Arquitectura de Sistemas paralelos

Tema 3: Paralelismo en sistemas multicore/multithread de memoria compartida

Asignatura: Arquitectura de sistemas paralelos

Profesor: Francisco Javier Gómez Arribas
Departamento de Tecnología Electrónica y de las Comunicaciones

UNIVERSIDAD AUTONOMA DE MADRID

Contenidos

- **Programación de multiprocesadores mediante compiladores paralelos (OpenMP)**

Compilador Paralelo Open MP

Todos los compiladores paralelos funcionan de forma similar

- OpenMP specification
  http://www.openmp.org/
- OpenMP Compiler
- Intel (currently free for personal or educational use on Linux)
  http://developer.intel.com/software/products/compilers/
- gcc
  http://phase.etl.go.jp/Omni/

Compilador omcc (man omcc) sobre Linux (Omni Project)

- **Secuencial:**
  - **Compilación: gcc**
- **Paralelo:**
  - **Compilación: omcc**
  - **Ejecución: variables de entorno**

Compilador Paralelo Open MP

Conceptos.

- SPMD (Single Program Multiple Data)
  - un solo programa fuente
- Datos Privados / compartidos
  - Se distingue entre thread con datos privados y datos compartidos por todos los threads
- Se comparten trabajos
  - Se puede influenciar en la manera de compartir trabajos entre threads
- Sincronización
  - Se dispone de maneras para sincronizar threads

Ejemplo "Hello world"

- **Programa**

```
#include <omp.h>
int main(int argc, char **argv)
{
#pragma omp parallel
  printf("Hello world\n");
}
```

- **Compilación**

```
icc -openmp openmp_test.c
```

- **Ejecución**

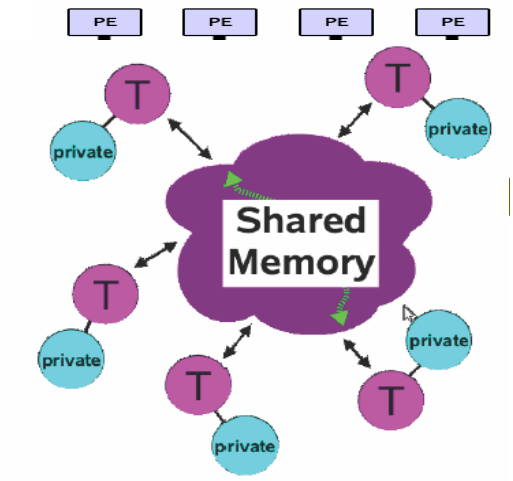
```
export OMP_NUM_THREADS=2
./a.out
Hello world
Hello world

export OMP_NUM_THREADS=3
./a.out
Hello World
Hello World
Hello World
```

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 Programación 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 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 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];
 }
```

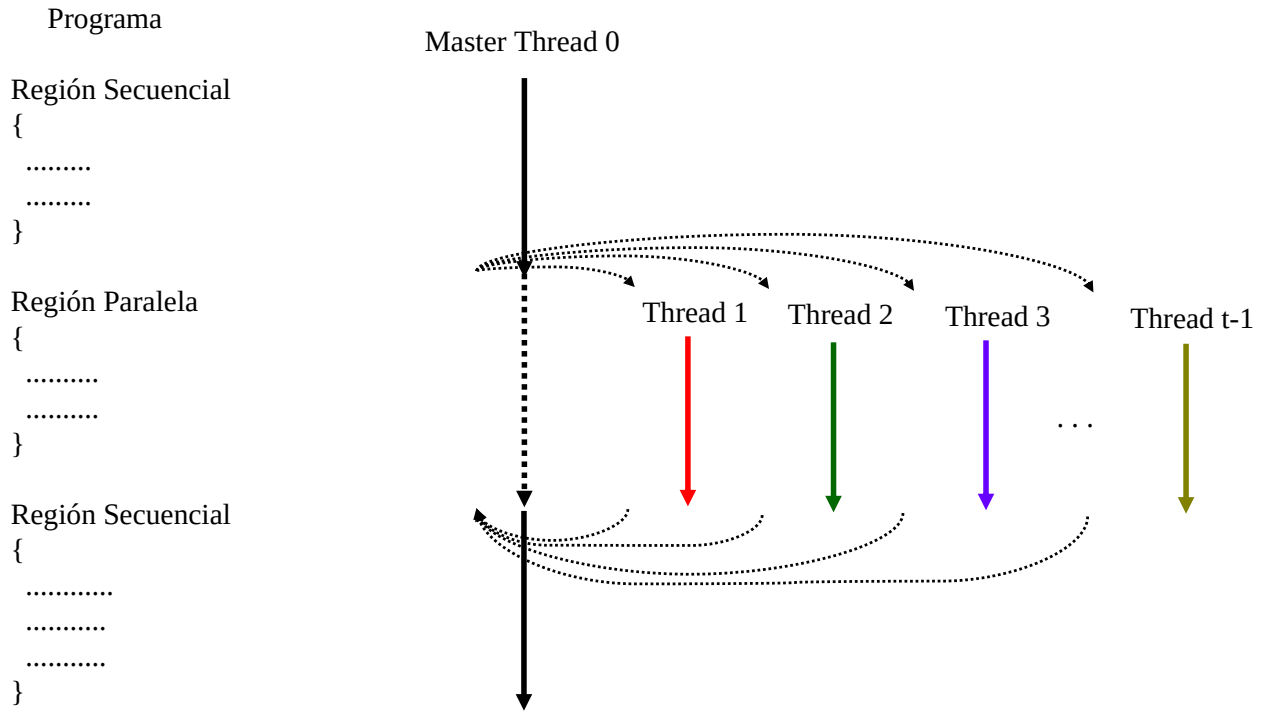
Son vistas como comentarios por otros compiladores

Cuando hay conflictos entre directivas, prevalece la última.

Términos 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 y Secuenciales



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

- Gestión de Threads: ¿Qué Thread ? ¿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.

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: 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*.

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

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

Regiones Paralelas: 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.

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.

Región Paralela: Cláusulas

- Cláusulas de la región paralela
  - `shared`, `private`, `firstprivate`(var)
    `default`(shared/none)
    `reduction`(op:var)
    `copyin`(var)
  - `if` (expresión)
  - `num_threads`(expresión)

Región Paralela: 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*.

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

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).

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.

- `lastprivate()`

  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()`

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 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.

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

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.

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.

R.P.: Cláusulas de ámbito

Para variables declaradas como threadprivate, un thread no puede acceder a la copia threadprivate de otro thread (ya que es privada).

`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. No se necesita *copyout* ya que si que se mantiene el valor de la variables threadprivate durante la ejecución del programa.

Región Paralela: Otras cláusulas

- `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

Reparto de tareas en paralelo con OpenMP

- 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*.**

Paralelización Guiada o Explícita

Construcciones de trabajo compartido

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

OpenMP define las siguientes construcciones de trabajo compartido:

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

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

- Construcción **`sections`**
  - 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`)

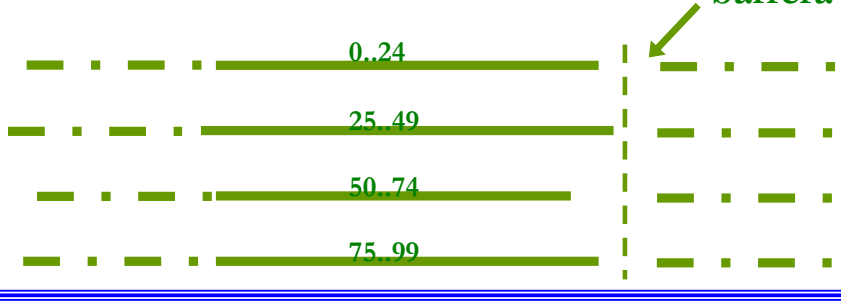
1 Directiva `for`

```
#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.

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.**

- 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.**

Cláusulas

- Cuando la directiva es **`parallel for`** (una región paralela que sólo tiene un bucle **`for`**), pueden usarse las cláusas 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 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

2 Directiva `sections`

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

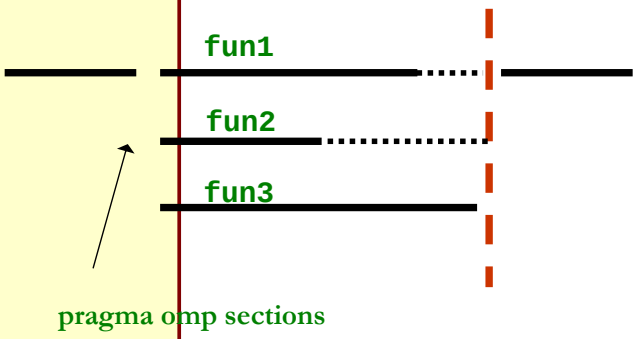
Cada sección paralela es ejecutada por un sólo *thread*, y cada *thread* ejecuta alguna sección o ninguna.

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.]`

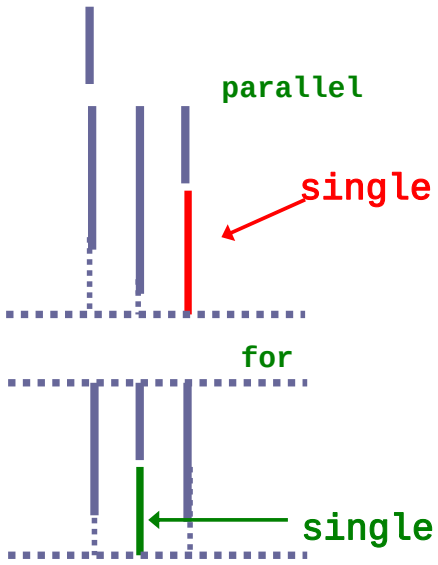
  Cláusulas: `(first)private, nowait`

  `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*.

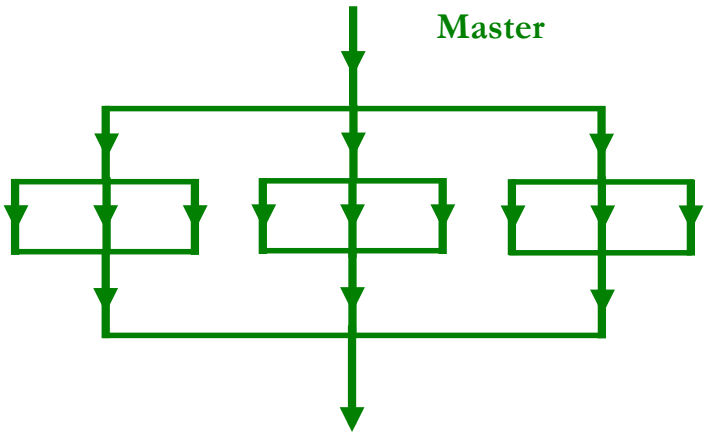
- Puede hacerse que el número de *threads* sea dinámico, en función del número de procesadores disponibles en cada instante:

  ```
  > 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, ...

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 (SC)

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!

Exclusión mútua

- 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]; }
 }
```


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.

Variables cerrojo

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.

Variables cerrojo (lock)

- **`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.

Variables cerrojo (lock)

> 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);
  }
}
```


Eventos

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*.

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 (barreras)

> 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);
}
```

Eventos (*flags*)

- Sincronización punto a punto

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

```
/* 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.

Eventos (*flags*)

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.

```
/* 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;
  ...
```

Eventos (*flags*)

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;
  ...
```


Eventos (`ordered`)

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

Sincronización

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.

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/