



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Departamento de Computación
Estructuras de datos y algoritmos II

Tema 6

Introducción a los algoritmos paralelos

6 Introducción a los algoritmos paralelos

Objetivo: Clasificar los elementos a considerar en el diseño y análisis de algoritmos paralelos versus algoritmos seriales para su programación.

6 Introducción a los algoritmos paralelos

6.1 Niveles de paralelismo. Granularidad.

6.2 Algoritmos con memoria compartida.

 6.2.1 Carrera de datos.

 6.2.2 Inconsistencia de datos.

 6.2.3 Modelo PRAM.

6.3 Técnicas de desarrollo de algoritmos.

 6.3.1 Rediseño de estructuras de datos.

 6.3.2 Rediseño de algoritmos.

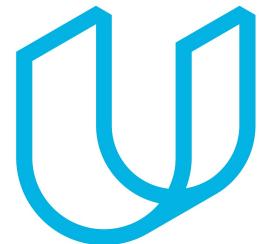
6.4 Análisis de desempeño de algoritmos paralelos.

 6.4.1 Trabajo y profundidad.

 6.4.2 Ejemplos clásicos.

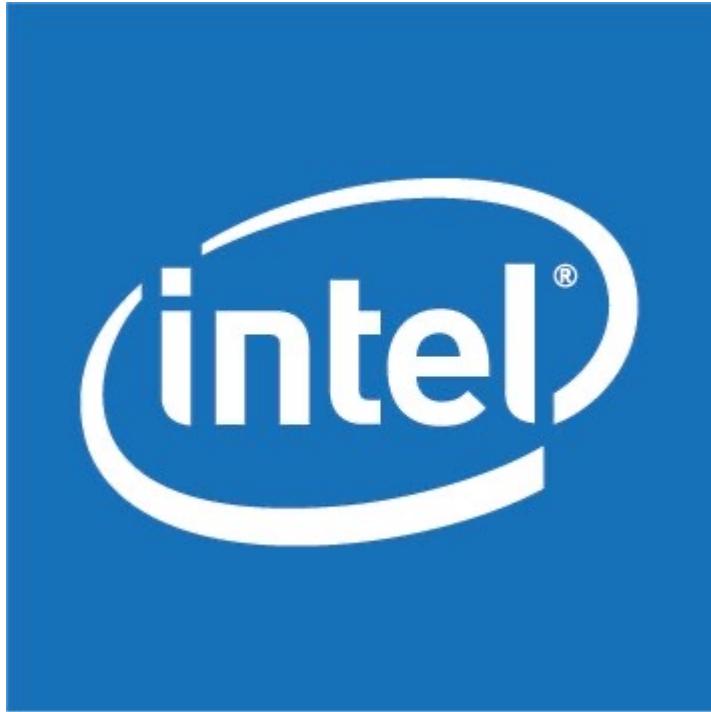
Referencia

Udacity: Intro to parallel
programming.



UDACITY

Referencia



Introduction to OpenMP - Tim Mattson (Intel)

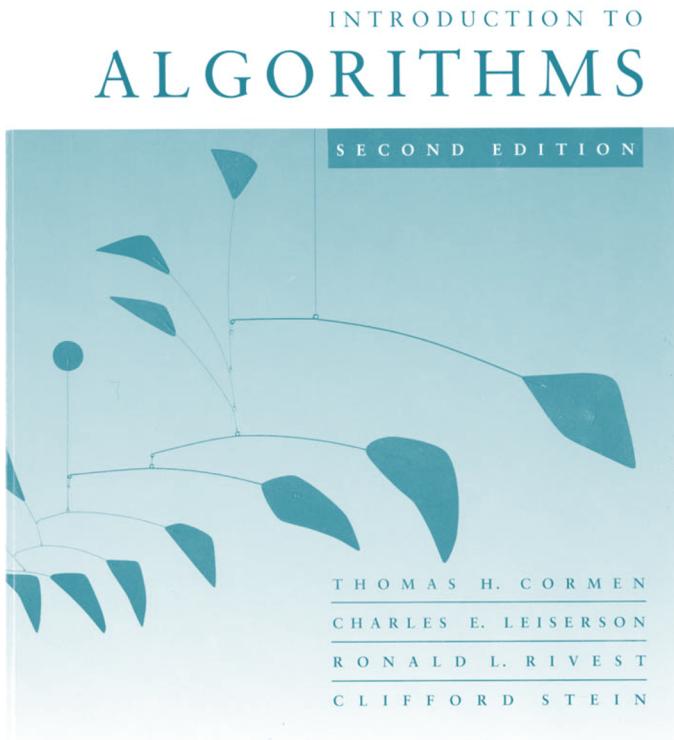
http://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Referencia

<http://www.openmp.org/>



Bibliografía



Introduction to Algorithms.
Thomas H. Cormen, Charles E.
Leiserson, Ronald L. Rivest,
Clifford Stein, McGraw-Hill.

Introducción a los algoritmos paralelos

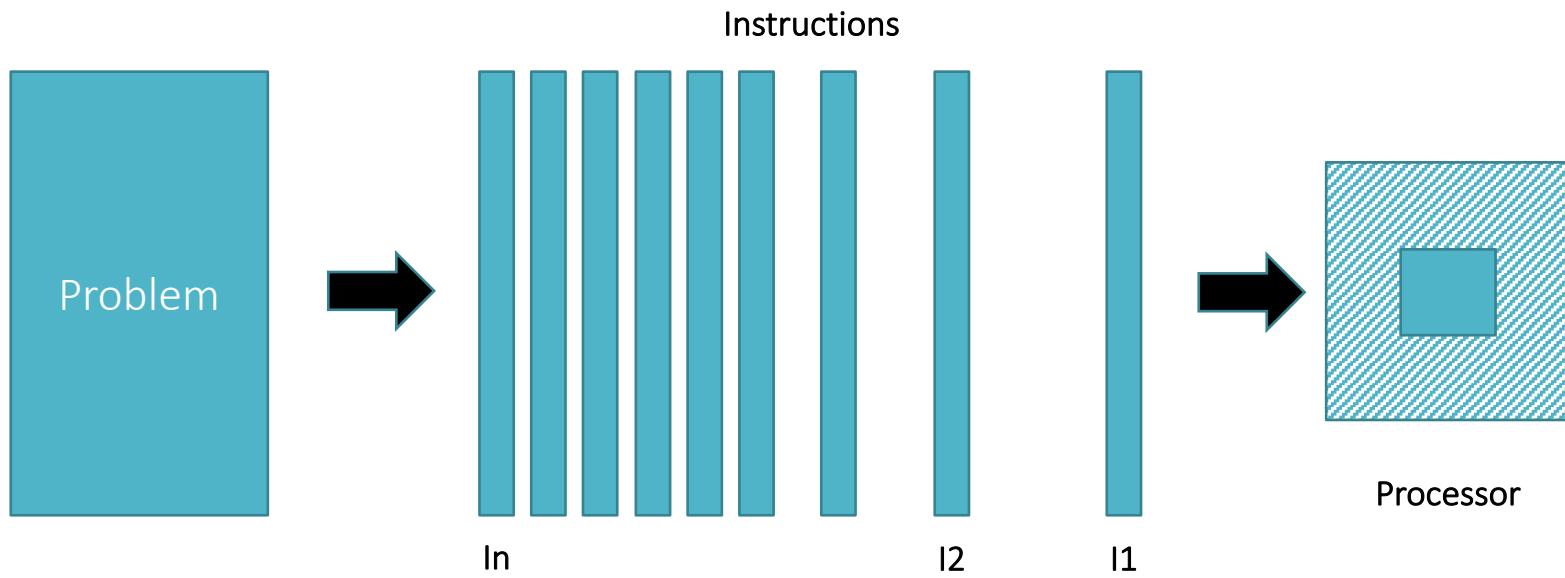
Tema 6

“Hardware eventually fails. Software eventually works.”

Michael Hartung
(System Administrator)

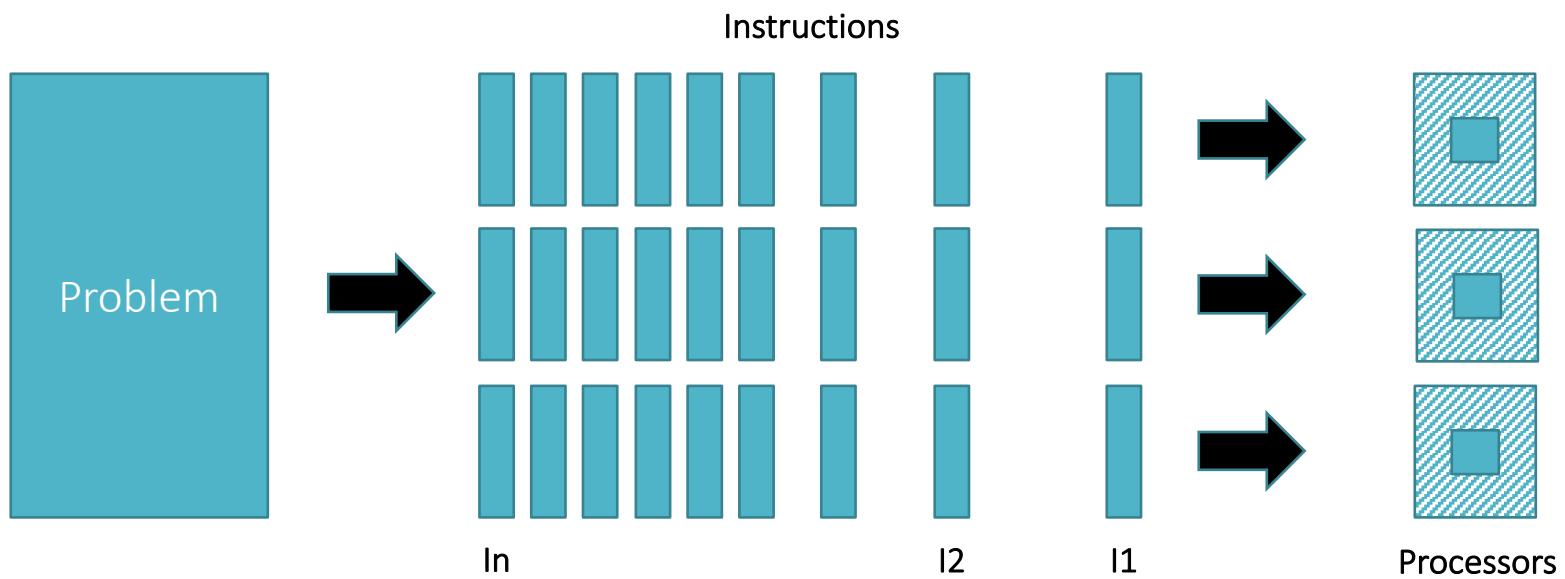
De manera tradicional, el software se escribe (se piensa) de manera serial, es decir, un problema se descompone en una serie de instrucciones, las cuales se ejecutan de manera secuencial (una tras otra), se ejecuta sobre un solo procesador y sólo se puede ejecutar una a la vez.

Diagrama de un problema resuelto de manera serial:



El cómputo paralelo se refiere al uso simultáneo de diversos recursos de cómputo para resolver un problema, es decir, un problema se descompone en partes que pueden ser resueltas de manera concurrente, cada parte está compuesta por una serie de instrucciones, cada una de las cuales se ejecuta de manera simultánea en diferentes procesadores.

Diagrama de un problema resuelto de manera paralela:



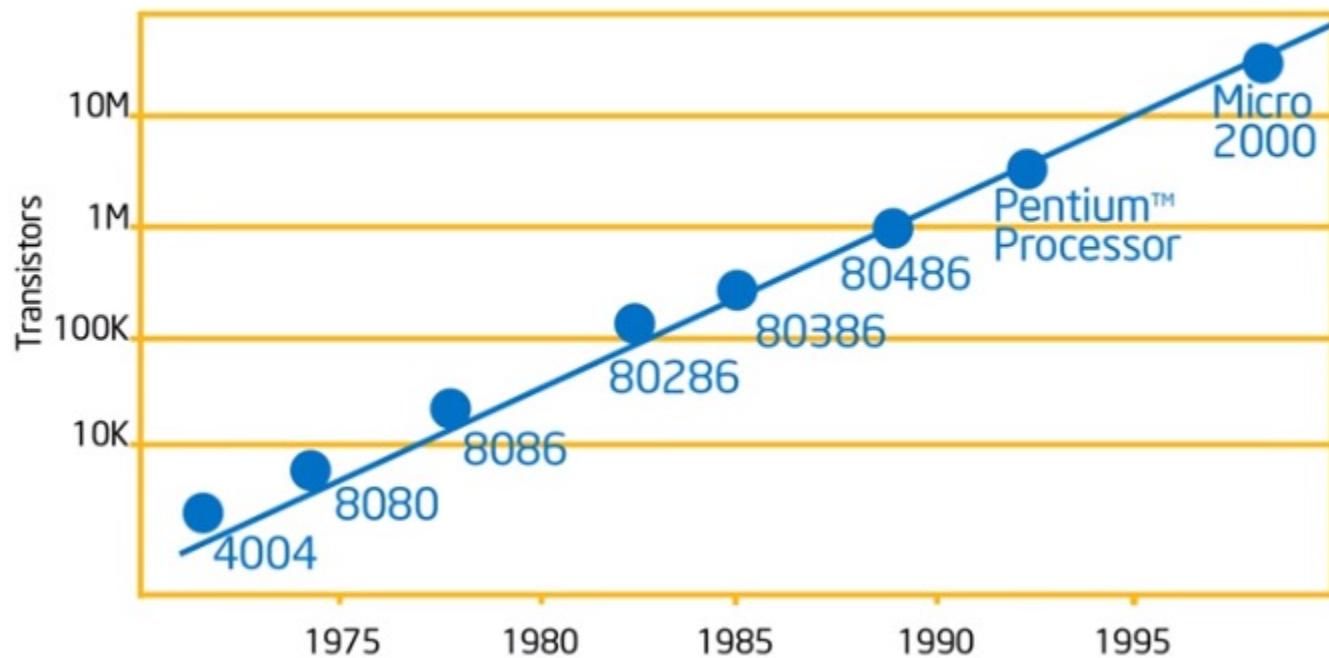
No todos los problemas ni todos los entornos son paralelizables. Para que un problema se pueda resolver de manera paralela, éste debe ser capaz de descomponerse en partes que puedan ser resueltas de manera simultánea. Para que un entorno sea paralelo, éste debe ser capaz de ejecutar múltiples instrucciones a la vez.

Los recursos paralelos son, típicamente, dos: una computadora con múltiples procesadores (núcleos) o un número arbitrario de computadoras conectadas en red (clúster).

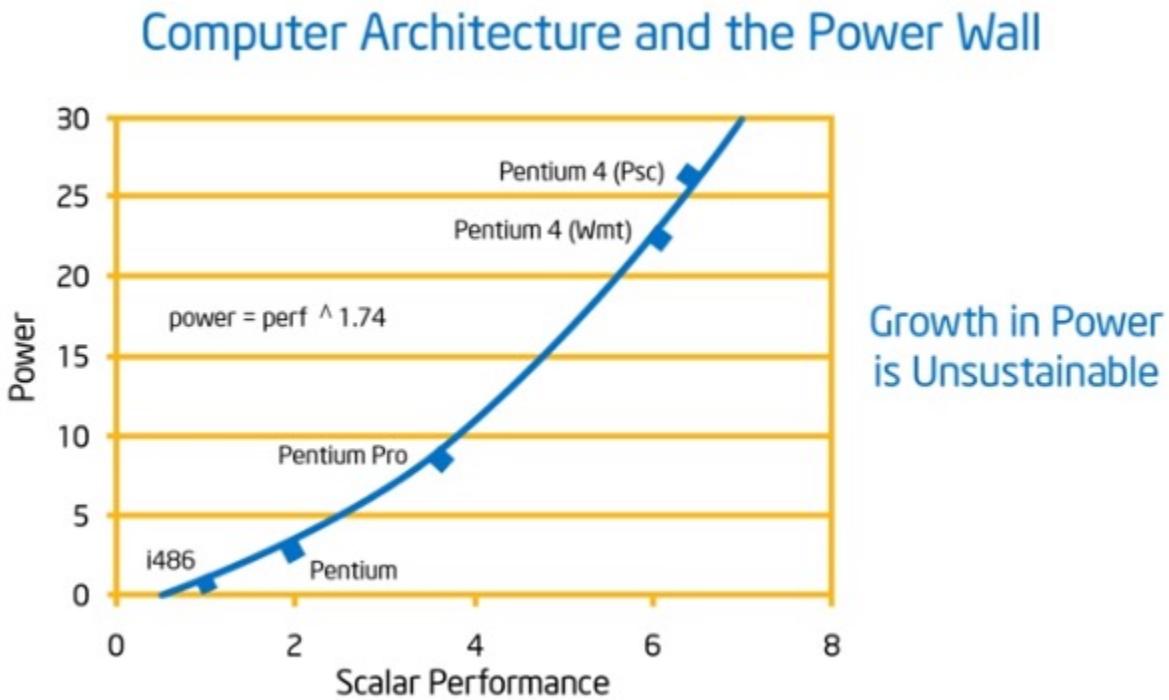
La ley de Moore

El cofundador de Intel Gordon Moore en el artículo de la revista Electronics Magazine titulado “Cramming More Components onto Integrated Circuits” del 19 de abril de 1965 mencionó “pronostico que el número de transistores incorporados en un chip se duplicará aproximadamente cada 24 meses”.

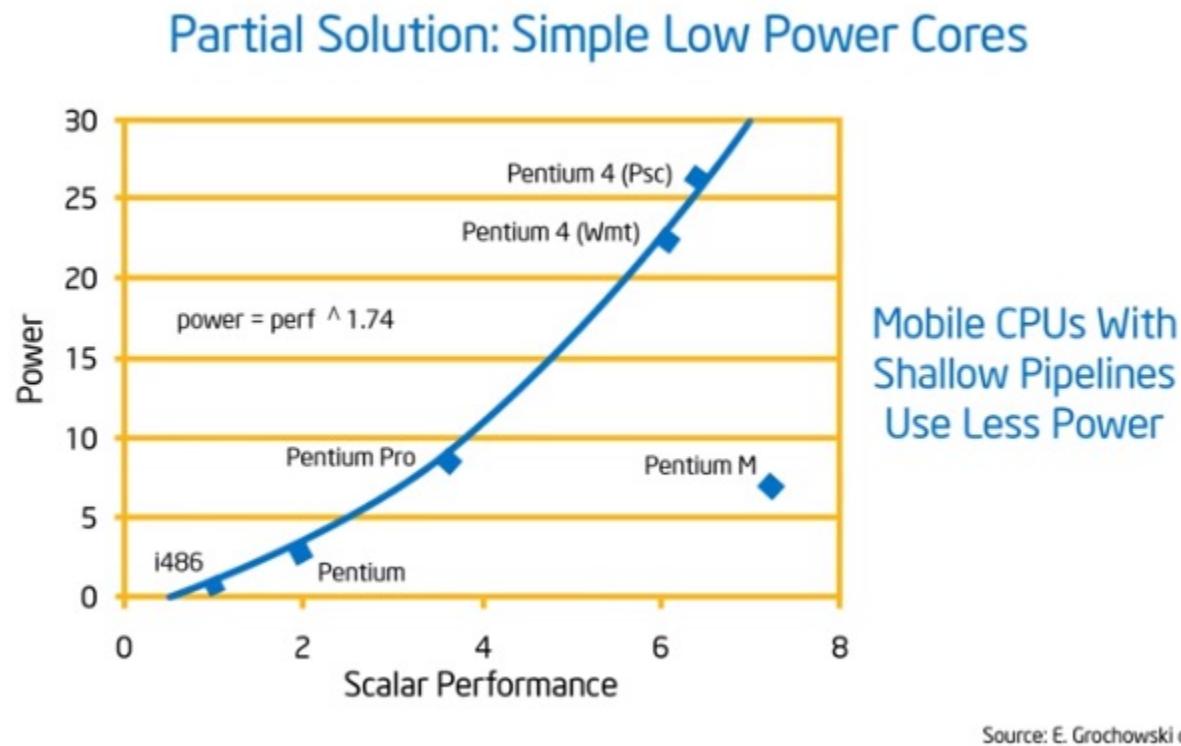
Moore's Law



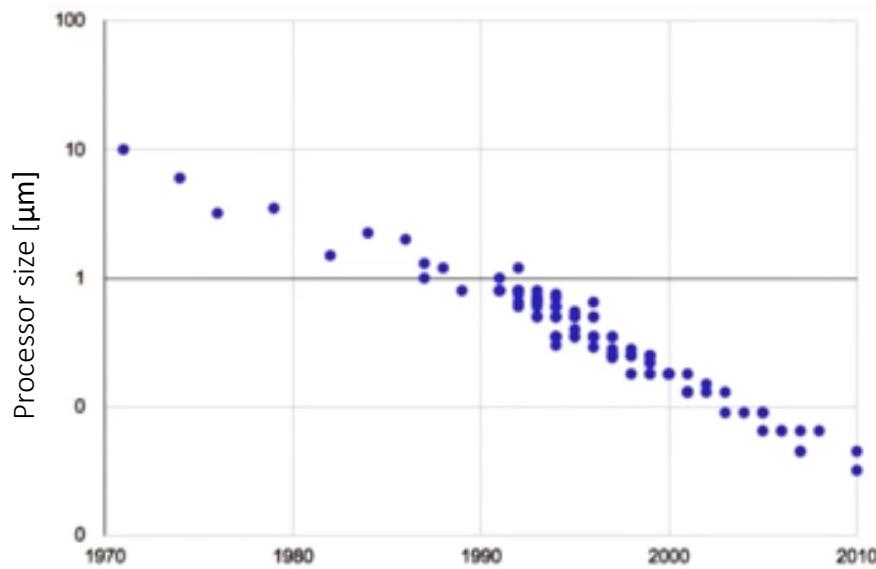
Power Wall o muro de potencia se refiere al crecimiento en la demanda de poder conforme se desarrollan diversas generaciones de procesadores, el cual es insostenible conforme avanza el desarrollo.



Por supuesto, el muro de potencia existe cuando la demanda de procesamiento es mucha, para procesadores de dispositivos móviles, la potencia que se demanda es menor, igual que el procesamiento.

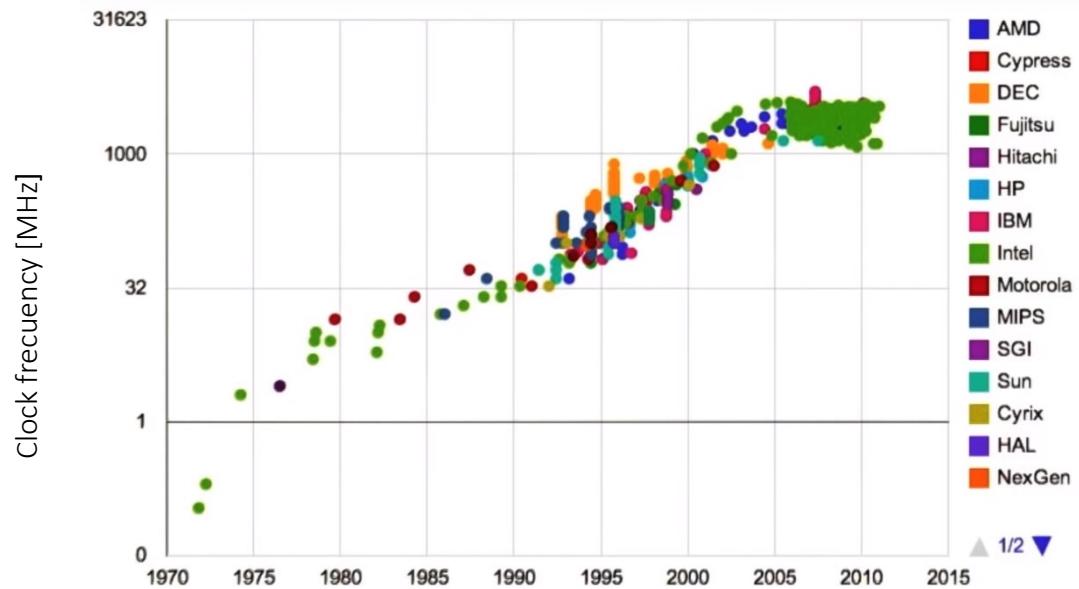


La siguiente gráfica es parte del proyecto CPUDB de Standford, la cual muestra el desarrollo en tamaño que han tenido los procesadores a lo largo del tiempo. Se puede observar como ha disminuido el tamaño de los procesadores e incrementado su capacidad de procesamiento, usando menos energía y uniendo más de un chip, cumpliendo con esto con la ley de Moore.



<http://cpudb.stanford.edu/>

La frecuencia de reloj de los equipos de cómputo ha ido incrementando con el paso de los años de manera lineal. Sin embargo, en los últimos años esta frecuencia se ha mantenido prácticamente constante. La rapidez de procesamiento se debe a la cantidad de transistores que tiene un circuito integrado.



<http://cpudb.stanford.edu/>

Capacitancia

Se refiere a la capacidad que tiene un circuito para almacenar energía. La capacitancia está definida por:

$$C = q / V \quad (1)$$

$$q = C * V \quad (2)$$

Donde C es la capacitancia del circuito, q es la carga y V es el voltaje.

Trabajo

El trabajo se refiere a la fuerza que se requiere para empujar algo (por ejemplo, una carga q) a lo largo de una distancia. En términos de electrostática empujar q de 0 a V .

$$W = V * q \quad (3)$$

Donde W es el trabajo, q es la carga y V es el voltaje. Sustituyendo 2 en 3 se tiene

$$W = C * V^2 \quad (4)$$

Potencia

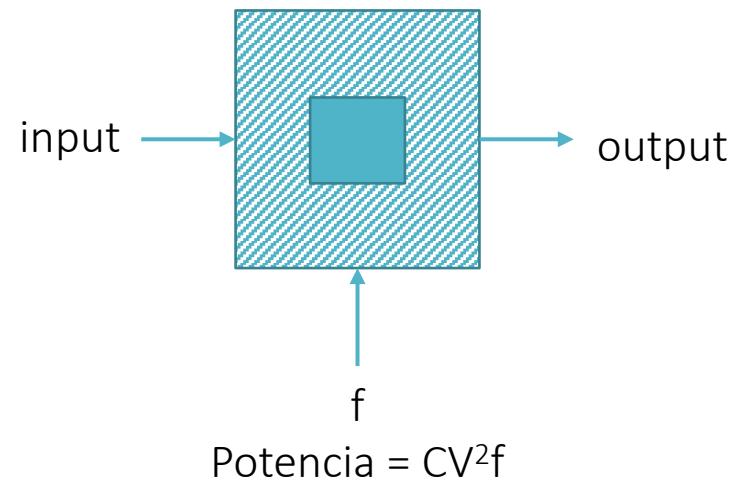
La potencia está determinada por el trabajo durante un periodo de tiempo, o lo que es lo mismo, cuantas veces por segundo se oscila el circuito.

$$P = W * F \quad (5)$$

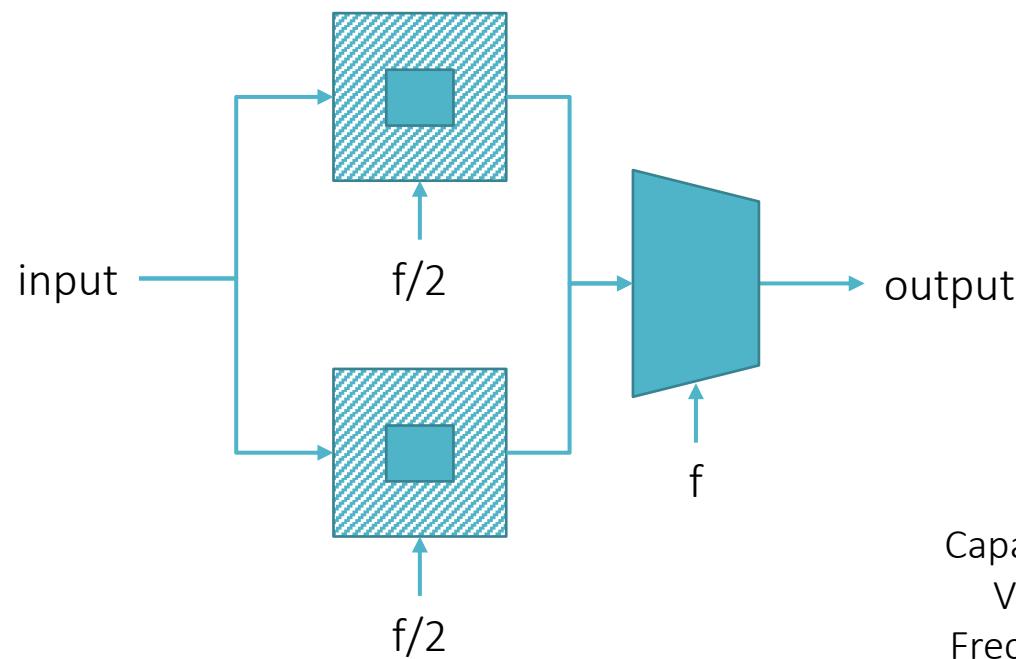
Donde P es la potencia, W es el trabajo y F es la frecuencia. Si se sustituye 3 en 4 se tiene:

$$P = C * V^2 * F \quad (6)$$

En la siguiente arquitectura se posee un procesador para atender la solicitud de entrada.



En la siguiente arquitectura se poseen dos procesadores para atender la solicitud de entrada.



Capacitancia = 2.2 C
Voltaje = 0.6 V
Frecuencia = 0.5 Hz
Potencia = 0.396 CV²f

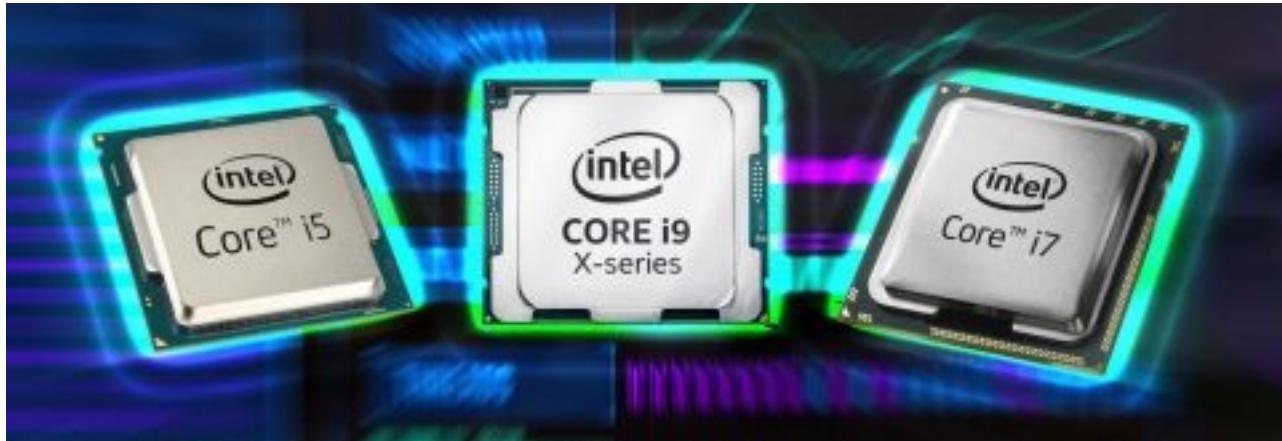
Como se puede observar en la ecuación anterior, la potencia disminuyó al 40 %, lo que representa un enorme ahorro.

Por lo anterior, la programación paralela brinda la posibilidad de generar el mismo trabajo (salida) con varios procesadores, utilizando una frecuencia más baja y ahorrando potencia.

Así mismo, se puede afirmar que el performance lo brinda el software, no el hardware.

Sin embargo, no existe un compilador que pueda transformar un programa serial en uno paralelo, por lo tanto, esa labor es solo del programador, hay que tratar de generar programas que permitan el procesamiento paralelo, en lugar del procesamiento serial.

¿Qué diferencia existe entre procesadores Intel core i3, i5, i7 e i9?



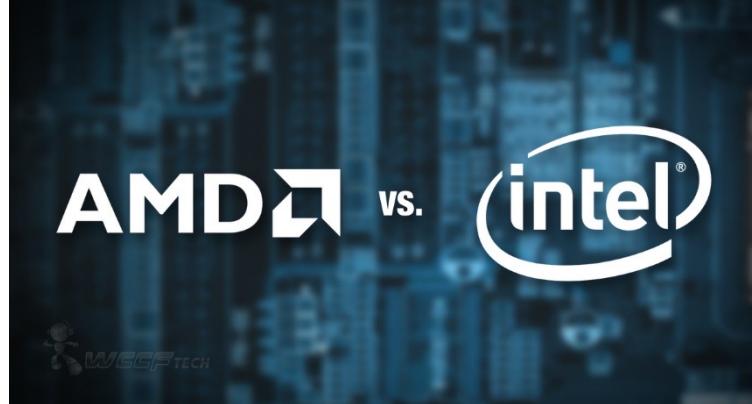
¿Qué diferencia existe entre procesadores Intel core e Intel Xeon?



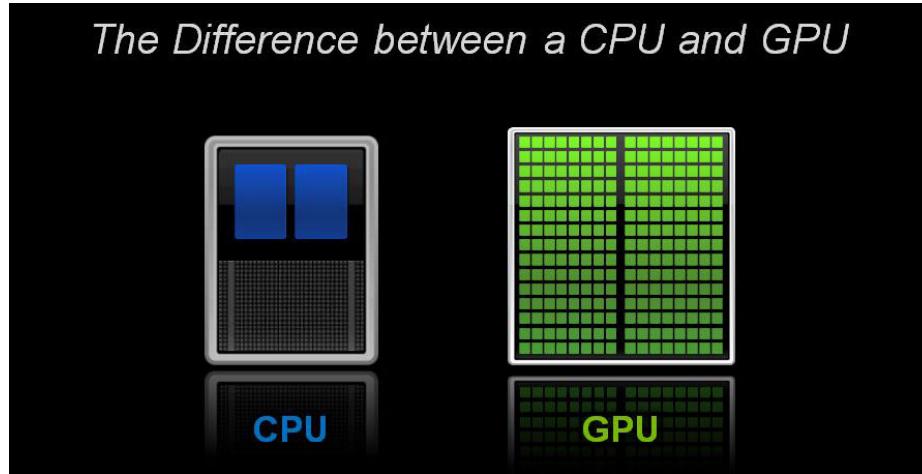
OR



¿Qué diferencia existe entre procesadores AMD e Intel?



¿Qué diferencia existe entre un CPU y un GPU?



Latencia vs rendimiento

La latencia se refiere al tiempo que se requiere para completar una tarea. Se mide en unidades de tiempo, segundos, por ejemplo.

El rendimiento se refiere al número de tareas completadas por unidad de tiempo. El rendimiento se puede medir en tareas completadas por hora, por ejemplo.

Desafortunadamente, estos dos conceptos no están necesariamente alineados.

Un CPU optimiza la latencia, ya que trata de minimizar el tiempo de una tarea.

Una GPU optimiza el rendimiento, ya que intenta maximizar el número de tareas en un lapso de tiempo.

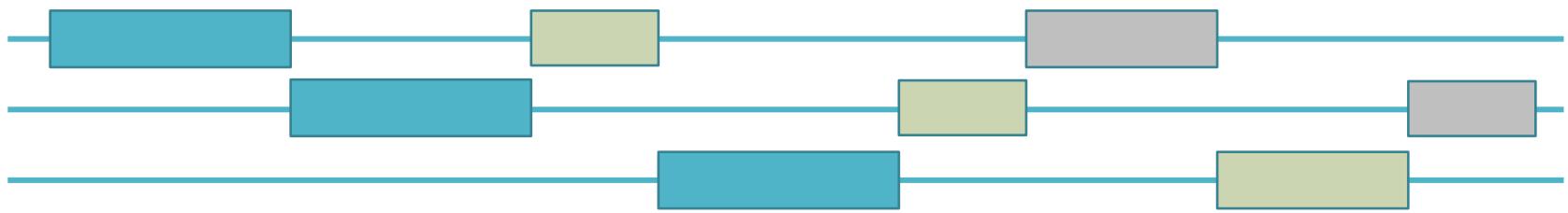
Por ejemplo, el cómputo gráfico se preocupa más de los pixeles por segundo que de la latencia de un pixel en particular.

Concurrencia vs paralelismo

La concurrencia es la condición de un sistema en la cual múltiples tareas son activadas de manera lógica, todas a la vez.

El paralelismo es la condición de un sistema en la cual múltiples tareas están realmente activas, todas a la vez.

Concurrencia



Paralelismo



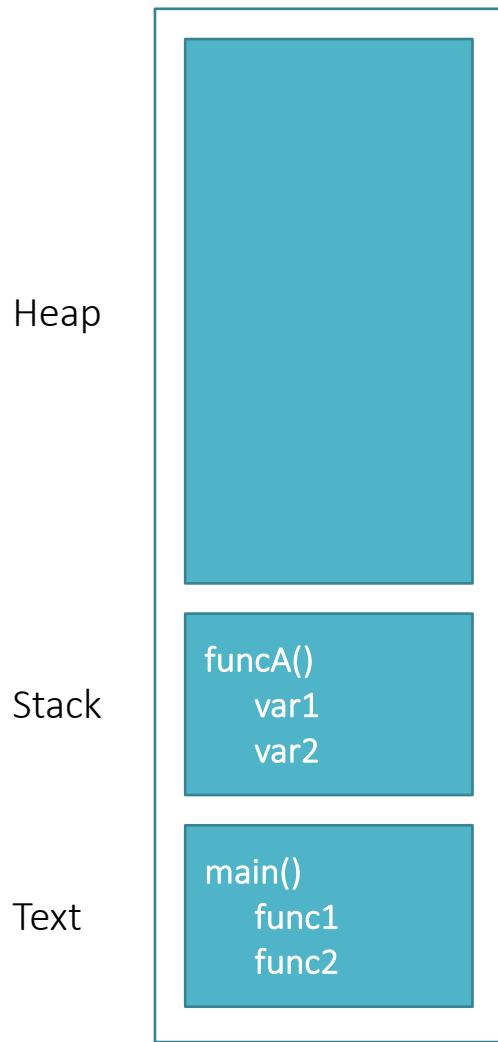
La estructura de un programa (aplicación) para su estudio de forma paralela, se puede representar de la siguiente manera:



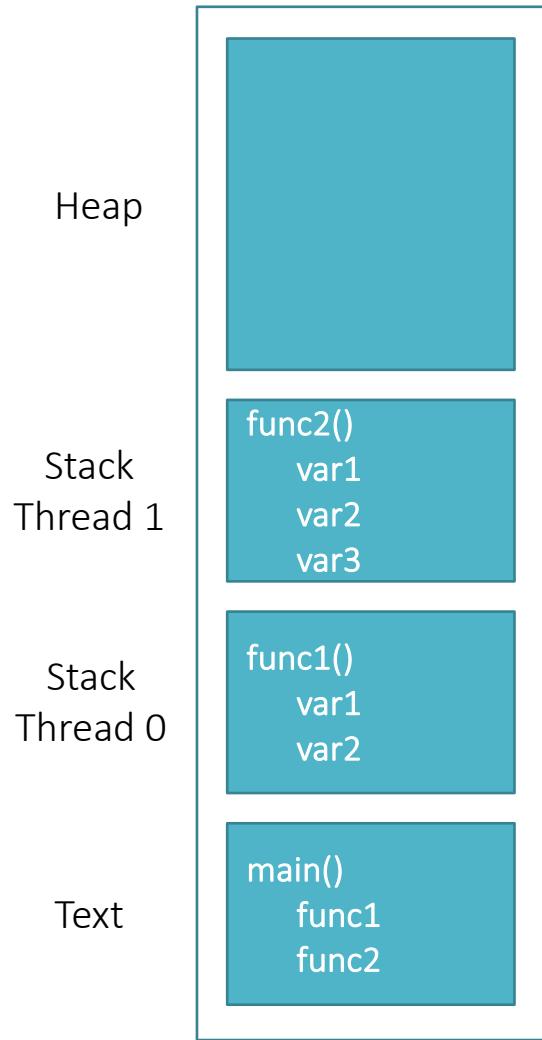
Proceso vs hilo

Un proceso es un programa en ejecución dentro de su propio espacio de direcciones.

Un hilo es una secuencia de código en ejecución dentro del contexto de un proceso, debido a que los hilos requieren la supervisión de un proceso.



Un proceso es una instancia de un programa en ejecución que se ejecuta dentro del contexto del sistema operativo (bajo la supervisión de éste).



Los hilos son procesos ligeros (light weight processes), los cuales comparten el estado del proceso que los contiene entre los diferentes hilos, reduciendo el costo de cambiar de contexto

Trabajar y controlar hilos es muy complicado por la naturaleza propia de éstos de ser procesos independientes. Además, los hilos comparten memoria, toda la información que se encuentra en el heap puede ser accedida por todos los hilos en ejecución.

Los hilos compiten por el procesador y, mientras que un hilo puede hacer la validación de un dato, otro lo puede estar modificando. Para evitar ambigüedades en la información se puede sincronizar la ejecución, pero este proceso es muy tardado y podría llevar a programar de manera serial.

Importancia de programar en paralelo

Dado un procesador con las siguientes características:

| | |
|--------------------------------|------------------------|
| Procesador Intel IVY Bridge: | 8 núcleos |
| Op vectoriales AVX por núcleo: | 8 operaciones / núcleo |
| Hyperthreading: | 2 hilos / núcleo |

Se tienen 128 posibilidades o vías de paralelismo.

Si en el procesador anterior se ejecuta un programa en lenguaje C completamente serial, se utilizará menos del 1 % de las capacidades de el equipo.

No hay duda de que la programación paralela es más complicada (al inicio) que la programación serial, sin embargo, dicha complejidad adicional vale la pena por las mejoras en rendimiento que se alcanzan.

“Fallacies of distributed computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.”

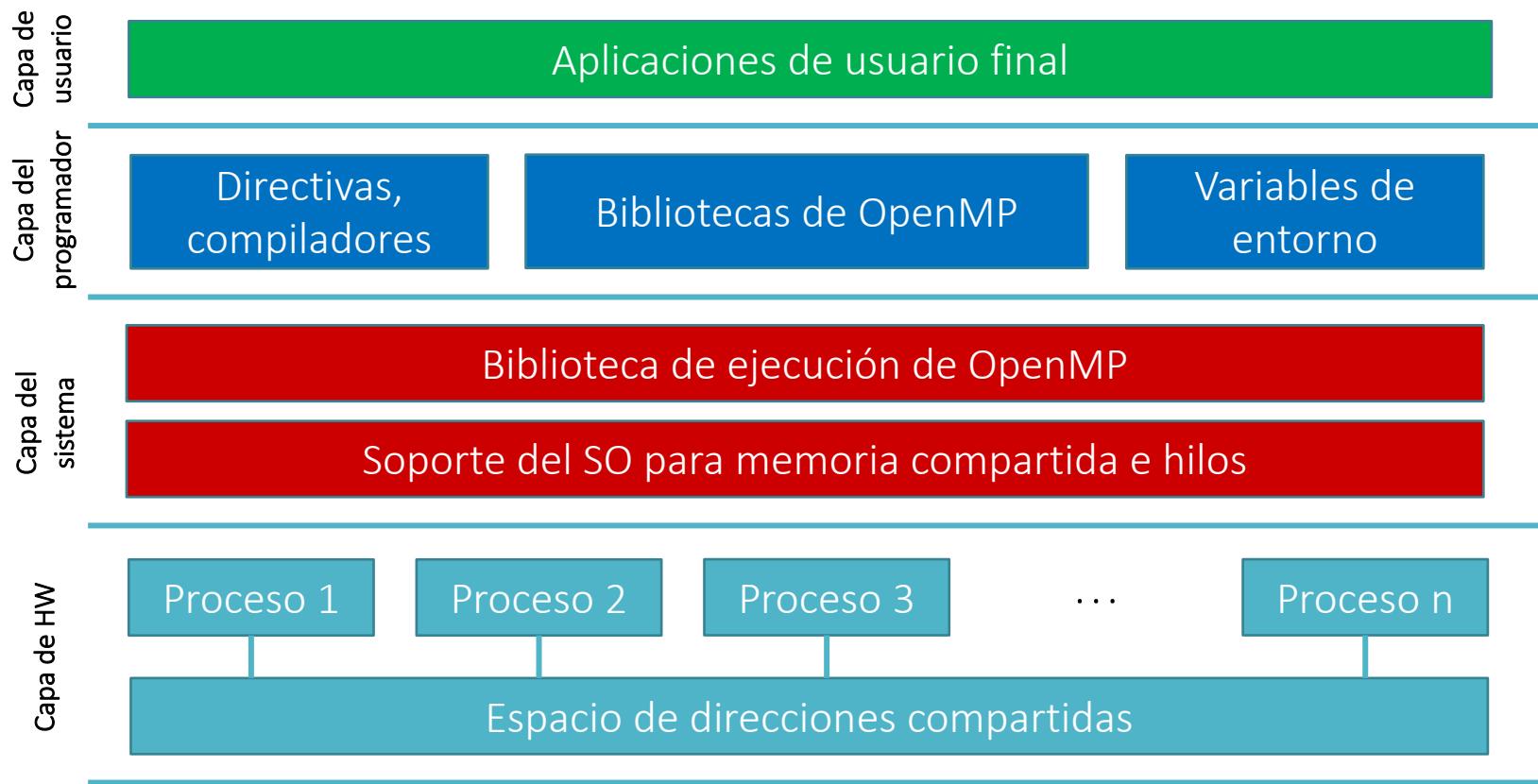
@CompSciFact

OpenMP

OpenMP es un modelo de memoria compartida que permite escribir aplicaciones multihilos, la cual contiene un conjunto de directivas y bibliotecas con funciones y rutinas que permiten desarrollar aplicaciones paralelas.

OpenMP permite transcribir de manera simple programas seriales desarrollados en Fortran, C y C++ en programas multihilo.

La pila de soluciones de los componentes básicos de OpenMP se describe a continuación:



OpenMP se puede ejecutar en diversas plataformas con diversas herramientas.



Sintaxis básica de OpenMP

Para que un archivo en lenguaje C realice programación paralela se debe incluir la biblioteca omp.h. En OpenMP la mayoría de los constructores son directivas del compilador.

```
#pragma omp construct [ clausula [clausula] ... ]  
#pragma omp parallel num_threads(4)
```

La mayoría de los constructores de OpenMP aplican a un bloque estructurado. Un bloque estructurado contiene una o más sentencias con un punto de entrada al inicio y un punto de salida al final.

Parallel

La única manera de crear hilos en OpenMP es utilizando el constructor parallel. La directiva parallel genera el número de hilos establecidos para iniciar la ejecución paralela del programa. El código a ejecutar de manera paralela está delimitado por llaves.

Las variables que se declaran fuera del bloque pragma omp parallel son compartidas por todos los hilos, las variables que se declaran dentro del bloque de la directiva se generan de manera individual para cada hilo.

Por tanto, el constructor parallel permite crear hilos en regiones paralelas, de la siguiente manera:

```
#pragma omp parallel
{
    // bloque paralelo
}
```

Cada hilo va a ejecutar una copia del código que se encuentra dentro de la estructura parallel.

Ejemplo

```
#include<stdio.h>

int main() {

    int ID = 0;
    printf("Hello(%d)", ID);
    printf("world!!(%d)\n", ID);

    return 0;
}
```

```
gcc name.c -o name
```

Ejemplo

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

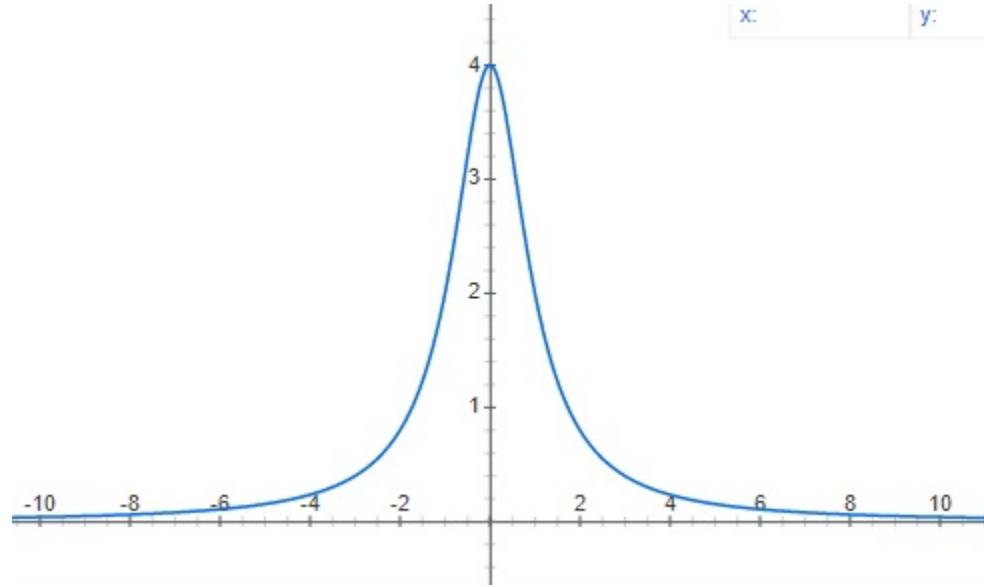
int main() {
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("Hello(%d)", ID);
        printf("world!!(%d)\n", ID);
    }
    return 0;
}
```

```
gcc –fopenmp name.c -o name
```

Ejemplo

Se desea realizar la siguiente integral:

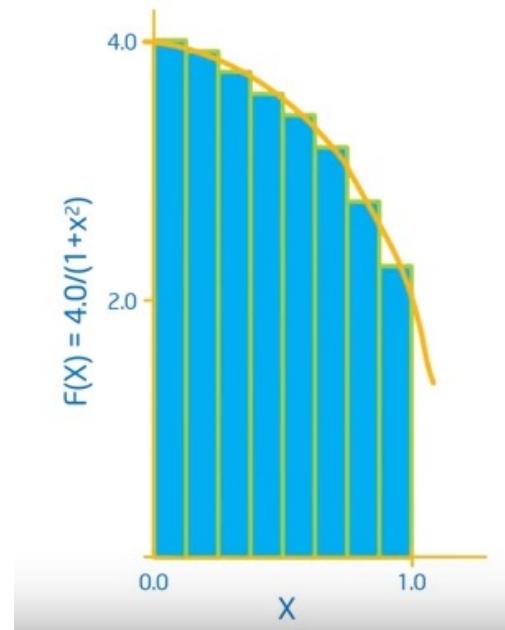
$$\int_0^1 \frac{4.0}{1 + x^2} dx = \pi$$



Ejemplo

Se desea realizar la siguiente integral:

$$\int_0^1 \frac{4.0}{1+x^2} dx = \pi$$



Ejemplo

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

static long num_steps = 100000;
double step;

int main(){
    int i;
    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("PI = %f\n", pi);
}
```

Ejercicio

Para obtener el tiempo de ejecución OpenMP tiene la siguiente función

```
double omp_get_wtime()
```

Modificar el ejemplo anterior para medir el tiempo de ejecución del programa.

Ejercicio

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

static long num_steps = 100000;
double step;

int main(){
    int i;
    double x, pi, sum = 0.0, init_time, finish_time;
    step = 1.0 / (double)num_steps;
    init_time = omp_get_wtime();
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    finish_time = omp_get_wtime()-init_time;
    pi = step * sum;
    printf("PI = %f\n", pi);
    printf("Time = %f\n", finish_time);
}
```

| Constructor | Función | Clausula |
|----------------------|----------------------|----------|
| #pragma omp parallel | omp_get_thread_num() | |
| | omp_get_wtime() | |

6.1 Niveles de paralelismo. Granularidad.

Introducción a los algoritmos paralelos



La granularidad es una forma de medir el grado o nivel de paralelismo que puede explotar un sistema, indicando la cantidad de procesamiento que pueden realizar los procesadores sin interaccionar mutuamente.

El tamaño del grano (granularidad) mide la cantidad de procesamiento necesaria en un proceso (número de instrucciones en un segmento de programa).

Un programa paralelo se puede particionar en distintos módulos concurrentes (módulos que se ejecutan en paralelo), a estos módulos se les conoce como granos. La partición se puede realizar en distintos niveles, los cuales se clasifican dependiendo del tamaño del grano.

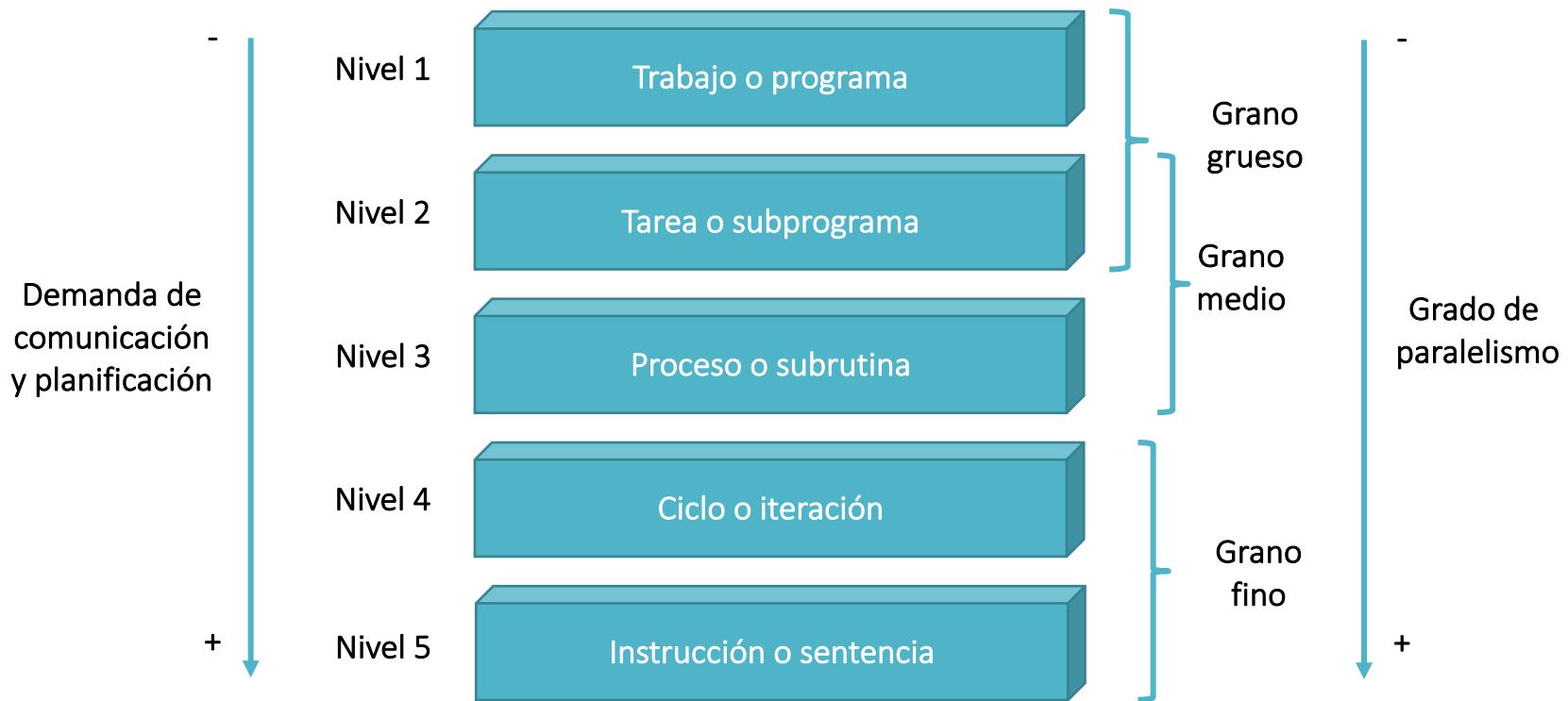
La granularidad o tamaño del grano se describen normalmente como granularidad gruesa, granularidad media y granularidad fina.

En un modelo de granularidad gruesa, el programa se divide en varias partes, las cuales precisan poca comunicación entre sí. Los sistemas paralelos formados por procesadores potentes y débilmente interconectados, parecen más adecuados para este tipo de granularidad.

En un modelo de granularidad fina, la comunicación entre los procesadores es más continua, se ejecutan relativamente pocas instrucciones sin necesidad de comunicación entre ellos. Los sistemas con procesadores sencillos y fuertemente interconectados entre sí resultan más eficientes para este tipo de granularidad.

El paralelismo puede estudiarse en varios niveles:

- Trabajo o programa: Dos programas distintos que se ejecutan en paralelo.
- Tarea o subprograma: Varias tareas se ejecutan de manera independiente, formando éstas parte de un mismo programa. Las tareas pueden interaccionar entre sí.
- Proceso o subrutina: Los procesos son bloques de instrucciones con una funcionalidad bien definida. Una serie de procesos componen una tarea.
- Ciclo o iteración: Los ciclos permiten repetir un bloque de instrucciones hasta cumplir una condición. Los ciclos son parte de un proceso.
- Instrucción o sentencia: Una instrucción puede formar parte de un ciclo o, incluso, de un proceso, los cuales son la base del cómputo.



SPMD

Single Program Multiple Data (SPMD) es una estrategia para construir algoritmos paralelos, en la cual la información global se partitiona, asignando una porción de datos a cada nodo.

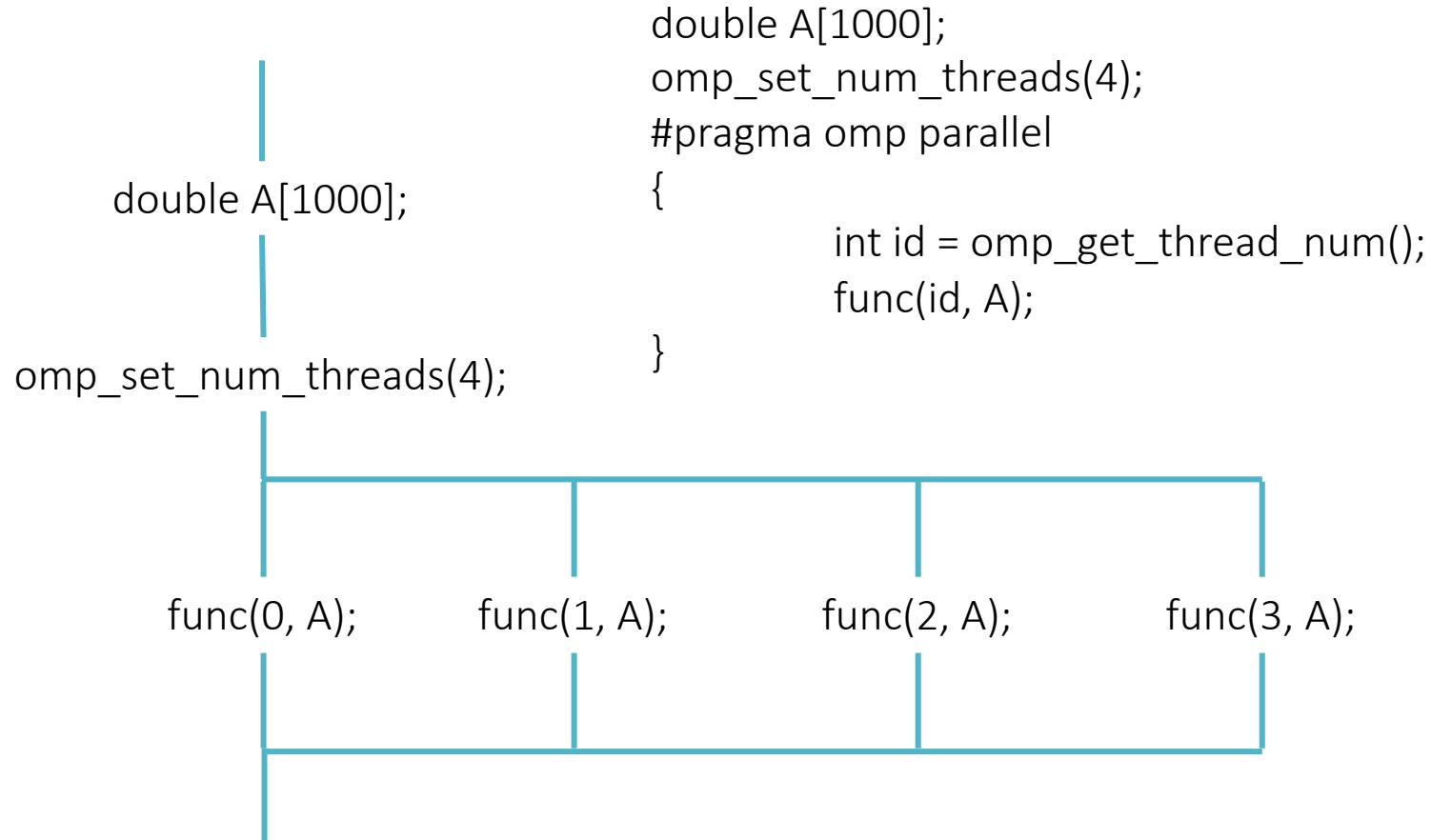
Este patrón de diseño es muy general y ha sido utilizado como soporte para la mayoría (no todos) de los patrones de estrategias de algoritmos paralelos.

Es posible establecer el número de hilos que se desean crear en el bloque paralelo de dos maneras distintas:

```
double A[1000];  
  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    // bloque paralelo  
}
```

```
double A[1000];  
  
#pragma omp parallel num_threads(4);  
{  
    // bloque paralelo  
}
```

Creación hilos en regiones paralelas



Rutinas de entorno de ejecución

OpenMP proporciona varias funciones para mostrar las cualidades del entorno paralelo en tiempo de ejecución.

Para revisar y modificar el número de hilos se tienen las funciones `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()` y `omp_get_max_threads()`.

Para saber si se está en una región paralela activa se tiene la función `omp_in_parallel()`.

También es posible variar de manera dinámica el número de hilos de un constructor paralelo a otro, para ello se utilizan las funciones `omp_set_dynamic()` y `omp_get_dynamic()`.

Para saber cuántos procesadores posee el sistema se ocupa la función `omp_get_num_procs()`.

Ejemplo

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

int main(){
    // disable dynamic adjustment of number of threads
    omp_set_dynamic(0);
    int procs = omp_get_num_procs();
    printf("Procs: %d\n", procs);
    printf("Max threads: %d\n", omp_get_max_threads());
    omp_set_num_threads(procs);
    printf("In parallel: %d\n", omp_in_parallel());
    #pragma omp parallel
    {
        int threads = omp_get_num_threads();
        printf("Threads: %d\n", threads);
        int id = omp_get_thread_num();
        printf("ID: %d\n", id);
        printf("In parallel: %d\n", omp_in_parallel());
    }
    return 0;
}
```

Ejercicio

Realizar una versión paralela del programa que permite calcular la integral definida.

Ejercicio

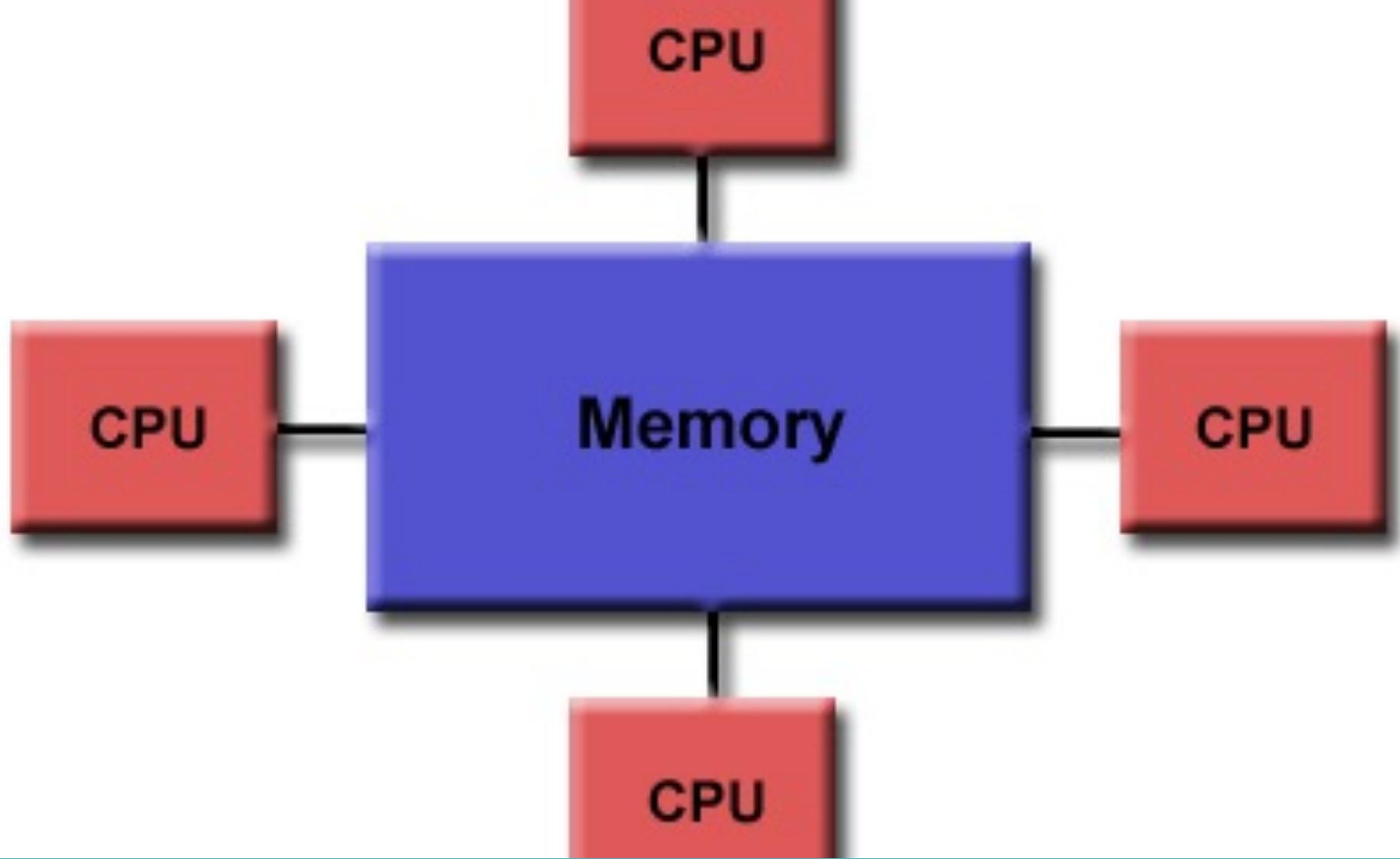
```
#include<stdio.h>
#include<omp.h>
#define NUM_THREADS 2
static long num_steps = 100000;
double step;

int main(){
    int i, nthreads;
    double pi, sum[NUM_THREADS];
    step = 1.0 / (double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0)
            nthreads = nthrds;
        for (i=id, sum[id]=0.0 ; i<num_steps ; i=i+nthrds){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for (i=0, pi=0.0 ; i < nthreads; i++)
        pi += sum[i]*step;
    printf("PI = %f\n", pi);
    return 0;
}
```

Ejercicio

Modificar el ejemplo anterior para medir el tiempo de ejecución con diversos números de hilos.

| Constructor | Función | Clausula |
|----------------------|--------------------------|----------------|
| #pragma omp parallel | omp_get_thread_num() | num_threads(4) |
| | omp_get_wtime() | |
| | omp_set_num_threads(NUM) | |
| | omp_get_num_threads() | |
| | omp_get_num_procs() | |
| | omp_set_dynamic(NUM) | |
| | omp_in_parallel() | |



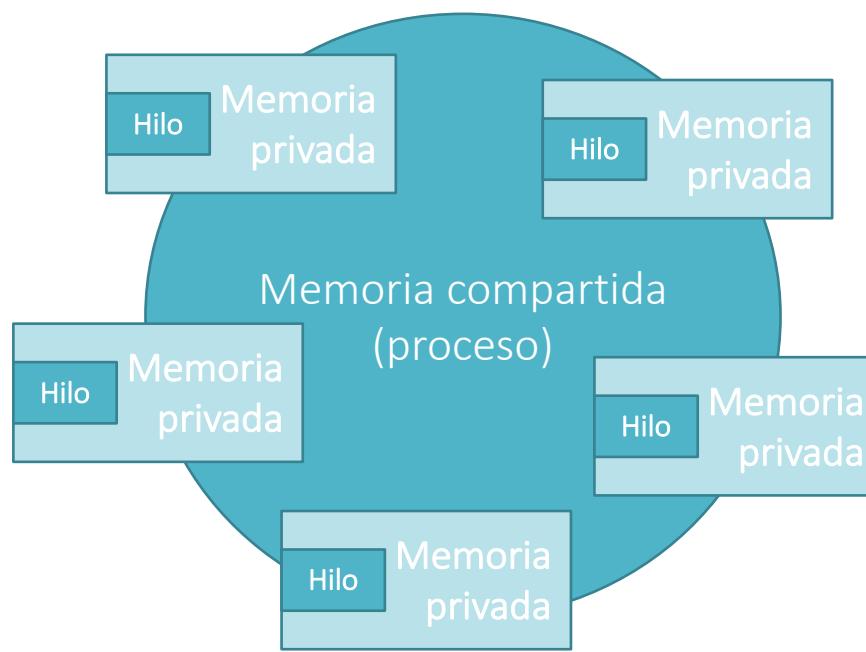
6.2 Algoritmos con memoria compartida.

Introducción a los algoritmos paralelos

Una computadora de memoria compartida es aquella que está compuesta por múltiples elementos de procesamiento, los cuales comparten un espacio de direcciones. Existen dos clases de computadoras de este tipo:

- Multiprocesador simétrico (SMP): Un espacio de memorias compartido el cual garantiza una misma cantidad de tiempo de acceso para cada procesador, el SO trata cada procesador de la misma manera.
- Espacio de direcciones multiprocesador no uniforme (NUMA): Diferentes regiones de memoria tienen un costo de acceso diferente.

Un programa de memoria compartida es una instancia que representa un proceso y puede contener muchos hilos. Los hilos pueden interactuar directamente en la memoria compartida (la memoria del proceso). El SO se encarga de decidir la ejecución de los hilos.



En OpenMP se pueden crear variables compartidas y variables privadas. Las variables compartidas son accedidas por cualquier hilo en ejecución, lo cual puede provocar inconsistencia en la información, ya que el acceso a esas variables no es controlado.

Las variables privadas son creadas para cada hilo, por tanto, cada uno posee una copia de dicha variable, sin embargo, estas variables desaparecen con el hilo, es decir, cuando éste termina su ejecución, las variables se eliminan y no hay manera de recuperarlas.

Las variables declaradas fuera de la región parallel se crean en la región de memoria compartida. También se puede establecer de manera explícita que una variable es compartida a través del método shared(), separando por coma cada variable.

Las variables declaradas dentro de la región parallel son privadas para cada hilo. También se puede establecer de manera explícita que una variable es privada a través del método private(), separando por coma cada variable.

Dado el siguiente código:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    func(id, A);
}
```

El arreglo A está en la memoria compartida. La variable id se genera para cada hilo creado. El ámbito de la variable id está delimitado por el bloque parallel. El ámbito del arreglo A está delimitado por la función que lo contiene.

Un constructor parallel crea un programa con una estructura SPMD, donde cada hilo de manera redundante ejecuta el mismo código.

Cuando se realiza trabajo compartido (worksharing) es posible separar el código durante el camino de los hilos en ejecución dentro de un equipo. Para lograr este objetivo OpenMP posee varios constructores:

- Constructor de ciclo
- Constructor de secciones
- Constructor sencillo
- Constructor de tareas

Constructor de ciclo

El constructor de ciclo (loop construct) es el más común de los constructores para realizar trabajo compartido, su sintaxis es la siguiente:

```
#pragma omp parallel
{
    int i, id;
    id = omp_get_thread_num();
    #pragma omp for
    for (i=0; i<ENE ; i++){
        do_it(id);
    }
}
```

Con la sentencia for el compilador se encarga de dividir el ciclo entre los hilos disponibles, sin necesidad de hacer más.

Cuando la sentencia for es lo único dentro del bloque paralelo, es posible unir las instrucciones pragma, es decir:

```
double res [MAX];
int i;
#pragma omp parallel for
{
    for (i=0; i<MAX ; i++){
        res[i] = do_it(i);
    }
}
```

¿El constructor Parallel no es suficiente?

Se podría pensar que con el constructor parallel (y algún método de sincronización, los cuales se verán más adelante) se tienen las herramientas suficientes para generar procesos paralelos, entonces, ¿para qué se creó el constructor de ciclo?

Dadas dos matrices A y B, se quiere obtener la suma de éstas. El programa serial quedaría de la siguiente manera:

```
for (i=0 ; i<N ; i++) {  
    a[i] = a[i] + b[i];  
}
```

En líneas de código es muy simple, sin embargo, la eficiencia es serial y, por ende, más lento el cómputo.

Si se intenta paralelizar el programa anterior, sin utilizar el constructor de ciclo, se generaría un código similar al siguiente:

```
#pragma omp parallel
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1)
        iend = N;
    for (i=istart; i<iend; i++) {
        a[i] = a[i] + b[i];
    }
}
```

Este código hace más eficiente el cómputo (porque lo paraleliza), pero se genera un código más extenso y no tan directo.

Si se paralleliza el programa inicial utilizando el constructor de ciclo, se generaría un código similar al siguiente:

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

Por tanto, el constructor de ciclo para trabajo compartido es una herramienta muy útil para generar código paralelo.

Para determinar el uso correcto del constructor de ciclo se deben tener en cuenta los siguientes aspectos:

- Encontrar los ciclos que generen la mayor cantidad de cómputo del proceso.
- Hacer que las iteraciones del ciclo sean lo más independientes posibles, para que estos se puedan ejecutar sin depender del resultado de otro ciclo.
- Colocar la directiva de OpenMP en el lugar apropiado.
- Identificar las variables que deben ser compartidas y las que deben ser privadas.

Es posible anidar múltiples ciclos, mientras que éstos se encuentren anidados, utilizando la clausula collapse(num), especificando en el argumento el número de ciclos anidados a paralelizar.

```
#pragma omp parallel for collapse (2)
for (int i=0; i<M; i++){
    for (int j=0; j<N; j++){
        // sentences
    }
}
```

OpenMP generará un solo loop de longitud MxN y ese es el que paralelizará.

Planificación

La planificación (scheduling) consiste en ordenar los granos y asignarlos a los distintos procesadores. De manera general, la planificación puede ser estática (antes de la ejecución) o dinámica (en tiempo de ejecución).

En OpenMP, la planeación afecta la manera en la que los ciclos o iteraciones están mapeadas en los hilos. Se tienen 5 tipos de planeaciones:

- Estática
- Dinámica
- Guiada
- En tiempo de ejecución
- Automática

Planificación estática

La planificación estática consiste en asignar bloques de iteraciones a cada hilo. La cantidad de los bloques se puede asignar de manera explícita. Este tipo de planificación se recomienda cuando se conoce o se puede determinar el tamaño de la muestra. La planeación estática genera menos trabajo en tiempo de ejecución, ya que ésta se realiza en tiempo de compilación.

`schedule(static [, cantidad])`

Ejemplo

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

#define THREADS 4
#define N 8

int main ( ) {
    int i;
    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        sleep(i);
        printf("Thread %d iteration %d.\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

Planificación dinámica

La planificación dinámica consiste en que cada hilo ejecuta una cantidad de iteraciones, obtenida de la cola de iteraciones, hasta que todo el procesamiento se haya realizado. Este tipo de planeación se realiza cuando no es posible predecir el tamaño de las iteraciones. La planificación dinámica genera una mayor carga de trabajo en tiempo de ejecución.

```
schedule(dynamic [, cantidad])
```

Planificación guiada

En esta planificación los hilos ejecutan bloques de iteración de manera dinámica. El tamaño del bloque es grande al principio y va disminuyendo hasta llegar a la cantidad establecida. Es un tipo especial de planificación dinámica el cual reduce la sobrecarga del planificador.

```
schedule(guided [, cantidad])
```

Planificación en tiempo de ejecución

Los valores de la planificación y la cantidad de iteraciones se toman ya sea de la variable de entorno OMP_SCHEDULE, o de la biblioteca de ejecución (runtime library). Es un tipo especial de planificación dinámica.

schedule(runtime)

Planificación automática

Los valores de la planificación y la cantidad se obtiene de alguna planificación dinámica previa, es decir, la ejecución “aprende” de ejecuciones previas del mismo ciclo.

schedule(auto)

Ejemplo

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

#define THREADS 4
#define N 8

int main ( ) {
    int i;
    #pragma omp parallel for schedule(dynamic, 4) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        sleep(i);
        printf("Thread %d iteration %d.\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

| Constructor | Función | Clausula |
|--------------------------|--------------------------|------------------|
| #pragma omp parallel | omp_get_thread_num() | num_threads(NUM) |
| #pragma omp parallel for | omp_get_wtime() | collapse(NUM) |
| #pragma omp for | omp_set_num_threads(NUM) | schedule(type) |
| | omp_get_num_threads() | |
| | omp_get_num_procs() | |
| | omp_set_dynamic(NUM) | |
| | omp_in_parallel() | |



6.2.1 Carrera de datos.

6.2 Algoritmos con memoria compartida.

En la programación multihilo que permite memoria compartida, se puede compartir información de manera involuntaria por la pelea natural de los hilos por el procesador. A esta pelea se le conoce como race conditions.

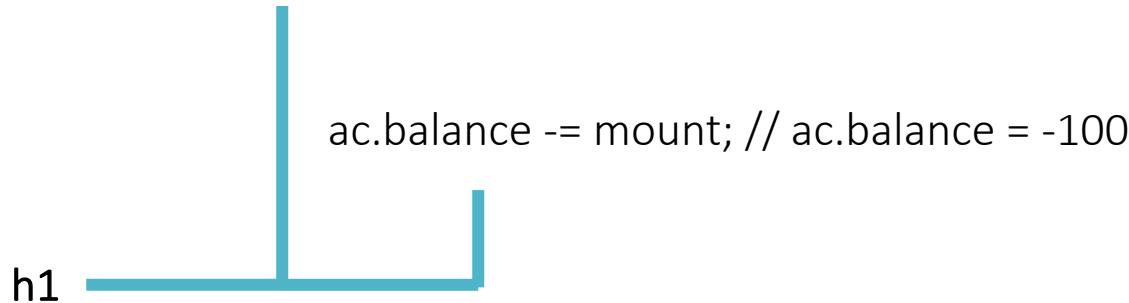
Las condiciones de carrera pueden generar pérdida o inconsistencia de información, debido a que no hay certeza de qué hilo está ejecutando qué instrucción en algún instante de tiempo.

Supóngase que se tienen dos hilos ejecutándose de manera paralela, éstos hilos realizan un retiro de efectivo en una cuenta bancaria. El código que ejecutan es el siguiente:

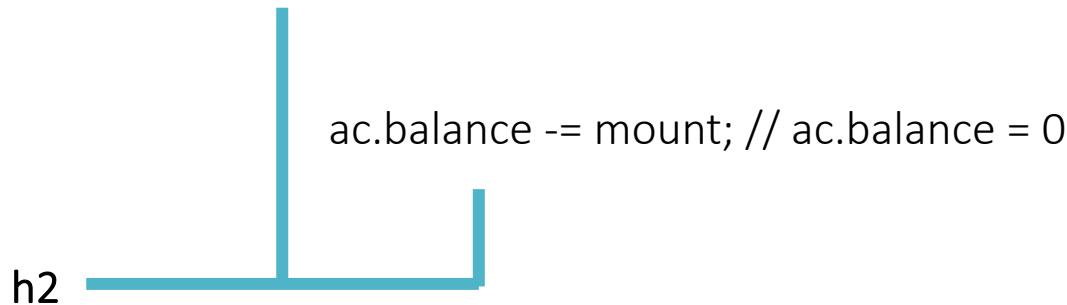
```
int withdraw (int mount, account ac)
    if (ac.balance > 0 && ac.balance >= mount){
        ac.balance -= mount;
        return 1:
    }
    return 0;
}
```

```
ac.balance = 100  
mount = 100
```

```
if (ac.balance > 0 && ac.balance >= mount) // ac.balance = 100
```



```
if (ac.balance > 0 && ac.balance >= mount) // ac.balance = 100
```





6.2.2 Inconsistencia de datos.

6.2 Algoritmos con memoria
compartida.



La carrera de datos puede provocar enormes problemas de integridad, debido a que el acceso a la información no es controlada de manera natural en el bloque paralelo.

Para controlar las condiciones de carrera se utiliza la sincronización, que consiste en proteger los datos que son propensos a entrar en conflicto. La sincronización es un proceso caro, por tanto, hay que tratar de minimizar su uso.

En OpenMP existen diferentes tipos de sincronización y, por ende, diversos constructores para sincronizar hilos. Así mismo, estos constructores se clasifican en dos niveles de sincronización: alto y bajo.

- El nivel alto de sincronización maneja cuatro constructores: critico, atómico, barrera y ordenado.
- El nivel bajo de sincronización maneja dos constructores flush y locks.

Barrier

En la sincronización por barrera cada hilo se espera en el punto marcado (la barrera) hasta que todos los hilos lleguen a ese punto. Para definir una barrera se utiliza la siguiente sentencia:

```
#pragma omp barrier
```

Ejemplo

```
#include<stdio.h>
#include<omp.h>
#define NUM_THREADS 4

int main(){
    omp_set_num_threads(NUM_THREADS);
    int A[NUM_THREADS], B[NUM_THREADS], id;
    #pragma omp parallel
    {
        id = omp_get_thread_num();
        A[id] = big_call1(id);
        #pragma omp barrier
        B[id] = big_call2(id);
    }
    return 0;
}
```

Critical

En la sincronización crítica solo un hilo a la vez puede ingresar a la región marcada. Para definir una región crítica se utiliza la siguiente sentencia:

```
#pragma omp critical
```

Ejemplo

```
#include<stdio.h>
#include<omp.h>
int main(){
    int niters = 10;
    float res = 0;
    #pragma omp parallel
    {
        float B;
        int i, id, nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        for (i=id ; i<niters ; i+=nthrds){
            B = big_job(i);
            // mutual exclusion
            #pragma omp critical
            res += consume(B);
        }
        printf("res = %d\n", res);
    }
    return 0;
}
```

Atomic

La sincronización atómica provee una exclusión mutua, pero solo aplica para la actualización de una localidad de memoria. Para definir una región atómica se utiliza la siguiente sentencia:

```
#pragma omp atomic
```

Ejemplo

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

int main(){
    int A = 0;
    float res;
    #pragma omp parallel
    {
        double tmp, B;
        B = getNumber();
        tmp = funcBig(B);
        #pragma omp atomic
        A += tmp;
    }
    return 0;
}
```

Ejercicio

Modificar el programa que calcula la integral definida, utilizando algún método de sincronización de hilos de nivel alto, para eliminar el arreglo sum.

Ejercicio

```
#include<stdio.h>
#include<omp.h>
#define NUM_THREADS 4
static long num_steps = 100000;
double step;

int main(){
    int i, nthrds;
    double pi = 0.0;
    step = 1.0 / (double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x, sum = 0.0;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0)
            nthrds = nthrds;
        for (i=id ; i<num_steps ; i=i+nthrds){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum*step;
    }
    printf("PI = %f\n", pi);
}
```

Ejercicio

¿Qué ocurre si se ingresa la sección crítica dentro del ciclo?

```
#pragma omp parallel
{
    int i, id, nthrds;
    double x, sum = 0.0;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0)
        nthrds = nthrds;
    for (i=id ; i<num_steps ; i=i+nthrds){
        x = (i+0.5)*step;
        #pragma omp critical
        sum += 4.0/(1.0+x*x);
    }
    pi += sum*step;
}
```

Ejercicio

¿Qué ocurre si se utiliza la sección atómica?

```
#pragma omp parallel
{
    int i, id, nthrds;
    double x, sum = 0.0;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0)
        nthrds = nthrds;
    for (i=id ; i<num_steps ; i=i+nthrds){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    #pragma omp atomic
    pi += sum;
}
```

nowait

Cuando se establece una barrera, los hilos deben esperar la ejecución de todos para realizar el siguiente proceso, sin embargo, la clausula nowait permite generar una excepción a esa regla, es decir, le dice al compilador, para ejecutar el siguiente código, no es necesario esperar a los otros hilos.

Ejemplo

```
#pragma omp parallel shared(A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = work1(id);
    printf("Thread %d finish work1\n", id);
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<TAM; i++){
        C[i]=work2(i,A);
    }
    printf("Thread %d finish work2\n", id);
    #pragma omp for nowait
    for (i=0; i<TAM; i++){
        B[i]=work3(C,i);
    }
    printf("Thread %d finish work3\n", id);
    A[id] = work4(id);
    printf("Thread %d finish work4\n", id);
}
```

Constructor master

El constructor master delimita el bloque de código que solo es ejecutado por el hilo principal, es decir, ese código sólo puede ser ejecutado una vez y debe ser ejecutado por el hilo 0. Los otros hilos simplemente saltan ese bloque, sin necesidad de establecer algún tipo de sincronización.

Ejemplo

```
#pragma omp parallel
{
    #pragma omp master
    {
        id = omp_get_thread_num();
        printf("Master block thread %d.\n", id);
    }

    id = omp_get_thread_num();
    printf("Parallel block thread %d.\n", id);
}
```

Constructor single

El constructor single delimita un bloque de código que sólo es ejecutado por un único hilo (este hilo no es necesariamente el hilo principal). Al final del constructor single se establece una barrera de manera implícita, la cual se puede quitar (como ya se comentó) con la cláusula *nowait*.

Ejemplo

```
#pragma omp parallel
{
    #pragma omp single
    {
        id = omp_get_thread_num();
        printf("Single block thread %d.\n", id);
    }

    id = omp_get_thread_num();
    printf("Parallel block thread %d.\n", id)
}
```

Constructor de secciones

El constructor sections proporciona una manera de segmentar un bloque estructurado (seccionado), de tal manera que cada bloque sea ejecutado por un hilo distinto. Por defecto, existe una barrera al final del bloque sections, la cual se puede quitar (como ya se comentó) con la clausula *nowait*.

Ejemplo

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        section_a(omp_get_thread_num());
        #pragma omp section
        section_b(omp_get_thread_num());
        #pragma omp section
        section_c(omp_get_thread_num());
    }

    id = omp_get_thread_num();
    printf("Parallel block thread %d.\n", id);
}
```

Funciones de bloqueo (lock)

Un bloqueo o lock es similar a una sección crítica, ya que garantiza que una instrucción sea ejecutada por un solo proceso a la vez. La principal diferencia entre un bloqueo y una sección crítica es que lock protege la información, no el código. Con un bloqueo se asegura que la información solo es accedida por un solo proceso a la vez.

Ejemplo

```
#include <stdio.h>
#include<stdlib.h>
#include <omp.h>
#define NUM_VALUES 20
int NUM_BUCKETS = 0;

int take_a_number(){
    return rand()%NUM_BUCKETS;
}

int main() {
    int i;
    omp_set_dynamic(0);
    NUM_BUCKETS = omp_get_num_procs();
    omp_set_num_threads(NUM_BUCKETS);
    printf("Buckets number = %d\n", NUM_BUCKETS);
    omp_lock_t hist_locks[NUM_BUCKETS];
    int hist[NUM_BUCKETS];
```

Ejemplo

```
#pragma omp parallel for
for (i=0; i<NUM_BUCKETS; i++){
    omp_init_lock(&hist_locks[i]);
    hist[i] = 0;
}
#pragma omp parallel for
for (i=0; i<NUM_VALUES; i++){
    int id = omp_get_thread_num();
    printf("Thread %d\n", id);
    int val = take_a_number();
    omp_set_lock(&hist_locks[id]);
    hist[val]++;
    omp_unset_lock(&hist_locks[id]);
}
for (i=0; i<NUM_BUCKETS; i++){
    printf("hist[%d] = %d\n", i, hist[i]);
    omp_destroy_lock(&hist_locks[i]);
}
}
```

Los códigos de bloqueo pueden ser muy peligrosos, ya que, si no se sincroniza correctamente, se puede generar lo que se conoce como deadlock (bloqueo de la muerte o abrazo mortal).

Un abrazo mortal o deadlock sucede cuando un hilo A se bloquea esperando la ejecución de otro hilo B, el cuál, a su vez, se bloquea esperando la ejecución del primero, debido a que ninguno termina su proceso, el bloqueo se mantienen impidiendo el flujo del programa.

Ejemplo

```
#include <stdio.h>
#include <omp.h>
#define N 100

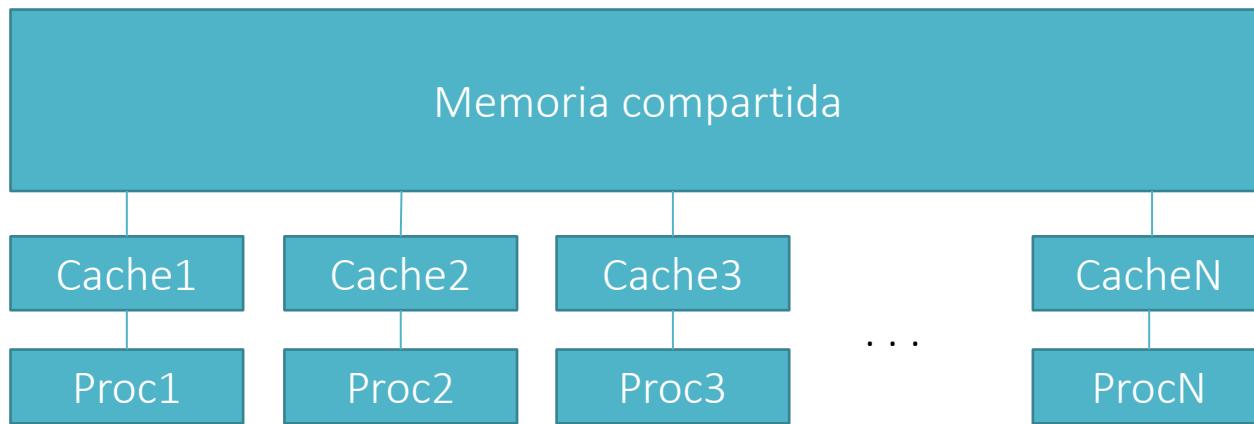
int main(){
    int a[N], b[N], i, nthreads;
    omp_lock_t locka, lockb;
    #pragma omp sections nowait
    {
        printf("Section one\n");
        #pragma omp section
        {
            omp_set_lock(&locka);
            printf("Init a\n");
            for (i=0; i<N; i++)
                a[i] = i;
            omp_set_lock(&lockb);
            printf("Init b\n");
            for (i=0; i<N; i++)
                b[i] = N - a[i];
            omp_unset_lock(&lockb);
            omp_unset_lock(&locka);
        }
    }
}
```

Ejemplo

```
#pragma omp section
{
    printf("Section two\n");
    omp_set_lock(&lockb);
    printf("Modify b\n");
    for (i=0; i<N; i++)
        b[i] = N-i;
    omp_set_lock(&locka);
    printf("Modify a\n");
    for (i=0; i<N; i++)
        a[i] = b[i] + i;
    omp_unset_lock(&locka);
    omp_unset_lock(&lockb);
}
}
return 0;
}
```

Flush

El modelo de memoria de OpenMP es un modelo de memoria compartida entre los hilos en ejecución.



Cuando un procesador trabaja con un valor **a** de la memoria compartida, éste genera una copia del valor **a** en su memoria caché, sin embargo, cuando otro procesador debe trabajar con el valor **a** de la memoria compartida, no hay certeza de qué valor es el que tendrá, si el de la memoria compartida o el de la memoria caché.

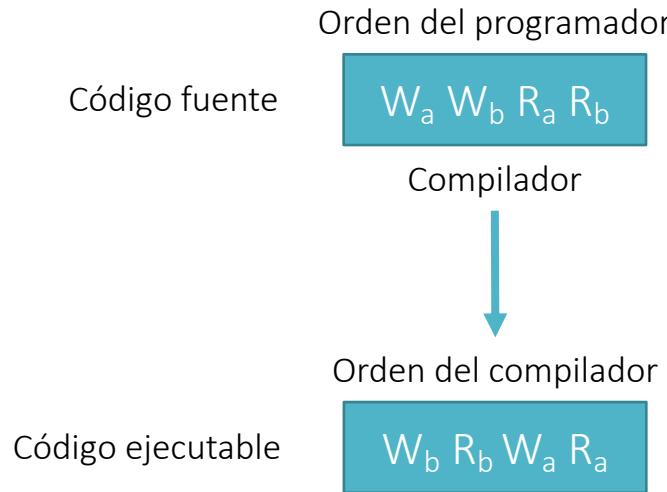
Si se desea generar un programa que se comporte bien al procesar la información, se debe tener un mecanismo para poder controlar el valor de **a** que se deseé.

Un modelo de memoria se define en términos de coherencia y consistencia.

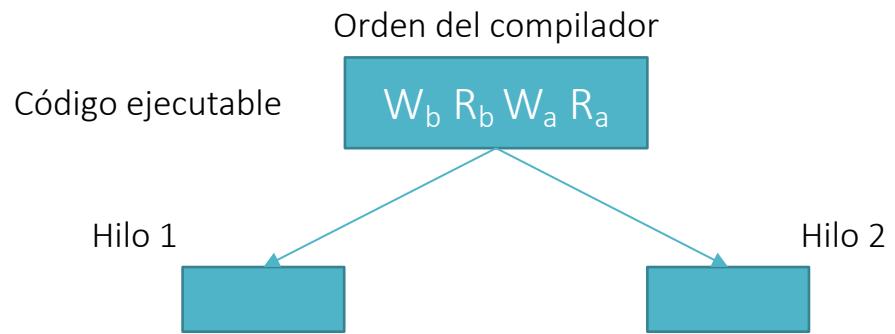
La coherencia de la memoria se refiere al comportamiento del sistema de memoria cuando una localidad de memoria es accedida por varios hilos.

La consistencia de la memoria se refiere al ordenamiento de lecturas, escrituras o sincronización (RWS) con varias localidades de memoria y por varios hilos.

Un programa posee un orden preestablecido por el programador, sin embargo, el compilador puede cambiar el orden de ejecución de ciertas instrucciones para optimizar el desempeño del código ya en ejecución.



A pesar de las mejoras que puede hacer el compilador, cuando se ejecuta el código en paralelo, la ejecución se puede realizar de diferentes maneras y no necesariamente como lo optimizó el compilador.



Un multi procesador posee consistencia secuencial en las operaciones de lectura, escritura y sincronización (RWS) si se mantiene un orden para cada procesador. Lo que significa que el orden del programador es igual al orden del compilador que es igual al orden de ejecución, lo que implica un desastre en el performance.

Por otro lado, también existe la consistencia relajada, la cual consiste en eliminar algunas de las restricciones de orden para las operaciones de memoria RWS.

Constructor flush

OpenMP define su consistencia como una variante de la consistencia débil, en la cual no se puede reordenar las operaciones de sincronización con las operaciones de lectura o escritura en el mismo hilo. La consistencia débil garantiza:

$$S \gg W, S \gg R, R \gg S, W \gg S, S \gg S$$

Para realizar esta operación de sincronización se utiliza al constructor flush.

Flush define una secuencia de puntos en la cual garantiza que un hilo puede acceder a una vista consistente de la memoria con respecto a un conjunto flush.

Un conjunto flush está compuesto por todas las variables que pueden ser vistas por un hilo para un constructor flush de manera consistente. Por lo tanto, el conjunto flush es una lista de variables que utilizan los hilos cuando el constructor flush es utilizado.

El uso de flush es un tema en verdad avanzado, no es trivial saber donde establecer el constructor. Es por ello que OpenMP utiliza el constructor flush de manera implícita en diversas operaciones sin necesidad de que el programador se preocupe por establecer el constructor correspondiente:

- A la entrada o salida de regiones paralelas.
- En las barreras implícitas o explícitas.
- A la entrada o salida de regiones críticas.
- Cuando se establece o se quita un bloqueo.

Por tanto, es mejor no utilizar el constructor flush para evitar inconsistencia de datos, a menos que se tenga claro el concepto y la implementación.

Ejemplo

```
int main(){
    double sum, runtime;
    int flag = 0;
    A = (double *) malloc(N*sizeof(double));
    runtime = omp_get_wtime();
    fill_rand();
    sum = sum_array();
    runtime = omp_get_wtime()-runtime;
    printf("Sum = %lf\nRuntime = %lf\n", sum, runtime);
    return 0;
}
```

Ejemplo

```
#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    {
        fill_rand();
        #pragma omp flush      // flush the array
        flag = 1;
        #pragma omp flush(flag)
    }
    #pragma omp section
    {
        #pragma omp flush(flag)
        while (flag == 0) {
            // For the compiler not to copy the flag value on the register
            #pragma omp flush(flag)
        }
        #pragma omp flush      // flush the array
        sum = sum_array();
    }
}
```

El programa anterior genera el resultado correcto el 99.99 % de las veces, lo que implica que no es 100 % confiable.

En el código anterior se puede ver que existen condiciones de carrera, las cuales se pueden solventar utilizando el constructor atomic. Atomic ayuda a proteger las operaciones de la memoria de manera atómica, para ello se expande su uso de la siguiente manera:

```
#pragma omp atomic [read | write | update | capture]
```

Por tanto, el constructor atomic puede proteger la carga de los valores a memoria. Entonces, si se tiene:

```
#pragma omp atomic read  
v = x
```

El constructor atomic garantiza que el valor que se está asignando a v es el final, el completo, no una aproximación (se espera hasta que el valor sea cargado en su totalidad).

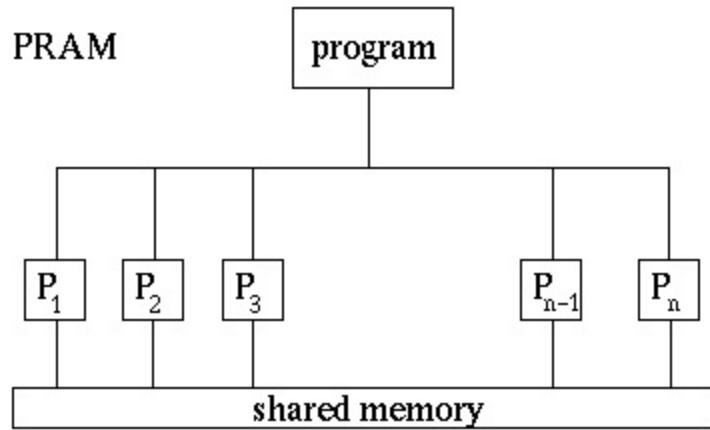
Ejemplo

```
#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    {
        fill_rand();
        #pragma omp flush
        #pragma omp atomic write
        flag = 1;
        #pragma omp flush(flag)
    }
    #pragma omp section
    {
        while (1) {
            #pragma omp flush(flag)
            #pragma omp atomic read
            tmp_flag = flag;
            if (tmp_flag == 1)
                break;
        }
        #pragma omp flush
        sum = sum_array();
    }
}
```

| Constructor | Función | Clausula |
|--------------------------|--------------------------------|------------------|
| #pragma omp parallel | omp_get_thread_num() | num_threads(NUM) |
| #pragma omp parallel for | omp_get_wtime() | collapse(NUM) |
| #pragma omp for | omp_set_num_threads(NUM) | schedule(type) |
| #pragma omp barrier | omp_get_num_threads() | Nowait |
| #pragma omp critical | omp_get_num_procs() | shared(vars) |
| #pragma omp atomic | omp_set_dynamic(NUM) | private(vars) |
| #pragma omp master | omp_in_parallel() | |
| #pragma omp single | omp_init_lock(omp_lock_t *) | |
| #pragma omp sections | omp_destroy_lock(omp_lock_t *) | |
| #pragma omp flush | omp_set_lock(omp_lock_t *) | |
| | omp_unset_lock(omp_lock_t *) | |

6.2.3 Modelo PRAM.

6.2 Algoritmos con memoria compartida.



Modelo RAM

El modelo RAM se refiere a la representación de una computadora hipotética que permite evaluar la eficiencia del diseño de un algoritmo de manera independiente de la arquitectura (hardware) donde se implemente.

El modelo RAM asume las siguientes reglas:

- Cada operación simple (+, -, *, /, selección o llamada) toma exactamente un paso de tiempo.
- Los ciclos están compuestos por operaciones simples, por lo tanto, el tiempo que toma un ciclo depende del número de iteraciones del mismo.
- El acceso a memoria toma exactamente un paso de tiempo. La memoria de un modelo RAM es infinita.

Un modelo RAM permite medir el tiempo que consume la ejecución de un algoritmo, contando el número de pasos que contiene el mismo. Si se asume que un modelo RAM ejecuta un cierto número de pasos por segundo, en automático se puede obtener el tiempo de ejecución de un proceso.

La RAM es un modelo simple que intenta representar el funcionamiento de un equipo, y, por tanto, hay que tener ciertas consideraciones.

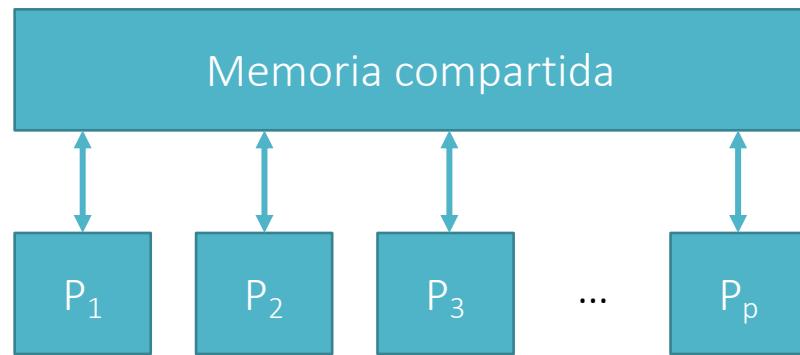
En la mayoría de los procesadores, multiplicar dos números toma más tiempo que sumarlos. Esto viola la primera regla del modelo.

Además, los procesos paralelos (multihilos) así como ciertos compiladores pueden violar la segunda regla.

Por si fuera poco, el acceso a la memoria difiere en tiempo dependiendo del lugar donde se encuentren los datos, con lo que se viola la tercera regla.

Modelo PRAM

El modelo PRAM (Parallel Random Access Machine) es la generalización sencilla y natural del modelo RAM, ya que cada procesador se modela como RAM. Cada procesador opera de manera síncrona. Este modelo es el más utilizado y conocido para modelar la eficiencia del procesamiento paralelo.



El modelo PRAM se refiere a una máquina computacional formada por varios procesadores, donde cada procesador comparte un espacio de memoria común. Todos los procesadores comparten una misma señal de reloj, lo cual permite ejecutar instrucciones de manera síncrona. Así mismo, posee mecanismos para resolver conflictos de acceso concurrente a la información.

El modelo síncrono PRAM está diseñado para una máquina ideal SIMD de memoria compartida. En el modelo SIMD (Single Instruction Multiple Data) se tiene una instrucción que puede operar sobre algunos montones de información al mismo tiempo. Por ejemplo, cuando se realiza una multiplicación de matrices de manera paralela.

Dentro del modelo PRAM un paso (ciclo de reloj) consiste de tres etapas:

- Leer: cada procesador puede leer un valor de la memoria compartida.
- Ejecutar: cada procesador va a computar operaciones con su información local.
- Escribir: cada procesador puede escribir un valor en la memoria compartida.

Así mismo, el modelo PRAM permite manejar acceso de lectura o escritura concurrente:

- Lectura exclusiva / escritura exclusiva (EREW)
- Lectura concurrente / escritura exclusiva (CREW)
- Lectura concurrente / escritura concurrente (CRCW)

En el modelo PRAM de lectura / escritura concurrente (CRCW) se pueden tener los siguientes escenarios:

- Lectura / escritura concurrente común: todos los procesadores deben escribir sobre el mismo valor.
- Lectura / escritura concurrente arbitraria: solo un procesador puede escribir.
- Lectura / escritura concurrente prioritaria: el procesador con la prioridad más alta (master) puede escribir.

Suponiendo que existen dos modelos PRAM, y que ambos resuelven el mismo problema, se puede afirmar que el más poderoso es aquel que cumple con:

- El tiempo de complejidad es asintóticamente menor para resolver un problema en uno que en otro, o.
- En modelos que tienen el mismo tiempo de complejidad, pero la complejidad de trabajo es asintóticamente menor en uno que en otro.

De los modelos mostrados, se muestran a continuación del menos poderoso al más poderoso:

- EREW
- CREW
- CRCW común
- CRCW arbitrario
- CRCW prioritario

Se puede afirmar que un algoritmo diseñado en un modelo menos poderoso se puede ejecutar en la misma complejidad tanto de tiempo como de trabajo si se utiliza un modelo más poderoso.

Así mismo, un algoritmo diseñado para un modelo poderoso puede ser simulado en un modelo más débil utilizando más procesadores o llevando más tiempo su ejecución.

Suponiendo que se tiene un modelo CRCW prioritario y se quiere simular en un modelo EREW, se tiene el siguiente teorema:

Un algoritmo que se ejecuta en T tiempo en el pe-ésimo procesador del modelo CRWR prioritario, puede ser simulado por el modelo PRAM EREW para se ejecutado en un tiempo $O(T \log p)$.

Ejemplo

Multiplicación de matrices

Dada una multiplicación de matrices de NxN utilizando n procesadores, cada elemento C_{ij} dado por

$$C_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Se puede computar utilizando el modelo PRAM CREW en paralelo utilizando n procesadores en un tiempo $O(\log(n))$.

Ejemplo

Por otro lado, si se utiliza el modelo PRAM EREW, las lecturas excluyentes de los valores de a_{ij} y b_{ij} se pueden satisfacer creando n copias de a y b , lo cual toma un tiempo $O(\log(n))$ utilizando n procesadores

A pesar de que el tiempo sigue siendo logarítmico, el requerimiento de memoria es mayor, debido a las n copias de a y b .

6.3 Técnicas de desarrollo de algoritmos.

Introducción a los
algoritmos paralelos



La diferencia entre un programador novel y un programador experto en programación paralela no es el hecho de conocer todas las directivas de un lenguaje, si no saber los patrones en los que se basa un algoritmo paralelo.

Por ello es importante conocer los principales patrones de diseño de algoritmos paralelos y saber que, cuando se genera un programa paralelo, en el fondo, se está ocupando (se desee o no) un patrón de diseño.

SPMD (Single Program Multiple Data)

Este patrón de diseño ejecuta el mismo programa en las p unidades de procesamiento que se tengan. Para ello, utiliza un rango de procesadores que se identifican desde 0 hasta $P-1$, para ejecutar un conjunto de tareas para manipular cualquier estructura de datos compartida. Este es el patrón de diseño más común y el más utilizado.

Ejemplo

```
#pragma omp parallel
{
    int i, id, nthrds;
    double x, sum = 0.0;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0)
        nthrds = nthrds;
    for (i=id ; i<num_steps ; i=i+nthrds){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    #pragma omp atomic
    pi += sum;
}
```

Paralelismo de ciclos (Loop parallelism)

Este patrón describe colecciones de tareas que son definidas como iteraciones de uno o más ciclos. Estas iteraciones se dividen entre una colección de elementos de procesamiento para realizar tareas independientes en paralelo.

Ejemplo

```
int main(){
    int i;
    double x, pi, sum = 0.0, init_time, finish_time;
    init_time = omp_get_wtime();
    step = 1.0 / (double)num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    finish_time = omp_get_wtime()-init_time;
    pi = step * sum;
    printf("PI = %f\n", pi);
    printf("Time = %f\n", finish_time);
}
```

Divide y vencerás

Se utiliza cuando un problema incluye un método que se divide en sub problemas y existe al final una manera de combinar las soluciones de los sub problemas para generar una solución global.

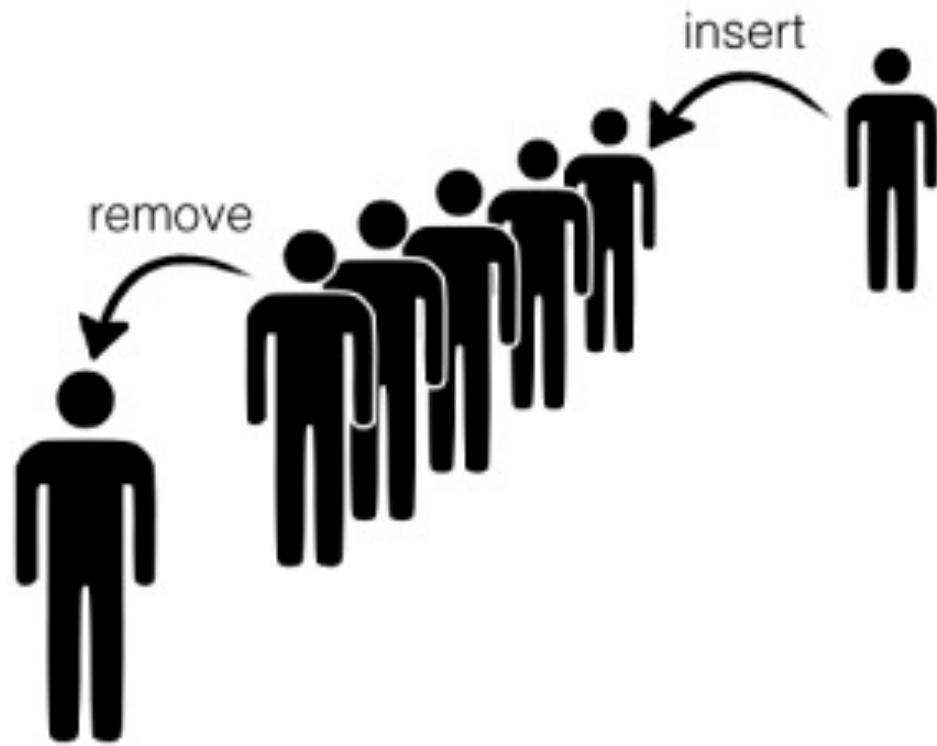
Este patrón se caracteriza por tener una operación de separación del problema hasta que éste se pueda resolver de manera directa, para al final combinar las soluciones de los problemas en una solución global.

Ejemplo

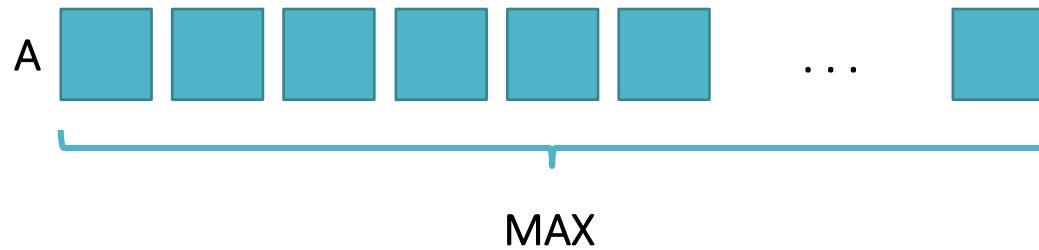
```
int fibonacci_p(int n){  
    int x, y;  
    printf("%d\n", omp_get_thread_num());  
    if (n<2)  
        return n;  
    #pragma omp task shared(x)  
    x = fibonacci_p(n-1);  
    #pragma omp task shared(y)  
    y = fibonacci_p(n-2);  
  
    #pragma omp taskwait  
    return x + y;  
}  
  
int main () {  
    #pragma omp parallel  
    res = fibonacci_p(NUM);  
}
```

6.3.1 Rediseño de estructuras de datos.

6.3 Técnicas de desarrollo
de algoritmos.



Se desea obtener el promedio de un conjunto de elementos, ¿el programa se puede paralelizar?



```
#include<stdio.h>
#include<omp.h>
#define MAX 5

int main() {
    double ave=0.0, A[MAX];
    int i;
    for (i=0; i<MAX; i++) {
        A[i] = i+1.0;
    }

#pragma omp parallel for
for (i=0; i<MAX; i++) {
    ave += A[i];
}
ave /= MAX;
printf("%f\n",ave);
return 0;
}
```

El ciclo es completamente dependiente de los bloques de iteraciones contiguos.

Reducción

Cuando, en una estructura de datos, se desea realizar un cálculo entre todos los elementos del conjunto, pero existe una dependencia entre los ciclos y esta dependencia no es posible eliminarla, se puede utilizar un patrón conocido como reducción.

La mayoría de los entornos de programación paralelo proporcionan soporte para las operaciones de reducción.

En OpenMP la reducción se realiza de la siguiente manera:
reduction(operation: list)

Ya sea dentro de un bloque paralelo o dentro de un constructor de ciclo se cumplen las siguientes premisas al realizar una reducción:

- Una copia local de cada variable es creada e inicializada según la *operation* que se haya definido.
- Las actualizaciones de las variables especificadas se realizan en una copia local.
- Las copias locales son reducidas en un solo valor y combinado con el valor original global.

Los operandos con sus valores iniciales se muestran a continuación:

| Operador | Valor inicial |
|----------|---------------------|
| + | 0 |
| * | 1 |
| - | 0 |
| min | Positivo más grande |
| max | Valor más negativo |

| Operador | Valor inicial |
|----------|---------------|
| & | ~0 |
| | 0 |
| ^ | 0 |
| && | 1 |
| | 0 |

Ejemplo

```
#include<stdio.h>
#include<omp.h>
#define MAX 5

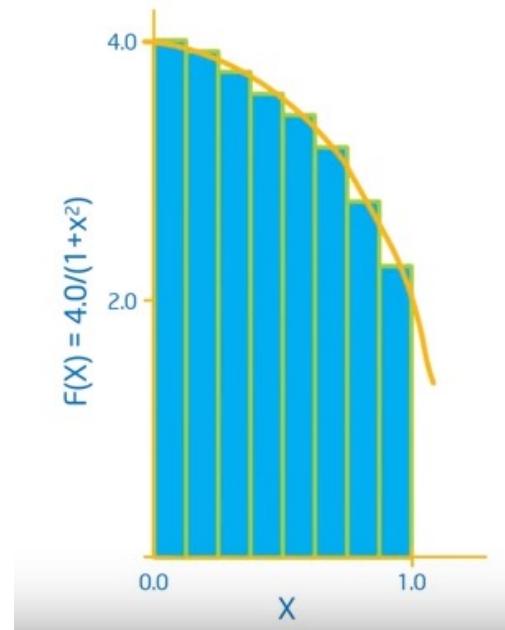
int main() {
    double ave=0.0, A[MAX];
    int i;
    for (i=0; i<MAX; i++) {
        A[i] = i+1.0;
    }

#pragma omp parallel for reduction(+: ave)
    for (i=0; i<MAX; i++) {
        ave += A[i];
    }
    ave /= MAX;
    printf("%f\n",ave);
    return 0;
}
```

Ejemplo

Dada la integral:

$$\int_0^1 \frac{4.0}{1 + x^2} dx = \pi$$



Ejemplo

A partir del siguiente código, qué cambios se tienen que realizar para obtener el valor de la integral de manera paralela, de forma más eficiente.

```
#include<stdio.h>
#include<omp.h>
static long num_steps = 100000;
double step;
int main(){
    int i;
    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("PI = %f\n", pi);
}
```

Private y firstprivate

Como ya se comentó, por defecto, OpenMP ejecuta un entorno de datos de memoria compartida, donde la mayoría de las variables son compartidas. Las variables globales y las variables declaradas en una función fuera de la región paralela son compartidas por todos los hilos en la región paralela.

Sin embargo, las variables declaradas dentro de la región paralela, así como las variables declaradas en funciones que son llamadas dentro de una región paralela son variables PRIVADAS, es decir, existe una referencia a ellas en el stack de cada hilo.

Ejemplo

```
void wrong_func(){  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j<1000; ++j)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

tmp no está inicializada

tmp vale 0

Ejemplo

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

void func(){
    int tmp = 0;
    #pragma omp parallel for firstprivate(tmp)
    for (int j = 0; j<1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

Dadas las siguientes condiciones:

variables: A = 1, B = 1, C = 1

```
#pragma omp parallel private(B) firstprivate(C)
```

¿Qué variables son compartidas y qué variables son privadas dentro de la región paralela?

¿Cuáles son los valores iniciales de cada variable dentro de la región paralela?

¿Cuáles son los valores finales de cada variable fuera de la región paralela?

Ejercicio

Modificar el programa de la integral definida serial de tal manera que, haciendo la menor cantidad de modificaciones, se calcule la integral utilizando procesamiento paralelo.

Ejercicio

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

static long num_steps = 100000;
double step;

int main(){
    int i;
    double x, pi, sum = 0.0, init_time, finish_time;
    init_time = omp_get_wtime();
    step = 1.0 / (double)num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    finish_time = omp_get_wtime()-init_time;
    pi = step * sum;
    printf("PI = %f\n", pi);
    printf("Time = %f\n", finish_time);
}
```

threadprivate

La clausula `threadprivate` hace que los datos globales sean privados para un hilo.

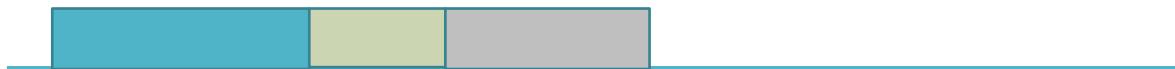
Una variable `private` enmascara una variable global (a pesar de ser declarada fuera de la región paralela, el valor solo existe dentro de la ejecución del hilo). Una variable `threadprivate` preserva el alcance global con cada hilo.

Ejercicio

```
int counter = 0;  
#pragma omp threadprivate(counter)  
  
int increment_counter(){  
    return ++counter;  
}
```

Constructor de tareas

Una tarea se encarga de manejar unidades independiente de trabajo dentro de un equipo.



Serial

Tasks

Una tarea se encarga de manejar unidades independiente de trabajo dentro de un equipo.



Paralelo

Las tareas están compuestas por:

- El código a ejecutar.
- El entorno de datos.
- Variables de control internas.

Los hilos realizan el trabajo de cada una de las tareas definidas. El sistema es el que decide cuándo se va a ejecutar una tarea.

Para declarar una tarea se utiliza el siguiente constructor:

```
#pragma omp task
```

La serie de instrucciones (código) se asigna a los datos creados cuando un hilo encuentra con la región del constructor task.

Para garantizar que una tarea (o conjunto de tareas) han concluido, se cuenta con 2 elementos:

- Barrera de hilos: #pragma omp barrier
- Barrera de tareas: #pragma omp taskwait

Ejercicio

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

int main(){
    #pragma omp parallel
    {
        #pragma omp task
        do_it();
        #pragma omp barrier
        #pragma omp single
        {
            #pragma omp task
            end_it();
        }
    }
    return 0;
}
```

Pero, en realidad, ¿cómo trabaja internamente una tarea? ¿Cómo es posible que una tarea pueda tener unidades independientes de trabajo?

Para mostrar el funcionamiento de las tareas se va a utilizar como caso de estudio la estructura de datos lista simple. Una lista simple posee una referencia hacia el primer elemento de la estructura, a través de esta referencia se puede acceder a todos los datos de la lista, recorriendo los valores a los que apunta cada nodo de manera cíclica. Las operaciones básicas de una lista son insertar, eliminar y mostrar.

Dentro de las operaciones básicas de una lista, la única que se presta para generar en paralelo es la función mostrar.

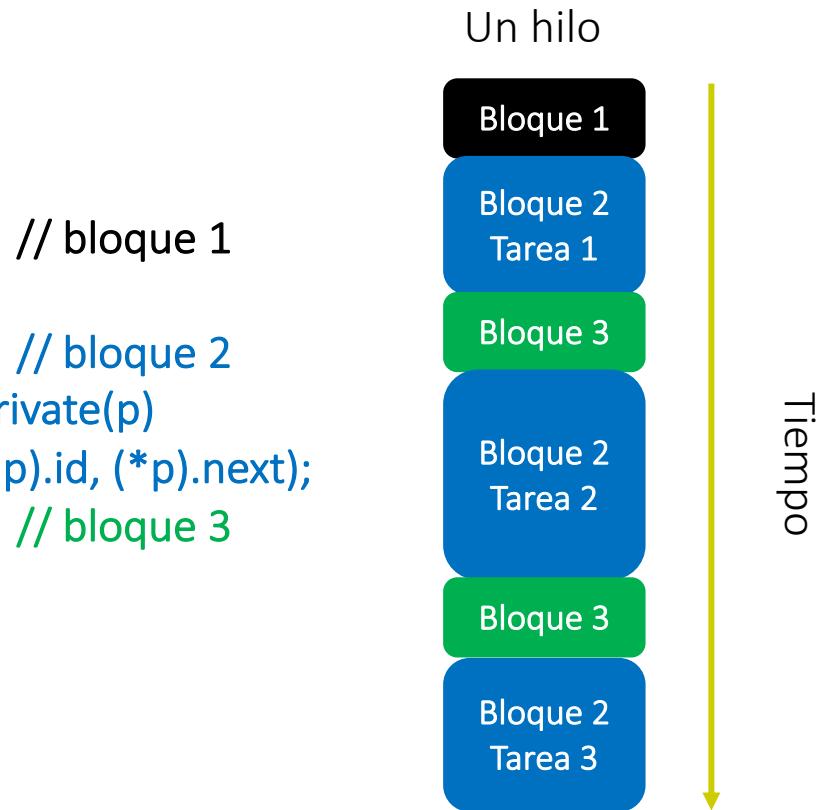
```
#pragma omp parallel
{
    #pragma omp single
    {
        struct item *p = list;
        while (p!=NULL){
            #pragma omp task firstprivate(p)
            printf("id: %d ->%d\t", (*p).id, (*p).next);
            p = (*p).next;
        }
    }
}
```

Dentro del programa anterior se pueden identificar 3 bloques importantes para su análisis.

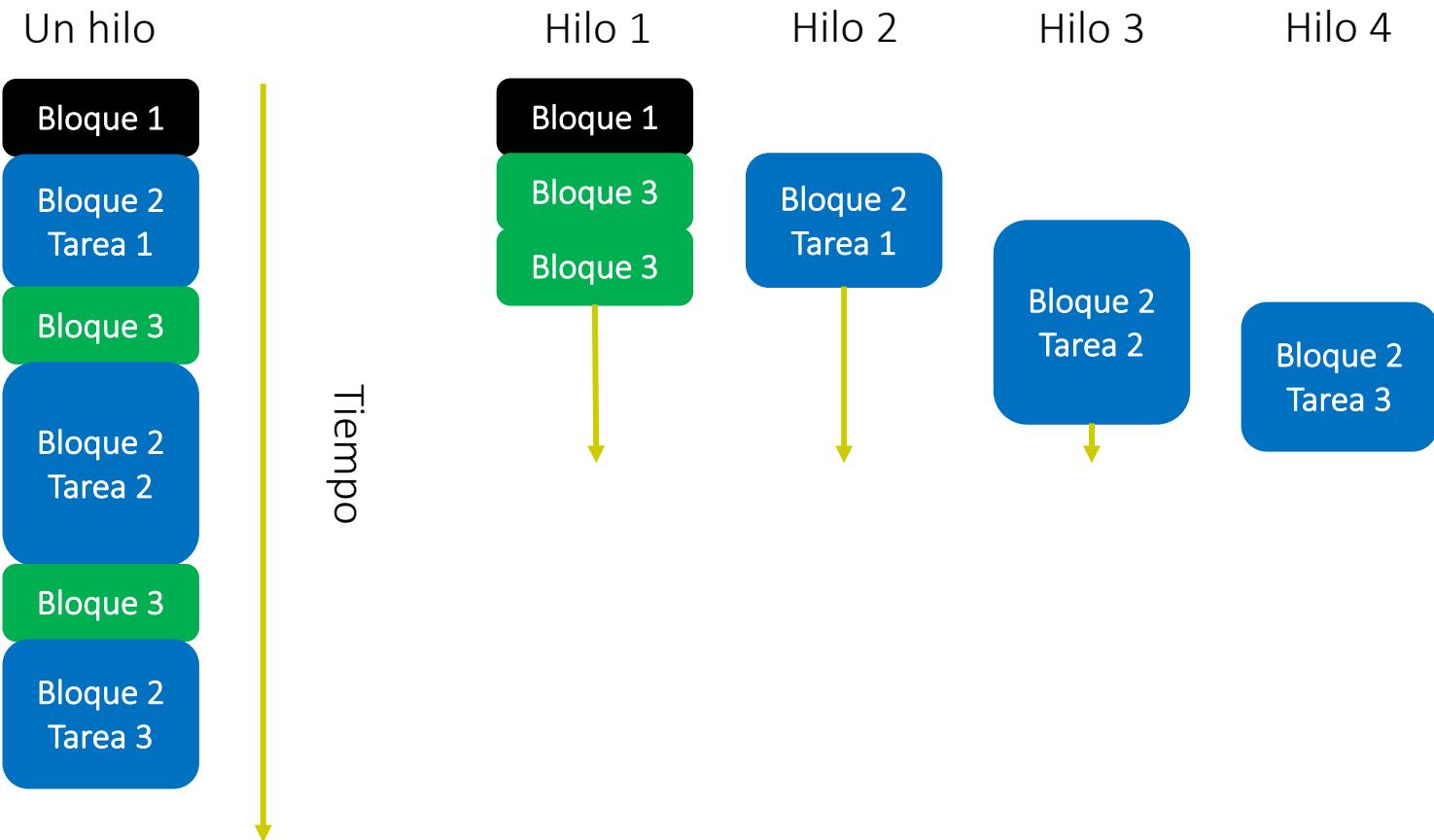
```
#pragma omp parallel
{
    #pragma omp single
    {
        struct item *p = list;                                // bloque 1
        while (p!=NULL){                                     // bloque 2
            #pragma omp task firstprivate(p)
            printf("id: %d ->%d\t", (*p).id, (*p).next);
            p = (*p).next;                                  // bloque 3
        }
    }
}
```

Si el programa se ejecutase con un solo hilo, se tendría lo siguiente:

```
#pragma omp parallel
{
    #pragma omp single
    {
        struct item *p = list;
        while (p!=NULL){           // bloque 1
            #pragma omp task firstprivate(p)
            printf("id: %d ->%d\t", (*p).id, (*p).next);
            p = (*p).next;          // bloque 2
        }
    }
}
```



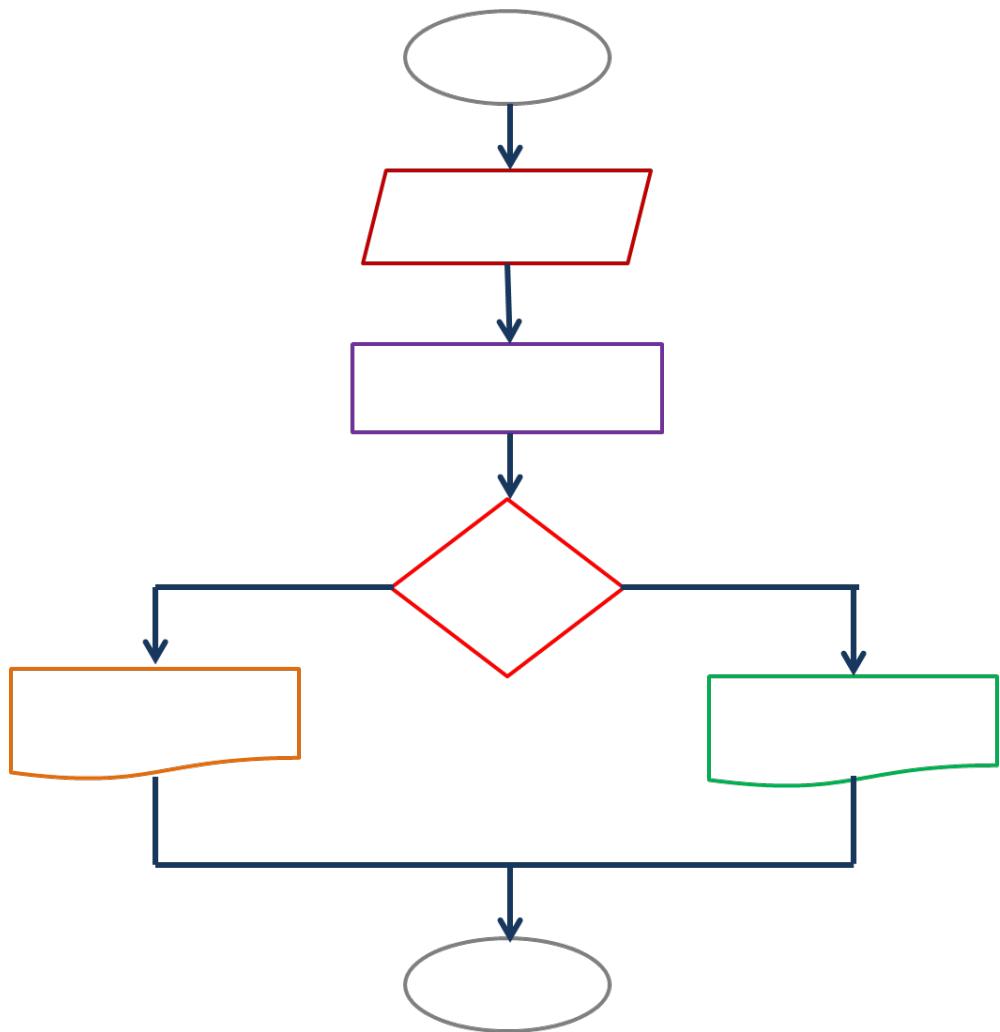
Sin embargo, si el programa se ejecutase con un varios hilos, se tendría lo siguiente:



Por lo tanto, lo que realiza el constructor de tarea es, en realidad, ejecutar el bloque asignado con el número de hilos que se tengan disponibles, brindando otra opción para hacer más eficiente la ejecución de un código utilizando procesamiento paralelo.

6.3.2 Rediseño de algoritmos.

6.3 Técnicas de desarrollo
de algoritmos.



Para diseñar un algoritmo paralelo se deben tener en cuenta cuatro etapas básicas:

- Partición
- Comunicación
- Agrupación
- Asignación

Partición

La solución de un problema implica el procesamiento (cómputo) y los datos (información). Cuando se partitiona un problema serial para parallelizar, se requiere tratar de dividir tanto los cómputos como los datos. Por tanto, esta descomposición se clasifica en:

- Descomposición de dominio: Se refiere a la partición de los datos para poder ser procesados de manera paralela.
- Descomposición funcional: Se refiere a la partición del procesamiento para poder obtener los datos deseados.

Comunicación

La comunicación de información o procesamiento se refiere a la manera en la que los procesadores pueden intercambiar información (memoria compartida) y los mensajes que se pueden enviar entre procesadores (bloqueos, sincronización, etc.). La comunicación debe tener en cuenta lo siguiente:

- Las tareas deben realizar el mismo número de operaciones de comunicación.
- La comunicación entre tareas debe ser tan pequeña como sea posible.
- Las operaciones de comunicación podrían ser concurrentes.
- Los cómputos de diferentes tareas podrían ser concurrentes.

Agrupación

La agrupación es un concepto de eficiencia, mediante el cual se analiza si es posible reducir la cantidad de datos a enviar, logrando con ello reducir el número de mensajes y el costo de la comunicación. Cuando se realiza una agrupación hay que validar:

- Si la agrupación redujo los costos de comunicación.
- Si las tareas resultantes tienen costos de cómputo y comunicación similares.
- Si es posible reducir más el número de tareas sin generar desbalances de cargas o reducir la extensibilidad.

Asignación

La asignación se refiere al procesador que va a realizar el cómputo. Esta asignación puede ser dinámica o estática.

En la asignación dinámica se utilizan las unidades de procesamiento en tiempo de ejecución, lo que requiere un balanceo de carga entre las tareas del programa y las del sistema operativo.

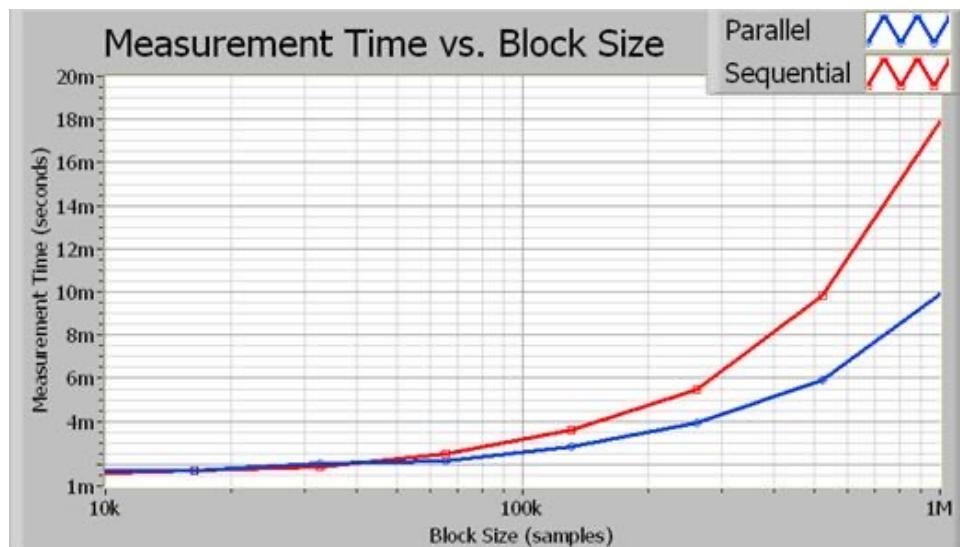
6.4 Análisis de desempeño de algoritmos paralelos.

Introducción a los algoritmos paralelos

208

6.4.1 Trabajo y profundidad.

6.4 Análisis de desempeño
de algoritmos paralelos.



Si se desea realizar un análisis específico de un equipo paralelo se debe tener en cuenta la arquitectura del procesador (lo cual no es general). El modelo de trabajo y profundidad se enfoca en el algoritmo per se, no en la arquitectura de la computadora.

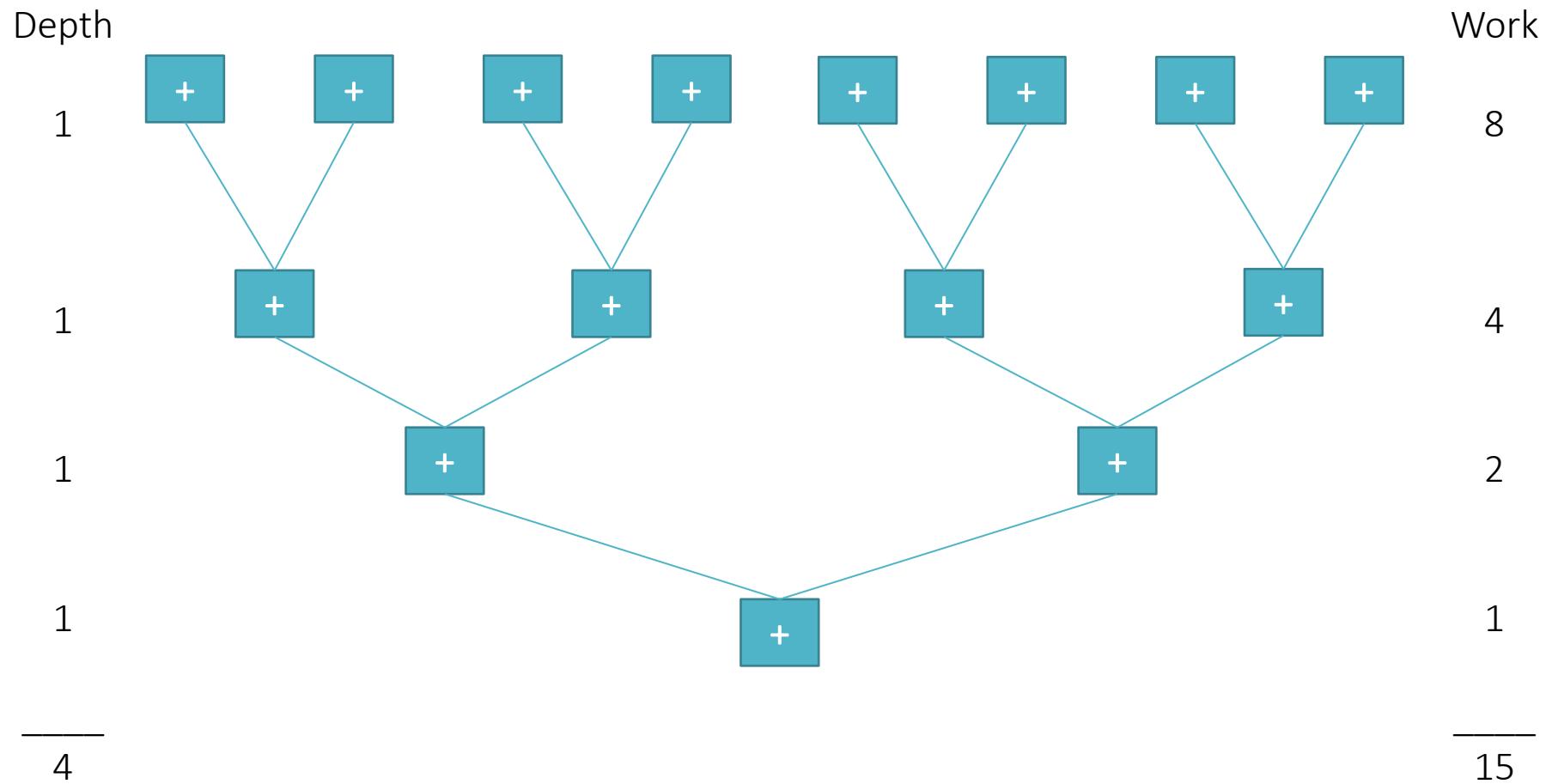
El modelo de trabajo y profundidad (work and depth) se enfoca en el número total de operaciones que un programa realiza (work), así como la cadena más larga de dependencias entre ellas (depth).

El trabajo W se refiere al número de operaciones que se realizan en el proceso. El trabajo permite conocer el tiempo que tarda en ejecutarse el algoritmo en un solo procesador (T_1).

La profundidad D se refiere a la cadena más larga de dependencias entre las operaciones realizadas en el proceso. La profundidad también se puede definir como la longitud de la ruta crítica, o el tiempo que tardaría el algoritmo en ejecutarse en una cantidad infinita de procesadores (T^∞).

Ejemplo

Se desean sumar todos los nodos de la siguiente estructura



Ejemplo

El trabajo W requerido para computar el resultado es de 15 operaciones (se realizan 15 sumas). La profundidad D computacional requerida es de 4 operaciones, siendo esta la longitud de la cadena más grande.

El trabajo es visto como una medida de el costo total del proceso, lo que representa la complejidad del algoritmo de forma serial. La profundidad, por otro lado, represente el mejor tiempo de ejecución en una máquina ideal con un número ilimitado de procesadores.

El trabajo y la profundidad también permiten calcular la razón $P=W/D$, la cual se conoce como paralelismo del algoritmo.

Dada la definición del paralelismo del algoritmo, en una máquina con recursos paralelos infinitos el tiempo de ejecución del algoritmo es del orden de la profundidad de éste, es decir, $T_\infty(n) = \Theta(D(n))$. Por otro lado, en una máquina con un solo procesador, el tiempo de ejecución es del orden del trabajo éste, es decir, $T_1(n) = \Theta(W(n))$.

Work & Depth vs Processor-based

Para realizar el análisis comparativo de los dos modelos se va a tomar como caso de estudio el algoritmo de ordenamiento por Quicksort:

procedimiento $\text{QUICKSORT}(S)$:

Si S contiene al menos un elemento

regresa S

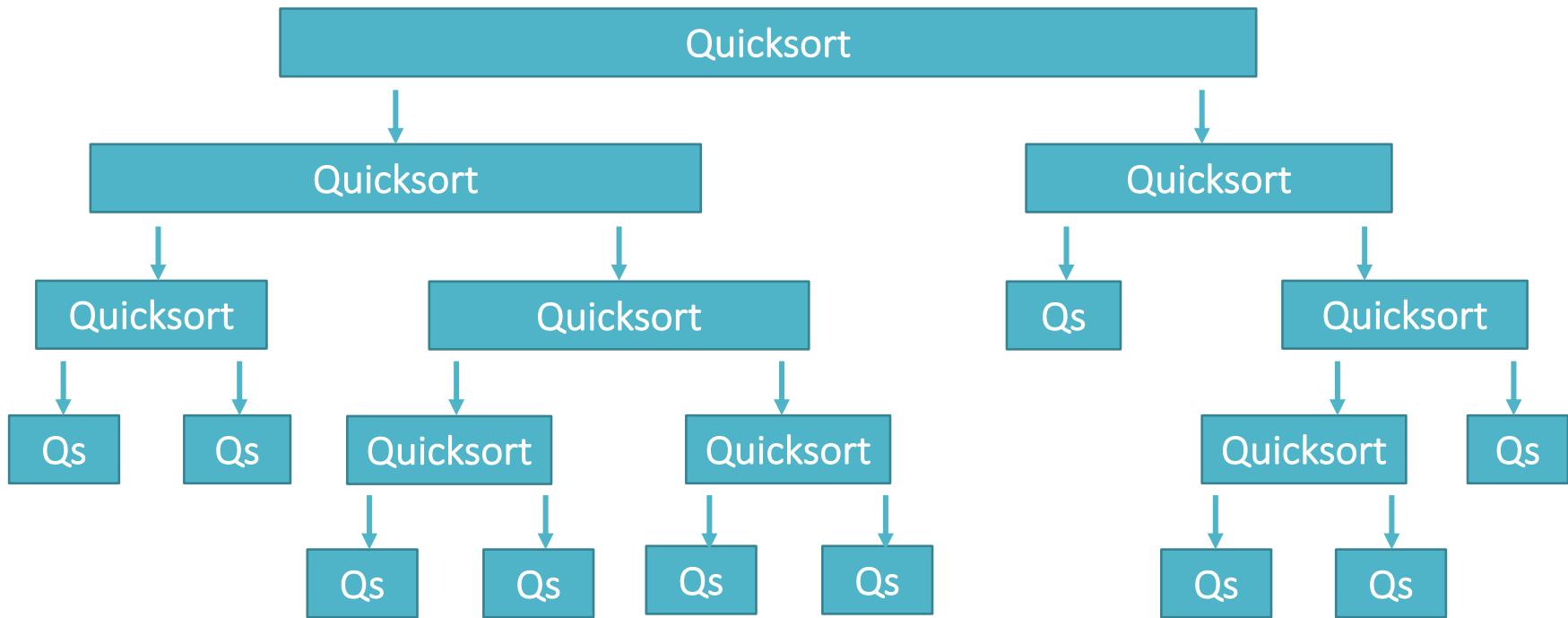
de lo contrario

Se elige un elemento α aleatorio de S

Sea S_1 , S_2 y S_3 una secuencia de elementos en S menores que, iguales a y mayores a respectivamente

regresa $\text{QUICKSORT}(S_1)$, seguido de S_2 , seguido de $\text{QUICKSORT}(S_3)$

Es posible paralelizar Quicksort haciendo que las dos llamadas recursivas se ejecuten al mismo tiempo.



Si se realiza el análisis del algoritmo paralelo de Quicksort utilizando el modelo work and depth, se puede observar que el trabajo requerido es el mismo tiempo requerido por el programa serial, multiplicado por un pequeño factor constante de tiempo, es decir, $O(n \log(n))$.

Por otro lado, la profundidad requerida está dada por el número de divisiones realizadas, donde el número máximo está dado por $O(\log(n))$.

Como se puede observar, el análisis del desempeño del algoritmo se realizó a alto nivel, sin necesidad de especificar el número de procesadores requeridos.

Si se realiza el análisis del algoritmo paralelo de Quicksort utilizando un modelo basado en procesadores con p procesadores, se requeriría aterrizar el algoritmo a un lenguaje de programación para obtener de manera dinámica los procesadores disponibles y particionar entre ellos el proceso, lo cual es complicado porque las particiones de Quicksort no son del mismo tamaño.

Además, se requeriría especificar un método de sincronización para unir todos los proceso y analizar los datos que deben ser compartidos y los datos que deben ser privados.

Por lo tanto, el análisis de un algoritmo paralelo es más práctico hacerlo en el modelo de trabajo y profundidad, para obtener un análisis muy real y sin involucrar detalles técnicos (lo cual es la base del análisis).

6.4.2 Ejemplos clásicos.

6.4 Análisis de desempeño de algoritmos paralelos.

• $T_p = \frac{n^3}{p} + \log pt_s + 2\frac{n^2}{\sqrt{p}}t_w$

| | |
|----|----|
| P1 | P1 |
| P2 | P2 |
| P3 | P3 |
| P4 | P4 |

A

*

| | |
|----|----|
| P1 | P2 |
| P3 | P4 |
| P1 | P2 |
| P3 | P4 |

B

=

| | |
|----|----|
| P1 | P2 |
| P3 | P4 |

C

Multiplicación de matrices

Las operaciones que se realizan para el cálculo del producto de dos matrices son similares a la operación realizada en la formación de un producto escalar de dos vectores. Podría ser útil pensar en el proceso de la formación del elemento C_{ij} (donde C es la matriz resultante) tomando el producto escalar del renglón i de la primera matriz por la columna j de la segunda matriz.

Para poder realizar la multiplicación de la matriz A con la matriz B se deben revisar las dimensiones de éstas.

$$A_{m \times n} \quad B_{n \times l} = C_{m \times l}$$


El elemento C_{ij} está dado por:

$$C_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

| | |
|--|--|
| | |
| | |
| | |
| | |

A

*

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |

B

=

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |

C

Ejemplo

```
void multiply(int a[RA][CA], int b[RB][CB], int c[RA][CB]) {  
    int i, j, k;  
    for (i=0; i<RA; i++)  
        for (j=0; j<CB; j++)  
            for (k=0; k<CA; k++)  
                c[i][j] += a[i][k] * b[k][j];  
}
```

6 Introducción a los algoritmos paralelos

Objetivo: Clasificar los elementos a considerar en el diseño y análisis de algoritmos paralelos versus algoritmos seriales para su programación.

6.1 Niveles de paralelismo. Granularidad.

6.2 Algoritmos con memoria compartida.

- 6.2.1 Carrera de datos.

- 6.2.2 Inconsistencia de datos.

- 6.2.3 Modelo PRAM.

6.3 Técnicas de desarrollo de algoritmos.

- 6.3.1 Rediseño de estructuras de datos.

- 6.3.2 Rediseño de algoritmos.

6.4 Análisis de desempeño de algoritmos paralelos.

- 6.4.1 Trabajo y profundidad.

- 6.4.2 Ejemplos clásicos.