes to be used leads to the problem being solved more easily. However, the number of possible assignments is limited.

One scheme of the Scatter Search technique is shown in figure 2. Seven actions have been underlined. There are different possibilities for each of these actions. Experiments with some of these possibilities have been carried out, and the most important results are summarized.

```
generate initial reference set
improve initial reference set
while convergence not reached
  select elements to be combined
  combine selected elements
  improve combined elements
  include the most promising elements in the reference set
  include in the reference set the most scattered elements with respect
    to the most promising ones
endwhile
```

Fig. 2. A Scatter Search scheme

Previous underlined actions have been tuned to our mapping problem.

This is the basic Scatter Search scheme, but there is a large number of variants of Scatter Search. Different possibilities for each part of the scheme are discussed below:

Generate:

It is necessary to generate an initial reference set where each element represents a possible mapping, that is, a possible solution of the problem. But there are various possibilities. It is possible to randomly assign processes to processors, or to assign an equal number of processes to each processor.

Improve:

A greedy method is applied to each element in the initial reference set and in the sets obtained by combination of elements. For each element, all the mappings obtained by asigning one additional process to a processor are considered. The execution time of all the mappings are computed and the method advances to the mapping with lowest time. The iteration finishes when the execution times of all the new mappings are greater than that of the previous one.

Convergence:

The algorithm stops when the convergence of the problem is reached, that is, when a solution is found, but here again there are various posibilities. We can consider the convergence is reached when the best of the new solutions is not better than the best of the previous ones, or when the average of the new solutions is not better than that of the previous ones.

Select:

The algorithm selects certain elements of the present reference set for combination, but which and why? It is possible to select all the elements for combination, or to select the best elements to be combined with the worst ones (the best and the worst will be governed by the execution time function).

Combine:

Because Scatter Search is based on random decisions, each pair of selected elements is combined component to component, and the number of processes increases with more probability in processors with more computational capacity, and decreases in the slower ones.

Include:

As is seen in the algorithm, a new reference set is generated from the present reference set in every iteration. It includes the most promising elements and those most scattered with respect to the most promising ones. The most scattered elements are those furthest (using Euclidean distance) or those which are most "different" (number of different coordinates) from the most promising ones.

4 Experimental Results

For each of the previous hightlined actions there are many possibilities. The combine and improve actions are always the same. In the experiments only two options have been considered for generate and convergence actions. Thus, four different variants of the Scatter Search have been analyzed. To analyze the influence of the different possibilities a large number of tests have been carried out, both in simulations and in real systems.

In the simulations, the parameters of the system have been chosen as follows: the number of processors varies between 20 and 100, the maximum number of processes is obtained in the range $[15, \ldots, 15 + proc]$ where *proc* is the number of processors of the system, and the values of the computational parameters (t_{c_i}) have been randomly varied in the range $\left[10^{-10}, \ldots, 10^{-10} + 10^{-10} * randmax\right]$ where *randmax* is a specific number of each system. To estimate the values of the communication parameters (t_s and t_w) an adjustment by least square has been made considering a homogeneous system.

In table 2, the two possibilities considered of the generate and convergence actions are compared. The table shows the percentage of experiments in which the Scatter Search with the indicated possibility provides a total execution time lower than that obtained when the other option is considered. The results confirm what was expected: it is better to generate the initial reference set considering the number of processors of the system and it is also better that the algorithm finishes when the average of the new reference set is not better than the average of the previous reference set.

Thus, once the generate and convergence actions have been fixed, it is necessary to choose the best options for select and include actions.

Table 2. Comparison of two possibilites of generation of the initial reference set and of convergence criterion

Generate initial reference set	Best results	Convergence reached	Best results
Considering processors	80%	Average	80%
Randomly	20%	Best element	20%

Table 3. Comparison of two select and include criteria with respect to backtracking with pruning

	Select elements	
Include elements	All elements	Best vs worst elements
Longest distance	AE-LD: 90%	BW-LD: 85%
More different	AE-MD: 91%	BW-MD: 86%

Table 4. Modelled Times and Number of Iterations for different criteria

Simulation	Iterations	Initial Modelled Time	Final Modelled Time	Decision Time
AE-LD	96	5365.25	1796.04	5.89
AE-MD	69	5633.23	1797.83	5.33
BW-LD	11	5874.36	2237.24	0.18
BW-MD	2	5636.36	2411.21	0.47

Table 3 represents the comparisons between the four previous combinations with respect to an exact method (backtracking with pruning), with simulations named AE-LD, AE-MD, BW-LD and BW-MD. The parameters have been varied as previously explained. The numbers in the table represent the percentage of simulations in which the Scatter Search provides lower execution times (the mapping time plus the modeled time) with respect to the backtracking.

The best results are obtained when all elements are selected to be combined, and when the number of differences is used as a measure to determine the "most scattered" elements with respect to the most promising.

Table 4 shows the results obtained for the 4 previous combinations in a particular simulation. Different runnings would give different results, but the behaviour is similar. Simulations BW-LD and BW-MD have lower decision time (0.18 and 0.47 seconds, and 5.89 and 3.33 simulations AE-LD and AE-MD), but the total time is lower in AE-LD and AE-MD. AE-MD, which has the best figures in table 3, has a number of iterations smaller than AE-LD, but the modeled time are very similar, which implies that AE-MD is the best.

Hundreds of simulations have been made and the results are different: table 5 compares backtracking with pruning and Scatter Search techniques. The complexity is a parameter reflecting the complexity of computations in comparison with communications. Tests have been carried out with values 10, 50, 100, 200, 300, 400. It does not influence communications, only computation. The size of the problem has an influence on the computacional and the communication parts.

Values 10000, 50000, 100000, 500000, 750000 and 1000000 have been used in the experiments. In that way, tests have been carried out for small and large problems, and for problems with low and high computation/communication rate. In the experiments the parameters have been varied as previously explained. Each percentage represents the number of simulations where the corresponding technique is the best, that is, the total execution time (the mapping time together with the modelled time) is the lowest with respect to the other techniques.

For small problem sizes, exact techniques are better than Scatter Search, independently of the problem complexity, but, when the size grows, the Scatter Search technique outperforms the others. It can be said that in general the results are good, especially when the problem is big and complex.

Table 5. Comparison backtracking with pruning and Scatter Search techniques

Size	Complexity	backtracking	Scatter Search
Small $[1,\ldots,100000]$	Small $[1,\ldots,100]$	92%	8%
Small $[1,\ldots,100000]$	Big $[100,\ldots,400]$	73%	27%
Medium $[100000,\ldots,500000]$	Small $[1,\ldots,100]$	42%	58%
Medium $[100000,\ldots,500000]$	Big $[100,\ldots,400]$	40%	60%
Big $[500000,\ldots,1000000]$	Small $[1,\ldots,100]$	22%	78%
Big $[500000,\ldots,1000000]$	Big $[100,\ldots,400]$	5%	95%

Table 6 shows the grouped results for simulations with growing size and complexity. For each combination (SIM_i) of size, complexity, number of processors (proc) and number of processes, a large number of tests were made as previously explained. Each percentage represents the number of simulations where the Scatter Search technique is better than backtracking with pruning.

Table 6. Comparison of backtracking with pruning and Scatter Search techniques with increasing sizes and complexities

Simulation	Size	Complexity	System1	System2	System3	System4	System5
			20 proc	40 proc	60 proc	80 proc	100 proc
SIM_1	100000	100	100%	100%	60%	43%	33%
SIM_2	100000	400	100%	100%	96%	96%	70%
SIM_3	750000	100	100%	100%	96%	94%	77%
SIM_4	750000	400	100%	98%	100%	98%	97%

Results for a real system are also included. The system has 4 nodes, mono and bi-processors, with different processor speeds, and communicated by a fast ethernet network. The results are shown in table 7. When the size of the problem is small, exact techniques are better than Scatter Search, independently of the problem complexity, but when the size grows the Scatter Search performs equally well. The results are worse in this cluster than in the simulations because the explored tree is very small.

Table 7. Comparison of backtracking with pruning and Scatter Search techniques in a real system

Size	Complexity	backtracking	Scatter Search
Small $[1, \ldots, 100000]$	Small $[1, \ldots, 100]$	82%	18%
Small $[1, \ldots, 100000]$	Big $[100, \ldots, 400]$	80%	20%
Medium $[100000, \ldots, 500000]$	Small $[1, \ldots, 100]$	63%	37%
Medium $[100000, \ldots, 500000]$	Big $[100, \ldots, 400]$	59%	41%
Big $[500000, \ldots, 1000000]$	Small $[1, \ldots, 100]$	52%	48%
Big $[500000, \ldots, 1000000]$	Big $[100, \ldots, 400]$	47%	53%

5 Conclusions and Future Work

The application of heuristic techniques to processes to processors mapping has been studied. The technique is based on the exploration of a mapping tree where each node has an associated theoretical execution time. Heuristic techniques allow us to obtain a satisfactory mapping in a reasonable time, but some parameters of the basic heuristic search need to be tuned.

According to the results, the Scatter Search is a promising technique for mapping in heterogeneous systems, specially when they are large distributed systems. In the future we plan to apply to the same problem other heuristic techniques (tabu search, simulated annealing, genetic algorithms, ... [13,17,19]) which have been succesfully applied to other optimization problems, and to study the application of the method to other parallel algorithmic schemes [20,21].

Acknowledgment. This work has been funded in part by the Spanish MCYT and FEDER under Grants TIC2003-08238-C02-02 and TIN2005-09037-C02-01 and Fundación Séneca 02973/PI/05.

References

1. Kalinov, A., Lastovetsky, A.: Heterogeneous distribution of computations solving linear algebra problems on networks of heterogenous computers. Journal of Parallel and Distributed Computing 61, 520–535 (2001)
2. Zhao, W., Ramamritham, K.: Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constrains. The Journal of Systems and Software 7(3), 195–205 (1987)
3. Lennerstad, H., Lundberg, L.: Optimal Scheduling Results for Parallel Computing, SIAM News, 16–18 (1994)
4. Banino, C., Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Robert, Y.: Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. IEEE Transactions on Parallel and Distributed Systems 15(4), 319–330 (2004)
5. Legrand, A., Renard, H., Robert, Y., Vivien, F.: Mapping and Load-Balancing Iterative Computations. IEEE Transactions on Parallel and Distributed Systems 15(6), 546–558 (2004)

6. Almeida, F., González, D., Moreno, L.: Master-Slave Paradigm on Heterogeneous Systems: A Dynamic Programming Approach for the Optimal Mapping. In: 12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing, pp. 266–273 (2004)
7. Fujita, S., Masukawa, M., Tagashira, S.: A Fast Branch-and-Bound Scheme for the Multiprocessor Scheduling Problem with Communication Time. In: Proceedings of the, International Conference on Parallel Processing Workshop (ICPPW'03), Kaohsiung, Taiwan, pp. 104–111 (2003)
8. Sabin, G., Kettimuthu, R., Rajan, A., Sadayappan, P.: Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 87–104. Springer, Heidelberg (2003)
9. Almeida, F., González, D., Moreno, L., Rodríguez, C.: Pipelines on heterogeneous systems: models and tools. Concurrency - Practice and Experience 17(9), 1173–1195 (2005)
10. Cuenca, J., Giménez, D., Martínez-Gallar, J.P.: Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. Parallel Computing 31, 711–735 (2005)
11. Cuenca, J., García, L.-P., Giménez, D., Dongarra, J.: Processes Distribution of Homogeneous Parallel Linear Algebra Routines on Heterogeneous Clusters. In: Proceedings of the 2005 IEEE International Conference on Cluster Computing, IEEE, Boston (2005)
12. Hung, W.N.N., Song, X.: BDD Variable Ordering By Scatter Search. In: Proc. Int. Conf. Computer Design: VLSI in Computers & Processors, IEEE, Los Alamitos (2001)
13. Dréo, J., Pétrowski, A., Siarry, P., Taillard, E.: Metaheuristics for Hard Optimization. Springer, Heidelberg (2005)
14. Brassard, G., Bratley, P.: Fundamentals of Algorithms. Prentice-Hall, Englewood Cliffs (1996)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1991)
16. Giménez, D., Martínez-Gallar, J.P.: Automatic Optimization in Parallel Dynamic Programming Schemes. In: Daydé, M., Dongarra, J.J., Hernández, V., Palma, J.M.L.M. (eds.) VECPAR 2004. LNCS, vol. 3402, pp. 639–649. Springer, Heidelberg (2005)
17. Hromkovič, J.: Algorithmics for Hard Problems, 2nd edn. Springer, Heidelberg (2003)
18. Pérez, M., Almeida, F., Moreno-Vega, J.M.: On the Use of Path Relinking for the p-Hub Median Problem. In: Gottlieb, J., Raidl, G.R. (eds.) EvoCOP 2004. LNCS, vol. 3004, pp. 155–164. Springer, Heidelberg (2004)
19. Blum, C., Roli, A.: Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Computing Surveys 35(3), 268–308 (2003)
20. Wilkinson, B., Allen, M.: Parallel Programming. Prentice Hall, Englewood Cliffs (1999)
21. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Reading (2003)

The parXXL Environment: Scalable Fine Grained Development for Large Coarse Grained Platforms

Jens Gustedt[1], Stéphane Vialle[2], and Amelia De Vivo[3,*]

[1] INRIA Lorraine & LORIA, France
Jens.Gustedt@loria.fr
[2] SUPELEC, 2 Rue Edouard Belin, FR-57070 Metz, France
Stephane.Vialle@supelec.fr
[3] Università degli Studi della Basilicata, Italy

Abstract. We present a new integrated environment for cellular computing and other fine grained applications. It is based upon previous developments concerning cellular computing environments (the ParCeL family) and coarse grained algorithms (the SSCRAP toolbox). It is aimed to be portable and efficient, and at the same time to offer a comfortable abstraction for the developer of fine grained programs. A first campaign of benchmarks shows promising results on clusters.

1 Motivations and Objectives

Nowadays, many research areas consider multi-scale simulations based on *ab initio* computations: they aim to simulate complex macroscopic systems at microscopic level, using fundamental physical laws. Obviously, huge amounts of CPU are mandatory to run these simulations. Modern supercomputers and Grids, with large and scalable number of powerful processors, are interesting architectures to support these simulations.

However designers and developers of algorithms and code for large scale applications are often confronted with a paradoxical situation: their modeling and thinking is *fine-grained*, speaking *e.g.* of atoms, cells, items, protein bases and alike, whereas modern computing architectures are *coarse-grained* providing few processors (up to several thousands $\approx 10^3$) to potentially huge amount of data (thousands of billions of bytes $\approx 10^{12}$) and linking a substantial amount of resources (memory in particular) to each processor. Only few tools (for both, modeling and implementation) are provided to close this gap in expectation, competence and education.

This article introduces the parXXL development environment, specially designed to close this gap between *fine-grained* modeling and *coarse-grained* computing architectures. It stems from two previous research projects that have

* In memoriam to our colleague Amelia De Vivo who passed away during the PARA-2006 conference on June 21st, 2006, in Umeå, Sweden.

investigated optimization of computing resources (CPU, memory, communications, synchronization ...) and cellular oriented programming (to implement *fine-grained* models). Some collaborations with researchers in optic components and hot plasma (from LMOPS and LPMIA laboratories) guide parXXL design to an ease-to-use tool, and allow to identify collateral challenges. For example, some hot plasma simulation codes of our partners have been specially designed for global shared memory parallel computers. Intensive computation steps are split by optimized data rearrangement operations inside the global shared memory. But on large distributed architectures this kind of operations would be prohibitive. Some codes need to be re-designed and based on local computations, in order to support efficient runs on large distributed memory architectures and straightforward *fine-grained* implementations with parXXL. We are convinced that this algorithmic and programming methodology is required to achieve large multi-scale simulations.

2 Related Work

On the modeling side, Valiant's seminal paper on the BSP, see [1], has triggered a lot of work on different sides (modeling, algorithms, implementations and experiments) that showed very interesting results on narrowing the gap between, on one hand, fine grained data structures and algorithms and, on the other, coarse grained architectures. But when coming to real life, code developers are usually left alone with the *classical* interfaces, even when they implement with a BSP-like model in mind.

Development environment with explicit coarse grained parallelism, like MPI and OpenMP, usually lead to efficient executions but are not adapted to fine grained programming, and require experimented parallel developers. At the opposite, high level computation tools like Mathematica or Mathlab are comfortable tools to implement a simulation model from mathematical equations. But these high level tools have poor performances and use C++ code generators and classical distributed C++ libraries to create more efficient codes (an interface between parXXL and a Mathematica EDP solver based on cellular automata is under development). Some distributed OS managing a virtual shared memory like Kerrighed [2] allow to easily implement parallel algorithms, but they are limited to small clusters and require coarse-grained algorithms and optimized memory management to achieve performances.

Many generic cellular languages and distributed object libraries have been designed for parallel and distributed architectures, like Cape [3] or Carpet [4]. But they focus on *cellular automata* with cell connection limited to a predefined neighboring and with synchronous cell communications (similar to the *buffered* parXXL mode). Moreover, these researches seems to have slowed down since 2000. Finally, some Java based distributed environments exist, based on message passing and remote method invocations like ProActive [5] or on virtual shared memory like JavaSpaces [6]. Our personal experiments have exhibited good speedup and good scalability, and Java Virtual Machine performances are improving. But Java has not been yet adopted by the scientific community to implement intensive computations, and remains slower than C++.

So, implementing dynamic data structures (such as *cellular networks*) efficiently on a large scale often remains an insurmountable hurdle for real life applications. parXXL as proposed in this paper was created to lower that hurdle, in that it implements a general purpose development environment for fine grained computation on large parallel and distributed systems.

3 Software Architecture

The parXXL development environment is split into several, well-identified layers which historically come from two different project sources, SSCRAP and ParCeL6. Its software architecture is introduced on Fig. 1, and demonstrates the split of these two main parts into the different layers. The (former) SSCRAP part introduces all the necessary parts to allow for an efficient programming in coarse grained environments; interfaces for the C++ programming language, the POSIX system calls, tools for benchmarking, a memory abstraction layer and the runtime communication and control. The (former) ParCeL6 part introduces a *cellular* development environment and a set of predefined and optimized cell networks. These programming models of SSCRAP and ParCeL6 are detailed in the next sections.

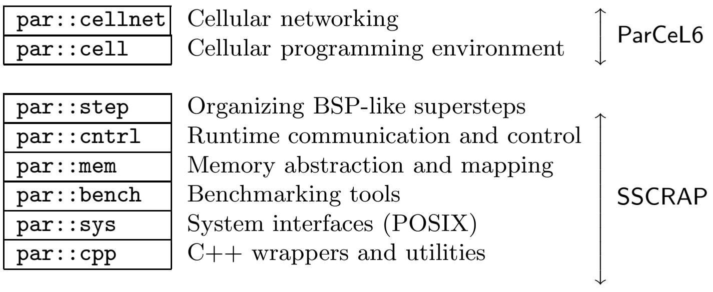

par::cellnet
Cellular networking
par::cell
Cellular programming environment
ParCeL6
par::step
Organizing BSP-like supersteps
par::cntrl
Runtime communication and control
par::mem
Memory abstraction and mapping
par::bench
Benchmarking tools
par::sys
System interfaces (POSIX)
par::cpp
C++ wrappers and utilities
SSCRAP

Fig. 1. parXXL software architecture

4 SSCRAP Programming Model

SSCRAP is a programming environment that is based on an extension of the BSP programming model [1], called PRO [7]. It proved to be quite efficient for a variety of algorithms and platforms, see [8]. Its main features what concern this paper are:

Supersteps with relaxed synchronization: Originally, BSP was designed with strong synchronization between the supersteps. PRO (and thus SSCRAP) allows a process to resume computation as soon as it receives all necessary data for the next superstep. The `par::cntrl` layer (see Fig. 1) implements these features in parXXL.

A well identified range of applicability: SSCRAP is clearly designed and optimized for *coarse grained* architectures. These are architectures for which each processor has access to a private memory that allows it to keep track of one communication to every other processor. If the platform has p processors, as a minimal condition this private memory must thus hold p machine words, a fact to which we refer as the architecture having "*substantially more memory than there are processors*". All modern high performance computing architectures (mainframes and clusters) easily fulfill this criterion.

Comfortable encapsulation of data: The work horse of SSCRAP is a data type (`chunk` in the `par::mem` layer) that encapsulates data situated on different supports such as memory and files which then can be mapped efficiently into the address space of the processes. Thereby SSCRAP can efficiently handle huge data (*e.g* larger than the address space) without imposing complex maintenance operations to the programmer.

Portability: SSCRAP is uniquely based on normalized system interfaces (`par::sys`), most important are POSIX file systems, POSIX threads and MPI for communication in distributed environments. Therefore it should run without modification on all systems that implement the corresponding POSIX system calls and/or provide a decent MPI implementation.

Performance: This portability is *not* obtained by trading for efficiency. In the contrary, we provide two run-times, one for shared memory architectures (threads) and one for distributed computing (MPI). These are designed to get the best out of their respective context: avoiding unnecessary copies on shared memory and latency problems when distributed. All this is achieved by only linking against the respective library, no recompilation is necessary.

5 ParCeL6 Programming Model

The `par::cell` level of parXXL architecture (see Fig. 1) implements the ParCeL6 *extended cellular* programming model [9]. It is based on *cells* that are distributed on different processors, and on a sequential *master* program: ParCeL6 developers design and implement some cell behavior functions, and a sequential program to install and to control a parallel cellular net. This mixed programming model is easy to use and facilitates the design of *cellular servers*: a classical client can connect to the sequential program, that runs cellular computations *on demand.* Main features of ParCeL6 cellular model are:

A dynamic cellular network: Starting from an empty network of cells, the sequential program creates cells on all available processors. Each cell has an individual set of parameters, and the first action of these cells is usually to connect each other to create a cellular network (a cell output can be connected to an unlimited number of cell inputs). This network is dynamic and may evolve at any point of the execution (cells and connections can be created or removed).

Six cell components: A cell is composed of (1) a unique *cell registration*, some (2) *parameters* and (3) *private variables*, (4) some *cell behavior functions*,

(5) a unique multi valued *output channel*, and (6) several multi valued *input channels*. The first is imposed by ParCeL6 mechanisms, the others are defined by the developer.

A cyclic/BSP execution of the cell net: A ParCeL6 cycle consists of three steps: computation, net evolution and communication. Each cell is activated once during the computation step, where it sequentially reads its inputs, updates its output, and issues some cell net *evolution requests*. These requests define, kill, connect or disconnect some cells, and are executed during the net evolution step.

Three modes of cellular communications: During the communication step, *buffered outputs* are copied to their connected cell inputs. Their propagation is fast and is adapted to synchronous fine grained computation (cell inputs do not change during the computation steps). The propagation of a *direct output* to a connected cell input is triggered each time a *refresh* command is executed for it. This mechanism has a large overhead but is required by some asynchronous fine grained computations [9]. *Hybrid outputs* are an attempt to get both fast and asynchronous cellular computations: they propagate their value one time per computation step and per processor (cells on different processors can read different values during one computation step).

Moreover, some *collector* mechanisms allow the cells to save data during their computation steps and the *master program* to gather, sort and store these data at the end of a computation step. Symmetrically, some *global* cell net communication mechanisms allow the sequential program to send input data to the cells (like camera images).

Finally, a classic ParCeL6 application code includes three main parts: the *sequential master routine* controlling the cell net installation and running cellular computation steps, the cell creation and connection operations to establish the cell net (can be easily implemented using the cell net library, see next section), and the cell computation functions that implement a fine grained computing model (like Maxwell equations, neural network computations...). So, a fine grained algorithm can be straightforwardly implemented on ParCeL6, especially when an adapted cell net template exists in the library, without dealing with parallel processing difficulties.

6 parXXL Main Functionalities

Optimized Cell Network Library. The `par::cellnet` library (see Fig. 1) is a collection of *cell network installers*: application code can easily deploy a cell network just using an *installer* object. Each *installer* has to be set with the application cell behavior functions, the cell parameter and cell variable types, and the cell network size. Then, it installs the cells and their parameters, and connects the cells according to a predefined communication scheme. Deployments are optimized: (1) the number of cells is balanced among the processors, (2) with preference neighboring cells are installed on the same processor, and (3) cell net installation is split into small steps to limit the memory required by the deployment operations.

The `par::cellnet` library currently includes network *installers* that place the cells on a 2- or 3-dimensional grid, and that will connect a given cell to all its neighbors that are 'close' with respect to a certain 'norm': e.g an installer of type

```
par::cellnet::mesh< 3, L2, 2, applicationType >
```

will position the cells on a 3-dimensional grid, and connect all cells that are at distance 2 in the Euclidean (L_2) norm. The remaining information of the particular application network (such as the individual cell functions etc) is specified via the type parameter `applicationType` that is defined by the application. Currently available norms are L_0, L_1 and L_2.

We are currently working on an extension of this setting to general dimensions. More generally, other types of regular networks (honeycomb, regular crystals) and other types of norms (e.g ellipses) may be implemented easily if needed.

Process Specification. Generally a parXXL program executes several parXXL-processes in an MIMD fashion. The kind of execution (POSIX threads or MPI processes) and the number of processes is not fixed in the code but only decided at link or launch time, respectively.

According to ParCeL6 programming model (see Section 5), the `par::cell` layer restricts the MIMD execution model so that a `par::cell` program is composed of a sequential *master* program and a set of *worker* processes hosting and running cells. The *master* process installs and controls the cell net deployed on the *workers* using some high level cell management functions described above.

This *master-worker* architecture is adapted to many scientific computing applications, but *industrial* applications use *client-server* architectures. To support both scientific and industrial applications, the parXXL *master* process may also implement a *server* interface (installing a cell net, accepting client connections and running cell net computations *on demand*) and run on a specific *server* machine. The user can point out this machine at runtime among the pool used to distribute the parXXL program, using the `-s` option (`S`erver name). For example, on a cluster using the MPI parXXL runtime: "`mpirun -np 100 MyAppli -s PE20 ...`" runs the application "MyAppli" on 100 *workers* and installs the *master-server* process on the "PE20" machine.

To avoid the *master-server* being overloaded by cell computations, the optimized cell net library `par::cellnet` allows to install or not to install cells on the *master-server* process, using the `-E` option (`E`xclude) at runtime. By default, the *master* process is the the parXXL-process 0 and hosts cells.

Cellular Network Management. The main parXXL functions to easily manage a cellular network from the sequential program of the master process are shown in Listings 1 and 2. They are all members of the `par::cell::context_t` class.

Function `define_cell` allows to define a new cell, specifying its number of output values, its connection mode (*buffered*, or *hybrid*, see Section 5). parXXL

```
int define_cell(descriptor_t const& Descr, register_t& Reg);
int kill_cell(register_t const& Reg);

template< class TPARAM >
    int define_cell_param(TPARAM const& param,
                          register_t const& Reg);
```

Listing 1. `par::cell::context_t`, main cell management functions

```
int conductCellUpgrade(void);
int conductParamInstall(void);

int conductComputation(ActivKind_t kind, size_t PermutIndex);
int conductLinkUpgrade(void);
int conductOutPropagate(void);
int conductCollect(size_t CollectorId);

int conductHalt(void);
```

Listing 2. `par::cell::context_t`, main cellular network management functions

defines its registration and its host processor. However it is possible to specify the host processor to optimize the cell net mapping; this is done by the `par::cellnet` library to create optimized cell networks. Usually the cells are defined by the sequential program of the master process, but they can be defined on any processor in parallel. This strategy will be exploited in the next version of the `par::cellnet` library to create larger networks faster. When some cells have been defined on one or several processors, function `conductCellUpgrade` executed on the master processor runs some processor communications and creates cells on different processors. Similarly, functions `define_cell_param` and `conductParamInstall` allow to define and associate some datastructures to cells (the cell parameters) and to send and install these parameters in the corresponding cell bodies on their host processors.

The next group of functions is usually called in a *computation loop* exploiting the cellular network. Function `conductComputation` activates all cells on each processor for one compute cycle: each cell runs its current behavior function once. On a particular parXXL-processor the *order* of the cell activation may be specified: the order of their storage (default), its reverse, or in the specific order of a permutation table. In most cases, this order has no impact on the program result and has not to be considered in the design of the program. But some rare and *asynchronous* cellular programs are sensitive to the cell activation order (when using the *hybrid* communication mode). So, parXXL allows to quickly change this order to check the sensitivity of the program to this parameter using the parameters of function `conductComputation`.

Function `conductLinkUpgrade` allows to establish the cell links that were defined during the previous cell computation step. It generates some processor communications and datastructure update, that are mandatory to send cell output into connected cell input buffers on remote processors. Then function `conductOutPropagate` can route each *buffered* cell output to its connected input cell buffers, and the *hybrid* cell outputs can be routed automatically during the cell computation steps. Functions `conductCellUpgrade` and `conductLinkUpgrade` have only to be called when a the cellular network is established or changes during a cycle; in the common case that a cellular network is created and linked during the first cycles and fixed thereafter, they may be avoided once the network structure remains stable.

Function `conductCollect` is called to gather data that cells stored in a *collector* during the previous computation steps on the master process. Thus, *collectors* are distributed datastructures allowing to *collect* some results on the master process.

The last function, `conductHalt`, allows to halt all the processes excepted the master process running the sequential program. No parXXL function can be called after this function has been executed.

Memory Allocation. As already mentioned above, efficient handling of large data sets is crucial for a good performance of data intensive computations. The template class `par::mem::chunk` provides comfortable tools that achieve that goal. Its main characteristic is that it clearly separates the *allocation* of memory from the effective *access* to the data.

Allocation can be done on the heap (encapsulating `malloc`), at a fixed address (*e.g* for hardware buffers) or by mapping a file or POSIX memory segment into the address space (encapsulating `open`, `shm_open` and `mmap`). By default the decision between these different choices is left for the time of execution and can thus easily be adapted according the needs of a specific architecture.

Access to the data is obtained by an operation called *mapping*. Mapping associates an address for the data in the address space of the parXXL process and returns a pointer where the programmer may access it. When the memory is not used anymore, it will in general be unmapped. The template class `par::mem::apointer` provides a comfortable user interface that, as the name indicates, may basically be used as if it was an ordinary C pointer.

Mapping and unmapping can happen several times without the data being lost. In general it is much healthier for the application to free resources during the time they are not needed. In particular the system may chose to *relocate* the data at different addresses for different mapping periods, and will be thereby be able to react to increase or decrease of the size of individual `chunk` s.

Mapping can also just request parts of the total memory (called *window* in parXXL). In such a way a program may *e.g* handle a large file quite efficiently: it may just map (and unmap) medium sized `chunks` one after another and handle them separately. Thereby a program may handle files that do not fit entirely into RAM or do not even fit in the address space of the architecture (4GiB for 32bit architectures).

An important property of `chunks` is that they may *grow* while they are not mapped. `par::mem::stack` uses this property for a simple stack data structure. This is intensively used by the `par::cell` layer to collect (and withhold) data during a computation phase on each processor before this data then can be communicated in its entirety in a communication phase. Thereby parXXL is able to avoid the fragmentation of cell communications into large numbers of small and unefficient messages (distributed architecture) or memory writes (shared memory).

7 Application and Performance Examples

Application Introduction. To start to validate the scalability of parXXL we have designed and experimented a 3D Jacobi relaxation on a *cube* of cells, with up to hundred millions of cells. This application has been implemented like a classical parXXL program. The sequential *master* process installs a cellular network on a pool of *workers*, and conducts a computational loop executed by each *worker*. There is no client-server mechanism implemented in this application, but it could easily be turned into a parallel Jacobi relaxation server with parXXL functionalities (see Section 6).

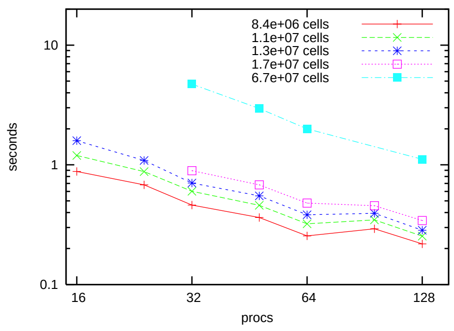
During the cell net installation steps, the cells are created with one output value, and are connected to their neighbor cells: up to six neighbors for a cell inside the cube. To easily deploy large cubes of cells, we have used the `par::cellnet` library, that installs optimized cellular networks (see Section 6). Then, the parXXL program enters a long loop of cell computations and cell output propagation (to the connected cells). The cells inside the cube update their output value with the average of their neighbor output values, while cells on the cube border maintain their output value unchanged. All the cell outputs of this application are routed to the connected cell inputs after each computation steps (*buffered* communication mode). At the end of the relaxation, some slices of the cell cube results are *collected* on the *master* process to be stored or displayed (see *collector* introduction in Section 5).

Experimental Performances. Fig. 2(a) shows for different problem sizes how execution time of a relaxation cycle decreases as a function of the number of processors. The benchmark platform is the *Grid-eXplorer*[1] cluster composed of bi-processor machines (a large PC cluster), running parXXL and its MPI-based runtime. These experiments exhibit regular decreases.

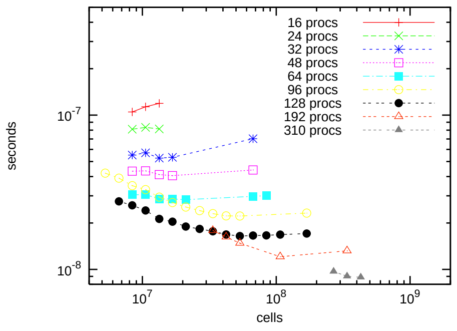
parXXL scales: On large enough problems, for the same problem size the time decreases linearly when increasing the number of processors.

However, the execution time of a complete relaxation cycle depends on the problem size (the number of cells): it has complexity $O(N)$, where N is the number of cells in the particular parXXL execution. To easily compare the execution times

[1] `https://www.grid5000.fr/mediawiki/index.php/Orsay:Home`



(a) total time per cell-cycle



(b) amortized time per cell and cycle

Fig. 2. Benchmarks on Grid-eXplorer: up to 400 million cells using up to 310 processors

for increasing problem sizes, in Fig. 2(b) we show the average time to run a cell once: the execution time per cycle and per cell.

parXXL is robust: This time remains constant for a fixed number of processors, independently of the problem size or still decreases when running more cells on more processors (see the curves for 128, 192 and 310 processors).

By that, parXXL succeeded to scale up to 400 millions of cells and 310 processors on a PC cluster.

8 Conclusion and Perspectives

Main parts of the parXXL architecture are implemented and operational, and first experiments show that the parXXL architecture scales up to some hundred processors for a large, fine grained application. Further development will consist in the following: (1) improve the *global communication mechanisms* to send data from the sequential program to the cells, such as camera images, (2) design and implement an efficient *hybrid cell communication mode*, (3) full generalization of the `par::cellnet` regular networks and a parallel deployment of these nets.

Future experiment will be run on a larger number of processors of the Grid-eXplorer machine, and on a Grid of clusters using Grid5000 (the French nation-wide experimental Grid). Our short-term goal by the end of this year (2006) is to deploy more than a billion of cells for simulations of laser-crystal interaction, a collaboration with researchers from the LMOPS laboratory.

To ease the implementation of this type of application from physics, a parXXL interface with a Mathematica PDE solver is under development (together with LMOPS). Our goal is to automatize the translation of a Mathematica code for PDE solving into a distributed large scale cellular computation. It will allow to speedup many research steps, avoiding long and tedious code translations.

Acknowledgments

Part of this research has been made possible via a visiting grant for Amelia De Vivo by the French programme ACI ARGE, and another part is supported by Region Lorraine. The experimental facet of this research is part of the Grid-eXplorer research initiative and relies on this experimental platform.

References

1. Valiant, L.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
2. Morin, C., Gallard, P., Lottiaux, R., Vallée, G.: Towards an efficient single system image cluster operating system. Future Generation Computer Systems (2004)
3. Norman, M., Henderson, J., Main, I., Wallace, D.: The use of the CAPE environment in the simulation of rock fracturing. Concurrency: Practice and Experience 3, 687–698 (1991)
4. Talia, D.: Solving problems on parallel computers by cellular programming. Proc. of the 3rd Int. Workshop on Bio-Inspired Solutions to Parallel Processing Problems BioSP3-IPDPS, LNCS, Springer-Verlag (2000) 595–603 Cancun, Mexico.
5. Baduel, L., et al.: Programming, Composing, Deploying for the Grid. In: Grid Computing: Software Environments and Tools (ch. 9), Springer, Heidelberg (2006)
6. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces Principles, Patterns, and Practive. Pearson Education (1999)
7. Gebremedhin, A., Guérin Lassous, I., Gustedt, J., Telle, J.: PRO: a model for parallel resource-optimal computation. In: 16th Annual International Symposium on High Performance Computing Systems and Applications, pp. 106–113 (2002)
8. Essaïdi, M., Gustedt, J.: An experimental validation of the PRO model for parallel and distributed computation. In: 14th Euromicro Conference on Parallel, Distributed and Network based Processing, pp. 449–456 (2006)
9. Ménard, O., Vialle, S., Frezza-Buet, H.: Making cortically-inspired sensorimotor control realistic for robotics: Design of an extended parallel cellular programming models. In: International Conference on Advances in Intelligent Systems - Theory and Applications (2004)

Performance Analysis of Two Parallel Game-Tree Search Applications

Yurong Chen[1], Ying Tan[1], Yimin Zhang[1], and Carole Dulong[2]

[1] Intel China Research Center,
8/F, Raycom Infotech Park A, No.2, Kexueyuan South Road,
Zhong Guan Cun, Haidian District, Beijing 100080, China
yurong.chen@intel.com
[2] Microprocessor Technology Lab, Intel Corporation, Santa Clara, CA, USA

Abstract. Game-tree search plays an important role in the field of artificial intelligence. In this paper we analyze scalability performance of two parallel game-tree search applications in chess on two shared-memory multiprocessor systems. One is a recently-proposed Parallel Randomized Best-First Minimax search algorithm (PRBFM) in a chess-playing program, and the other is Crafty, a state-of-the-art alpha-beta-based chess-playing program. The analysis shows that the hash-table and dynamic tree splitting operations used in Crafty result in large scalability penalties while PRBFM prevents those penalties by using a fundamentally different search strategy. Our micro-architectural analysis also shows that PRBFM is memory-friendly while Crafty is latency-sensitive and both of them are not bandwidth bound. Although PRBFM is slower than Crafty in sequential performance, it will be much faster than Crafty on middle-scale multiprocessor systems due to its much better scalability. This makes the PRBFM a promising parallel game-tree search algorithm on future large-scale chip multiprocessor systems.

1 Introduction

Tree search is one of the fundamental problems to artificial intelligence (AI). In the past half a century, a number of application-independent search algorithms (branch and bound [1,2], the Minimax algorithm, alpha-beta pruning [3,4], etc.) have been proposed to improve the search efficiency in many real life applications. One important experimental hotbed of various search algorithms has been in the field of game playing, in which game-tree search is widely used to find the best moves for two-player games. Computer programs based on advanced search algorithms and general purpose processors have achieved great success in popular games such as checkers, Othello and chess.

Alpha-beta pruning has been the algorithm of choice for game-tree search for over forty years [3,4]. Its success is largely attributable to a variety of enhancements to the basic algorithm that can dramatically improve the search efficiency [4,5]. It is well-known that the standard method of alpha-beta pruning conflicts with parallelism. In other words, alpha-beta pruning can be parallelized, but not easily, and speedups are typically limited to a factor of six or so regardless of how many processors you have used [6,7]. The reason mainly comes from the occurrence of sequential bottlenecks in the control structure of the program.

In the past thirty years, a large amount of innovative parallel game-tree search algorithms have been proposed in literature as more and more multiprocessor systems become available. Although there are a large amount of parallel game-tree search algorithms, we can divide them into two categories: alpha-beta-based algorithms and algorithms based on other search paradigms [6]. Previous work is mostly focused on researching various search strategies and their specific implementations using a variety of languages on different systems.

In this work, we focus on performance analysis and comparison of two parallel game-tree search applications in chess on two Intel Xeon Shared-memory MultiProcessor (SMP) systems. One is a recently-proposed Parallel Randomized Best-First Minimax search algorithm (PRBFM) [8] in a chess-playing program which is also denoted as PRBFM for convenience. The other is Crafty [10], a state-of-the-art alpha-beta-based chess-playing program. A detailed performance characterization of these two programs is also conducted with some Intel performance analysis tools such as MPI Trace Collector/Analyzer, Thread Profiler and Vtune Analyzer.

The rest of this paper is organized as follows. Sect. 2 gives a brief overview of PRBFM and Crafty. Sect. 3 introduces our experimental environment and compares their serial performance. Sect. 4 shows their scalability performance on our two platforms and presents our analysis results. Sect. 5 performs a detailed micro-architectural analysis and comparison of these two applications. Finally, we conclude in Sect. 6.

2 Overview of PRBFM and Crafty

The PRBFM algorithm is a randomized version of Korf and Chickering's best-first search [9], which is not alpha-beta-based. This algorithm was implemented in a chess program using Message Passing Interface (MPI) in a master-slave model as shown in Fig.1. One processor keeps the game tree in its local memory, and in charge of selecting leaves and backing up the scores. Whenever it selects a leaf, it sends a message to an idle slave processor. By calling Crafty's board evaluation code, the slave processor computes the possible moves from the leaves and their heuristic evaluations. Then it sends this information back to the master, which updates the tree. While the master waits for the reply of the slave, it assigns work to other idle slaves. In this model, leaf selections and score updates are completely serialized, but tree updates involve no communication. The serialization does not hurt performance in practice, as long as leaf expansion is significantly more expensive than selecting leaves and backing up scores. The maximum number of processors that this implementation can use depends on the cost of leaf evaluation relative to the cost of tree updates and leaf selections.

Crafty [10] is probably the strongest non-commercial chess program and Crafty19.1 running on a SMP system even got the second place in the 12th world computer speed chess championship. It uses an alpha-beta-based parallel game-tree search framework (as shown in Fig.2), and more specifically, a similar Dynamic Tree Splitting (DTS) algorithm [11] implemented with Pthreads. This algorithm can be viewed as a peer-to-peer approach rather than a master-slave approach. Its basic idea is to "split" the "move list nodes" (siblings) into several blocks, and whenever detect that one or more threads are in their idle loop, invoke this thread to begin a new alpha-beta search from the current node through copying the search state space for each thread working at this node, then sending everyone off to "SearchSMP" function to search this node's block siblings.

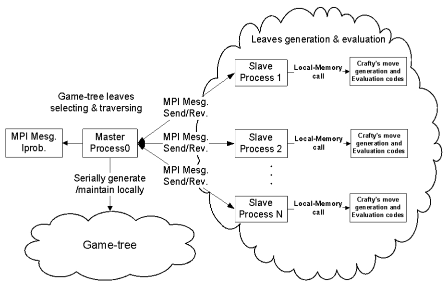

Leaves generation & evaluation
Slave Process 1
Local-Memory call
Crafty's move generation and Evaluation codes
Game-tree leaves selecting & traversing
MPI Mesg. Send/Rev.
MPI Mesg. Iprob.
Master Process0
MPI Mesg. Send/Rev.
Slave Process 2
Local-Memory call
Crafty's move generation and Evaluation codes
Serially generate /maintain locally
MPI Mesg. Send/Rev.
Slave Process N
Local-Memory call
Crafty's move generation and Evaluation codes
Game-tree

Fig. 1. The framework of PRBFM

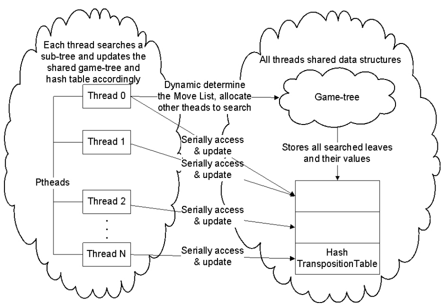

Each thread searches a sub-tree and updates the shared game-tree and hash table accordingly
All threads shared data structures
Thread 0
Dynamic determine the Move List, allocate other theads to search
Game-tree
Thread 1
Serially access & update
Serially access & update
Stores all searched leaves and their values
Pthreads
Thread 2
Serially access & update
Thread N
Serially access & update
Hash TranspositionTable

Fig. 2. The framework of Crafty

As a powerful chess-playing program, Crafty uses many enhancements, including transposition tables, killer-moves, null-move heuristics etc. to the search algorithm [4,5]. Among these enhancements, the transposition table (actually a large hash table) undoubtedly plays a very important role. The primary purpose of the table is to enable recognition of move transpositions that have lead to position (sub-tree) that has already been completely examined. In such a case there is no need to search again. This technique has indeed great enhancement to single thread chess program, however, for multithreaded program to maintain this table requires great synchronization costs.

3 Experimental Environment and Serial Performance

Our analysis work is based on two Intel Xeon based shared-memory multiprocessor platforms. The first is a 4-way SMP system referred to as QP(Quad-Processors)-machine (Xeon MP 2.8GHz, L3 2MB, FSB 400MHz), and the second is a Unisys-es7000 system which consists of 16 processors (Xeon MP 3.0GHz, L3 4MB, FSB speed 400MHz, L4 32MB shared by each 4 processors, 4x4 Crossbar interconnection). The key characteristics of these two systems show the 16-way system has larger memory access latency because the memory transactions need to go through L4 cache before visiting the memory. For software configuration, on both platforms, we use Intel C/C++ Compiler Version 8.1 to compile the two

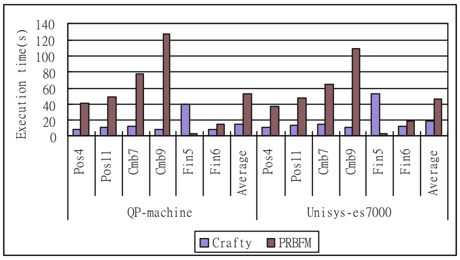


Fig. 3. The sequential runtime of Crafty and PRBFM on the two systems

programs under Linux Kernel 2.4.20smp with full compiler optimization (-tpp7 -O3 -g).

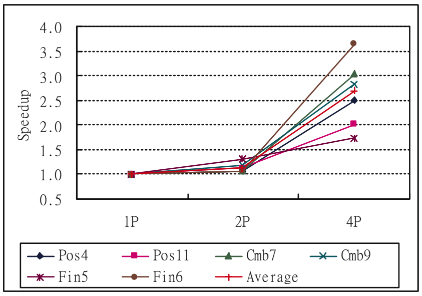
The original PRBFM is based on Crafty17.9, and it is about 7 times slower than Crafty17.9 in serial [8]. We modified the program based on Crafty19.9 [10], and obtained a new version of PRBFM. For Crafty, we set the hashtable and hashpawntable both with maximize size 768MB. Besides, for both programs, we turned off the opening book and endgame database search, since our test positions are not included in those databases.

In this study, six positions chosen from the Louguet Chess Test II, version 1.21[1] are used as test datasets (their winning moves are known). Fig.3 shows the sequential runtime of Crafty (Version 19.9) and PRBFM for these positions on both systems. The runtime for each position is an average of 20 runs. We can see that for these positions PRBFM runs on average 2.4 to 3 times slower than Crafty although it had achieved more than 3X speedup compared to the original version. The main reason is that the node evaluation module of PRBFM contains some redundant work in order to get more accurate node evaluation values by calling Crafty code to do a shallow search [8]. Although Crafty performs better in sequential mode, PRBFM is more scalable and has better time-to-solution on more processors as shown in the following section.

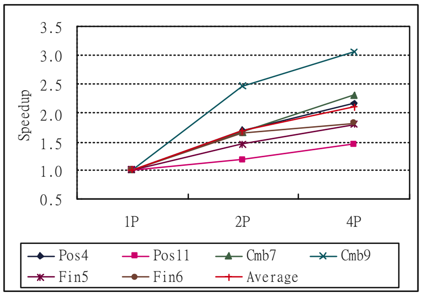
4 Scalability Performance Analysis

Fig.4 shows the speedup curves of PRBFM and Crafty on the two SMPs respectively. From this figure, we can see that PRBFM achieves on average 2.8X speedup while Crafty achieves 2.1X speedup on the 4-way system with 4 processors. Fig.4(c) indidates that PRBFM reaches an average of 14X speedup on the 16-way system with 16 processors. For some datasets like Cmb7 and Cmb9, it even achieves super-linear speedup on 12 or 16 processors. This is because sometimes the parallel search finds the solution that enables the program to stop before completing all the work of the serial computation. In other words, for those positions PRBFM searched much less nodes in parallel to find the solution than the serial version did. In contrast, Crafty gets an overall decreasing speedup

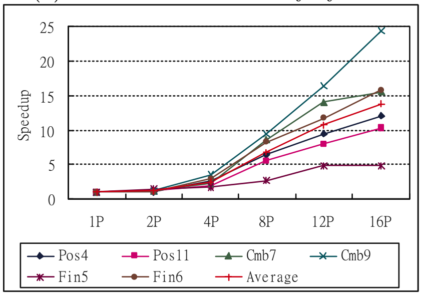
[1] http://www.icdchess.com/wccr/teting/LCTII/lct2faq.html



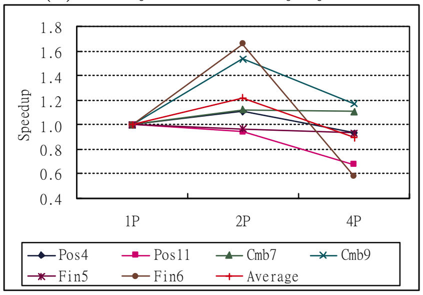
(a) PRBFM on the 4-way system



(b) Crafty on the 4-way system



(c) PRBFM on the 16-way system



(d) Crafty on the 16-way system

Fig. 4. Scalability of PRBFM and Crafty on the two systems

on the 16-way system. It only gets an average 0.9X speedup on 4 processors as shown in Fig.4(d). In particular, Fin6 gets the highest speedup of 1.7X on 2 processors and also gets the lowest speedup of only 0.6X on 4 processors.

In order to find out the reason of the poor scalability of Crafty, we profiled it with the Intel Vtune Analyzer. Fig.5 shows the serial region and parallel region of Crafty obtained by the Thread Profiler on the 4-way system. From this figure, we can see that the serial region of Crafty even costs more than 30% of execution time in most cases on the 4-way system.

Fig.6 shows the execution time breakdown of Crafty on the two hardware platforms. From this figure, we can see that the search control operations in Crafty constantly account for 4% to 6% of the total execution time regardless of the number of processors and the hardware platform we use. Similarly, the evaluation execution time does not vary dramatically with the number of processors, and it takes usually 20% to 30% of the total execution time. The move generation is mostly bitboard operations with integer operations like AND, OR and SHIFT. With the increasing number of processors, its cost decreases from about 40% for one processor to about 25% for 4 processors on the 4-way system, and from about 30% for one processor to about 15% for 4 processors on the 16-way system.

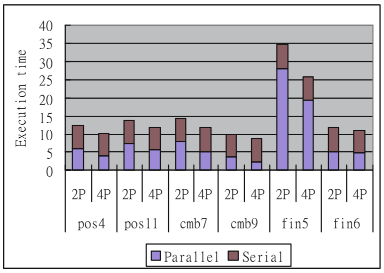


Fig. 5. Parallel and serial regions of Crafty on the 4-way system

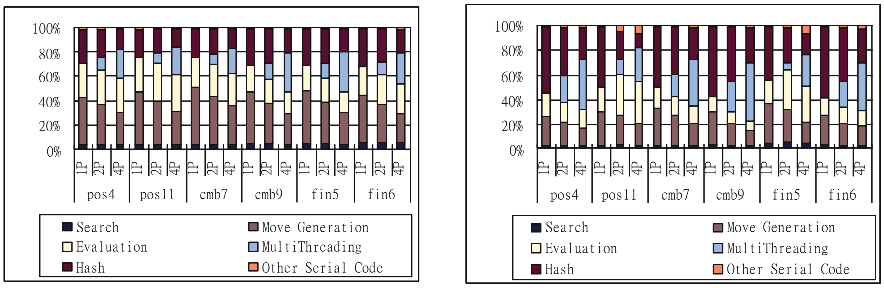
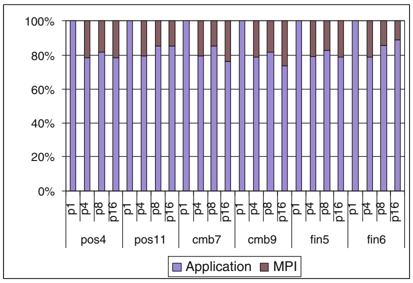


Fig. 6. Execution time breakdown of Crafty on the 4-way system (left) and the 16-way system (right)

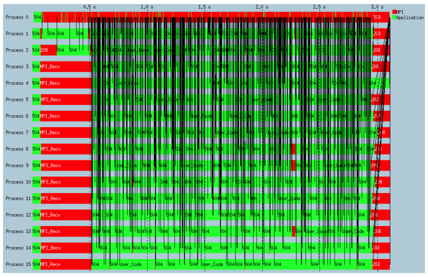
It is surprising to see that hash operations take so much execution time, which is about 20% on QP-machine and about 30% on the 16-way system with 4 processors, respectively. This cost on the 16-way system is much higher than that on QP-machine. This is because hash operations are mostly memory access operations, and the hash table size is much bigger than the total cache size. So the 16-way system with an added L4 cache has much more penalties for such operations. Furthermore, we can see that the dynamic tree splitting makes a lot of multithreading overheads. With up to 4 processors, it even reaches about 20% to 30% on QP-machine and about 25% to 50% on the 16-way system.

From above analysis, we can see that the larger serial region and multithreading overhead prevent Crafty from scaling well on both systems. Furthermore, the longer memory access latency on the 16-way system further increases those penalties suffered by Crafty, which leads to its much poorer scalability on that system.

By comparison, we also got the MPI trace data from the Intel MPI Trace Collector/Analyzer for PRBFM. The data shows that the MPI overhead of PRBFM is only about 20% of whole runtime on both systems even with 16 processors on the Unisys machine (see Fig.7(a)). This overhead mainly comes from the communication at the beginning and end stages as shown in Fig.7(b). Note that



(a) Percentage of MPI and application



(b) Summary timeline

Fig. 7. MPI traces of PRBFM on the 16-way SMP system

PRBFM uses speculation to achieve parallelism like all parallel game-search algorithms, and from the summary timeline figure, we can see that it performs fine-grained and high-quality speculations which results in a high level of parallelism with little additional work, especially, in the middle stage. It is the fundamentally different search strategy that makes the program significantly reduce those penalties encountered by Crafty.

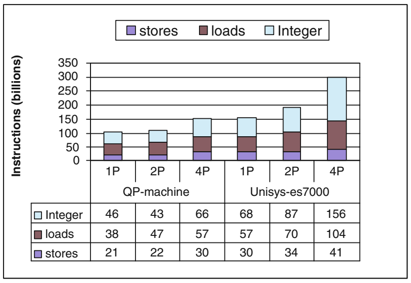
5 Micro-architectural Analysis

5.1 Instruction Type Distribution

Fig.8 shows the average instructions executed (in billions) and breakdown for PRBFM and Crafty for all data sets. We split the instructions into integer, loads and stores. There are no floating-point instructions in both programs. We can see that both programs are memory access dominant programs, in most cases the loads and stores instructions together consist of about 50% of the total instructions and the number of stores is about half of the number of loads. These similarities are understandable, because PRBFM frequently calls Crafty's move generation and evaluation code to generate its own game tree.

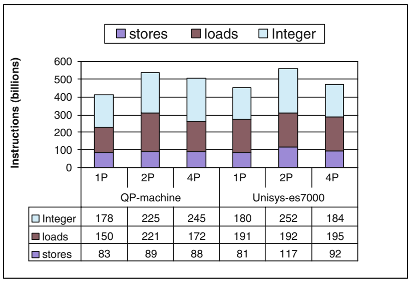
As shown in Fig.3 and Fig.8(a), Crafty is more sensitive to memory hierarchy than to processor speed. When running on the 16-way system with a single processor, the average runtime is 25% slower and the instructions executed are 1.5 times more than that on QP-machine. When the processor number increases, the performance gap between two platforms becomes larger: with 4 processors, the average runtime is 1.9 times slower and the instructions executed are 2 times more than that on QP-machine. This is because Crafty requires more dynamic instructions for running the spin loops to solve the test positions while suffering more cache-misses or more synchronization. Further analysis confirms that the one hotspot spin-loop spends 28% of the Unisys dynamic instructions compared to 12% on QP-machine.

Unlike Crafty, as shown in Fig.8(b), when the number of processors increases, the total instructions executed by PRBFM are not always increasing either.



	1P	2P	4P	1P	2P	4P
	QP-machine			Unisys-es7000		
□ Integer	46	43	66	68	87	156
■ loads	38	47	57	57	70	104
■ stores	21	22	30	30	34	41

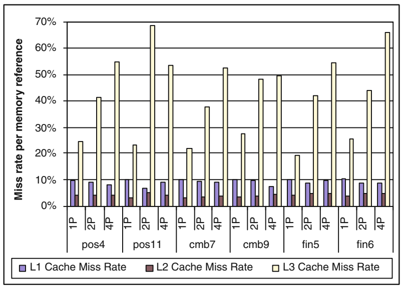
(a) Crafty



	1P	2P	4P	1P	2P	4P
	QP-machine			Unisys-es7000		
□ Integer	178	225	245	180	252	184
■ loads	150	221	172	191	192	195
■ stores	83	89	88	81	117	92

(b) PRBFM

Fig. 8. Instructions executed and breakdown of Crafty and PRBFM on the two SMPs



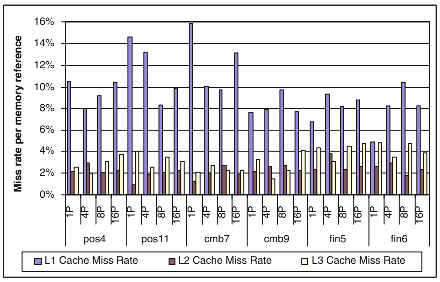


Fig. 9. Cache miss rate of Crafty (left) and PRBFM (right) on the 16-way SMP system

It is interesting that the number of total executed instructions varies little or even decreases with the increasing number of processors. This means when using more processors, the PRBFM could solve the same problem with equal or less instructions than by using fewer processors, which shows the good scalability potential of PRBFM.

5.2 Cache Miss Behavior

Fig.9 shows the data miss rates per memory reference for the 3 levels of caches on the Unisys machine for Crafty and PRBFM respectively. For all the datasets, L1 cache miss rates of Crafty are around 10%, the L2 cache miss rates are around 5%, and L3 cache miss rates increase from around 25% to more than 50% with the increasing number of processors. Comparatively, L1 cache miss rates of PRBFM are around 10%, the L2 cache miss rates are around 3%, and L3 cache miss rates are around 3%. Through a simple calculation, we can find that PRBFM has about 25 times fewer global cache misses than Crafty. The functional breakdown data shows the higher cache miss rate of Crafty mainly comes from the hash operations.

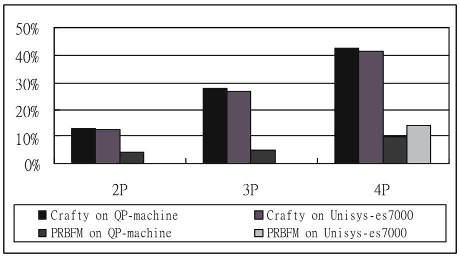


Fig. 10. The average coherence ratios of Crafty and PRBFM on the two systems

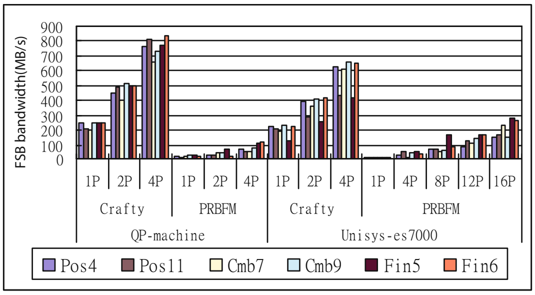


Fig. 11. The FSB bandwidth requirement of Crafty and PRBFM on the two systems

The coherence ratios, percentage of the coherence misses in the whole cache misses, are also listed in Fig.10. As expected PRBFM has 4 to 5 times lower coherence ratios than Crafty. The lower coherence ratios mainly arise from the randomization nature of the algorithm and thus are relatively independent with the detailed implementation. The less coherence ratios make PRBFM more scalable.

5.3 FSB Bandwidth Requirement

Fig.11 shows the memory bandwidth usage of Crafty and PRBFM on our two platforms. As expected, the FSB utilization rates of both applications increase with the number of processors and the FSB bandwidth requirement of Crafty is much larger than PRBFM. However, the bus remains under-utilized on both platforms. The largest bandwidth usage of Crafty is about 800MB/s on the QP-machine with 4 processors and this value is still much less than the peak FSB bandwidth (2.1GB/s). For PRBFM, only less than 9% of the peak bandwidth (3.2GB/s) is used on the Unisys machine with 16 processors. This observation shows that both applications are not bandwidth-limited.

6 Conclusions

This paper characterizes and compares two parallel chess game-tree search applications, PRBFM and Crafty, on two Intel Xeon shared-memory multiprocessor systems. Our data shows that the hash-table and dynamic tree splitting operations used in Crafty lead to large scalability penalties. They totally consume 35% to 50% and 40% to 75% of the runtime on our two systems with 4 processors respectively. By comparison, similar functions of PRBFM only cost about 20% of the runtime, and it is the fundamentally different search strategy that prevents those scalability penalties.

Our micro-architectural analysis also shows that PRBFM is memory-friendly while Crafty is latency-sensitive and both of them are not bandwidth bound. Although PRBFM is on average 2.4 to 3 times slower than Crafty in sequential performance, it is much faster than Crafty on middle-scale multiprocessor systems due to its much better scalability. This makes the PRBFM a promising parallel game-tree search algorithm on future large-scale chip multiprocessor systems.

Acknowledgement. The authors would like to thank Shoham Y. and Toledo S. for providing the source code of the original PRBFM. Thanks to Hyatt R. for making Crafty publicly available and for allowing us to use it in this research.

References

1. Baravykaitė, M., Čiegis, R., Žilinskas, J.: Template Realization of the Generalized Branch and Bound Algorithm. Mathematical Modelling and Analysis 10(3), 217–236 (2005)
2. Le, C.B., Roucairol, C.: BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms. Technical Reports, No.95/16, Universiti de Versailles Saint Quentin (1995)
3. Knuth, D., Moore, R.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6, 293–326 (1975)
4. MarsLand, T.A.: A Review of Game-Tree Pruning. Int. Comp. Chess Assoc. Journal 9(1), 3–19 (1986)
5. Schaeffer, J., Plaat, A.: New Advances in Alpha-Beta Searching. In: Proc. of ACM 24th annual conference on Computer Science, pp. 124–130. ACM Press, New York (1996)
6. Brockington, M.G.: A Taxonomy of Parallel Game-Tree Search Algorithms. Int. Comp. Chess Assoc. Journal 19(3), 162–175 (1996)
7. Joerg, C.F., Kuszmaul, B.C.: The *Socrates Massively Parallel Chess Program. In: Bhatt S.N. (ed.) Parallel Mathematics and Theoretical Computer Science, vol. 30, pp. 117–140. American Mathematical Society, Providence, RI (1996)
8. Shoham, Y., Toledo, S.: Parallel Randomized Best-Fisrt MiniMax Search. Artificial Intelligence 137, 165–196 (2002)
9. Korf, R.E., Chickering, D.M.: Best-First Minimax Search. Artificial Intelligence 6, 293–326 (1975)
10. Hyatt, R.M.: Crafty, Computer program, `ftp://ftp.cis.uab.edu/pub/hyatt`
11. Hyatt, R.M.: The Dynamic Tree-Splitting Parallel Search Algorithm. Int. Comp. Chess Assoc. Journal 20(1), 3–19 (1997)

Implementation of a Black-Box Global Optimization Algorithm with a Parallel Branch and Bound Template

Raimondas Čiegis and Milda Baravykaitė

Vilnius Gediminas Technical University,
Saulėtekio av. 11, LT-10223 Vilnius, Lithuania
{rc, mmb}@fm.vtu.lt

Abstract. A new derivative-free global optimization algorithm is proposed for solving nonlinear global optimization problems. It is based on the Branch and Bound (BnB) algorithm. BnB is a general algorithm to solve optimization problems. Its implementation is done by using the developed template library of BnB algorithms. The robustness of the new algorithm is demonstrated by solving a selection of test problems. We present a short description of our template implementation of the BnB algorithm. A paradigm of domain decomposition (data parallelization) is used to construct a parallel BnB algorithm. MPI is used for underlying communications. To obtain a better load balancing, the BnB template has a load balancing module that allows the redistribution of a search space among the processors at a run time. A parallel version of the user's algorithm is obtained automatically from a sequential algorithm.

1 Problem Formulation

Many problems in engineering, physics, economic and other fields may be formulated as optimization problems, where the minimum/maximum value of an objective function should be found. Branch and bound (BnB) is a general technique to solve optimization problems. It can be used in many optimization algorithms, for example to solve combinatorial optimization or covering global optimization problems. Its general structure can be implemented as an algorithm template that will simplify implementation of specific BnB algorithms to solve a particular problem. Similar template ideas applied for a parallel algorithm relieve users from doing the actual parallel programming.

Consider a minimization problem, formulated as

$$f^* = \min_{X \in D} f(X), \tag{1}$$

where $f(X)$ is an objective function, X are decision variables, and $D \subset R^n$ is a search space. Besides of the minimum f^*, one or all minimizers $X^* : f(X^*) = f^*$ should be found.

The main idea of the BnB algorithm is to detect the subspaces not containing the global minimizers and discard them from the further search. The initial search space D is subsequently divided into smaller subspaces D_i. Then each subspace is evaluated trying to find out if it can contain the optimal solution. For this purpose a lower bound of the objective function $LB(D_i)$ is calculated over the subspace and compared with the upper bound $UB(D)$ for the minimum value. If $LB(D_i) \geq UB(D)$, then the subspace D_i cannot contain the global minimizer and therefore it is rejected from the further search. Otherwise it is inserted into the list of unexplored subspaces. The algorithm terminates when there are no subspaces in the list.

Unlike the data parallel applications (e.g., algorithms for a solution of partial differential equations) optimization problems are characterized by an unpredictably varying unstructured search space [20]. This property produces additional difficulties for creation of parallel BnB algorithms: a) the change of space search order with respect to the sequential one, b) a load unbalance of processors, c) costs of additional communications. We note that any BnB algorithm depends on few rules, and the optimal selection of these rules is strongly problem dependent.

For many engineering applications only values of the objective function $f(X)$ can be computed and we do not have information on the derivatives of f. In this paper, we present a new derivative-free algorithm for a solution of nonlinear global optimization problems. The objective function is computed by a black-box algorithm. This BnB algorithm is implemented by using a new template library of BnB algorithms. The main goal of this tool is to present a flexible and extendable template library which enables users to make experiments with different strategies of subspace selection, branch and bound rules and techniques for computation of lower bounds of the objective function $LB(D_i)$. We also present a short description of the developed template library. A parallel version of user's algorithm is obtained automatically from the sequential algorithm. Some results of numerical experiments illustrate the efficiency of the template library.

The rest of the paper is organized as follows. A generalized BnB algorithm is described in Section 2. A new black-box global optimization algorithm is presented and investigated in Section 3. In Section 4 a template based implementation of the BnB algorithm is considered. Some final conclusions are done in Section 5.

2 A Generalized BnB Algorithm

The branch and bound technique is used for managing the list of sub-regions and the process of discarding and partitioning. The general branch and bound algorithm is shown in Figure 1, where L denotes a candidate set, S is the solution, $UB(D_i)$ and $LB(D_i)$ denote upper and lower bounds for the minimum value of the objective function over sub-space D_i.

A selection rule, a lower bound computation rule and a branch rule define the given BnB algorithm. There are many different strategies for a selection order of subproblems. The most popular strategies are defined as the best first search, the last first search and the breadth first search rules. A bound rule is

```
BnBAlgorithm ()
begin
  (1)  Cover solution space $D$ by $L = \{L_j | D \subseteq \cup_{j=1}^{m} L_j\}$ using covering rule.
  (2)  $S = \emptyset$,   $UB(D) = \infty$.
  (3)  while  (subspace list is not empty $L \neq \emptyset$) do
  (4)          Choose $I \in L$ using selection rule, exclude $I$ from $L$.
  (5)          if ( $LB(I) < UB(D) + \epsilon$) then
  (6)             Branch $I$ into $p$ subspaces $I_j$ using branching rule.
  (7)             for all  $(I_j, j = 1, 2, \dots, p)$ do
  (8)                     Find $UB(I_j \bigcap D)$ and $LB(I_j)$ using bounding rules.
  (9)                     $UB(D) = \min(UB(D), UB(I_j \bigcap D))$.
  (10)                    if  $(LB(I_j) < UB(D) + \epsilon)$ then
  (11)                        if  ( $I_j$ is a possible solution) then $S = I_j$.
  (12)                        else $L = L \cup \{I_j\}$.
  (13)                        end if
  (14)                    end if
  (15)            end for
  (16)         end if
  (17) end while
end  BnBAlgorithm
```

Fig. 1. General BnB algorithm

problem dependent, but some general techniques are known for specific classes of problems, e.g., for Lipschitz functions.

Parallel BnB algorithms

Three main steps are performed during development of any parallel algorithm: partitioning, mapping, communication [8]. First we need to distribute the initial search space among the processors. In our BnB template a paradigm of domain decomposition (data parallelization) is used to construct a parallel algorithm. The initial search space is divided into several large subspaces that are mapped to processors and each processor performs `BBAlgorithm` independently and asynchronously. The user should decide how many subspaces are generated. The number of subspaces can coincide with or exceed the number of processors p, the decision depends on a priori knowledge of the computational complexity of subspaces. A random distribution of larger number of subspaces can improve the global load balance among processors.

A subspace is eliminated from the further search by comparing the lower bound $LB(D_i)$ for the objective function over the subspace with the upper bound $UB(D)$. The best currently found value of the objective function is used for the upper bound. In a simplest version of the parallel algorithm, processors know only local values of the objective function. This can result in a slower subspace elimination. In a modified version of the algorithm processors can share $UB(D)$. When a new value of the upper bound is found, it is broadcasted asynchronously to the other processors.

A load balancing module for BnB algorithms

To obtain a better load balancing, BnB template uses the load balancing module that allows the redistribution of search spaces among the processors at run time. The objective of the data redistribution strategies is to ensure that there is no idle processors while others are heavily loaded, i.e to guarantee a useful work for all processors, but not to obtain the equal workload between processors [20].

The balancing module has few basic methods needed for the load balancing. A version of the diffusion load balancing algorithm is implemented as a default method [20]. The balancing process is initialized by a receiver processor. The measure of work-load is based on the number of subproblems belonging to the local list. More accurate estimates are obtained if estimates of the complexity of subproblems are known apriori. This information should be defined by a user of the BnB template. The full step of load balancing consists of the exchange of information among neighbour-processors on their work-load, the selection of partners and the redistribution of subproblems among neighbour-processors. The termination of the parallel BnB algorithm requires special protocols if the load balancing process was started. The implemented balancing module can be extended with other balancing algorithms as well.

3 A Black-Box Global Optimization Algorithm

For many engineering applications only values of the objective function $f(X)$ can be computed and we do not have information on the derivatives of f or on the Lipschitz constant of this function. Thus the objective function is computed by a black-box algorithm or code and the gradient computation is unavailable. The target applications are simulation-based optimization problems characterized by a small number of variables (i.e., $n < 20$) and by expensive objective function evaluations (e.g., they can require solution of a system of nonlinear PDEs [3]). Thus estimation of derivatives by finite differences may be prohibitively costly. A good review on derivative-free methods is given in [5].

Black-box optimization algorithms are derivative-free, only function values are required for the optimization. Parallel versions of these algorithms can greatly reduce the total solution time. A well-known library implementing derivative-free direct search algorithms is APPSPACK [11]. Parallelism is achieved by assigning the individual function evaluations to different processors. The asynchronism enables a better load balancing.

We propose a new black-box global optimization algorithm, which is based on the BnB method. The algorithm is implemented by using the developed template of BnB algorithms, thus a parallel version of the algorithm is obtained automatically by running BnB code in parallel. Thus parallelism is achieved on a coarse grain level of the algorithm and a parallel version of the algorithm is obtained automatically by running the sequential BnB code concurrently. Our algorithm is only heuristic and the main novelty of it is introduced in the definition of the bounding rule. The remaining rules are taken from the general template of the BnB algorithm. Thus we present in detail only the outline of the bounding rule.

A bounding rule

In each sub-space D_i two sets of trial points are generated. The first set of $(2n+1)$ regular points cover the sub-space in quasioptimal way and the remaining M points are distributed randomly. We note that Sobol's sequence and the lattice rule can be used to distribute random points more uniformly.

Then a local search is done from all trial points by using the Simplex local optimization algorithm (it is a gradient-descent type method, but its realization is derivative-free). The following three cases are considered:

1. If no local minimum points are obtained in D_i then this sub-space is eliminated from the search list L.
2. If exactly one local minimum point is obtained, the information on $UB(D)$ is updated. The sub-space is eliminated from the search list.
3. If two or more local minimum points exist in the sub-space, then a new $LB(D_i)$ estimate is computed

$$LB(D_i) = \min_i LM_i - C(\max_i LM_i - \min_i LM_i).$$

In order to increase the robustness of the algorithm up to K local minimizers in each sub-space D_i are saved for a future usage. All of them are included into the new list of local minimizers. We note that this list is updated at each iteration.

Many black-box optimization algorithms suffer from a serious drawback, that after rapid initial improvement of an initial approximation, the following computations give no further improvement of the solution and the algorithm is stalling. This property depends mainly on the rule defining when a sub-region can be excluded from the list of promising sub-spaces. In most real-world applications it is sufficient to find a good approximation of the global minimizer (using only modest resources of the CPU time). Thus we add to the algorithm two additional rules which define cases when a sub-space is eliminated from the search list L.

1. The number of sub-divisions of each initial sub-space is restricted to NS.
2. If after L subsequent divisions the value of a best known local minimum $UB(D_i)$ is not updated, then this sub-space is excluded from the search list.

Such rules guarantee that expansive computations do not concentrate too long in some particular part of the domain D, and the whole region is tested during a reasonable time of computations.

Test functions and computational results

To assess the robustness of the new algorithm we have solved a selection of problems from [14]. The main characteristics of these problems are given in Table 1.

Up to 16 processors were used to solve each problem. For any number of processors the BnB algorithm converged to the optimal solution for problems 1-7. In the case of Griewank's problem the accuracy of the computed solution

Table 1. The dimensions and the numbers of local and global minimizers of test functions

Function	n	No. of local minim.	No. of global minim.
1. Rosenbbrock	2	1	1
2. McCormick	2	1	1
3. Box Betts	3	1	1
4. Shekel5	4	5	1
5. Levy4	4	71000	1
6. Paviani	10	1	1
7. Generalized Rosenbrock	15	1	1
8. Griewank	10	1000	1

depended on the specified domain D. If we take $D = [-5, 7] \times \cdots \times [-5, 7]$, then the optimal global minimizer is obtained. For $D = [-50, 70] \times \cdots \times [-50, 70]$, we have computed only approximation of the exact minimizer. In solving this test problem a similar behaviour is typical for most derivative-free black-box global optimization algorithms. Since Griewank's problem is multidimensional and it has a very large number of local minimizers, the computed $LB(D_i)$ bounds were not very accurate and subproblems were eliminated from the search list L mainly according to the two additional rules, given above. We note that an improvement of the accuracy still was obtained by increasing the number of possible divisions and the number M of randomly distributed initial approximations.

4 A Template of BnB Algorithms

In this section, we give a very brief description of a new template of BnB algorithms. The idea of the template programming is to implement a general structure of the algorithm that can be later used to solve different problems. Here we use skeleton-like templates, not data templates (or classes parameterized with data types like in STL). All general features of the BnB algorithm and its interaction with the particular problem are implemented in the template. The particular features related to the problem must be defined by the template user.

We mention some popular examples of parallel templates used for implementation of various algorithms: Master–slaves template [16], combinatorial optimization library of software skeletons Mallba [1], CODE [18]. A template based programming is very useful in parallel programming. It was proposed by M. Cole in his PhD thesis [4], see also [6,12,15]. Any template of a parallel algorithm must fully or partially specify the main features of a parallel algorithm: partitioning, communication, agglomeration and mapping. From the user's point of view, all or nearly all coding should be sequential and almost all the parallel aspects should be provided by the tool. A parallel template is a re-usable, application-independent encapsulation of a commonly used parallel computing pattern. It is implemented as a re-usable code-template for quick and reliable development

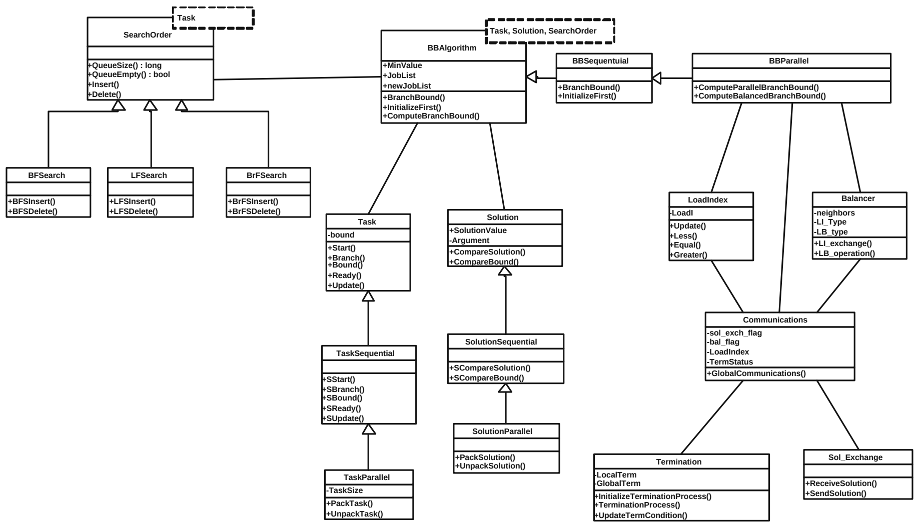


Fig. 2. BnB class scheme

of parallel applications. We mention examples of BnB parallelization tools BOB [13] , PICO [7], PPBB [19], PUBB [17].

A new BnB algorithm template is implemented by using C++ object oriented language. Thus it is a portable and easily extendable tool. Our main goal was to build a parallel template, which has a simple and flexible structure and enables users to modify the template according their needs. The experience of usage of BnB algorithms proves that such modifications are frequently required when real world problems are solved. There is no good recipes for definition of a general BnB strategy when a black-box type global optimization problem is solved.

Comparing our tool with the other BnB templates, we note that practically all tools define a similar basic functionality. The difference is mainly in the implementation technology and in a set of methods, which are implemented for the main rules of BnB algorithms. We use the technology of template expressions and define only very few basic branch and bound methods. It is assumed that in most cases specific combinations of these methods will be needed. Therefore our aim was to develop a flexible and portable structure of the tool, which can be easily extended according specific needs of users.

The C++ class scheme of BnB algorithm template is presented in Figure 2. The template implements all main parts of sequential and parallel BnB algorithms. The algorithm is performed using `Task`, `Solution` and `SearchOrder` instances. An implementation of the `BBAlgorithm` is presented in the template, and users can extend this class with other useful methods and algorithms, such as simulated annealing, genetic programming, the α–β search algorithm and the other popular algorithms used to solve game problems.

`SearchOrder` defines the strategies how to select a next subspace from the list of subspaces for subsequent partitioning. The most popular strategies such as the best first search, last first search and breadth first search are already implemented as methods and they are ready for application. Class `Task` defines the problem to be solved. It should have the basic BnB algorithm methods: `Initialize`, `Branch`, `Bound`. Some often used `Branch` methods are already implemented in the template. Standard `Bound` calculation methods (e.g., for Lipschitz functions) are included into the template. Class `Balancer` is used for parallel applications to balance the processor load. The data communication level is implemented using MPI and this level of the library is hidden from the user.

It is well-known that efficient algorithms for global optimization problems are problem dependent. An optimization of specific parameters of these algorithms is an essential task when real-world problems are solved. This conclusion is even more important for parallel BnB algorithms. Thus a template of the BnB algorithm is very useful for tuning specific parameters and rules of a general BnB algorithm.

A domain decomposition method is used to build parallel BnB algorithms in our template. There is a big difference between classical applications of the domain decomposition for numerical solution of PDEs and for solution of global optimization problems. Parallel optimization algorithms have an unpredictably varying unstructured search space [20]. Due to the domain decomposition the order of search can differ for parallel and sequential BnB algorithms even when the same subset selection and branching rules are used. Sub-spaces which were eliminated in the sequential algorithm can be explored in the parallel one. Thus it is possible that a total number of the sub-spaces searched in the parallel algorithm can be larger than in the sequential case.

Let us define the number of nodes in the generated search tree as a unit to measure the complexity of the BnB algorithm. We propose to estimate the growth of the number of sub-spaces in the parallel algorithm by using the *search overhead factor*

$$SOF = \frac{W_p}{W_0}\,,$$

where W_p is the number of sub-spaces processed in the parallel algorithm, and W_0 is the number of sub-spaces processed by the sequential algorithm. This parameter is problem dependent and it can be calculated only a posteriori, but it helps us to explain the obtained experimental results, when the complexity of sub-problems is very different and the graph of generated jobs changes nondeterministically depending on the number of processors [9]. In computational experiments we minimized the following four Lipschitz functions with given Lipschitz constants from the well-known test-suit [10]:

$$f_1(x_1, x_2) = 0.5x_1^2 - 9x_1 + 20 + 0.5x_2^2 - 9x_2 + 20,$$
$$f_2(x_1, x_2) = \frac{-1}{(x_1-4)^2 + (x_2-4)^2 + 0.7} - \frac{1}{(x_1-2.5)^2 + (x_2-3.8)^2 + 0.73},$$

$$f_3(x_1, x_2) = -\sin(2x_1 + 1) - 2\sin(3x_2 + 2),$$

$$f_4(x_1, x_2, x_3) = 100\Big(x_3 - \frac{1}{4}(x_1 + x_2)^2\Big)^2 + (1 - x_1)^2 + (1 - x_2)^2.$$

Experiments were performed on VGTU cluster *http://vilkas.vtu.lt.* It is a cluster of Pentium 4 processors which are connected by Gigabit Ethernet network (two Gigabit Smart Switch communicators). The ratio between computation and communication speeds is typical for clusters of PCs, thus the results will be even better for specialized supercomputers, such as IBM SP5.

In Table 2 values of the efficiency coefficient and SOF of the parallel BnB algorithm are given for different numbers of processors. The initial distribution of subspaces was done by using the breadth first search, and then the BnB algorithm with the best first search selection rule is used by all processes. It follows from the presented results that the decreased efficiency can be explained by increased value of the SOF coefficient. The unstructured search space varies unpredictably and it is impossible to guarantee that the efficiency of the parallel algorithm will be close to one. The load balancing helps to distribute surplus problems to free processors, but this step also enlarges the search overhead factor.

Table 2. The efficiency and SOF coefficients of the parallel BnB algorithm

	Problem	$p = 1$	$p = 2$	$p = 4$	$p = 6$	$p = 8$	$p = 10$
E_p	1	1	0.752	0.523	0.176	0.180	0.174
SOF_p	1	1	1.008	1.217	1.623	2.245	2.511
E_p	2	1	0.529	0.576	0.615	0.783	0.613
SOF_p	2	1	1.081	0.974	1.032	1.045	1.011
E_p	3	1	0.527	0.271	0.218	0.205	0.189
SOF_p	3	1	1.106	1.217	1.421	1.485	1.751
E_p	4	1	0.612	0.634	0.583	1.174	1.080
SOF_p	4	1	0.876	0.901	0.986	0.905	0.912

5 Conclusions

A new derivative-free algorithm is proposed for solution of nonlinear global optimization problems. It is based on the BnB method and its implementation is done by using the developed BnB algorithm template. The robustness of the new algorithm is demonstrated by solving a selection of test problems. A selection of optimal values of parameters C, K, NS of the proposed algorithm is not investigated here. This analysis will be presented in a separate paper.

The main advantage of using templates of BnB algorithms is due to a possibility to implement fastly new algorithms and to tune problem specific parameters. These advantages are even more important for parallel versions of BnB algorithms, since a space of parameters increases in this case. The proposed template realization of the BnB algorithm defines popular methods of the algorithm and gives a possibility for a user to extend the library with problem specific implementations of the selection and bounding rules. The object oriented implementation makes this task quite easy and straightforward for users.

A parallel version of the algorithm is obtained automatically from a sequential algorithm. The load balancing level of the BnB algorithm template implements a variant of the diffusion method. It can be extended by the other balancing methods tailored for the given application.

Acknowledgment. This work was supported by the Lithuanian State Science and Studies Foundation within the framework of the Eureka Project EUREKA E!3691 OPTCABLES .

The authors would like to thank the referees for their constructive criticism which helped to improve the clarity of this note.

References

1. Alba, E., Almeida, F., et al.: Mallba: A library of skeletons for combinatorical optimization. Technical report (2001)
2. Baravykaitė, M., Čiegis, R., Žilinskas, J.: Template realization of generalized branch and bound algorithm. Mathematical Modelling and Analysis 10(3), 217–236 (2005)
3. Baravykaitė, M., Belevičius, R., Čiegis, R.: One application of the parallelization tool of Master – Slave algorithms. Informatica 13(4), 393–404 (2002)
4. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman and MIT Press, Cambridge (1989)
5. Conn, A.R., Scheinberg, K., Toint, P.L.: Recent progress in unconstrained nonlinear optimization without derivatives. Mathematical programming 79, 397–414 (1997)
6. Dorta, I., Leon, C., Rodriquez, C.: Parallel branch and bound skeletons: message passing and shared memory implementations. In: Wyrzykowski, R., Dongarra, J.J., Paprzycki, M., Waśniewski, J. (eds.) Parallel Processing and Applied Mathematics. LNCS, vol. 3019, pp. 286–291. Springer, Heidelberg (2004)
7. Eckstein, J., Hart, W.E., Phillips, C.A., Pico: An object-oriented framework for parallel branch and bound. Technical Report 40-2000, Rutgers University, Piscataway, NY (2000)
8. Foster, I.: Designing and building parallel programs. Addison-Wesley, Reading (1995)
9. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison-Wesley, Reading (2003)
10. Hansen, P., Jaumard, B.: Lipschitz optimization. In: Handbook of Global Optimization. Nonconvex Optimization and Its Applications, vol. 2, pp. 404–493. Kluwer Academic Publishers, Dordrecht (1995)
11. Kolda, T.G.: Revisiting asynchronous parallel pattern search for nonlinear optimization. SIAM Journal on Optimization 16(2), 563–586 (2005)
12. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
13. Le Cun, B., Roucairol, C.: Bob: a unified platform for implementing branch-and-bound like algorithms. Technical Report 95/16 sep., Université de Versailles - Laboratoire PRiSM (1995)

14. Madsen, K., Žilinskas, J.: Testing of attraction based subdivision and interval methods for global optimization. IMM-REP-2000-04, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark (2000)
15. Preiss, B., Goswami, D., Singh, A.: From design patterns to parallel architecture skeletons. Journal of Parallel and Distributed Computing 62(4), 669–695 (2002)
16. Šablinskas, R.: Investigation of algorithms for distributed memory parallel computers. PhD thesis, Kaunas, Vytautas Magnus University (1999)
17. Shianno, Y., Fujier, T.: Pubb (parallelization utility for branch-and-bound algorithms). User manual. Technical Report, Version 1.0 (1999)
18. Singh, A., Szafron, D., Schaeffer, J.: Views on template-based parallel programming. In: CASCON 96 CDRom Proceedings,Toronto (October 1996)
19. Tschoke, S., Polzer, T.: Portable parallel branch-and-bound library ppbb-lib. User manual. Technical Report Version 2.0, Department of Computer Science, University of Paderborn (1996)
20. Xu, C., Lau, F.: Load balancing in parallel computers: theory and practice. Kluwer Academic Publishers, Dordrecht (1997)

Parallelization Techniques for Tabu Search

Jacek Dąbrowski

Gdańsk University of Technology,
ul. Gabriela Narutowicza 11/12, 80-952 Gdańsk, Poland
Jacek.Dabrowski@eti.pg.gda.pl

Abstract. *Tabu search* is a simple, yet powerful meta-heuristic based on local search. This paper presents current taxonomy of *parallel tabu search* algorithms and compares three parallel TS algorithms based on *Tabucol*, an algorithm for graph coloring. The experiments were performed on two different high performance architectures: an Itanium-based cluster and a shared-memory Sun server. The results are based on graphs available from the DIMACS benchmark suite.

1 Introduction

A coloring of a graph $G = (V, E)$, where V is the set of $n = |V|$ vertices and E is the set of edges, is a mapping $c : V \mapsto 1..k$, such that for each edge $\{u, v\} \in E$ we have $c(u) \neq c(v)$. Optimization version of GCP is stated as follows: given a graph G, find a coloring with the minimum number k of colors used. This number is referred to as $\chi(G)$, the *chromatic number of graph G*. The GCP is a well-known NP-hard combinatorial optimization problem.

Tabu search(TS) is a metaheuristic based on a local search approach: it is an iterative procedure that tries to improve current solution by exploring its *neighborhood* and the best candidate as the new current solution. The main concept in tabu search is a *tabu list* which keeps track of recently visited solutions to avoid cycling or getting trapped in a local optimum.

Parallel computation aims at solving problems quicker than sequential methods. In broad terms, this means either ”find a solution of a similar quality faster” or ”find a solution of a better quality in a comparable time”. Parallel implementations are often more robust than the sequential ones, providing better solutions consistently over diverse sets of problem instances.

This paper is organized as follows. Section 2 reviews parallel search models. Section 3 describes in detail three parallel algorithms used in this work. Section 4 presents experimental results from several coloring runs on standard benchmark graphs[1] as well as random graphs. Section 5 concludes the paper.

2 Taxonomy of Parallel Tabu Search

There are two main approaches to search parallelization. It can be done at a *low level*, where the parallel processing is used only to speed up tasks with high

[1] ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/

computation cost (e.g., neighborhood evaluation). This means that the behavior of the search is the same as that of a sequential algorithm, the difference is in wall-clock time. *High level* parallelism means simultaneous operation of multiple searches either independent or cooperating.

In [5] Crainic, Toulouse and Gendreau introduced a three-dimensional taxonomy for parallel tabu search methods. The first dimension, *Search Control Cardinality*, decides whether the search process is controlled by a single *master* processor (*1-control, 1C*) or each of p processors controls its own search (*p-control, pC*).

The second dimension, *Control and Communication Type*, is based on the communication scheme. There are four stages that define the operation mode (synchronous / asynchronous) and the type and amount of information shared. The first stage, *Rigid Synchronization* (*RS*), corresponds to simple synchronous communication with limited information exchange like simple task delegation. The second stage, *Knowledge Synchronization* (*KS*), increases the level of communication, allowing knowledge exchange. Within this stage the processors might exchange good solutions after a specified number of iterations.

Last two stages operate in asynchronous mode. It means that processes communicate after events (e.g., finding an improved solution) rather than at a specific stage of the algorithm or after predetermined number of moves. In the *Collegial* (*C*) stage each process executes a tabu search. When an improved solution is found, it is broadcast to other peers. The most advanced model is *Knowledge collegial* (*KC*). Here, the contents of communications are analyzed to retrieve additional information concerning global search trajectory and global characteristics of good solutions.

The third dimension, *Search Differentiation Strategy*, indicates whether the searches start from a single point or from different points in solution space and whether the used search strategies are the same or different. The four cases are: *SPSS: Single (initial)Point Single Strategy*, *SPDS: Single Point Different Strategies*, *MPSS: Multiple Points Single Strategy* and *MPDS: Multiple Points Different Strategies.*

It should be noted that similar parallelization taxonomies have been devised for other global optimization algorithms, e.g. branch and bound algorithms[2].

3 Parallel Tabu Search Algorithms for Graph Coloring

This section contains descriptions of three parallelization models used in this paper. All three are based on a well-studied *Tabucol* algorithm introduced in 1987 by Hertz and de Werra [10].

Tabucol is the first tabu search algorithm proposed for GCP. In particular it solves the decision version of the problem. For a given graph G and k the search space explored is the set of k-colorings of graph G. The goal is to find a coloring without any conflicting edges.

It must be noted that *tabucol* is often used as a local search operator in more general evolutionary heuristics, e.g. Galiniers and Hao's hybrid evolutionary algorithm for graph coloring [8].

The evaluation function f measures the number of conflicting edges in the solution. For a coloring $c : V \mapsto 1..k$ the value of $f(c)$ is equal to $|\{u, v\} \in E : c(u) = c(v)|$. *Tabucol* uses a simple *1-exchange* neighborhood, where a move is a pair (v, i) denoting assignment of color i to vertex v. After a move is performed the pair (v, i) becomes a tabu move for $\lfloor L + \lambda f(c) \rfloor$ succeeding iterations. Most implementations use $\lambda = 0.6$ and choose L randomly in [0..9].

Algorithms presented in this paper solve optimization version of the graph coloring problem. The initial coloring is generated using DSATUR greedy heuristic [3]. At the beginning of each iteration if the current solution x is a valid coloring ($f(x) = 0$) the smallest color class is removed, and the vertices are reassigned. The tabu search is used to reduce the number of conflicting edges in the solution.

3.1 Master-Slave Search

Algorithm 1. Master-slave search

```
x ← DSATUR(G)                    ▷ use a greedy heuristic to generate initial solution
while stop-condition do
    while $f(x) = 0$ do                                  ▷ while $x$ is a valid coloring
        $x$ ←REDUCE($x$)                                 ▷ reduce number of colors
    end while
    $m$ ←best allowed move in $N_i$                       ▷ ties are broken randomly
    if processor is slave then
        send $m$ to master
        $M$ ← receive selected move from master
    else
        moves[] ← receive moves from slaves
        $M$ ← select best move from moves[]               ▷ ties are broken randomly
        send $M$ to slaves
    end if
    $x \leftarrow x + M$                                  ▷ perform the chosen move
    update tabu list
end while
```

The first algorithm is a low-level parallelization of *Tabucol*. The neighborhood N is divided into p subsets N_i of size $\lfloor |N|/p \rfloor$ or $\lceil |N|/p \rceil$. The size of the neighborhood is $O(|V| * k)$, which is usually several orders of magnitude greater than the number of processors. Therefore even though the processors work on subsets of different sizes the load imbalance is negligible.

During each iteration every processor evaluates its part of the neighborhood and sends the best move to the *master*. The *master* evaluates the candidate moves, selects the best among them and broadcasts it. To speed up the exploration every process keeps its own copy of *tabu list*.

According to the aforementioned taxonomy this algorithm should be classified as *1C/RS/SPSS*. The behavior of this algorithm is almost the same as that of a sequential implementation. The difference is that ties are broken at N_i level, which means in fact that not all best moves have the same probability of being chosen. For large graphs the speedup is expected to be proportional to the number of processors.

3.2 Independent Search

A completely different approach to parallelization has been used in the second algorithm. Here we use p independent searches, each working on its own solution. Since there is no knowledge sharing between processors, the algorithm can be classified as *pC/RS*. As for the third dimension, *Search Differentiation Strategy*, two versions of the algorithm were compared - *MPSS* and *MPDS*. In the latter case the difference lies in the *tabu tenure*: λ parameter is distributed evenly in the range $[0.4, 0.9]$.

This approach should broaden the search increasing the rate of success. A small improvement in speed is also expected due to random nature of the search process.

Algorithm 2. Independent search

```
$x \leftarrow$ DSATUR($G$)                 ▷ use a greedy heuristic to generate initial solution
while stop-condition do
    if $f(x) = 0$ then
        while $f(x) = 0$ do                              ▷ while $x$ is a valid coloring
            $x \leftarrow$REDUCE($x$)                     ▷ reduce number of colors
        end while
    end if
    $M \leftarrow$best allowed move in $N$                ▷ ties are broken randomly
    $x \leftarrow x + M$                                   ▷ perform the chosen move
    update tabu list
end while
report the solution to master
```

3.3 Cooperating Search

The last approach can be classified as *pC/C/MPSS*. It is based on the independent search with the following improvement. Whenever a new valid coloring is found, it is sent to the *master*. The *master* reduces the number of colors and sends the new coloring to all processors. The *slaves* abandon their current search and use the received solution to start a new one. The tabu list is not copied.

Algorithm 3. Cooperating search

```
$x \leftarrow$ DSATUR($G$)                      ▷ use a greedy heuristic to generate initial solution
while stop-condition do
    if $f(x) = 0$ then
        while $f(x) = 0$ do                                         ▷ while $x$ is a valid coloring
            $x \leftarrow$REDUCE($x$)                                ▷ reduce number of colors
        end while
        send $x$ to ***master***
    end if
    if processor is ***slave*** then
        if a new solution is available from ***master*** then
            $x \leftarrow$ receive solution from ***master***
        end if
    else
        if a better solution is available from any of the ***slaves*** then
            $x \leftarrow$ receive solution from ***slave***
            send $x$ to all ***slaves***
        end if
    end if
    $M \leftarrow$best allowed move in $N$                          ▷ ties are broken randomly
    $x \leftarrow x + M$                                            ▷ perform the chosen move
    update *tabu list*
end while
```

4 Results

Most of the experiments were performed on *holk* cluster in TASK Academic Computer Centre. *Holk* has 256 1.3 GHz Intel Itanium 2 processors with 3MB L3 cache memory. The 2-processor nodes are connected with a Gigabit Ethernet network. For master-slave algorithm it was important to compare its behavior on a cluster and on a shared-memory machine. This was done using *lomond*, a Sun Fire 15K server with 52 900MHz Ultrasparc III processors at Edinburgh Parallel Computing Centre.

4.1 Results for Master-Slave Search

The behavior of the proposed master-slave algorithm is the same as of the sequential algorithm, therefore the speedup can be measured by the sheer number of tabu search iterations per second. Figure 1 presents the performance for two distinct machines.

The difference in the behavior of the algorithm can be explained using a simple theoretical model. Complexity of a single iteration of a sequential tabu search algorithm can be defined as:

$$T_0 = T_{upd} + T_{best},$$

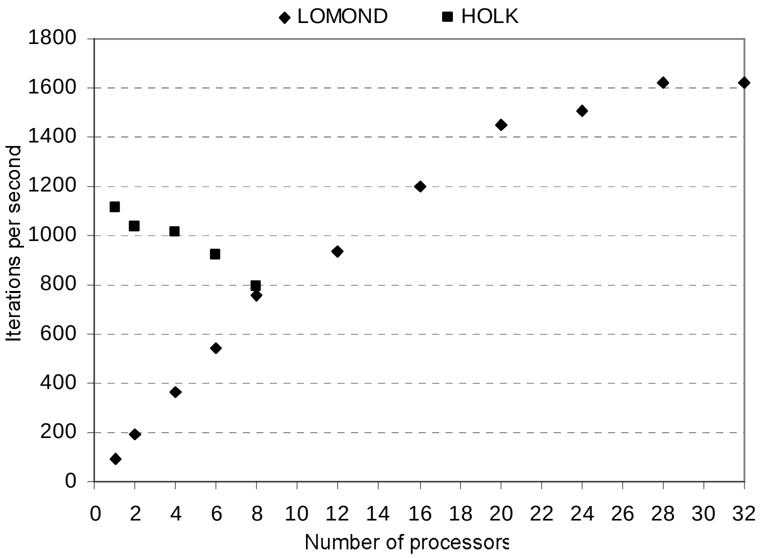


Fig. 1. Performance of the master-slave algorithm

where T_{upd} is the time required to update the tabu list, T_{best} is the time required to find the best move in N. For a parallel algorithm the time of finding the best move is divided among p processors, but there is an additional communication cost:

$$T_p = T_{upd} + T_{best}/p + \alpha p.$$

My *tabucol* implementation updates the tabu list in a single operation at the expense of memory usage. This means that the update cost is negligible. Using the experimental data it can be estimated that for $p \geq 2$:

$$T_{holk}(p) \approx 8.95 \cdot 10^{-3}/p + 1.84 \cdot 10^{-3}p,$$

$$T_{lomond}(p) \approx 1.07 \cdot 10^{-2}/p + 9.87 \cdot 10^{-6}p.$$

This is a rough estimate as the model used is quite simple, but shows the major difference between the two machines - for *holk* the communication cost is of the same order of magnitude as the neighborhood evaluation cost, whereas for *lomond* it is almost 3 orders of magnitude smaller. This explains why for *holk* even for small values of p the increase of communication cost caused by using one more processor exceeds the speedup of neighborhood evaluation. Experimental data shows that the parallel implementation performs worse than the sequential one not only in terms of total computation time, but also in terms of wall-clock time. There were no experiments with more than 8 processors of *holk* as the trend was obvious.

On the other hand for *lomond* an almost linear speedup is observed up to eight processors. Using sixteen processors *lomond* outperforms *holk* even though its single processor is more than ten times slower than *holk*s. The maximum number of processors allocated to a single task was limited to 32, but one can notice that there is almost no increase in speed between 28 and 32 processors.

4.2 Results for Independent Search

In this experiment the behavior of independent search threads was investigated. The best solution of all threads is considered to be the best solution of the parallel algorithm. Therefore the average performance of the parallel search is better than the performance of a single search. Table 1 shows how long it takes to find a valid k-coloring for different values of k.

Table 1. Results for independent search

		1 processor		10 processors	
graph	k	successful runs	time [s]	successful runs	time [s]
`DSJC1000.5`	115	10/10	3	3/3	0
$\|V\| = 1000$	105	10/10	42	3/3	29
$\|E\| = 249826$	100	10/10	142	3/3	104
	95	7/10	1381	3/3	887
	94	2/10	2503	3/3	2427
`DSJC500.5`	60	10/10	4	3/3	0
$\|V\| = 500$	55	10/10	32	3/3	24
$\|E\| = 6262$	54	8/10	66	3/3	44
	53	5/10	421	3/3	221

4.3 Independent Search vs. Cooperating Search

More experiments were performed to check if knowledge sharing would improve the results. Table 2 provides results for several graphs from the DIMACS suite.

For the two large graphs: `DSJC1000.5` and `flat1000_50_0` the time limit was set to two hours, for smaller graphs it was one hour. The runs were conducted using 10 processors.

Knowledge sharing improves the behavior of the algorithm. For easy colorings that are found within one minute the speedup is not obvious because of the communication overhead created by frequent colorings exchange. The cooperating search also improves the success rate within the specified time limit. For some tough cases the average search time for the independent search is lower than for the cooperating search. The reason for that is the time limit used - for the independent search only 'lucky' runs that end within the limit are averaged, whereas for the cooperating one most of the runs end successfully, but the average time is closer to the limit.

4.4 Results for Random Graphs

This section gives the results of experiments on random graphs. A random graph $G_{n,p}$ has n vertices and every two vertices are joined by an edge with probability

Table 2. Results for cooperating search

graph	k	*pC/RS/MPSS* successful runs	time [s]	*pC/KS/MPSS* successful runs	time [s]
`DSJC1000.5`	100	10/10	157	10/10	118
$\|V\| = 1000$	95	10/10	960	10/10	722
$\|E\| = 249826$	94	10/10	2533	10/10	2092
	93	0/10	-	2/10	4355
`DSJC500.5`	55	10/10	25	10/10	23
$\|V\| = 500$	54	10/10	42	10/10	38
$\|E\| = 62624$	53	10/10	239	10/10	216
`flat1000_50_0`	110	10/10	7	10/10	7
$\|V\| = 1000$	100	10/10	149	10/10	121
$\|E\| = 245000$	95	10/10	331	10/10	298
$\chi = 50$	93	10/10	3308	10/10	1640
	92	4/10	6465	9/10	6812
`flat300_28_0`	36	10/10	3	10/10	3
$\|V\| = 300$	35	10/10	8	10/10	5
$\|E\| = 21695$, $\chi = 28$	34	3/10	72	10/10	66
`le450_25c`	28	10/10	3	10/10	3
$\|V\| = 450$	27	10/10	62	10/10	58
$\|E\| = 17343$, $\chi = 25$	26	0/10	-	0/10	-

Table 3. Average number of colors used

n	p	$\chi*$	χ_*	DSAT	time limit [s]	processors 1	4	16
100	0.25	10	6	10.3	30	9.0	9.0	9.0
	0.50	16	11	17.8		16.0	15.6	15.1
	0.75	26	19	30.0		25.9	25.2	25.0
300	0.25	19	14	22.5	90	19.0	18.5	18.4
	0.50	35	27	43.2		35.7	35.4	35.1
	0.75	57	48	72.7		60.6	59.6	58.9
500	0.25	27	20	33.1	150	27.5	27.3	27.0
	0.50	50	41	65.2		54.1	53.6	53.1
	0.75	86	74	110.6		92.4	91.6	91.0

p. For every pair of n and p five graph instances were generated. In [7] Johri and Matula have presented means to calculate $\chi*$, the estimated chromatic number and χ_*, a probabilistic lower bound for the chromatic number. In our experiments the probability that $\chi(G_{n,p}) \geq \chi_*$ was greater than $1 - 10^{-6}$.

Tables 3 and 4 compare greedy algorithm DSATUR and sequential tabu search with cooperating parallel tabu search using 4 and 16 processors. The results were averaged over five runs for each of the five instances of $G_{n,p}$. Computation time of DSATUR algorithm was less than one second for all graphs.

In the first experiment tabu search was allowed to run for a specified amount of time. Table 3 shows the average number of colors used in the best coloring found.

Based on the results from the first experiment tabu search was configured to look for solutions using at most k colors. Table 4 shows the success rate and the average time needed to find a solution of a specified quality for $p = 0.5$. The run was considered unsuccessful if a solution was not found within the time limit of sixty seconds for every hundred vertices.

Using 16 processors improved the best coloring found for each instance by one or two colors. Cooperating search found good solutions faster and with greater probability.

Table 4. Average times of finding a valid k-coloring

				1 processor		4 processors		16 processors	
n	$\chi*$	DSAT	k	succ	time(σ)	succ	time(σ)	succ	time(σ)
100	16	17.7	16	25	0.1(0.3)	25	0.2(0.6)	25	0.0(0.2)
			15	3	2.3(3.3)	11	0.7(1.1)	22	2.4(9.8)
200	26	31.0	26	24	7.7(10.0)	25	2.8(4.0)	25	1.6(1.4)
			25	1	35.0(0.0)	3	8.0(4.3)	16	28.9(30.5)
300	35	42.6	36	23	19.7(22.4)	25	6.0(3.0)	25	3.4(1.6)
			35	6	38.8(25.5)	19	37.2(33.4)	25	23.2(19.6)
400	43	54.5	45	25	40.9(30.9)	25	22.1(16.9)	25	13.3(5.9)
			44	6	64.8(36.3)	16	85.2(45.8)	22	42.8(29.2)
500	50	64.8	54	25	78.4(47.4)	25	46.6(17.0)	25	31.0(8.7)
			53	11	122.5(40.3)	19	164.7(69.2)	25	105.9(61.7)

5 Concluding Remarks

For the coarse-grained architecture of *holk* cluster it was shown that the simple master-slave algorithm is slower than the sequential algorithm executed on a single processor. However this is not true for a shared-memory architecture of *lomond*. An almost linear speedup is achieved for up to eight processors, and the performance for sixteen or more processors is better than that of *holk*.

The multiple path strategy proved to be successful in improving the quality of best solution found. It also reduced time needed to find k-colorings for low values of k. Nevertheless on a shared-memory computer the increase of search depth for a master-slave algorithm has a much greater impact on the performance for high values of k (worse colorings).

Many of recently proposed graph coloring techniques use tabu search or other local search algorithms as means to improve solutions. Galinier and Hao [8] in their genetic-local search hybrid method perform a TS before inserting the result of a crossover into the population. In *variable neighborhood search* during every iteration the algorithm makes a big, variable move (change) to current solution and tries to improve it with *Tabucol*. Both techniques could perform significantly better if they were to make use of parallel computation. This possibility should be investigated further.

Acknowledgements

The experiments were performed on two high performance computers: *holk* cluster of the TASK Academic Computer Centre[2] in Gdañsk and *lomond*, a Sun Fire 15K server of the Edinburgh Parallel Computing Centre[3].

Access to *lomond* was granted under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme.

References

1. Avanthay, C., Hertz, A., Zufferey, N.: Variable Neighborhood Search for Graph Coloring. European Journal of Operational Research 151, 379–388 (2003)
2. Baravykaitė, M., Čiegis, R., Žilinskas, J.: Template realization of the generalized branch and bound algorithm. Mathematical Modelling and Analysis 10(3), 217–236 (2005)
3. Brélaz, D.: New Methods to Color the Vertices of a Graph. Communications of the ACM 22, 251–256 (1979)
4. Chiarandini, M., Dumitrescu, I., Stülze, T.: Local Search for the Colouring Problem. A Computational Study. Technical Report AIDA-03-01, Darmstadt University of Technology (2003)
5. Crainic, T.G., Toulouse, M., Gendreau, M.: Towards a Taxonomy of Parallel Tabu Search Heuristics. INFORMS Journal of Computing 9, 61–72 (1997)
6. Crainic, T.G.: Parallel Computation, Co-operation, Tabu Search. In: Rego, C., Alidaee, B. (eds.) Adaptive Memory and Evolution: Tabu Search and Scatter Search, Kluwer Academic Publishers, Dordrecht (2002)
7. Johri, A., Matula, D.W.: Probabilistic bounds and heuristic algorithms for coloring large random graphs, Technical Report 82-CSE-6, Southern Methodist University (1982)
8. Galinier, P., Hao, J.-K.: Hybrid Evolutionary Algorithm for Graph Coloring. Journal of Combinatorial Optimization 3, 379–397 (1999)
9. Galinier, P., Hertz, A.: A survey of Local Search Methods for Graph Coloring. Computers & Operations Research (to appear)
10. Hertz, A., de Werra, D.: Using Tabu Search Techniques for Graph Coloring. Computing 39, 345–351 (1987)

[2] http://www.task.gda.pl/

[3] http://www.epcc.ed.ac.uk/

TreeP: A Self-reconfigurable Topology for Unstructured P2P Systems

Euloge Edi, Tahar Kechadi, and Ronan McNulty

University College Dublin, Belfield Dublin 4, Ireland
edi@ucd.ie

Abstract. We present an efficient and robust network topology for managing distributed resources in P2P-Grid environments. This topology is called *TreeP* and exploits both features of P2P and grid computing models. It uses P2P properties in looking for available resources, while using Grid properties to implement communications and computations on a distributed computing platform. Here, we present how this architecture is distributively built and maintained, the related properties and our first experimental results. We show that this topology is very scalable, robust, load-balanced, and easy to construct and maintain.

1 Introduction

1.1 Motivations

Peer-to-Peer (P2P) and Grid Computing are computation paradigms for which usability doesn't need to be proved out to both scientific and non-scientific users. P2P technology allows ad-hoc communities of low-end clients to advertise and access resources (data, video, music, ...) on the communal computer which is transparent to the users. This model is exemplified by systems such as Gnutella [11] and Kazaa [8] which allow users to share the storage capacity and the network bandwidth for files facilities. On other hand, Computing Grids are mostly used to allow seamless access to supercomputer and data set for computation [9]. The set of resources participating to such kind of system are generally dedicated and known in advance. Even though two of them provide different services with different functionality implemented in different ways, they could be merged to provide high computation facilities which is highly scalable. This paper concerns the design and the implementation of a flexible virtual computing environment, called *TreeP*; Which takes advantage of both P2P and grid computing techniques to build a robust and competitive computing system with *infinite* resources. The nodes joining the system are organized in a tree-hierarchy topology following B^+_tree model [4]. The *TreeP* topology is motivated by two main goals: firstly, the computing environment is very volatile as the participating peers are not booked and fixed in advance. Therefore, the performance prediction of such a system needs a frequent update of the system state. It can be easily achieved by using tree topology. Secondly, P2P systems have scalability and look-up issues that need to be delt with. The tree topology guarantees a diameter, which increases logarithmically with the number of nodes; While maintain the system due to nodes migration, introduces small overheads.

1.2 Related Work

Three different type of applications and researches topics are conducted on this area. All of them build a virtual topology [5] on top of the real existing one. Knowing that this topology is implemented by allocating new routing table, the challenge resides in the size of each neighbourhood in order to provide a robust network. The first type concerns the look-up services. They are applications and services aimed to provide efficient data location on a P2P based systems. The leading idea consists in defining an overlapping network topology with low diameter in order to reduce the cost of looking-up through the network. We have then some applications for which we give the couple (degree, diameter). CAN$(d, dN^{1/d})$[1] [10], Chord$(log(N), log(N))$ [12], PASTRY$(log(N), log(N))$ [3]; N being the total number of nodes in the system. The second group applications [6] is built to provide robust computing environment or to be an ideal basis middleware for others computing environment. They don't necessary take into account the computing performance. They all provide mechanisms and tools for unstructured network topologies, with very specialised features, such as Grid FTP. Some provide additional and more sophisticated features: to mask the Grid heterogeneity and complexity; to interface with other middleware systems; to check-point and migrate tasks [13]; to provide high-level security components [2,6]. The third approach concerns itself with performance criteria which require that the target system scales well, reacts efficiently to *the leaving and joining* of peers, and provides a small diameter for better network performance. These systems include DGET[2] [7] (which uses *TreeP* topology) and XtremWeb [2]. *TreeP* then builds a topology for which the size of the neighbourhood is in $O(d)$ while the diameter of the system is in $log_d(N)$.

In this paper we propose a flexible virtual network topology as a solution to the network performance issue. In section 2, we present the design of the system followed by its implementation in section 3. This theory is validated through experimental results in section 4; And we conclude in section 5.

2 *TreeP* Design

Let $\mathcal{V}$ be the set of all the physical nodes of the system with $|\mathcal{V}| = N$ and d the order of the tree built. We are in the case where a given application can't be performed efficiently on a single computing node. Then an initial node which we call the root launches a handler which hunts resources on the network. Nodes are then found independently. With the set of available physical nodes, the problem consists in building a virtual computing Grid with characteristics

[1] CAN uses a d-dimensional Cartesian coordinates.

[2] DGET is a P2P-based Grid Computing environment on which peers are organized following *TreeP* model.

such as coherency, robustness and as cheaper as possible considering the amount of messages exchanged between nodes to reach this goal. To achieve this goal, we organize nodes hierarchically using a tree based topology (B^+*_tree* [4]). The strength of B^+*_tree* lies in properties such as: The insertion/deletion executed in $O(log(N))$ relatively to the running time and which leave the tree balanced; The height h of the tree which is logarithmic in the number of nodes $(2(\lceil \frac{2d}{3} \rceil + 1)^{h-1} - 1 \leq N \leq (2d+1)^h - 1)$; And the longest path between any two nodes is in $O(log_d(N))$. Even if we follow the B^+*_tree* concept, all the elementary operations and characteristics we inherited from this model have to be meaningful for our topology. In this way, several differences exist from *treeP*:

Nodes are part of the system at the time they are hooked to the hierarchy. Then, if several nodes are found from the beginning which should belong to the hierarchy, they should contribute to the system at the same time and not been inserted one by one like one does in B^+*_tree* construction. A block in a *treeP* topology is a set of nodes which know each other. A physical block is at level_$h-1$ while a virtual one is at level lower than $h-1$. Father/child relation are between nodes. It represents responsibility relationship between nodes implemented by the fact that children have to report to their father about their state. We define a sub-tree which can be rooted at any level as a tessellation.

Being the set of parameters $P_i(t) = \left(p_i^1(t), p_i^2(t), \ldots, p_i^l(t), \ldots, p_i^L(t)\right)$, for each $v_i \in \mathcal{V}$, describing its characteristics such as CPU power, its network capacity, its network connection bandwidth, its storage free space, its workload, etc. Some parameter values depend on the overall system behaviour over time and should therefore be updated dynamically. A *TreeP* like tree topology (Figure 1) of order d and of height h is a balanced tree of h levels; on which nodes at a given level are clustered on full connected sub-networks (called block) which are themselves connected to their father which is a node at the level just above.

Initially, all these nodes do not know themselves. We suppose that they are all known by the first node which initiate the demand of resources. These nodes can communicate only by sending messages. peers of the same level are connected by reliable connection.

Proposition 1. *Given the constraint that each physical node has one and only one virtual parent, there exists always a hierarchy with respect to* B^+*_tree characteristics.*

Proof. In the tree built following *treeP* characteristics there is only one virtual node associated to one physical node. This proposition shows that there is enough virtual nodes to fill the blocks created on the sub-tree rooted at the root of the tree and where the leaves are at the level $h-1$ of the global tree.

In fact there is more nodes at the leaves level on the entire tree than the number of virtual node in the sub-tree. If we take a regular tree, with θ children for one master, the sub-tree contains $\frac{\theta^h - 1}{\theta - 1}$ nodes for a tree of height h; But there are θ^h nodes at the leaves of the global tree.

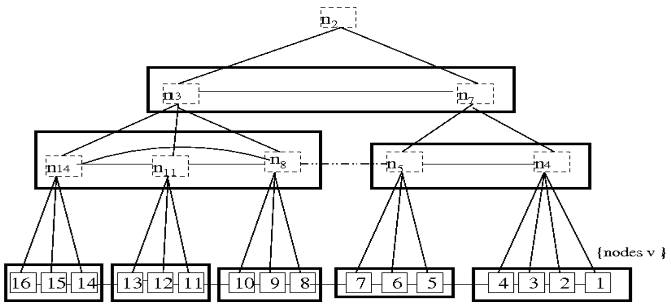


Fig. 1. Tree topology representing *TreeP*. The tree is an overlay topology (virtual topology) built on top of the existing one. In this example, $h = 4$ and $N = 16$ and $d = 2$. Nodes on the lowest level are the leaves and the real physical ones. Nodes on the level other than the leaves are virtual one. Node n_{11} is the virtual representation of the node v_{11}. The set $\{v_{16}, v_{15}, v_{14}\}$ forms a physical block while the set $\{n_{14}, n_{11}, n_8\}$ forms a virtual one. *TreeP* topology consists in allocate new routing table and create dedicated paths to connect nodes. The system works using the new routing table allocated on the nodes. A computing node can send or receive messages to/from processors which are only adjacent to it (ie connected by virtual edge).

3 *TreeP* Implementation

3.1 Hierarchy Creation

To build this hierarchy, we need a global and unique identifier for each physical and virtual node. In our case, we assign to each node a couple of parameter (Id, w_i) where Id is the identifier of the node, which is unique and can be generated by any hash function. Id can be an IP-address, an arrival rank, etc. w_i is the weight of the node. The weight should take into account the processing power of each node and its time since it joined the system. The weights are updated as soon as one of the node parameter has changed. For a given node v_i, its associated weight is

$$W_i(t) = \begin{cases} \sum_l \alpha_i^l(0) * l & \text{if t=0} \\ e^{\frac{1}{t_{join}}} \sum_l \alpha_i^l(t) * l & \text{otherwise} \end{cases}$$

where $\alpha_i^l(t) = p_i^l(t)/max_{n \in Universe} p^l$; node's parameters value are normalized using the highest value of the given parameter among the existing computing machines(fixed reference). i is a random rank given to the node on the system $(1 \leq i \leq N)$. t is the current time-step, t_{join} is the arrival time. So the nodes with the biggest weight are the most powerful. The algorithm (algorithm 1) used to build the hierarchy is presented here:

Proposition 2. $N_{block} * 2d \geq N$. *By construction, there is any underflowed blocks at any level.*

Algorithm 1. *TreeP* hierarchy pseudo-algorithm

```
1: Count the number of available nodes ($nn = N$) and sort them in the increasing
   order of their weight;
2: while $nn > 1$ do
3:   Compute the number of block $N_{block} = \left\lfloor \frac{\lfloor \frac{nn}{2} \rfloor}{2d} \right\rfloor + \left\lceil \frac{\lfloor \frac{nn}{2} \rfloor}{\lceil \frac{2d}{3} \rceil} \right\rceil$ ;
4:   if $nn \le 2d$ then
5:     all the nodes belong to $block_0$
6:   else
7:     $\beta \leftarrow \lfloor \frac{nn}{N_{block}} \rfloor + 1$ ;
8:     $\alpha \leftarrow N - N_{block} \lfloor \frac{nn}{N_{block}} \rfloor$
9:     if $\alpha = 0$ then
10:      $my_{block} = \lfloor node_{rank}/\beta \rfloor$
11:    else
12:      if $node_{rank} < (N_{block} - \alpha)\beta$ then
13:        $my_{block} = block_{\lfloor \frac{node_{rank}}{\beta} \rfloor}$
14:      else
15:        $my_{block} = block_{\lfloor \frac{node_{rank}}{\beta - 1} \rfloor} - 1$
16:      end if
17:    end if
18:  end if
19:  Elect[1] a virtual node for tessellation ($nn$ is the number of promoted nodes);
20: end while
21: Elect the root of the tree;
```

Proof. We use proof by contradiction.
Suppose that we have $\lfloor \frac{n}{N_{block}} \rfloor < \lceil \frac{2d}{3} \rceil$. It means

$$\lfloor \frac{n}{N_{block}} \rfloor \le \lfloor \frac{n}{\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2d} \rfloor + \lfloor \frac{\lfloor \frac{n}{2} \rfloor}{\lceil \frac{2d}{3} \rceil} \rfloor} \rfloor \le \frac{2d}{3} < \lceil \frac{2d}{3} \rceil$$

$$\text{it implies } \frac{3}{2d} < \frac{\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2d} \rfloor + \lfloor \frac{\lfloor \frac{n}{2} \rfloor}{\lceil \frac{2d}{3} \rceil} \rfloor}{n} \le 2\frac{\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{\lceil \frac{2d}{3} \rceil} \rfloor}{n} \le 2\frac{\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{\lceil \frac{2d}{3} \rceil} \rfloor}{n} \le \frac{\lfloor \frac{n}{\lceil \frac{2d}{3} \rceil} \rfloor}{n}$$

$$\text{and } \frac{3}{2d} < \frac{1}{\lceil \frac{2d}{3} \rceil} \text{ which is not possible.}$$

So the number of blocks created at each level of the tree contains enough space to hold all the processing units that belong to this level. The blocks are filled in respect with B^{+}*_tree* limitations.

Proposition 3. *If the parameters of the peers do not change during the construction phase, then building TreeP hierarchy should be done by using $O(Nlog(N))$ messages. The storage capacity used on each node to allow the execution of the different steps of the algorithm is $O(1)$.*

Proof.
- At each level_i from the root to the leaves, we have at most $\frac{N}{k^{h-i+1}}$ blocks, considering that $k = \lceil \frac{2d}{3} \rceil$. The leader election process at each level and the replacement process is considered as electing a leader two times in the same number of blocks. Knowing that the number of blocks in the system is less than $2\frac{N}{k}$, we have the given complexity.
- each node needs only to store its rank, the number of total node which is replaced at each stage by the number of block, and the number of block.

3.2 Node Insertion

Insert node in the system is a regular operation executed several times. The node which joins the system should be inserted in a block without disturb the balance of the tree and without consume the resources of the system. All nodes join the system from the root of the tree. In fact, an incoming node ask for a service at the level_0 (root level) of the tree. This service concerns the physical block it should belong to. Depending of its weight, the node goes down the tree to reach the right block it should belong to at the level_$h-1$ (leaves). At each level above the leaves, a virtual node is chosen by its parent to hold the new entry. Simple criteria like *the one which has less than* $2d-1$ *brothers* can be used. This procedure is the same which is ran on the B^+*_tree*.

Being the node v_k which joins the tree $\mathcal{T}$, the insertion algorithm (algorithm 2) is shown below:

Algorithm 2. *TreeP* insertion algorithm: Insertion($\mathcal{T}$, v_k)

```
$level = 0$
while $level \leq h-1$ do
    find the right place for $v_k$
    $level \leftarrow level + 1$
end while
insert $v_k$ in the right block (being $block_k$ ) ;
if block overflowed then
    $block \leftarrow block_k$
    while block is overflowed do
        create new block
        redistribute nodes with one of my neighbours
        elect new leader
        promotes leaders to our father block
        $block \leftarrow block's\ \ father$
    end while
end if
```

Proposition 4. *The insertion algorithm is in $O(log_d(N))$ in the number of messages sent if there is no balancing. The balancing is done in $O(log_d(N))$.*

Proof. At each level of the tree, the new node makes comparisons with nodes which form the block of its right tessellation. For each of these blocks, it receives at most a constant number of message $(2d - 1)$.

3.3 Node Deletion

Node deletion on the hierarchy has the meaning of a node which leaves the system for any reason. This node can at most belong to two levels (its virtual representation and its physical one at the leaves).

Being v_k the node to be deleted, B_k respectively (B'_k) the physical block (the virtual block) to which it belongs; The deletion algorithm (algorithm 3) is shown below:

Algorithm 3. *TreeP* deletion algorithm: Deletion($\mathcal{T}$, v_k)

```
block ← B_k,  level ← h − 1
while level > 0 do
  if B_k is underflowed then
    if B_k has a neighbour then
      if  this neighbour has more than ⌈2d/3⌉ nodes then
        borrow one element from him
      else
        decrease from one level block'sfather
        merge these two blocks
      end if
    else
      decrease from one level all nodes in block
      decrease from one level block'sfather
    end if
  else
    if block == B'_k then
      elect one node in the tessellation which will be promoted to replace k
    else
      block ← block's  father
      level ← level − 1
    end if
  end if
end while
```

Proposition 5. *Deletion algorithm is of complexity $O(log_d(N))$, in the number of messages sent to balance the tree.*

Proof. In the worst case, the deletion algorithm merges blocks at each level from the leaves to the root of the tree. merging two blocks is done by sending at most $2d$ messages to the nodes which belong to the two blocks to be merged.

We propose creation, insertion and deletion algorithms to build an hierarchic balanced tree with the same properties as the B^+*_tree*. The leader election is done independently by each concerned block. This construction algorithm adapts itself to node reorganization due to changes on the computing node parameters. It doesn't depend of the size of the network.

Proposition 6. *In a stable system, the describe algorithms build a tree where there isn't empty level.*

Proof. By construction there is not empty level. If a block becomes unbalanced, it's after an insertion or a deletion; Because nodes migrations on the system leaves the tree balanced. But for insertion and deletion, we provide balancing strategies.

This proposition shows that all the services provided on *TreeP* topology are always assumed by a given node on the hierarchy.

3.4 Look-Up Algorithm

The look-up algorithm 4 is based on the different routing tables allocated on the computing nodes. Its goal is to find a path between two distinct nodes of *TreeP*. Due to the fact that any nodes has a global view of the system, the look-up algorithm is based on a blind seeking strategy

Algorithm 4. *L*ook-up pseudo-algorithm.

```
1: if the seeking node is on your routing table then
2:     stop looking
3: else
4:     send the request to your father;
5:     send the request to the nodes you know at your level;
6:     send the request to your children if and only if your weight is greater than that
       of the seeking node;
7: end if
```

Proposition 7. *Algorithm 4 has a complexity of $O(h)$ number of steps executed and of $O(N)$ number of nodes visited. In the worst case, the total exchanged messages is of $O(N)$.*

Proof. The look-up message goes from the leaves to the root and returns to the leaves. In the worst case each node receives the message.

4 Simulation

The TreeP model is implemented on a cluster of Pcs. We use 8 SUN mono processor running solaris 5.9; connected by fast Gigabyte Ethernet. On this machine, we use LAM/MPI.7.1.1 as parallel computing environment. The peers are represented by parallel processes distributed over the computing platform. Each

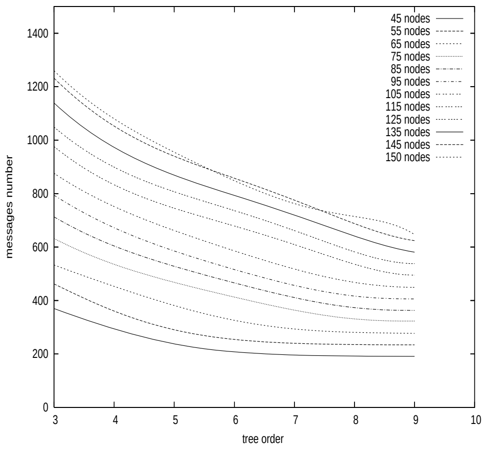


Fig. 2. Cost of building the hierarchy of different order

We plot the number of messages sent to build the hierarchy against the order of the tree for different numbers of nodes. This figure shows that the cost is decreasing while the order of the tree increases. We have that tree of higher order are more cheaper to build. But in fact the order of the tree influences directly the size of the routing tables managed by the father. So a balanced order should be find to satisfy both of these criteria.

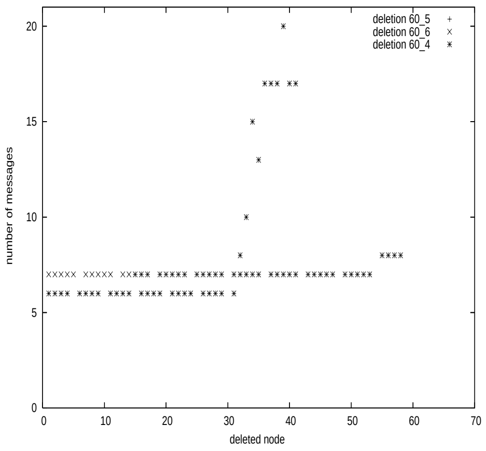


Fig. 3. Node deletion onto the system

Here the cost of deletion is shown on a tree containing 60 nodes while the order of the tree varies from 4 to 6. This shows that nodes which are at the leaves level (the most important part of nodes) are cheaply deleted. The deletion is more expensive for nodes belonging to the higher level of the tree, due to the fact that the tree is re-balanced if needed. *TreeP* promotes nodes to the higher level of the hierarchy depending on their computing performances and their stability; Therefore nodes at higher levels are supposed to be deleted rarely.

peers represents a computing node. For this experimentation, we give them different weights which are constant during the experiments. At the beginning, each peers implements routing tables for the underlying topology. We run different configurations of the TreeP system in order to simulate different scenarios and therefore evaluate its performance. We are interested in the cost of its construction expressed by the number of messages sent. In experience 1 (Figure 2), we vary the number of nodes on the system from 45 to 150 and the order of the tree from 3 to 9. We then count the number of messages exchanged by peers in order to build the full hierarchy. Experience 2 (Figure 3) concerns deletion onto the system. After building the hierarchy, we measure the cost of deleting each

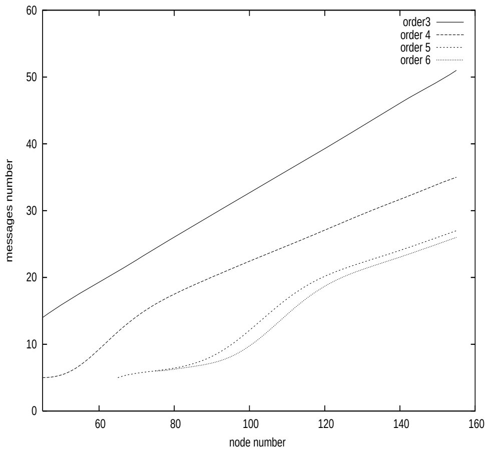


Fig. 4. Number of nodes against the number of messages sent for each order of the tree

For the routing, we put a time-to-live constraint on the look-up message so that it is destroyed after it crosses $h - 1$ nodes; h being the height of the tree. We plot the number of messages against the number of nodes for different order of the tree. In particular it is noted that the number of messages required decreases with increasing order of the tree; And also that the system is not flooded like shown by the worst case in the routing algorithm.

node from the entire system. In experience 3 (Figure 4), we test the look-up algorithm defined above. We randomly choose two nodes into the hierarchy and we count the number of message sent into the system to look for the second node beginning by the first one.

5 Conclusion

The main contributions of this paper concerns a novel P2P structured topology based on theoretical study with distributed algorithms to build and maintain the hierarchy; a system with $O(d)$ routing table per node while the diameter of the network is $O(log_d(N))$ and an experimental proof of its effectiveness. Here we supposed that the underlying topology is full connected. So we don't care about the charge induced by build an maintain *TreeP* topology on the real existing node and the network. Future work will explore this cost, it will be focus on defining a mathematical model which suit to this topology and therefore applications deployment on this architecture for distributed data mining.

Acknowledgments. This work is supported by the Marie-CurieGrant no MTKD-CT-2004-003134.

References

1. Beal, J.: Leaderless Distributed Hierarchy Formation, AI Memo 2002-021, p. 2139. MIT, Cambridge (2002)
2. Cappello, F., Djilali, S., Fedak, G., Herault, T., Magniette, F., Neri, V., Lodygenski, O.: Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. Future Gener. Comput. Syst. 21(3), 417–437 (2005)

3. Castro, M., Drushel, P., Kernarrec, A., Rowstron, A.: One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks. In: Proceedings of the First Workshop on Real, Large Distributed Systems, San Francisco, CA (December 5, 2004)
4. Comer, D.: Ubiquitous B-tree. ACM Comput. Surv. 11(2), 121–137 (1979)
5. Ferreira, R.A., Jagannathan, S., Grama, A.: Locality in a Structured Peer-to-Peer Network. Journal of Parallel and Dist. Comp. 66(2), 257–273 (2006)
6. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. The International Journal of Supercomputer Applications and High Performance Computing 11(2), 115–128 (1997)
7. Hudzia, B., Kechadi, M.T., Ottewill, A.: Treep: A Tree Based P2P Network Architecture. In: Proceedings of cluster 2005, Boston, Massachusetts, USA, pp. 1–13 (September 27-30, 2005)
8. Leibwitz, N., Ripeanu, M., Wierzbicki, A.: Deconstructing the Kazaa Network. In: WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications, Washington, DC, USA, p. 112. IEEE Computer Society Press, Los Alamitos (2003)
9. Pearce, S.: Gridpp: A UK Computing Grid for Particle Physics. The European Research Cons. for Info. and Math. News (ERCIM), 59(17) (October 2004)
10. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A Scalable and Content-Adressable Network. In: Proceedings of the 2001 ACM SIGCOMM Conference, San Diego, CA, USA, ACM Press, New York (2001)
11. Saroiu, S., Gummadi, K.P., Gribble, S.D.: Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts. Multi. Syst. 9(2), 170–184 (2003)
12. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proceedings of the 2001 ACM SIGCOMM Conf., San Diego, CA, ACM Press, New York (2001)
13. Thain, D., Tanenbaum, T., Livny, M.: Condor and the Grid. In: Berman, F., Fox, G.C., Hey, J.G. (eds.) Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons Inc., Chichester (2002)

A Middleware for Job Distribution in Peer-to-Peer Networks

Thomas Fischer, Stephan Fudeus, and Peter Merz

Department of Computer Science,
University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
{fischer, fudeus, pmerz}@informatik.uni-kl.de
http://dag.informatik.uni-kl.de/

Abstract. Recently, *Peer-to-Peer* (P2P) technology has become important in designing (desktop) grids for large-scale distributed computing over the Internet. We present a middleware for distributed computing based on Peer-to-Peer systems. When combining public-resource computation ideas with concepts of P2P networks, new challenges occur due to the lack of global knowledge as there is no central administration possible. Our Peer-to-Peer desktop grid (P2P Grid) framework includes an efficient and fault-tolerant communication scheme for job distribution combining epidemic algorithms with chord-style multicasts. We show that this hybrid scheme is more efficient than both epidemic algorithms and chord-style multicasting alone.

1 Introduction

Desktop grids allow the utilization of available computing power provided by modern desktop computers. Systems such as BOINC [1] or Fedak [2] are examples of desktop grids in which a central server coordinates the distribution of tasks among available worker nodes. The nodes perform the computation and send the results back to the server. The main drawback of this approach is that the central server may become a bottleneck and it represents a single point of failure. This problem has been addressed in [3], where a tree of scheduling nodes splits the "job" (scheduled computation) into subtasks before assigning subtasks to processing nodes.

An alternative approach is used in *Peer-to-Peer* (P2P) overlay networks [4], where independent nodes (also called *peers*) are connected by an arbitrary communication technology and no node has a dedicated role (e. g. central server). Usually nodes are connected using Internet technology and are heterogeneous regarding both connectivity and hardware. Characteristic for P2P systems is the creation of *overlay networks* which allow point-to-point connections between any two nodes. Therefore, both structured (e. g. Chord [5] and hypercubes [6]) and unstructured [7] networks can be built. P2P overlays may be either strict having no dedicated nodes or less strict allowing the concept of super-peers [8]. To increase reliability in network structure, techniques such as epidemic algorithms [9] may be utilized. Another aspect of P2P networks is that participating nodes may leave the network at any time and other nodes may join. Additionally, participating nodes may disconnect from the P2P network only temporarily. The following features make P2P network superior to "classical" approaches like client-server systems: (1) *Scalability* as there is no central component. Using appropriate algorithms, an operation such as a broadcast can be realized with a time complexity of $O(\log n)$. (2) *Robustness* as in theory any node can perform another node's work. The used job distribution algorithm has to consider this by introducing e. g. redundancy or state replication. (3) *Administrative overhead* is negligible for P2P networks, as nodes build a self-organizing network structure. New nodes have to know an initial entry point only. Disadvantages of P2P networks include the absence of global knowledge (no global view possible) and security considerations as e. g. credentials may have to be distributed and are no longer controlled by a central instance.

P2P systems are mostly known for file-sharing applications such as Gnutella [10]. But the general advantages of P2P systems and the interest for desktop grids motivate the approach to combine both techniques in a single middleware. Organic Grid [11] is a system which arranges peers in a tree, where high throughput nodes get located close to the root. Other approaches such as CompuP2P [12] or P-Grid [13] rely on distributed hash tables (DHTs), but to handle node failures, additional constructs such as making checkpoints periodically have to be be introduced. Super-peer structures are used in approaches like P3 [14] , but again fault tolerance is not addressed. Using epidemic algorithms, the distributed resource machine (DRM) [15] is fault tolerant, but requires more time and messages during the job distribution.

In this paper we present the aspect of job distribution in a middleware for scientific distributed computing following strictly the P2P paradigm combining both efficiency and fault tolerance. Our middleware allows any P2P node to initiate distributed computations, distributing both code and parameters efficiently and reliably among participating nodes, including the gathering of the computation's results. Jobs comprising communicating tasks such as in distributed evolutionary algorithms (DEA) [16] are supported, too.

Unlike Condor [17], our middleware focuses on P2P environments while keeping the architecture simple. E. g. we consider checkpoints to be too expensive in dynamic P2P environments. Furthermore, we allow applications running on top of the middleware to use the middleware's overlay network for inter-node communication making the need for PVM or MPI setups obsolete.

This paper is structured as follows: Section 2 discusses basic concepts of the used distribution techniques. In Sec. 3, our experimental setup is presented and experimental results are summarized in Sec. 4. Finally, concluding remarks as well as an outline of our future work are made in Sec. 5.

2 Middleware Design

Our middleware allows the execution of scientific distributed algorithms including code and parameter distribution as well as result gathering in a heterogeneous, dynamic and unreliable P2P environment. Existing distributed algorithms can easily be integrated by implementing simple Java interface classes. Compared to [18], we use a "flat" organization without hierarchy or node grouping. Furthermore, our middleware allows inter-node communication enabling to run cooperating algorithms (e. g. island model based DEAs).

The middleware consists of two logical layers. The bottom layer builds the P2P network by maintaining a list of neighboring nodes at each node. The top layer uses this neighbor list to select nodes during the job distribution process.

Internally, the middleware consists of several layers having separate, but synchronized message queues to communicate with their lower and upper layers.

2.1 Epidemic Algorithms

As implemented in our middleware, neighbor lists are exchanged using an *epidemic algorithm* [9], which provides a method for distributing information over a given network using the *gossip dissemination* paradigm. Inspired by biological epidemics, in epidemic algorithms nodes propagate incoming information to some of the other nodes they stay in contact with. After the initial outspread phase, epidemics are very resistant against obliteration compared to tree-based broadcast algorithms.

In epidemic algorithms, each node maintains an incomplete (and possibly outdated) list of nodes (including time stamps and idle status) participating in the same P2P system. Once within a time interval, a node will select another node at random from its neighbor list and contact it by sending a message containing the local node list. The contacted node will answer by sending back its own node list and therefore help propagating changes in network membership. Receiving nodes may integrate the incoming node list into their own neighbor lists. A node is member of an epidemic algorithm's network if it is in the neighbor list of at least one other node. As a large number of nodes may participate in the P2P network, each node has to limit the number of nodes it actually "knows". Therefore, outdated nodes will be removed from the list based on a given criterion (e. g. age [19]). An initial neighbor list is provided by the first node a peer contacts. This first contact node has to be provided by the user the first time the middleware is started. As nodes exchange lists even in idle mode, they permanently cause network traffic, but are resistant against obliteration.

Using an epidemic algorithm to broadcast information, a node would forward an incoming broadcast message to a number of randomly selected nodes from its neighbor list. The number of recipients determines the spread speed and network load of a broadcast. Given enough time, there is a high probability that every node will be informed. Epidemic algorithms have a high level of redundancy, which comes with additional overhead as messages may be sent several times to the same node [7].

2.2 Job Splitting and Distribution

Running computations in distributed environments requires that the job to be executed on top of the middleware can be efficiently "split" among the participating nodes. The knowledge how to split a job has to be user-supplied and is basically a function that for a given job returns an array of new job descriptions. For our middleware we propose two types of splitting a job into a set of subtasks depending on the algorithm class:

- The original problem is split into independent, different subtasks that can be computed independently on different nodes. The results of all subtasks will be sent back to and recombined by the initiating node using a merge-function of the initial task.
- A job is split into n identical subtasks, which due to stochastic components in the distributed algorithm may lead to n different results for the same problem. Nodes deliver their results back to the job initiator after finishing their local subtasks. In cases where nodes cooperate during the computation, the initiator's local result is used as the final result and no result retrieval and recombination from other nodes is necessary.

For a possible job distribution initiated by any peering node, subtasks have to be efficiently distributed among the peer set using a multicast or a broadcast. Using an existing spanning tree for a broadcast, job distribution would require $n-1$ messages. As there is no global knowledge of the network topology available in P2P networks, different broadcast algorithms have to be used. To distribute a job (set of subtasks), we propose two algorithms plus a hybrid form, all resembling the following general distribution scheme. At the beginning, the initiator node inserts a set of subtasks into its subtask queue. Every node regularly checks its queue and, if not empty, takes one subtask for local processing (if not already doing so) and iteratively forwards half of the remaining subtasks to neighbor nodes. The initiator's middleware instance keeps a copy of the original subtask list. If after some user-defined, application-specific timeout period not all subtask results have been returned to the initiator, missing subtasks will be reissued into the P2P network.

Nodes start processing subtasks from their local queue in parallel to distributing fractions of this list, allowing a node to process all subtasks by its own if it does not find a suitable neighbor to forward subtasks to.

Within the *epidemic job distribution* scheme, subtask lists get forwarded to randomly selected neighbors known to be in an idle state currently. Due to random distribution and possibly outdated neighbor lists, a job announcement may be sent to a node already working on a job, especially if the neighbor list is small compared to the number of tasks in the network. Nodes may reject incoming subtasks by not sending an acknowledgment, which will reside in the sender's queue for resubmission within the next interval. Subtask duplication is possible, if a node's acknowledgment is received too late at the sending node, falsely assuming subtask rejection at the receiver. The advantage of the epidemic distribution is its robustness against node failure compared to tree-based approaches.

Figure 1 shows an example how subtasks get distributed within a network. Nodes keep a single subtask for local processing before forwarding subtask list fractions to available nodes.

Epidemic algorithms cause some overhead as nodes may reject incoming tasks, so that the sender has to send the list to several nodes until an available node is found. In order to reduce this overhead, we arrange the nodes in a logical

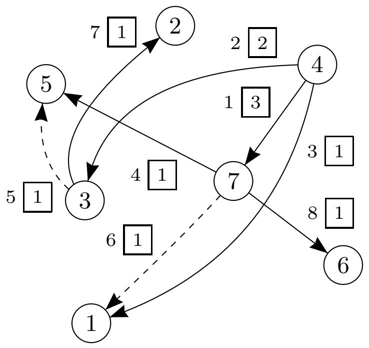
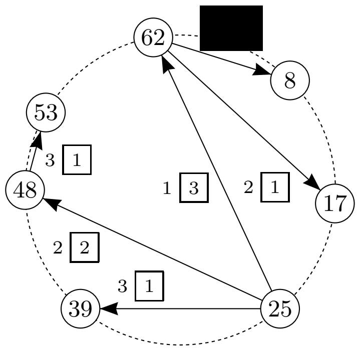


Fig. 1. Example how subtasks get propagated using the epidemic job distribution scheme. Node 4 starts broadcasting a job consisting of 7 subtasks. Arrows represent job announcement messages, arrow labels describe their chronological ordering (first number) and the number of subtasks transported with this message (boxed value). Rejected (ignored) announcement messages are shown as dashed arrows.

chord-like ring [5] as follows: Every node selects an unique id ranging from 0 to $2^r - 1$ holding $n \ll 2^r$ where n is the number of nodes and r a defined constant (here, $r = 63$). Now, a node with id i stores nodes with ids $i + 2^0$, $i + 2^1$, ..., $i + 2^{r-1}$ (mod 2^r) in a local table. As the ring will be sparse in most cases, nodes may select their id by random choice, but therefore it is very unlikely that a node with a requested id exists or is known to node i. In this case the node with the next higher id will be put into the corresponding table position. The table gets filled with nodes taken from the epidemic algorithm's neighbor list and thus getting along without additional messages. Additional memory is required to store the lookup table with r rows.

Using this ring a *chord-like job distribution* scheme [20] can be proposed as follows: A node that receives a job announcement message will iteratively split the list of subtasks and send one half to a node from the table until only one subtask is left which will be processed locally. Node selection depends on the distribution depth and the local distribution iteration. The initiator will send the half of the subtasks to the node at position $r - 1$ during the first iteration, the second iteration's receiver is taken from table position $r - 2$ and so on. The receiving node from table position p will start its subtask distribution at table position $p - 1$. If due to incomplete knowledge table positions are empty, these positions will be skipped. After reaching table position 0, remaining subtasks have to be processed locally.

See Fig. 2 for an example job distribution with 3 steps and 7 peers. Here, node 25 starts propagating a new job announcement message with 7 subtasks. Ideally, it would contact node 57 first, but as this node does not exist, node 25 contacts node 62 to delegate the propagation of the information for all nodes up to node 24. Next, node 25 contacts node 48 instead of the not existing node 41. Again, a part of the responsibility is propagated to node 48 to inform all nodes between node 49 and node 56. This procedure continues until all subtasks are distributed.



i	Theoretically	Actual Node
0	$25 + 2^0 = 26$	39
1	$25 + 2^1 = 27$	39
2	$25 + 2^2 = 29$	39
3	$25 + 2^3 = 33$	39
4	$25 + 2^4 = 41$	48
5	$25 + 2^5 = 57$	62

Node table for node 25.

Fig. 2. A chord-like broadcast in a ring with $2^r = 64$ possible node ids and $n = 7$ actual nodes. Node 25 starts broadcasting a job consisting of 7 subtasks. Arrows represent job announcement messages, arrow labels describe their chronological ordering (first number) and the number of subtasks transported with this message (boxed value). It is assumed that every node knows every other node.

The structured approach's advantage is that it is faster (only $\log(k)$ time steps and $k - 1$ messages for k subtasks) than the unstructured epidemic algorithm. Then again it may not be complete, as a node may not know appropriate nodes to send subtasks to (incomplete knowledge), forcing the node to process all remaining subtasks locally.

This problem is addressed by the proposed *hybrid job distribution* scheme. Whenever the chord-style distribution fails to send its remaining subtasks to other nodes, the subtasks will be handed over to the epidemic job distribution scheme avoiding that a single node has to process a larger number of subtasks if there are any free nodes known to the overloaded node. Since the chord-like algorithm distributes the subtasks in a structured way, the epidemic algorithm can start from nodes that are spread well within the network. Thus, the hybrid approach can distribute the subtasks better than each of the previous approaches alone, without inducing an additional overhead.

In all distribution schemes node failures are detected at the initiator if subtask results are missing. The initiator will then reissue the affected subtasks.

Whereas failures of computing nodes can be compensated, a failing initiator node will result in the loss of the computation. However, as all network communication is connectionless, minor network breakdowns are transparent from the middleware's point of view.

3 Experiments

To evaluate the behavior of the proposed job distribution techniques, we created an experimental setup injecting a simple dummy job into the network of middleware instances. This dummy job could easily be split into an arbitrary number of subtasks, where each subtask would sleep for 100 s regardless of the number of splits. Analyzing the behavior of a distribution technique included measuring the following aspects:

– Time until the last subtask arrives at its destination node
– Time until the last subtask has been processed
– Number of messages exchanged between nodes until all subtasks are distributed
– Number of computing nodes
– Maximum number of subtasks a single node had to process

Our experiments have been performed on a Linux cluster consisting of 16 computers, each equipped with a 3.0 GHz Pentium 4 and 512 MB RAM communicating over a switched Gigabit Ethernet. On each computer, 10 middleware instances were started with different process priority levels to simulate a P2P network consisting of 160 heterogeneous nodes. Computations were started after the stabilization of the P2P system (each node has a filled neighbor list). All experimental setups were repeated 20 times and average values were taken for further discussion. The initiator's timeout to reissue subtasks was set to 400 s, but never triggered in our controlled environment.

For this set of experiments, three parameters have been varied: (1) The number of subtasks a job was split into was set to either 64, 128, or 256. Having more subtasks than nodes in the largest setup, the middleware was forced to schedule more than one subtask to at least one node. (2) The size limit for the neighbor host list of each middleware instance was set to either 10, 40, or 160. For the two smaller values, nodes would always have an incomplete knowledge of nodes available in the P2P network, whereas for the largest value complete knowledge could be achieved. (3) The used job distribution scheme was either the epidemic, chord-style, or hybrid approach.

Compared to real-world P2P systems, this setup is synthetic as no real computation or resource consumptions is performed, but it is acceptable as our focus is on the job distribution performance. In reality, clients in P2P systems have to cope with shared, limited and unreliable resources such as network connection, main memory or CPU time.

4 Results

Results from the job distribution experiments are summarized in Tab. 1. The first three columns determine the experimental setup under consideration. The following two columns show the time required to distribute (last job announcement has been received) and to finish (all results have been sent back to the initiator) a job. The next column summarizes the number of nodes that are involved in the computation of subtasks. The two columns to follow show the number of job announcements sent using the epidemic or the chord-like distribution scheme, respectively. The last column shows the maximum number of subtasks computed by a single node within a distributed computation.

Evaluating the durations required to distribute and to finish all jobs, it can be observed that within each setup with fixed number of subtasks and neighbor

Table 1. Experimental results for different job distribution setups. The first three columns describe setup parameters. Results are averaged over 20 runs.

Number of Subtasks	Neighbor List Size	Distribution Scheme	Time [s] to distribute	Time [s] to finish	# Computing Nodes	Epidemic Job Announ.	Chord-like Job Announ.	Max. # Subtasks/Node
64	10	Chord-like	0.6	395.8	55.30	–	54.30	3.95
		Epidemic	14.1	121.7	63.90	89.45	–	1.10
		Hybrid	6.9	111.3	63.95	19.90	54.95	1.05
	40	Chord-like	0.3	420.7	55.45	–	54.45	4.20
		Epidemic	8.1	108.2	64.00	76.95	–	1.00
		Hybrid	4.1	104.6	64.00	12.65	55.30	1.00
	160	Chord-like	0.3	545.8	55.35	–	54.35	5.45
		Epidemic	7.4	107.5	64.00	74.90	–	1.00
		Hybrid	3.8	104.4	64.00	15.80	51.45	1.00
128	10	Chord-like	0.7	882.1	88.40	–	87.40	8.80
		Epidemic	92.0	194.9	123.90	271.00	–	1.80
		Hybrid	77.2	186.2	126.15	200.10	81.45	1.60
	40	Chord-like	0.6	1022.4	89.05	–	88.05	10.20
		Epidemic	17.5	117.6	127.95	226.20	–	1.00
		Hybrid	14.1	114.2	128.00	100.70	91.80	1.00
	160	Chord-like	0.4	792.0	95.15	–	94.15	7.90
		Epidemic	11.2	111.3	128.00	183.40	–	1.00
		Hybrid	9.1	113.8	127.85	79.25	87.45	1.05
256	10	Chord-like	0.7	1663.8	110.15	–	109.15	16.60
		Epidemic	177.2	280.5	158.15	730.40	–	2.55
		Hybrid	143.7	255.6	159.05	826.20	114.20	2.40
	40	Chord-like	0.6	1622.8	124.85	–	123.85	16.20
		Epidemic	115.0	215.1	160.00	1391.10	–	2.00
		Hybrid	111.7	211.8	160.00	1344.20	124.85	2.05
	160	Chord-like	0.6	1403.1	122.50	–	121.50	14.00
		Epidemic	112.5	212.6	160.00	712.65	–	2.00
		Hybrid	109.3	209.4	160.00	578.75	120.20	2.00

list size the chord-like distribution scheme requires the least time to distribute compared to both other schemes. It also holds that the chord-like distribution scheme requires the most time to finish a computation job (each subtask has been completed and results got sent back to the job initiator). Both effects can be explained by the fact that the chord-like distribution stops earlier when no appropriate nodes for further distribution are known and nodes have to process all subtasks by their own, which would be otherwise distributed further. On the other side, the epidemic distribution scheme has the highest distribution times which can be explained by two facts. First, a node with subtasks to distribute randomly selects a supposedly idle neighbor and sends a list of subtasks to it. As the receiving node may reject this subtask list if it is already working (detected at the sender only by a timeout), the sender has to select another neighbor. Especially with small neighbor lists it can take several tries until a willing neighbor is found. Second, if the number of subtasks is higher than the number of nodes (in our case always 160), there is simply no node available after assigning the first 160 subtasks, but nodes with subtasks continue to send out job announcement messages to find available nodes. Finally, the hybrid distribution scheme requires the least time to finish a computation.

From all job distribution schemes it can be expected that they assign at most one subtask to a free node, given that enough nodes are available. However, both the epidemic and the hybrid approach have more nodes participating in their computation in most cases, compared to the chord-like approach. This can be explained by the non-completeness of the chord-like distribution tree, when not being perfectly uniformly distributed. Both the epidemic and the hybrid approach utilize a close-to-optimum number of nodes for their computations which correlates to the low number of subtasks per node.

Regarding the number of job announcement messages exchanged among the nodes, for the chord-like scheme the number of job announcement message stays at the same level for a constant number of subtasks, increasing only slowly with the size of the neighbor list (column "Chord-like Job Announ."). This small increase can be explained by a more complete distribution tree due to a larger number of nodes known to other nodes. For the epidemic job distribution, the number of messages decreases with an increasing size of the neighbor list. An exception can be observed for the setup with 256 subtasks and neighbor list sizes of 10 and 40. Here, the number of epidemic job announcement messages for neighbor list sizes of 10 is considerably smaller compared to setups with a list size of 40 (730 and 826, respectively, versus 1391 and 1344). This is due to a side effect within the epidemic job distribution: Nodes select target nodes to send subtasks to based on the (possibly outdated) idle information from the epidemic neighbor list. If no node is marked available, the epidemic job distribution pauses. For setups with a list size of 10, list entries are mostly current, as outdated information is removed quickly. For setups with a list size of 40, the list contains entries with busy nodes still marked as available. The epidemic job distribution picks a supposedly available node, but the sent subtasks will be rejected.

Both the epidemic and the hybrid distribution scheme show a smooth subtask propagation as the maximum number of subtasks per node is always close to the optimum, contrary to the chord-like distribution scheme, where this number equals to 5–10% of the number of subtasks. Regarding the neighbor list size, it can be observed that the distribution time for epidemic and hybrid schemes decreases considerably when switching from list sizes of 10 to 40, but stays at the same level when switching to lists with 160 positions. It can be concluded that large neighbor lists or even complete knowledge of all nodes does not speed up the epidemic job distribution. Larger neighbor lists also tend to contain outdated information which may lead to undesired side effects.

Although this paper focuses on job distribution, we performed some basic experiments with simple distributed applications. For a fractal computation where splitting subtasks meant dividing the fractal image into smaller stripes, we observed that the absolute time to compute the image decreases with increasing number of subtasks due to better load-balancing. For a setup with a simple traveling salesman problem (TSP) algorithm utilizing middleware-supported inter-node cooperation by exchanging intermediate solutions, the final tour's quality increased with the number of subtasks (keeping the number of TSP algorithm's iterations constant for each subtask).

5 Conclusion

A middleware has been presented for distributed computing in desktop grids based on peer-to-peer overlay networks. The middleware incorporates a hybrid job distribution scheme combining fault-tolerant epidemic algorithms and efficient DHT information propagation. In experiments we have shown the effectiveness of this hybrid scheme compared to epidemic or chord-style job distribution alone.

If message complexity is critical, the chord-like distribution scheme is the best choice, but having the drawback of distributing subtasks very unbalanced among available nodes. The epidemic distribution scheme is preferred in cases with unreliable networks as it is resistant against obliteration. Both approaches are combined in the hybrid distribution scheme, which has in most cases the lowest times to finish, the highest number of computing nodes and the smoothest subtask distribution.

Future work will focus on two areas: First, we are researching on integrating concepts such as "network coordinates" to build an overlay network which incorporates the underlying network's structure and the differentiation of nodes into common and privileged peers. Second, using this middleware we are developing a distributed evolutionary algorithm (DEA) for large TSP instances.

References

1. Anderson, D.P.: BOINC: A System for Public-Resource Computing and Storage. In: 5th IEEE/ACM International Workshop on Grid Computing (2004)
2. Fedak, G., Germain, C., Neri, V., Cappello, F.: Xtremweb: A generic global computing system. In: CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid, Washington, DC, USA, p. 582. IEEE Computer Society Press, Los Alamitos (2001)
3. Page, A., Keane, T., Allen, R., Naughton, T.J., Waldron, J.: Multi-Tiered Distributed Computing Platform. In: Proc. of the PPPJ '03, pp. 191–194, Computer Science Press, Inc. (2003)
4. Clark, D.: Face-to-Face with Peer-to-Peer Networking. Computer 34(1), 18–21 (2001)
5. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. IEEE/ACM Trans. Netw. 11(1), 17–32 (2003)

6. Schlosser, M., Sintek, M., Decker, S., Nejdl, W.: HyperCuP – Hypercubes, Ontologies and Efficient Search on P2P Networks. In: Moro, G., Koubarakis, M. (eds.) AP2PC 2002. LNCS (LNAI), vol. 2530, pp. 112–124. Springer, Heidelberg (2003)
7. Portmann, M., Seneviratne, A.: Cost-effective Broadcast for Fully Decentralized Peer-to-peer Networks. Computer Communications 26(11), 1159–1167 (2003)
8. Yang, B., Garcia-Molina, H.: Designing a Super-peer Network. In: Proceedings of the 19th International Conference on Data Engineering (ICDE) (2003)
9. Eugster, P.T., Guerraoui, R., Kermarrec, A.M., Massoulie, L.: From Epidemics to Distributed Computing. IEEE Computer (2004)
10. Ripeanu, M.: Peer-to-Peer Architecture Case Study: Gnutella Network. In: Proc. of P2P 2001, p. 99. IEEE Computer Society Press, Washington (2001)
11. Chakravarti, A.J., Baumgartner, G., Lauria, M.: The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. In: Proceedings of the International Conference on Autonomic Computing (ICAC '04), New York (2004)
12. Gupta, R., Somani, K.: A.: CompuP2P: An Architecture for Sharing of Computing Resource. In: Peer-to-Peer Networks With Selfish Nodes. In: Proc. of the Second Workshop on the Economics of P2P Systems, Harvard University (2004)
13. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-Grid: A Self-organizing Structured P2P System. ACM SIGMOD Record 32(3), 29–33 (2003)
14. Oliveira, L., Lopes, L.M.B., Silva, F.M.A.: P: Parallel peer to peer. In: Gregori, E., Cherkasova, L., Cugola, G., Panzieri, F., Picco, G.P. (eds.) NETWORKING 2002. LNCS, vol. 2376, pp. 274–288. Springer, Heidelberg (2002)
15. Jelasity, M., Preuß, M., Paechter, B.: A Scalable and Robust Framework for Distributed Applications. In: Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002), pp. 1540–1545. IEEE Press, Los Alamitos (2002)
16. Bäck, T.: Evolutionary Algorithms in Theory and Practice. Oxford University Press, New York (1996)
17. Tannenbaum, T., Wright, D., Miller, K., Livny, M.: Condor – A Distributed Job Scheduler. In: Sterling, T. (ed.) Beowulf Cluster Computing with Linux, MIT Press, Cambridge (2001)
18. Verbeke, J., Nadgir, N., Ruetsch, G., Sharapov, I.: Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. In: Parashar, M. (ed.) GRID 2002. LNCS, vol. 2536, pp. 1–12. Springer, Heidelberg (2002)
19. Eugster, P.T., Guerraoui, R.: Probabilistic Multicast. In: Proceedings of the 3rd IEEE International Conference on Dependable Systems and Networks (DSN 2002), pp. 313–322 (2002)
20. Merz, P., Gorunova, K.: Efficient Broadcast in P2P Grids. In: Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid, Cardiff, UK, ACM Press, New York (2005)

Mobile Agents Self-optimization with MAWeS

Emilio Pasquale Mancini[1], Massimiliano Rak[2],
Salvatore Venticinque[2], and Umberto Villano[1]

[1] Università del Sannio, Benevento, Italy
{epmancini, villano}@unisannio.it
[2] Seconda Università di Napoli, Via Roma 29, IT-81031 Aversa (CE), Italy
{massimiliano.rak, salvatore.venticinque}@unina2.it

Abstract. In mobile agents systems, classical techniques for system optimization are not applicable due to continuous changes of the execution contexts. MAWeS (MetaPL/HeSSE Autonomic Web Services) is a framework whose aim is to support the development of self-optimizing autonomic systems for Web service architectures. In this paper we apply the autonomic approach to the reconfiguration of agent-based applications. The enrichment of the Aglet Workbench with a Web Services interface is described, along with the extensions to the MAWeS framework needed to support the mobile agents programming paradigm. Then a mobile agents application solving the N-Body problem is presented as a case study.

1 Introduction

The mobile agents programming paradigm is an emerging approach for distributed programming. Agents-based platforms are considered good solutions in many fields, such as GRID [1,2,3] or SOA (Service Oriented Architecture) [4,5]. In mobile agents systems, classical techniques for system optimization (such as *ad-hoc* tuning, performance engineered software development, ...) are hard to apply. This is essentially due to continuous changes of the execution contexts, as an agent is able to suspend its own execution, to transfer itself to another agent-enabled host and to resume its execution at that destination. So, even if the mobile agents approach may help to develop performance-oriented applications, in practice the only solution to guarantee critical requirements seems to be the use of an architecture able to auto-configure and to auto-tune until the given requirements are met. Moreover, when an agent moves itself, it impacts the state of the new system. A prediction of the modified state can help to make good choices for agents reconfiguration.

Autonomic computing [6,7,8,9], whose name derives from the autonomic nervous system, aims to bring automated self-management capabilities into computing systems. In previous papers, we have introduced MAWeS (MetaPL/HeSSE Autonomic Web Services) [10,11], a framework whose aim is to support the development of self-optimizing autonomic systems for Web service architectures. It adopts a simulation-based methodology, which allows to predict system performance in different status and load conditions. The predicted results are used for a feedforward control of the system, which self-tunes *before* the new conditions and the subsequent performance losses are actually observed.

MAWeS is based on two existing technologies: the MetaPL language [12] and the HeSSE simulation environment [13]. The first is used to describe the software system and the interactions inside it; the latter, to describe the system behavior and to predict performance using simulation. Using MAWeS, it is possible to add self-adaptive features both at *service level*, i.e., building up services that optimize themselves in function of their overall usage [11], and at *application level*, i.e., focusing on the application behavior [10]. In the latter case, a standard client application interface, `MAWeSclient`, provides the general services that can be used and extended to develop new applications.

In this paper we propose to apply the proposed approach to the reconfiguration of agent-based applications integrating the Web Service and mobile agent programming paradigm. We aim at defining the potentiality of this approach and at proposing an architecture able to build self-optimizing agents. In order to achieve this result, we had to extend both mobile agent platforms and the framework architecture. Moreover, we had to build extensions to the description language and the simulation engine.

The reminder of the paper is organized as follows. The next section presents previous work and the scientific background. Section 3 describes our proposal, presenting the extension made to both the mobile agents platform and the MAWeS framework. Section 4 deals with an example, showing a mobile agents-based NBody application and its integration within MAWeS. The paper will end with our conclusions, which summarize the results obtained, and with a discussion of our future work.

2 Background

Our proposal for mobile agents self-optimization relies on previous research and on a large amount of existing software tools. In order to present the work done, in this section we introduce the main concepts about the mobile agent programming paradigm adopted, describing the chosen platform. Moreover, we describe the state of the art of the optimization framework adopted, MAWeS.

A mobile agent is a software agent with an added feature: the capability to migrate across the network, together with its own code and execution state. This paradigm allows both a pull and a push execution model [14]. In fact, the user can choose to download an agent, or to move it to another host. Mobility can provide many advantages for the development of distributed applications. System reconfiguration by agent migration can help to optimize the execution time by reducing network traffic and interactions with remote systems. Furthermore, stateful migration allows to redistribute dynamically the agents for load balancing purposes. Several different criteria can guide agent distribution, such as moving the execution near to the data, exploiting new idle nodes, or allocating agents on the nodes in such a way that communications are optimized.

Due to previous experiences and for the facilities it offers, we chose to adopt the Aglet Workbench, developed by IBM Japan research group [15]. Af far as interoperability is concerned, the Aglet Workbench is compliant with the MASIF specification [16]. It relies on a transport protocol that is an extension of http.

The MAWeS Framework has been developed to support the predictive autonomicity in Web Service-based architectures. It is based on two existing technologies: the MetaPL language [12] and the HeSSE simulation environment [13]. The first is used to describe the software system and the interactions inside it; the second, to describe the system behavior and to predict performance using simulation.

MetaPL is an XML-based meta-language for parallel program description, which, like other prototype languages, can also be used when applications are not (completely) available [12]. It is language independent, and can be adopted to support different programming paradigms. It is structured in layers, with a core that can be extended through *Language Extensions*, implemented as XML DTDS. These extensions introduce new constructs into the language. Starting from a MetaPL program description, a set of extensible filters makes it possible to produce different program *views*.

HeSSE is a simulation tool that allows to simulate the performance behavior of a wide range of distributed systems for a given application, under different computing and network load conditions. It makes it possible to describe distributed heterogeneous systems by interconnecting simple components, which reproduce the performance behavior of a section of the complete system (for instance a CPU, a network . . .). These components have to reproduce both the functional and temporal behavior of the subsystem they represent.

The MAWeS framework uses MetaPL descriptions and HeSSE configuration files to run simulations. Through the execution of multiple simulations, differentiated by one or more parameter values, it chooses the parameter set that optimizes the software execution. MAWeS is structured in three layers. The first one is the front-end, which contains the software modules used by final users to access the MAWeS services. The second one is the core, which includes the components that manage MetaPL files and make optimization decisions. The last one contains the Web Services used to obtain simulations and predictions through MetaPL and HeSSE.

MAWeS operates adopting two strategies:

Service Call optimizations (simulation, evaluation of the choices and application tuning) take place at application startup (increasing the application startup latency).

Reactive optimizations (simulation, evaluation of the choices and application tuning) take place asynchronously with the application and do not affect the application performance, except when the framework decides to change the application behavior.

The MetaPL/HeSSE WS interface defines a set of services that make it possible to automate the application of the methodology. When the MAWeS framework is firstly used, it is necessary to describe the components in MetaPL. This can be done before starting the development to obtain a prototype view to analyze (and to optimize), or, in parallel with it, to verify the design choices.

Inside the meta-description, it must be suitably identified the set of parameters that can be modified by the optimization engine. MAWeS will automatically perform a set of simulations varying the values of these parameters to find the optimal value set. The user can specify the tunable parameters by means of the *autonomic* MetaPL extensions, which define new MetaPL elements for the `Mapping` section [10].

3 MAWeS and Agents Integration

As pointed out in the introduction, the mobile agents paradigm is a powerful programming technique, which can help to reconfigure an application in a very easy and clean way, distributing data in a distributed system and possibly achieving better performance through load balancing. On the other hand, evaluation of overhead and actual performance of the application may be a hard task: agents run in an hosted environment (the mobile agent platform) that hides completely the hardware behavior. So, it is difficult to find out the best configuration for an application. The most viable solution seems to be the development of self-adapting applications.

The MAWeS Framework was initially developed in order to help application self-configuration, embedding into them a client that evaluates many different hardware/software configurations (i.e., distributions of software onto a distributed system and application parameters) in a completely different context, namely Web Services and hence service-oriented applications. In order to adapt the MAWeS framework to applications based on the mobile agents paradigm, it is necessary:

– to describe the evolution of a mobile agent application, in order to make it possible the prediction of its performance behavior on the target environment. This leads to the requirement for a Mobile Agent Extension for the MetaPL description language;
– to simulate the execution of the mobile agent based application. This involves the development of a new HeSSE library modeling the agents and their runtime support;
– the framework should be able to control and to change the mobile agents-based application. So it has to communicate with agents and/or their platform and to be able to move/clone/destroy/... the agents and to communicate with them through messages. Moreover, an agent should be able to invoke the framework in order to notify new changes into the environment.

The last point opens a new universe of problems, as it involves the integration between mobile agents application and the web services programming paradigm, which is the only form of interaction supported by MAWeS. We focused on two different ways to integrate a WS interface to mobile agent systems:

Mobile Agents as WS clients. A mobile agent is able to access to an external web services, i.e., it directly performs SOAP requests. This approach lets an agent to invoke MAWeS services.

Mobile Agent Platform as a WS server. The mobile agents platform has a Web Services interface. A WS client can access the mobile platform and send messages to the agents and/or change their state (clone, migrate, ...)

In order to simplify the implementation, here we focus only on the MAWeS Service Call Optimization strategies. This means that we aim at optimizing the starting distribution of the mobile agents to the distributed environment. Once the application is started, its behavior does not change. As a consequence, we implemented only the second proposed technique for mobile agent and WS integration, modifying the existing Aglets platform in order to add a Web Services interface.

Moreover, we assume that the available distributed environment does not change frequently, and so the data about the available platforms and their performance may be collected off-line, thus being available when the application starts. This implies that the MAWeS framework does not need to discover dynamically the changes in the execution environment (such as the availability of a new agent platform), as this information is provided to the framework off-line.

3.1 Mobile Agents and Web Services

To support the interaction between a mobile agent system and MAWeS, we enriched the Aglet Workbench [15] with a Web Service interface. We developed a Web Service that implements a SOAP bridge, enabling any requestor to invoke the agent platform facilities and to interact with the agents hosted locally or with remote ones. Furthermore, many basic platform facilities such as creation, cloning, dispatching have been exported as services. The available methods provide functionalities for agents creation, disposal, cloning, migration and discovery, along with point-to-point and multicast messaging. These methods invoke the Aglets API to exploit the functionalities of the agent context that has been created at startup. In order to add a Web Service interface to the agent platform, we turned it into a web application, to let it be executed in a container of the chosen application server, Jakarta Tomcat.

It should be noted that communication among agents and among agent servers exploits the Agent Transfer Protocol. On the other hand, communication between web service clients and the platform relies on SOAP messages, which are handled directly by Tomcat. SOAP messages may be service requests, such as agent creation or migration, or messages to be forwarded to the agents. The enhanced web agent platform will take care of distinguishing between the two, forwarding the incoming SOAP messages to the agents or performing the invoked service.

3.2 The MAWeS Agent Extended Architecture

Mobile agent-based applications launch depends on the platform chosen for their execution. Usually the graphical interface offers a screen in which the user chooses the class to be executed. The agents code is already on the server or accessible from a remote codebase. Thanks to the newly developed Web Services interface, we are able to create and to run an agent through a web services client application, which invokes the target environment. This client application is an extension of the standard MAWeS client.

The agent-based MAWeS client contains the description of the target mobile agents, MetaPL documents, and the link to their code (i.e., the link needed by the platform to create and to start up the agent). When the user starts up the client, it queries the MAWeS engine, which returns the list of servers to be adopted, the number of agents to start on each server and, possibly, application-dependent parameters.

The MAWeS core was extended in order to maintain information about the available mobile agents platforms. At the state of the art, the core retrieves the list of the available hosts and their performance parameters using an UDDI register. This means that parameters are statically stored. Future extensions will provide a protocol to check that the platforms are active and to measure their performance. The resulting architecture is shown in Figure 1.

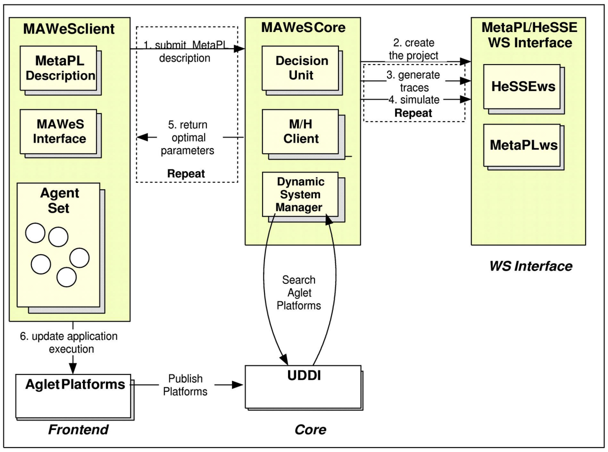


Fig. 1. The MAWeS Extended Framework for Mobile Agents

3.3 MetaPL Extension

The newly developed MetaPL extension defines a set of new MetaPL elements:

`Agent` this element replaces the standard MetaPL `Task` element and contains all the agent code. `name` and `id` attributes identify the agent.

`Create` this element represents an operation of agent creation. It describes the case in which an agent invokes the platform in order to create a new agent locally. The attribute `agent` contains the name of the type of agent created.

`Clone` this element represents an operation of cloning by an agent; it describes the case in which an agent invokes the platform in order to duplicate itself.

`Dispose` this element represents an operation of disposal by an agent (or an application). It describes an invocation to the platform in order to destroy an agent. The attribute `agentID` contains the identifier of the agent to be disposed.

`Activate/Deactivate` these elements represent an operation of activation/deactivation by an agent, an invocation to the platform in order to pause or awake an agent. The attribute `agentID` contains the identifier of the agent affected by the instruction.

`handleMessage` this element represents the activation of the message handling mechanism inside the agent. Message passing adopts the standard message passing MetaPL extension.

`Platform` this element, containedin the mapping section, represents an available platform. It supports the `platformID` attribute, which identifies the platform.

`AgentInstance` this element, contained in the mapping section, represents an available agent. It supports the `agentID` attribute, which uniquely identifies the agent and a `StartPlatformID`, which describes the platform on which it is created at startup.

All the elements support an optional `PlatformID` attribute, which reports the identifier of the target mobile agent platform. Figure 2 shows an example of MetaPL description. It contains the description of an agent A which creates an agent B and migrates on a new platform. Agent B just migrates on a new platform.

3.4 HeSSE Library

HeSSE is a complex simulation environment, easily extensible due to the adoption of a component-based approach. It is out of the scope of this paper to give

```
<MetaPL>
   <Code>
    <Agent name="A">
        <Create name="B" /> <Migrate aglet="test" platformID="2" />
    </Agent>
    <Agent name="B"> <Migrate aglet="test" platformID="2" /> </Agent>
  </Code>
  <Mapping>
    <Platform platformID="1"/> <Platform platformID="2"/>
    <AgentInstance name="A" platformID="1"/>
    <AgentInstance name="B" platformID="1"/>
  </Mapping>
</MetaPL>
```

Fig. 2. An Example of Mobile Agent MetaPL Description

a detailed description of the simulation models adopted by the simulator and of the details of the models developed to simulate the agents. This section will give an overview of the newly developed components and of their behavior and usage. In order to simulate the Aglets mobile agents platform, we developed a library that extends the features of an existing Message Passing library. The newly developed library offers the following components:

Aglet_Daemon. It is the Aglets daemon, and corresponds to the mobile agent platform. This component offers to the simulation environment the service to control the agents.

Aglet_Data. This component is used only to maintain common information among multiple platforms.

Aglet. It reproduces the behavior of a single agent. It is fed with a trace file, whose format is the same of the "original" HeSSE Message Passing trace files, extended with constructs to take into account the typical agent operations (create, clone, dispose, migrate, activate, deactivate).

4 An Example of Tool Execution

As case study, we propose a mobile agent application able to solve the problem of gravitational interaction between n different bodies (NBody problem) [14]. The sequential algorithm that solves the well-known N-body problem computes, during a fixed time interval, the positions of N bodies moving under their mutual attraction. The program repeats, in steps of a fixed time interval, the construction of an octal tree (octree) whose nodes represents groups of nearby bodies, the computation of the forces acting on each particle through a visit of the octree, the update of position and speed of the N particles. Figure 3 describes the behavior of the application. Note that the application involves a set of different agents. The main agent, `master`, waits for a message from the available workers, in order to coordinate them. Each worker, once created, signals the master and starts computations. The MAWeS Client contains the MetaPL descriptions, and at application startup, it sends them to the MAWeS Core. In this case we does not take into account any application specific parameter, so the tool has only to get the list of available platforms, and decide how many agents to start and where. The list of available platforms is stored in an UDDI register together with their performance indexes useful for simulation. The MAWeS core, so, generates a new set of mapping sections, as shown in Figure 3 (mapping section) composed of the list of available platforms, the Number of Agents to be created and the list of Agent Instances. The tool generates a different mapping section for each different configuration to be tested. By default, it generates a number of mapping sections that is two times the number of available platforms, with an increasing number of agents of the worker type. The resulting document is given to the MAWeS component, which performs performance evaluations (MH-Client), simulating the configuration and returning the predicted response time. The MAWeS core returns to the client the mapping section, together with a

```
<Code>
 <Agent name="master" >
    <Loop iteration="Nstep" > <Block>
        <Loop iteration="NAgentstep" > <Block>
            <Receive kind="ready" />
        </Block></Loop>
        <Multicast kind="go" /> <Multicast kind="subtree"/>
        <Multicast kind="subforces"/> <Multicast kind="posANDvel"/>
    </Block></Loop>
 </Agent>
 <Agent name="Worker">
   <CodeBlock region="Initialize" />
    <Loop iteration="Nstep" > <Block>
       <Send to="master" kind="ready" /> <Receive kind="go" />
       <Codeblock region="BuildTree" />
       <Multicast kind="subtree" /> <Receive kind="subtree" />
       <Codeblock region="Compute" />
       <Multicast kind="subforces"/> <Receive kind="multicast" />
       <Codeblock region="Update" />
       <Multicast kind="posANDvel"/> <Receive kind="posANDvel" />
    </Block></Loop>
  </Agent>
</Code>
 <Mapping>
   <Platform platformID="1"/> <Platform platformID="2"/>
   <NumberOfAgents value="3" variable="NAgent"/>
   <AgentInstance name="master" platformID="1"/>
   <AgentInstance name="worker" platformID="1"/>
   <AgentInstance name="worker" platformID="2"/>
 </Mapping>
```

Fig. 3. NBODY with mobile Agent MetaPL Description

link to the available platforms. Then the MAWeS client starts the agent on the chosen platforms.

5 Conclusions and Future Work

In this paper we have proposed an extension to the MAWeS framework for building self-optimizing mobile agent applications. We have shown its new architecture, and the main extension to the description language MetaPL and to the HeSSE simulator. Moreover, the Aglet mobile agents platform has been extended to turn it into a web application. We have shown how the framework can work on a simple example. As this paper presents only the tool architecture and its main features, a next step for our research will be a detailed performance analysis of the approach, pointing out the conditions in which the tool can be useful.

References

1. Zhang, Z.R., Luo, S.W.: Constructing grid system with mobile multiagent. In: Int. Conf. on Machine Learning and Cybernetics, vol. 4, pp. 2101–2105 (November 2003)
2. Tveit, A.: jfipa - an architecture for agent-based grid computing. In: AISB'02 Convention, Symposium on AI and Grid Computing (2001)
3. Aversa, R., Martino, B.D., Mazzocca, N., Rak, M., Venticinque, S.: Mobile agents approach the grid. In: Martino, B.D.M., et al. (eds.) Engineering the Grid: status and perspective, American Scientific Publishers (2006)
4. Greenwood, D., Calisti, M.: Engineering web service - agent integration. In: IEEE Conference of Systems, Man and Cybernetics, The Hague, IEEE Computer Society Press, Los Alamitos (2004)
5. Lyell, M., Rosen, L., Casagni-Simkins, M., Norris, D.: On software agents and web services: Usage and design concepts and issues. In: 1st International Workshop on Web Services and Agent Based Engineering, Sydney, Australia (July 2003)
6. Birman, K.P., van Renesse, R., Vogels, W.: Adding high availability and autonomic behavior to web services. In: Proc. of 26th Int. Conf. on Soft. Eng., Edinburgh, UK, pp. 17–26. IEEE Computer Society Press, Los Alamitos (2004)
7. IBM Corp.: An architectural blueprint for autonomic computing. IBM Corp., USA (October 2004)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
9. Zhang, Y., Liu, A., Qu, W.: Software architecture design of an autonomic system. In: Proc of 5th Australasian Workshop on Software and System Architectures, Melbourne, Australia, pp. 5–11 (April 2004)
10. Mancini, E., Rak, M., Torella, R., Villano, U.: Predictive autonomicity of web services in the MAWeS framework. J. of Computer Science 2(6), 513–520 (2006)
11. Mancini, E., Rak, M., Villano, U.: Autonomic web service development with MAWeS. In: Proc. of 20th Int. Conf. AINA06, Austria, pp. 504–508 (April 2006)
12. Mazzocca, N., Rak, M., Villano, U.: MetaPL a notation system for parallel program description and performance analysis. In: Malyshkin, V. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 80–93. Springer, Heidelberg (2001)
13. Mancini, E., Mazzocca, N., Rak, M., Torella, R., Villano, U.: Performance-driven development of a web services application using MetaPL/HeSSE. In: Proc. of 13th Euromicro Conf. on Parallel, Distributed and Network-based Processing, Lugano, Switzerland (February 2005)
14. Grama, A., Kumar, V., Sameh, A.: Scalable parallel formulations of the barnes-hut method for n-body simulations. Parallel Computing 24, 797–822 (1998)
15. Lange, D., Oshima, M.: Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley, Reading (1998)
16. Milojicic, D., et al.: Masif: The omg mobile agent system interoperability facility. In: Rothermel, K., Hohl, F. (eds.) MA 1998. LNCS, vol. 1477, pp. 50–67. Springer, Heidelberg (1998)

Performance Impact of Resource Conflicts on Chip Multi-processor Servers

Myungho Lee, Yeonseung Ryu, Sugwon Hong, and Chungki Lee

Department of Computer Software, MyongJi University,
Yong-In, Gyung Gi Do, Korea 449-728
{myunghol, ysryu, swhong, cklee}@mju.ac.kr
http://www.mju.ac.kr/myunghol

Abstract. Chip Multi-Processors (CMPs) are becoming mainstream microprocessors for High Performance Computing and commercial business applications as well. Multiple CPU cores on CMPs allow multiple software threads executing on the same chip at the same time. Thus they promise to deliver higher capacity of computations performed per chip in a given time interval. However, resource sharing among the threads executing on the same chip can cause conflicts and lead to performance degradation. Thus, in order to obtain high performance and scalability on CMP servers, it is crucial to first understand the performance impact that the resource conflicts have on the target applications. In this paper, we evaluate the performance impact of the resource conflicts on an example high-end CMP server, Sun Fire E25K, using a standard OpenMP benchmark suite, SPEC OMPL.

1 Introduction

Recently, microprocessor designers have been considering many design choices to efficiently utilize the ever increasing effective chip area with the increase of transistor density. Instead of employing a complicated processor pipeline on a chip with an emphasis on improving single thread's performance, incorporating multiple processor cores on a single chip (or Chip Multi-Processor) has become a main stream microprocessor design trend. As a Chip Multi-Processor (CMP), it can execute multiple software threads on a single chip at the same time. Thus a CMP provides a larger capacity of computations performed per chip for a given time interval (or throughput). Examples are Dual-Core Intel Xeon [3], AMD Opteron [1], UltraSPARC IV, IV+, T1 microprocessors from Sun Microsystems [12], [14], IBM Power 5 [5], among others. Shared-Memory Multiprocessor (SMP) servers based on CMPs are already introduced in the market, e.g., Sun Fire E25K [12] from Sun Microsystems based on dual-core UltraSPARC IV processors. They are rapidly adopted in High Performance Computing (HPC) applications as well as in commercial business applications.

Although CMP servers promise to deliver higher chip-level throughput performance than the servers based on the traditional single core processors, resources on the CMPs such as cache(s), cache/memory bus, functional units, etc., are shared among the cores on the same processor chip. Software threads running on the cores of the same processor chip compete for the shared resources, which can cause conflicts and hurt performance. Thus exploiting the full performance potential of CMP servers is a challenging task. In this paper, we evaluate the performance impact of the resource conflicts among the processor cores of CMPs on a high-end SMP server, Sun Fire E25K.

For our performance evaluation, we use HPC applications parallelized using OpenMP standard [9] for SMP: SPEC OMPL benchmark suite [11]. Using the Sun Studio 10 compiler suite [13], we generate fairly high optimized executables for SPEC OMPL programs and run them on E25K server. In order to evaluate the performance impact of the resource conflicts on the shared resources, level-2 cache bus and main memory bus, 64-thread (and 32-thread) runs were conducted using both cores of 32 CMPs (16 CMPs for 32-thread run) and using only one core of 64 CMPs (32 CMPs for 32-thread run). The experimental results show 17˜18% average (geometric mean for the 9 benchmark programs) slowdowns for the runs with resource conflicts than without the conflicts. Benchmarks which intensively utilize the memory bandwidth or allocate large amounts of memory suffer more due to the resource conflicts.

The rest of the paper is organized as follows: Section 2 describes the architecture of an example CMP server, Sun Fire E25K. Section 3 describes the OpenMP programming model and our test benchmark suite, SPEC OMPL. It also describes how to generate optimized executables for SPEC OMPL. Section 4 first shows the settings for utilizing Solaris 10 Operating System features useful for achieving high performance for SPEC OMPL. Then it shows the experimental results on E25K. Section 5 wraps up the paper with conclusions.

2 Chip Multi-processor Server

In this section, we describe the architecture of an example high-end CMP server which we used for our performance experiments in this paper. The Sun Fire E25K server is the first generation throughput computing server from Sun Microsystems which aims to dramatically increase the application throughput by employing dual-core CMPs. The server is based on the dual-core UltraSPARC IV processor and can scale up to 72 processors executing 144 threads (two threads per each UltraSPARC IV processor) simultaneously. The system offers up to twice the compute power of the UltraSPARC III Cu (predecessor to UltraSPARC IV processor) based high-end systems.

The UltraSPARC IV contains two enhanced UltraSPARC III Cu cores (or Thread Execution Engines: TEEs), a memory controller, and the necessary cache tag for 8 MB of external L2 cache per core (see Fig. 1). The off-chip L2 cache is 16 MB in size (8 MB per core). The two cores share the Fireplane System Interconnect, as well as the L2 cache bus. Thus they become the potential source of performance bottlenecks.

The basic computational component of the Sun Fire E25K server is the UniBoard [12]. Each UniBoard consists of up to four UltraSPARC IV processors,

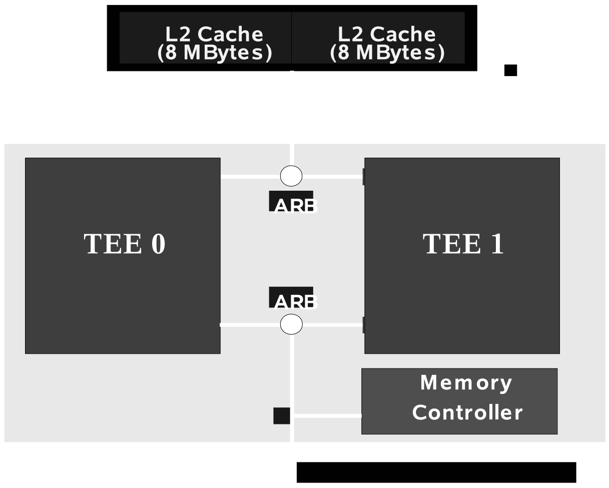


Fig. 1. UltraSPARC IV processor

their L2 caches, and associated main memory. Sun Fire E25K can contain up to 18 UniBoards, thus at maximum 72 UltraSPARC IV processors. In order to maintain cache coherency system wide, the snoopy cache coherency protocol is used within the UniBoard and directory-based cache coherency protocol is used among different UniBoards. The memory latency, measured using `lat_mem_rd( )` routine of lmbench, to the memory within the same UniBoard is 240nsec and 455nsec to the memory in different Uniboard (or remote memory).

3 SPEC OMPL Benchmarks

The SPEC OMPL is a standard benchmark suite for evaluating the performance of OpenMP applications. It consists of application programs written in C and Fortran, and parallelized using the OpenMP API [11]. The underlying execution model for OpenMP programs is fork-join (see Fig. 2) [9]. A master thread executes sequentially until a parallel region of code is encountered. At that point, the master thread forks a team of worker threads. All threads participate in executing the parallel region concurrently. At the end of the parallel region (the join point), the team of worker threads and the master synchronize. After then the master thread alone continues sequential execution. OpenMP parallelization incurs an overhead cost that does not exist in sequential programs: cost of creating threads, synchronizing threads, accessing shared data, allocating copies of private data, bookkeeping of information related to threads, and so on.

The SPEC OMPL benchmark suite consists of nine application programs representative of HPC applications from the areas of chemistry, mechanical engineering, climate modeling, and physics. Each benchmark requires a memory size up to 6.4 GB when running on a single processor. Thus

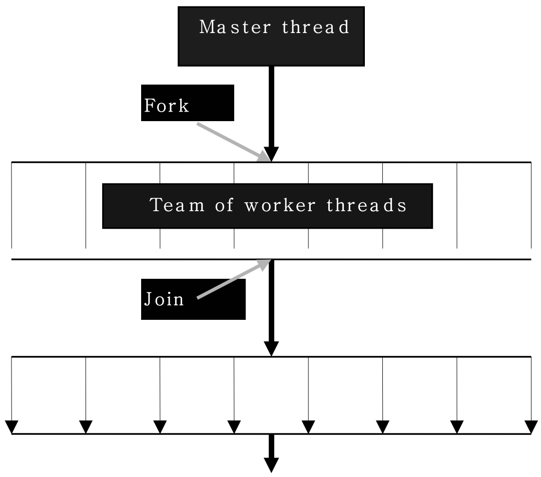


Fig. 2. OpenMP execution model

the benchmarks target large-scale systems with 64-bit address space. Table 1 lists the benchmarks and their application areas.

Table 1. SPEC OMPL Benchmarks

Benchmark Programs	Application Areas	Programming Languages

Using Sun Studio 10 compiler suite [13], we've generated executables for the benchmarks in SPEC OMPL suite. By using combinations of compiler options provided by the Sun Studio 10, fairly high level of compiler optimizations is applied to the benchmarks. Commonly used compiler flags are `-fast -openmp -xipo=2 -autopar -xprofile -xarch=v9a`. Other further optimization flags are applied to individual benchmark also. These options provide many common and also advanced optimizations such as scalar optimizations, loop transformations, data prefetching, memory hierarchy optimizations, interprocedural optimizations, profile feedback optimizations, among others. (Please see [13] for more details on the compiler options.)

The `-openmp` option processes openmp directives and generate parallel code for execution on multiprocessors. The `-autopar` option provides automatic parallelization by the compiler beyond user-specified parallelization. This can further improve the performance.

4 Performance Results

Using the compiler options described in section 3, weve generated highly optimized executables for SPEC OMPL. In this section, we first describe the system environments on which the optimized executables are executed. We then show the performance results, impact of resource conflicts, on Sun Fire E25K. We also show one example compiler technique which can reduce the impact of resource conflicts along with the experimental results.

4.1 System Environments

The Solaris 10 Operating System provides features which help improve performance of OpenMP applications. They are Memory Placement Optimization (MPO) and Multiple Page Size Support (MPSS). MPO feature can be useful in improving performance of programs with intensive data accesses to localized regions of memory. With the default MPO policy called first-touch, memory accesses can be kept on the local board most of the time, whereas, without MPO, those accesses would be distributed all over the boards (both local and remote) which can become very expensive. MPSS can improve performance of programs which use a large amount of memory. Using large size pages (supported by MPSS), the number of TLB entries needed for the program and the number of TLB misses can be significantly reduced. Thus performance can be significantly improved [10]. We are enabling both MPO and MPSS for our runs of SPEC OMPL executables.

OpenMP threads can be bound to processors using the environment variable SUNW_MP_PROCBIND which is supported by thread library in Solaris 10. Processor binding, when used along with the static scheduling, benefits applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region will either be in the local cache from a previous invocation of a parallel region, or in local memory due to the OS's first-touch memory allocation policy.

4.2 Impact of Resource Conflicts on CMP

As mentioned in Section 2, two cores on one UltraSPARC IV CMP share the L2 cache bus and the memory bus, which are potential sources of performance bottlenecks. In order to measure the performance impact of these resource conflicts on SPEC OMPL, we've measured the performance of 64-thread (and 32-thread) runs in two ways:

1. Using 64 (32) UltraSPARC IV processors, thus using only one core per processor.
2. Using 32 (16) UltraSPARC IV processors, thus using both cores of the processor. In this case, there are possible resource conflicts between the two cores.

Table 2. 64-thread cases-64x1 vs. 32x2

Benchmark Programs	64 chips x 1 core (sec)	32 chips x 2 cores (sec)	Speed-up of 1 core over 2 cores

Table 2 (and Table 3) shows the run times for both 1 and 2 using 64 threads (32 threads) measured on Sun Fire E25K with 1050Mhz UltraSPARC IV processors. They also show the speed-ups of 1 over 2. Overall, 1 performs 1.18x (1.17x) better than 2 in 64-thread run (32-thread run). Benchmarks with greater performance gains from 1 core show the following characteristics:

– 313.swim_l: This is a memory bandwidth-intensive benchmark. For example, there are 14 common arrays accessed all over the program. All the arrays are

Table 3. 32-thread cases-32x1 vs. 16x2

Benchmark Programs	32 chips x 1 core (sec)	16 chips x 2 cores (sec)	Speed-up of 1 core over 2 cores

of the same size (7702 x 7702) and each array element is 8 bytes long. Thus the total array size is 6,342 Mbytes. The arrays are seldom reused in the same loop iteration and the accesses stride through the arrays continuously. When only one core is used per processor, it can fully utilize the L2 cache and the main memory bandwidth available on the processor chip, whereas when two cores are used the bandwidth is effectively halved between the two cores. This led to 1.39x gain of 1 over 2. Unless some aggressive compiler optimizations are performed to increase the data reuses, the benchmark will suffer from the continuous feeding of data to the processor cores which burns all the available memory bandwidths.

– 315.mgrid_l: This benchmark, like 313.swim_l, requires high memory bandwidth. Although this benchmark shows some data reuses (group reuse) of the three dimensional arrays which are intensively accessed, the same data is reused at most three times. Therefore, the accesses stride through the arrays. Using only one core can have much higher memory bandwidth, as in 313.swim_l's case, which leads to 1.20x gain.
– 325.apsi_l and 331.art_l: These benchmarks allocate large amount of memory per thread at run-time. For example, 325.apsi_l allocates 6,771 Mbytes of an array at run-time besides many other smaller arrays. The dynamic memory allocation can be parallelized, however it still requires a large memory space per processor core. Thus, instead of allowing 8 threads to allocate large memory on the same UniBoards memory, allowing only 4 threads, by using only one core per each UltraSPARC IV, can have significant performance benefit. 331.art_l also shows similar characteristics.
– 327.gafort_l: In this benchmark, the two hottest subroutines have critical sections inside the main loops. Also they both suffer from intensive memory loads and stores generated from the critical section loops. These take up large portions of the total run time. Placing 2 threads on one UltraSPARC IV (by using both cores) can reduce the overhead involved in the locks and unlocks. However, allocating 8 threads on two different UniBoards (by using only one core in each UltraSPARC IV) reduces the pressure on the memory bandwidth significantly compared with allocating 8 threads on the same UniBoard. The benefit from the latter dominates that of the former.

Benchmarks other than the above (311.wupwise_l, 317.applu_l, 321.equake_l, 329.fma3d_l) relatively give less pressure on the memory bandwidth and/or consume smaller amount of memory. Thus the performance gap between 1 and 2 is smaller. These benchmarks are not heavily affected by the resource conflicts and are more suitable for execution on CMP servers.

In order to show the performance impact due to resource conflicts from a different perspective, we've calculated the speed-ups from 32-thread runs to 64-thread runs in two ways:

– Calculating scalabilities from 32 x 1 run to 64 x 1 run, i.e. when only core is used.
– Calculating scalabilities from 32 x 1 run to 32 x 2 run. Thus 64-thread run is performed with resource conflicts.

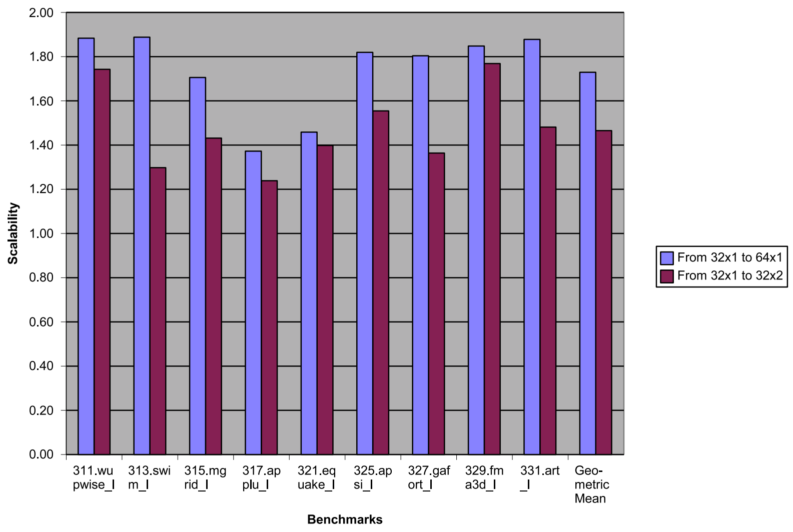


Fig. 3. Scalabilities from 32-thread runs to 64-thread runs

Fig. 3 shows the scalabilities in both cases. For the benchmarks which are affected more due to the resource conflicts, the two scalability bars show bigger gaps.

4.3 Algorithmic/Compiler Techniques to Reduce Resource Conflicts on CMP

For benchmarks which suffer a lot due to resource conflicts, algorithmic and/or compiler techniques are needed to reduce the penalties. For example, aggressive procedure inlining and skewed tiling [7] technique can be used for 313.swim_l. The skewed tiling, when applied to 313.swim_l, can convert a major portion of the memory accesses to cache accesses by increasing data reuses. Thus can significantly cut down the traffic to main memory and make a large performance gain.

Using the compiler flags `-Qoption iropt -Atile:skewp` provided in Sun Studio10 Fortran compiler, we've generated a new executable for 313.swim_l. We've run both the original and the new executables on a smaller Sun SMP server (SunFire E2900 employing 12 UltraSPARV IV processors) using both cores of each UltraSPARC IV. For these runs we've reduced the array sizes to 1/4th of the original sizes. (There are fourteen two-dimensional arrays with sizes 7702 x 7702 in 313.swim_l. Were reduced them into 3802 x 3802.) We've also reduced the number of loop iterations from 2400 to 1200. Then we've conducted the following two runs:

– Using 8 threads, the original executable runs in 1431 sec and the new one runs in 624 sec, resulting in 2.29x speed-up.

– Using 16 threads, the original executable runs in 1067 sec and the new one runs in 428 sec, resulting in 2.49x speed-up.

Above results show the effectiveness of skewed tiling for 313.swim_l. Other algorithmic/compiler techniques are being sought for benchmarks which are affected more by the resource conflicts.

5 Conclusion

In this paper, we first described the architecture of an example CMP server, Sun Fire E25K, in detail. Then we introduced the OpenMP execution model along with the SPEC OMPL benchmark suite used for our performance study. We also showed how to generate highly optimized executables for SPEC OMPL using the Sun Studio 10 compiler. We then described the system settings on which we run the optimized executables of SPEC OMPL. They include features in Solaris 10 OS (MPO, MPSS) which help improve HPC application performance and binding of threads to processors. Using these features, we've measured the performance impact of the resource conflicts on CMPs for SPEC OMPL using either one core or both cores of UltraSPARC IV CMPs in the system. It turned out that the benchmarks which have high memory bandwidths requirements and/or use large amounts of memory suffer in the presence of the resource conflicts. Algorithmic and compiler techniques are needed to reduce the conflicts on the limited resources shared among different cores.

Acknowledgments. This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2006-311-D00785). The authors would like to extend their thanks to the Center for Computing and Communications of the RWTH Aachen University for allowing the accesses of Sun Fire E25K and E2900 servers.

References

1. AMD Multi-Core: Introducing x86 Multi-Core Technology & Dual-Core Processors (2005), `http://multicore.amd.com/`
2. Chaudhry, S., Caprioli, P., Yip, S., Tremblay, M.: High-Performance Throughput Computing, IEEE Micro (May-June 2005)
3. Intel Dual-Core Server Processor, `http://www.intel.com/business/bss/products/server/dual-core.htm`
4. Intel Hyperthreading Technology, `http://www.intel.com/technology/hyperthread/index.htm`
5. Kalla, R., Sinharoy, B., Tendler, J.: IBM POWER5 chip: a dual core multithreaded processor, IEEE Micro (March-April 2004)
6. Li, Y., Brooks, D., Hu, Z., Shadron, K.: Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In: 11th International Symposium on High-Performance Computer Architecture (2005)
7. Li, Z.: Optimal Skewed Tiling for Cache Locality Enhancement. In: International Parallel and Distributed Processing Symposium (IPDPS'03) (2003)

8. Olukotun, K., et al.: The Case for a single Chip-Multiprocessor. In: International Conference on Architectural Support for Programming Languages and Operating Systems (1996)
9. OpenMP Architecture Review Board, `http://www.openmp.org`
10. Solaris 10 Operating System, `http://www.sun.com/software/solaris`
11. The SPEC OMP benchmark suite, `http://www.spec.org/omp`
12. Sun Fire E25K server,
`http://www.sun.com/servers/highend/sunfire_e25k/index.xml`
13. Sun Studio 10 Software,
`http://www.sun.com/software/products/studio/index.html`
14. Sun UltraSPARC T1 microprocessor, `http://www.sun.com/processors/UltraSPARC-T1`
15. Tullsen, D., Eggers, S., Levy, H.: Simultaneous MultiThreading: Maximizing On-Chip Parallelism. In: International Symposium on Computer Architecture (1995)

An Implementation of Parallel 1-D FFT Using SSE3 Instructions on Dual-Core Processors

Daisuke Takahashi

Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
daisuke@cs.tsukuba.ac.jp

Abstract. In the present paper, an implementation of a parallel one-dimensional fast Fourier transform (FFT) using Streaming SIMD Extensions 3 (SSE3) instructions on dual-core processors is proposed. Combination of vectorization and the block six-step FFT algorithm is shown to effectively improve performance. The performance results for one-dimensional FFTs on dual-core Intel Xeon processors are reported. We successfully achieved performance of approximately 2006 MFLOPS on a dual-core Intel Xeon PC (2.8 GHz, two CPUs, four cores) and approximately 3492 MFLOPS on a dual-core Intel Xeon 5150 PC (2.66 GHz, two CPUs, four cores) for a 2^{20}-point FFT.

1 Introduction

The fast Fourier transform (FFT) [1] is an algorithm widely used today in science and engineering.

Today, a number of processors have short vector SIMD instructions, e.g., Intel's SSE/SSE2/SSE3, AMD's 3DNow!, and Motorola's Altivec. These instructions provide substantial speedup for digital signal processing applications. Efficient fast Fourier transform (FFT) implementations with short vector SIMD instructions have been investigated thoroughly [2,3,4,5,6,7].

Many FFT algorithms work well when the data sets fit into the cache. However, when a problem size exceeds the cache size, the performance of these FFT algorithms decreases dramatically. The key issue in the design of large FFTs is minimizing the number of cache misses. Thus, both vectorization and high cache utilization are particularly important with respect to high performance on processors that have short vector SIMD instructions.

In the present paper, an implementation of parallel one-dimensional FFT using Streaming SIMD Extensions 3 (SSE3) instructions on dual-core processors is proposed.

A block six-step FFT-based parallel one-dimensional FFT has been implemented on dual-core Intel Xeon processors and the resulting performance is reported herein.

Section 2 describes a vectorization of FFT kernels. Section 3 describes a block six-step FFT algorithm used for problems that exceed the cache size. Section 4 describes the in-cache FFT algorithm used for problems that fit into a data

```
#include <pmmintrin.h>

static __inline __m128d ZMUL(__m128d a, __m128d b)
{
    __m128d ar, ai;

    ar = _mm_movedup_pd(a);         /* ar = [a.r a.r] */
    ar = _mm_mul_pd(ar, b);         /* ar = [a.r*b.r a.r*b.i] */
    ai = _mm_unpackhi_pd(a, a);     /* ai = [a.i a.i] */
    b = _mm_shuffle_pd(b, b, 1);    /* b = [b.i b.r] */
    ai = _mm_mul_pd(ai, b);         /* ai = [a.i*b.i a.i*b.r] */

    return _mm_addsub_pd(ar, ai); /* [a.r*b.r-a.i*b.i a.r*b.i+a.i*b.r] */
}
```

Fig. 1. An example of complex multiplication using SSE3 intrinsics

cache, and parallelization. Section 5 gives performance results. In section 6, we provide some concluding remarks.

2 Vectorization of FFT Kernels

The SSE3 instructions were introduced into the IA-32/EM64T architecture in Intel Pentium 4 and Xeon processors [8]. These extensions are designed to enhance the performance of IA-32/EM64T processors.

The most direct way to use the SSE3 instructions is to insert the assembly language instructions inline into the source code. However, this can be time-consuming and tedious, and assembly language inline programming is not supported on all compilers. Instead, Intel provides easy implementation through the use of API extension sets referred to as intrinsics [9]. The SSE3 intrinsics were used to access the SIMD hardware. In the SSE3 instructions, new instructions (`addsubps`, `addsubpd`, `movsldup`, `movshdup` and `movddup`) are designed to improve performance with respect to complex arithmetic. An example of complex multiplication using the SSE3 intrinsics is shown in Fig. 1.

The `__m128d` data type in Fig. 1 is supported by the SSE3 intrinsics. The `__m128d` data type holds two packed double-precision floating-point values. In complex multiplication, the `__m128d` data type is used as a double-precision complex data type.

The inline function `ZMUL` in Fig. 1 can be used to multiply two double-precision complex values. Addition of two double-precision complex values can be performed using the intrinsic function `_mm_add_pd` in Fig. 1.

To vectorize FFT kernels, the SSE3 intrinsics and the inline function `ZMUL` can be used. An example of vectorized radix-2 FFT kernel using the SSE3 intrinsics is shown in Fig. 2.

```
#include <pmmintrin.h>

__m128d ZMUL(__m128d a, __m128d b);

void fft_vec(double *a, double *b, double *w, int m, int l)
{
  int i, i0, i1, i2, i3, j;
  __m128d t0, t1, w0;

  for (j = 0; j < l; j++) {
    w0 = _mm_load_pd(&w[j << 1]);
    for (i = 0; i < m; i++) {
      i0 = (i << 1) + (j * m << 1); i1 = i0 + (m * l << 1);
      i2 = (i << 1) + (j * m << 2); i3 = i2 + (m << 1);
      t0 = _mm_load_pd(&a[i0]); t1 = _mm_load_pd(&a[i1]);
      _mm_store_pd(&b[i2], _mm_add_pd(t0, t1));
      _mm_store_pd(&b[i3], ZMUL(w0, _mm_sub_pd(t0, t1)));
    }
  }
}
```

Fig. 2. An example of vectorized radix-2 FFT kernel using SSE3 intrinsics

3 A Block Six-Step FFT Algorithm

The discrete Fourier transform (DFT) is given by

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \quad 0 \le k \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n has factors n_1 and n_2 $(n = n_1 \times n_2)$, then the indices j and k can be expressed as:

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{2}$$

We can define x and y as two-dimensional arrays (in Fortran notation):

$$x_j = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{3}$$

$$y_k = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{4}$$

Substituting the indices j and k in equation (1) with those in equation (2), and using the relation of $n = n_1 \times n_2$, we can derive the following equation:

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{5}$$

We note that the above factorization was appeared in the Cooley-Tukey paper [1].

This derivation leads to the following six-step FFT algorithm [10,11]:

Step 1: Transpose

$$x_1(j_2, j_1) = x(j_1, j_2).$$

Step 2: n_1 individual n_2-point multicolumn FFTs

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1)\omega_{n_2}^{j_2 k_2}.$$

Step 3: Twiddle-factor multiplication

$$x_3(k_2, j_1) = x_2(k_2, j_1)\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 4: Transpose

$$x_4(j_1, k_2) = x_3(k_2, j_1).$$

Step 5: n_2 individual n_1-point multicolumn FFTs

$$x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1 k_1}.$$

Step 6: Transpose

$$y(k_2, k_1) = x_5(k_1, k_2).$$

We combine the multicolumn FFTs and transpositions to reduce the number of cache misses, and we modify the original six-step FFT algorithm to reuse data in the cache memory. We assume in the following that $n = n_1 n_2$ and that n_b is the block size, and assume that each processor has a multi-level cache memory. A block six-step FFT algorithm [12] can be stated as follows.

1. Consider the data in main memory as an $n_1 \times n_2$ complex matrix. Fetch and transpose the data n_b rows at a time into an $n_2 \times n_b$ matrix. The $n_2 \times n_b$ array fits into the L2 cache.
2. For each of n_b columns, perform n_b individual n_2-point multicolumn FFTs on the $n_2 \times n_b$ array in the L2 cache. Each column FFT fits into the L1 data cache.
3. Multiply the resulting data in each of the $n_2 \times n_b$ complex matrices by the twiddle factors. Then transpose each of the resulting $n_2 \times n_b$ matrices, and return the resulting n_b rows to the same locations in the main memory from which they were fetched.
4. Perform n_2 individual n_1-point multicolumn FFTs on the $n_1 \times n_2$ array. Each column FFT fits into the L1 data cache.
5. Transpose and store the resulting data on an $n_2 \times n_1$ complex matrix.

Fig. 3 gives the Fortran program for this block six-step FFT algorithm. Here the twiddle factors $\omega_{n_1 n_2}^{j_1 k_2}$ in equation 5 are stored in array `U`, and the array `WORK` is the work array. The parameters `NB` and `NP` are the blocking parameter and padding parameter, respectively.

```
COMPLEX*16 X(N1,N2),Y(N2,N1)
COMPLEX*16 U(N1,N2),WORK(N2+NP,NB)
DO II=1,N1,NB
  DO JJ=1,N2,NB
    DO I=II,II+NB-1
      DO J=JJ,JJ+NB-1
        WORK(J,I-II+1)=X(I,J)
      END DO
    END DO
  END DO
  DO I=1,NB
    CALL IN_CACHE_FFT(WORK(1,I),N2)
  END DO
  DO J=1,N2
    DO I=II,II+NB-1
      X(I,J)=WORK(J,I-II+1)*U(I,J)
    END DO
  END DO
END DO
```

```
DO JJ=1,N2,NB
  DO J=JJ,JJ+NB-1
    CALL IN_CACHE_FFT(X(1,J),N1)
  END DO
  DO I=1,N1
    DO J=JJ,JJ+NB-1
      Y(J,I)=X(I,J)
    END DO
  END DO
END DO
```

Fig. 3. A block six-step FFT algorithm

4 In-Cache FFT Algorithm and Parallelization

The Stockham autosort algorithm [13] works well until the problem size exceeds the cache size. When the problem exceeds cache size, the block six-step FFT algorithm should be used.

Table 1. Real inner-loop operations for radix-2, 4 and 8 FFT kernels based on the Stockham autosort algorithm

	Radix-2	Radix-4	Radix-8
Loads and stores	8	16	32
Multiplications	4	12	32
Additions	6	22	66
Total floating-point operations ($n \log_2 n$)	5.000	4.250	4.083
Floating-point instructions	10	34	98
Floating-point/memory ratio	1.250	2.125	3.063

The radix-2, 4 and 8 Stockham autosort algorithms were used for in-cache FFT. Table 1 shows the number of operations required for radix-2, 4 and 8 FFT kernels. Higher radices are more efficient in terms of both memory and floating-point operations. A high ratio of floating-point instructions to memory operations is particularly important in cache-based processors. In view of the high ratio of floating-point instructions to memory operations, the radix-8 FFT is preferable to the radix-4 FFT.

Table 2. Specification of machines

Platform	Intel Xeon PC	Intel Xeon 5150 PC
Number of CPUs	2	2
Number of cores	4	4
CPU Type	Intel Xeon Paxville 2.8 GHz	Intel Xeon 5150 Woodcrest 2.66 GHz
L1 Cache	I-Cache: 12 KB D-Cache: 16 KB	I-Cache: 32 KB D-Cache: 32 KB
L2 Cache	2 MB	4 MB
Main Memory	DDR2-SDRAM 2 GB	DDR2-SDRAM 4 GB
Chipset	Intel E7525	Intel 5000X
OS	Linux 2.6.9-5.ELsmp	Linux 2.6.18-1.2798.fc6

Table 3. Performance of FFTE 4.0 (SSE3) on a dual-core Intel Xeon (Paxville 2.8 GHz) PC

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00014	1725.26	0.00014	1725.25	0.00014	1725.98
2^{13}	0.00029	1820.93	0.00029	1818.07	0.00029	1817.15
2^{14}	0.00062	1835.34	0.00062	1835.01	0.00062	1835.28
2^{15}	0.00138	1774.97	0.00139	1773.74	0.00139	1773.99
2^{16}	0.00444	1180.03	0.00252	2082.55	0.00161	3259.36
2^{17}	0.01053	1057.67	0.00622	1790.11	0.00464	2398.53
2^{18}	0.02339	1008.77	0.01537	1534.60	0.01135	2079.42
2^{19}	0.04917	1013.06	0.03177	1567.96	0.02387	2086.19
2^{20}	0.10258	1022.20	0.06493	1615.04	0.05227	2006.10

Although higher radix FFTs require more floating-point registers in order to hold intermediate results, the Intel Xeon EM64T processor has 16 XMM registers.

A power-of-two point FFT (except for 2-point FFT) can be performed by a combination of radix-8 and radix-4 steps containing at most two radix-4 steps. That is, the power-of-two FFTs can be performed as a length $n = 2^p = 4^q 8^r$ ($p \geq 2,\ 0 \leq q \leq 2,\ r \geq 0$).

We parallelized the block six-step FFT by using OpenMP. The outermost loop of each FFT algorithm shown in Fig. 3 is distributed across the processors.

5 Performance Results

To evaluate the implemented parallel one-dimensional FFT, referred to as FFTE (version 4.0), its performance was compared to that of the FFT library of FFTW (version 3.1.2, using Posix threads) [14]. Since the latest Intel Math Kernel Library (MKL, version 9.0) [15] does not support threaded parallel one-dimensional FFT, the MKL was not evaluated.

Table 4. Performance of FFTE 4.0 (x87) on a dual-core Intel Xeon (Paxville 2.8 GHz) PC

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00019	1268.72	0.00019	1268.66	0.00019	1268.28
2^{13}	0.00040	1335.32	0.00040	1335.31	0.00040	1335.11
2^{14}	0.00087	1316.86	0.00087	1316.93	0.00087	1316.55
2^{15}	0.00192	1277.66	0.00192	1281.89	0.00192	1280.70
2^{16}	0.00613	854.67	0.00358	1463.97	0.00256	2045.75
2^{17}	0.01465	760.70	0.00877	1270.93	0.00623	1789.44
2^{18}	0.03310	712.84	0.02107	1119.66	0.01569	1503.96
2^{19}	0.06844	727.79	0.04444	1120.87	0.03244	1535.26
2^{20}	0.13874	755.77	0.08833	1187.13	0.06782	1546.16

Table 5. Performance of FFTW 3.1.2 on a dual-core Intel Xeon (Paxville 2.8 GHz) PC

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00009	2832.57	0.00009	2844.51	0.00009	2836.96
2^{13}	0.00019	2822.81	0.00019	2848.04	0.00019	2825.57
2^{14}	0.00041	2826.76	0.00041	2814.07	0.00041	2823.65
2^{15}	0.00090	2736.13	0.00090	2728.64	0.00098	2501.34
2^{16}	0.00235	2228.51	0.00235	2228.43	0.00232	2259.15
2^{17}	0.00633	1760.64	0.00561	1985.35	0.00555	2009.27
2^{18}	0.01386	1701.82	0.01201	1964.44	0.00977	2415.21
2^{19}	0.03083	1615.66	0.02465	2020.35	0.02088	2385.71
2^{20}	0.06792	1543.76	0.05136	2041.79	0.04406	2380.10

The elapsed times obtained from 10 executions of complex forward FFTs were averaged, where the input and output were in the usual order. The FFTs were performed on double-precision complex data, and the table for twiddle factors was prepared in advance.

All routines were written in C and FORTRAN 77. The specifications for the two platforms used are shown in Table 2.

The compilers used were the Intel C Compiler (`icc`, version 9.1) and the Intel Fortran Compiler (`ifort`, version 9.1). For the FFTE (SSE3), the compiler options used were specified as "`icc -O3 -xP`" and "`ifort -O3 -xP -openmp`". For the FFTE (x87), the compiler options used were specified as "`ifort -O3 -march=pentiumpro -openmp`". For the FFTW, the compiler options used were specified as "`icc -O3 -xP`" and "`ifort -O3 -xP`". All programs were run in 64-bit mode.

The FFTE routines are single-threaded for small problem sizes and multi-threaded for large problem sizes.

Table 6. Performance of FFTE 4.0 (SSE3) on a dual-core Intel Xeon 5150 (Woodcrest 2.66 GHz) PC

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00006	4128.46	0.00006	4128.80	0.00006	4141.40
2^{13}	0.00014	3912.61	0.00014	3900.81	0.00014	3925.46
2^{14}	0.00028	4030.83	0.00029	4020.14	0.00028	4036.37
2^{15}	0.00060	4121.60	0.00060	4113.43	0.00060	4106.24
2^{16}	0.00143	3676.79	0.00141	3713.05	0.00141	3717.98
2^{17}	0.00500	2228.17	0.00380	2931.55	0.00226	4921.67
2^{18}	0.01340	1761.12	0.00747	3159.97	0.00472	4995.93
2^{19}	0.02989	1666.54	0.01678	2968.24	0.01341	3715.39
2^{20}	0.06675	1570.84	0.03735	2807.18	0.03003	3491.69

Table 7. Performance of FFTE 4.0 (x87) on a dual-core Intel Xeon 5150 (Woodcrest 2.66 GHz) PC

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00010	2414.28	0.00010	2414.04	0.00010	2413.01
2^{13}	0.00022	2475.71	0.00021	2480.89	0.00021	2480.22
2^{14}	0.00047	2463.42	0.00047	2464.68	0.00047	2465.73
2^{15}	0.00099	2470.15	0.00100	2463.51	0.00099	2472.63
2^{16}	0.00237	2214.24	0.00237	2214.47	0.00237	2212.98
2^{17}	0.00741	1503.71	0.00486	2290.17	0.00275	4053.13
2^{18}	0.01848	1276.75	0.00977	2415.99	0.00591	3992.05
2^{19}	0.03901	1276.83	0.02123	2346.35	0.01546	3222.04
2^{20}	0.08344	1256.70	0.04492	2334.48	0.03344	3135.95

Table 8. Performance of FFTW 3.1.2 on a dual-core Intel Xeon 5150 (Woodcrest 2.66 GHz) PC

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00005	5340.65	0.00005	5324.67	0.00005	5370.66
2^{13}	0.00010	5246.95	0.00010	5250.29	0.00010	5321.29
2^{14}	0.00022	5288.20	0.00022	5238.77	0.00022	5331.34
2^{15}	0.00050	4959.85	0.00050	4955.32	0.00049	4971.35
2^{16}	0.00111	4722.97	0.00110	4782.46	0.00119	4405.83
2^{17}	0.00376	2963.07	0.00400	2785.17	0.00396	2811.49
2^{18}	0.00997	2366.58	0.01011	2333.85	0.01000	2358.56
2^{19}	0.02351	2118.26	0.02288	2177.28	0.02166	2299.11
2^{20}	0.05060	2072.33	0.04003	2619.43	0.03698	2835.30

5.1 Performance Results on a Dual-Core Intel Xeon PC

Tables 3, 4 and 5 compare the FFTE (SSE3 and x87) and the FFTW in terms of their run times and MFLOPS. The first column shows the number of points of FFTs. The next six columns indicate the average elapsed times in seconds and the average execution performances in MFLOPS. The MFLOPS values are each based on $5n \log_2 n$ for a transform of size of $n = 2^m$.

Tables 3 and 4 show that the SSE3 instructions provide performance enhancement for the FFTE.

On the other hand, the FFTE (SSE3) is slower than the FFTW except for the cases of $n = 2^{16}$ and $n = 2^{17}$ (two CPUs, four cores), as shown in Tables 3 and 5. This is because the L2 cache size of the Intel Xeon processor is 2 MB and it holds a low cache-miss ratio up to size 2^{15} points. Moreover, the FFTW works well when the data sets fit into the cache. These are two reasons why for $n \leq 2^{15}$ the FFTE (SSE3) is slower than the FFTW.

We found that the dual-core Intel Xeon 2.8 GHz PC does not have good scalability continuing up to two CPUs (four cores). This is mainly because the memory bandwidth of the Intel E7525 Chipset is still 6.4 GB/s.

5.2 Performance Results on a Dual-Core Intel Xeon 5150 PC

Tables 6, 7 and 8 compare the FFTE (SSE3 and x87) and the FFTW in terms of their run times and MFLOPS. The first column shows the number of points of FFTs. The next six columns indicate the average elapsed times in seconds and the average execution performances in MFLOPS. The MFLOPS values are each based on $5n \log_2 n$ for a transform of size of $n = 2^m$.

The FFTE (SSE3) is faster than the FFTW except for the cases of $n \leq 2^{20}$ (one CPU, one core) and $n \leq 2^{16}$ (two CPUs, four cores), as shown in Tables 6 and 8. The speedup of the FFTE is better than that of the FFTW.

The performance of the implemented parallel one-dimensional FFT remains at a high level, even for the larger problem size, because of cache blocking. The FFTE exploits the SSE3 new instructions (`addsubpd` and `movddup`). These are two reasons why the FFTE is more advantageous than the FFTW.

6 Conclusion

In the present paper, the implementation of a parallel one-dimensional FFT using SSE3 instructions on dual-core processors was proposed. In addition, FFT kernels were vectorized using the SSE3 instructions, and the block six-step FFT was parallelized using OpenMP.

As a result of cache blocking, the performance of the implemented parallel one-dimensional FFT remains high even for larger problems. The speedup of the FFTE is better than that of the FFTW.

These results demonstrate that the implemented FFT utilizes cache memory effectively. We successfully achieved performance of approximately 2006 MFLOPS on a dual-core Intel Xeon PC (2.8 GHz, two CPUs, four cores) and approximately 3492 MFLOPS on a dual-core Intel Xeon 5150 PC (2.66 GHz, two CPUs, four cores) for a 2^{20}-point FFT.

References

1. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. 19, 297–301 (1965)
2. Nadehara, K., Miyazaki, T., Kuroda, I.: Radix-4 FFT implementation using SIMD multimedia instructions. In: Proc. 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '99), vol. 4, pp. 2131–2134 (1999)
3. Franchetti, F., Karner, H., Kral, S., Ueberhuber, C.W.: Architecture independent short vector FFTs. In: Proc. 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001), vol. 2, pp. 1109–1112 (2001)
4. Rodriguez, V.P.: A radix-2 FFT algorithm for modern single instruction multiple data (SIMD) architectures. In: Proc. 2002 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2002), vol. 3, pp. 3220–3223 (2002)
5. Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C.W.: SIMD vectorization of straight line FFT code. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 251–260. Springer, Heidelberg (2003)
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE 93, 216–231 (2005)
7. Franchetti, F., Kral, S., Lorenz, J., Ueberhuber, C.W.: Efficient utilization of SIMD extensions. Proc. IEEE 93, 409–425 (2005)
8. Intel Corporation: IA-32 Intel Architecture Software Developer's Manual. Basic Architecture, vol. 1 (2006)
9. Intel Corporation: Intel C++ Compiler Documentation (2006)
10. Bailey, D.H.: FFTs in external or hierarchical memory. The Journal of Supercomputing 4, 23–35 (1990)
11. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia (1992)
12. Takahashi, D.: A blocking algorithm for FFT on cache-based processors. In: Hertzberger, B., Hoekstra, A.G., Williams, R. (eds.) High-Performance Computing and Networking. LNCS, vol. 2110, pp. 551–554. Springer, Heidelberg (2001)
13. Swarztrauber, P.N.: FFT algorithms for vector computers. Parallel Computing 1, 45–63 (1984)
14. Frigo, M., Johnson, S.G.: FFTW, `http://www.fftw.org`
15. Intel Corporation: Intel Math Kernel Library Reference Manual (2005)

Author Index