

TUTORIAL OPENMP

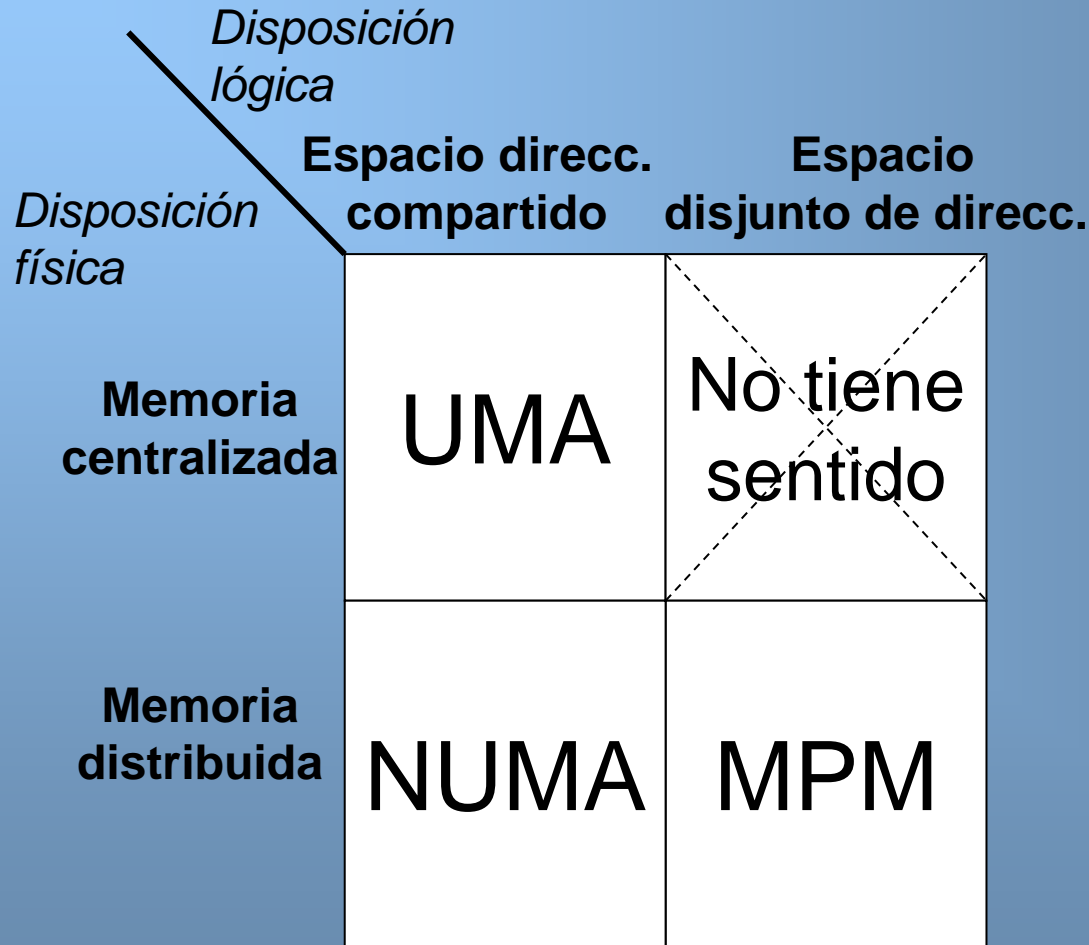
ARQUITECTURA DE SISTEMAS
DISTRIBUIDOS

www.atc.us.es

**Dpto. de Arquitectura y Tecnología de
Computadores. Universidad de Sevilla**

. Taxonomía y modelos de procesamiento paralelo

- Según Organización de la memoria principal

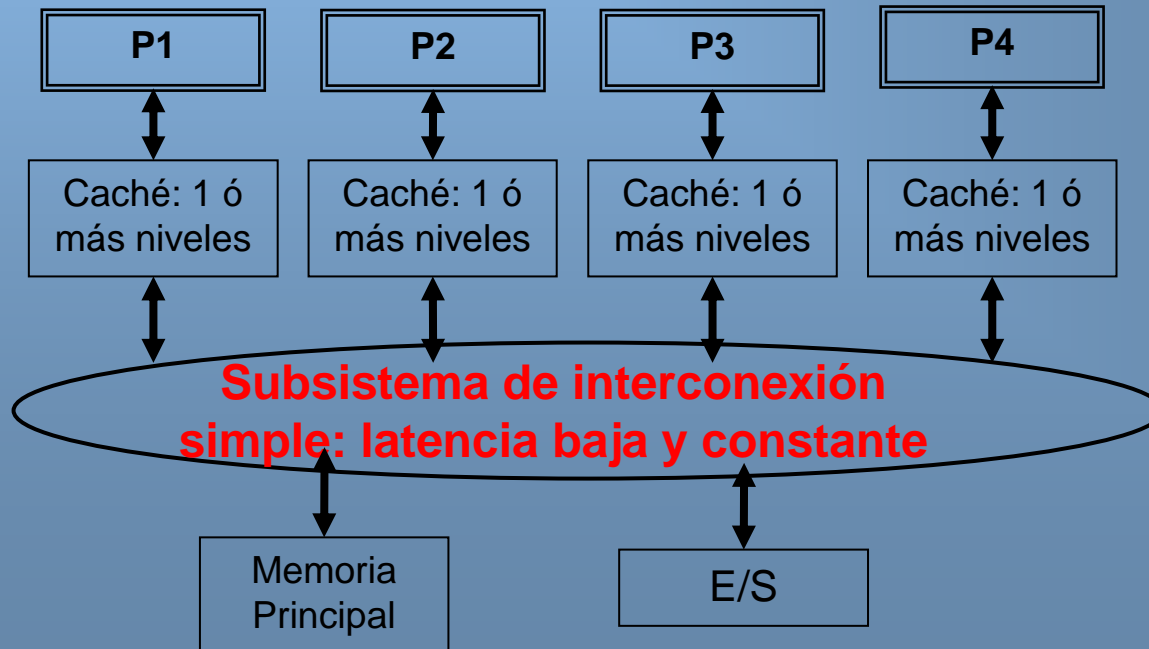


OPENMP Igual software,
distinto hardware.
Threads en Paralelo

Procesos
en Paralelo

UMA (Uniform Memory Access)

- Tiempo de acceso uniforme para toda dirección
- Tb .llamados SMP's (Multiprocesadores simétricos)
- Ej: casi todos los multicore actuales
- Cuello de botella: acceso a memoria principal :
 - Grandes cachés en cada procesador
 - AB solicitado por cada procesador crece cada vez más
 - Número de procesadores no puede ser alto. Hoy $N \leq 32$.



Coherencia entre cachés

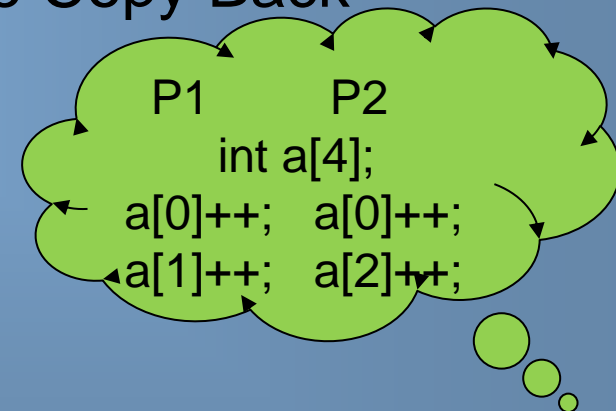
- Comunicación **implícita** (variables compartidas)
- Una misma línea replicada en varios procesadores

⇒ Hard adicional para solucionarlo.

- En UMA: Protocolos de husmeo (**snooping**):

- Protocolo ISX (y otros): similar al de Copy Back

- Línea Inválida (Invalid)
- Línea Compartida (Shared)
- Línea eXclusiva (eXclusive)



- CUIDADO: falsas comparticiones (**false sharing**).

Herramientas de programación

- Muchos intentos (y siguen...)
- Lenguaje de programación paralelo:
 - No es factible: Difícil de imponer
- Espacio de direcciones compartido
 - **OpenMP**: estándar (varios fabricantes) de directivas del precompilador.
- Espacio de direcciones disjuntos: clusters de computación.
 - Librería Message Passing Interface (**MPI**): especificación para librerías de paso de mensajes

OpenMP

- Modelo de programación paralela
 - Directivas del precompilador
 - Tb. Pequeña API
- Paralelismo de memoria compartida
- Fácil para bucles (paralelizables directa o indirectamente)
- Extensiones a lenguajes de programación existentes (Fortran, C, C++)

Sintaxis de OpenMP

- Pragma en C y C++:

```
#pragma omp construct [clause [clause]...]
```

- En Fortran, directivas :

```
C$OMP construct [clause [clause]...]
```

```
!$OMP construct [clause [clause]...]
```

```
*$OMP construct [clause [clause]...]
```

- Un programa puede ser compilado por compiladores que no soportan OpenMP.
 - ATENCIÓN: NO olvidar “Compatibilidad OpenMP” en el compilador

Programa sencillo

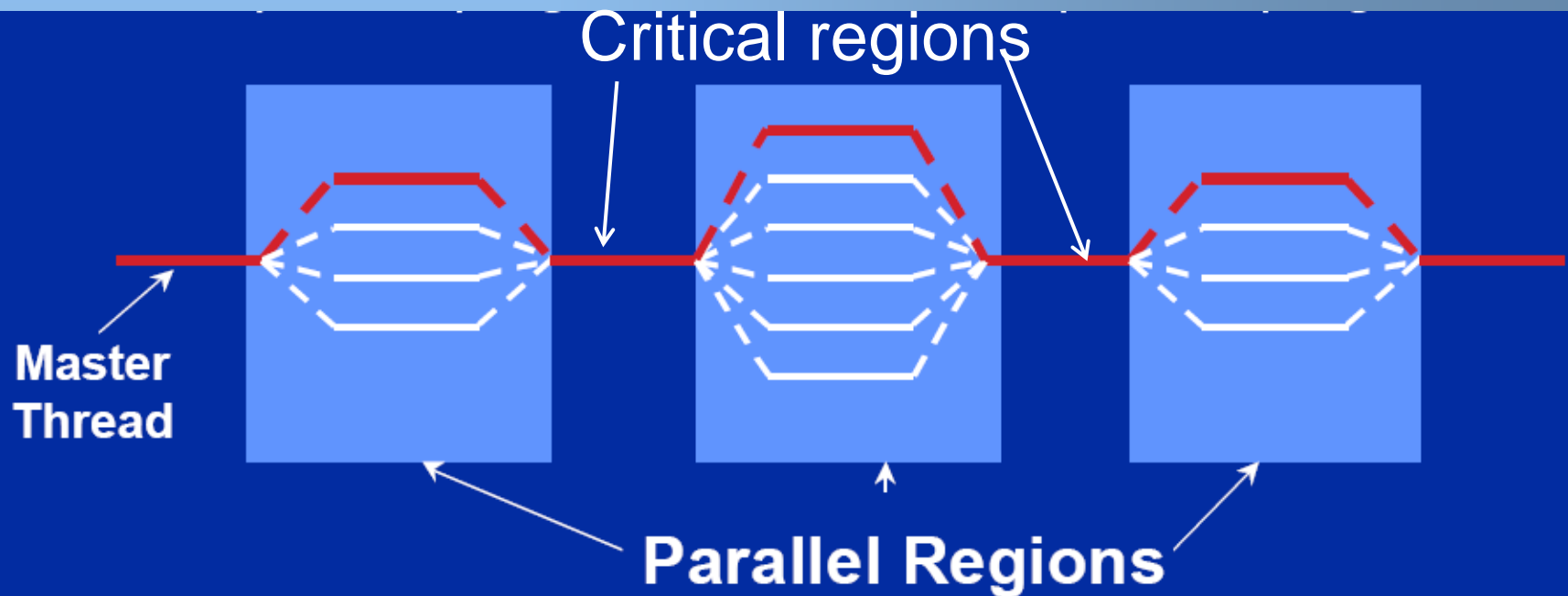
Programa Secuencial

```
void main() {  
    double a[1000],b[1000],c[1000];  
    for (int i = 0; i< 1000; i++){  
        a[i] = b[i] + c[i];  
    }  
}
```

Programa Paralelo

```
void main() {  
    double a[1000],b[1000],c[1000];  
#pragma omp parallel for  
    for (int i = 0; i< 1000; i++){  
        a[i] = b[i] + c[i];  
    }  
}
```


Regiones paralelas y críticas (I)



OpenMP runtime library

`omp_get_num_threads()`

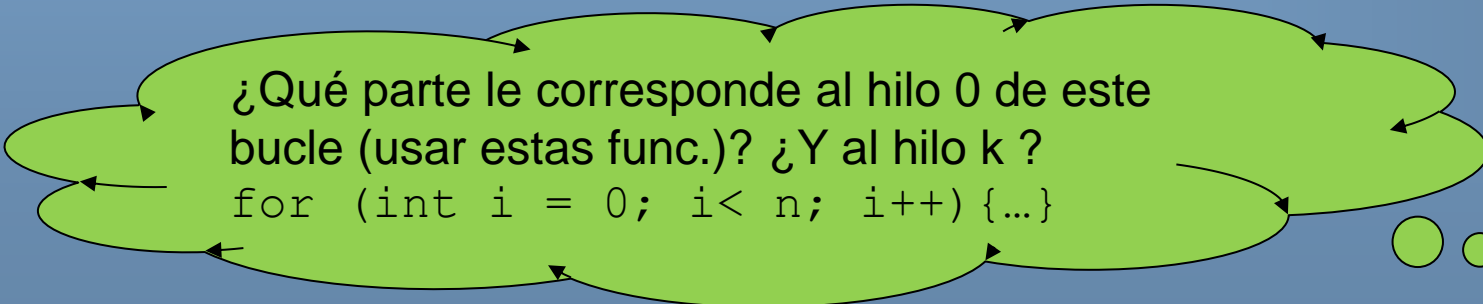
- devuelve el número actual de threads

`omp_get_thread_num()`

- devuelve el identificador de ese thread

`omp_set_num_threads(n)`

- activa el número de threads



¿Qué parte le corresponde al hilo 0 de este bucle (usar estas func.)? ¿Y al hilo k ?

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

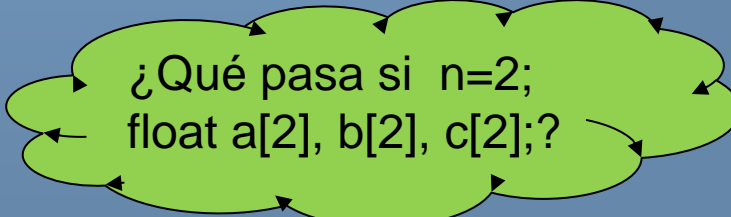


Declarando Ámbito de datos (I)

- **shared**: variable es compartida por todos los procesos. Ej: vectores del proceso.
- **private**: Cada proceso tiene una copia

```
#pragma omp parallel for
    shared(a,b,c,n) private(i, temp)
for (i = 0; i < n; i++) {
    temp = a[i]/b[i];
    a[i] = b[i] + temp * c[i];
}
```

- Hay reglas para decidir por defecto el ámbito, pero mejor no arriesgarse
- CUIDADO: false sharing



¿Qué pasa si `n=2`;
`float a[2], b[2], c[2];`?

Declarando Ámbito de datos (II)

- Variables REDUCTION: operaciones colectivas sobre elementos de un array

```
asum = 0.0;
```

```
aproduct = 1.0;
```

```
#pragma omp parallel for reduction(+:asum)  
reduction(*:aproduct)
```

```
for (i = 0; i < n; i++) {  
    asum = asum + a[i];  
    aproduct = aproduct * a[i];  
}
```

EJEMPLO: Cálculo de PI (Secuencial)

```
static long num_steps = 1000000;
double step;
void main () {
int i; double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
for (i=1;i<= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x) ;
}
pi = step * sum;
}
```

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = \arctg(x) \Big|_0^1 = \frac{\pi}{4} - 0$$

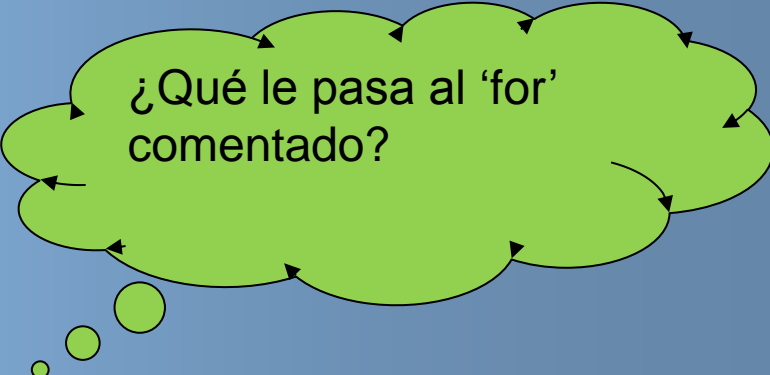
EJEMPLO: Cálculo de PI (Omp Reduction)

```
#include <omp.h>
static long num_steps = 100000; double step;
void main () {
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
omp_set_num_threads(2);

#pragma omp parallel for reduction(+:sum)
    private(x)
    for (i=1; i<= num_steps; i++) {
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Cálculo de PI: omp “reducción artesana”

```
#include <omp.h>
static long num_steps = 100000; double step;
void main () {
int i; double x, pi, sum[2];
step = 1.0/(double) num_steps;
omp_set_num_threads(2);
#pragma omp parallel
{ double x; int id;
  id = omp_get_thread_num();
  //for (i=id, sum[id]=0.0;i<= num_steps; i=i+2){
  for (i=id* num_steps/2, sum[id]=0.0;
    i<= (id+1)* num_steps/2; i++ )    {
    x = (i+0.5)*step;
    sum[id] += 4.0/(1.0+x*x);
  }
}
for(i=0, pi=0.0;i<2;i++) pi+=sum[i]*step;
}
```



¿Qué le pasa al ‘for’ comentado?

Planificación de Tareas: SCHEDULE (I)

Diferentes formas de asignar iteraciones a *threads*

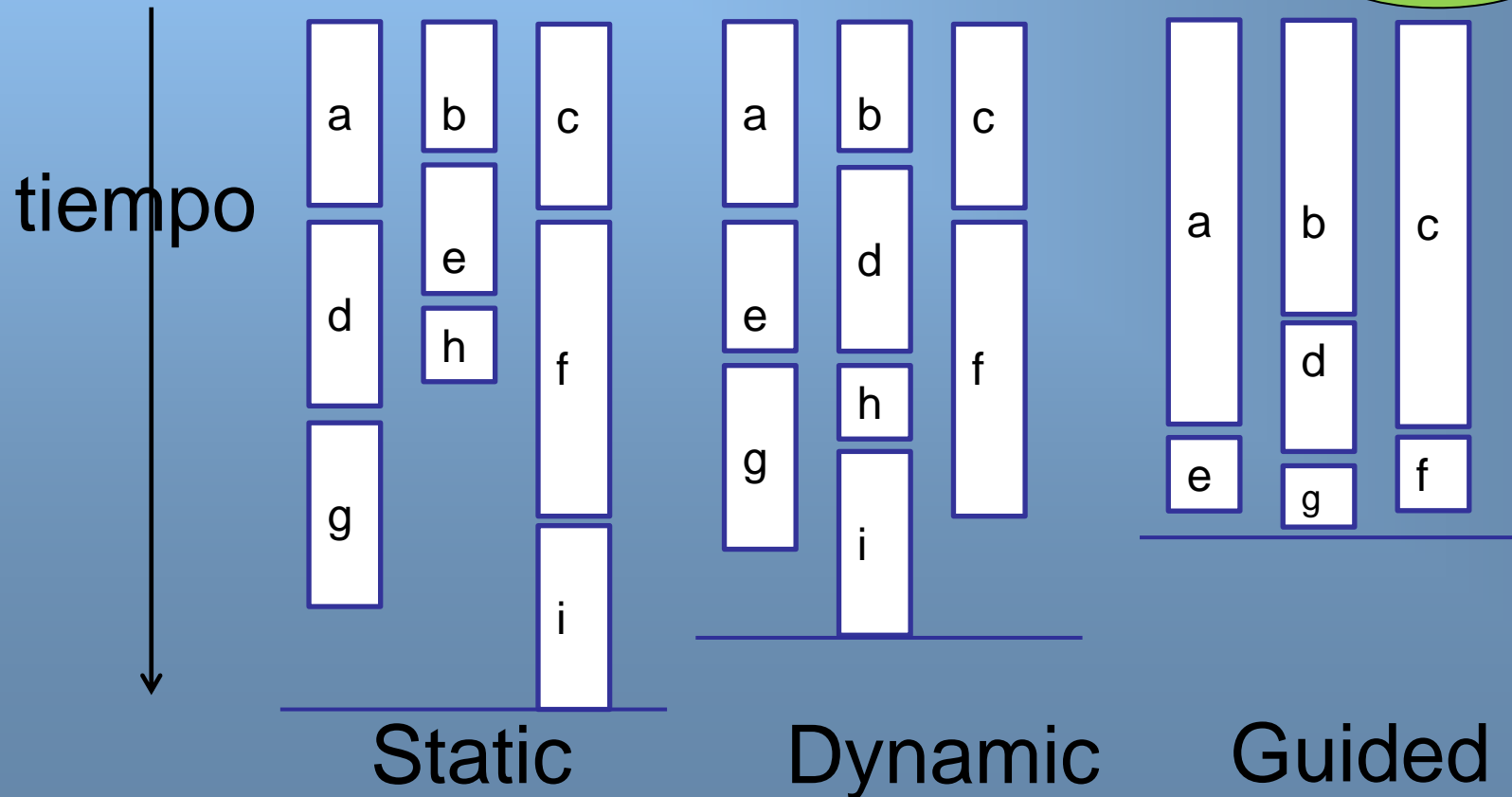
- `schedule(static [,chunk])`
 - “chunk” iteraciones se asignan de manera estática a los *threads en round-robin*
- `schedule (dynamic [,chunk])`
 - Cada thread toma “chunk” iteraciones cada vez que está sin trabajo
- `schedule (guided [,chunk])`
 - Cada *thread* toma progresivamente menos iteraciones (dinámicamente)

Planificación de Tareas: SCHEDULE (II)

- igual num iteraciones para static, dynamic.
- exponencialm. decreciente para guided

• Hilo 0 1 2 0 1 2

Static: bucles simples
Dynamic si carga varia



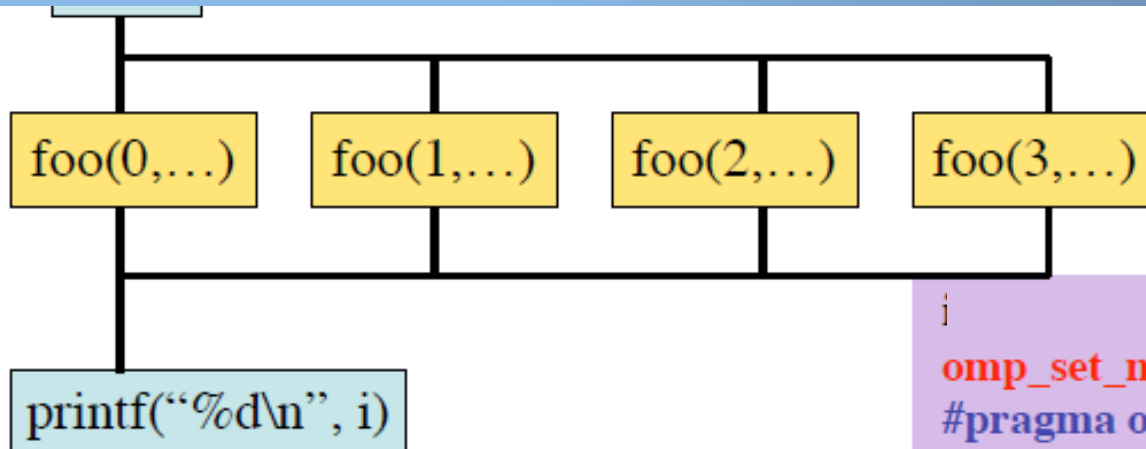
Regiones Paralelas

#pragma omp parallel

{

/* Bloque básico replicado (ejecutado) por
cada thread */

}



Control de tareas en
f(thread_id). Pag 19
¿Qué imprime?

```
i  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
  • i = omp_thread_num();  
    foo(i,a,b,c);  
}  
#pragma omp end parallel  
printf(\"%d\\n\", i);
```

Secciones (una por hilo)

```
#pragma omp parallel sections
```

```
{
```

```
    #pragma omp section
```

```
    {
```

```
        printf ("id s1=%d,\n", omp_get_thread_num());
```

```
        for (i=0; i<tam/2; i++)
```

```
            y[i] = a*x[i] + y[i];
```

```
    }
```

```
    #pragma omp section
```

```
    {
```

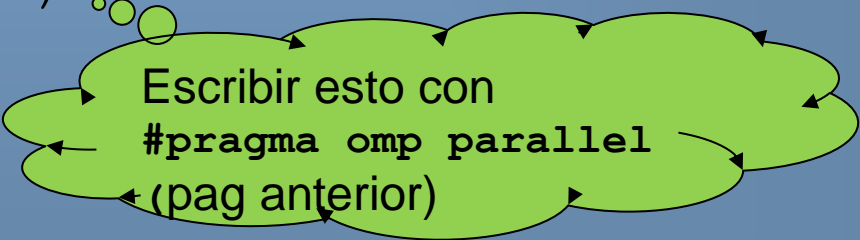
```
        printf ("id s2=%d,\n", omp_get_thread_num());
```

```
        for (i=tam/2; i<tam; i++)
```

```
            y[i] = a*x[i] + y[i];
```

```
    }
```

```
}
```



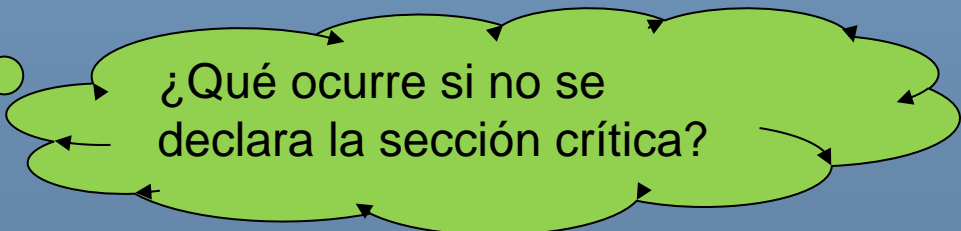
Escribir esto con
#pragma omp parallel
(pag anterior)

Regiones paralelas y críticas

- Los threads se comunican utilizando variables compartidas.
- El uso inadecuado de variables compartidas origina carreras
- Uso de sincronización o exclusión para evitarlas
- NOTA: sincronización es muy costosa en tiempo: usar lo menos posible

Exclusión Mutua: Sección Crítica

```
#pragma omp parallel shared(x,y)
{
    ...
    #pragma omp critical (section1)
        actualiza(x); //only one thread
    ...
    usa (x) ;
    ...
    #pragma omp critical(section2)
        actualiza(y); //only one thread
    ...
    usa (y) ;
}
```



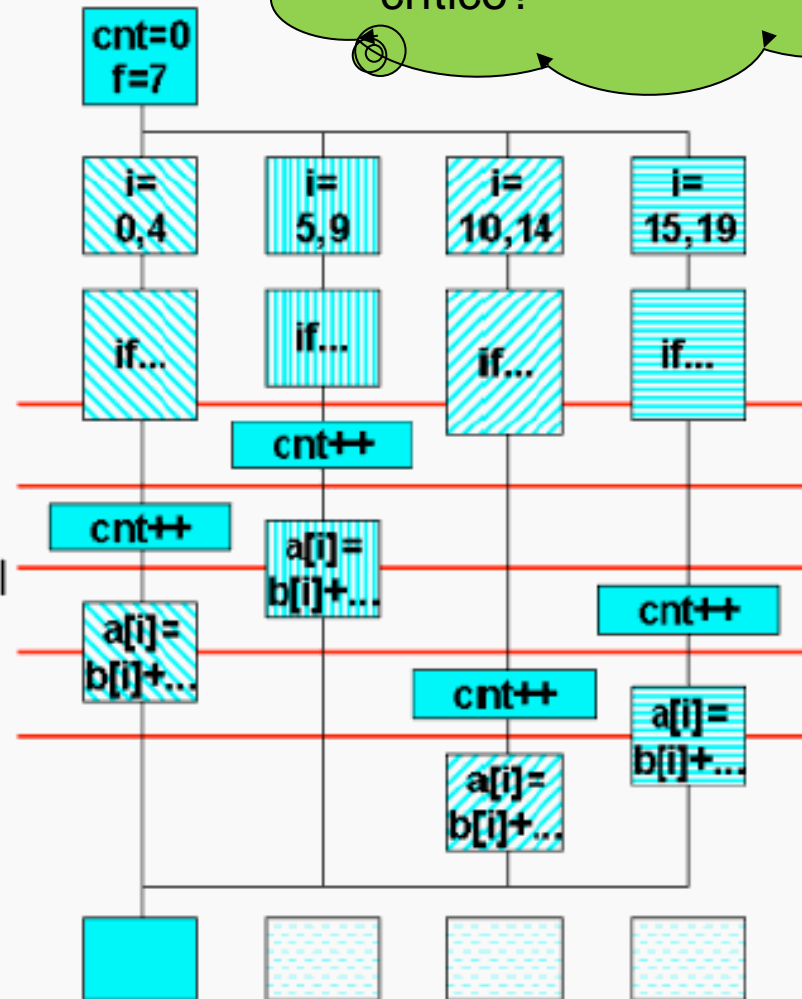
¿Qué ocurre si no se declara la sección crítica?

Exclusión Mutua: Sección Crítica EJ.

¿Qué pasa si cnt++ no se declara como crítico?

```
C++: cnt = 0;
      f=7;
#pragma omp parallel
{
#pragma omp for
  for (i=0; i<20; i++) {
    if (b[i] == 0) {
      #pragma omp critical
      cnt ++;

      } /* endif */
      a[i] = b[i] + f * (i+1);
    } /* end for */
  } /*omp end parallel */
```



Sincronización: Barreras

↓ Los *threads* se detienen hasta que alcancen la barrera

```
nt=omp_get_num_threads();
```

#pragma omp parallel private (i, id)

{

```
id=omp_get_thread_num();
```

```
for (i=id*tam/nt; i<(id+1)*tam/nt; i++) {
```

```
    y[i] = a*x[i] + y[i];
```

```
}
```

#pragma omp barrier

// aquí seguro que todo y[] está actualizado

```
for (i=id*tam/nt; i<(id+1)*tam/nt; i++) {
```

```
    z[i] = b + y[tam-i-1];
```

```
}
```

```
}
```

¿Qué pasa si se fusionan los bucles?