Grado en Ingeniería Informática

Tema 4 (Parte2): Introdución a OpenMP

Asignatura: Arquitectura de Ordenadores

Profesores:

G131: Iván González Martínez

G130 y 136: Francisco Javier Gómez Arribas

Departamento de Tecnología Electrónica y de las Comunicaciones

UNIVERSIDAD AUTONOMA DE MADRID

Contenidos

- **Modelos de programación**
  - multiprocesadores de memoria compartida
  - multiprocesadores de memoria distribuida
- **OpenMP**
  - Objetivos y Funcionamiento
  - Consideraciones de rendimiento.
  - Directivas y ejemplos

Modelos de programación

- **PARALELISMO: Posibilidad de ejecutar varias acciones simúltaneamente con el objetivo de incrementar el trabajo realizado y disminuir el tiempo de ejecución.**
  - A nivel de programas
  - A nivel de subrutinas

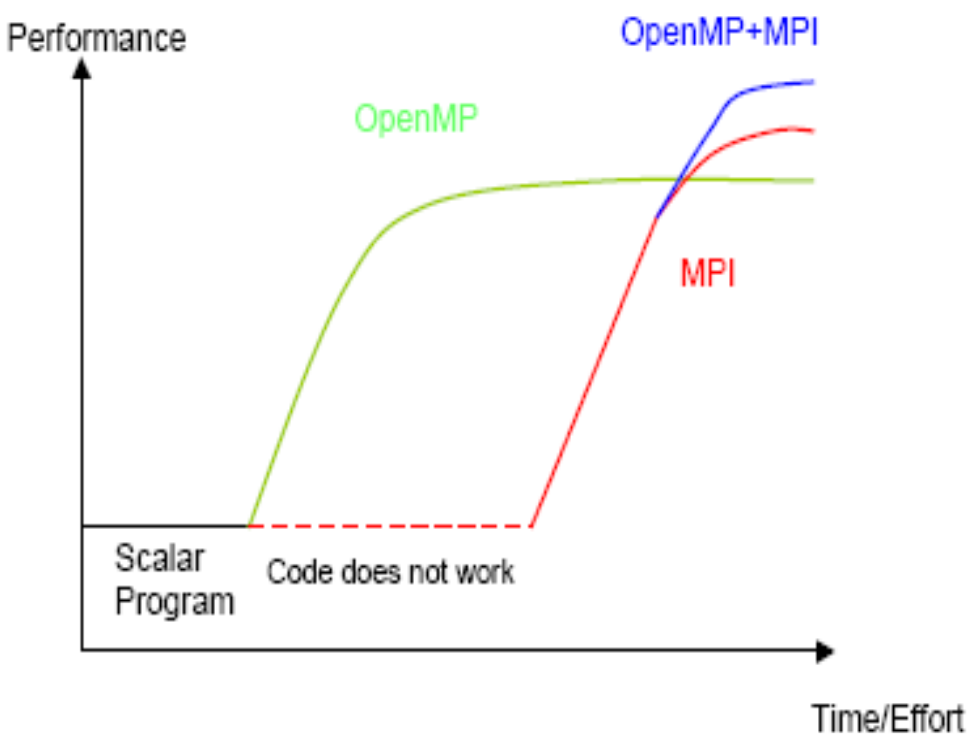
  → PARALELISMO GRANO GRUESO
  - A nivel de bucles
  - A nivel de sentencias

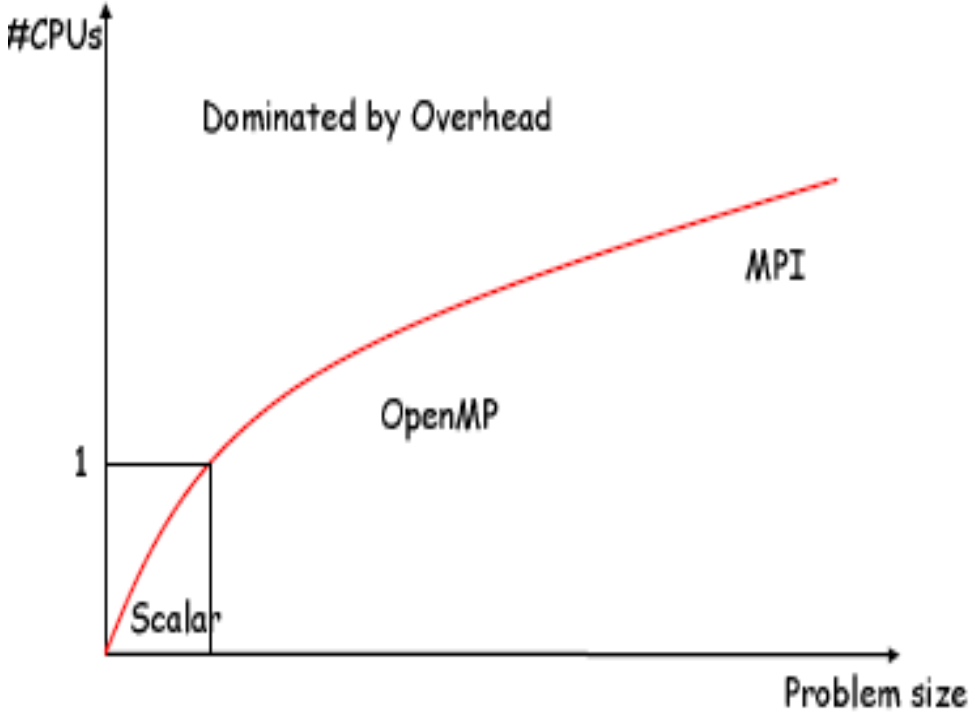
  → PARALELISMO GRANO FINO

- **Modelo de Programación para explotar el paralelismo**
  - **Programación en Memoria Compartida**: Explota el paralelismo de grano fino.
    - HPC
    - OpenMP,
    - pthreads,…
  - **Programación de Paso de Mensajes para memoria distribuida**: Explota el paralelismo grano grueso
    - MPI

Modelos de programación

OpenMP versus MPI

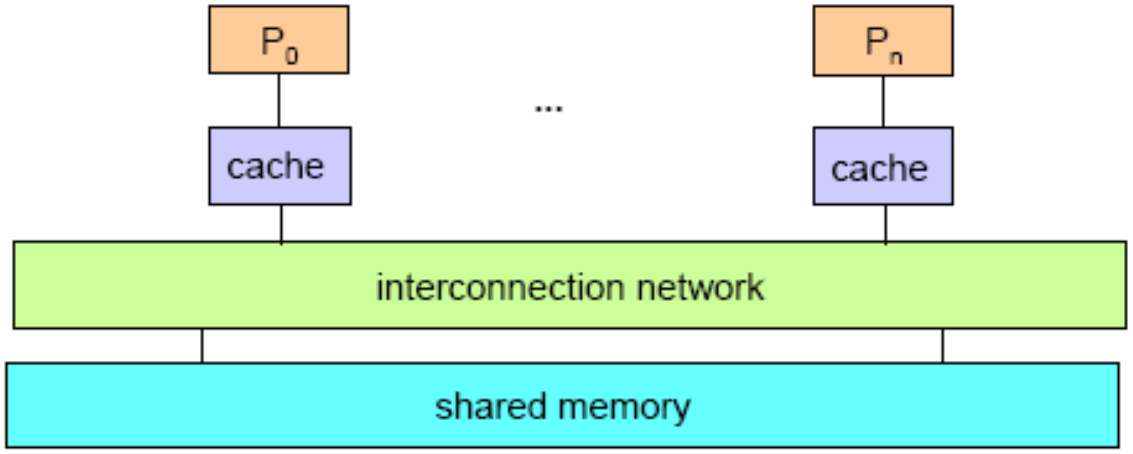




Compilador Paralelo Open MP

Se usa en la programación de computadores paralelo con un espacio de direcciones compartido.

- **OpenMP define una API para programar con C/C++ y Fortran**
- **En la API se definen directivas, funciones, y variables de entorno.**
- **No realiza una paralelización automática.**



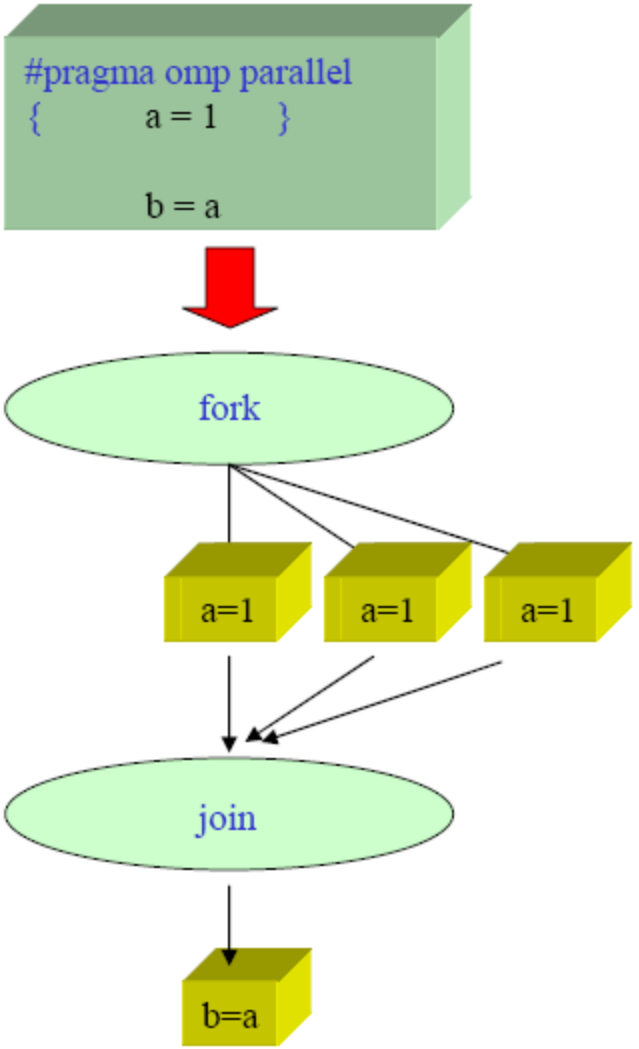
Objetivos de OpenMP

- Establecerse como un estándar para los distintos tipos de arquitecturas o plataformas de memoria compartida
- Establecer un conjunto de directivas simple y limitado para la programación de máquinas basadas en memoria compartida
- Fácil de usar
  - **Permite paralelizar incrementalmente un programa serie**
  - **Permite implementar paralelismo de grano grueso y grano fino**
- Portable
  - **Soporta Fortran (77, 90, and 95), C, and C++**
  - **Es posible ser miembro o participar en la definición del API**

Más información en http://www.openmp.org/

Modelo de Ejecución

- **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.
- **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 Master thread.**



Efectos por compartir memoria.

Al ser OpenMP un modelo de memoria compartida.

- **Los threads se comunican utilizando variables compartidas.**
- **El rendimiento puede disminuir debido al propio hecho de compartir**

Los *threads* al compartir la memoria disponible en los *procesadorres/cores* en los que se están ejecutando provocan:

- **Conflictos de almacenamiento.**
- **Incluso llegando a guardar valores incorrectos por data races.**
- **Fallos de cache.**

La arquitectura de los procesadores determinará si los *threads* comparten o tienen acceso a caches separadas. Compartir caches entre dos cores puede suponer reducir a la mitad el tamaño de cache disponible para cada *thread,* mientras que caches separadas puede hacer que cuando se compartan datos comunes se realice de manera menos eficiente.

Efectos debidos a compartir variables

- **Carreras críticas o "Data races"**
  - **Cuando uno o más *threads* acceden a la misma posición de memoria para actualizarla. Aparte de la necesidad de evitar el conflicto en memoria, y debido a que la planificación de *threads* lanzados en paralelo no garantiza ningún orden, el resultado final almacenado puede variar de una ejecución a otra del mismo programa.**
- **False Sharing.**
  - **Esta situación se produce cuando los *threads*, aunque no estén accediendo a las mismas variables, comparten un bloque cache que contiene las diferentes variables. Debido a los protocolos de coherencia cache, cuando un *thread* actualiza una variable en el bloque cache y otro *thread* quiere acceder a una variable diferente pero que está en el mismo bloque, este bloque antes de utilizarse se escribe en memoria. Cuando dos o mas *threads* actualizan repetidamente variables del mismo bloque, este bloque puede ir y volver a memoria por cada actualización.**

Para controlar las carreras críticas :

- **Uso de sincronización para protegerse de los conflictos de datos.**
- **Mejor es modificar cómo se almacenan los datos se puede minimizar la necesidad de sincronización….**

Consideraciones de Rendimiento en OpenMP

Granularidad

- **No se debe paralelizar un bucle o región a menos que tome mucho más tiempo en ejecutarse que la ejecución paralela más la sobrecarga de paralelización.**
- **Hay que tener en cuenta que a esta sobrecarga hay que añadir el barrier implícito y el control de la caché y sincronización.**

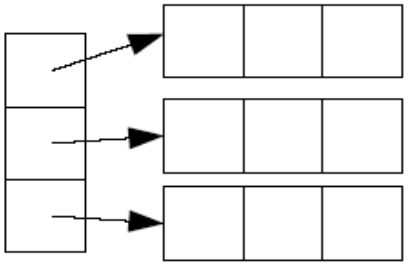
Equilibrio de carga

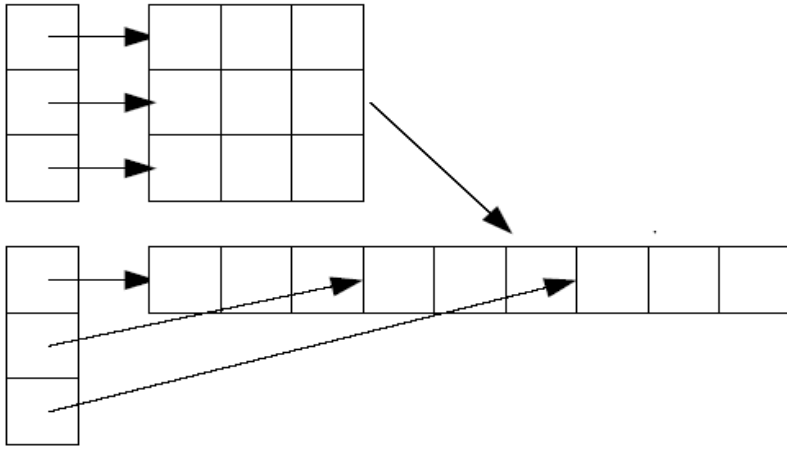
- **Un bucle paralelo (asumiendo que no ha sido indicado nowait) no acabará hasta que el último hilo complete sus iteraciones.**
- **OpenMP ofrece esquemas dinámicos que causando muy poca sobrecarga por equilibrado de carga, producen un descenso de tiempo de procesamiento en bucles cuyas iteraciones requieren tiempos de ejecución dispares.**

Consideraciones de Rendimiento en OpenMP (2)

Principio de Localidad

- **La localidad es aprovechada por las cachés**
- **La memoria reservada para un vector es contigua en memoria, mientras que la memoria reservada para los vectores que forman una matriz no necesariamente lo es.**
- **Es mejor reservar una matriz como un vector, que reservar una matriz como una serie de vectores indexados.**





Consideraciones de Rendimiento en OpenMP (3)

Principio de Localidad

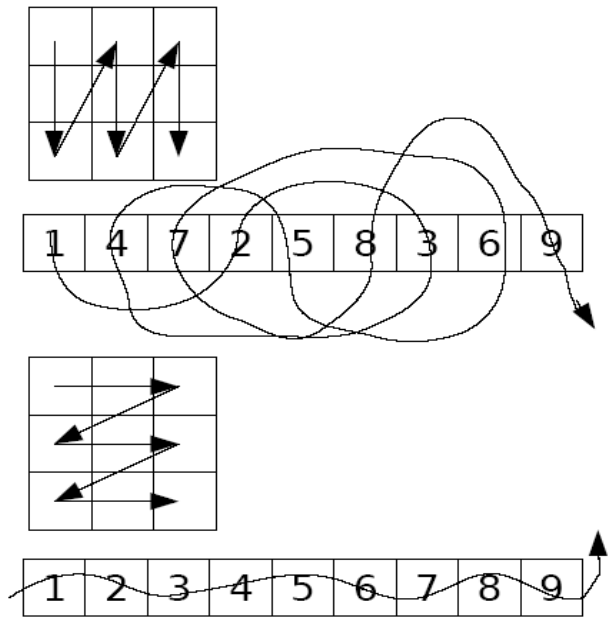
También es importante acceder a los elementos en memoria siguiendo el mismo orden en el que se encuentran en esta.

En lugar de acceder a una matriz así:

```
int Matriz[1600*1600];
for (x = 0;x < 1600;x++)
  for (y = 0;y < 1600;y++)
    aux = Matriz[x+y*1600];
```

Hacerlo así:

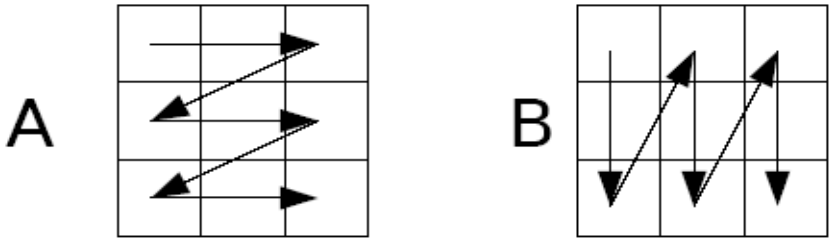
```
int Matriz[1600*1600];
for (y = 0;y < 1600;y++)
  for (x = 0;x < 1600;x++)
    aux = Matriz[x+y*1600];
```



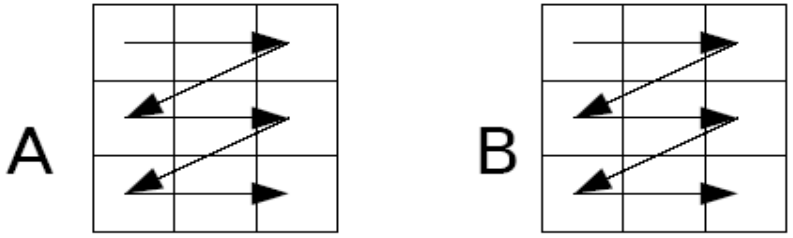
Consideraciones de Rendimiento en OpenMP (4)

Principio de Localidad

Si se necesita acceder a una matriz en un orden que inhibe la localidad (por ejemplo, a la matriz B en una multiplicación de matrices AxB):



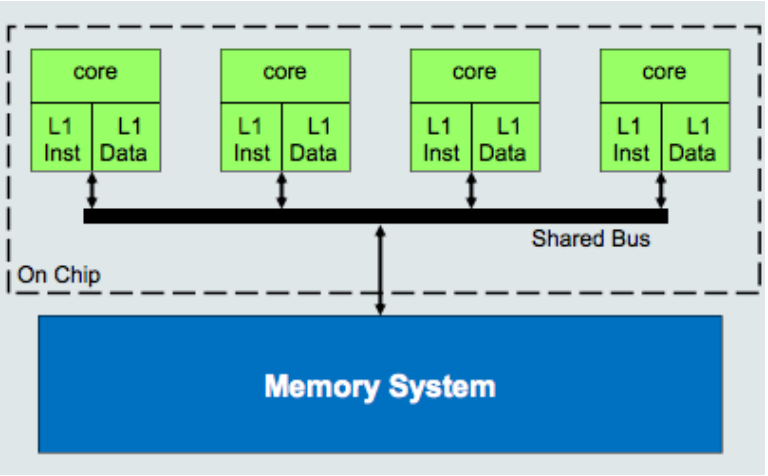
Mejora si se organiza la matriz (para la multiplicación, se puede almacenar la traspuesta de la matriz B y acceder a ella en el otro orden:



Consideraciones de Rendimiento en OpenMP (5)

Localidad mal gestionada

En los sistemas multiprocesador, cuando un procesador modifica unos datos, estos se copian a su caché y esa línea se marca como sucia en las cachés de los demás procesadores, para que dos procesadores o más no puedan estar trabajando sobre versiones distintas de los mismos datos.

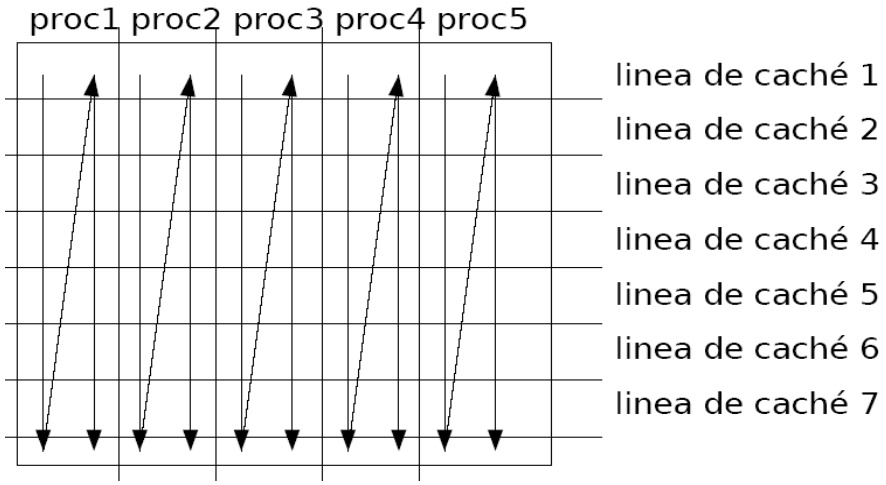


Si distintos procesos intentan modificar la mismas direcciones de memoria en repetidas ocasiones, se puede producir hasta un fallo de caché por cada acceso a esa posición de memoria.

Consideraciones de Rendimiento en OpenMP (6)

Localidad mal gestionada

A ese comportamiento se lo conoce como false sharing (falso compartir) y se da principalmente en casos del tipo matriz dividida por columnas:



Caso de que se itere en orden contrario a la disposición de la memoria

Consideraciones de Rendimiento en OpenMP (7)

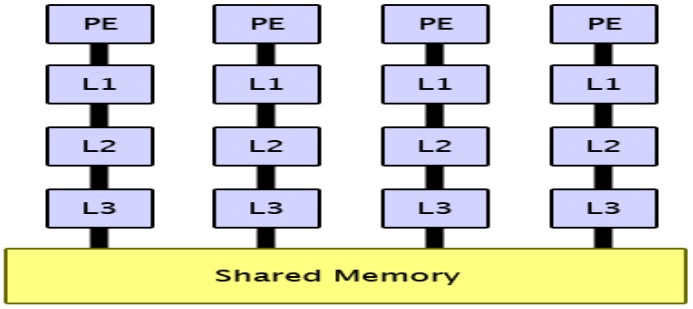
Sincronización

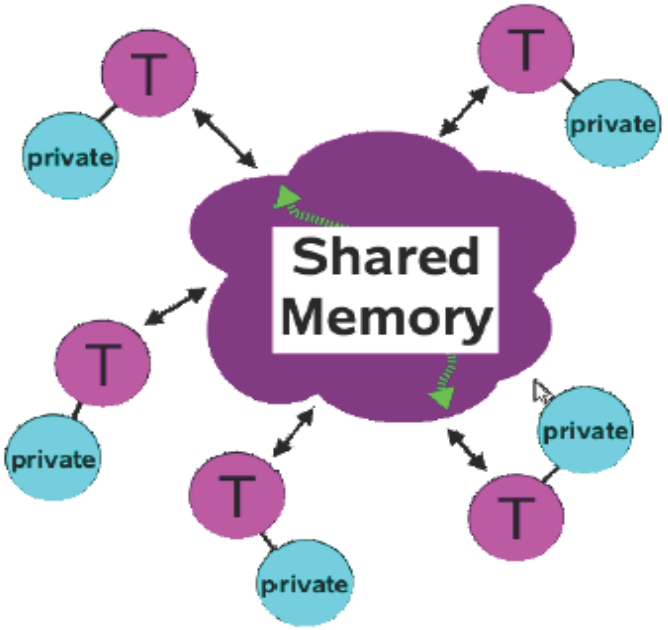
- **Identificar bucles cuyas iteraciones pueden ser entrelazadas y unirlos.**
- **La exclusión mutua, está implementada en OpenMP, pero es muy costosa.**
- **Los problemas surgen cuando varios procesos entran a una sección crítica varias veces por iteración.**
- **En lugar de que varios procesos modifiquen variables globales, es recomendable que cada uno de ellos mantenga una variable local y al final se haga un reducción.**

Características de OpenMP

- OpenMP es una API que permite definir explícitamente paralelismo multi-thread en sistemas de memoria compartida
- OpenMP esta divido en tres componentes principales:
  - **Directivas de compilación**
  - **Librería de rutinas (Runtime Library)**
  - **Variables de entorno**

Características de OpenMP





- El acceso a la memoria es compartido por todos los procesadores/cores
- Cada thread puede tener datos compartidos y/o privados
  - **Los datos compartidos son comunes a todos los threads**
  - **Los datos privados solo los procesa el thread propietario**
- La transferencia de datos es transparente al programador
- Se producen sincronizaciones (implícitas)

Características NO soportadas por OpenMP

- OpenMP :
  - **NO Soporta sistemas de memoria distribuida (por sí mismo)**
  - **NO Esta necesariamente implementado de la misma forma por todos los fabricantes**
  - **NO Garantiza el mejor uso posible de la memoria compartida**
  - **NO Comprueba si hay dependencia o conflictos de datos, dependencias de ejecución (race conditions) o situaciones de bloqueo**
  - **NO Realiza paralelización automática ni aporta directivas para ayudar al compilador a realizar paralelización automática**
  - **NO Garantiza que la entrada/salida a un mismo fichero sea síncrona cuando se ejecutan en paralelo. El programador es responsable de esta sincronización**

Componentes

Directivas

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

Variables de Entorno

- **Nº threads**
- **Tipo scheduling**
- **Ajuste dinámico threads**
- **Paralelismo anidado**

Runtime API

- **Nº threads**
- **ID thread**
- **Tipo scheduling**
- **Ajuste dinámico threads**
- **Paralelismo anidado**

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)
```

Formato de las directivas

Directivas: *Las directivas poseen un comando principal y pueden 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,n) reduction(*:alfa)
for (i=0; i<n; i++){
   alfa= alfa * y[i];
 }
```

Son vistas como comentarios por otros compiladores

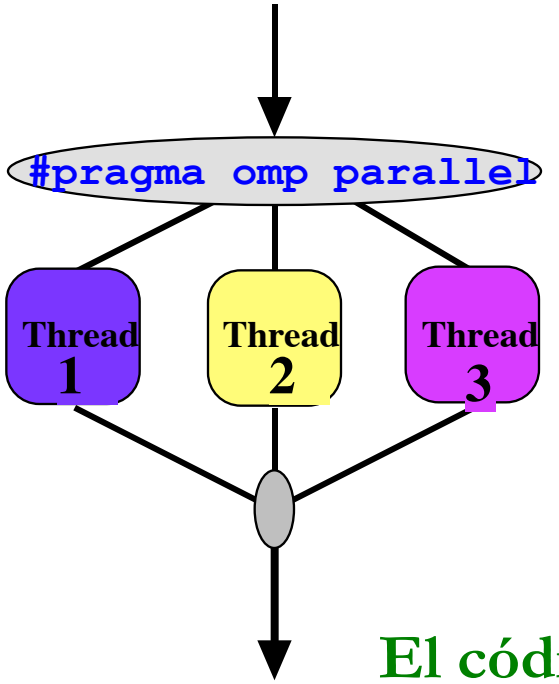
Cuando hay conflictos entre directivas, prevalece la última.

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**

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 código que se define en una región paralela debe ser un bloque básico.

Regiones paralelas

- Cómo controlar el número de *threads* que se generan para ejecutar una región paralela:

  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)
     ```

¿Cuántos Threads puedo/ debo lanzar?

Regiones paralelas

- Gestión de Threads: ¿Qué Thread soy ? ¿Cuántos hay?
  - **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.

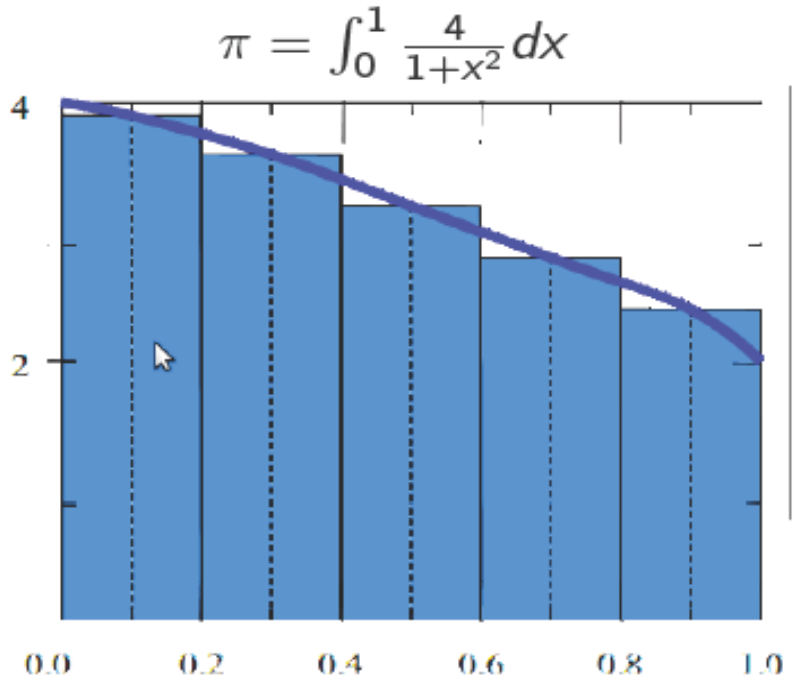
Ejemplo de región paralela

Un ejemplo de la directiva omp parallel:

```
...
#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) numthreads(4)
 { 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

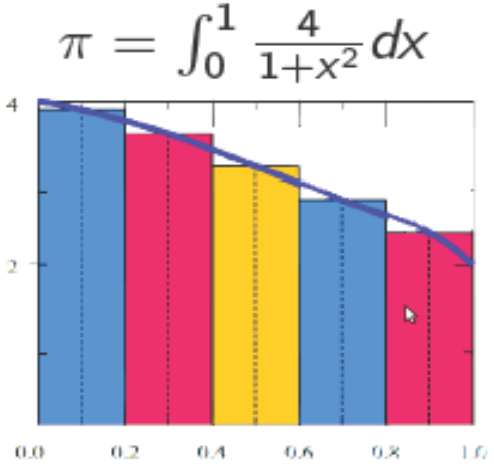
Ejemplo: Cálculo de PI
Programa secuencial a paralelizar con OpenMP




```
h = 1.0 / (double)n;
double x;

for(i=1; i<=n; i++) {
  x = h * ((double)i - 0.5);
  sum += 4.0 / (1.0 + x*x);
}
pi = h * sum;
```

Ejemplos OpenMP: Tutorial sobre OpenMP de Vicente Blanco, Universidad de La Laguna, presentado en ANACAP'09

Ejemplo: Cálculo de PI
Programa OpenMP con problemas de rendimiento

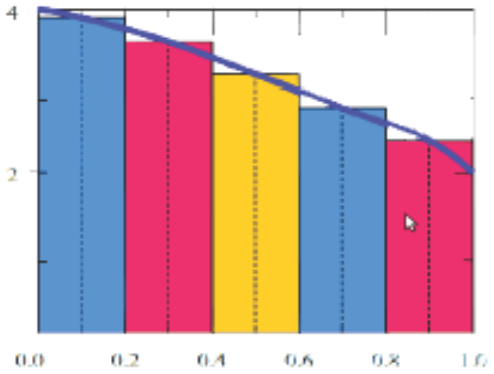


False sharing !!

```
h = 1.0 / (double)n;
#pragma omp parallel
{
  double x;
  int i, tid;
  tid = omp_get_thread_num();
  sum[tid] = 0.0;
  for(i = tid + 1; i <= n; i += numThreads) {
    x = h * ((double) i - 0.5);
    sum[tid] += 4.0 / (1.0 + x*x);
  }
}
pi = 0.0;
for(i = 0; i < numThreads; i++)
  pi += sum[i];
pi = h * pi;
```

Ejemplo: Cálculo de PI
Programa OpenMP con buen rendimiento

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$




```
#pragma omp parallel
{
  double x, priv_sum;
  int i, tid;
  tid = omp_get_thread_num ();
  priv_sum = 0.0;
  for(i = tid + 1; i <= n; i+=numThreads) {
    x = h * ((double) i - 0.5);
    priv_sum += 4.0 / (1.0 + x*x);
  }
  sum[tid] = priv_sum;
}
pi = 0.0;
for (i = 0; i < numThreads; i++)
  pi += sum[i];
pi = h * pi;
```

Regiones Paralelas: 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 como *stack frame* para las rutinas invocadas por el *thread*.**
- **Como cada *thread* utiliza su propia pila, las variables declaradas en la propia región paralela (o en una rutina) son privadas.**

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*.**
- **Será necesario que algunas variables sean propias de cada *thread*, privadas.**
- **Por defecto, las variables son shared.**

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

Regiones Paralelas: 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.

- **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 paralela.**

Cláusulas de la región paralela

```
shared, private, firstprivate(var)
default(shared/none)
reduction(op:var)
copyin(var)
```

- `if (expresión)`
- `num_threads(expresión)`

Regiones Paralelas: 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.

`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).

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

Región Paralela: Cláusulas de ámbito

Las variables privadas no están inicializadas al comienzo ,ni dejan rastro al final. Si se necesita existen las clausulas:

- `firstprivate( )`

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

  Es privada al thread pero se inicializa con el valor de la variable del mismo nombre en el thread master.

Regiones Paralelas: 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)

Regiones Paralelas: Cláusulas de ámbito

- `reduction( )`

  Las operaciones de reducción utilizan variables a las que acceden todos los procesos y sobre las que se efectúa alguna operación de "acumulación" en modo atómico.

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

  E control de la operación se deja en manos de OpenMP, si se declaran la variable de tipo reduction.

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

La propia cláusula indica el operador de reducción a utilizar.

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

Regiones Paralelas: 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)
```

Regiones Paralelas: 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.

threadprivate: la variable es global, pero es privada en cada región paralela en tiempo de ejecución.

La diferencia entre *threadprivate* y *private* es el ámbito global asociado a threadprivate y por tanto el valor es preservado entre regiones paralelas.

Regiones Paralelas: Cláusulas de ámbito

```
#include <omp.h>

int  a, b, i, tid;
float x;

#pragma omp threadprivate(a, x)

main(int argc, char *argv[]) {

  /* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);

  printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
  {
  tid = omp_get_thread_num();
  a = tid;
  b = tid;
  x = 1.1 * tid +1.0;
  printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }  /* end of parallel region */

  printf("************************************\n");
  printf("Master thread doing serial work here\n");
  printf("************************************\n");

  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  {
  tid = omp_get_thread_num();
  printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }  /* end of parallel region */

}
```

```
Output:

1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 2:   a,b,x= 2 2 3.200000
Thread 3:   a,b,x= 3 3 4.300000
Thread 1:   a,b,x= 1 1 2.100000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 3:   a,b,x= 3 0 4.300000
Thread 1:   a,b,x= 1 0 2.100000
Thread 2:   a,b,x= 2 0 3.200000
```

Regiones Paralelas: Cláusulas de ámbito

Para variables declaradas como threadprivate, un thread no puede acceder a la copia threadprivate de otro thread (por se privadas).

`copyin(X)`

Similar a firstprivate para variables `private`.

La cláusula `copyin` permite copiar (al comienzo de la región paralela) en cada *thread* el valor de la variable con el mismo nombre en el *thread* máster.

Las variables globales *threadprivate* no estan inicializadas a no ser que se use copyin para copiar el valor de la variable global del mismo nombre.

Regiones Paralelas: Cláusulas de ámbito

- `if (expresión)`

  La región paralela 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 utillizar en la región paralela.

  Precedencia: vble. entorno >> función >> cláusula

Directiva de la Región Paralela: omp parallel

Resumiendo: La directiva omp parallel es la construcción paralela fundamental de OpenMP

Admite las siguientes clausulas y opciones:

```
#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
```

Paralelización Guiada con Reparto Implicito o Explicito

Construcciones de trabajo compartido para repartir iteracciones/tareas.

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`

  Reparte la ejecución de las iteraciones de un bucle entre todos los threads de la región paralela (bloques básicos y número de iteraciones conocido).

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

- Construcción `sections`

  - Define una construcción no iterativa formada por regiones que deben dividirse entre los threads definidos

  ```
  #pragma omp sections [clausulas ...]
    { [#pragma omp section <cr>]
      bloque estructurado
      [#pragma omp section <cr>]
      bloque estructurado
    }
  ```

Reparto de tareas (`for`)

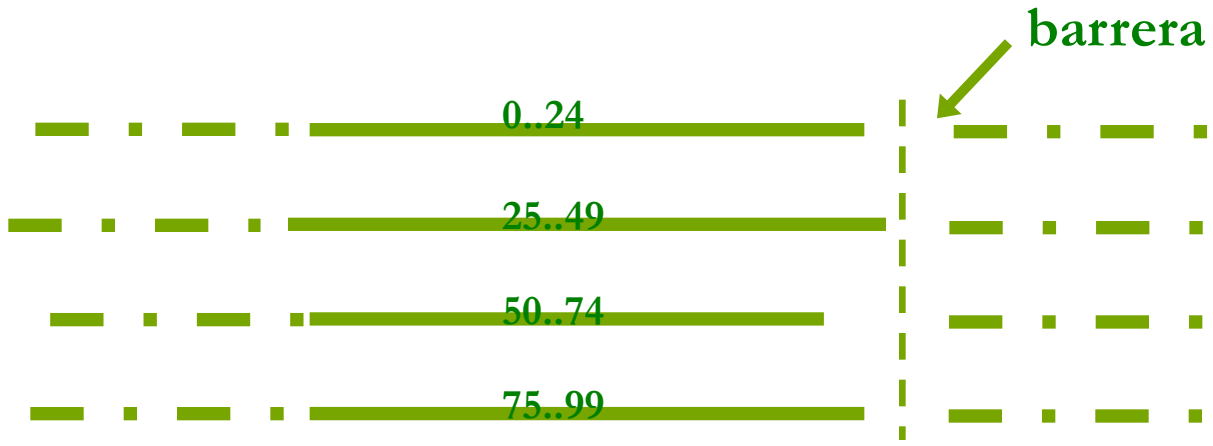
1 Directiva `for`

```
#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.

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.

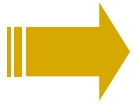
La directiva **`parallel for`** implica la generación de P *threads*, que se repartirán la ejecución del bucle.

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;
 }
```



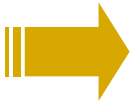
```
#pragma omp parallel for
   private (j,X)

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.

Construcción "Work-Sharing": Directiva *for*

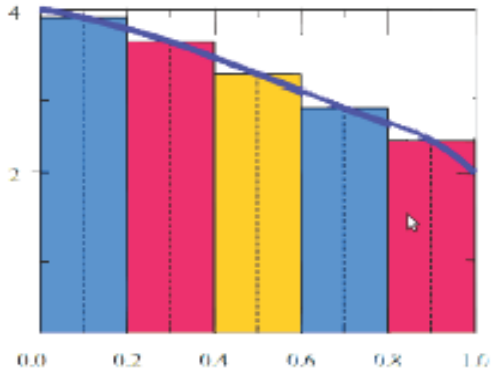
La directiva *for* especifica que las iteraciones del bucle deben ser ejecutadas en paralelo por el equipo de threads

- **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
```

Ejemplo: Cálculo de PI
Reparto de tareas con la directiva for

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$




```
h = 1.0 / (double)n;
omp_set_num_threads(numThreads);
#pragma omp parallel
{
  double x; int id;
  id = omp_get_thread_num();
  sum[id] = 0.0;
  #pragma omp for
    for(i = 1; i <= n; i++) {
      x = h * ((double)i - 0.5);
      sum[id] += 4.0 / (1.0 + x*x);
    }
}
pi = 0.0;
for(i = 0; i < numThreads; i++)
  pi += sum[i];
pi = h * pi;
```

Cláusulas de ámbito en la directiva for

- 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.

Es privada al thread. Si la iteración es la última de un bucle se copia a la variable del mismo nombre en el thread master.

Una variable puede ser `firstprivate()` y `lastprivate()`

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.**

- La sintaxis de la cláusula es:

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

  Los tipos pueden ser: static, dinamic, guided, runtime,auto

Cláusula `schedule: tipos`

- **`static,k`**

  Planificación estática con trozos de tamaño k. La asignación es *round robin*.

  Si no se indica k, se reparten trozos de tamaño N/P.

  `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`: Tipos

- **`runtime`**

  El tipo de planificación se define previamente a la ejecución en la variable de entorno `OMP_SCHEDULE`

  ```
  > export OMP_SCHEDULE="dynamic,3"
  ```

3.0 - **`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.

Se permite a las implementaciones añadir nuevos métodos de planificación.

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.**

Directiva parallel for

- 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 bucle (for)

```
#pragma omp parallel 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.]
```

Ejemplo: Producto Matriz-Vector

```
void mxv_row (int m, int n, double *a, double *b, double *c) {
  int i, j;
  double sum;

  #pragma omp parallel for default(none) \
              private(i, j, sum) shared(m, n, a, b, c)
  for(i=0; i<m; i++) {
    sum = 0.0;
    for(j=0; j<n; j++)
      sum += b[i*n+j] * c[j];
    a[i] = sum;
  } /*-- End of parallell for --*/
}
```

```
Compilación: gcc -fopenmp -o omp_mxv omp_mxv.c -lgomp
```

Reparto de tareas: funciones

Directiva `sections`

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

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

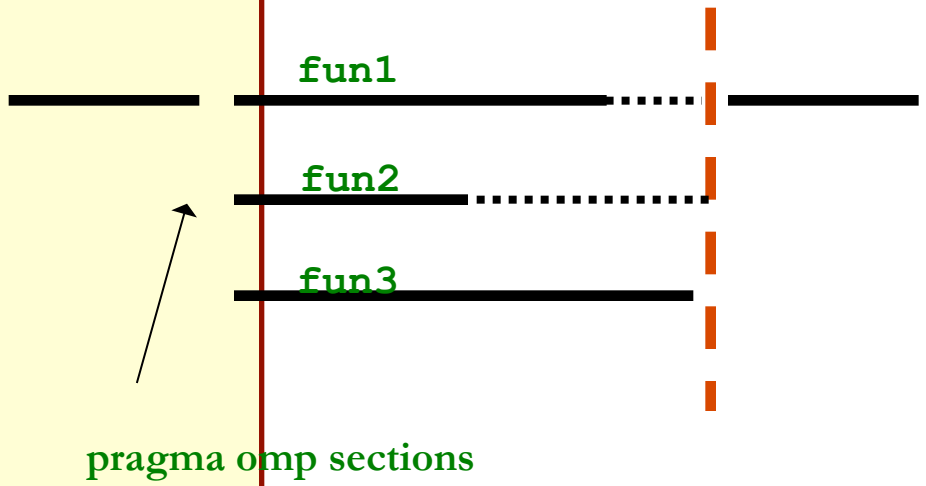
Una barrera implícita sincroniza el final de las secciones o tareas.

Cláusulas: `private (first-, last-),`

`reduction, nowait`

Reparto de tareas (sections)

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



- Al igual que con el pragma **`for`**, si la región paralela sólo tiene secciones, pueden juntarse ambos pragmas en uno solo:

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

Reparto de tareas (`single`)

3 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*.

```
#pragma omp single [cláus.]
```

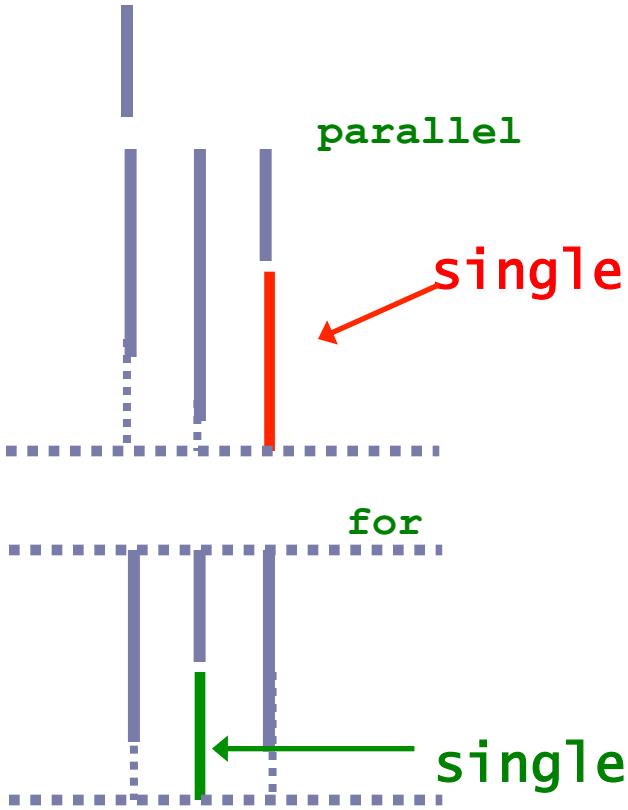
Cláusulas: `(first)private, nowait`

`copyprivate(X)`

⋆ Para pasar al resto de *threads* el valor de una variable **`threadprivate`**, modificada dentro del bloque **`single`**.

Reparto de tareas (`single`)

```
#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

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.

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

- **Puede obtenerse el número de procesadores disponibles mediante**

```
omp_get_num_proc() ;
```

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

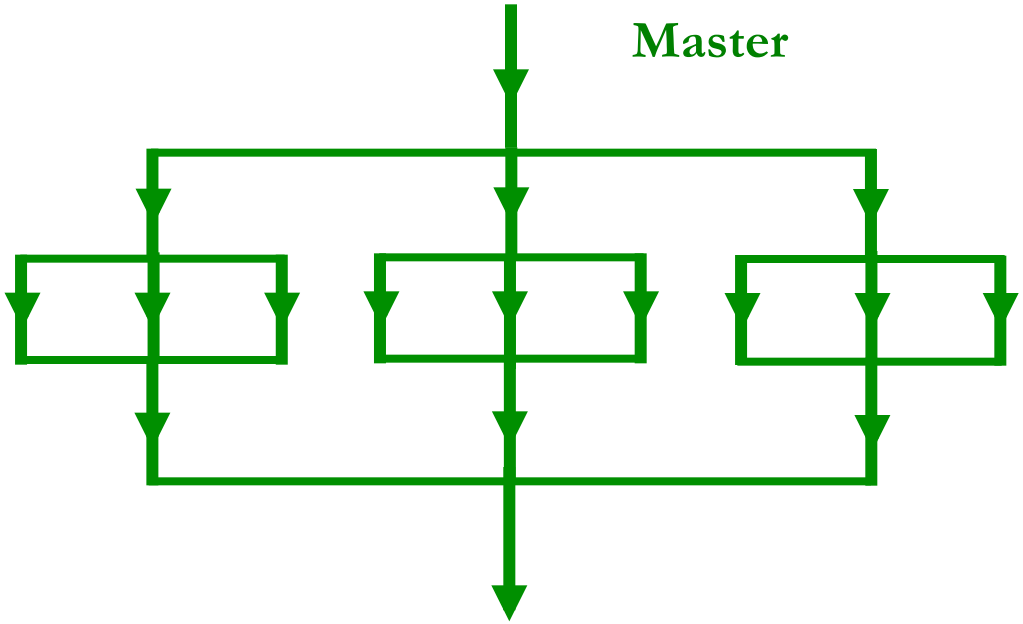
- Puede hacerse que el número de *threads* sea dinámico, en función del número de procesadores disponibles en cada instante:

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

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

```
omp_get_dynamic();
```

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:

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 **`omp_get_ancestor_thread_num(n)`** y al número de *threads* en dicho nivel.

Paralelización Guiada

Construcciones Master y Sincronizaciones

Directiva `master`: bloque ejecutado por el "master thread"

```
#pragma omp master <cr>
    bloque estructurado
```

Directiva `critical`: restringe la ejecución del bloque a un único thread cada vez. La región crítica se identifica con un nombre.

```
#pragma omp critical [(nombre)] <cr>
    bloque estructurado
```

Directiva `barrier`: sincroniza todos los threads en ese punto

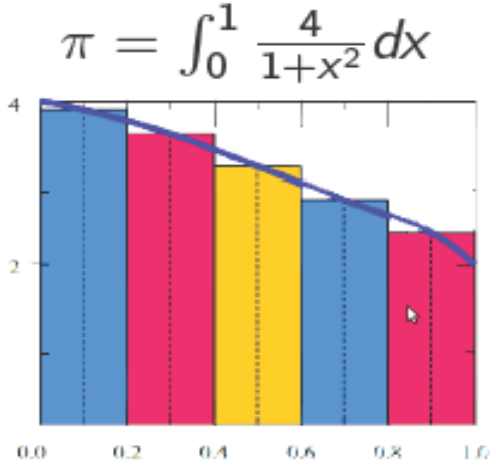
```
#pragma omp barrier <cr>
```

Directiva `atomic`: asegura la actualización atómica de una localización de memoria, no múltiple, por parte de los threads

```
#pragma omp atomic <cr>
    expresión-sentencia (op.simple no sobrecargado:+,-,*,/,&,^,|,<<,>>
```

Otras: single, flush, ordered, ...

Ejemplo: Cálculo de PI
Clausula *private* + Clausula *critical*




```
h = 1.0 / (double)n;
omp_set_num_threads(numThreads);
double x;

#pragma omp parallel private(x, sum)
{
  int id;
  id = omp_get_thread_num();
  sum = 0.0;
  for(i = id + 1; i <= n; i += numThreads) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  #pragma omp critical
    pi += sum;
}
pi *= h;
```

Sincronización de *threads*

- 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.

Exclusión mútua (Sección Crítica)

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`.

Exclusión mútua

- 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; }
 ...
}
```

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

Ejemplo : Producto Escalar

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

¡Puede que NO FUNCIONE por DATA RACE!

Ejemplo : Producto Escalar protegiendo

Hay que proteger el acceso a datos compartidos (shared) , o modificables

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

Exclusión mútua

- 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.

Sincronización

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.

Eventos (barreras)

- Sincronización global: barreras

```
#pragma omp barrier
```

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

Muchos constructores paralelos llevan implícita una barrera final.

Eventos (ordered)

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

Referencias OpenMP

Quinn Michael J, *Parallel Programming in C with MPI and OpenMP* McGraw-Hill Inc. 2004. ISBN 0-07-058201-7

R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000. ISBN 1558606718

B. Chapman et al. Using OpenMP, The MIT Press, 2008.

Especificación OpenMP http://www.openmp.org/

Compiladores:

- **El Intel C++ Compiler (icc) da un rendimiento muy alto para procesadores Intel pero es de pago si se utiliza para fines comerciales.**
- **GOMP es la implementación de OpenMP integrada en gcc desde noviembre del 2005.**

Repositorio de código OpenMP http://sourceforge.net/projects/ompscr/