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

UAM
UNIVERSIDAD AUTONOMA
DE MADRID

Contenidos

- **Multiprocesador: Modelos de Programación Paralela**
  - **Características distintivas de cada modelo**
  - **Ejecución en Threads o Procesos**
  - **Compiladores Paralelizadores**
- **Paralelización de bucles**
  - **Análisis de dependencias.**
  - **Criterios de paralelización.**
  - **Principales optimizaciones**
- **Programación de multiprocesadores con OpenMP**
  - **Consideraciones de rendimiento**
    - False Sharing
    - Carreras de datos
  - **Programación hibrida OpenMP + MPI**

Modelos de programación en Sistemas Multiprocesador

- 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.

- **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 se puede utilizar una mezcla de ambos.**

Modelos de Programación

En Memoria Compartida (MC)

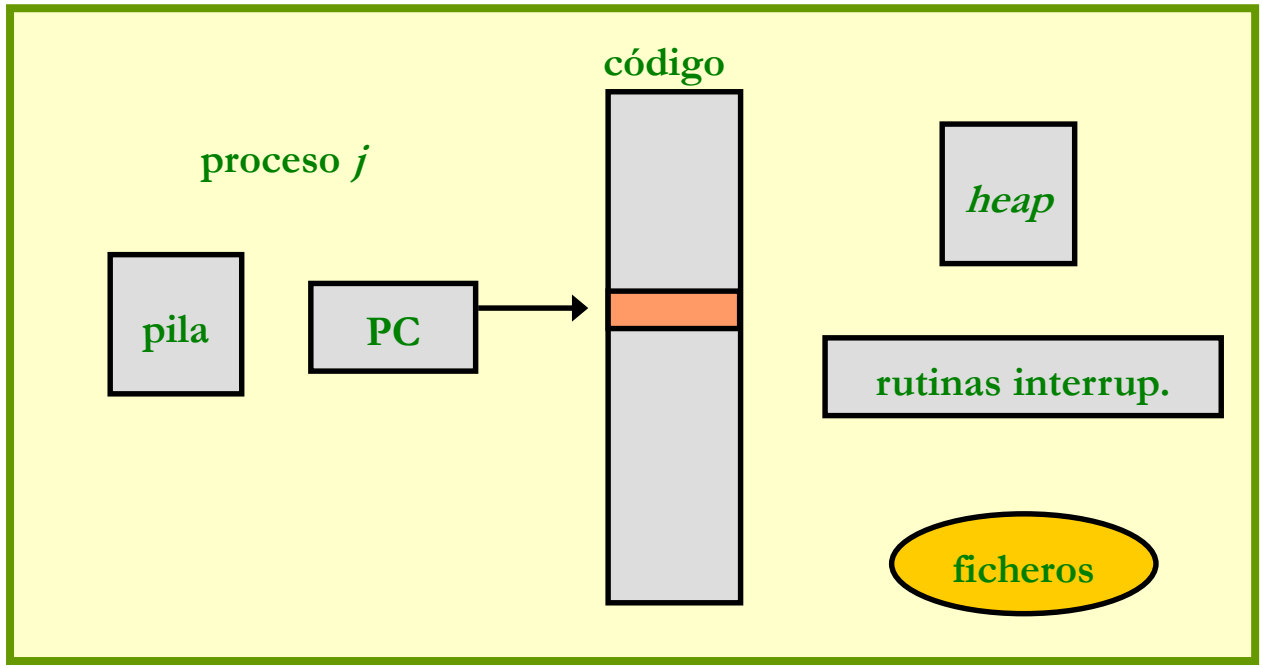
- **Varios procesos o threads ejecutándose en un espacio de direcciones común**
- **Comunicación implícita en acceso (lectura o escritura) a memoria compartida (MC)**
- **Sincronización ≡ escritura en MC**
- **Procesos (pesados) → hebras o *threads* (ligeros)**
- **Alternativas:**
  - **usar un lenguaje 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. OpenMP…**
  - **usar un lenguaje secuencial junto con rutinas de librería.**

> usar *threads:* Pthreads (POSIX), Java,

Entidades de Ejecución

Procesos (Pesados)

- **Un proceso tiene su espacio de direcciones virtual, con código como datos y los recursos necesarios para ejecutar un programa.**
- **En cada proceso sólo hay un contexto de ejecución**
- **Cada proceso puede ser ejecutado por un procesador distinto**



Procesos concurrentes:

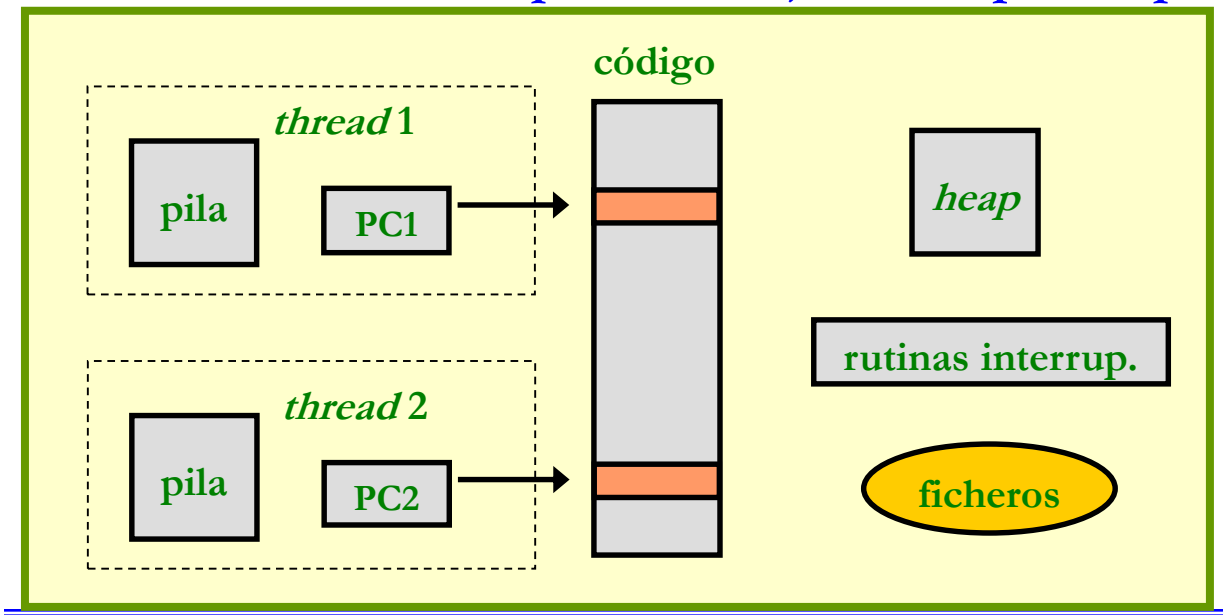
Fork / Join

- gran *overhead*, de creación

Entidades de Ejecución

Threads (procesos ligeros, hebras, etc.)

- **Los threads comparten el espacio de direcciones del proceso que les ha creado. Un proceso puede/suele tener varios threads**
- **Cada thread tiene accceso al segmento de datos de su proceso, pero tiene asociado un contexto que incluye: Contador de programa, registros del procesador, variables locales y pila**
- **Creación y destrucción rápida, tienen poco overhead.**
- **Cada contexto puede ser ejecutado por un procesador distinto.**



- Los *threads* son más "ligeros"
- La sincronización (variables compartidas) es sencilla
- Es habitual en todos los S.O. (linux, windows …)

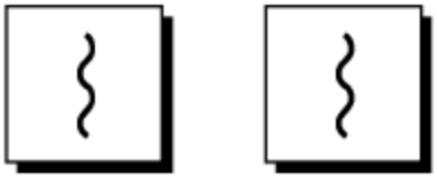
Estándares: Pthreads (Posix threads) y Java.

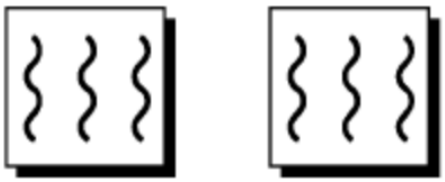
Diferencias entre Procesos e Hilos de Ejecución

- Los hilos comparten el espacio de direcciones del proceso que lo creó; Los procesos tienen su propio espacio de direcciones.
- Los hilos tienen acceso directo al segmento de datos de su proceso; Los procesos tienen su propia copia del segmento de datos del proceso padre.
- Los hilos pueden comunicarse directamente con otros hilos de su proceso; Los procesos deben utilizar la comunicación entre procesos para comunicarse entre sí.
- Los hilos casi no tienen sobrecarga; mientras que los procesos requieren la duplicación de recursos del proceso padre.
- Los hilos pueden ejercer control sobre los hilos del mismo proceso; Los procesos solo pueden ejercer control sobre los procesos secundarios.
- Los cambios en el hilo principal (cancelación, cambio de prioridad, etc.) pueden afectar el comportamiento de los otros hilos del proceso; los cambios en el proceso primario no afectan los procesos secundarios.


Un proceso,
un hilo


Un proceso,
múltiples hilos


Múltiples procesos,
un hilo por proceso


Múltiples procesos,
múltiples hilos por proceso

Paralelismo

- **¿Cómo explotar el paralelismo?**
  - **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

- **¿Qué características deben tener un programa paralelo?**

  Los programas usan simultáneamente más de un elemento de proceso.

  - **Programación en Memoria Compartida**: Explota el paralelismo de grano fino
  - **Programación de Paso de Mensajes**: Explota el paralelismo grano grueso

Paralelismo

- Tipos de paralelismo a explotar (1)

Paralelismo de datos

```
do i = 1, 3000
  A(i) = func(i)
enddo
```

P0

```
do i = 1, 1000
  A(i) = func(i)
enddo
```

P1

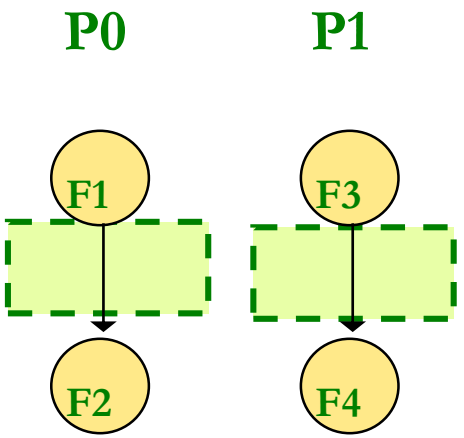
```
do i = 1001, 2000
  A(i) = func(i)
enddo
```

P2

```
do i = 2001, 3000
  A(i) = func(i)
enddo
```

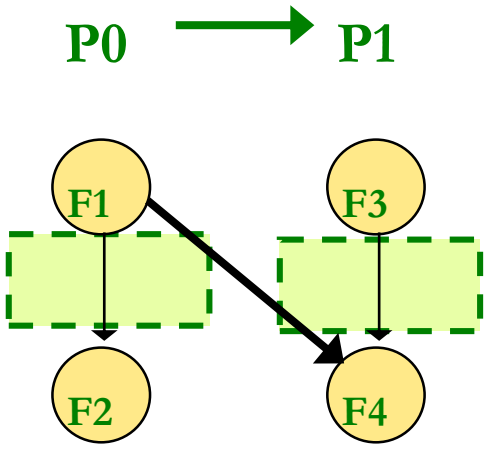
- Tipos de paralelismo a explotar (2)

Paralelismo de función



Paralelismo

- ¿ Cómo tratar la dependencias entre tareas?



Sincronización

- global (barreras)
- punto a punto (eventos)

Paralelismo

Objetivo a nivel práctico: Paralelización de bucles

- **Explotar el paralelismo de grano fino / medio**
- **Metodología: reparto de iteraciones del bucle**
- **Problema a resolver: análisis de dependencias**

¿Cómo hacerlo de manera eficiente?

- **El programador vs el compilador**

- Se necesita que una fracción importante del código pueda ejecutarse en paralelo. ¡No olvidar la ley de Amdahl!

Modos de Paralelización

- **Modo Automático (no soportado por OpenMP)**
  - El compilador analiza el código y paraleliza los bucles (los internos)
  - Las prestaciones dependen de la organización del código
  - Malas prestaciones en algunos casos.
- **Modo guiado o explícito**
  - Compilación: `omcc programa.c`
  - El programador indica mediante directivas qué bloques de código se deben paralelizar y sus condiciones adicionales
  - Directiva: Comando inmediatamente anterior al bucle o sección a paralelizar

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x[i]= alfa*y[i];
  }
```

Compiladores Paralelos

- **Desarrollados a mediados de los 80 para SMP (Symmetric Multiproc.)**
- **Mal uso → reducción de prestaciones**
- **Funcionamiento → Ejecutar bucles y secciones de código en paralelo**
- **Ejemplos:**

a) Ej. un proceso/un contexto

```
for (i=0; i<4*n; i++) {
  .......
 }
```

b) Ej. cuatro contextos de ejecución

```
Contexto j = (0,1,2,3)
 for (i=j*n; i<(j+1)*n; i++){
.......
 }
```

c) Bucles Paralelizable

```
for (i=0; i<n; i++){
  x[i]=alfa*y[i];
}
```

d) Bucles no paralelizable:

```
for (i=0; i<n-1;i++){
    x[i]=x[i+1]+y[i];
    }
```

Paralelización Automática

Bucles paralelizables

- **Bucles** for **(no** while **o** for **infinito)**
- **Sin dependencias de datos**
- **Sin modificación de variables escalares**
- **Sin condiciones de salto fuera del bucle**
- **El bucle exterior no ha sido paralelizado**
- **Sin llamadas a funciones o procedimientos del usuario**
- **Sin Entrada/Salida**

Dependencias aparentes

- **El compilador asume que hay dependencias entre iteraciones**
- **Las prestaciones son bajas si el compilador no paraleliza el bucle**

Solución: Hay que reorganizar el código o recurrir a la paralelización explícita

Paralelización Guiada o Explícita

Pasos a seguir

- **1.- Desarrollo del programa secuencial**
- **2.- Analizar los bucles para detectar dependencias. Ir a 3/4**
- **3.- Modificar el código con otras secuencias de resolución del problema. Introducir optimizaciones de procesador. Ir a 2/4**
- **4.- Insertar directivas en los bucles**
- **5.- Compilar con las opciones adecuadas**
- **6.- Comprobar la paralelización del código**
- **7.- Iniciar las variables de entorno de ejecución**
- **8.- Ejecutar el programa y comprobar los resultados. Las directivas mal usadas pueden provocar resultados erróneos**

Contenidos

- Modelos de Programación Paralela en Sistemas multiprocesador.
  - Características distintivas de cada modelo
  - Ejecución en Threads o Procesos
  - Compiladores Paralelizadores
- **Paralelización de bucles**
  - **Análisis de dependencias.**
  - **Criterios de paralelización.**
  - **Principales optimizaciones**
- Programación de multiprocesadores con OpenMP

Paralelización de Bucles: Análisis de dependencias

dependencias verdaderas

dependencias de nombre

- dependencia

RAW

```
i:  A =
    ...
j:      = A
```

- antidependencia

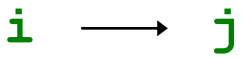
WAR

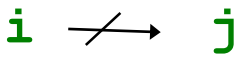
```
i:      = A
    ...
j:  A =
```

- dependen. de salida

WAW

```
i:  A =
    ...
j:  A =
```





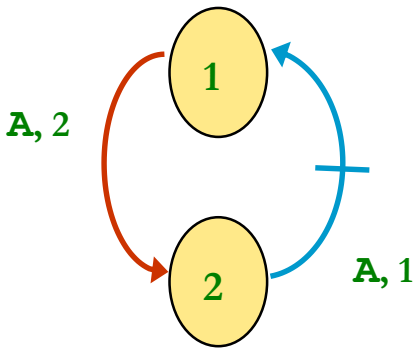
Paralelización de Bucles: Grafo de dependencias

Bucles

+ Grafo de dependencias

+ Distancia de la dependencia

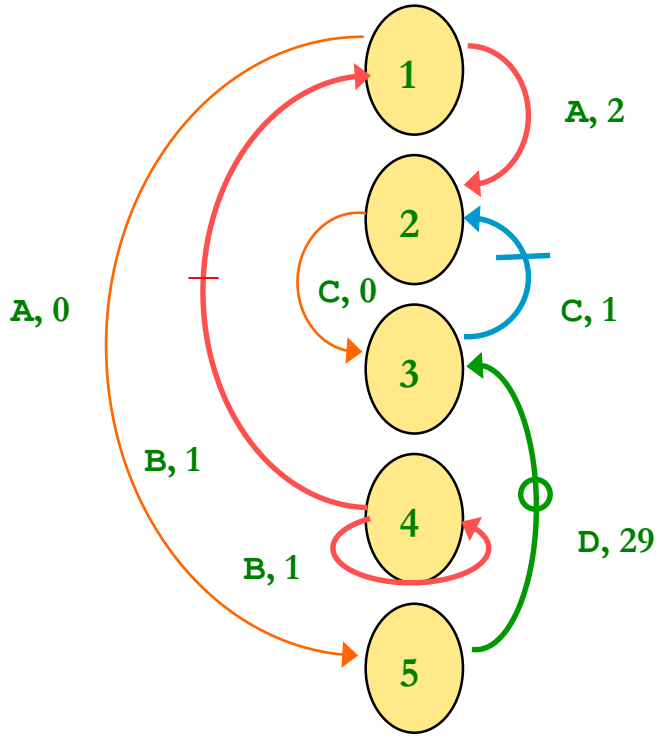
```
do i = 2, N-2
1   A(i) = B(i) + 2
2   C(i) = A(i-2) + A(i+1)
enddo
```




```
A(2) = B(2) + 2
C(2) = A(0) + A(3)

A(3) = B(3) + 2
C(3) = A(1) + A(4)

A(4) = B(4) + 2
C(4) = A(2) + A(5)

A(5) = B(5) + 2
C(5) = A(3) + A(6)
```

Paralelización de Bucles: Análisis de dependencias

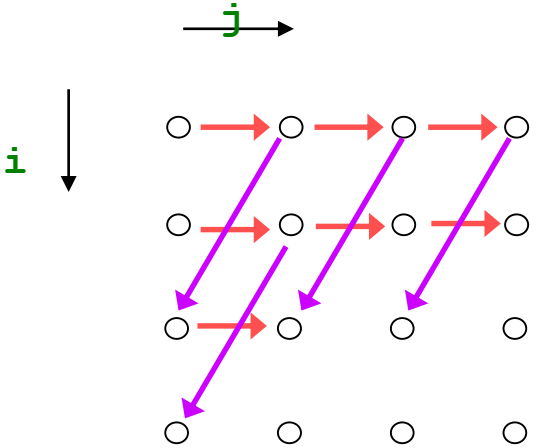
```
do i = 2, N-31
1   A(i) = B(i) + 2
2   C(i) = A(i-2)
3   D(i+1) = C(i) + C(i+1)
4   B(i+1) = B(i) + 1
5   D(i+30) = A(i)
enddo
```

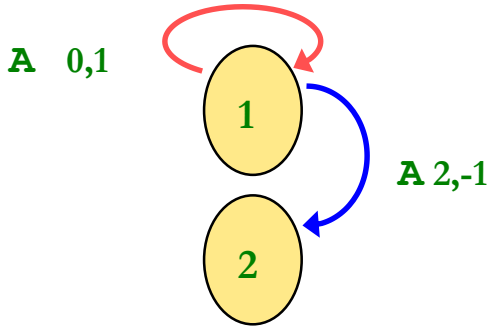


Paralelización de Bucles: Análisis de dependencias

Espacio de iteraciones

```
do i = 2, N-1
  do j = 1, N-2
  1  A(i,j) = A(i,j-1) * 2
  2  C(i,j) = A(i-2,j+1) + 1
  enddo
enddo
```



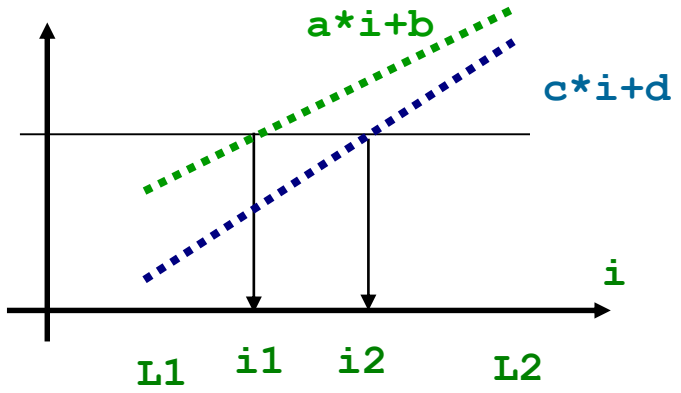


Paralelización de Bucles: Test de dependencias

- Test de dependencias: únicamente para funciones lineales del índice del bucle.

```
do i = L1, L2
  X(a*i+b) =
           = X(c*i+d)
enddo
```

?



$$\frac{d-b}{MCD(c,a)} \notin Z \rightarrow \text{no hay depend.}$$

Paralelización de Bucles: Repartir iteraciones

- Paralelizar el código significa repartir las iteraciones de un bucle a threads o procesadores.
  - Pero hay que respetar las dependencias de datos.
  - El problema principal son las dependencias de **distancia > 0**, y sobre todo aquellas que forman **ciclos** en el grafo de dependencias.
- Siempre es posible ejecutar un bucle en P threads o procesadores, añadiendo sincronización para asegurar que se respetan las dependencias de datos…

… lo que no significa que siempre sea sensato hacerlo, pues hay que tener en cuenta el coste global: cálculo + sincronización (comunicación).

Paralelización de Bucles: Repartir iteracciones

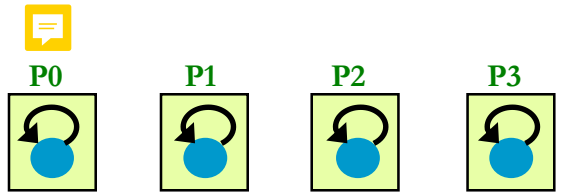
Objetivos:

- **Repartir las iteraciones de un bucle entre los procesadores, para que se ejecuten "simultaneamente".**
- **Siempre que se pueda, que se ejecuten de manera independiente, sin necesidad de sincronizar (dependencias de distancia 0).**
- **En función de las características del sistema (comunicación, reparto de tareas…) hay que intentar que las tareas tengan un cierto tamaño.**

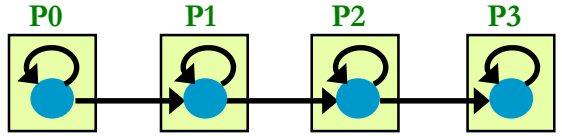
Limitaciones :Tal vez sólo se pueda utilizar un número limitado de procesadores.

Paralelización de Bucles: Casos habituales

```
do i = 0, N-1
   A(i) = A(i) + 1
   B(i) = A(i) * 2
enddo
```

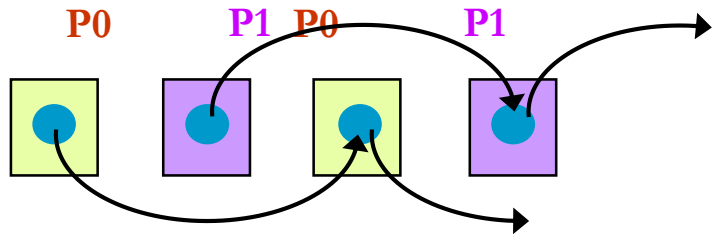



```
do i = 0, N-2
   A(i) = B(i) + 1
   B(i+1) = A(i) * 2
enddo
```



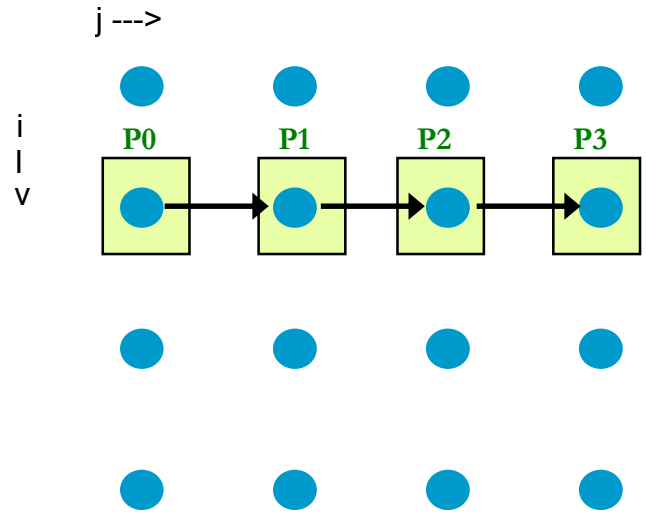
?

```
do i = 0, N-3
   A(i+2) = A(i) + 1
enddo
```

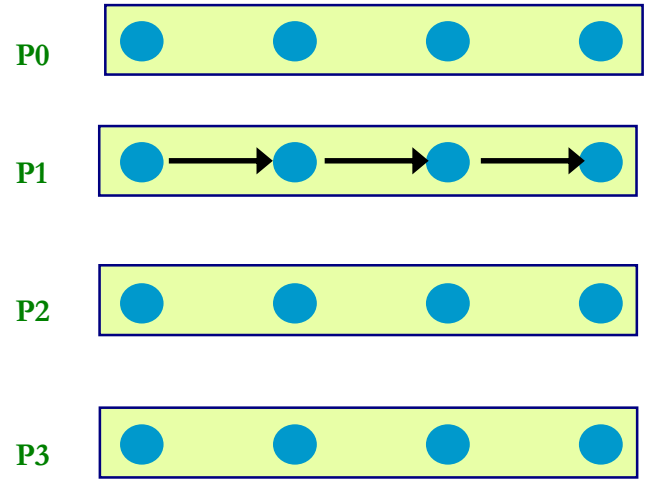


Paralelización de Bucles: Casos habituales

```
do i = 0, N-1
 do j = 1, M-1
   A(i,j) = A(i,j-1) + 1
  enddo
enddo
```




```
do i = 0, N-1
 do j = 1, M-1
   A(i,j) = A(i,j-1) + 1
  enddo
enddo
```



Paralelización de Bucles: Criterios

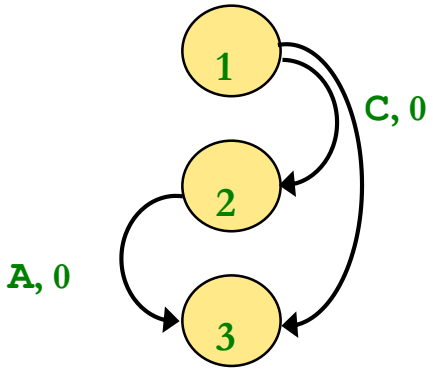
- Si todas las dependencias son de distancia 0 (las iteraciones son independientes), éstas se pueden repartir como se quiera entre los procesadores, sin tener que sincronizarlas: doall.
- Si hay dependencias entre iteraciones, pero todas van hacia adelante, se pueden sincronizar mediante barreras: forall (o doall + barrier).
- Si las dependencias forman ciclos, hay que utilizar sincronización punto a punto: doacross.

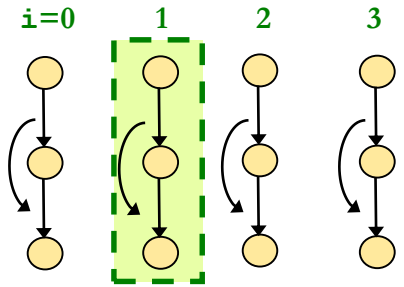
Paralelización de Bucles: iteraciones independientes

- Las iteraciones son independientes, por lo que el reparto puede hacerse como se quiera: doall.

```
do i = 0, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i) / A(i)
enddo
```

```
doall i = 0, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i) / A(i)
enddoall
```





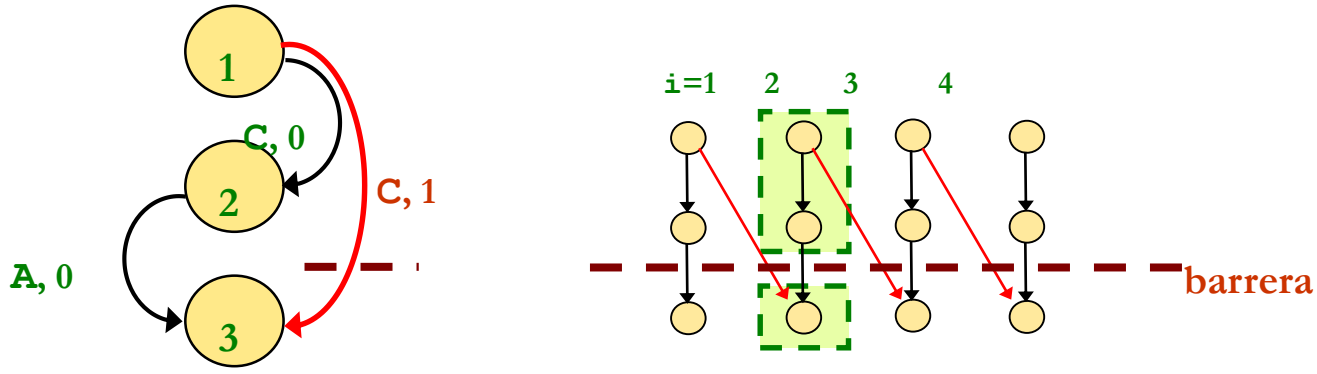
Paralelización de Bucles: dependencia forward

- Hay dependencias entre iteraciones, pero todas van "hacia adelante": forall.
- Cada dependencia puede sincronizarse con una barrera: los procesos esperan a que todos hayan ejecutado una determinada instrucción antes de pasar a ejecutar la siguiente.
- Hay más sincronización que la estrictamente necesaria, pero es sencillo de implementar.

Paralelización de Bucles: dependendencia forward

```
do i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i-1) / A(i)
enddo
```

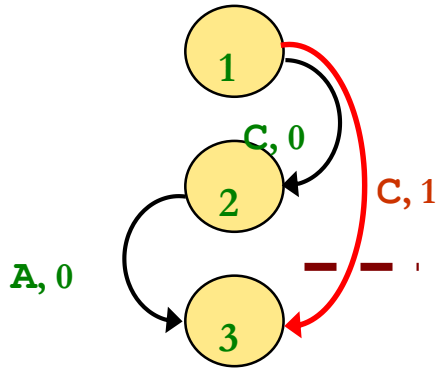
```
forall i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  BARRERA (...)
  D(i) = C(i-1) / A(i)
endforall
```

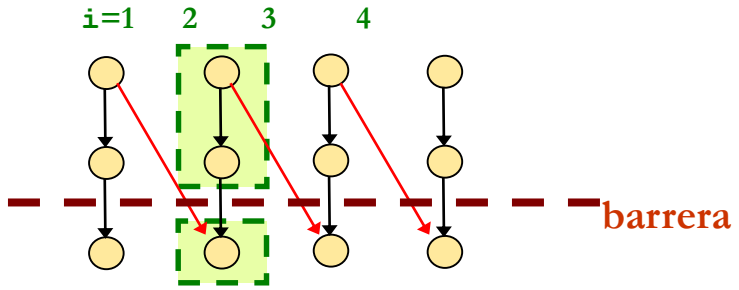


Paralelización de Bucles: dependencia forward

```
do i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
  D(i) = C(i-1) / A(i)
enddo
```

```
doall i = 1, N-1
  C(i) = C(i) * C(i)
  A(i) = C(i) + B(i)
enddoall
[ BARERRA (...) ]
doall i = 1, N-1
  D(i) = C(i-1) / A(i)
enddoall
```





Paralelización de Bucles: dependencia con ciclos

- Las dependencias forman ciclos: doacross, sincronización punto a punto (productor / consumidor).

- **Sincronización de las dependencias mediante vectores de eventos:**

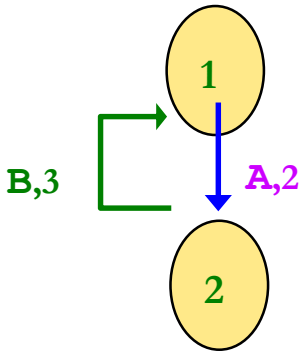
  `post(vA,i)` → `vA(i) := 1`

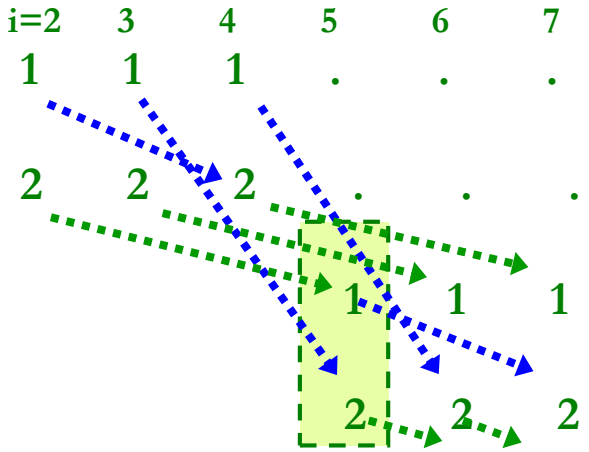
  `wait(vA,i)` → esperar a que `vA(i) = 1`

- **Si `vA(i) = 0`, aún no se ha ejecutado la iteración i de la instrucción que se está sincronizando.**

Paralelización de Bucles: dependencia doacross

```
do i = 2, N-2
  A(i) = B(i-2) + 1
  B(i+1) = A(i-2) * 2
enddo
```

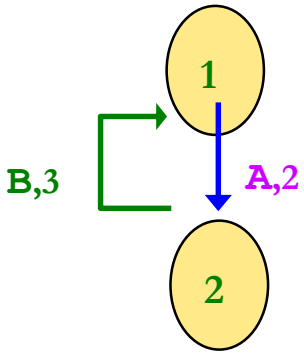


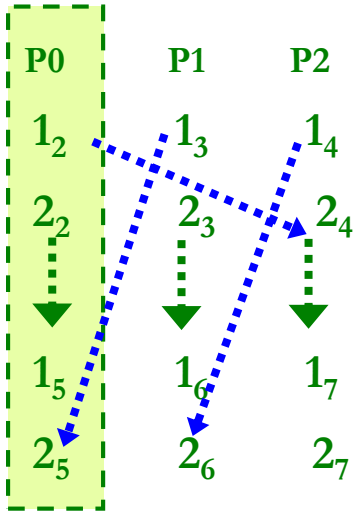



```
doacross i = 2, N-2

  wait (vB,i-3)
  A(i) = B(i-2) + 1
  post (vA,i)

  wait (vA,i-2)
  B(i+1) = A(i-2) * 2
  post (vB,i)

enddoacross
```

Paralelización de Bucles: dependencia doacross

```
do i = 2, N-2
  A(i) = B(i-2) + 1
  B(i+1) = A(i-2) * 2
enddo
```



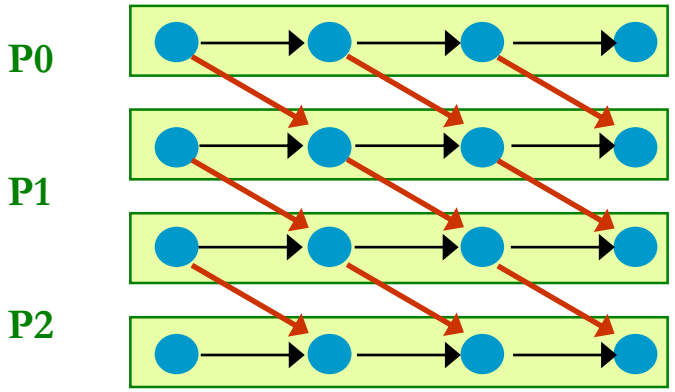



```
doacross i = 2, N-2, 3
  wait (vB,i-3)
  A(i) = B(i-2) + 1
  post (vA,i)

  wait (vA,i-2)
  B(i+1) = A(i-2) * 2
  post (vB,i)

enddoacross
```

Paralelización de Bucles: dependencia doacross

- Vectores de eventos en dos dimensiones

```
do i = 0, N-2
 do j = 0, N-2
   A(i+1,j+1) = A(i+1,j) + 1
   B(i,j) = A(i,j)
 enddo
enddo
```



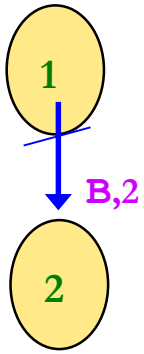

```
doacross i = 0, N-2
 do j = 0, N-2
   A(i+1,j+1) = A(i+1,j) + 1
   post (vA,i,j)
   wait (vA,i-1,j-1)
   B(i,j) = A(i,j)
 enddo
enddoacross
```

Paralelización de Bucles: dependencia doacross

- Antidependencias / Dependencias de salida

  Si son entre procesos, hay que sincronizarlas.

```
do i = 0, N-3
  A(i) = B(i+2) / A(i)
  B(i) = B(i) + C(i)
enddo
```



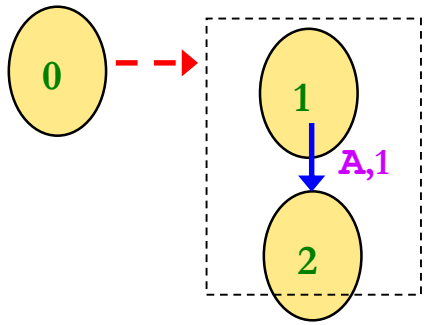

```
doacross i = 0, N-3
  A(i) = B(i+2) / A(i)
  post (vB,i)
  wait (vB,i-2)
  B(i) = B(i) + C(i)
enddoacross
```

```
LD B(i+2)
post
...
wait
ST B(i)
```

Paralelización de Bucles: dependencia doacross

- Instrucciones de tipo if

```
do i = 1, N-1
  if (B(i) > 0) then
     A(i) = A(i) + B(i)
     C(i) = A(i-1) / 2
   endif
enddo
```




```
doacross i = 1, N-1

  if (B(i) > 0) then
     A(i) = A(i) + B(i)
     post (vA,i)
     wait (vA,i-1)
     C(i) = A(i-1) / 2

  else post (vA,i)

  endif
end
```

Paralelización de Bucles: Optimizaciones

0. Deshacer la definición de constantes y las variables de inducción.

1. Eliminar todas las dependencias que no son intrínsecas al bucle. Por ejemplo:

```
do i = 0, N-1
  X = A(i) * B(i)
  C(i) = SQRT(X)
  D(i) = X - 1
enddo
```

convertir X en una variable privada

```
doall i = 0, N-1
  X(i) = A(i) * B(i)
  C(i) = SQRT(X(i))
  D(i) = X(i) - 1
enddoall
```

Paralelización de Bucles: Optimizaciones

2. Fisión del bucle.

Si una parte del bucle hay que ejecutarla en serie, romper el bucle convenientemente y ejecutar lo que sea posible en paralelo.

```
do i = 1, N-1
  A(i) = A(i-1) / 2
  D(i) = D(i) + 1
enddo
```

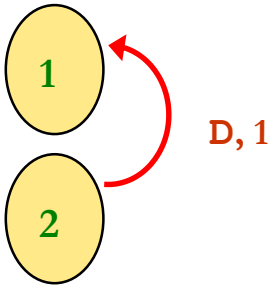
```
do i = 1, N-1
  A(i) = A(i-1) / 2
enddo

doall i = 1, N-1
  D(i) = D(i) + 1
enddoall
```

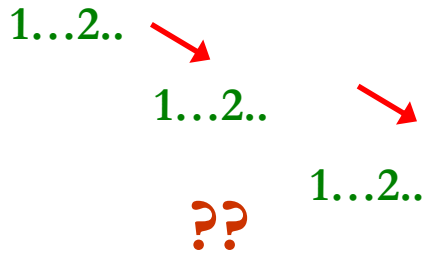
Paralelización de Bucles: Optimizaciones

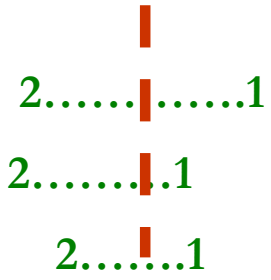
3. Ordenar las dependencias (hacia adelante).

```
do i = 1, N-1
  A(i) = D(i-1) / 2
  D(i) = C(i) + 1
enddo
```

```
forall i = 1, N-1
  D(i) = C(i) + 1
  BARRERA (...)
  A(i) = D(i-1) / 2
endforall
```



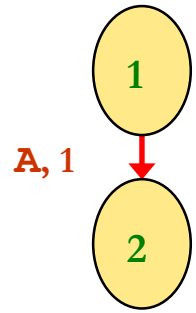


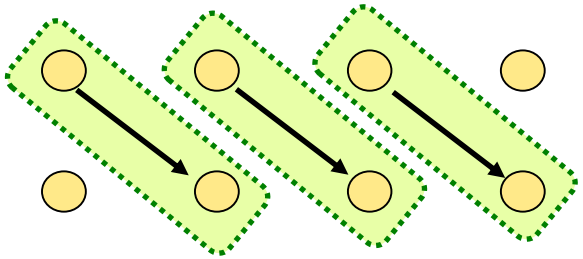


Paralelización de Bucles: Optimizaciones

4. Alinear las dependencias: *peeling*.

```
do i = 1, N-1
  A(i) = B(i)
  C(i) = A(i-1) + 2
enddo
```

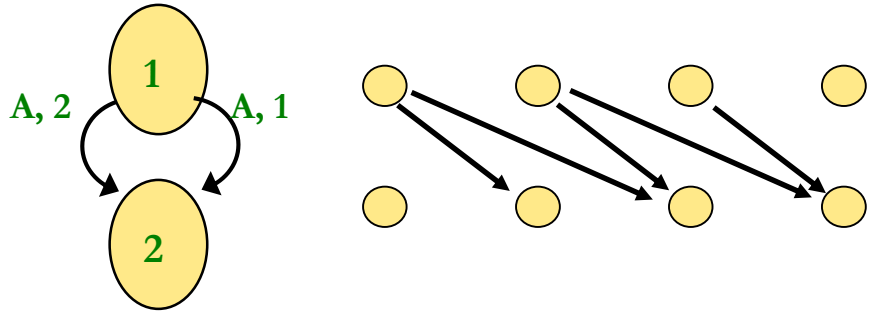




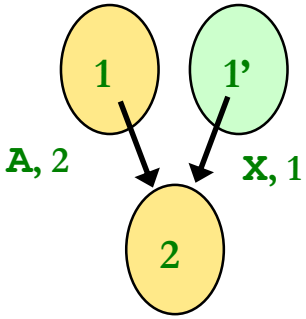
```
C(1) = A(0) + 2
doall i = 1, N-2
  A(i) = B(i)
  C(i+1) = A(i) + 2
enddoall
A(N-1) = B(N-1)
```

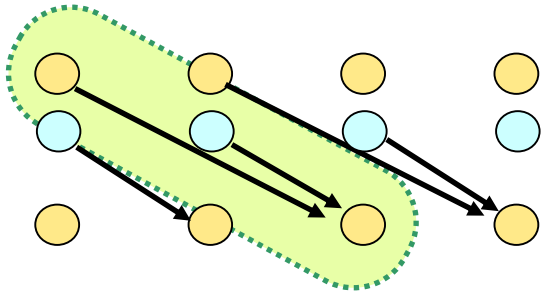
Paralelización de Bucles: Optimizaciones

```
do i = 2, N-1
  A(i) = B(i)
  C(i) = A(i-1) + A(i-2)
enddo
```




```
do i = 2, N-1
  A(i) = B(i)
  X(i) = B(i)
  C(i) = X(i-1) + A(i-2)
enddo
```



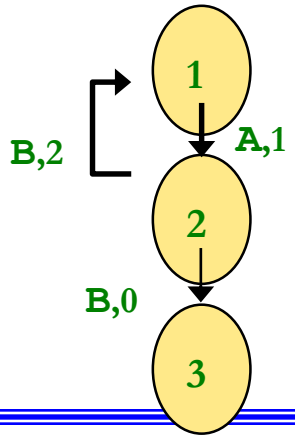


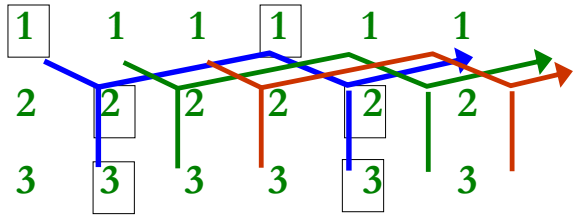
```
C(2) = A(1) + A(0)
C(3) = B(2) + A(1)

doall i = 2, N-3
  A(i) = B(i)
  X(i+1) = B(i+1)
  C(i+2) = X(i+1) + A(i)
enddoall

A(N-2) = B(N-2)
A(N-1) = B(N-1)
```

Paralelización de Bucles: Optimizaciones

5. Generar hilos independientes

```
do i = 0, N-3
  A(i+1) = B(i) + 1
  B(i+2) = A(i) * 3
  C(i) = B(i+2) - 2
enddo
```



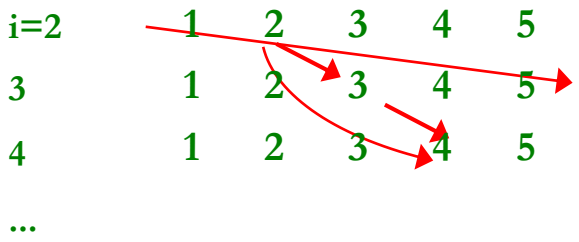



```
B(2) = A(0) * 3
C(0) = B(2) - 2

doall k = 0, 2
 do i = pid, N-4, 3
   A(i+1) = B(i) + 1
   B(i+3) = A(i+1) * 3
   C(i+1) = B(i+3) - 2
 enddo
enddoall

A(N-2) = B(N-3) + 1
```

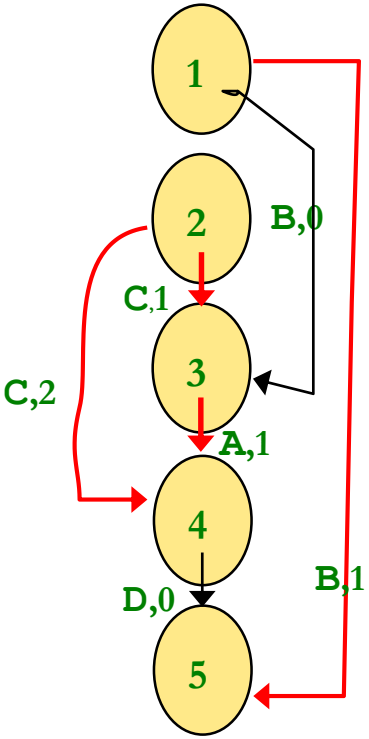
Paralelización de Bucles: Optimizaciones

6. Minimizar la sincronización

```
do i = 2, N-1
  B(i) = B(i) + 1
  C(i) = C(i) / 3
  A(i) = B(i) + C(i-1)
  D(i) = A(i-1) * C(i-2)
  E(i) = D(i) + B(i-1)
enddo
```

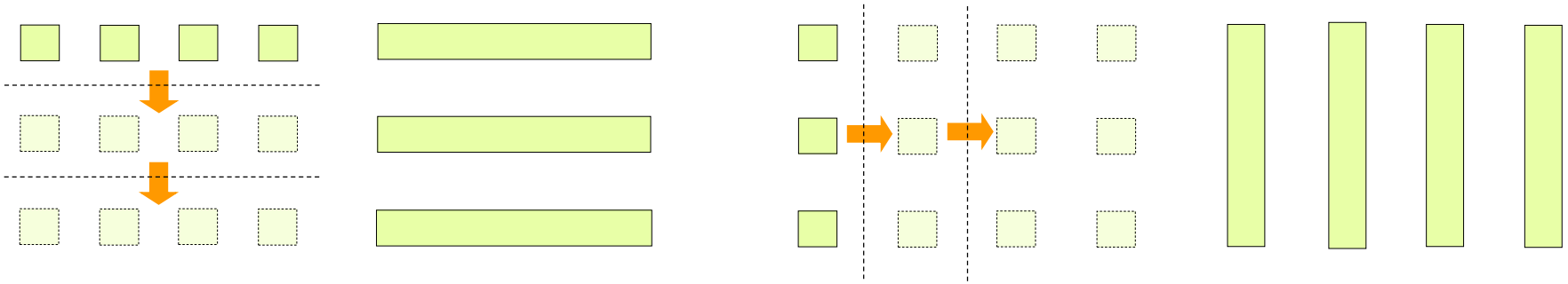



```
doacross i = 2, N-1
  B(i) = B(i) + 1
  C(i) = C(i) / 3
  post (vC,i)

  wait (vC,i-1)
  A(i) = B(i) + C(i-1)
  post (vA,i)

  wait (vA,i-1)
  D(i) = A(i-1) * C(i-2)
  E(i) = D(i) + B(i-1)
enddoacross
```



Paralelización de Bucles: Optimizaciones

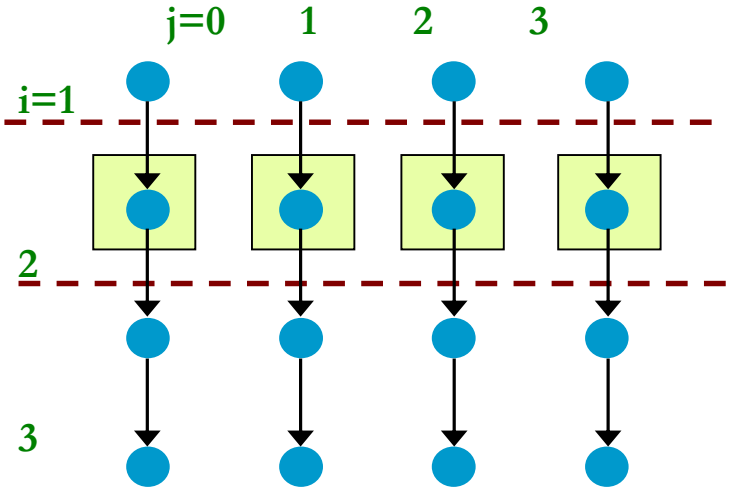
7. Tratamiento de bucles: intercambio.



Paralelización de Bucles: Optimizaciones

Ejemplo:

```
do i = 1, N-1
  do j = 0, N-1
   A(i,j) = A(i-1,j) + 1
  enddo
enddo
```

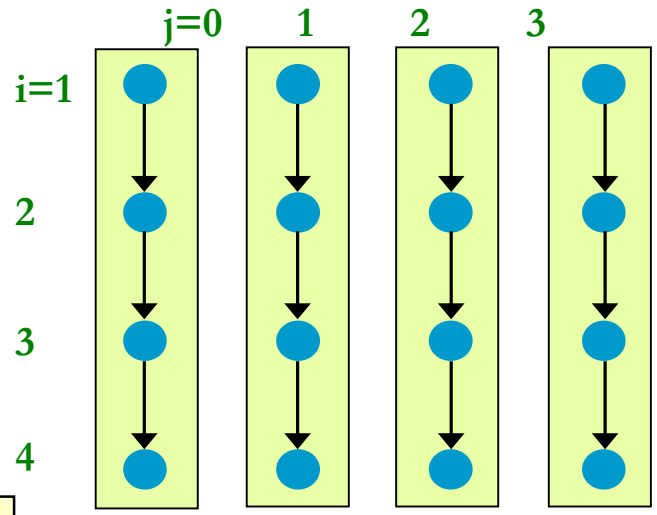



```
do i = 1, N-1
  doall j = 0, N-1
   A(i,j) = A(i-1,j) + 1
  enddoall
enddo
```

Paralelización de Bucles: Optimizaciones

Ejemplo:

```
do j = 0, N-1
  do i = 1, N-1
    A(i,j) = A(i-1,j) + 1
   enddo
enddo
```



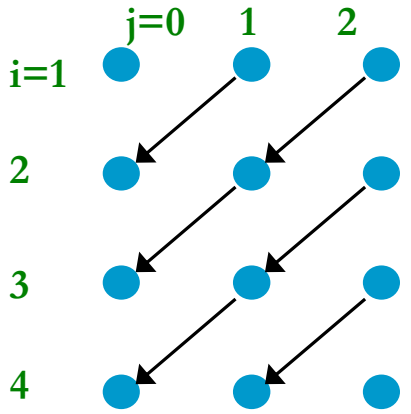

```
doall j = 0, N-1
  do i = 1, N-1
    A(i,j) = A(i-1,j) + 1
  enddo
enddoall
```

Paralelización de Bucles: Optimizaciones

8. Tratamiento de bucles: cambio de sentido.

```
do i = 1, 100
 do j = 0, 2
   A(i,j) = A(i,j) - 1
   D(i) = A(i-1,j+1) * 3
 enddo
enddo
```

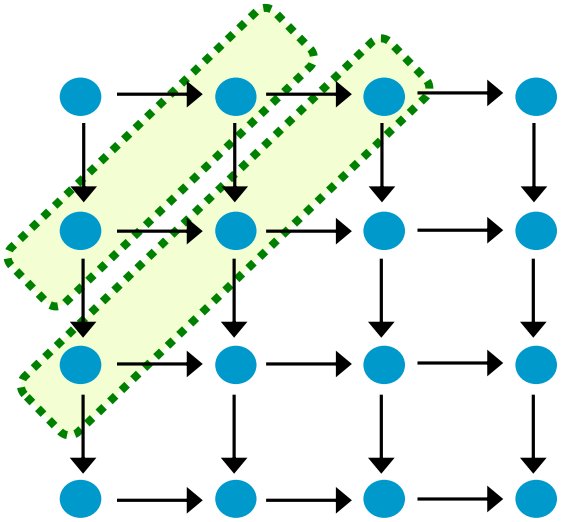
```
do j = 2, 0, -1
 doall i = 1, 100
   A(i,j) = A(i,j) - 1
   D(i) = A(i-1,j+1) * 3
 enddoall
enddo
```



Paralelización de Bucles: Optimizaciones

9. *Skew*

```
do i = 1, N
 do j = 1, N
   A(i,j) = A(i-1,j) + A(i,j-1)
 enddo
enddo
```



```
do j = 1, 2N-1
 doall i = max(1,j-N+1), min(j,N)
   A(i,j-(i-1)) = A(i-1,j-(i-1)) + A(i,j-1-(i-1))
 enddoall
enddo
```

Paralelización de Bucles: Optimizaciones

10. Otras optimizaciones típicas

Aumento del tamaño de grano y reducción del *overhead* de la paralelización.

- **Juntar dos bucles en uno (¡manteniendo la semántica!): fusión.**
- **Convertir un bucle de dos dimensiones en otro de una sola dimensión: colapso o coalescencia.**

…

Paralelización de Bucles: Optimizaciones

- ¿Cómo se reparten las iteraciones de un bucle entre los procesadores?

  Si hay tantos procesadores como iteraciones, tal vez una por procesador.

- Pero si hay menos (lo normal), hay que repartir.

  El reparto puede ser:

  estático: en tiempo de compilación.

  dinámico: en ejecución.

Paralelización de Bucles: reparto de iteraciones

- El objetivo: intentar que el tiempo de ejecución de los trozos que se reparten a cada procesador sea similar, para evitar tiempos muertos (*load balancing*).

- **En todo caso, hay que terner en cuenta las dependencias (sincronización), el tamaño de grano, la localidad de los accesos y el coste del propio reparto.**

Paralelización de Bucles: reparto de iteraciones

- **Planificación estática**

  Lo que se ejecuta en cada procesador se decide en tiempo de compilación. Es por tanto una decisión prefijada.

  Cada proceso tiene una variable local que lo identifica, pid [0..P-1].

  Dos opciones básicas: reparto consecutivo y reparto entrelazado.

Paralelización de Bucles: reparto de iteraciones

- **Consecutivo**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
principio = pid * N/P
fin = (pid+1) * N/P - 1
do i = principio, fin
   ...
enddo
```

- **Entrelazado**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

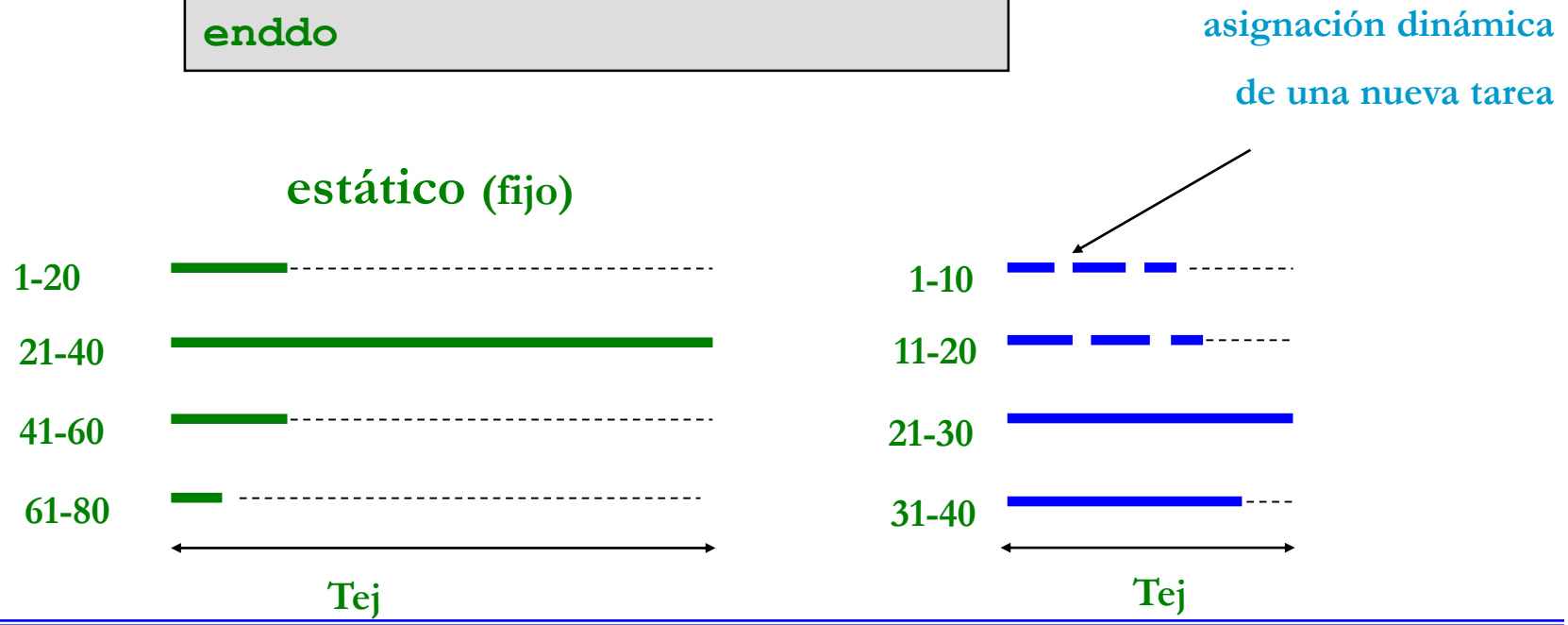
```
do i = pid, N, P
   ...
enddo
```

- No añade carga a la ejecución de los *threads*.
- Pero no asegura el equilibrio de la carga entre los procesos.
- Permite cierto control sobre la localidad de los accesos a cache.

Paralelización de Bucles: reparto de iteraciones

- Equilibrio en el reparto de carga

```
do i = 1, 80
  if (A(i) > 0) calcular()
enddo
```



Paralelización de Bucles: reparto de iteraciones

Planificación dinámica

Para intentar mantener la carga equilibrada, las tareas se van escogiendo en tiempo de ejecución de un cola de tareas. Cuando un proceso acaba con una tarea (un trozo del bucle) se asigna un nuevo trozo.

Dos opciones básicas: los trozos que se van repartiendo son de tamaño constante o son cada vez más pequeños.

Paralelización de Bucles: reparto de iteraciones

- ***Self / Chunk scheduling***

Las iteraciones se reparten una a una, o por trozos de tamaño Z.

Añade carga a la ejecución de los *threads*.
Hay que comparar ejecución y reparto.

```
LOCK (C);
  mia = i;
  i = i + Z;      Z = 1 → self
UNLOCK (C);
while (mia <= N-1)
  do j = mia, min(mia+Z-1, N-1)
    ...
  enddo
  LOCK (C)
   mia = i;
   i = i + Z;
  UNLOCK (C)
endwhile
```

Paralelización de Bucles: reparto de iteraciones

- ***Guided / Trapezoidal***

  Los trozos de bucle que se reparten son cada vez más pequeños según nos acercamos al final.

  - ***Guided*: parte proporcional de lo que queda por ejecutar:**

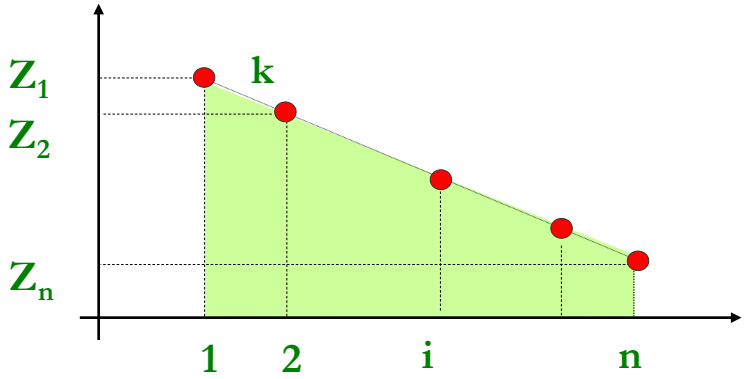
$$Z_s = (N - i) / P \qquad \text{(entero superior)}$$

que equivale a:

$$Z_i = Z_{i-1} (1 - 1/P)$$

Paralelización de Bucles: reparto de iteraciones

- ***Trapezoidal*: reduciendo el trozo anterior en una constante: $Z_i = Z_{i-1} - k$**



op. de planificación

$$\sum_{s=1}^{n} Z_s = \frac{Z_1 + Z_n}{2} n = \frac{Z_1 + Z_n}{2}\left(\frac{Z_1 - Z_n}{k} + 1\right) = N \quad \rightarrow \quad k = \frac{Z_1^2 - Z_n^2}{2N - (Z_1 + Z_n)}$$

Paralelización de Bucles: reparto de iteraciones

- En general, el reparto dinámico busca un mejor equilibrio en el reparto de carga, pero:

  - hay que considerar la carga que se añade (*overhead*), en relación al coste de las tareas que se asignan.

  - hay que considerar la localidad en los accesos a los datos y los posibles problemas de falsa compartición.

Paralelización de Bucles: reparto de iteraciones

- Ejemplo de reparto (1.000 iteraciones, 4 procesadores):
  - ***chunk*** **(Z = 100)**

    100 100 100 100 100 100 100 100 100 100 **(10)**

  - ***guided***

    quedan: **1000 750 562 421 … 17 12 9 6 4 3 2 1**

    se reparten: **250 188 141 106 … 5 3 3 2 1 1 1 1** **(22)**

  - ***trapezoidal*** **($Z_1 = 76$, $Z_n = 4$ >> k = 3)**

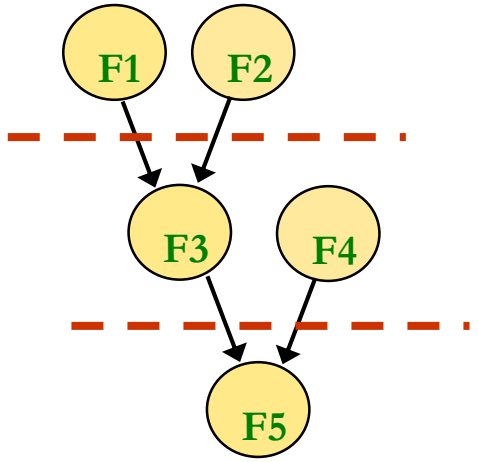
    76 73 70 67 … 16 13 10 7 4 **(25)**

Paralelización de Bucles: secciones paralelas

- Paralelismo a nivel de procedimiento o función:

  - **modelo Fork / Join**

  - ***Parallel sections***




```
Fork
   F1
   F2
Join
Fork
   F3
   F4
Join
F5
```

```
doall k = 0, 1
   if (pid=0) then F1
   if (pid=1) then F2
endoall
[barrera]
doall k = 0, 1
   if (pid=0) then F3
   if (pid=1) then F4
endoall
F5
```

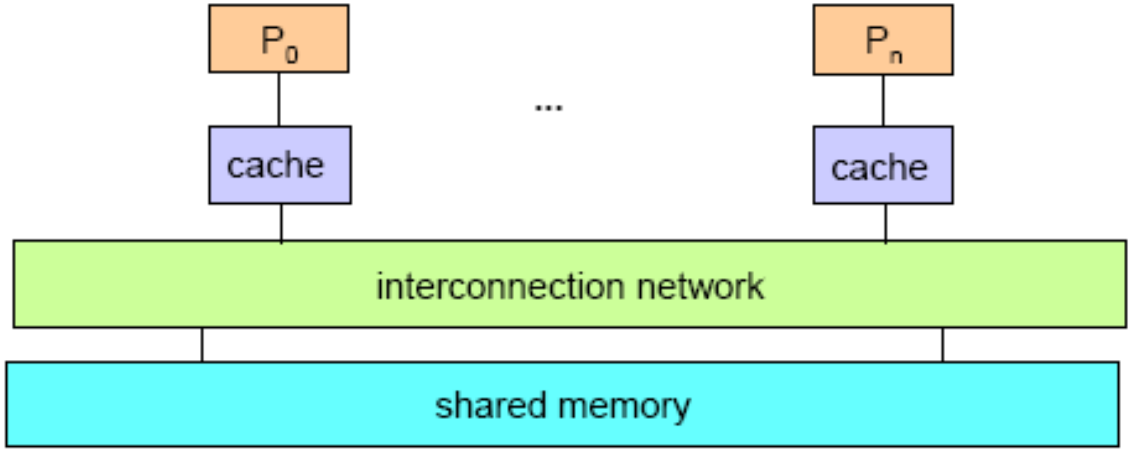
Contenidos

- **Modelos de Programación Paralela**
  - **Características distintivas de cada modelo**
  - **Ejecución en Threads o Procesos**
  - **Compiladores Paralelizadores**
- **Paralelización de bucles**
  - **Análisis de dependencias.**
  - **Criterios de paralelizacion.**
  - **Principales optimizaciones**
- **Programación de multiprocesadores mediante compiladores paralelos (OpenMP)**
- **Programación mediante paso de mensajes (MPI)**

Compilador Paralelo Open MP

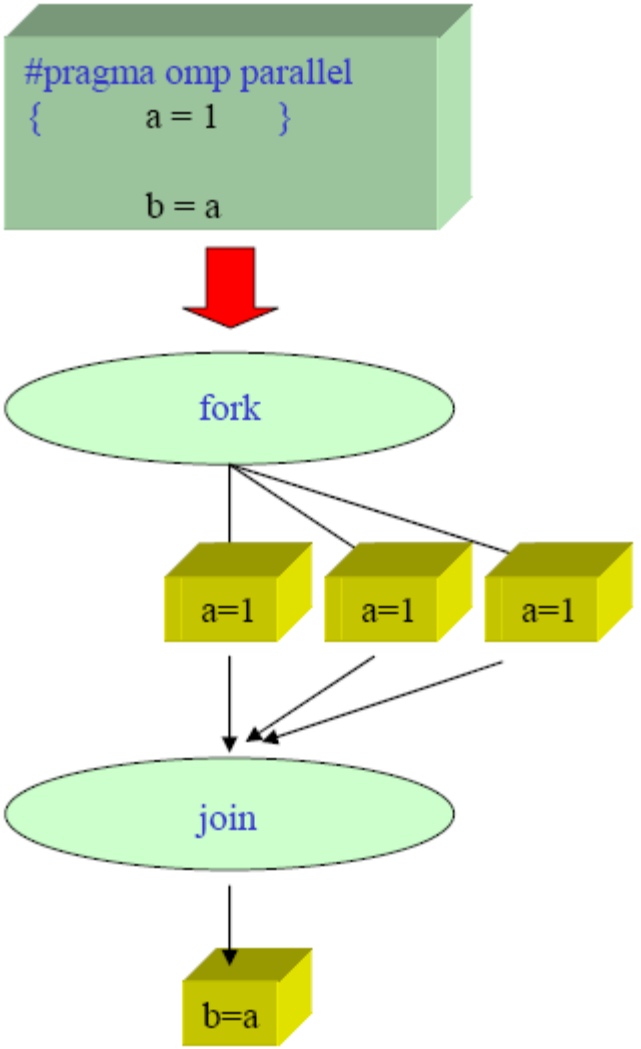
Se usa en la programación de computadores paralelo con un espacio de direcciones compartido.

- **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 (Aunque OpenMP puede usarse por debajo de una herramienta automática de paralelización)**



Modelo de Funcionamiento

- **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 la última hebra 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.

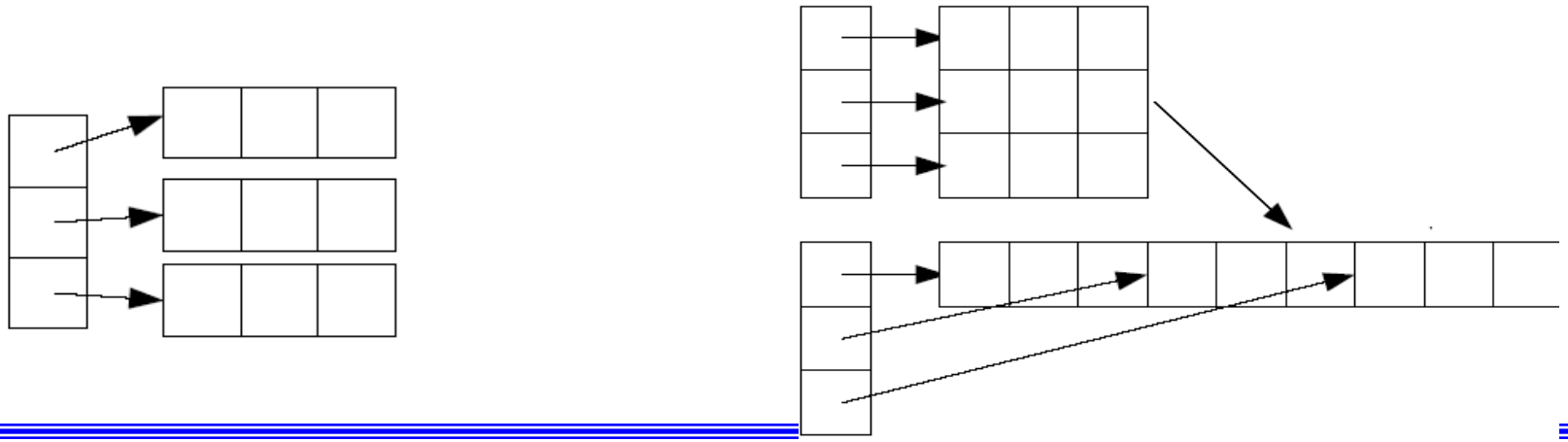
Consideraciones de Rendimiento en OpenMP (2)

Localidad

La localidad es aprovechada por las cachés

La memoria reservada para un vector es contigua en memoria, mientras que la memoria reservada para dos vectores que forman una matriz no necesariamente lo es.

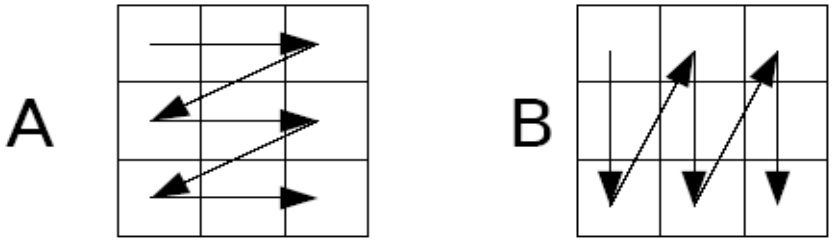
Es mejor reservar una matriz como un vector, que reservar una matriz como una serie de vectores indexados.



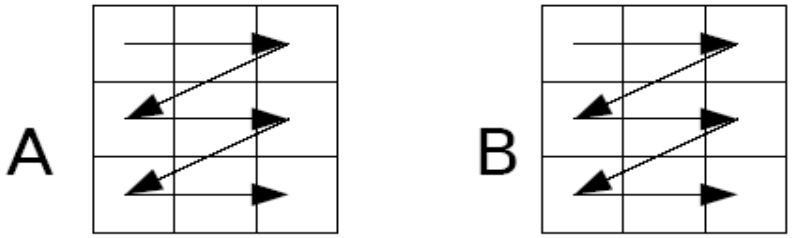
Consideraciones de Rendimiento en OpenMP (3)

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 (3)

Localidad

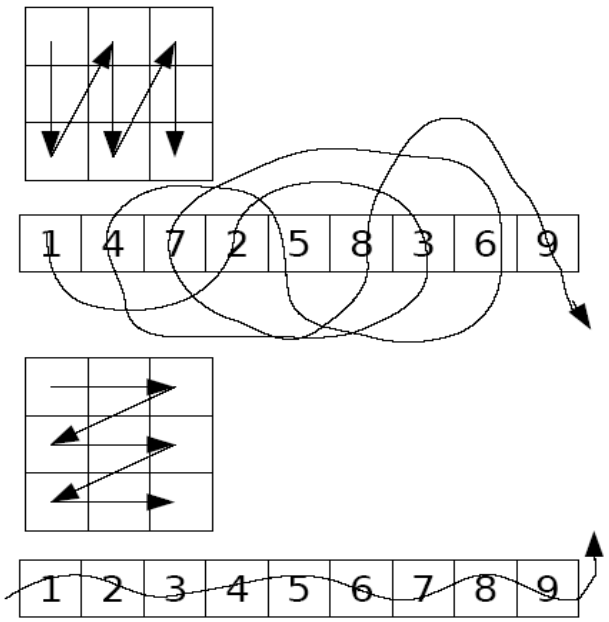
También es importante acceder a los elementos en memoria siguiendo el mismo orden en el que se encuentran en esta.

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)

Localidad

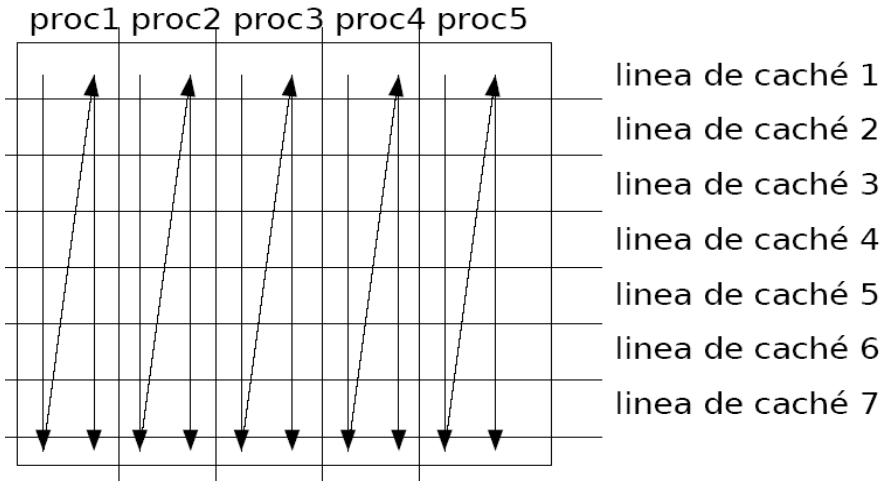
En los sistemas multiprocesador, cuando un procesador requiere unos datos, estos se copian a su caché y esa línea se marca como sucia en las cachés de los demás procesadores, para que dos procesadores o más no puedan estar trabajando sobre versiones distintas de los mismos datos.

Si distintos procesos intentan acceder a las 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 (5)

Localidad

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 (5)

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.**

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**

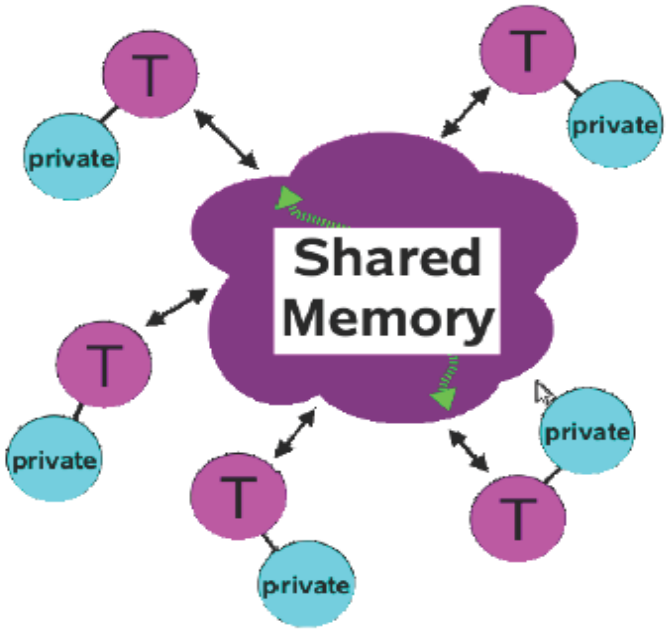
```
export OMP_NUM_THREADS=2
./a.out
Hello world
Hello world

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.

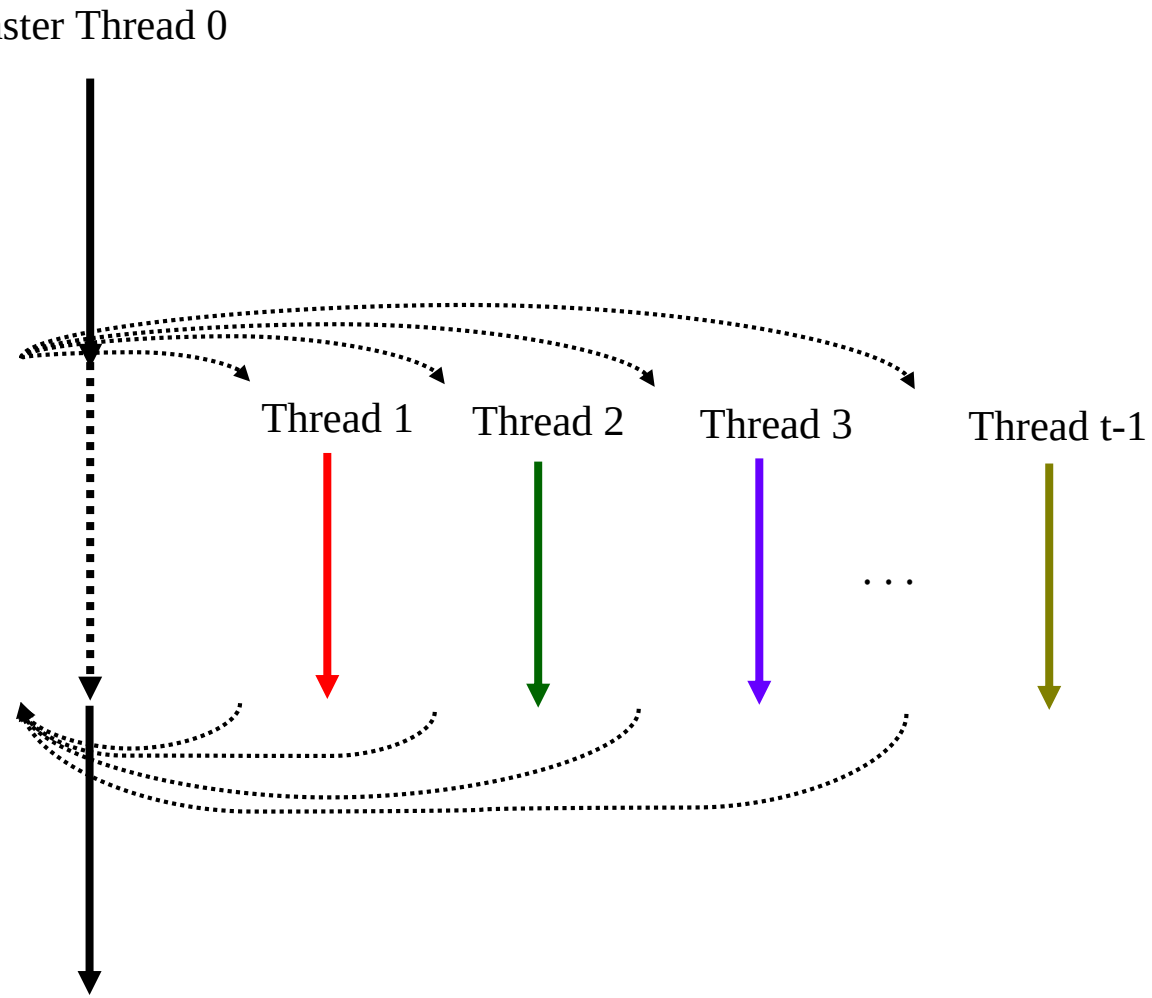
Términos Básicos

- 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

Programa

```
Región Secuencial
{
  .....
  .....
}

Región Paralela
{
  .....
  .....
}

Región Secuencial
{
  .....
  .....
  .....
}
```



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`)

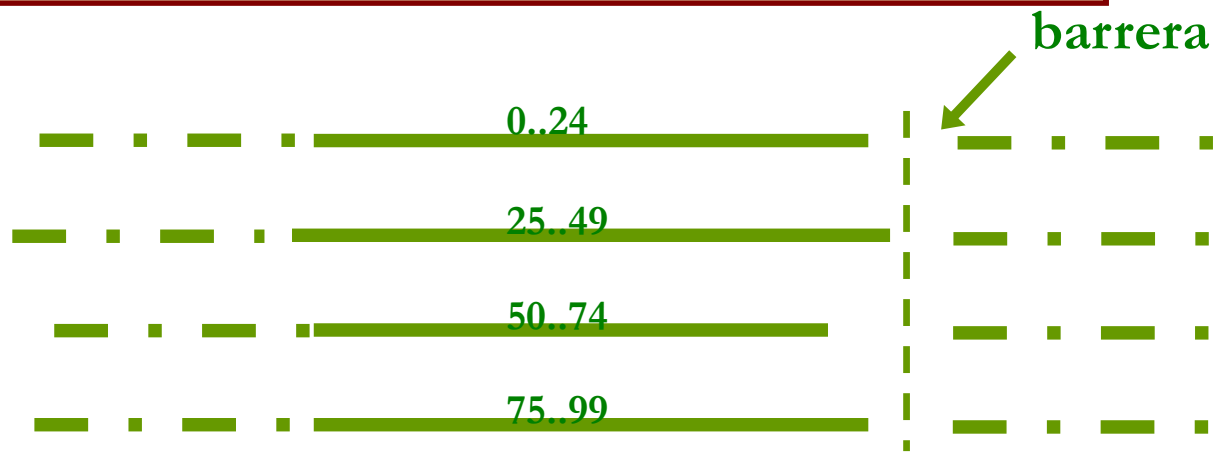
1 Directiva `for`

```
#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

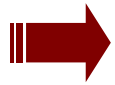
-- reordenación de instrucciones

-- 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.

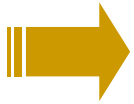
La directiva `parallel for` implica la generación de P *threads*, que se repartirán la ejecución del bucle.

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;
 }
```



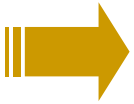
```
#pragma omp parallel for
   private (j,X)

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 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 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 parallel1 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.

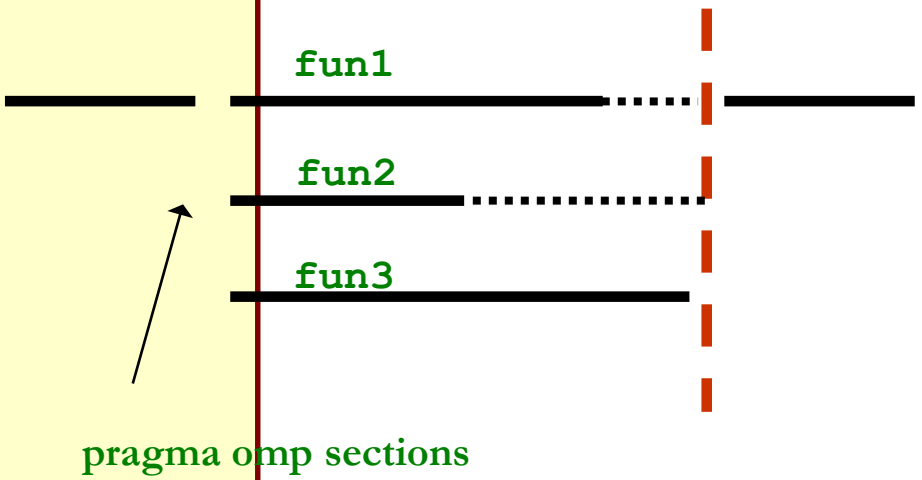
Cada sección paralela es ejecutada por un sólo *thread*, y cada *thread* ejecuta alguna sección o ninguna.

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.]
```

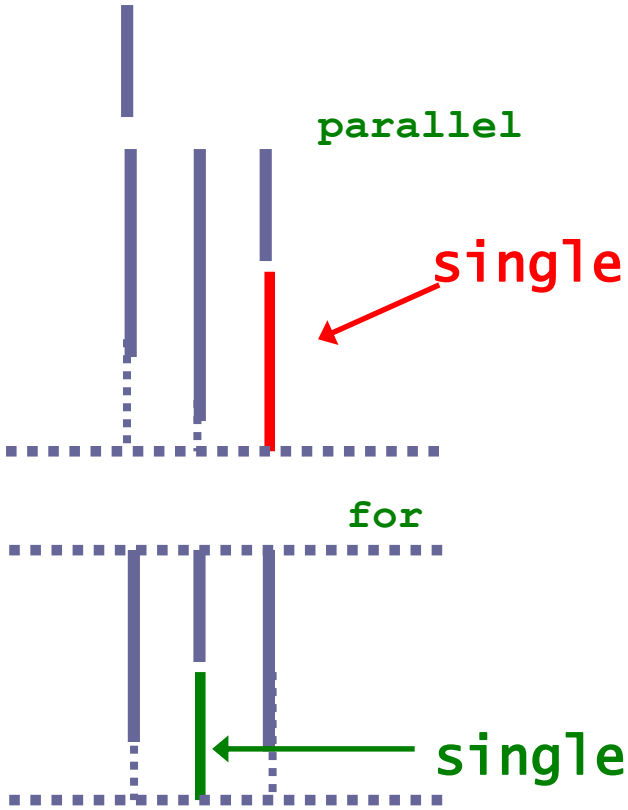
Cláusulas: `(first)private, nowait`

`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*.

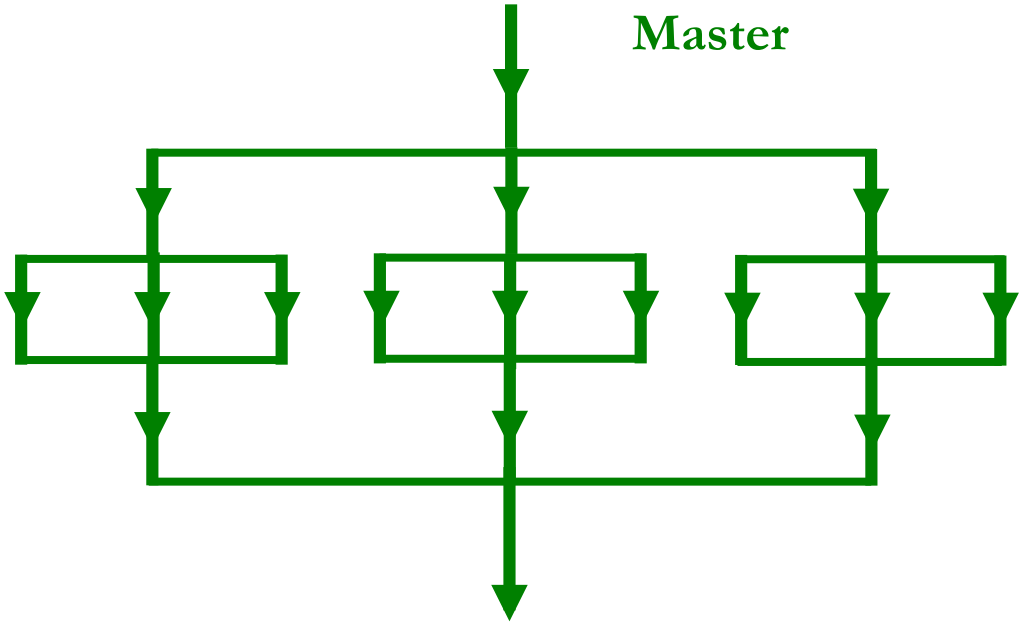
- Puede hacerse que el número de *threads* sea dinámico, en función del número de procesadores disponibles en cada instante:

```
> 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/