
. 2 .

PROGRAMACIÓN DE SISTEMAS DE MEMORIA COMPARTIDA (SMP): OPENMP



1. Introducción

Programación de Aplicaciones Paralelas. OpenMP.

- 2. Regiones paralelas.
- 3. Sincronización de los *threads*.
- 4. Tareas.
- 5. Otras cuestiones.

- ▶ Los sistemas paralelos MIMD presentan dos arquitecturas diferenciadas: **memoria compartida** y **memoria distribuida**.
- ▶ El modelo de memoria utilizado hace que la programación de aplicaciones paralelas para cada caso sea esencialmente diferente.

Alternativas:

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

- ▶ En el caso de memoria compartida, tenemos más opciones:
 - > trabajar con procesos (UNIX)
 - > usar *threads*: Pthreads (POSIX), Java, OpenMP...




- ▶ Para los sistemas de **memoria compartida**, de tipo SMP, la herramienta más extendida es **OpenMP**.
- ▶ Para los sistemas de memoria distribuida, el estándar actual de programación, mediante **paso de mensajes**, es **MPI**.
- ▶ En ambos casos hay más opciones, y en una máquina más general utilizaremos probablemente una mezcla de ambos.

▶ Qué esperamos de una “herramienta” para programar aplicaciones en máquinas de **memoria distribuida** (MPI):

- un mecanismo de **identificación** de los procesos.
- una librería de funciones de comunicación punto a punto: **send, receive...**
- funciones de comunicación global: **broadcast,, scatter, reduce...**
- alguna función para **sincronizar** procesos.

▶ Qué esperamos de una “herramienta” para programar aplicaciones en máquinas de **memoria compartida**:

- un mecanismo de **identificación** de los *threads*.
- un método para **declarar las variables** como privadas (*private*) o compartidas (*shared*).
- un mecanismo para poder definir “**regiones paralelas**”, bien sea a nivel de función o a nivel de iteración de bucle.
- facilidades para **sincronizar** los *threads*.

 **OpenMP** es el estándar actual para programar aplicaciones paralelas en sistemas de **memoria compartida tipo SMP**.

Entre otras características, es **portable**, permite **paralelismo “incremental”**, y es **independiente** del **hardware**.

Participan en su desarrollo los fabricantes más importantes: HP, IBM, SUN, SG...



No se trata de un nuevo lenguaje de programación, sino de un **API** (*application programming interface*) formado por:

- **directivas para el compilador**
- unas pocas **funciones de biblioteca**
- algunas **variables de entorno**



▶ El uso de **directivas** facilita la portabilidad y la “paralelización incremental”.

En entornos que **no usan OpenMP**, las **directivas** son tratadas como simples **comentarios** e ignoradas.



▶ Los lenguajes base con los que trabaja OpenMP son Fortran y C/C++.

▶ Las directivas de OpenMP se especifican de la siguiente manera:

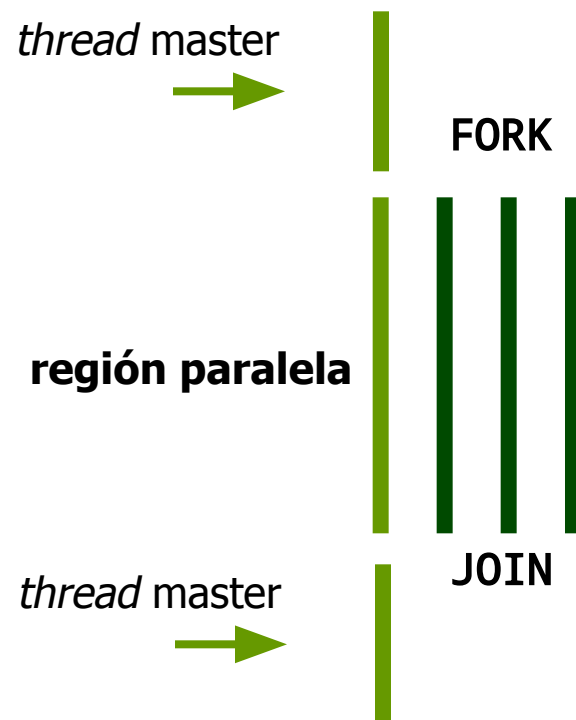
para C: `#pragma omp <directiva>`

para Fortran `!$omp <directiva>`



El modelo de programación paralela que aplica OpenMP es **Fork - Join**.

En un determinado momento, el *thread* master genera P *threads* que se ejecutan en paralelo.





Todos los *threads* ejecutan la **misma copia** del código (**SPMD**). A cada *thread* se le asigna un identificador (**tid**).



Para diferenciar las tareas ejecutadas por cada *thread*:



if (tid == 0) then ... else ...



constructores específicos de reparto de tareas (*work sharing*).

```
main () {  
    #pragma omp parallel private(tid)  
    {  
        tid = omp_get_thread_num();  
        printf (" thread %d en marcha \n", tid);  
  
        #pragma omp for schedule(static) reduction(+:B)  
        for (i=0; i<1000; i++)  
        {    A[i] = A[i] + 1;  
            B = B + A[i];  
        }  
  
        if (tid==0)    printf(" B = %d \n", B);  
    }  
}
```



Aspectos básicos a tratar en la paralelización de código:

- 1 Partiendo de un programa serie, hay que especificar qué partes del código pueden ejecutarse en paralelo (análisis de dependencias)
 - ▶ estructuras de control paralelo
 - ▶ reparto de tareas



Aspectos básicos a tratar en la paralelización de código:

- 2 Incluir la **comunicación** adecuada entre los diferentes *threads* que van a ejecutarse en paralelo. En este caso, a través de **variables compartidas** en el espacio común de memoria.

 ámbito de las variables

Aspectos básicos a tratar en la paralelización de código:

- 3 Sincronizar convenientemente la ejecución de los hilos. Las funciones principales de sincronización son las habituales: **exclusión mutua** y sincronización por **eventos** (por ejemplo, global mediante barreras).



- ▶ En resumen, partiendo de un programa serie, para obtener un programa paralelo OpenMP hay que añadir:
- **directivas** que especifican una región paralela (código replicado), **reparto de tareas** (específicas para cada *thread*), o **sincronización** entre *threads*.
 - **funciones de biblioteca** (`include <omp.h>`): para gestionar o sincronizar los *threads*.

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.

2. Regiones paralelas.

Gestión de *threads* y ámbito de las vbles.

Reparto de tareas.

Bucles FOR. Planificación de las iteraciones.

Secciones paralelas.

3. Sincronización de los *threads*.

4. Tareas.

5. Otras cuestiones.

- ▶ Una región paralela define un trozo de código que va a ser **replicado** y ejecutado en paralelo por varios *threads*.

La directiva correspondiente es (C):

```
#pragma omp parallel [cláusulas]
{
    código
}
```

El trozo de código que se define en una región paralela debe ser un **bloque básico**.

- ▶ El **número de *threads*** que se generan para ejecutar una región paralela se controla:
- a.** estáticamente, mediante una variable de entorno:
> **export OMP_NUM_THREADS=4**
 - b.** en ejecución, mediante una función de librería:
omp_set_num_threads(4);
 - c.** en ejecución, mediante una cláusula del “pragma parallel”:
num_threads(4)

▶ ¿Quién soy yo? ¿Cuántos somos?

Cada proceso paralelo se identifica por un número de *thread*. El 0 es el *thread* máster.

Dos funciones de librería:

```
tid = omp_get_thread_num();  
devuelve el identificador del thread.
```

```
nth = omp_get_num_threads();  
devuelve el número de hilos generados.
```



> Un ejemplo sencillo:

```
...  
#define N 12  
int i, tid, nth, A[N];  
main ( ) {  
    for (i=0; i<N; i++) A[i]=0;  
    #pragma omp parallel private(tid,nth) shared(A)  
    {  
        nth = omp_get_num_threads ();  
        tid = omp_get_thread_num ();  
        printf ("Thread %d de %d en marcha \n", tid, nth);  
        A[tid] = 10 + tid;  
        printf (" El thread %d ha terminado \n", tid);  
    }  
    for (i=0; i<N; i++) printf ("A(%d) = %d \n", i, A[i]);  
}
```

barrera



- ▶ 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 como *stack frame* para las rutinas invocadas por el *thread*.

- ▶ La compartición de 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**.

- ▶ Para poder especificar adecuadamente el **ámbito** de validez de cada variable, se añaden una serie de **cláusulas** a la directiva **parallel**, en las que se indica el carácter de las variables que se utilizan en dicha región paralela.

- ▶ La región paralela tiene como **extensión estática** el código que comprende, y como **extensión dinámica** el código que ejecuta (incluidas rutinas).
Las cláusulas que incluye la directiva **afectan únicamente al ámbito estático** de la región.

Las cláusulas principales que definen el ámbito de las variables son las siguientes:

- **shared (X)**

Se declara la variable **x** como **compartida** por todos los *threads*.

Sólo existe una copia, y todos los *threads* acceden y modifican dicha copia.

- **private (Y)**

Se declara la variable **y** como **privada** en cada *thread*.
Se crean P copias, una por *thread* (sin inicializar!).

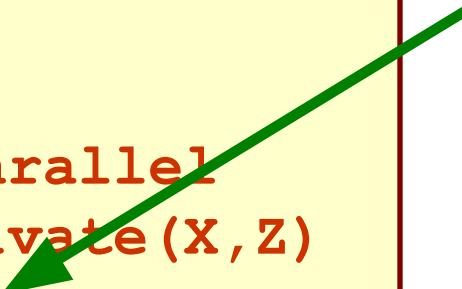
Se destruyen al finalizar la ejecución de los *threads*.



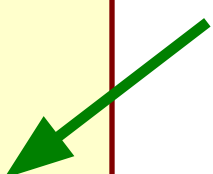
> Ejemplo:

```
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



Se declaran **objetos completos**: no se puede declarar un elemento de un array como compartido y el resto como privado.

Por defecto, las variables son **shared**.

Cada *thread* utiliza su propia pila, por lo que las variables declaradas en la propia región paralela (o en una rutina) son privadas.



- **firstprivate()**

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

Para poder pasar un valor a estas variables hay que declararlas **firstprivate**.



> Ejemplo:

```
X = Y = Z = 0;  
#pragma omp parallel  
  private(Y) firstprivate(Z)  
{  
    ...  
    X = Y = Z = 1;  
}  
...
```

valores **dentro** de la
región paralela?

X = **0**

Y = **?**

Z = **0**

valores **fuera** de la
región paralela?

X = **1**

Y = **? (0)**

Z = **? (0)**

- **reduction()**

Las operaciones de reducción son típicas en muchas aplicaciones paralelas. Utilizan variables a las que acceden todos los procesos y sobre las que se efectúa alguna operación de “acumulación” en modo atómico (RMW).


Caso típico: la suma de los elementos de un vector.

Si se desea, el control de la operación puede dejarse en manos de OpenMP, declarando dichas variables de tipo **reduction**.



> Ejemplo:

```
#pragma omp parallel private(X) reduction(+:sum)
{
    X = ...
    ...
    sum = sum + X;
    ...
}
```



La propia cláusula indica el operador de reducción a utilizar.

OJO: no se sabe en qué orden se va a ejecutar la operación
--> debe ser conmutativa (cuidado con el redondeo).



- **default (none / shared)**

none: obliga a declarar explícitamente el ámbito de todas las variables. Útil para no olvidarse de declarar ninguna variable (da error al compilar).

shared: las variables sin “declarar” son shared (por defecto).

(En Fortran, también `default(private)`: las variables sin declarar son privadas)



- Variables de tipo **threadprivate**

Las cláusulas de ámbito sólo afectan a la extensión estática de la región paralela. Por tanto, una variable privada sólo lo es en la extensión estática (salvo que la pasemos como parámetro a una rutina).

Si se quiere que una variable sea privada pero en toda la extensión dinámica de la región paralela, entonces hay que declararla mediante la directiva:

#pragma omp threadprivate (X)



Las variables de tipo **threadprivate** deben ser “estáticas” o globales (declaradas fuera, antes, del `main`). Hay que especificar su naturaleza justo después de su declaración.

Las variables **threadprivate** no desaparecen al finalizar la región paralela (mientras no se cambie el número de *threads*); cuando se activa otra región paralela, siguen activas con el valor que tenían al finalizar la anterior región paralela.



- **copyin(X)**

Declarada una variable como **threadprivate**, un *thread* no puede acceder a la copia **threadprivate** de otro *thread* (ya que es privada).

La cláusula **copyin** permite copiar en cada *thread* el valor de esa variable en el *thread* máster al comienzo de la región paralela.



- **if (expresión)**

La región paralela sólo se ejecutará en paralelo si la expresión es distinta de 0.

Dado que paralelizar código implica **costes** añadidos (generación y sincronización de los *threads*), la cláusula permite decidir en ejecución si merece la pena la ejecución paralela según el tamaño de las tareas (por ejemplo, en función del tamaño de los vectores a procesar).



- **num_threads (expresión)**

Indica el número de hilos que hay que utilizar en la región paralela.

Precedencia: vble. entorno >> función >> cláusula



Cláusulas de la región paralela

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

Índice

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.

2. **Regiones paralelas.**

Gestión de *threads* y ámbito de las vbles.

Reparto de tareas.

Bucles `for`. Planific. de las iteraciones

Secciones paralelas.

3. Sincronización de los *threads*.
4. Tareas.
5. Otras cuestiones.




Paralelización de bucles.

Los bucles son uno de los puntos de los que extraer paralelismo de manera “sencilla” (paralelismo de datos, *domain decomposition*, grano fino).

Obviamente, la simple replicación de código no es suficiente. Por ejemplo,

```
#pragma omp parallel shared(A) private(i)
{
    for (i=0; i<100; i++)
        A[i] = A[i] + 1;
}
```



Tendríamos que hacer algo así:

```
#pragma omp parallel shared(A)
                    private(tid,nth, ini,fin,i)
{   tid = omp_get_thread_num();
    nth = omp_get_num_threads();

    ini = tid * 100 / nth;
    fin = (tid+1) * 100 / nth;

    for (i=ini; i<fin; i++) A[i] = A[i] + 1;
}
```



- ▶ El reparto de trabajo entre los *threads* se puede hacer mediante una estrategia de tipo **SPMD**.
 - Utilizando el identificador de los *threads* y mediante sentencias `if`.
 - Definiendo una cola de tareas y efectuando un reparto dinámico de las mismas (*self-scheduling*, p. e.).
- ▶ OpenMP ofrece una alternativa “automática” al reparto de tareas “manual”, mediante el uso de directivas específicas de reparto de tareas (*work sharing*).

 Las opciones de que disponemos son:

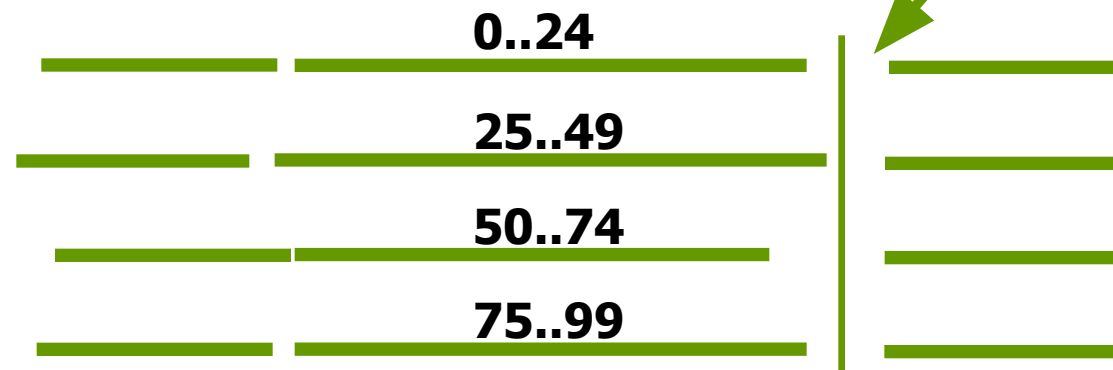
1. Directiva **for**, para repartir la ejecución de las iteraciones de un bucle entre todos los *threads* (bloques básicos y número de iteraciones conocido).
2. Directiva **sections**, para definir trozos o secciones de una región paralela a repartir entre los *threads*.
3. Directiva **single**, para definir un trozo de código que sólo debe ejecutar un *thread*.

1 Directiva **for**

```
#pragma omp parallel [...]  
{ ...  
    #pragma omp for [clausulas]  
    for (i=0; i<100; i++) A[i] = A[i] + 1;  
    ...  
}
```

ámbito variables
reparto iteraciones
sincronización

barrera



- ▶ Las directivas `parallel` y `for` pueden juntarse en `#pragma omp parallel for` cuando la región paralela contiene únicamente un bucle.
- ▶ En todo caso, la decisión de paralelizar un bucle debe tomarse tras el correcto análisis de las dependencias.

▶ Para facilitar la paralelización de un bucle, hay que aplicar todas las **optimizaciones** conocidas para la “eliminación” de dependencias:

- variables de inducción
- reordenación de instrucciones
- alineamiento de las dependencias
- intercambio de bucles, etc.

```
for (i=0; i<N; i++)  
    Z[i] = a * X[i] + b;
```



```
#pragma omp parallel for  
for (i=0; i<N; i++)  
    Z[i] = a * X[i] + b;
```

El bucle puede paralelizarse sin problemas, ya que todas las iteraciones son independientes.

La directiva **parallel for** implica la generación de *P threads*, que se repartirán la ejecución del bucle.

Hay una **barrera de sincronización** implícita al final del bucle. El máster retoma la ejecución cuando todos terminan.

El índice del bucle, **i**, es una variable **privada** (no es necesario declararla como tal).

```
for (i=0; i<N; i++)  
  for (j=0; j<M; j++)  
  {  
    X = B[i][j] * B[i][j];  
    A[i][j] = A[i][j] + X;  
    C[i][j] = X * 2 + 1;  
  }
```

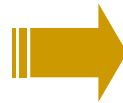


```
#pragma omp parallel for  
  private (j,X)  
  
for (i=0; i<N; i++)  
  for (j=0; j<M; j++)  
  {  
    X = B[i][j] * B[i][j];  
    A[i][j] = A[i][j] + X;  
    C[i][j] = X * 2 + 1;  
  }
```

Se ejecutará en paralelo el **bucle externo**, y los *threads* ejecutarán el bucle interno. Paralelismo de grano “medio”.

Las variables **j** y **x** deben declararse como **privadas**.

```
for (i=0; i<N; i++)  
  for (j=0; j<M; j++)  
  {  
    X = B[i][j] * B[i][j];  
    A[i][j] = A[i][j] + X;  
    C[i][j] = X * 2 + 1;  
  }
```



```
for (i=0; i<N; i++)  
  #pragma omp parallel for  
    private (X)  
    for (j=0; j<M; j++)  
    {  
      X = B[i][j] * B[i][j];  
      A[i][j] = A[i][j] + X;  
      C[i][j] = X * 2 + 1;  
    }
```

Los *threads* ejecutarán en paralelo el **bucle interno** (el externo se ejecuta en serie). Paralelismo de grano fino.

La variable **x** debe declararse como **privada**.

 Las cláusulas de la directiva `for` son de varios tipos:

- ✓ **scope** (ámbito): indican el ámbito de las variables.
- ✓ **schedule** (planificación): indican cómo repartir las iteraciones del bucle.
- ✓ **collapse**: permite colapsar varios bucles en uno.
- ✓ **nowait**: elimina la barrera final de sincronización.
- ✓ **ordered**: impone orden en la ejecución de las iteraciones.

- ▶ Las cláusulas de ámbito de una directiva `for` son (como las de una región paralela):

`private, firstprivate,`
`reduction, default`

Y se añade una cláusula más:

- `lastprivate(X)`

Permite salvar el valor de la variable privada `x` correspondiente a la última iteración del bucle.

▶ ¿Cómo se **reparten** las iteraciones de un bucle entre los *threads*?

Puesto que el `pragma for` termina con una **barrera**, si la **carga** de los *threads* está **mal equilibrada** tendremos una pérdida (notable) de eficiencia.

▶ La cláusula **`schedule`** permite definir diferentes estrategias de reparto, tanto **estáticas** como **dinámicas**.



La sintaxis de la cláusula es:

```
schedule(tipo [, tamaño_trozo])
```

Los tipos pueden ser:

- **static, k**

Planificación estática con trozos de tamaño k.

Si no se indica k, se reparten trozos de tamaño N/P.
La asignación es entrelazada (*round robin*).

- **dynamic, k**

Asignación dinámica de trozos de tamaño k.

El tamaño por defecto es 1.

- **guided, k**

Planificación dinámica con trozos de tamaño decreciente:

$$K_{i+1} = K_i (1 - 1/P)$$

El tamaño del primer trozo es dependiente de la implementación y el último es el especificado (por defecto, 1).



- **runtime**

El tipo de planificación se define previamente a la ejecución en la variable de entorno **OMP_SCHEDULE** (para las pruebas previas).

Por ej.: > **export OMP_SCHEDULE="dynamic,3"**

- **auto**

La elección de la planificación la realiza el compilador (o el *runtime system*).

Es dependiente de la implementación.

3.
0



Bajo ciertas condiciones, la asignación de las iteraciones a los *threads* se puede mantener para diferentes bucles de la misma región paralela.

3.
0

Se permite a las implementaciones añadir nuevos métodos de planificación.

RECUERDA:

estático menos coste / mejor localidad datos

dinámico más coste / carga más equilibrada



- **collapse(n)**

El compilador formará un único bucle y lo paralelizará.
Se le debe indicar el número de bucles a colapsar

3.
0

```
#pragma omp parallel for collapse(2)
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      for (k=0; k<L; k++)
      {
        ...
      }
```





Por defecto, una **región paralela** o un **for** en paralelo (en general, casi todos los constructores de OpenMP) llevan implícita una **barrera** de **sincronización final** de todos los *threads*.

El más lento marcará el tiempo final de la operación.



Puede eliminarse esa barrera en el **for** mediante la cláusula **nowait**, con lo que los *threads* continuarán la ejecución en paralelo sin sincronizarse entre ellos.



Cuando la directiva es `parallel for` (una región paralela que sólo tiene un bucle `for`), pueden usarse las cláusulas de ambos pragmas. Por ejemplo:

```
#pragma omp parallel for if(N>1000)
for (i=0; i<N; i++) A[i] = A[i] + 1;
```





Reparto de tareas (bucles `for`)

`#pragma omp for [clausulas]`

- `private`(var) `firstprivate`(var)
`reduction`(op:var) `default`(shared/none)
`lastprivate`(var)

-

`schedule`(static/dynamic/guided/runtime/auto[tam])

- `collapse`(n)
- `nowait`

`#pragma omp parallel for [claus.]`



Índice

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.

2. **Regiones paralelas.**

Gestión de *threads* y ámbito de las vbles.

Reparto de tareas.

Bucles `for`. Planificación de las iteraciones

Secciones paralelas.

3. Sincronización de los *threads*.
4. Tareas.
5. Otras cuestiones.



2 Directiva `sections`

Permite usar **paralelismo de función** (*function decomposition*). Reparte secciones de **código independiente** a *threads* diferentes.

Cada sección paralela es ejecutada por un sólo *thread*, y cada *thread* ejecuta ninguna o alguna sección.

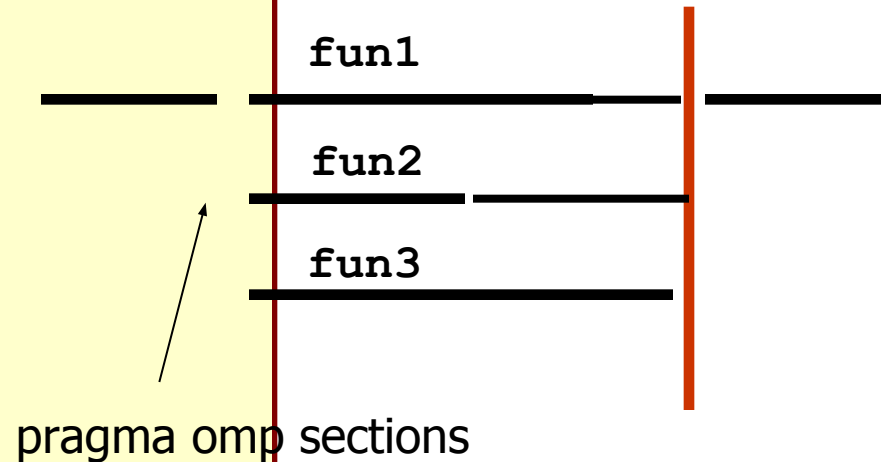
Una barrera implícita sincroniza el final de las secciones o tareas.

Cláusulas: `private (first-, last-),`
`reduction, nowait`



> Ejemplo:

```
#pragma omp parallel [clausulas]
{
  #pragma omp sections [clausulas]
  {
    #pragma omp section
    fun1();
    #pragma omp section
    fun2();
    #pragma omp section
    fun3();
  }
}
```





Igual que en el caso del pragma `for`, si la región paralela sólo tiene secciones, pueden juntarse ambos pragmas en uno solo:

```
#pragma omp parallel sections [cláusulas]
```

(cláusulas suma de ambos pragmas)



2 Directiva *single*

Define un bloque básico de código, dentro de una región paralela, que no debe ser replicado; es decir, que *debe ser ejecutado por un único thread*.
(por ejemplo, una operación de entrada/salida).

No se especifica qué *thread* ejecutará la tarea.



- ▶ La salida del bloque **single** lleva implícita una barrera de sincronización de todos los *threads*. La sintaxis es similar a las anteriores:

```
#pragma omp single [cláus.]
```

Cláusulas: (**first**)**private**, **nowait**
copyprivate(X)

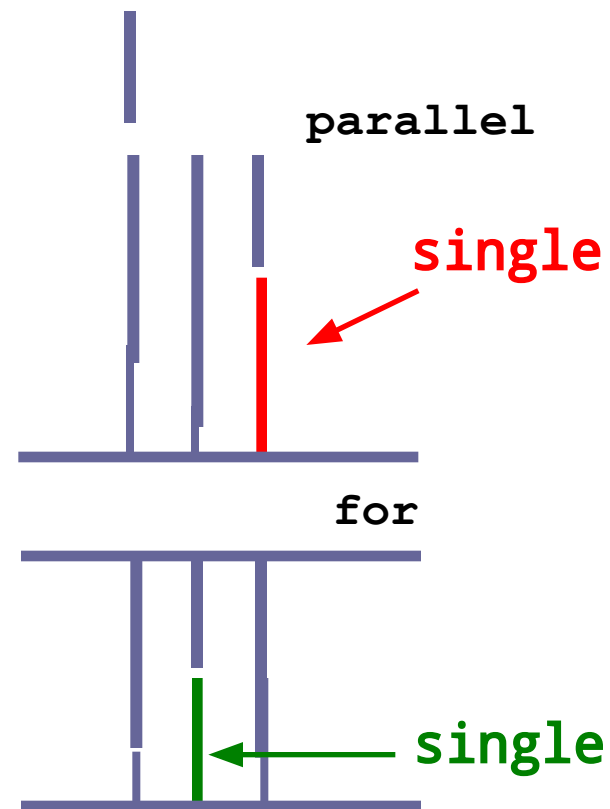
Para pasar al resto de *threads* (BC) el valor de una variable **threadprivate**, modificada dentro del bloque **single**.

> Ejemplo:

```
#pragma omp parallel
{
    ... ;
    #pragma omp single
    inicializar(A) ;

    #pragma omp for
    for(i=0; i<N; i++)
        A[i] = A[i] * A[i] + 1;
    ... ;

    #pragma omp single
    copiar(B,A) ;
}
```



Comentarios finales

El reparto de tareas de la región paralela debe hacerse en base a bloques básicos; además, todos los *threads* deben alcanzar la directiva.

Es decir:

- si un *thread* llega a una directiva de reparto, deben llegar todos los *threads*.
- una directiva de reparto puede no ejecutarse, si no llega ningún *thread* a ella.
- si hay más de una directiva de reparto, todos los *threads* deben ejecutarlas en el mismo orden.
- las directivas de reparto no se pueden anidar.



Las directivas de reparto pueden ir tanto en el ámbito lexicográfico (estático) de la región paralela como en el dinámico. Por ejemplo:

```
int X;  
#pragma omp  
threadprivate(X)  
...  
#pragma omp parallel  
{  X = ...  
    init();  
    ...  
}
```



```
void init()  
{  #pragma omp for  
    for (i=0; i<N;  
        i++)  
        A[i] = X * i;  
}
```

Si se llama a la función desde fuera de una región paralela (*#threads* = 1), no hay problema, simplemente no tiene ningún efecto.

Índice

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.

2. **Regiones paralelas.**

Gestión de *threads* y ámbito de las vbles.

Reparto de tareas.

Bucles `for`. Planificación de las iteraciones

Secciones paralelas.

Otras cuestiones

3. Sincronización de los *threads*.
4. Tareas.
5. Otras cuestiones.





Es posible **anidar** regiones paralelas, pero hay que hacerlo con “cuidado” para evitar problemas.

Por defecto no es posible, hay que indicarlo explícitamente mediante:

- una llamada a una función de librería

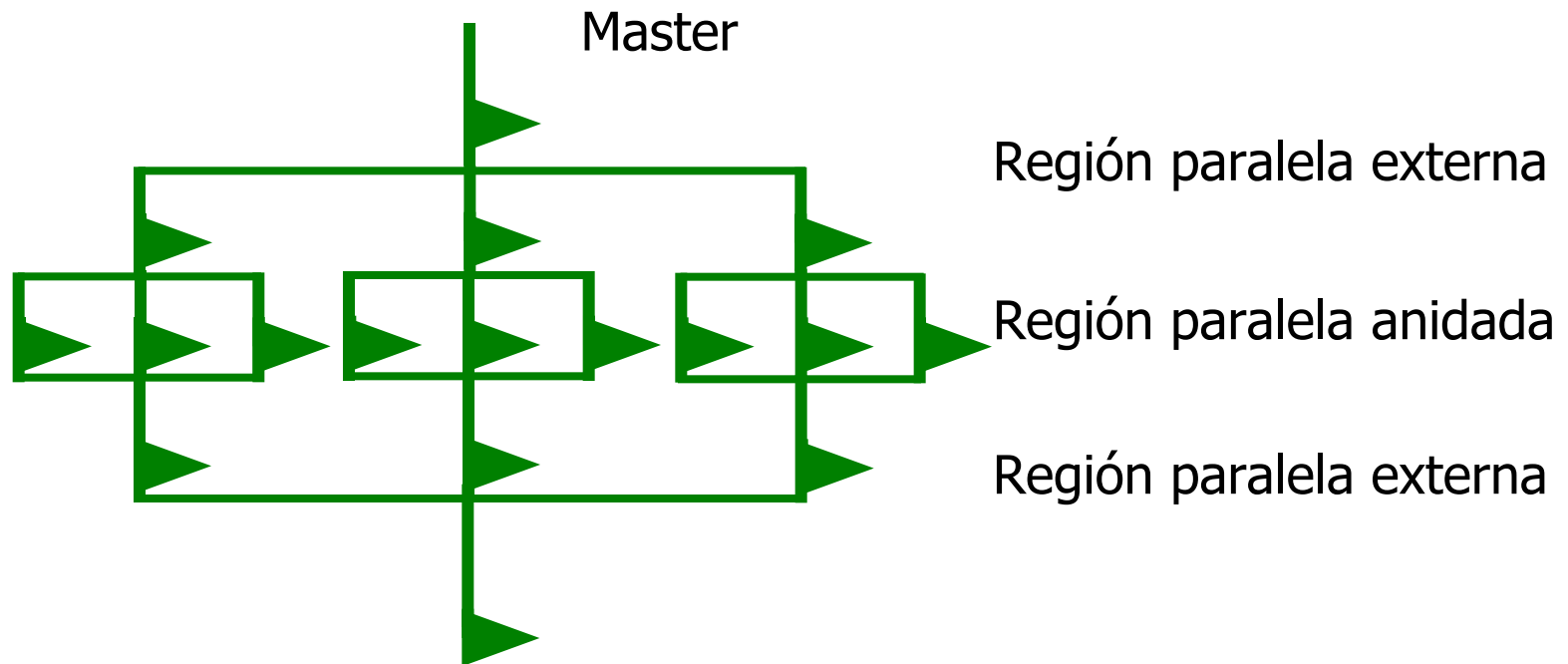
```
omp_set_nested(TRUE) ;
```

- una variable de entorno

```
> export OMP_NESTED=TRUE
```

Una función devuelve el estado de dicha opción:

```
omp_get_nested() ;    (true o false)
```



Se pueden anidar regiones paralelas con cualquier nivel de profundidad.



OpenMP 3.0 mejora el soporte al paralelismo anidado:

3.
0

- La función `omp_set_num_threads()` puede ser invocada dentro de una región paralela para controlar el grado del siguiente nivel de paralelismo.
- Permite conocer el nivel de anidamiento mediante `omp_get_level()` y `omp_get_active_level()`.
- Se puede acceder al `tid` del padre de nivel `n` mediante `omp_get_ancestor_thread_num(n)` y al número de *threads* en dicho nivel.





OpenMP 3.0 mejora el soporte al paralelismo anidado:

3.0

- Permite controlar el máximo número de regiones paralelas anidadas activas mediante funciones y variables de entorno.

```
omp_get/set_max_active_levels();
```

```
> export OMP_MAX_ACTIVE_LEVELS=n
```

- Permite controlar el máximo número de *threads* mediante funciones y variables de entorno

```
omp_get_thread_limit();
```

```
> export OMP_THREAD_LIMIT=n
```



- ▶ Puede usarse la cláusula `if` para decidir en tiempo de ejecución si paralelizar o no. Para saber si se está ejecutando en serie o en paralelo, se puede usar la función:

```
omp_in_parallel();
```

- ▶ Puede obtenerse el número de procesadores disponibles mediante

```
omp_get_num_proc();
```

y utilizar ese parámetro para definir el número de *threads*.

- ▶ Puede hacerse que el número de *threads* sea dinámico, en función del número de procesadores disponibles en cada instante:

```
> export OMP_DYNAMIC=TRUE/FALSE  
omp_set_dynamic(1/0);
```

Para saber si el número de *threads* se controla dinámicamente:

```
omp_get_dynamic();
```


▶ Se han añadido algunas variables de entorno para gestionar el tamaño de la pila:

3.0 > export **OMP_STACKSIZE** tamaño [B|K|M|G]

y para gestionar la política de espera en la sincronización:

> export **OMP_WAIT_POLICY** [ACTIVE|PASSIVE]

Índice

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.
2. Regiones paralelas.
- 3. Sincronización de los *threads*.
Exclusión mutua: secciones críticas.
Variables cerrojo.
Eventos: barreras...**
4. Tareas.
5. Otras cuestiones.



- ▶ Cuando no pueden eliminarse las dependencias de datos entre los *threads*, es necesario sincronizar su ejecución.

OpenMP proporciona los mecanismos de sincronización más habituales: **exclusión mutua** y **sincronización por eventos**.

1. Secciones Críticas

Define un trozo de código que no puede ser ejecutado por más de un *thread* a la vez.

OpenMP ofrece varias alternativas para la ejecución en exclusión mutua de secciones críticas. Las dos opciones principales son: **critical** y **atomic**.



▶ Directiva **critical**

Define una única sección crítica para todo el programa, dado que no utiliza variables de *lock*.

```
#pragma omp parallel firstprivate(MAXL)
{
    ...
    #pragma omp for
    for (i=0; i<N; i++) {
        A[i] = B[i] / C[i];
        if (A[i]>MAXL) MAXL = A[i];
    }
    #pragma omp critical
    { if (MAXL>MAX) MAX = MAXL; }
    ...
}
```

OJO: la sección crítica debe ser lo "menor" posible!

- Secciones críticas “específicas” (*named*)

También pueden definirse diferentes secciones críticas, controladas por la correspondiente variable cerrojo.

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
    A[i] = fun(i);

    if (A[i]>MAX)
        #pragma omp critical(M1)
        { if (A[i]>MAX) MAX = A[i]; }

    if (A[i]<MIN)
        #pragma omp critical(M2)
        { if (A[i]<MIN) MIN = A[i]; }
}
```



Directiva `atomic`

Es una caso particular de sección crítica, en el que se efectúa una operación RMW atómica sencilla (con limitaciones).

```
#pragma omp parallel ...  
{  
    ...  
    #pragma omp atomic  
    X = X + 1;  
    ...  
}
```

Para este tipo de operaciones, es más eficiente que definir una sección crítica.

2. Funciones con cerrojos

Un conjunto de funciones de librería permite manejar variables cerrojo y definir así secciones críticas “ad hoc”.

En C, las variables cerrojo deben ser del tipo predefinido: `omp_lock_t` **C**;

- `omp_init_lock(&C);`
reserva memoria e inicializa un cerrojo.
- `omp_destroy_lock(&C);`
libera memoria del cerrojo.



- `omp_set_lock (&C) ;`
coge el cerrojo o espera a que esté libre.
- `omp_unset_lock (&C) ;`
libera el cerrojo.
- `omp_test_lock (&C) ;`
testea el valor del cerrojo; devuelve T/F.

Permiten gran flexibilidad en el acceso en exclusión mutua. La variable de *lock* puede pasarse como parámetro a una rutina.



> Ejemplo

```
#pragma omp parallel private(nire_it)
{
    omp_set_lock(&C1);
    mi_it = i;
    i = i + 1;
    omp_unset_lock(&C1);

    while (mi_it < N)
    {
        A[mi_it] = A[mi_it] + 1;

        omp_set_lock(&C1);
        mi_it = i;
        i = i + 1;
        omp_unset_lock(&C1);
    }
}
```



- ▶ Un grupo similar de funciones permite el anidamiento de las secciones críticas, sin que se produzca un bloqueo por recursividad.

La sintaxis es la misma que en el grupo anterior, añadiendo `_nest`. Por ejemplo:

- `omp_set_nest_lock(var) ;` entrada a la SC
- `omp_unset_nest_lock(var) ;` salida de la SC

3. Eventos

La sincronización entre los *threads* puede hacerse esperando a que se produzca un determinado evento.

La sincronización puede ser:

- **global**: todos los *threads* se sincronizan en un punto determinado.
- **punto a punto**: los *threads* se sincronizan uno a uno a través de *flags*.



Sincronización global: barreras

```
#pragma omp barrier
```

Típica barrera de sincronización global para todos los *threads* de una región paralela.

Recordad que muchos constructores paralelos llevan implícita una barrera final.

> Ejemplo

```
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    A[tid] = fun(tid);

    #pragma omp
    barrier
    #pragma omp for
    for (i=0; i<N; i++) B[i] = fun(A,i);

    #pragma omp for nowait
    for (i=0; i<N; i++) C[i] = fun(A,B,i);

    D[tid] = fun(tid);
}
```



▶ Sincronización punto a punto

La sincronización entre procesos puede hacerse mediante *flags* (memoria común), siguiendo un modelo de tipo *producer* / *consumer*.

```
/* productor */  
...  
dat = ...;  
flag = 1;  
...
```



```
/* consumidor */  
...  
while (flag==0) {  
};  
... = dat;  
...  
...
```

Sin embargo, sabemos que el código anterior puede no funcionar correctamente en un sistema paralelo, dependiendo del **modelo de consistencia** de la máquina.

Tal vez sea necesario **desactivar las optimizaciones** del compilador antes del acceso a las variables de sincronización.



Para asegurar que el modelo de consistencia aplicado es el secuencial, OpenMP ofrece como alternativa la directiva:

#pragma omp flush(X)

que marca **puntos de consistencia** en la visión de la memoria (*fence*).

```
/* productor */  
...  
dat = ...;  
#pragma omp flush(dat)  
flag = 1;  
#pragma omp flush(flag)  
...
```

```
/* consumidor */  
...  
while (flag==0)  
{ #pragma omp flush(flag) };  
#pragma omp flush(dat)  
... = dat;  
...
```



El modelo de consistencia de OpenMP implica tener que realizar una operación de *flush* tras escribir y antes de leer cualquier variable compartida.

En C se puede conseguir esto declarando las variables de tipo **volatile**.

```
volatile int dat, flag;  
...  
  
/* productor */  
...  
dat = ...;  
flag = 1;  
...
```

```
volatile int dat, flag;  
...  
  
/* consumidor */  
...  
while (flag==0) {};  
... = dat;  
...
```



4. Secciones “ordenadas”

`#pragma omp ordered`

Junto con la cláusula `ordered`, impone el orden secuencial original en la ejecución de una parte de código de un `for` paralelo.

```
#pragma omp parallel for ordered
for (i=0; i<N; i++)
{
    A[i] = ... (cálculo);
    #pragma omp ordered
    printf("A(%d)= %d \n", i, A[i]);
}
```

Equivale a un `wait`-`post` construido mediante contadores.



5. Directiva **master**

#pragma omp master

Marca un bloque de código para que sea ejecutado solamente por el *thread* máster, el 0.

Es similar a la directiva **single**, pero sin incluir la barrera al final y sabiendo qué *thread* va a ejecutar el código.



Índice

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.
2. Regiones paralelas.
3. Sincronización de los *threads*.
- 4. Tareas.**
5. Otras cuestiones.

3.
0



Directiva **task**

Sirve para definir tareas explícitas, y permite gestionar eficientemente procedimientos recursivos y bucles en los que el número de iteraciones es indeterminado.

```
#pragma omp task [cláusulas]
{
}
```

cláusulas **if** (expresión)
 untied
 default (shared | none)
 private (var), **firstprivate** (var), **shared** (var)



Directiva **taskwait**

Una directiva que permite esperar a todas las tareas hijas generadas desde la tarea actual.

```
#pragma omp taskwait
```

La directiva **taskwait** permite introducir paralelismo de una forma sencilla en ciertas situaciones en las que era difícil hacerlo. Veamos un par de ejemplos.



> Ejemplo 1: lista ligada

```
...  
while (puntero)  
{  
    (void) ejecutar_tarea(puntero);  
    puntero = puntero->sig;  
}  
...
```

Sin la directiva **task**, habría que contar el número de iteraciones previamente para transformar el **while** en un **for**.



> Ejemplo 1: lista ligada - openmp

```
puntero = cabecera;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(puntero) {
            #pragma omp task firstprivate(puntero)
            {
                (void) ejecutar_tarea(puntero);
            }
            puntero = puntero->sig ;
        }
    }
}
```



> Ejemplo 2: fibonacci

```
long fibonacci(int n)
{ // f(0)=f(1)=1, f(n) = f(n-1) + f(n-2)

    long f1, f2, fn;

    if ( n == 0 || n == 1 ) return(n);

    f1 = fibonacci(n-1);
    f2 = fibonacci(n-2);

    fn = f1 + f2;

    return(fn);
}
```



> Ejemplo 2: fibonacci - openmp

```
long fibonacci(int n)
{
    long f1, f2, fn;
    if ( n == 0 || n == 1 ) return(n);

    #pragma omp task shared(f1)
        {f1 = fibonacci(n-1);}

    #pragma omp task shared(f2)
        {f2 = fibonacci(n-2);}

    #pragma omp taskwait

    fn = f1 + f2;
    return(fn);
}
```



> Ejemplo 2: fibonacci - openmp

```
#pragma omp parallel shared(nth)
{
    #pragma omp single nowait
    {
        result = fibonacci(n);
    }
}
```

Posibilidad de aplicar recursividad paralela a partir de un tamaño mínimo de cálculo?



Índice

1. Introducción. Programación de Aplicaciones Paralelas. OpenMP.
2. Regiones paralelas.
3. Sincronización de los *threads*.
4. Tareas.

5. Otras cuestiones.

Funciones para medir tiempos

Carreras y deadlocks

Funciones reentrantes

Speed-up



Un par de funciones para “medir tiempos”

- `omp_get_wtime()` ;

```
t1 = omp_get_wtime() ;
```

```
...
```

```
t2 = omp_get_wtime() ;
```

```
tiempo = t2 - t1 ;
```

- `omp_get_wtick()` ; precisión del reloj

- ▶ Programar aplicaciones SMP resulta “más sencillo” que repartir datos por diferentes procesadores y comunicarse por paso de mensajes.

Pero el uso de variables compartidas por varios *threads* puede llevar a **errores** no previstos si no se analiza detenidamente su comportamiento.

Algunos errores típicos pueden producir **carreras** (*races*) en los resultados o dejar bloqueada la ejecución (*deadlock*).

Carreras

Definimos una carrera (*race*) como la consecución de resultados inesperados e irreproducibles debido a problemas en el acceso y sincronización de variables compartidas.

```
#pragma omp parallel sections
{
    #pragma omp section
        A = B + C;

    #pragma omp section
        B = A + C;

    #pragma omp section
        C = B + A;
}
```

!?


```
CONT = 0;
#pragma omp parallel sections
{
    #pragma omp section
        A = B + C;
        #pragma omp flush (A)
    CONT = 1;
    #pragma omp flush (CONT)
    #pragma omp section
    {
        while (CONT<1)
        {
            #pragma omp flush (CONT)
            B = A + C;
            #pragma omp flush (B)
            CONT = 2;
            #pragma omp flush (CONT)
        }
    }
    #pragma omp section
    {
        while (CONT<2)
        {
            #pragma omp flush (CONT)
            C = B + A;
        }
    }
}
```

el contador
permite la
sincronización
entre las
secciones
(eventos)

las operaciones
de *flush*
aseguran la
consistencia de
la memoria.



```
#pragma omp parallel private(tid, X)
{
    tid = omp_get_thread_num();

    #pragma omp for reduction(+:total)
    nowait
    for (i=0; i<N; i++)
    {
        x = fun(i);
        total = total + x;
    }
    Y[tid] = fun2(tid, total);
}
```

!?



Deadlock

Si no se usan correctamente las operaciones de sincronización, es posible que el programa se bloquee (*deadlock*).

Por ejemplo, he aquí un par de errores típicos usando secciones críticas construidas mediante cerrojos:



(región paralela)

```
...  
omp_set_lock(&C1);  
A = A + func1();  
if (A>0) omp_unset_lock(&C1);  
else  
{  
    ...  
}  
...
```

(región paralela con secciones)

```
...  
#pragma omp section  
{ omp_set_lock(&C1);  
  A = A + func1();  
  omp_set_lock(&C2);  
  B = B * A;  
  omp_unset_lock(&C2);  
  omp_unset_lock(&C1);  
}  
  
#pragma omp section  
{ omp_set_lock(&C2);  
  B = B + func2();  
  omp_set_lock(&C1);  
  A = A * B;  
  omp_unset_lock(&C1);  
  omp_unset_lock(&C2);  
}  
...
```





Recomendaciones:

- prestar atención al ámbito de las variables: shared, private, etc.
- utilizar con cuidado las funciones de sincronización.
- disponer de una versión equivalente secuencial para comparar resultados (serán siempre iguales?).





Llamadas en paralelo a funciones de librería

¿habrá problemas con la activación simultánea de más de una instancia de dichas funciones?

Una librería es *thread-safe* (re-entrante) si lo anterior no es un problema. Si no es así, habría que utilizar una secuencia tipo:

LOCK / CALL / UNLOCK



Speed-up

El objetivo de programar una aplicación en un sistema paralelo es:

- ejecutar el problema **más rápido**.
- ejecutar un problema de **mayor tamaño**.

En ambos casos hay que tener en cuenta el ***overhead*** añadido al paralelizar el código.

- ▶ Escribir programas paralelos OpenMP es fácil... y también lo es escribir programas de bajo rendimiento.
- ▶ Principales fuentes de pérdida de eficiencia
 - el algoritmo en ejecución: **Amdahl** // **desequilibrio de carga**
 - sincronización: **grano** muy fino
 - comunicación: acceso a variables **shared**, **cache** (fallos, falsa compartición...)
 - implementación de OpenMP / S.O.

Ejemplos de mejora de la eficiencia:

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

XMED = xnorm / sum;

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

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

    #pragma omp single
        XMED = xnorm / sum;

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

```
#pragma omp parallel
    private (XMED)
{
    #pragma omp for nowait
    for (i=0;i<N;i++) { ... }

    XMED = xnorm / sum;

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

una sola sección paralela

cálculo repetido

Ejemplos de mejora de la eficiencia

Utilizar el mismo reparto (**¡los mismos datos!**) para todos los bucles (hacer a “mano” el *work sharing*).

```
...  
#pragma omp parallel ...  
{ ...  
    ini = pid*N/npr;  
    fin = (pid+1)*N/npr;  
  
    for (i=ini; i<fin; i++) { ... }  
    ...  
    for (i=ini; i<fin; i++) { ... }  
    ...  
}
```

- ▶ Por ello, interesa minimizar costes de creación y terminación de *threads*, sincronización, etc.

Hay que considerar también los problemas de **localidad de los datos**.

Diferentes aspectos:



- **La máquina: SMP o DSM**

La localidad de los datos es un factor determinante en una máquina DSM.

También es importante en el caso SMP, para mejorar la tasa de acierto de la cache.

- **La compartición falsa de datos**

Aquí puede resultar importante el tipo de *scheduling* que se efectúe, para evitar continuas anulaciones de bloques de cache.





En resumen, es necesario un cuidadoso análisis del reparto y uso de los datos, el tamaño de grano, la sincronización de los *threads*, el uso de memoria, etc., para minimizar *overheads* (p.e., el uso de barreras implícitas) y conseguir el máximo rendimiento de un sistema paralelo.

Existen herramientas que ayudan en estas tareas:
KAI, PORLAND...



OpenMP: referencias

TEXTOS

- **R. Chandra et al.**
Parallel Programming in OpenMP
Morgan Kaufmann, 2001.
- **B. Chapman et al.**
Using OpenMP
The MIT Press, 2008.

WEB

- **www.openmp.org**
(especificación, software...)

