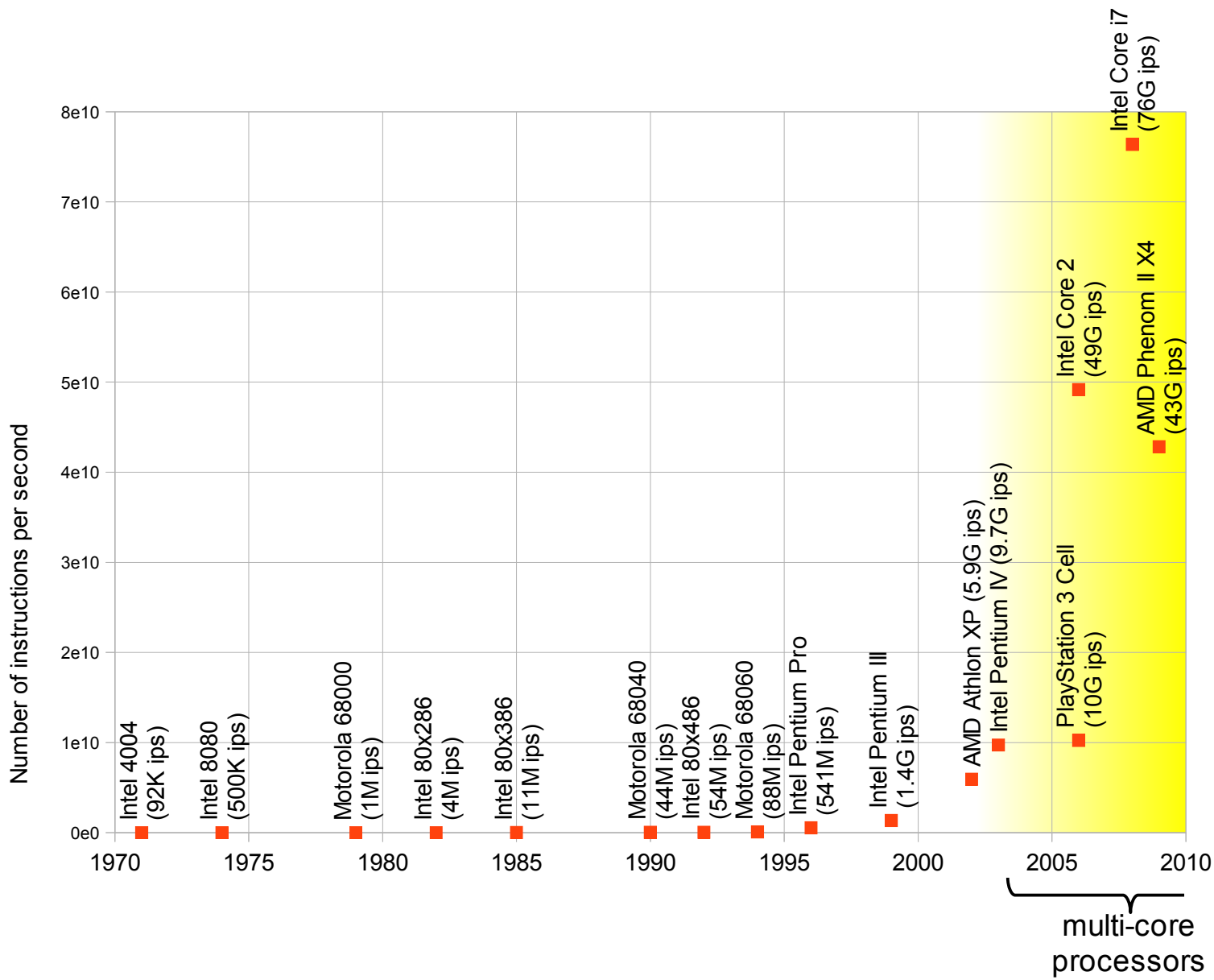
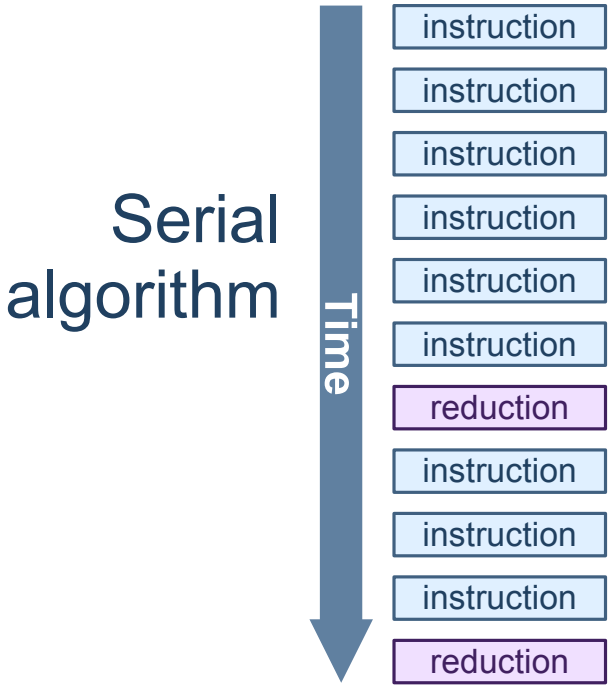
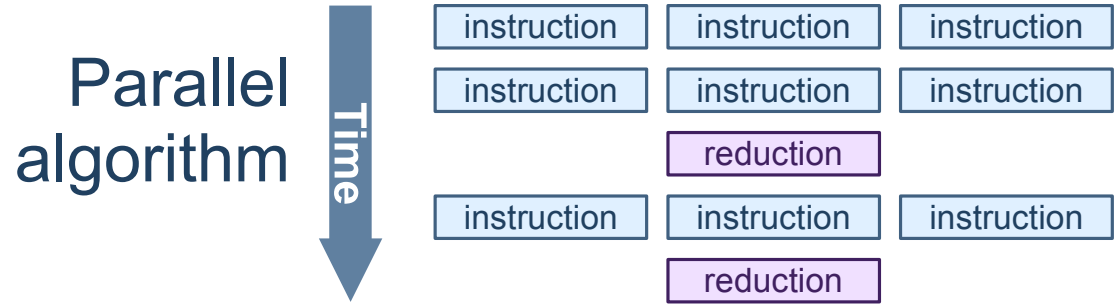
Velocidad de los procesadores de escritorio

Recientemente, la velocidad de procesamiento a crecido debido al uso de procesadores multi-core.



Tenemos que cambiar nuestros algoritmos para que corran lo más eficientemente posible en los procesadores multi-core.





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

La *buena* noticia es:

Hacer programas en paralelo con **OpenMP** es muy sencillo.

La *mala* noticia es:

Hacer programas en paralelo eficientes requiere un esfuerzo extra.

Operaciones matemáticas en paralelo

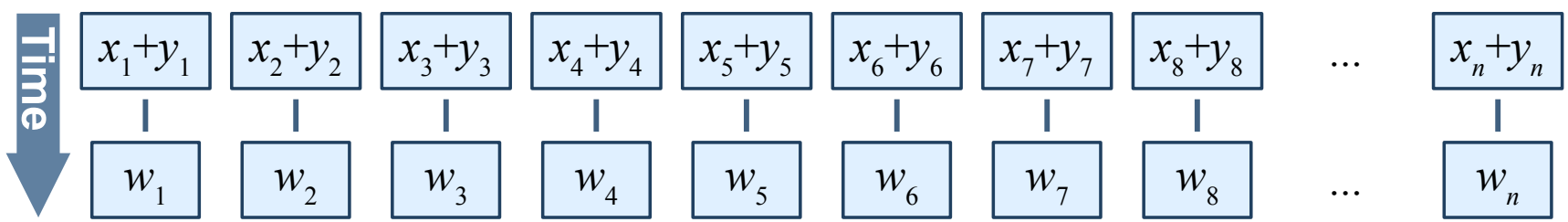
Fácilmente paralelizables

Esto significa que pueden separarse en varias sub-operaciones que puede realizarse de forma independiente.

Por ejemplo, la suma de dos vectores $\mathbf{w}=\mathbf{x}+\mathbf{y}$, con $\mathbf{w}, \mathbf{x}, \mathbf{y}\in\mathbb{R}^n$,

$$w_i = x_i + y_i.$$

Y supongamos que tenemos *n* procesadores.



En este caso las 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.

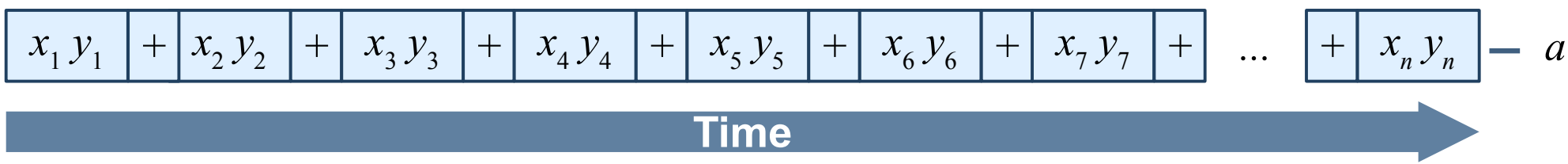
No tan fáciles de paralelizar

Por ejemplo el producto punto, $a = \langle \mathbf{x}, \mathbf{y} \rangle$, con $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,

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

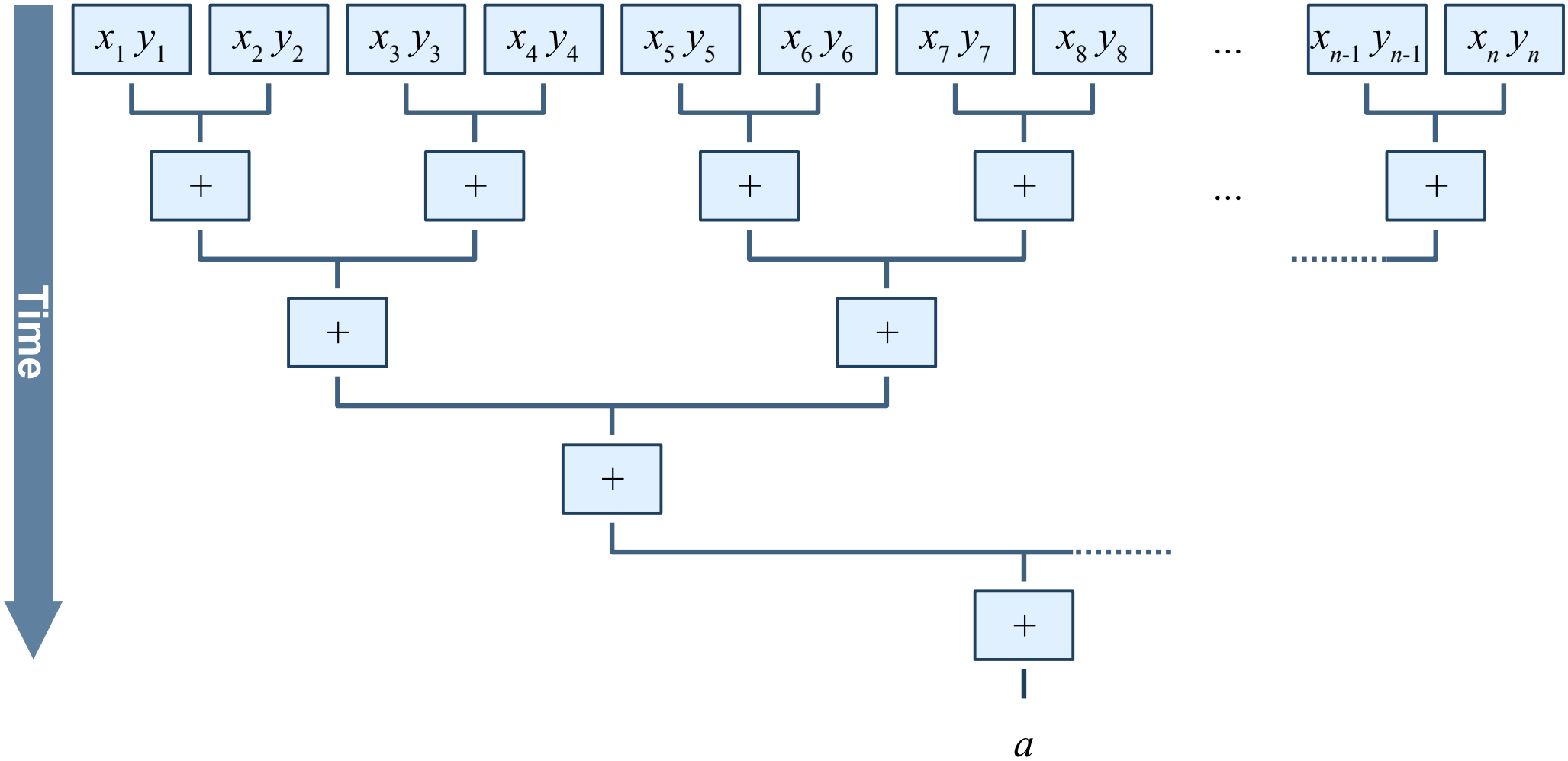
donde a es un escalar.

Una primera aproximación sería verlo como una secuencia de sumas de productos que requieren irse acumulando.



Al verlo así no es una operación paralelizable.

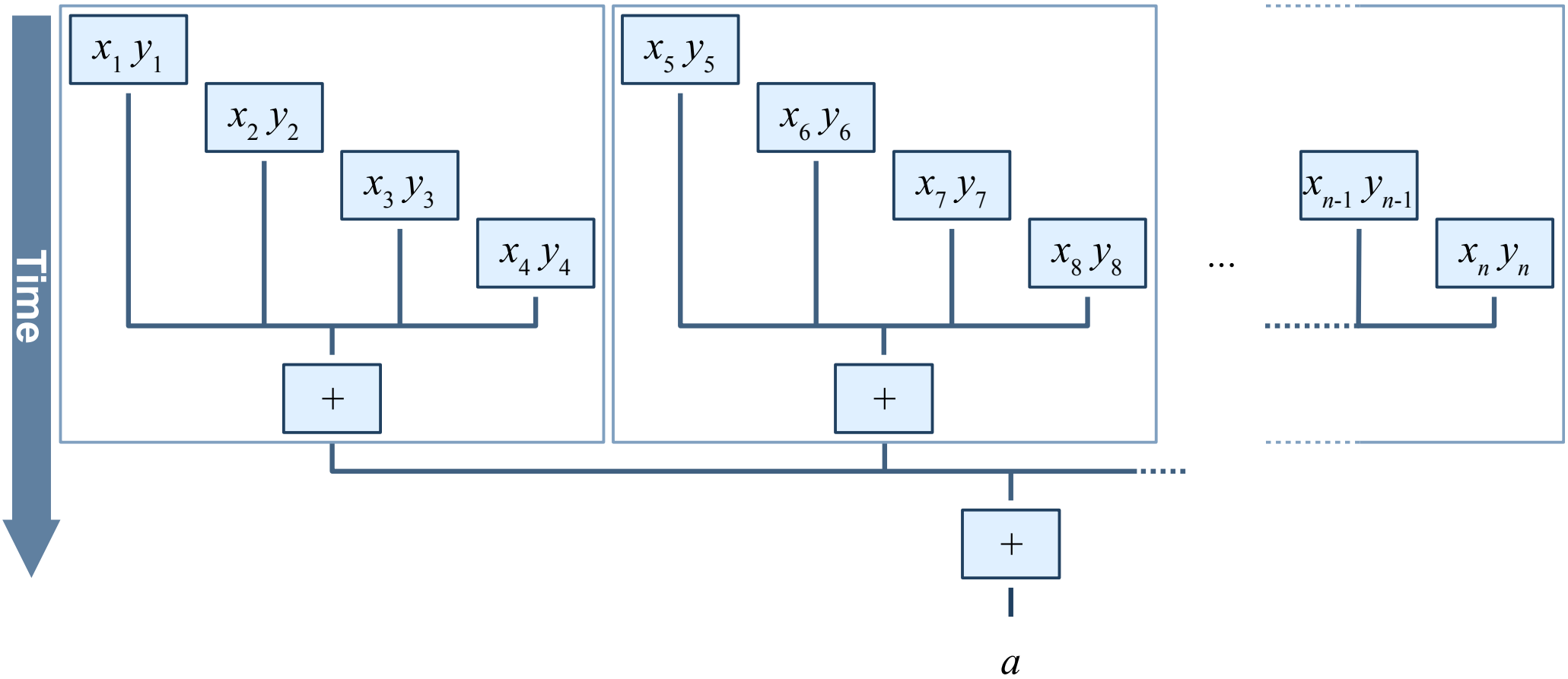
Sin embargo, podemos reorganizar el proceso como se muestra en la figura:



En este caso se tiene una paralelización eficiente de la multiplicación de las entradas de los vectores, después se va reduciendo la eficiencia al dividir las operaciones de suma en pares.

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 serie, lo que limita la eficiencia del paralelismo.



El esquema OpenMP

- Sirve para paralelizar programas en C, C++ o Fortran en computadoras multi-core.
- Busca que escribir código en paralelo sea sencillo.
- Es un esquema de paralelización con memoria compartida (todos los cores 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).

Una descripción más a detalle de OpenMP 2.5 la pueden encontrar en [Chap08].

Compiladores que soportan OpenMP

Algunos de los compiladores que tienen soporte para OpenMP son:

- GNU Compiler Collection GCC (versión $\geq$ 4.3.2).
- The LLVM Compiler Infrastructure (versión $\geq$ 3.7.0)
- Intel C++ and Fortran Compilers (version $\geq$ 10.1) [1]
- Microsoft Visual Studio Professional (version $\geq$ 2008) [2]
- Microsoft Visual Studio Express (version $\geq$ 2012) [3]
- Microsoft Visual Studio Community (2015) [4]

La lista completa se puede consultar en http://openmp.org/wp/openmp-compilers

[1] Versión gratuita para Linux para estudiantes.
http://software.intel.com/en-us/non-commercial-software-development

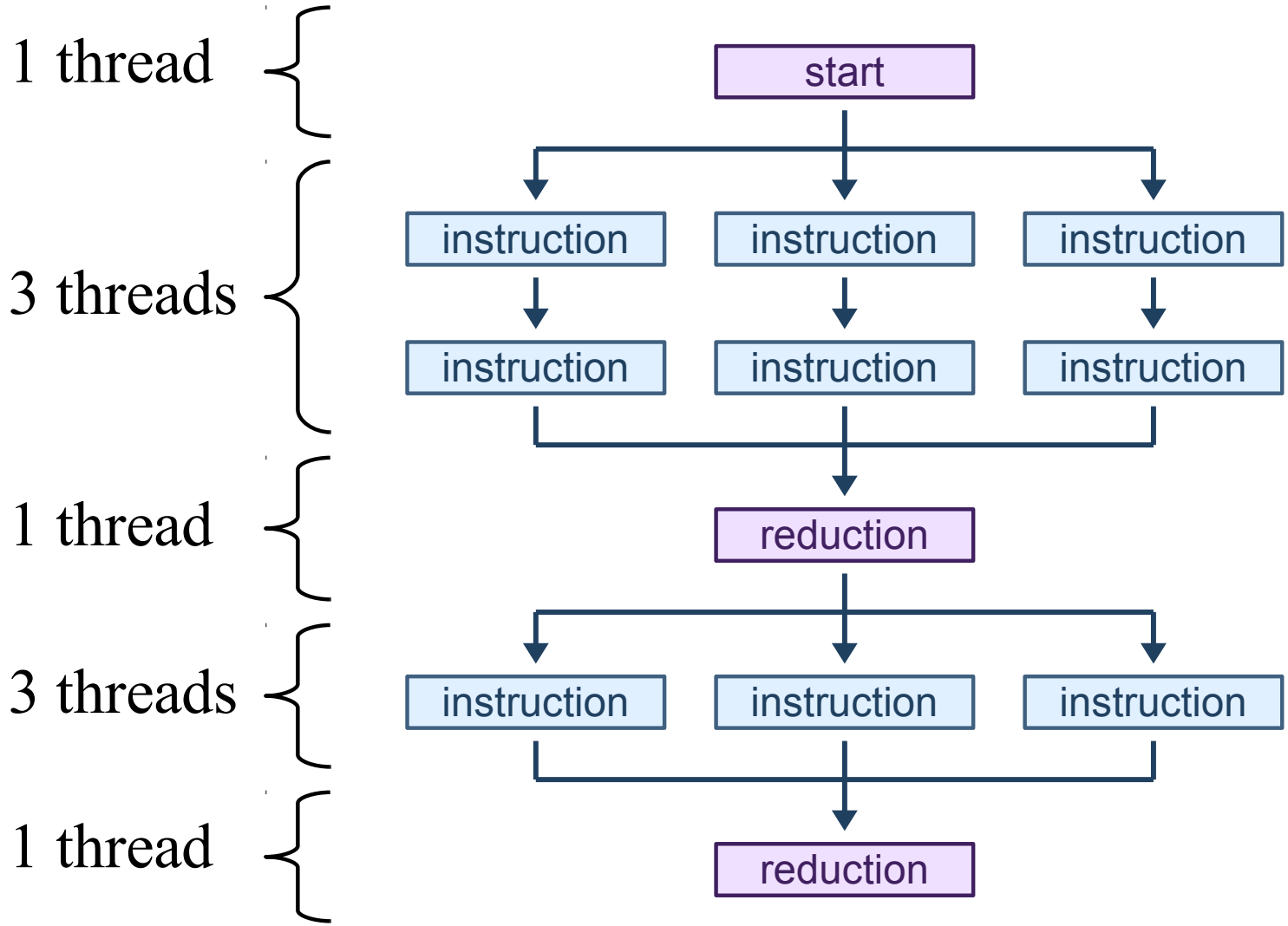
[2] Descargable de forma gratuita para estudiantes de instituciones de educación superior en Microsoft DreamSpark.
http://www.dreamspark.com

[3] Gratis para todo tipo de desarrollo.
http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products

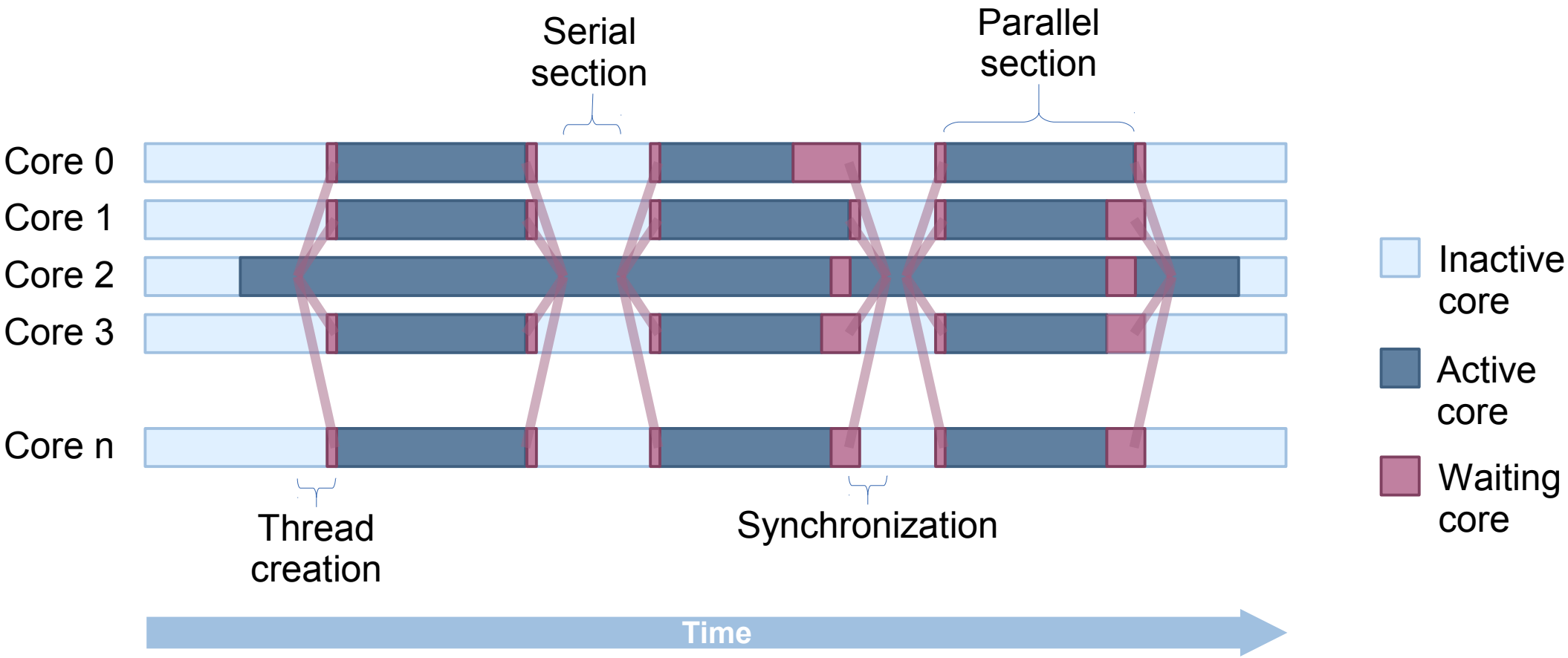
[4] Completamente gratis para empresas pequeñas y desarrollo open source.
https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

Programación con multi-hilos (multi-threads)

A la ejecución secuencial de una conjunto de instrucciones se le conoce como un hilo de procesamiento o thread.



En las computadoras multi-core logra la mejor eficiencia cuando cada core ejecuta sólo un thread.



Cuando se paraleliza no todos los threads terminan al mismo tiempo.

Dentro de un programa, cada thread posee sus propios registros de control y su propio stack de datos, mientas que comparte el uso de las regiones de memoria heap y data con los demás threads del programa.

Paralelización con OpenMP

Suma de vectores

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];
    }
}
```

Un **#pragma** es una directiva para el compilador, significa que tiene que insertar código especial que haga algo que no está definido en el estandar de C o C++.

En este caso el **#pragma** internamente inserta código oculto de OpenMP para paralelizar el **for**.

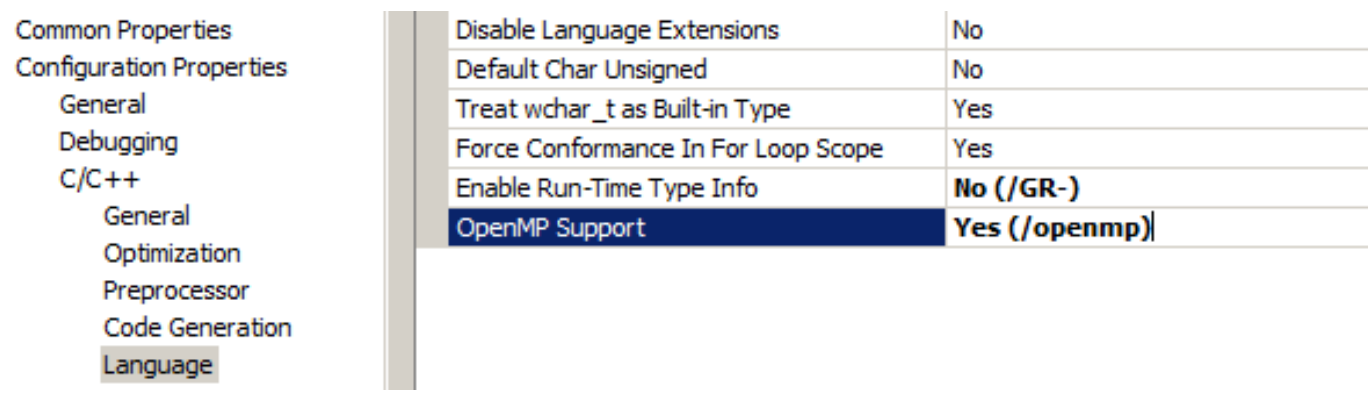
Compilación de un código con OpenMP

No es suficiente con agregar el #pragma, hace falta decirle al compilador que agregue el soporte para OpenMP.

Para compilar con GCC es necesario agregar:

```
g++ -o programa -fopenmp suma.cpp
```

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



Paralelización del **for**

La forma en que OpenMP paraleliza un **for** es formando bloques de índices (el tamaño de los bloques es controlable).

Supongamos que tenemos una computadora con 3 cores.

Para la suma de vectores size=30 y que los bloques son de tamaño 10.

```
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];
    }                                     }                                      }
}
```

OpenMP asigna un bloque de índices a cada core.

- Al core 0 le toca procesar el for con los índices 0-9
- Al core 1 le toca procesar el for con los índices 10-19
- Al core 2 le toca procesar el for con los índices 20-29

Por ejemplo, supongamos ahora que size=100 y que los bloques siguen siendo de tamaño 10.

La distribución de índices podría ser algo 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];
    }                                     }                                      }
    for (int i = 40; i < 50; ++i)         for (int i = 30; i < 40; ++i)          for (int i = 50; i < 60; ++i)
    {                                     {                                      {
        c[i] = a[i] + b[i];                   c[i] = a[i] + b[i];                    c[i] = a[i] + b[i];
    }                                     }                                      }
    for (int i = 80; i < 90; ++i)         for (int i = 70; i < 80; ++i)          for (int i = 60; i < 70; ++i)
    {                                     {                                      {
        c[i] = a[i] + b[i];                   c[i] = a[i] + b[i];                    c[i] = a[i] + b[i];
    }                                     }                                      }
    for (int i = 90; i < 100; ++i)
    {
        c[i] = a[i] + b[i];
    }
}
```

A los bloques se les llama ***chunk***, y su tamaño puede ser controlado por el programador.

La distribución de los índices es controlada por OpenMP.

Nosotros simplemente tenemos que escribir

```
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];
    }
}
```

y dejamos que OpenMP haga el resto.

Si el compilador no tiene soporte para OpenMP, entonces el **#pragma** es ignorado y tendremos la versión serial del programa.

Si la computadora no es *multi-core*, entonces también se ejecutará el código en serie.

Este esquema permite primero probar nuestro código en serie y luego ejecutarlo en paralelo.

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.

Controlar el número de threads

```
#include <stdio.h>
#include <omp.h>

int main()
{

    int max = omp_get_max_threads();
    printf("omp_get_max_threads = %i\n", max);

    #pragma omp parallel
    {
        int t = omp_get_thread_num();
        printf("  omp_get_thread_num = %i\n", t);
    }

    return 0;
}
```

```
g++ -o test -fopenmp test.cpp
./test
```

La función omp_get_max_threads regresa la cantidad máxima de threads establecida.

omp_get_thread_num regresa el **ID** del thread, los threads se identifican enumeran iniciando en 0.

Si una computadora tiene un procesador con 4 cores, podemos decidir no utilizarlos todos.

¿Cómo se define el número de cores/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. La variable se tiene que inicializar antes de ejecutar el programa.

En Linux/Unix (Bash):

```
export OMP_NUM_THREADS=3
./test
```

En Windows:

```
set OMP_NUM_THREADS=3
test.exe
```

Establecer número de threads con código

Una forma es utilizando una función de OpenMP, para esto es necesario incluir el header “omp.h”, el cual incluye varias funciones para el control de threads.

```
#include <omp.h>

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];
    }
}

int main()
{
    omp_set_num_threads(3);
    ...
    Suma(a, b, c, 100);
}
```

omp_set_num_threads permite establecer el número de threads a utilizar en todas las siguientes paralelizaciones.

Otra forma sería utilizando la clausula num_threads.

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

int main()
{
    int threads = 3;
    ...
    Suma(a, b, c, 100, threads);
}
```

De esta forma se puede tener un control más fino del número de threads para cada paralelización.

Reducciones

El ejemplo que mostraremos es la paralelización del 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;
}
```

¿Qué pasa si hacemos...?

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

La forma en que OpenMP permite hacer operaciones de este tipo es por medio de la cláusula **reduction**.

```
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;
}
```

Lo que hace **reduction** es crear tantas copias de c como threads existan e irlas acumulando por separado, al terminar se suman éstas y se tendrá un solo resultado.

Las operaciones soportadas en las reduciones son:

Operador de reducción	Valor de inicialización
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0
max	
min	

Para poner múltiples reduciones, puede ser como:

```
double minimum = a[0];
double maximum = a[0];

#pragma omp parallel for reduction(min:minimum) reduction(max:maximum)
for (int i = 1; i < size; ++i)
{
    if (minimum > a[i])
        minimum = a[i];
    else if (maximum < a[i])
        maximum = a[i];
}
```

Multiplicación matriz-vector

Ahora veamos la multiplicación matriz-vector.

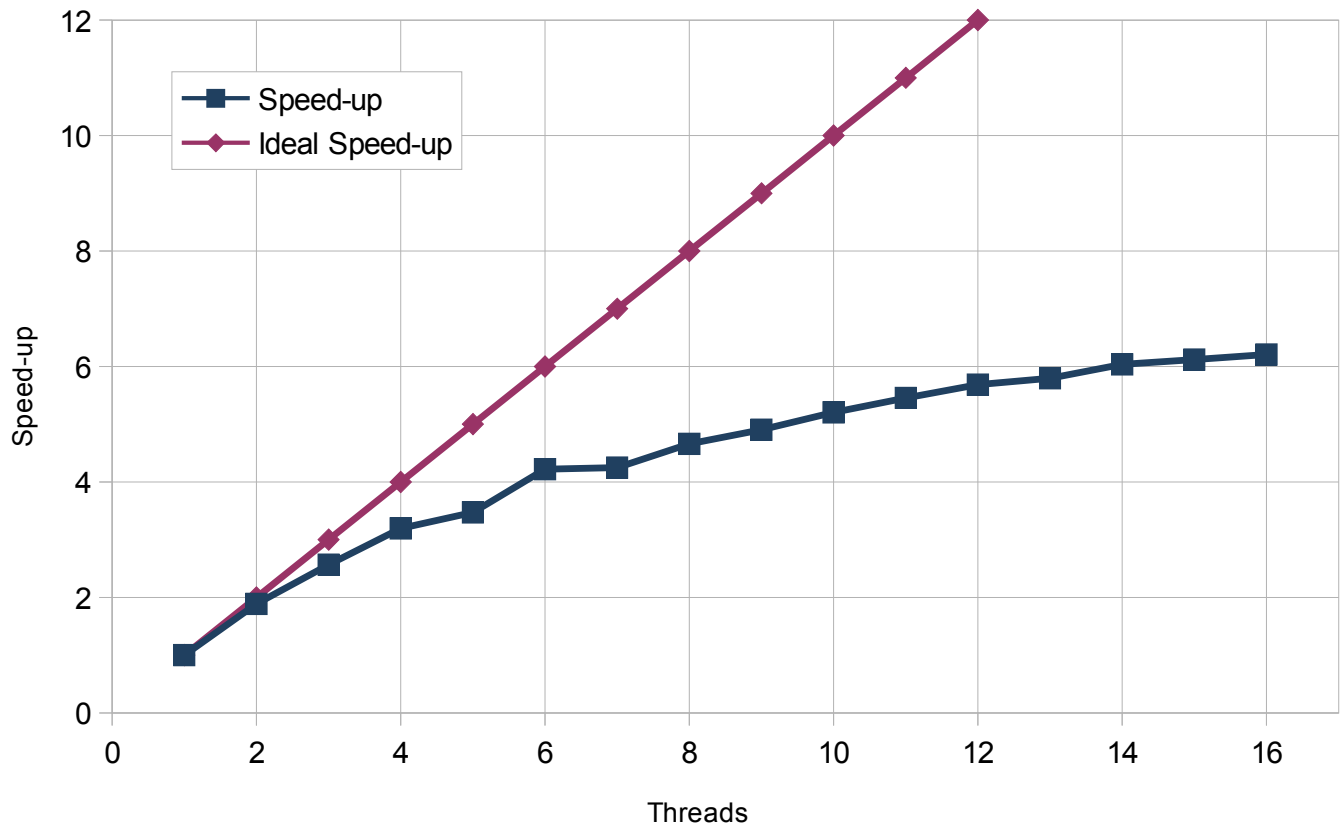
```
void Mult(double* A, double* x, double* y, int rows, int cols)
{
    #pragma omp parallel for
    for (int i = 0; i < rows; ++i)
    {
        double sum = 0;
        for (int k = 0; k < cols; ++k)
        {
            sum += A[i*cols + k]*x[k];
        }
        y[i] = sum;
    }
}
```

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

Speed-up

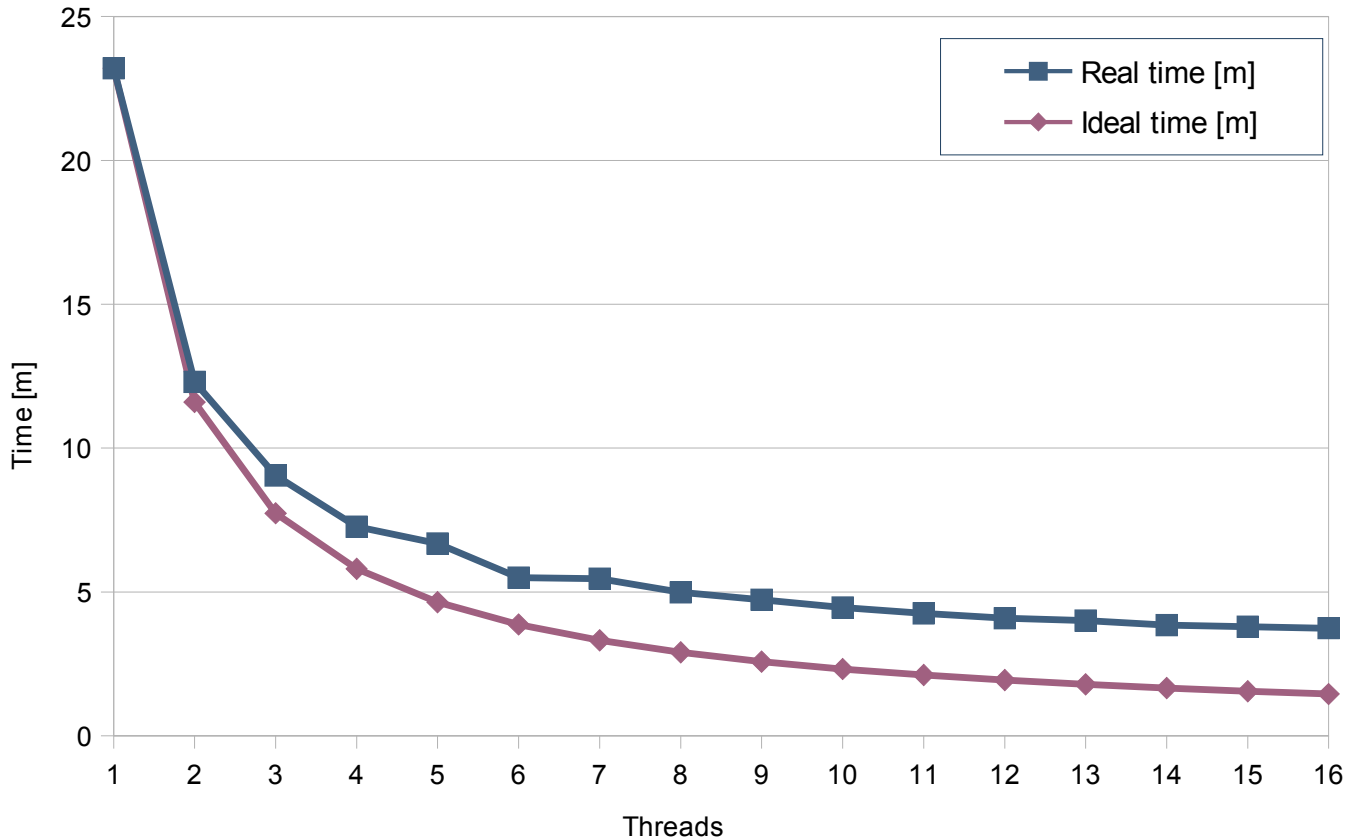
La aceleración (speed-up) de un programa en paralelo,

$$S = \frac{t_1}{t_n}.$$



Pérdida de eficiencia con el aumento de threads

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

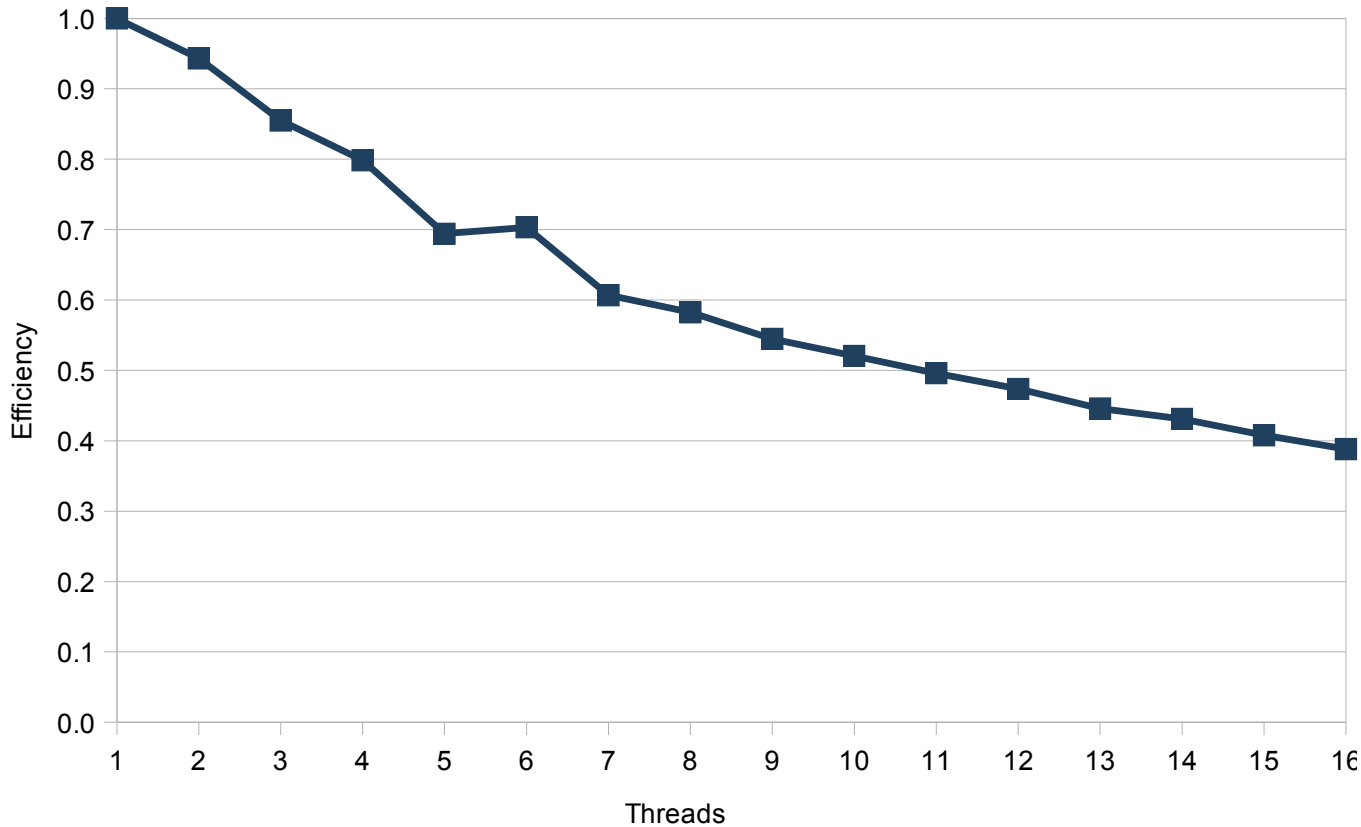


Sea t_1 el *tiempo real* que tardó el resolverse un problema con un thread, n el número de procesadores utilizado, el *tiempo ideal* de ejecución lo podemos definir como

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

Podemos dar una medida de eficiencia E de un algoritmo

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



Teorema de Geršgorin

Sea $\mathbf{A} \in \mathbb{C}^{n \times n}$, con entradas a_{ij}. Definamos los radios

$$r_i = \sum_{j \neq i} |a_{ij}|, \; i = 1, 2, \ldots, n,$$

como la suma de las entradas fuera de la diagonal del renglón i.

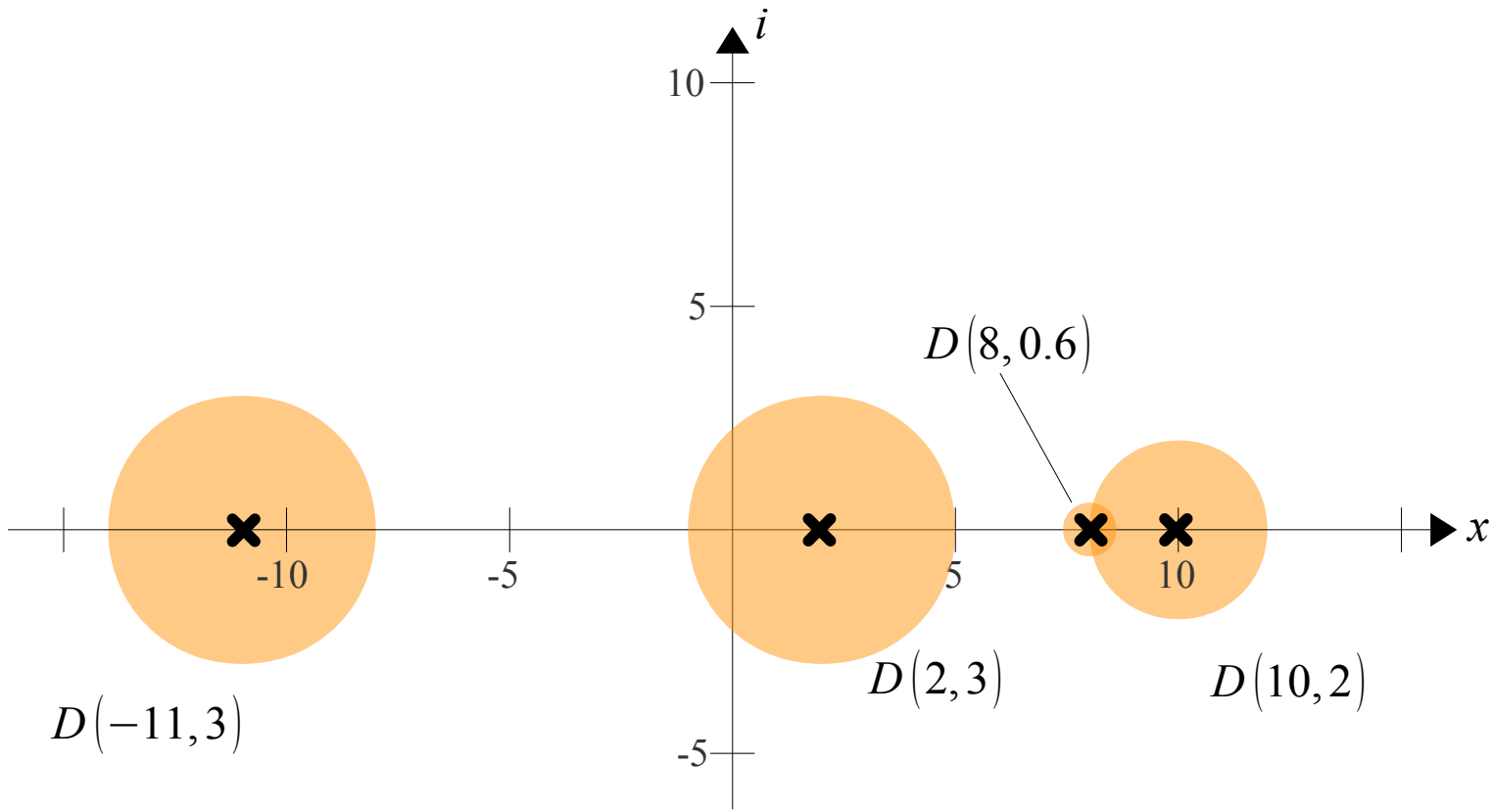
Sea $D(a_{ii}, r_i)$ un disco cerrado en el plano complejo, con centro en a_{ii} y radio r_i. Estos discos son llamados discos de Geršgorin [Varg04].

Teorema: Cada eigenvalor de $\mathbf{A}$ está dentro de alguno de los discos de Geršgorin $D(a_{ii}, r_i)$.

Ejemplo, podemos estimar los eigenvalores de

$$\mathbf{A}=\begin{pmatrix} 10 & -1 & 0 & 1 \\ 0.2 & 8 & 0.2 & 0.2 \\ 1 & 1 & 2 & 1 \\ -1 & -1 & -1 & -11 \end{pmatrix},$$

se obtienen cuatro discos $D(10,2)$, $D(8,0.6)$, $D(2,3)$, $D(-11,3)$.



Los eigenvalores de **A** son: 9.8218, 8.1478, 1.8995, -10.86.

El número de condición κ de una matriz $\mathbf{A}$ no singular, para una norma $\|\cdot\|$ está dado por

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|.$$

Para la norma $\|\cdot\|_2$,

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\| = \frac{\sigma_{max}(\mathbf{A})}{\sigma_{min}(\mathbf{A})},$$

donde σ son los valores singulares de la matriz.

Para una matriz $\mathbf{A}$ simétrica positiva definida,

$$\kappa(\mathbf{A}) = \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(\mathbf{A})},$$

donde λ son los eigenvalores de $\mathbf{A}$. Así, matrices con un número de condición cercano a 1 se dicen que están bien condicionadas.

Podemos utilizar el teorema de Geršgorin para encontar un aproximado del número de condición para de matrices dispersas (simétricas positivas definidas).

¿Preguntas?

miguelvargas@cimat.mx

Referencias

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

[Varg04] R. S. Varga. Geršgorin and His Circles. Springer. 2004.