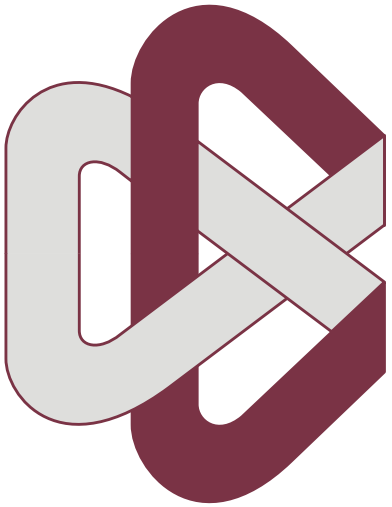


CIMAT

Brevísimo tutorial de OpenMP

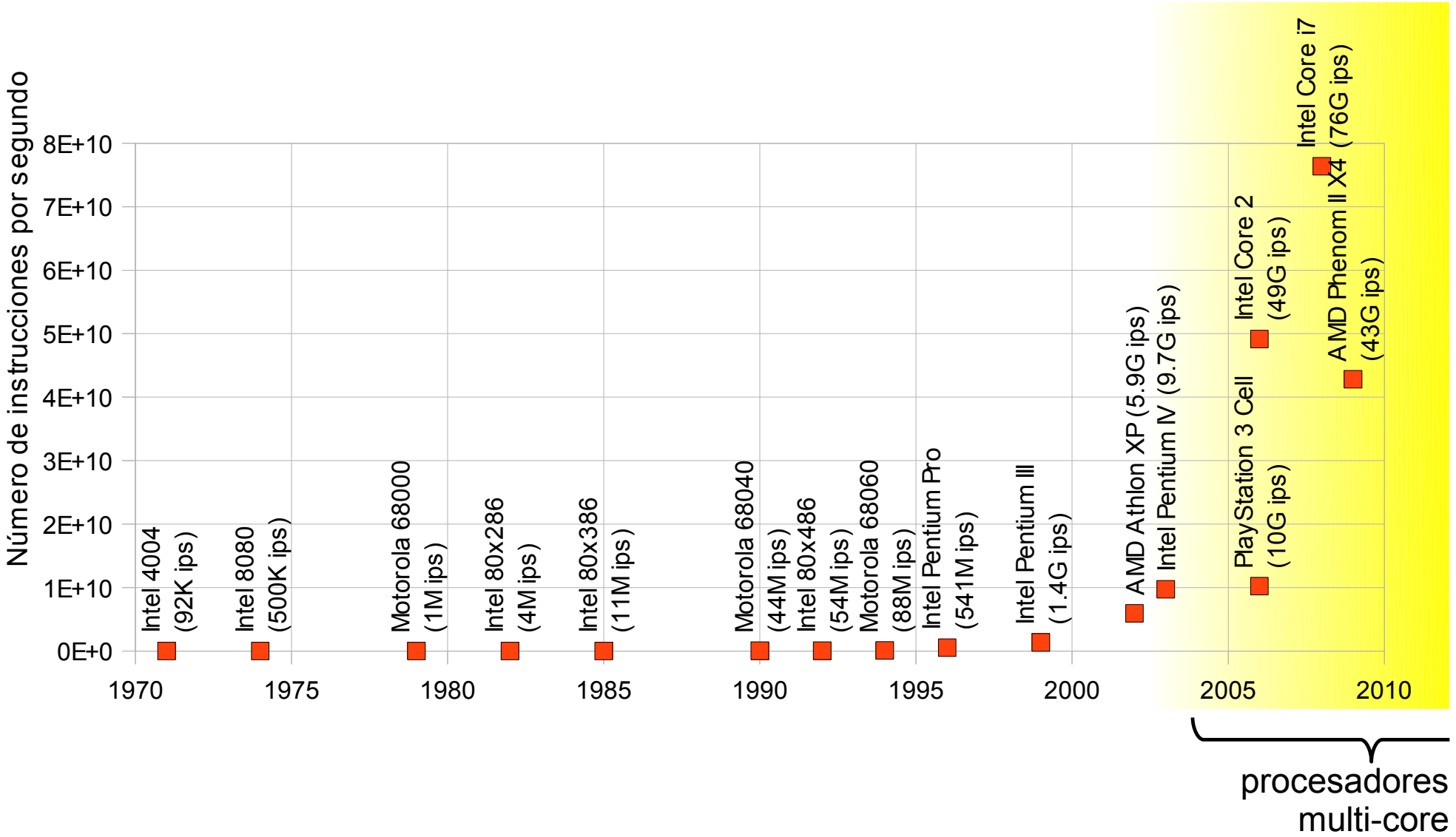
MSc Miguel Vargas-Félix
miguelvargas@cimat.mx
http://www.cimat.mx/~miguelvargas

Contenido

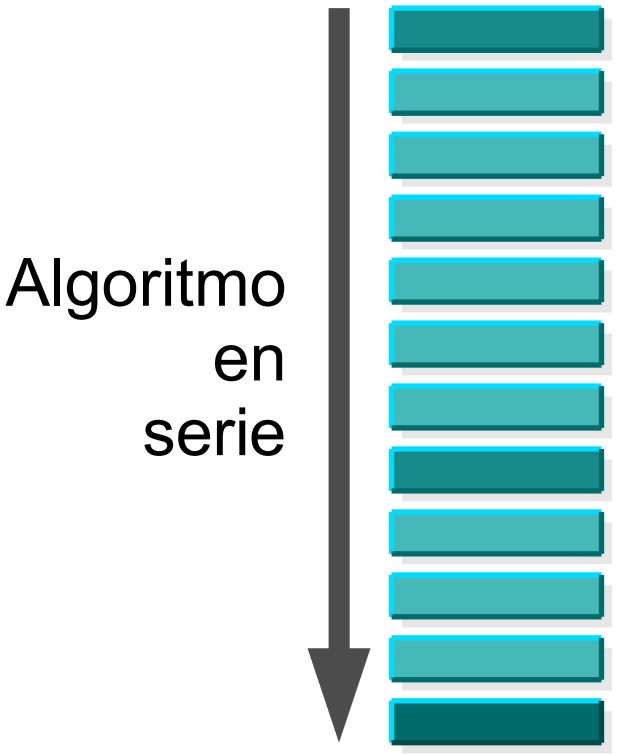
Cómputo en paralelo
Operaciones matemáticas en paralelo
Evolución de los procesadores de propósito general
El cache
El esquema OpenMP
Pérdida de eficiencia con el aumento de procesadores
Post data: Optimizar el código compilado
¿Preguntas?
Para saber más...

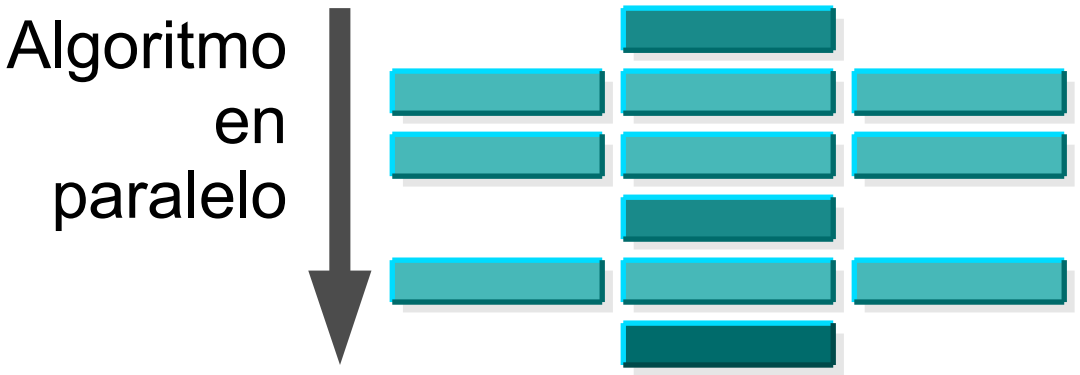
Cómputo en paralelo

La evolución de la velocidad de los procesadores



Tenemos ahora que desarrollar nuevos esquemas para diseñar algoritmos





La meta es ***reducir tiempo*** necesario para realizar una secuencia de instrucciones.

La buena noticia: Hacer programas en paralelo con **OpenMP** es muy sencillo.

La mala noticia: Hacer programas en paralelo eficientes requiere un esfuerzo extra.

Operaciones matemáticas en paralelo

Fácilmente paralelizables

Esto significa que pueden separarse en varias sub-operaciones que puede realizarse de forma independiente. Por ejemplo, el la suma de dos vectores $\boldsymbol{x}$, $\boldsymbol{y}$ para producir otro vector $\boldsymbol{c}$,

$$c_i = x_i + y_i,\ i = 1, \ldots, N.$$

En este caso las N sumas pueden realizarse simultáneamente, asignando una a cada procesador. Lo que hay que resaltar es que no hay dependencia entre los diferentes pares de datos, tenemos entonces el paralelismo más eficiente.

No tan fáciles de paralelizar

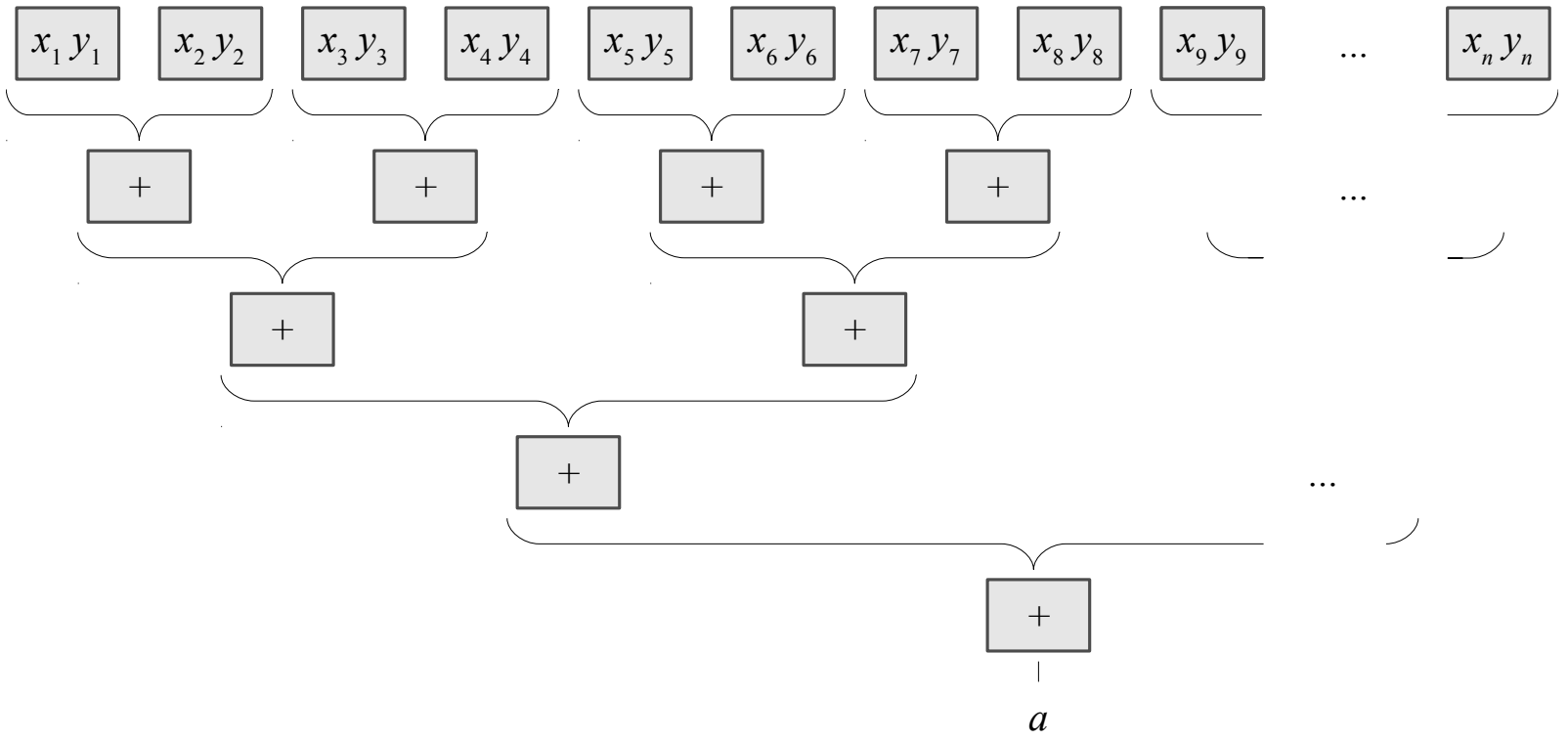
Por ejemplo el producto punto $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$

$$a = \sum_{i=1}^{N} x_i y_i,$$

donde a es un escalar, una primera aproximación sería verlo como una secuencia de sumas de productos que requieren irse acumulando, al verlo así no es una operación paralelizable.

Sin embargo, podemos reorganizar el proceso como se muestra en la figura:

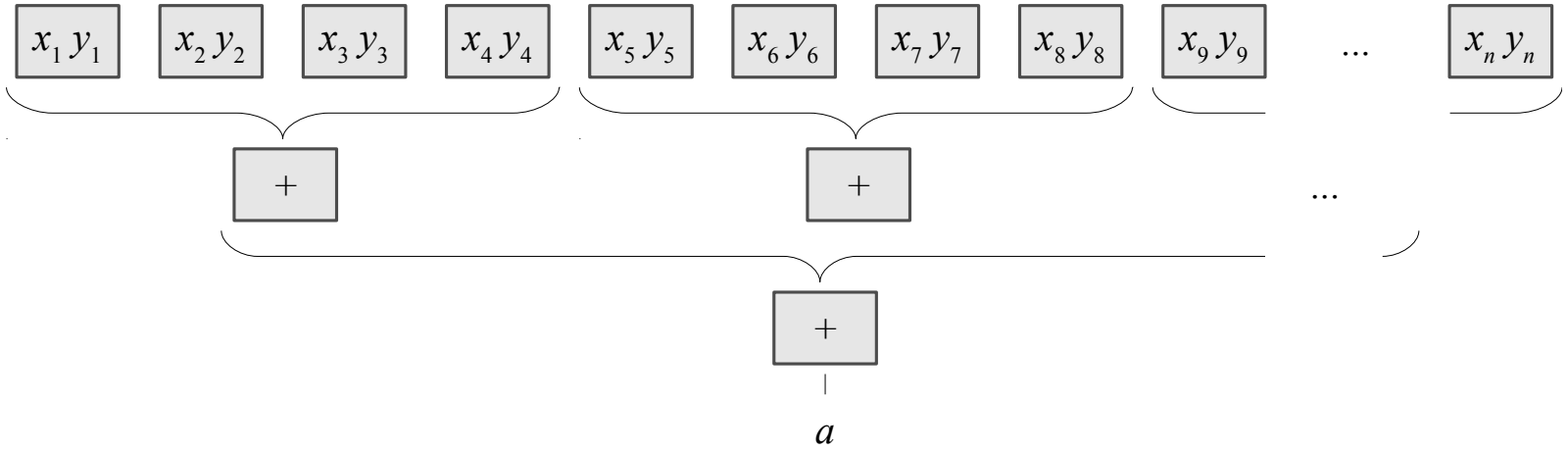
$$a = \sum_{i=1}^{N} x_i y_i$$



En este caso se tiene una paralelización eficiente de la multiplicación de las entradas de los vectores, después se va reduciendo la eficiencia. Muchos algoritmos seriales requieren, para ser paralelizados, de re-ordenar las operaciones con una estrategia de “divide y vencerás”, como en el caso del producto punto.

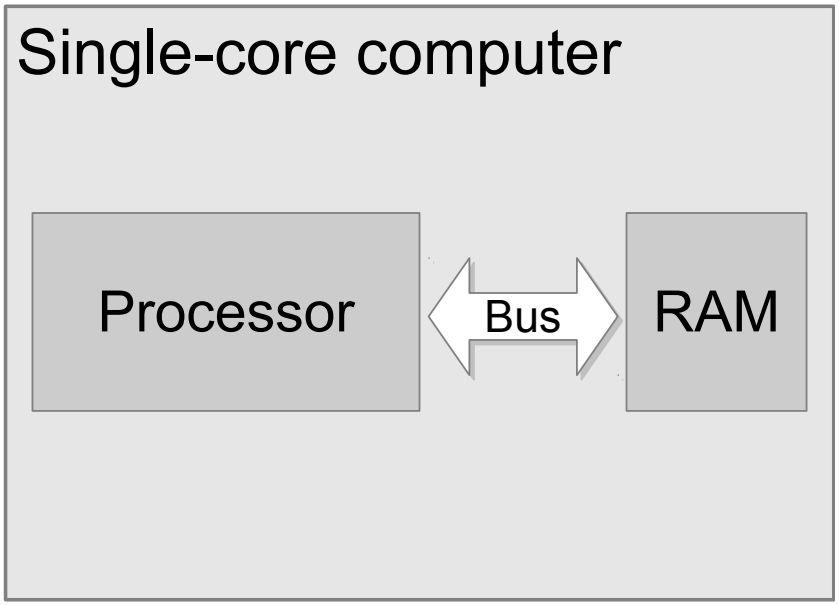
Usualmente se tendrán memos procesadores que el tamaño del vector, por lo que se asignan varias operaciones de un grupo a cada procesador, las cuales se ejecutarán en secuencia, lo que limita la eficiencia del paralelismo.

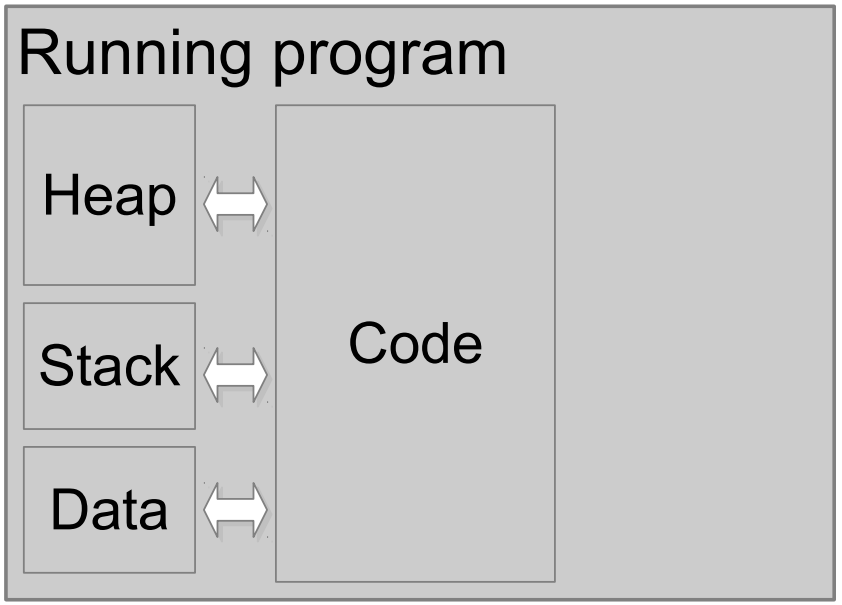
$$a = \sum_{i=1}^{N} x_i y_i$$



Evolución de los procesadores de propósito general

Hace unos 30+ años



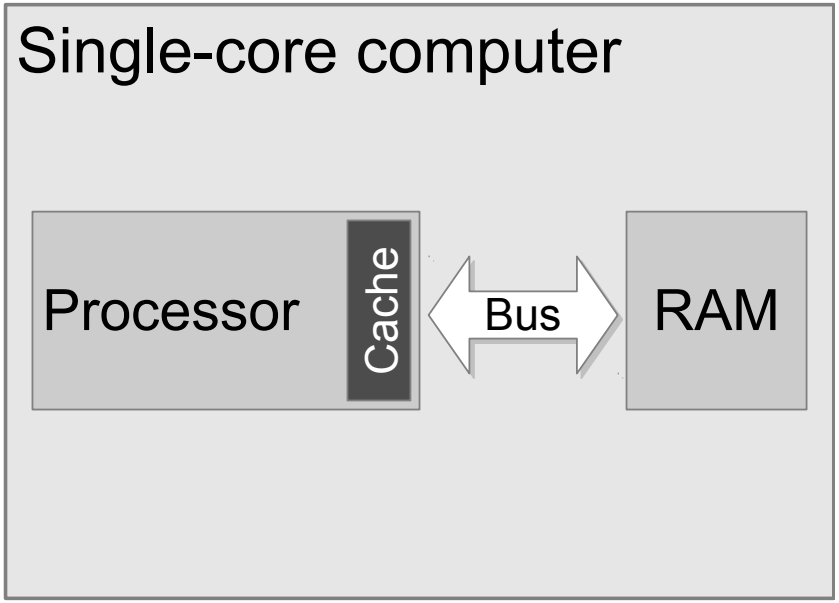


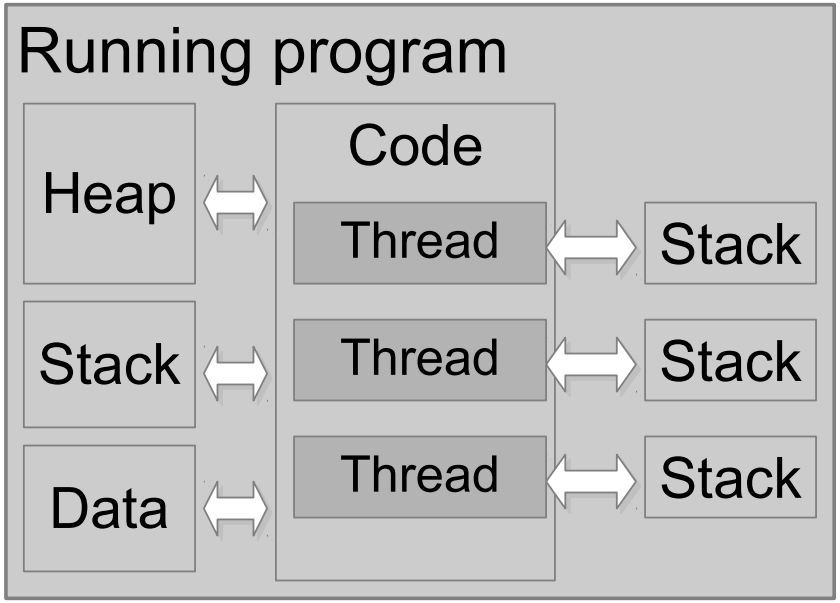
Desde un punto de vista físico muy simplificado, la computadora contiene un procesador (CPU) que accesa a la memoria (RAM) a través de un bus de datos/direcciones.

La velocidad de los procesadores y la memoria (RAM) era equivalente.

Desde el punto de vista de la ejecución de un programa, la memoria está dividida en *Heap* (o montículo), *Stack* (pila) donde se almacenan las variables declaradas dentro de una función y *Data* (región donde se almacenan las variables globales).

Hace unos 20+ años





La velocidad de los procesadores es mayor que la de la memoria RAM. Se introduce en el procesador una memoria intermedia o *cache* de alta velocidad. El tamaño del *cache* suele ser de 1/1000 del tamaño de la RAM.

Los programas ejecutan varios hilos o *threads* en un pseudo-paralelismo.

Cada *thread* es una secuencia de instrucciones que se ejecuta de forma independiente. Un programa puede tener uno o más threads.

Los *threads* comparten las regiones de memoria *Heap* y *Data*, y además trabajan con un *stack* independiente para cada uno.

El *cache*

La velocidad de operación de los procesadores es mucho mayor que la velocidad de acceso de la memoria RAM de las computadoras [Wulf95], esto es debido a que es muy costoso fabricar memoria de alta velocidad.

Para solventar esta diferencia en velocidad, los procesadores modernos incluyen memoria *cache*, en diferentes niveles (L1, L2 y en algunos casos L3). Estas memoria *cache* son de alta velocidad aunque de menor capacidad que la memoria RAM del sistema. Su función es la de leer de forma anticipada memoria RAM mientras el *core* está trabajando y modificar la información leida de forma local. Cuando entra nueva información al cache la información ya procesada es almacenada en la memoria RAM.

La siguiente tabla muestra los ciclos de reloj de CPU necesarios para accesar cada tipo de memoria en un procesador Pentium M:

Acceso a	Ciclos
Registro CPU	≤ 1
L1	~ 3
L2	~ 14
Memoria RAM	~ 240

Eso de *caches* es más eficiente cuando se leen o escriben localidades de memoria continuas [Drep07].

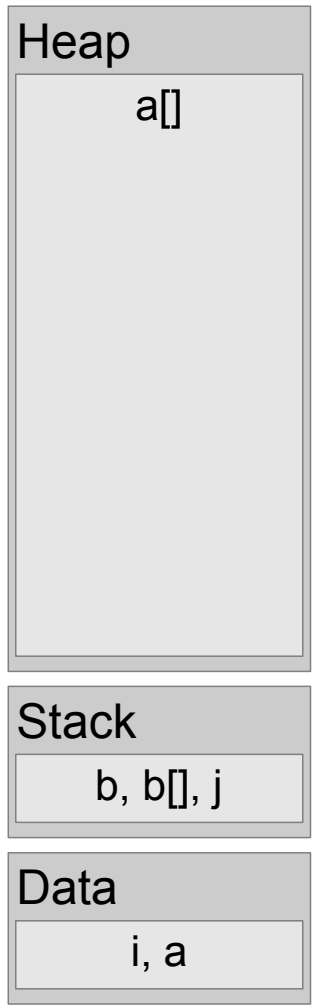
Para trabajar eficientemente con el *cache* es necesario que los datos con los cual trabaja el programa sean puestos de forma continua en la memoria RAM. Las variables deben ser declaradas dentro de la función en la cual serán utilizados, así estas estarán almacenados en el *stack*, lo cual facilitará que caigan en el *cache* L1 [Fog10].

```
#define S 2000

int i;
double* a = (double*)malloc(S*sizeof(double));

void funcion(void)
{
   double b[S];
   for (i = 0; i < S; ++i)
      b[i] = i*2.0;

   for (int j = 0; j < S; ++j) {
      b[j] += a[j];
      a[j] = 0;
   }
}
```



El programa anterior accesa localidades de memoria distantes.

El *cache* lee la memoría RAM en por líneas o bloques de N bytes contiguos (N ~ 64). Los circuítos del *cache* están diseñados para hacer esta lectura de forma mucho más rápida que haciendo lecturas a regiones no contiguas.

Una línea de cache se accesa más rápido si se la lee de forma secuencial hacia adelante (i++). Leerlo en forma inversa es un poco más lento (i--).

¿Nota usted la diferencia?

```
#include <stdio.h>

#define ROWS 10000000
#define COLS 200

int main(void)
{
    char* A = new char[ROWS*COLS];

    for (int j = 0; j < COLS; ++j)
    {
        for (int i = 0; i < ROWS; ++i)
        {
            A[i*COLS + j] = i + j;
        }
    }

    delete [] A;

    return 0;
}
```

126.760s

```
#include <stdio.h>

#define ROWS 10000000
#define COLS 200

int main(void)
{
    char* A = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLS; ++j)
        {
            A[i*COLS + j] = i + j;
        }
    }

    delete [] A;

    return 0;
}
```

6.757s

¡El código de la izquierda tardó 18.76 veces más que el de la derecha!

Desde hace unos 12+ años

Ahora cada *thread* se ejecuta en un *core*. Tenemos un paralelismo real.

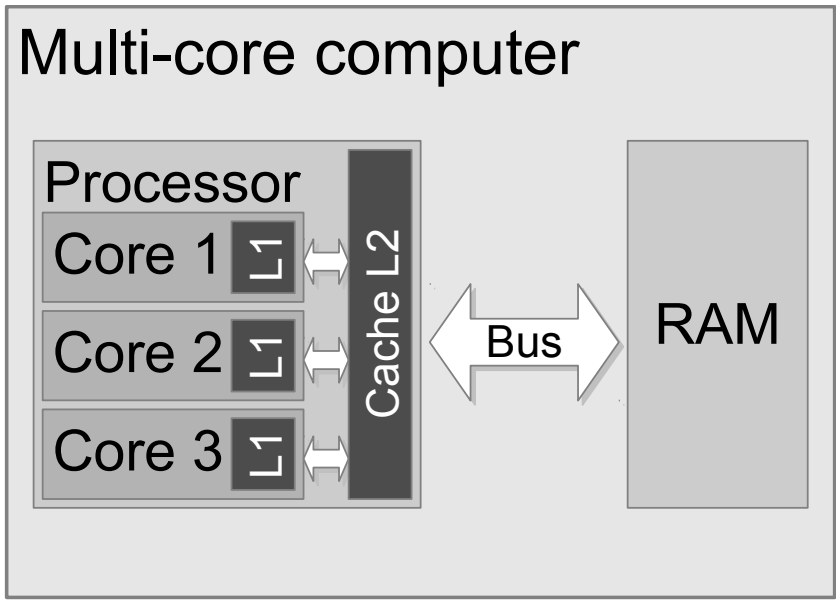
El principal cuello de botella es en el acceso a la RAM.

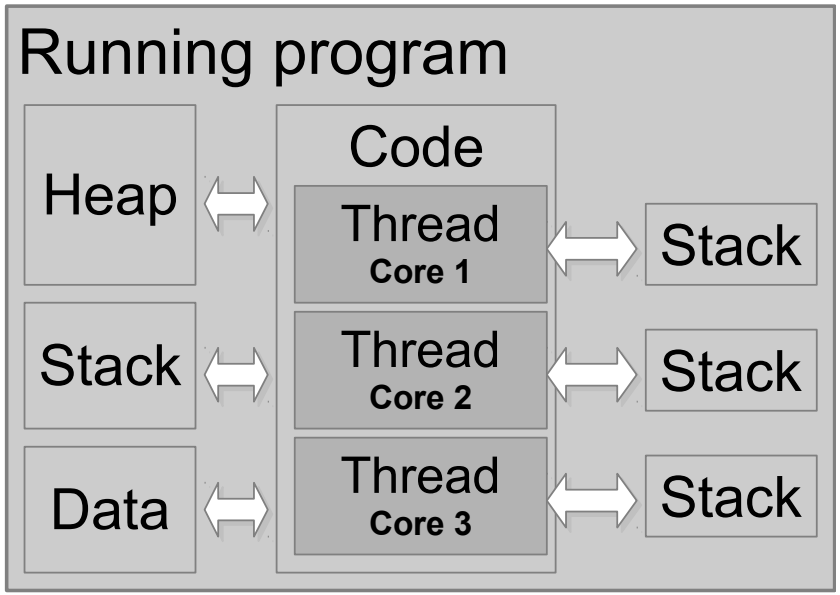
Sólo existe un canal de comunicación (bus de datos/direcciones) con la RAM.

Si dos cores tratan de leer de la RAM uno tendrá que esperar a que el otro termine.

Esto será más y mas notorio cuando el sistema tenga más procesadores.

Para reducir ésta latencia se le ha agregado otro nivel de cache, llamado L2.



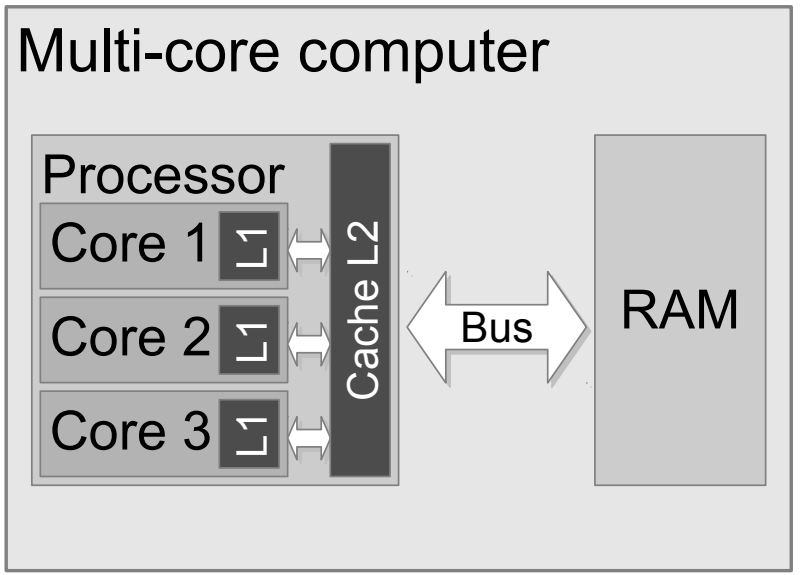


El *cache* con *multi-core*

El problema aumenta con del esquema de procesamiento en paralelo con memoria compartida.

Si un core trata de actualizar memoria de la cual existe copia en el cache de otro core, entonces se crea un ***cache-fault*** y es necesario que uno de los cores accese a L2 o la RAM para sincronizarse. Esto es muy costo.

Si un core trata de accesar un dato de memoria que ya está en su caché se le llama ***cache-hit***.



There is not free lunch

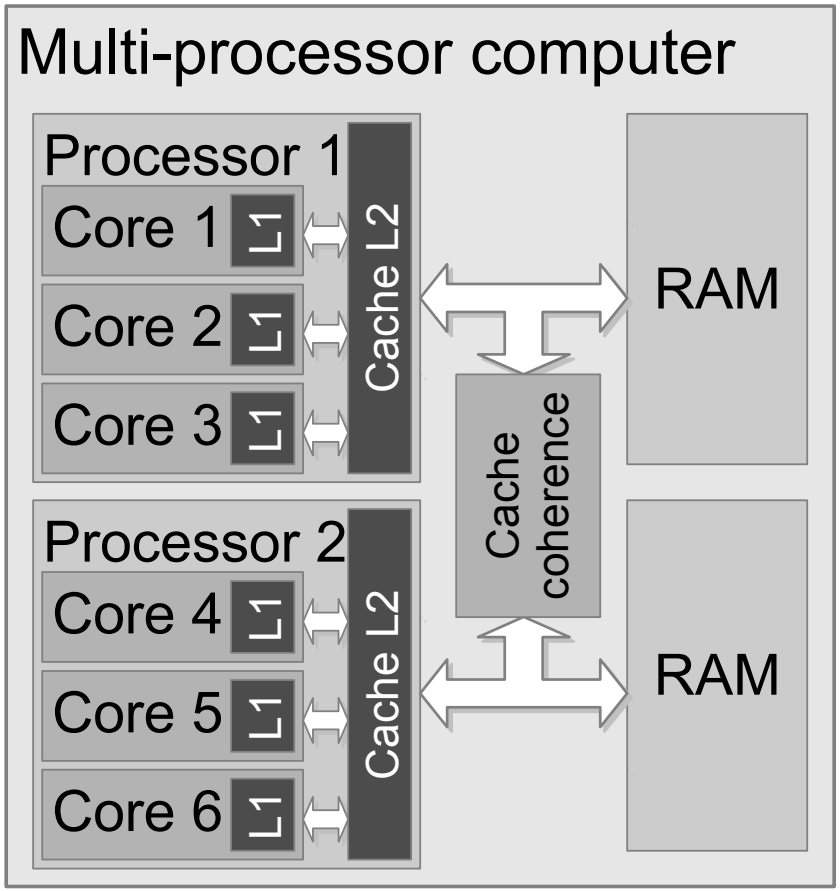
Mantener coherencia en la información cuando varios *cores* accesan la misma memoria RAM es complejo. Los detalles se pueden consultar en [Drep07].

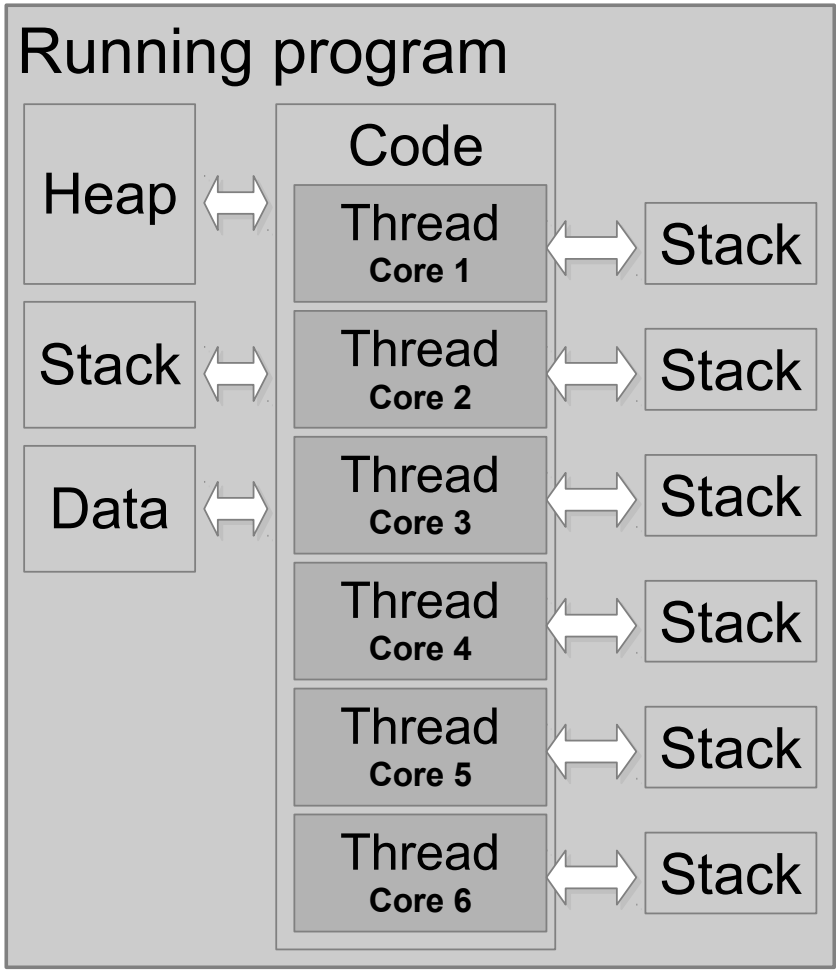
Estrategias para mantener a cada *core* accesando lo más posible su propio *cache*

- Operar con variables en el *stack.*
- No guardar variables en *data*.
- Reducir el número de escrituras al heap, realizar operaciones en el stack y solo escribir el resultado en el heap.
- Trabajar cada core con bloques de memoria contiguos.
- Hacer que *core* trabaje en localidades de memoria diferentes.
- Funciones que se usan juntas deben almacenarse cerca (en .lib o .a, no en .dll o .so)

Hace unos 8+ años

Las computadoras con más de un procesador *multi-core* están diseñadas para que cada procesador accese a un banco de memoria independiente de forma rápida (NUMA, non-uniform memory access). Accesar al banco de memoria del otro procesador es más lento. Ahora se requieren de circuítos que mantengan la coherencia del cache entre todos los cores.





El esquema OpenMP

Sirve para paralelizar programas en C, C++ o Fortran en computadoras *multi-core* o *multi-thread*.

Busca que escribir código en paralelo sea sencillo.

Es un esquema de paralelización con memoria compartida (todos los procesadores accesan a la misma memoria).

Su funcionamiento interno es con threads, en el caso de sistemas POSIX se utiliza la librería de POSIX-Threads (libpthread).

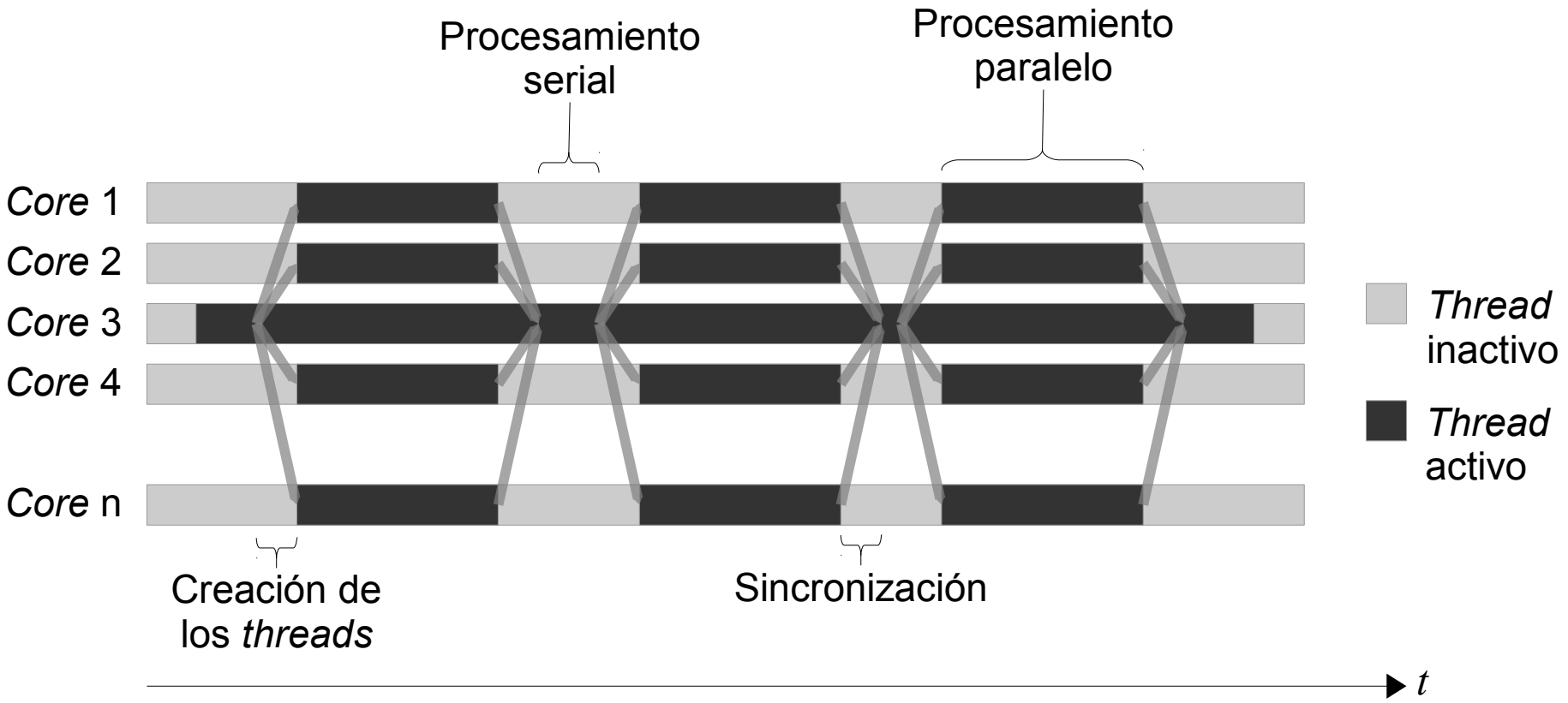
Algunos compiladores que soportan este esquema:

- GNU Compiler Collection GCC (versión >= 4.3.2)
- Intel Compiler Suite (version >= 10.1)
- Microsoft Visual Studio 2008 (Professional)
- Microsoft Visual Studio 2008 (Express) + Windows SDK for Windows Server 2008

La lista completa: http://openmp.org/wp/openmp-compilers

Programación en hilos (threads)

A la ejecución de cada secuencia de instrucciones se le conoce como un hilo de procesamiento o *thread*. Cada *thread* posee sus propios registros de control y su propio *stack* de datos, mientas que comparte el uso de la memorias del montículo (*heap*) y *data* con los demás *threads* del programa. En las computadoras *multi-core* logra la mejor eficiencia cuando cada CPU ejecuta sólo un *thread*.



Un programa simple con OpenMP

Comencemos con la paralelización de la suma de dos vectores:

```
void Suma(double* a, double* b, double* c, int size)
{
   for (int i = 0; i < size; ++i)
   {
      c[i] = a[i] + b[i];
   }
}
```

La paralelización sería...

...la paralelización sería:

```
void Suma(double* a, double* b, double* c, int size)
{
   #pragma omp parallel for
   for (int i = 0; i < size; ++i)
   {
      c[i] = a[i] + b[i];
   }
}
```

Para compilar con GCC es necesario agregar:

```
gcc -o programa -fopenmp main.c
```

En Visual C++ hay una opción en las preferencias del proyecto para activar OpenMP.

¿Cómo funciona?

Supongamos que size = 30, si la computadora tiene 3 procesadores, el código se ejecutaría como:

```
void Suma(double* a, double* b, double* c, int size)
{
   #pragma omp parallel for
   for (int i = 0; i < 10; ++i) for (int i = 10; i < 20; ++i)  for (int i = 20; i < 30; ++i)
   {                            {                              {
      c[i] = a[i] + b[i];          c[i] = a[i] + b[i];            c[i] = a[i] + b[i];
   }                            }                              }
}
```

¿Cómo se define el número de procesadores/threads a utilizar?

- Por default el número de threads es igual al número de cores en la computadora.
- Se puede establecer por medio de variables de ambiente
- Se puede definir en el código en *run-time*.

Establecer número de threads con variables de ambiente

Se hace a través de la variable de ambiente OMP_NUM_THREADS

En Linux/Unix (Bash):

```
export OMP_NUM_THREADS=3
programa
```

En Windows:

```
set OMP_NUM_THREADS=3
programa.exe
```

Establecer número de threads con código

Es necesario incluir el header “omp.h”, el cual incluye varias funciones para el control de threads.

```
#include <omp.h>

int threads = 3;

void Suma(double* a, double* b, double* c, int size)
{
   omp_set_num_threads(threads);

   #pragma omp parallel for
   for (int i = 0; i < size; ++i) {
      c[i] = a[i] + b[i];
   }
}
```

Reducciones

El ejemplo que mostraremos es el producto punto:

```
double ProductoPunto(double* a, double* b, int size)
{
   double c = 0;
   for (int i = 0; i < size; ++i)
      c += a[i]*b[i];
   return c;
}
```

Su paralelización sería:

```
double ProductoPunto(double* a, double* b, int size)
{
   double c = 0;
   #pragma omp parallel for reduction(+:c)
   for (int i = 0; i < size; ++i)
      c += a[i]*b[i];
   return c;
}
```

Paralelizar bloques de código

Además de paralelizar ciclos, es posible paralelizar bloques

```
#pragma omp parallel
{
   // ... codigo a ejecutar en paralelo...
   int thread = omp_get_thread_num();
}
```

omp_get_thread_num regresa el número de thread actual.

Paralelizar secciones de código

Se puede hacer que varias secciones de código se ejecuten de forma simultánea

```
#pragma omp parallel sections
{
   #pragma omp section
   {
      // ... algo de codigo
   }
   #pragma omp section
   {
      // ... otro codigo
   }
}
```

Variables *private* and *shared*

En el ejemplo del producto punto:

```
double ProductoPunto(double* a, double* b, int size)
{
   double c = 0;
   #pragma omp parallel for reduction(+:c)
   for (int i = 0; i < size; ++i)
   {
      c += a[i]*b[i];
   }
   return c;
}
```

OpenMP hace considera ***a*** y ***b*** como variables compartidas (*shared*), es decir, todos los *threads* las pueden accesar y modificar.

Para las variables del ciclo ***i*** y de la reducción ***c***, OpenMP hace un copias de las variables al *stack* de cada thread, con lo cual se convierten a privadas (*private*). Al terminar se sumarán las c de cada thread en la variable ***c*** que ahora se considerará *shared*.

Además, todas las variables declaradas dentro del bloque parallel serán privadas.

También es posible decidir de forma explícita que variables queremos que tengan un comportamiento privado o shared.

El siguiente ejemplo muestra como:

```
int i;
int j[10];
float* a;
float* b;
#pragma omp parallel private(i,j) shared(a,b)
{
   int k;
   // ... codigo ...
}
```

En este caso, tanto ***i***, ***j*** y ***k*** serán privadas y ***a*** y ***b*** serán compartidas.

Schedule

Por default, al paralelizar un ciclo el número de iteraciones se divide por igual entre cada *thread*. Sin embargo, esto no es eficiente si las iteraciones tardan tiempos diferentes en ejecutarse.

Es posible hacer entonces reajustar la distribución de iteraciones de forma dinámica, esto se hace con *schedule*.

```
#pragma omp parallel for schedule(type[, chunk_size])
for (int i = 0; i < size; ++i)
{
   c[i] = a[i] + b[i];
}
```

Los tipos de schedule son:

static Las iteraciones se dividen en *chunks* de un tamaño definido.

dynamic Cada thread ejecuta un *chunk* de iteraciones y al terminar su parte busca otro *chunk* para procesar.

guided Cada thread ejecuta un *chunk* de iteraciones y al terminar su parte busca otro *chunk* para procesar. El tamaño del *chunk* varía, siendo grande al iniciar y éste se va reduciendo.

auto La decición del tipo de schedule es decidido por el compilador/sistema operativo.

runtime El schedule y el tamaño del chunk son tomados de la variable sched-var ICV.

Otro ejemplo

Ahora veamos la multiplicación matriz-vector:

```
void Mult(double* A, double* x, double* y, int rows, int cols)
{
   #pragma omp parallel for
   for (int i = 0; i < rows; ++i)
   {
      double sum = 0;
      for (int k = 0; k < cols; ++k)
      {
         sum += A[i*cols + k]*x[k];
      }
      y[i] = sum;
   }
}
```

Hay que notar que las operaciones no se interfieren, dado que cada iteración sólo escribe en el elemento ***y[i]***.

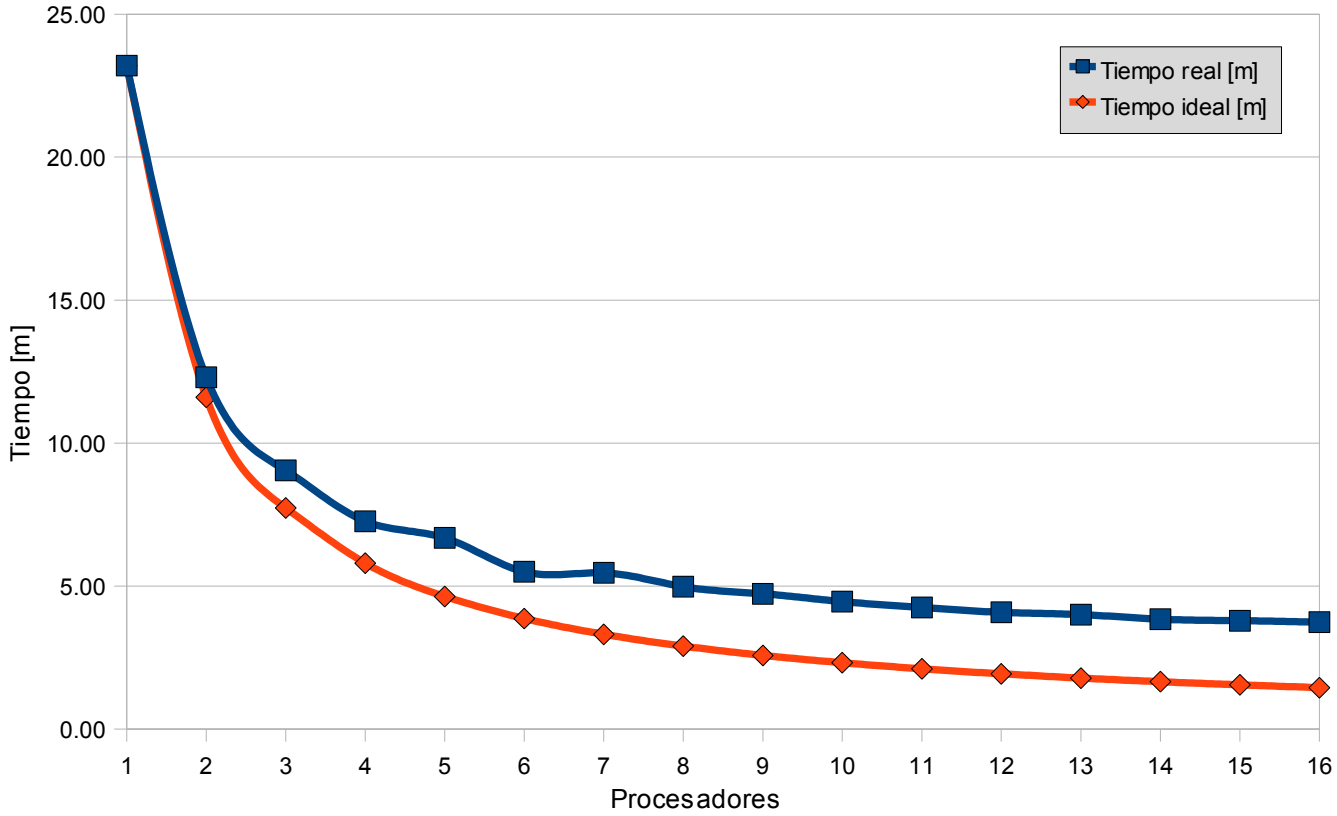
Secciones críticas

```
#pragma omp critical
{
   mpi.Send(p, tag_x, x);
}
```

La ejecucción se restringe a un solo *thread* a la vez.

Pérdida de eficiencia con el aumento de procesadores

Ejemplo de un programa trabajando en una computadora con 16 procesadores/threads

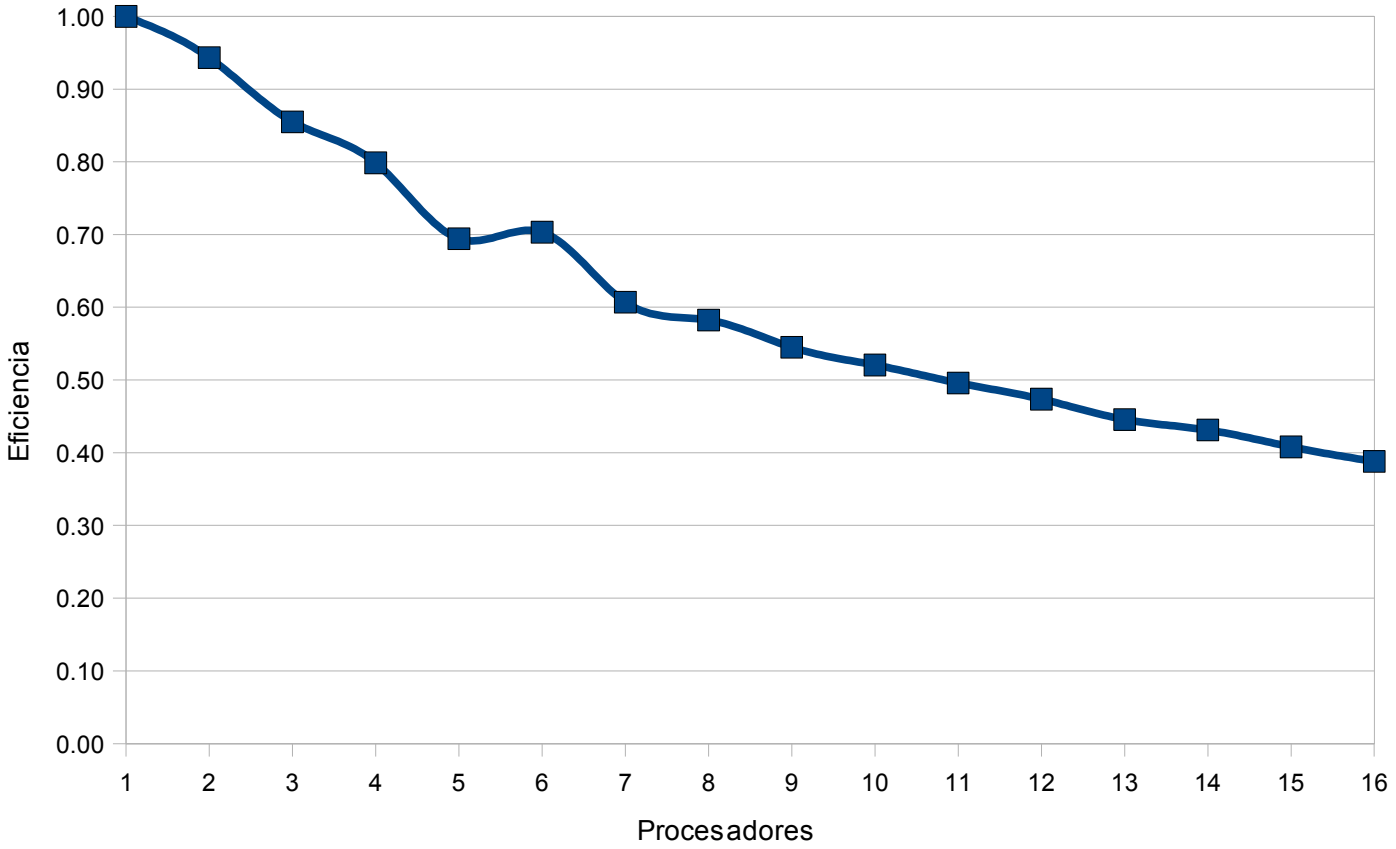


Sea t_1 el ***tiempo real*** que tardó el resolverse un problema con un *core*, n el número de procesadores utilizado, el ***tiempo ideal*** de ejecución lo podemos definir como

$$i_n = \frac{t_1}{n}.$$

Podemos dar una medida de ***eficiencia*** E de un algoritmo

$$E = \frac{i_n}{t_n} = \frac{t_1}{n\,t_n}.$$



Post data: Optimizar el código compilado

Los compiladores admiten opciones para mejorar el código objeto generado.

Algunas opciones interesantes:

- Habilitar SSE. SSE es un conjunto de instrucciones del procesador que hace las operaciones matemáticas de forma más eficiente.
- Incrementar el nivel de optimización. Se utilizan varias estrategías para hacer el código más rápido, como: desenrollar ciclos, reutilizar valores, poner valores temporales en registros, mezclar varias líneas de código. Estas optimizaciones pueden evitar que el código se pueda depurar (el debugger no puede identificar donde inicia o termina una línea de código).
- Eliminar chequeo en las operaciones de punto flotante.
- No compilar “asserts” insertados en el código.

Con GCC ésto se haría con

```
g++ -msse4 -O3 -ffast-math -DNDEBUG <archivo.cpp>
```

¿Preguntas?

Para saber más...

En la WWW:

http://openmp.org

http://www.google.com/#&q=openmp+tutorial

Bibliografía:

[Chap08] B. Chapman, G. Jost, R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2008.

[Drep07] U. Drepper. What Every Programmer Should Know About Memory. Red Hat, Inc. 2007.

[Fog10] A. Fog. Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms. Copenhagen University College of Engineering. 2010.

[Wulf95] W. A. Wulf , S. A. Mckee. Hitting the Memory Wall: Implications of the Obvious. Computer Architecture News, 23(1):20-24, March 1995.