

Arquitectura de Ordenadores

Práctica 4.

Explotar el potencial de las arquitecturas modernas.

ESCUELA POLITÉCNICA SUPERIOR. UAM.



Contenidos

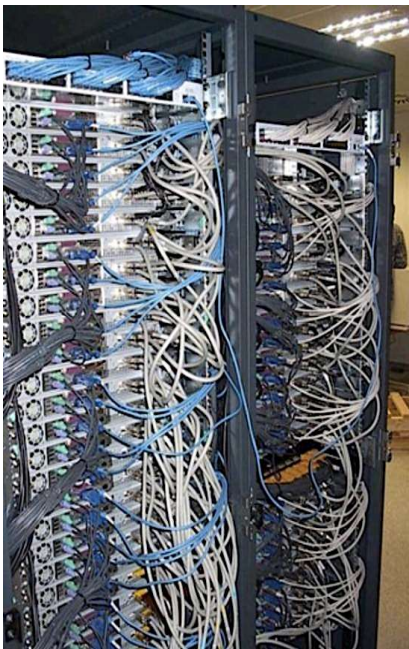
- **Arquitectura de computadores (Tema 5)**
 - Multicomputador, Multiprocesador, Multicore
 - Hyperthreading
- **Programación paralela en sistemas de memoria compartida (multiprocesador *multicore*)**
 - POSIX threads: pthreads
 - OpenMP
- **Práctica 4**
 - Entorno
 - Ejercicios

Arquitecturas para Computación paralela: (HPC)

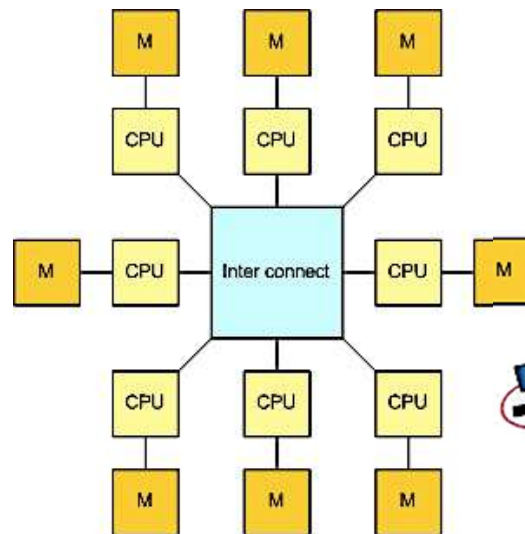
- **Hoy en día las máquinas con las que trabajamos disponen de muchas unidades de proceso interconectadas**
- **Diversos niveles de interconexión**
 - Multicomputador
 - Multiprocesador
 - Multicore
- **Mediante computación/programación paralela podemos sacar partido de estos sistemas**

Multicomputador

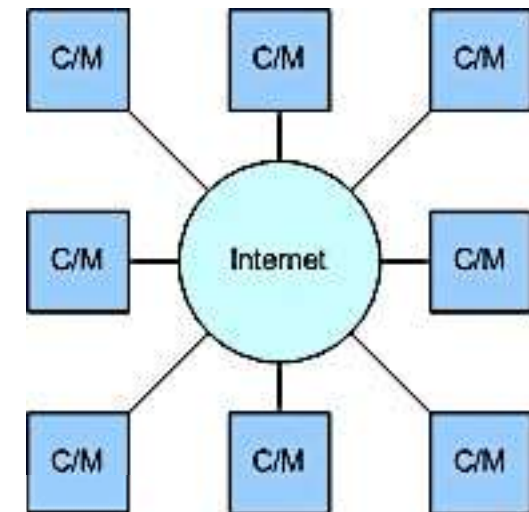
- Diversos ordenadores conectados entre sí.
- Comunicación basada en paso de mensajes.
- Memoria distribuida.



- **Clúster**

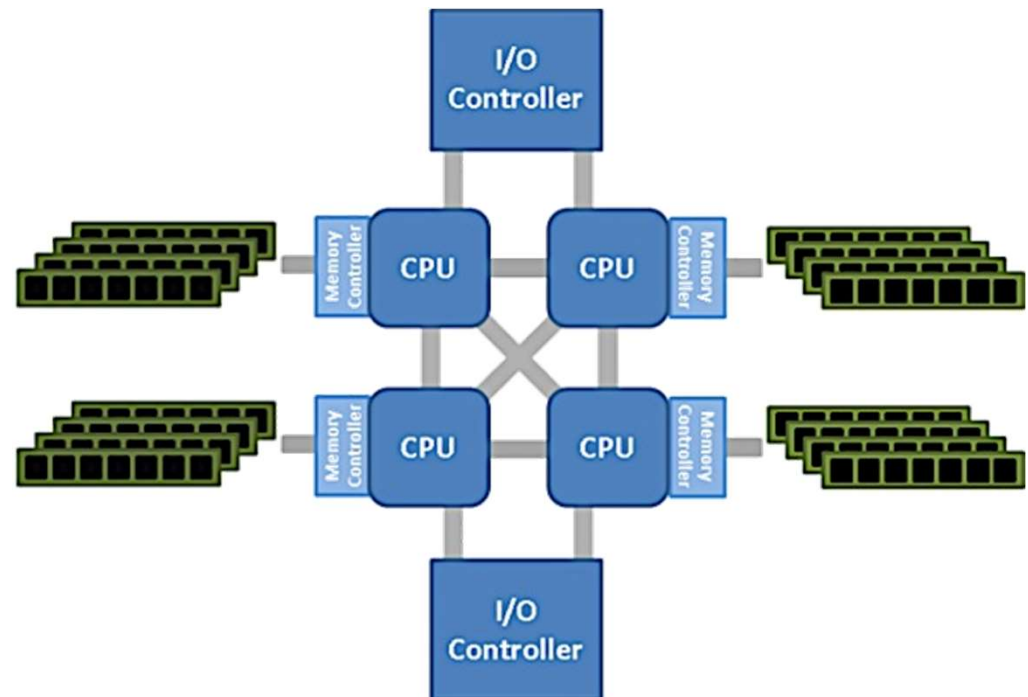


- **Grid**



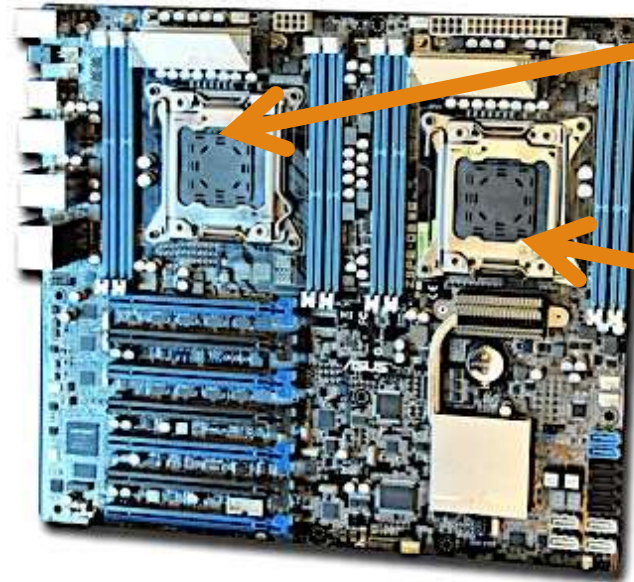
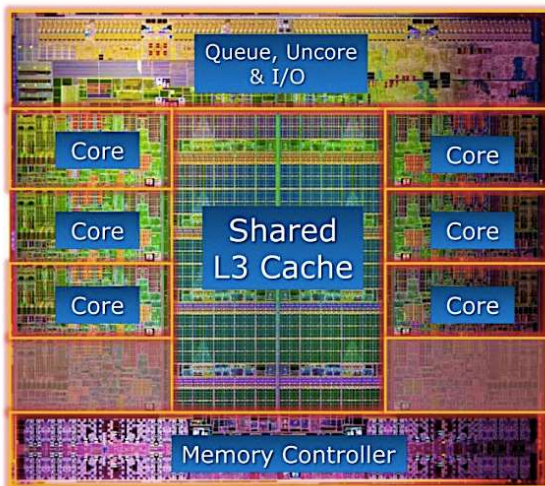
Multiprocesador

- Ordenador con más de una unidad física de proceso (procesador/nodo) conectadas a una memoria compartida
- Dos tipos
 - **UMA**
Uniform Memory Access
 - **NUMA**
Non-Uniform Memory Access
- Paralelismo
 - Procesos
 - Hilos (*Threads*)



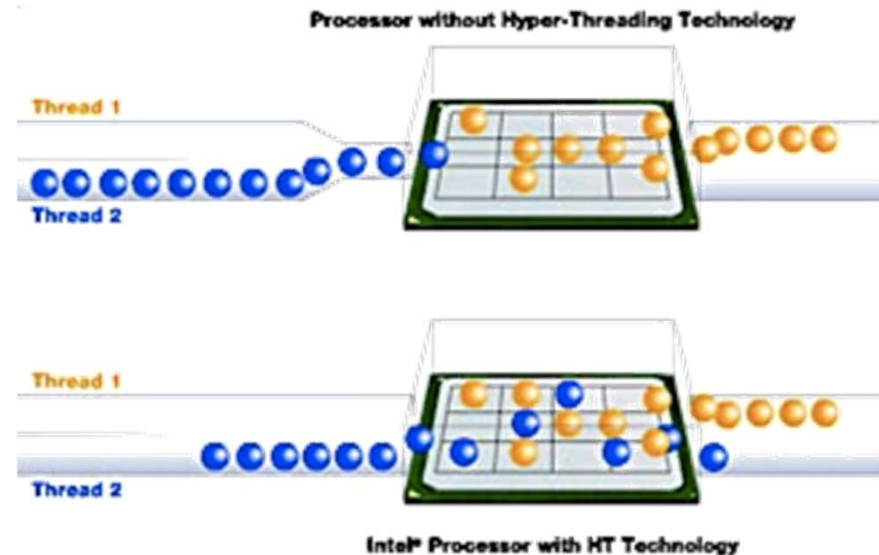
Sistemas de memoria compartida actuales

- Multiprocesador-multicore (MP-MC): Un procesador *multicore* combina en un mismo encapsulado dos o más microprocesadores independientes (demonina **core**)
- Memoria compartida: **Acceso NUMA:**



Cores Lógicos versus cores físicos: Hyperthreading

- **Cada procesador maneja dos *threads***
 - Cuando el que está ejecución se bloquea, entra el otro
 - Estructura HW para hacer un cambio de contexto del procesador (registros, ...)
- **Número de procesadores x 2**
 - ¡No es real!
 - Útil en sistemas de sobremesa
 - Muchos bloqueos
 - No siempre útil en HPC
 - Nº bloqueos mínimo



Modelo de programación de sistemas con memoria compartida

■ Características

- Varios procesos o threads ejecutándose en un espacio de direcciones común

■ Alternativas

- usar un lenguaje paralelo nuevo, o modificar la sintaxis de uno secuencial (HPF, UPC... / Occam, Fortran M...).
- trabajar directamente con procesos/threads: **Pthreads (POSIX)**
- usar un lenguaje secuencial junto con **directivas al compilador** con **rutinas de librería**. para especificar el paralelismo. **OPENMP**

- ¿Cómo explotar el paralelismo?

- A nivel de programas
- A nivel de subrutinas
- **A nivel de bucles**
- **A nivel de sentencias**



PARALELISMO
GRANO GRUESO



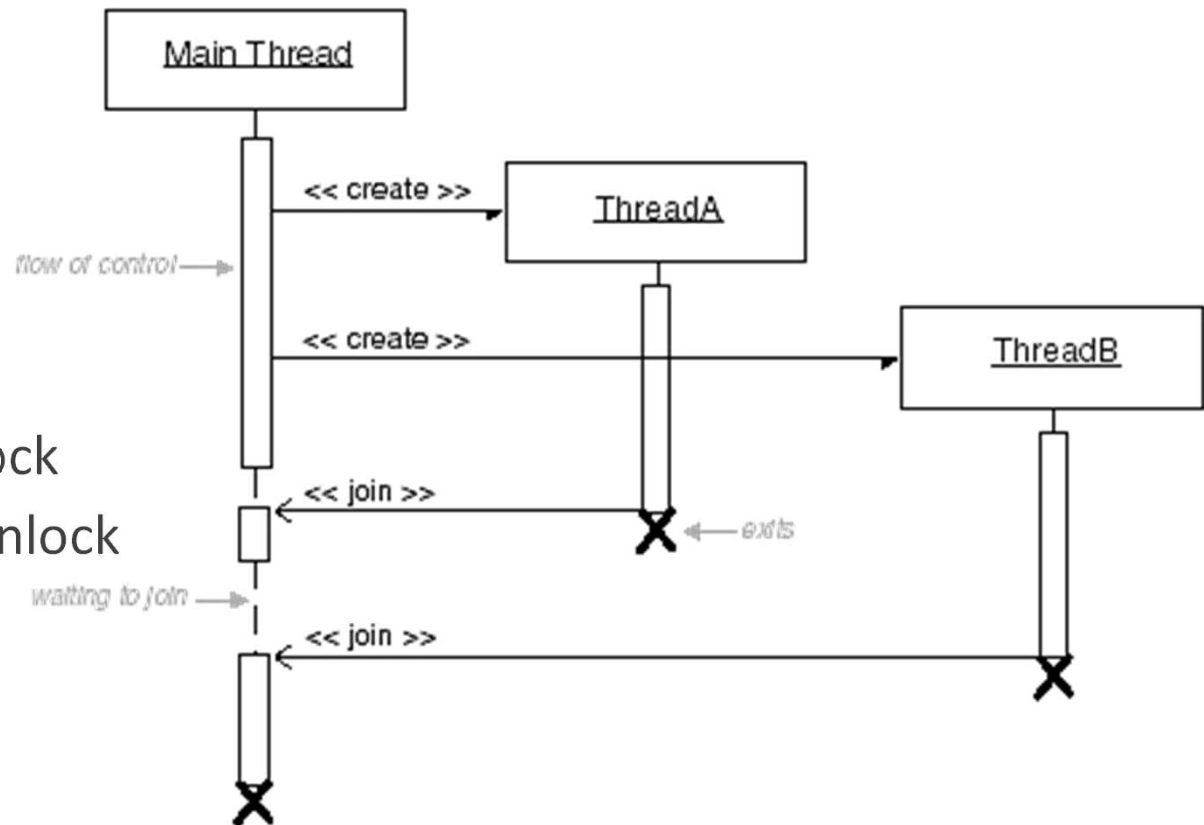
PARALELISMO
GRANO FINO

Modelo de programación de sistemas con memoria compartida: PThreads

- Programación basada en hilos comunicados por memoria compartida

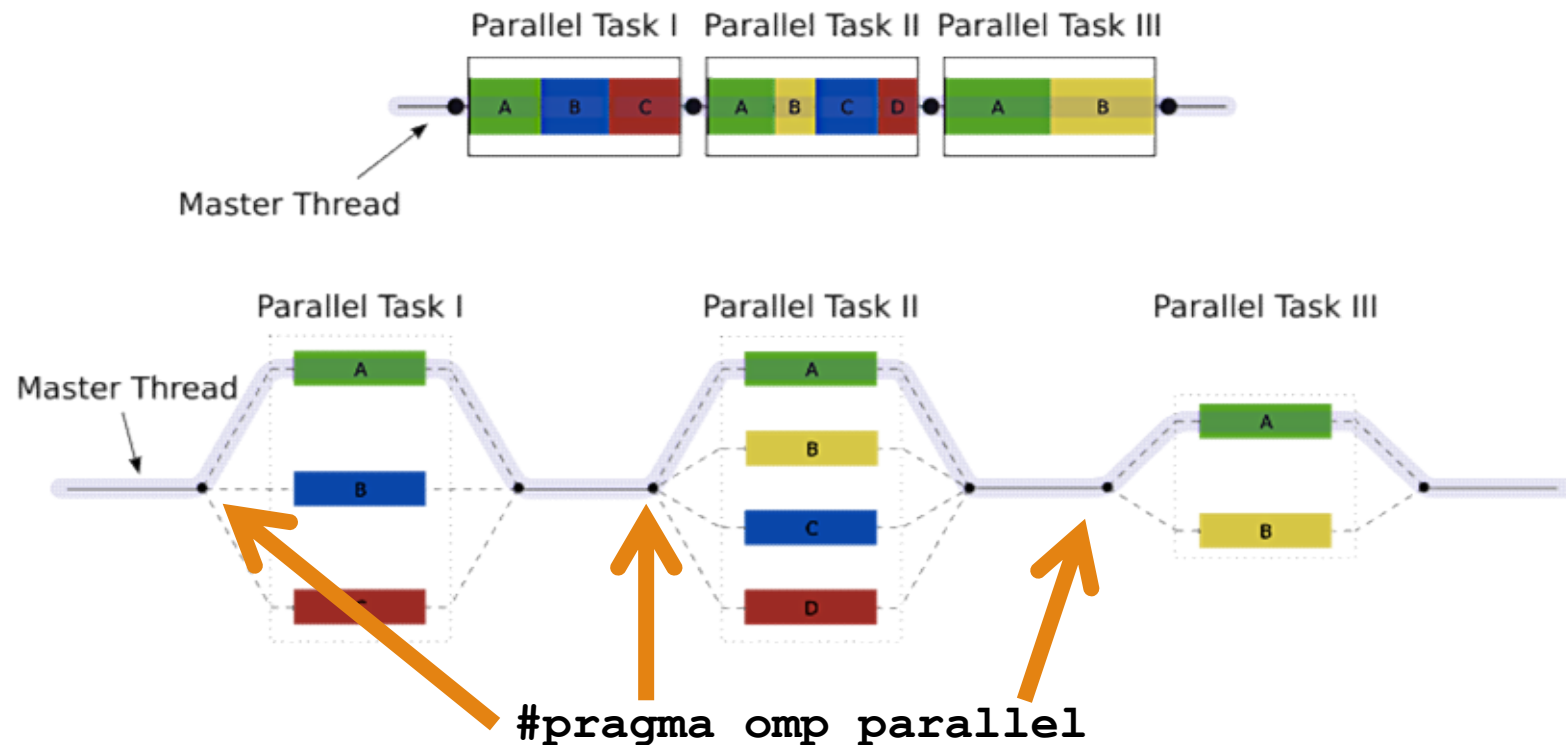
- **POSIX threads**

- pthread_create
- pthread_join
- pthread_mutex_lock
- pthread_mutex_unlock
- ...



Modelo de programación de sistemas con memoria compartida: OpenMP

- Ofrece un API basado en *pragmas* para simplificar el manejo de hilos en sistemas MP-MC



OpenMP

El estándar OpenMP se usa en la programación de computadores paralelo con **un espacio de direcciones compartido**.

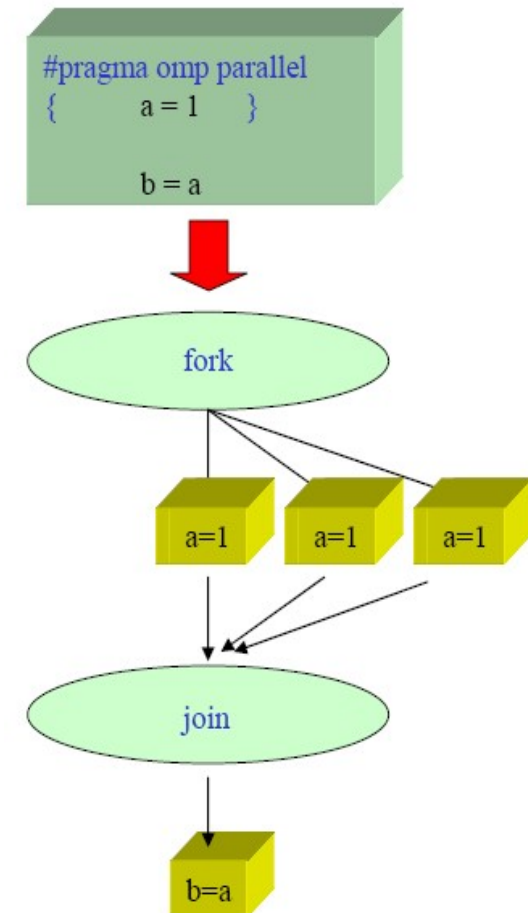
- OpenMP es una API (Application Program Interface) que permite definir explícitamente paralelismo multi-thread en sistemas de memoria compartida. (Compilador OpenMP para C/C++ y Fortran)

OpenMP esta dividido en tres componentes principales:

- Directivas de compilación
- Librería de rutinas (Runtime Library)
- Variables de entorno
- Las aplicaciones desarrolladas con OpenMP son portables a todas las arquitecturas de memoria compartida que cuenten con un compilador de OpenMP.
- No realiza una paralelización automática.

Modelo de Ejecución OpenMP

- OpenMP se basa en el modelo fork/join
 - Un programa OpenMP empieza con un solo thread(Master Thread).
 - Se lanzan varios threads (Team) en un región de código paralelo.
 - Al salir de la región paralela los threads lanzados retornan (sincronización).
- Un “Team” tiene una cantidad fija de threads ejecutando código replicado en la región paralela.
 - Construcciones para compartir trabajo especifican que threads ejecutan cada parte de trabajo
 - Una barrera de sincronización para todos los threads del team finaliza la región paralela.
- El código después de una región paralela se ejecuta secuencialmente por el thread Master.



OpenMP: Componentes

Directivas

- Regiones Paralelas
- Work sharing
- Sincronización
- Clausulas
 - private
 - firstprivate
 - lastprivate
 - shared
 - reduction

Variables de Entorno

- N° threads
- Tipo scheduling
- Ajuste dinámico threads
- Paralelismo anidado

Runtime API

- N° threads
- ID thread
- Tipo scheduling
- Ajuste dinámico threads
- Paralelismo anidado

OpenMP: Formato de las directivas

```
#pragma omp nombre_de_directiva [clause, ...]
```

- #pragma omp

Requerido por todas las directivas OpenMP para C/C++

- nombre_de_directiva

Un nombre valido de directiva. Debe aparecer después del pragma y antes de cualquier clausula.

- [clause , ...]

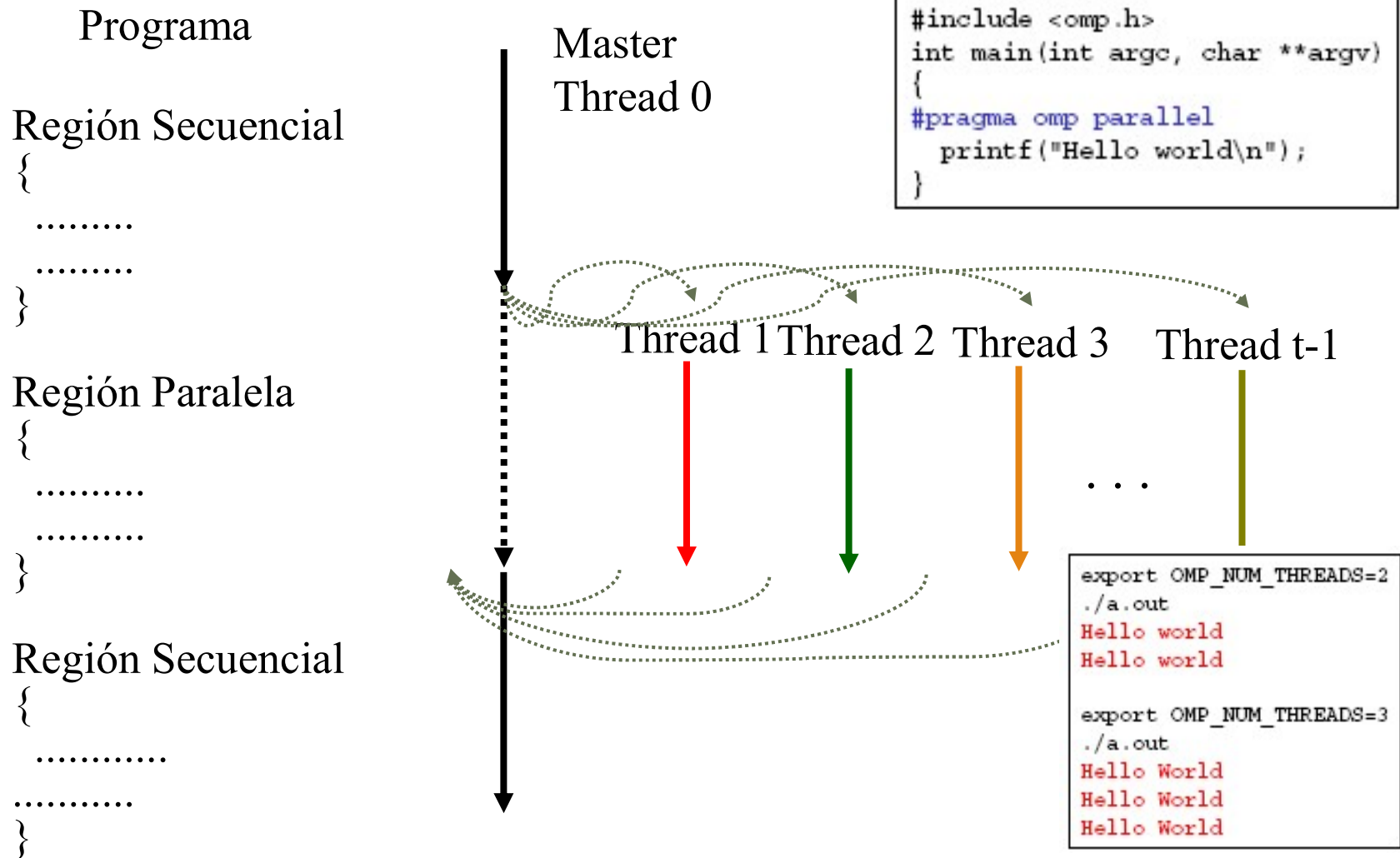
Opcionales. Las clausulas pueden ir en cualquier orden y repetirse cuando sea necesario a menos que haya alguna restricción.

```
#pragma omp parallel default(shared) private(beta, pi)
```

Directiva `parallel`

- Su aparición en el código significa la delimitación de una región paralela
- Al inicio de la región paralela, se lanzan tantos hilos como se haya indicado
- Al final de la región paralela, se realiza una sincronización de todos los hilos
 - Hasta que todos hayan acabado no se continúa con el programa
- Este lanzamiento/sincronización de hilos implica un coste en términos de tiempo de ejecución que no se puede obviar

Regiones paralelas y secuenciales



OpenMP:Región Paralela

Una región paralela es un bloque de código que será ejecutado por múltiples threads

- Es la construcción paralela fundamental de OpenMP

```
#pragma omp parallel [clause ...]  
    if (scalar_expression)  
    private (list)  
    shared (list)  
    default (shared | none)  
    firstprivate (list)  
    reduction (operator: list)  
    copyin (list)  
    num_threads (integer-expression)  
  
structured_block
```

OpenMP: programa de ejemplo

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int tid, nthr, nproc;
7      int arg;
8
9      nproc = omp_get_num_procs();
10     printf("Hay %d cores disponibles\n", nproc);
11
12     arg = atoi( argv[1] );
13     omp_set_num_threads(arg);
14     nthr = omp_get_max_threads();
15     printf("Me han pedido que lance %d hilos\n", nthr);
16
17     #pragma omp parallel private(tid)
18     {
19         tid = omp_get_thread_num();
20         nthr = omp_get_num_threads();
21         printf("Hola, soy el hilo %d de %d\n", tid, nthr);
22     }
23 }
```

región paralela



Conceptos Básicos

- Una región paralela es un bloque de código ejecutado por todos los threads simultáneamente
 - El thread maestro tiene el ID 0
 - El ajuste (dinámico) de threads (si esta activado) se realiza antes de entrar a la región paralela
 - Podemos anidar regiones paralelas, pero depende de la implementación
 - Podemos usar una clausula if para hacer que una región paralela se ejecute de forma secuencial
- Las construcciones “work sharing” permiten definir el reparto del trabajo a realizar en la región paralela, entre los threads disponibles
 - No crea nuevos threads

Otro ejemplo: suma de vectores

Paralelización de la suma de dos vectores:

```
#define NUMTHREADS 4
void Suma(double* a, double* b, double* c, int size)
{
    omp_set_num_threads(NUMTHREADS);
    #pragma omp parallel private (tid, i)
    {
        tid = omp_get_tread_num();
        for (int i = tid; i < size; i+=NUMTHREADS)
        {
            c[i] = a[i] + b[i];
        }
    }
}
```

Para compilar con GCC :

```
gcc -fopenmp ejemplosuma.c -o ejemplosuma -lgomp
```

Qué resuelve OpenMP

- Creación de los equipos de hilos
- Sincronización al terminar la región paralela
- Librería de funciones para acceder a/modificar datos relacionados con el equipo de trabajo
 - ¿Quién soy?: `omp_get_thread_num()`
 - ¿Con quién voy?: `omp_get_num_threads()`
 - Cambiar tamaño del equipo: `omp_set_num_threads()`
 - ...
- Generación de copias locales de las variables definidas como privadas
- Asignación de afinidad a cada *thread* del equipo

OpenMP: Runtime API

- El estandar OpenMP define un API de llamadas para realizar una serie de funciones:
 - Obtener/Establecer el número de threads a usar.
 - Obtener el identificador de thread.
 - Rutinas de bloqueo de propósito general (semaphores)
 - Establecer funciones de entorno de ejecución: paralelismo anidado, ajuste dinámico de threads, etc.
- *void omp_set_num_threads(int num_threads)*: Fija el número de threads que serán utilizados en la próxima región paralela. Sólo puede invocarse desde una sección secuencial de código
- *int omp_get_num_threads(void)*: Si se invoca desde una sección secuencial de código, o de una región paralela anidada que se ha serializado, retorna 1
- *int omp_get_thread_num(void)*: Devuelve el número del identificador del thread en el equipo desde el que se invoca. Este número estará entre 0 y omp_get_num_threads-1

Ejecución con varios Threads

Establecer numero de threads con variable de entorno

- Se hace a través de la variable de entorno OMP_NUM_THREADS
- Variable de entorno en Linux (Bash):

```
export OMP_NUM_THREADS=3  
ejemplosuma
```

Dentro del programa con código (función de librería)

```
#include <omp.h>  
int threads = 3;  
void Suma(double* a, double* b, double* c, int size)  
{  
    omp_set_num_threads(threads);  
    #pragma omp parallel for  
    for (int i = 0; i < size; ++i)  
    {  
        c[i] = a[i] + b[i];  
    }  
}
```

Consideraciones al programar OpenMP

- **Definir correctamente la accesibilidad de las variables utilizadas**
 - public/private
 - Ejercicio 1 y 4
- **Gestionar la recogida de resultados del equipo de hilos**
 - Reduction
 - Ejercicio 2 y 5
- **Colocación más adecuada del pragma `parallel`**
 - Ejercicio 3

OpenMP: Reparto de tareas

Construcciones de trabajo compartido

Distribuyen la ejecución de las sentencias asociadas entre los threads definidos. No lanzan nuevos threads.

OpenMP define las siguientes construcciones de trabajo compartido:

- Construcción `for`
 - Define una región donde las iteraciones del bucle deben ejecutarse entre los threads que lo encuentren

`#pragma omp for [clausulas ...]`

lazo for

- Construcción `sections`

Directiva `for`

- Puede usarse dentro de una región paralela ó combinado con `parallel` antes de un bucle (que pasará a ser la región paralela)
- Reparte de forma automática el trabajo entre los *threads* del equipo
 - Declara automáticamente la variable índice como privada

```
...  
for (i=0; i<100; i++)  
{  
    ...  
}  
...
```



Directiva for

```
#pragma omp parallel
{
    ...
    #pragma omp for
    for (i=0; i<100; i++)
    {
        ...
    }
    ...
}
```

```
...
for (i=0; i<25; i++)
{
    ...
}
...
```

Thread 0

```
...
for (i=50; i<75; i++)
{
    ...
}
...
```

Thread 1

```
...
for (i=25; i<50; i++)
{
    ...
}
...
```

Thread 2

```
...
for (i=75; i<100; i++)
{
    ...
}
...
```

Thread 3



Construcción “Work-Sharing”

Directiva *for*

La directiva *for* especifica que las iteraciones del bucle deben ser ejecutadas en paralelo por el equipo de threads

- Se asume que la región paralela ya ha sido iniciada, de otro modo se ejecutarán en serie

```
#pragma omp for [clause ...]  
    schedule (type [,chunk])  
    ordered  
    private (list)  
    firstprivate (list)  
    lastprivate (list)  
    shared (list)  
    reduction (operator: list)  
    collapse (n)  
    nowait
```

for_loop

Directiva parallel for

Las directivas `parallel` y `for` pueden juntarse en `#pragma omp parallel for`

cuando la región paralela contiene únicamente un bucle.

```
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];
    }
}
```


Clausulas y modificadores de las directivas

Las directivas suelen ir acompañadas de cláusulas o modificadores

```
#pragma omp directive_name [parameter list]
```

- ```
#pragma omp parallel private(iam, nthreads)
```

Ejemplo :

```
#pragma omp parallel for private(i) shared(y,x,alfa,n)
for (i=0; i<n; i++){
 x[i]= alfa * y[i];
}
```

# Ámbito de variables

---

El *thread* máster tiene como contexto el conjunto de variables del programa, y existe a lo largo de toda la ejecución del programa.

Al crearse nuevos *threads*, cada uno incluye su propio contexto, con su propia pila, utilizada para las rutinas invocadas por el *thread*.

Cómo se comparten las variables es el punto clave en un sistema paralelo de memoria compartida, por lo que es necesario controlar correctamente el **ámbito** de cada variable.

Las variables **globales** son compartidas por todos los *threads*. Sin embargo, algunas variables deberán ser propias de cada *thread*, **privadas**.

# Cláusulas de ámbito

---

- El **ámbito** de validez de cada variable, se indica con una serie de **cláusulas** a la directiva `parallel`

## Cláusulas

- `shared, private, firstprivate (var)`  
`default (shared/none)`
- `reduction (op:var)`  
`copyin (var)`
- `if (expresión)`
- `num_threads (expresión)`

Las variables **privadas no están inicializadas** al comienzo ,ni dejan rastro al final.

- Si se necesita hay que que declararlas `firstprivate` para poder pasar un valor a estas variables
- Es privada al thread pero se inicializa con el valor de la variable del mismo nombre en el thread master.

## Region Paralela: Cláusulas de ámbito

---

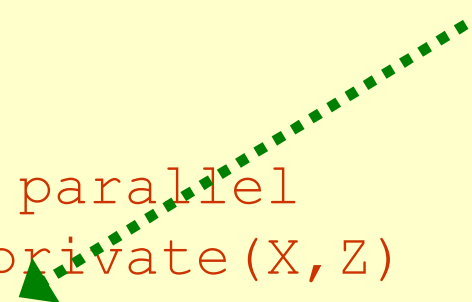
Ejemplo:

```
X = 2;
Y = 1;

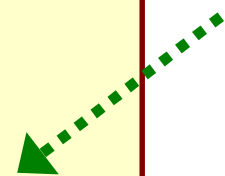
#pragma omp parallel
 shared(Y) private(X,Z)
 {
 Z = X * X + 3;
 X = Y * 3 + Z;
 }

printf("X = %d \n", X);
```

X no está  
inicializada!

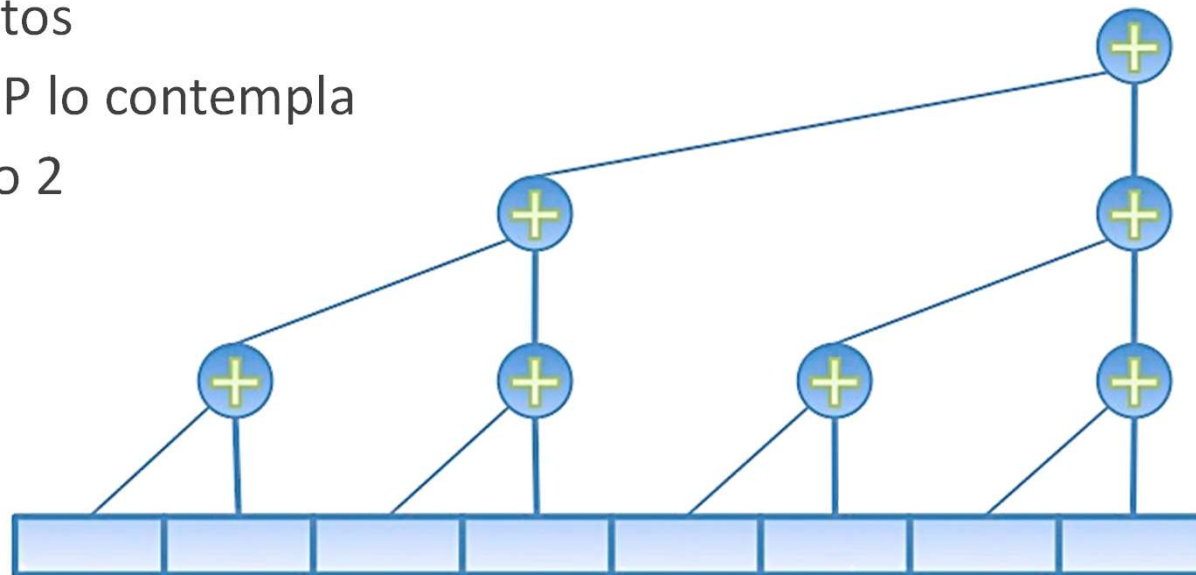


X no mantiene  
el nuevo valor



# Reducción

- Problema típico en programación paralela
- Cada hilo genera un resultado parcial, que se debe recoger y computar para obtener un resultado final
  - Ej: suma de  $N$  elementos con  $T$  hilos, cada hilo ha sumado  $N/T$  elementos
  - OpenMP lo contempla
  - Ejercicio 2



# Compilar programas OpenMP

---

- **Muchos compiladores soportan el uso de OpenMP**
- **Vamos a utilizar el soporte de gcc para openmp**
  - Flags al compilar:
    - -lgomp : librería con la funciones
    - -fopenmp : soporte de los pragmas
- **Está incluido en el fichero `Makefile` de partida**

# Práctica 4: entorno de de trabajo

---

- A lo largo de las prácticas 3 y 4 vamos a trabajar utilizando *software* para poner en práctica los conceptos *hardware* de teoría





# Práctica 4: material de partida

---

- **arqo4.c/arqo4.h**: librería de manejo de matrices/vectores de números en coma flotante
- **omp1.c/omp2.c**: ejemplos de programación utilizando OpenMP
- **pescalar\_serie.c**: versión serie de un programa que realiza el producto escalar de dos vectores
- **pescalar\_par1.c/pescalar\_par2.c**: versiones paralelas del producto escalar
- **Makefile**: fichero utilizado para la compilación de los ejemplos aportados.
- **pi\_serie.c** y **pi\_par1.c** a **pi\_par7.c**: versiones serie y paralelo de cálculo de pi por integración numérica.

# Ejercicio 0

---

- **Identificar de qué tipo es el PC del laboratorio**
  - ¿Multiprocesador?
    - #procesadores
  - ¿Multicore?
    - #cores
  - ¿Hyperthreading?

```
cat /proc/cpuinfo
```

# Ejercicios 1 y 2

---

- **Ejercicio1: Ejecutar y comprender ejemplos básicos de OpenMP**
  - Lanzamiento de un equipo de hilos
  - Utilización de las funciones básicas de OpenMP
  - Declaración de variables privadas/públicas
  - **Contestar a las preguntas de la memoria**
- **Ejercicio 2: Producto escalar de dos vectores**
  - Dos versiones paralelas
  - ¿Funcionan correctamente?
  - Análisis del rendimiento al variar parámetros
    - Tamaño del vector
    - Número de hilos
  - Contestar a las preguntas de la memoria

# Ejercicio 2

---

- **Producto escalar de dos vectores**
  - Dos versiones paralelas
  - ¿Funcionan correctamente?
  - Análisis del rendimiento al variar parámetros
    - Tamaño del vector
    - Número de hilos
  - Contestar a las preguntas de la memoria

# Ejercicio 3 a 5

---

- **Ejercicio 3: Multiplicación de matrices**

- ¿Qué bucle es mejor paralelizar?
- Análisis de rendimiento
- Contestar preguntas de la memoria

- **Ejercicios 4 y 5 : False Sharing**

- ¿ Que es el falso compartir?
- Análisis de rendimiento
- Contestar preguntas de la memoria

# Práctica 4: entrega

---

## Entregas:

- **Grupos Miércoles** – hasta el martes 11 de Diciembre de 2018 a las 23:59.
- **Grupos Jueves** – hasta el miércoles 12 de Diciembre de 2018 a las 23:59.

## EXAMEN P3 y P4:

- **Miércoles 12 de diciembre**
- **Jueves 13 de diciembre**

# Encuestas en SIGMA

---

- Profesor de prácticas
  - Profesor de teoría
  - Asignatura
- 
- SIGMA
    - Encuestas Web SIGMA
      - Rellenar encuestas

