

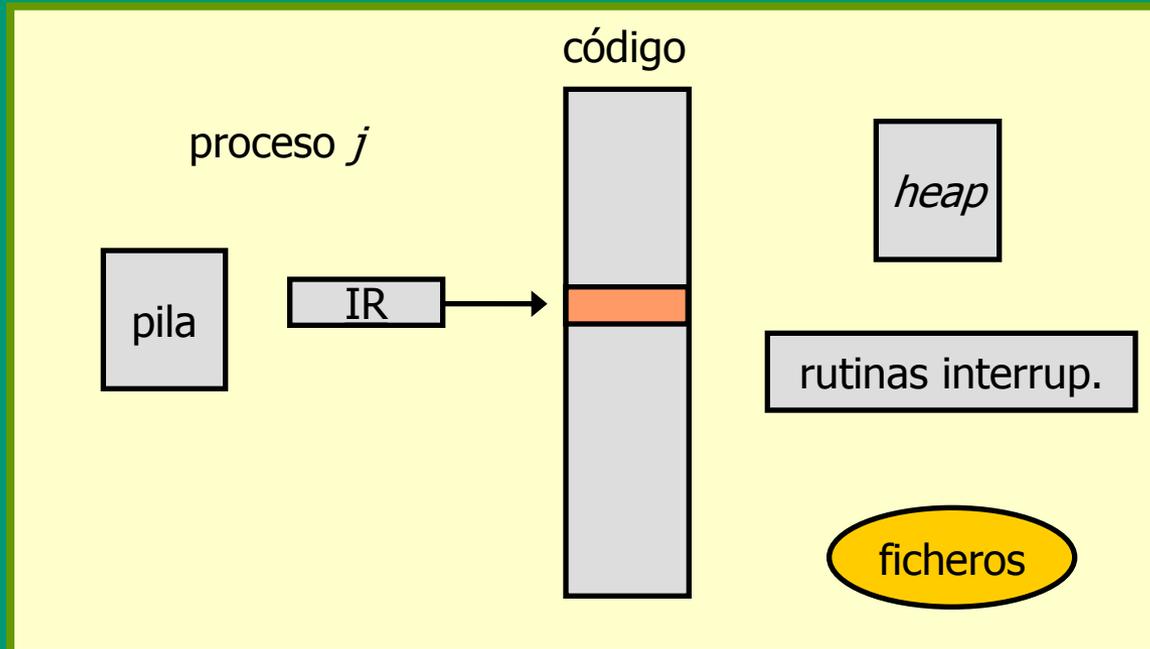
# Programación de Sistemas de Memoria Compartida

- ▶ 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.

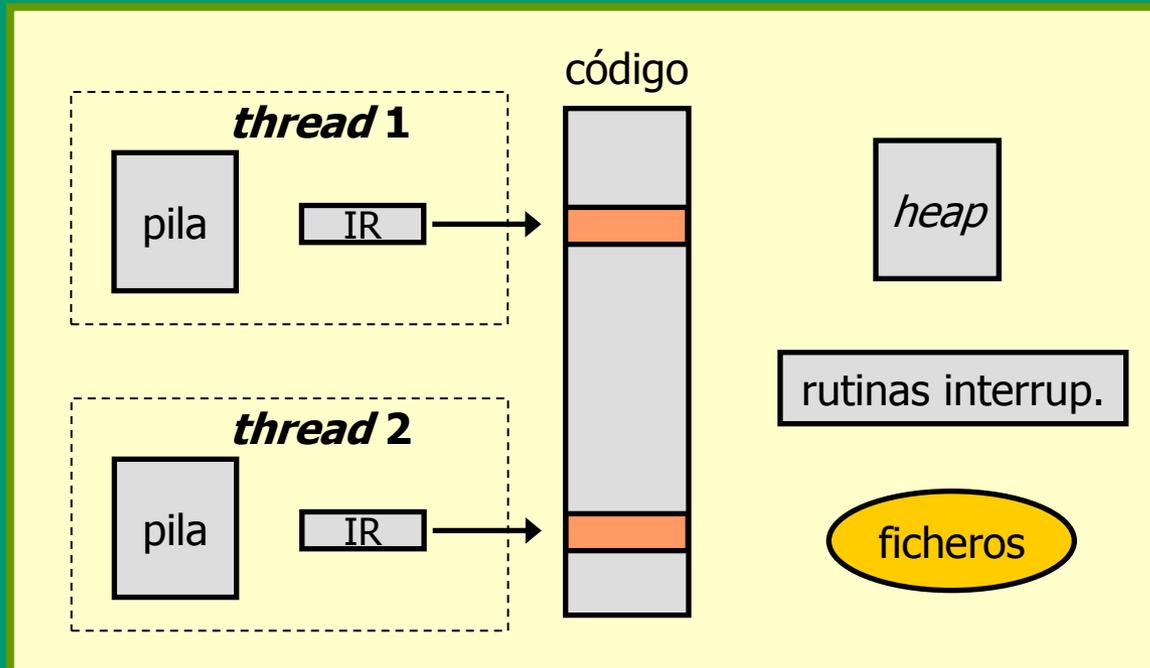
► Alternativas:

- 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...



- ▶ procesos concurrentes: Fork / Join
- ▶ gran *overhead*, son muy "caros"



- ▶ los *threads* son más "ligeros"
- ▶ la sincronización (variables compartidas) es sencilla
- ▶ habitual en el S.O. (solaris, windows XP...)

- ▶ 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*.

- ▶ Qué **no** esperamos de una “herramienta” para programar aplicaciones paralelas:
  - un analizador de dependencias entre tareas (queda en manos del programador).

# Introducción

- ▶ **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...

# Introducción

- ▶ 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**

# Introducción

- ▶ 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.

# Introducción

- ▶ Los lenguajes base con los que trabaja OpenMP son Fortran y C/C++.
- ▶ Las directivas de OpenMP se especifican de la siguiente manera:

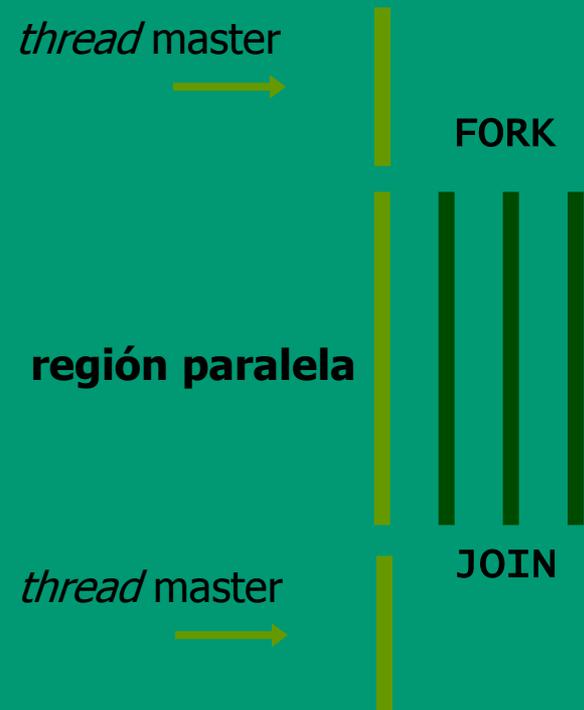
para C:            `#pragma omp <directiva>`

para Fortran      `!$omp <directiva>`

# Introducción

- ▶ 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.



# Introducción

- ▶ 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*).

# Ejemplo

```
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);
    }
}
```

# Introducción

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

# Introducción

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

# Introducción

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).

# Introducción

- ▶ 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*.

# Regiones paralelas

- ▶ 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**.

# Regiones paralelas

- ▶ 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)`

# Regiones paralelas

## ► ¿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.

# Regiones paralelas

> 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



# Regiones paralelas

- ▶ 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*.

# Regiones paralelas

- ▶ 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**.

## R.P.: Cláusulas de ámbito

- ▶ 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.

# R.P.: Cláusulas de ámbito

- ▶ 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:

# R.P.: Cláusulas de ámbito

- **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!).

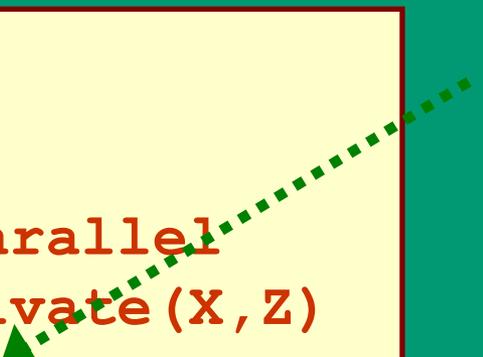
Se destruyen al finalizar la ejecución de los *threads*.

# R.P.: Cláusulas de ámbito

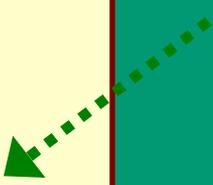
> Ejemplo:

```
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



## R.P.: Cláusulas de ámbito

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.

# R.P.: Cláusulas de ámbito

- `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**.

# R.P.: Cláusulas de ámbito

> 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)**

# R.P.: Cláusulas de ámbito

- **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**.

# R.P.: Cláusulas de ámbito

> 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).

# R.P.: Cláusulas de ámbito

- **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)

# R.P.: Cláusulas de ámbito

- 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)
```

## R.P.: Cláusulas de ámbito

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.

# R.P.: Cláusulas de ámbito

- **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.

# R.P.: Otras cláusulas

- **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).

# R.P.: Otras cláusulas

- **num\_threads (expresión)**

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

Precedencia: vble. entorno >> función >> cláusula

# R.P.: Resumen cláusulas

## ▶ Cláusulas de la región paralela

- **shared, private, firstprivate** (var)

**reduction** (op:var)

**default** (shared/none)

**copyin** (var)

-**if** (expresión)

-**num\_threads** (expresión)

# Reparto de tareas

## ► 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;
}
```



# Reparto de tareas

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;
}
```

# Reparto de tareas

- 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*).

# Reparto de tareas

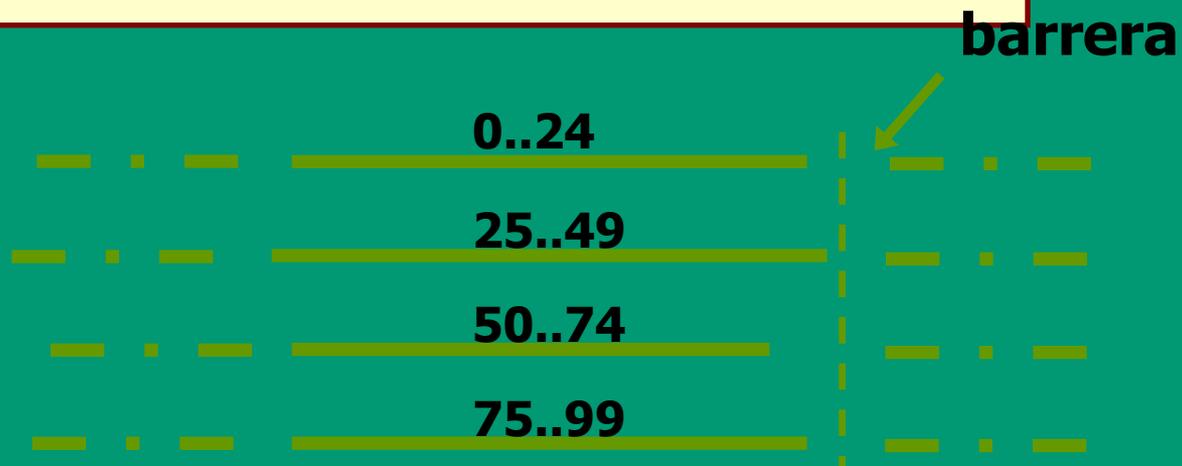
- ▶ Las opciones de que disponemos son:
  1. Directiva **for**, para repartir la ejecución de las iteraciones de un bucle entre todos los *threads* (bloques básicos y número de iteraciones conocido).
  2. Directiva **sections**, para definir trozos o secciones de una región paralela a repartir entre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

# Reparto de tareas (for)

## 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



# Reparto de tareas (for)

- ▶ 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.

# Reparto de tareas (for)

- ▶ 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.

# Reparto de tareas (for)

```
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.

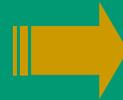
La directiva **parallel for** implica la generación de *P threads*, que se repartirán la ejecución del bucle.

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).

# Reparto de tareas (for)

```
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;  
  }
```



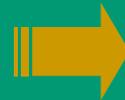
```
#pragma omp parallel for  
  private (j,X)  
for (i=0; i<N; i++)  
  for (j=0; j<M; j++)  
  {  
    X = B[i][j] * B[i][j];  
    A[i][j] = A[i][j] + X;  
    C[i][j] = X * 2 + 1;  
  }
```

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**.

# Reparto de tareas (for)

```
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**.

# Cláusulas (`for`)

- ▶ 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.

# Cláusulas de ámbito

- ▶ 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.

# Cláusula `schedule`

- ▶ ¿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**.

# Cláusula `schedule`

- ▶ 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*).

# Cláusula schedule

- **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).

# Cláusula `schedule`

- **`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.

# Cláusula `schedule`

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

Se permite a las implementaciones añadir nuevos métodos de planificación.

## **RECUERDA:**

**estático**      menos coste / mejor localidad datos

**dinámico**    más coste / carga más equilibrada

# Cláusula collapse

- **collapse (n)**

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

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

# Cláusula `nowait`

- ▶ 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.

# Cláusulas

- ▶ 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;
```

# Resumen

## ► 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.]`

# Rep. de tareas: funciones

## 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.

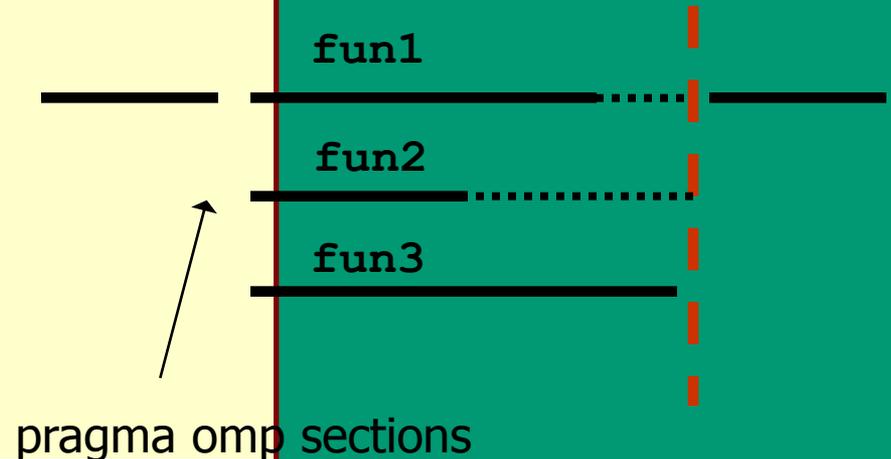
Una barrera implícita sincroniza el final de las secciones o tareas.

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

# Rep. de tareas (sections)

> Ejemplo:

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



# Rep. de tareas (sections)

- ▶ 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)

# Rep. de tareas (`single`)

## 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.

# Rep. de tareas (*single*)

- ▶ 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**.

# Rep. de tareas (single)

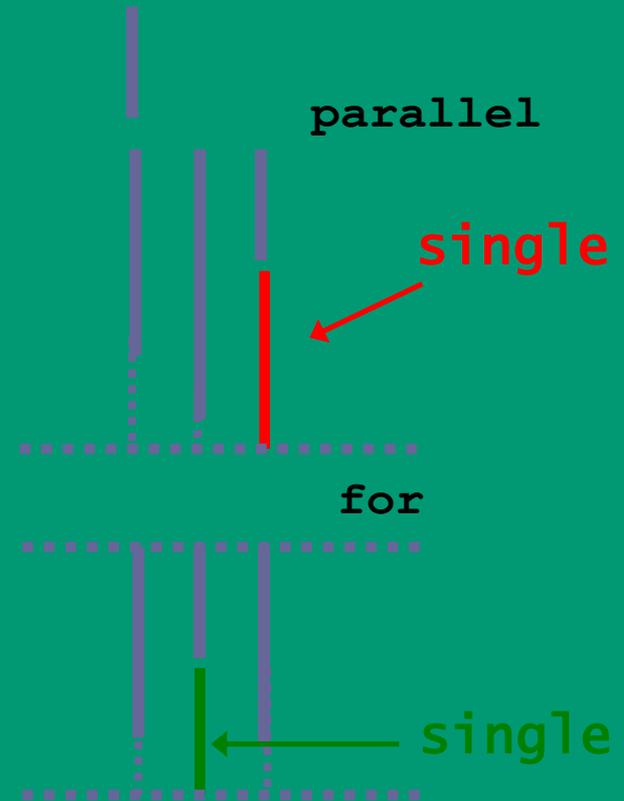
## > Ejemplo:

```
#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) ;
}
```



# Reparto de tareas

## ► 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.

# Rep. de tareas “orphaned”

- ▶ 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.

# Reg. paralelas anidadas

- ▶ 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);
```

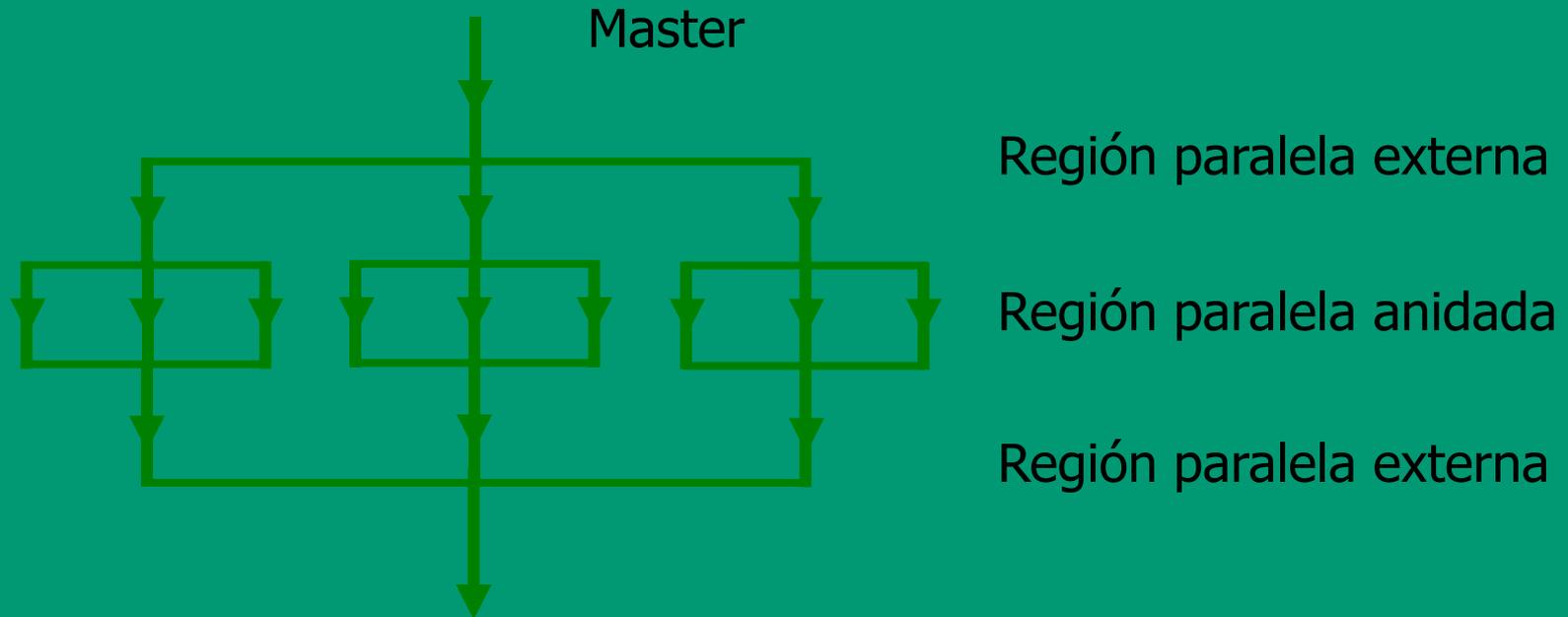
- una variable de entorno

- ```
> export OMP_NESTED=TRUE
```

Una función devuelve el estado de dicha opción:

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

# Reg. paralelas anidadas



Se pueden anidar regiones paralelas con cualquier nivel de profundidad.

# Reg. paralelas anidadas

- ▶ OpenMP 3.0 mejora el soporte al paralelismo anidado:
  - 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.

# Reg. paralelas anidadas

- ▶ OpenMP 3.0 mejora el soporte al paralelismo anidado:
  - 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
```

# Algunas funciones más

- ▶ 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*.

# Algunas funciones más

- ▶ 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();
```

# Algunas funciones más

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

```
> export OMP_STACKSIZE tamaño [B|K|M|G]
```

y para gestionar la política de espera en la sincronización:

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