3. Sistemas con más de un procesador: arquitecturas de memoria compartida

Apartado A

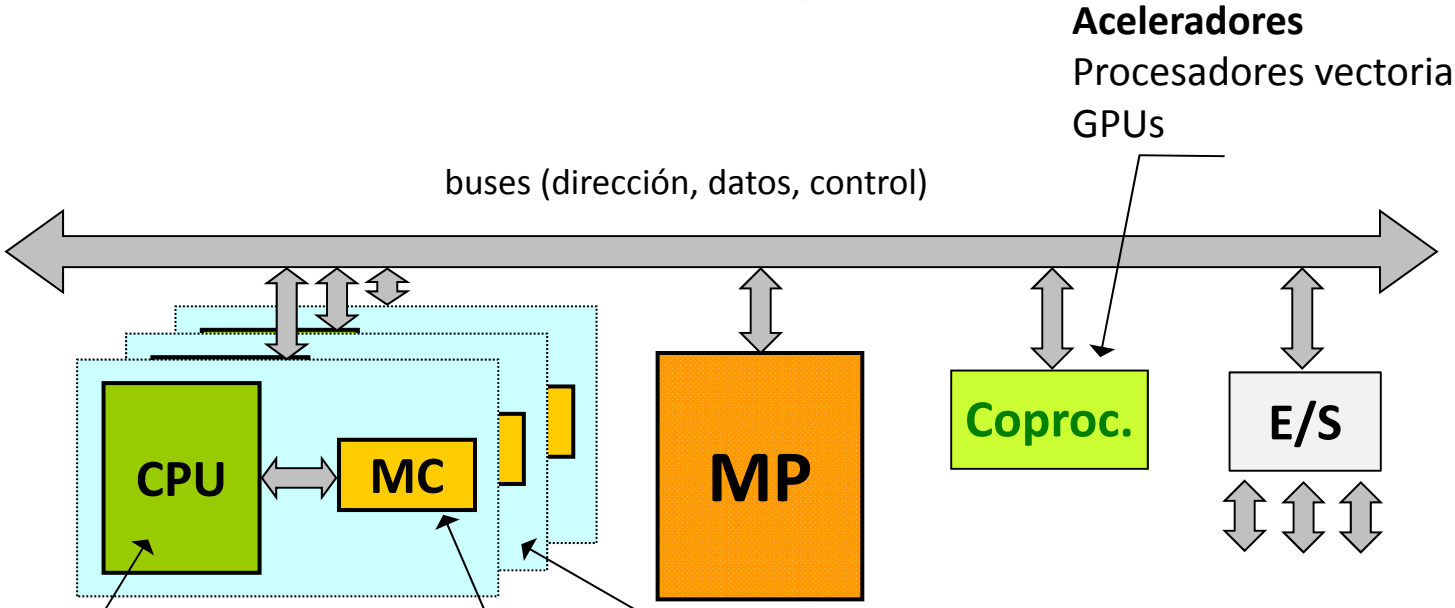
Conceptos básicos. Clasificación de Flynn.

Rendimiento. Leyes de Amdahl y Gustafson.

Estrategias principales para el reparto de trabajo.

Sincronización de procesos. Secciones críticas y barreras

Aumento de la velocidad de proceso



Segmentación (ILP)
Técnicas de compilación

Proc. superescalares
Desorden
Especulación

Memoria cache

Más de un núcleo/procesador

Multiprocesadores SMP

Multicomputadores MPP
C*luster*-s

- Estrategias básicas para acelerar la ejecución de los procesadores:
  - **memoria cache**
  - **segmentación**
    - multiciclo, compilador
    - desorden
    - especulación

    [Desorden y especulación en la asignatura PAR]

Todo ello permite aumentar la velocidad de cálculo de un procesador hasta los **10 Gflop/s**.

Esa velocidad sigue siendo **escasa** en algunos **ámbitos**: genética, meteorología, astrofísica, aeronaútica, *big data*, diseño de materiales, inteligencia artificial, farmacia, *machine learning*…

- ¿Cómo conseguir velocidades superiores ... Teraflop/s, Petaflop/s, Hexaflop/s...?

Recuerda:

Giga (**G**) 10^{9}	Tera (**T**) 10^{12}	Peta (**P**) 10^{15}	Exa (**E**) 10^{18}
nano (**n**) 10^{-9}	pico (**p**) 10^{-12}	femto (**f**) 10^{-15}	atto (**a**) 10^{-18}

1 Procesadores más rápidos.

Esta vía está agotada en este momento, por lo menos desde el punto de vista de la arquitectura del sistema.

2 Gracias a la tecnología electrónica se pueden integrar miles de millones de transistores en un chip. ¿Qué podemos hacer con ese número de **transistores**?

- Se han trabajado estas tres vías:
  - Unidades de control más complejas: desorden en la segmentación, ejecución especulativa, etc.
  - Más memoria cache en el chip, repartida en tres niveles.
  - Varios **núcleos (*core*)** en el chip.

- Hoy en día todos los procesadores son *multicore*, esto es, tienen más de una unidad de proceso dentro del chip
  - gama baja: 2 - 8 *core*
  - gama alta: 24 - 64... *core*

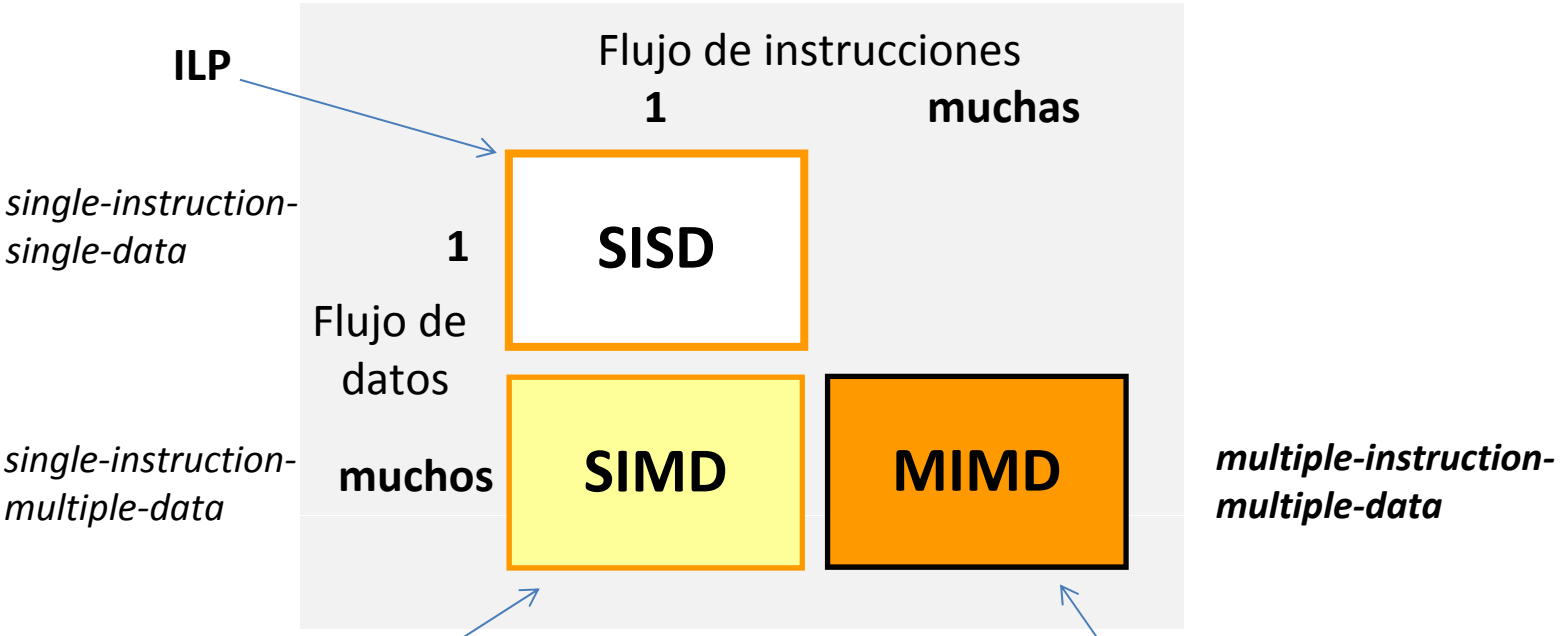
- Pueden utilizarse P procesadores o núcleos:
  - Para ejecutar P aplicaciones diferentes.
  - Para ejecutar una aplicación **P veces más rápido**.
  - Para ejecutar una aplicación **P veces "más grande"** en el mismo tiempo que la aplicación original

- Arquitecturas paralelas

Un computador paralelo es un máquina formada por muchas unidades de proceso (**procesadores**), que **trabajan conjuntamente para resolver problemas complejos (grandes)**.

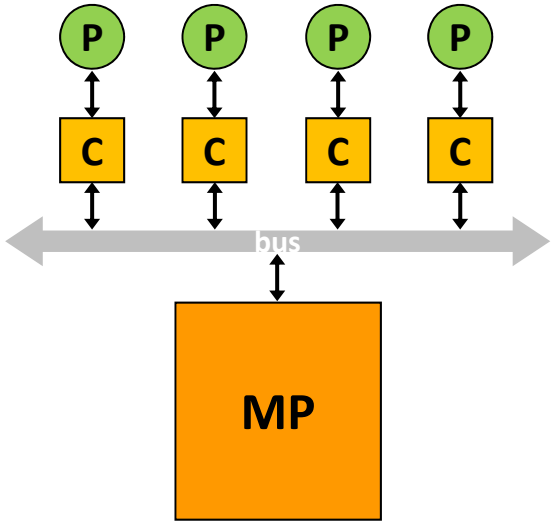
Mayor capacidad de cálculo ⟷ Nuevas aplicaciones

Clasificación de **Flynn**



Sistemas **MIMD**: dos arquitecturas básicas

1. **Memoria compartida** (*shared memory*)



núcleos + cache privada

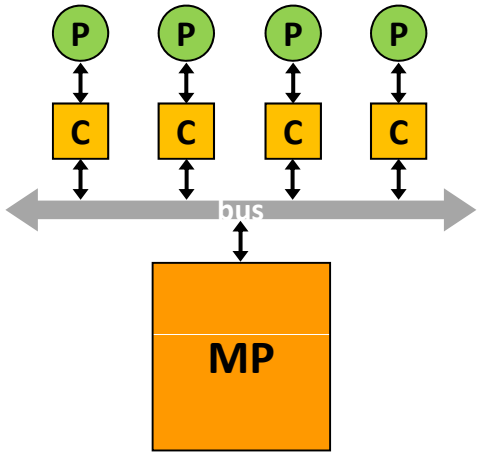
red de comunicación (bus)

memoria principal

Los núcleos/procesadores utilizan una cache privada, pero **comparten la memoria principal**.

Sistemas **MIMD**: dos arquitecturas básicas

1. **Memoria compartida** (*shared memory*)



- **Un espacio de direccionamiento único**: los procesadores pueden acceder a todas las direcciones de memoria.

  `LD A` > es la misma variable en todos los núcleos

  La latencia del acceso a memoria es la misma para cualquier palabra y núcleo (accesos "simétricos").

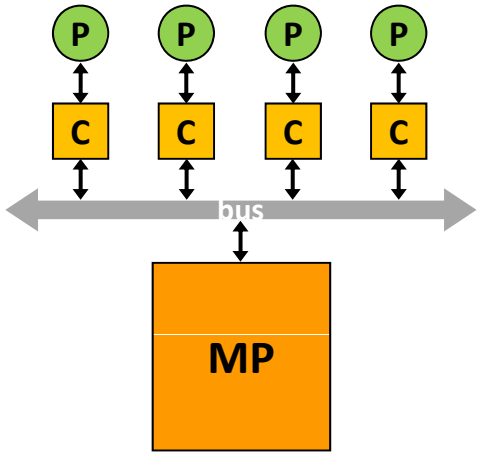
- La **comunicación** entre procesos se realiza mediante **variables compartidas**.

```
P0:  ... ST A
P1:  ...        LD A
```

- Los procesadores y la memoria se comparten mendiante un **bus**.

Sistemas **MIMD**: dos arquitecturas básicas

1. **Memoria compartida** (*shared memory*)



- El bus es una red "centralizada". El ancho de banda debe repartirse entre todos los núcleos.

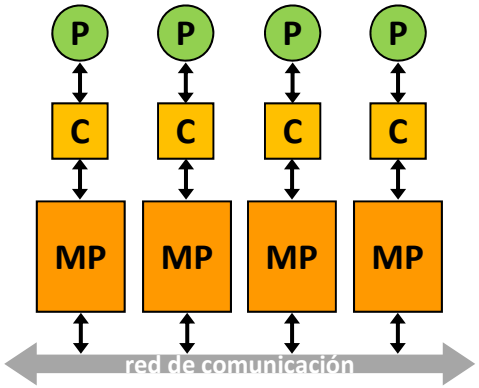
  En caso de muchos núcleos, **el bus puede saturarse**: no puede gestionar todas las peticiones de acceso a memoria.

- En general, el número de núcleos no es grande; a pesar de que el ancho de banda sea elevado, no pueden conectarse miles de núcleos con memoria a través de un bus.

Terminología: multiprocesador, **SMP** (*symmetric multiprocessor*), ***multicore***.

Sistemas **MIMD**: dos arquitecturas básicas

2. Memoria privada (*distributed memory*) (en la asignatura SCP)



- La memoria principal es **privada**: una por procesador. No existen variables compartidas.

  P0: LD A P1: LD A

  Son variables distintas en P0 y P1.

- **Comunicación** entre procesos**: paso de mensajes**.
- Redes de comunicación: mallas, toros, árboles...
- En general, **cientos, miles de procesadores**.

Terminología: multicomputadores, **MPP** (*massively parallel processors*).
Cluster-s (centros de datos, Google, Amazon...).

TOP500

#1 Fugaku (Fujitsu)

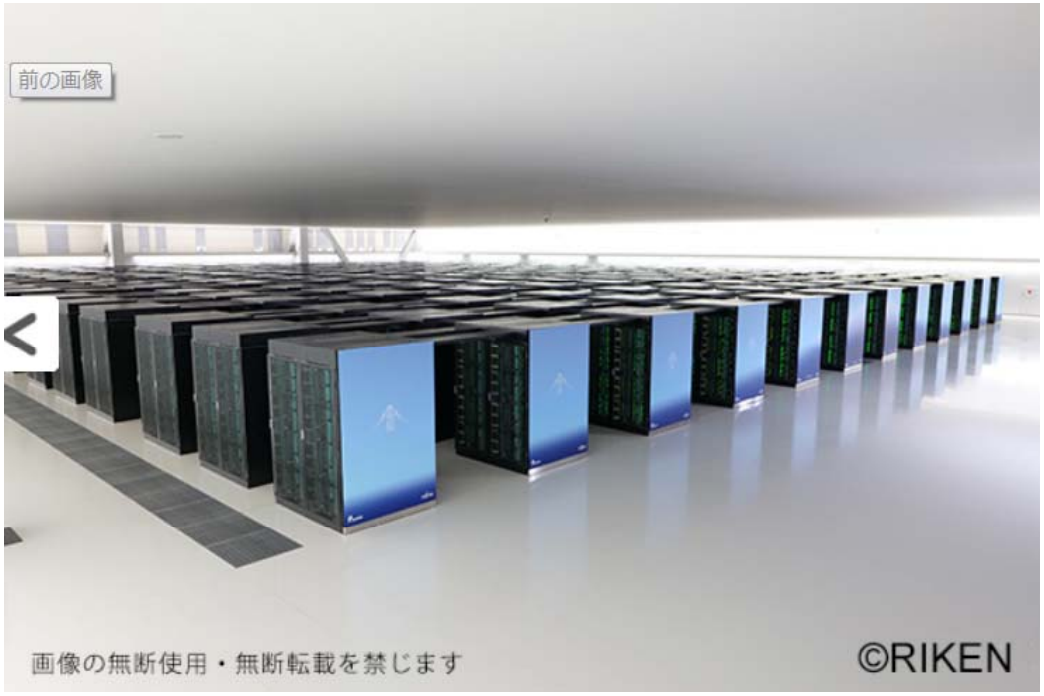
A64FX ARM (48C, 2.2GHz)

7.299.072 *core*

415,5 Pflop/s

Tofu interconnect (toro 6D)

Instrucciones vectoriales de 512 bits



$415 \times 10^{15} = 4{,}15 \times 10^{17}$ operaciones/segundo

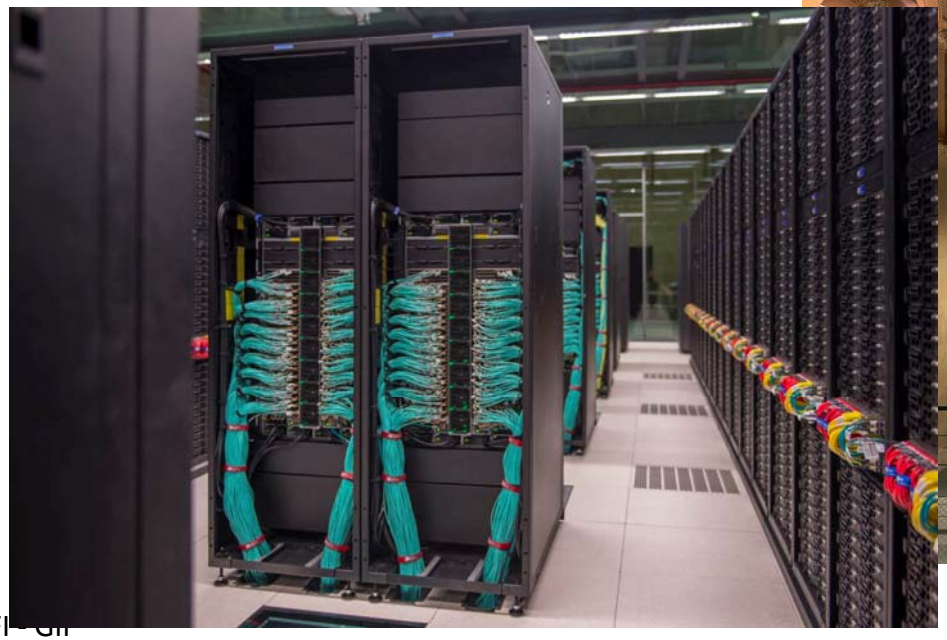
Edad del universo = 13.700 M años = $4{,}3 \times 10^{17}$ s

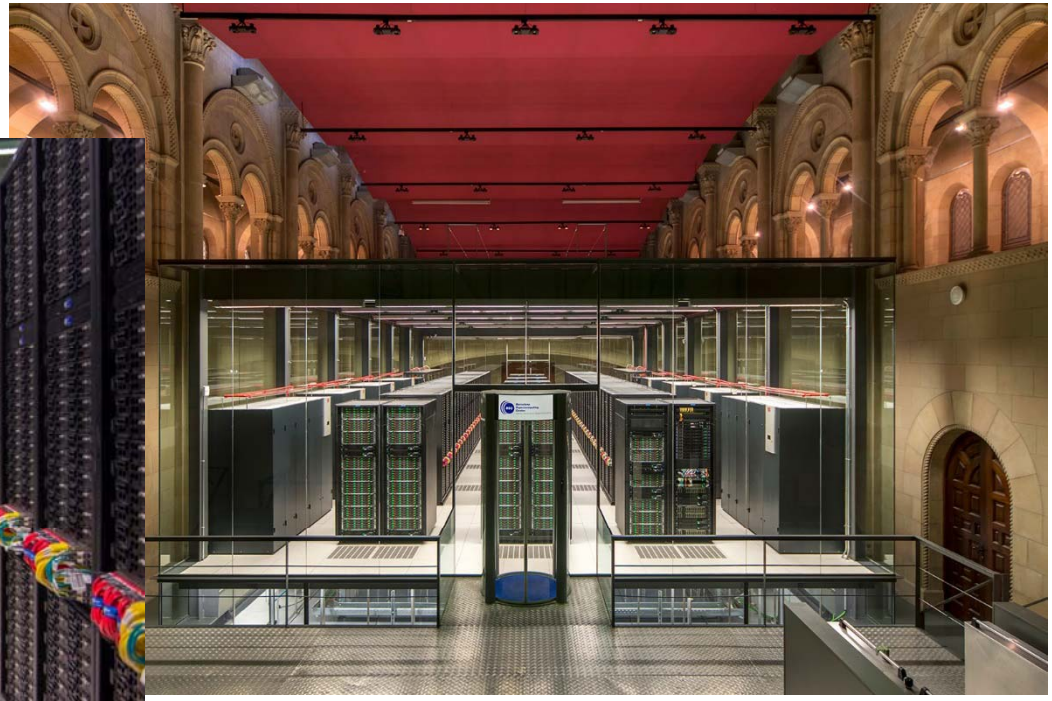
TOP500

#37 MareNostrum (Lenovo) **6,5 Pflop/s**

Xeon 8160 - 24 nukleo (2,1 GHz) **153.216** núcleos

Intel Omni-Path





- Ahora bien, ¿es sencillo utilizar P procesadores para ejecutar más rápidamente una aplicación? En general, no.

  Aspectos a tener en cuenta:

  - ¿Cómo repartir un algoritmo entre P procesos? **¿Todo el código puede ejecutarse en paralelo**? ¿Se consigue ejecutar una aplicación P veces más rápido?
  - **El reparto de trabajo está equilibrado**, o, por ejemplo, la carga de trabajo se ha repartido "80% - 20%" (*load balancing*)?
  - ¿Dónde están los datos, en nuestra cache? Si los datos están en otra cache, ¿cómo se mantiene la **coherencia**?
  - ¿Los procesos son independientes? ¿Se deben **sincronizar** en la ejecución de la aplicación? ¿Cómo?

- En esta asignatura sólo trataremos tres problemas principales en los sistemas SMP. Una vez entendidos, podremos empezar a programar aplicaciones paralelas.

  0 Cómo calcular el rendimiento (velocidad de cálculo).

  1 Influencia del código que se ejecuta en serie.

  2 Utilización de las variables compartidas y sincronización entre procesos.

  3 Estrategias de reparto del trabajo (iteraciones de un bucle) entre procesos paralelos.

Parámetros para expresar el rendimiento

1. Tiempo de ejecución

- ejecución serie: T_s
- ejecución paralelo: T_p Ideal: $T_p = T_s / P$

Hay muchos factores que limitan el rendimiento:

a. Sólo se puede ejecutar una **fracción** de código (f) en paralelo.

b. Hay que asumir una **sobrecarga** en la ejecución en paralelo (sincronización entre procesos).

c. El reparto de trabajo no va a ser "perfecto" y se generarán **tiempos de espera** en la ejecución en paralelo.

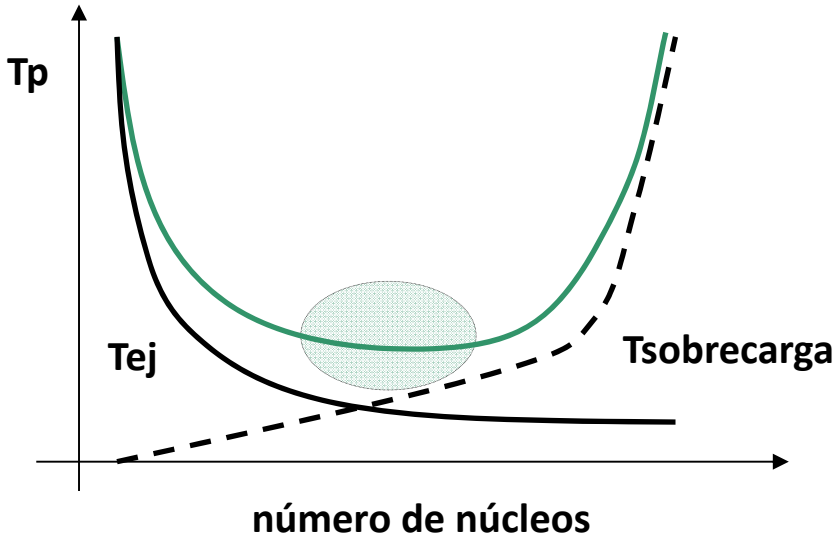
$$T_p = f\,\boxed{T_s / P} + (1\text{-}f)\,T_s + T_{sobrecarga} + T_{espera}$$

- Parámetros para expresar el rendimiento

1. **Tiempo de ejecución:** sobrecarga (f = 1)

$$T_p = T_s / P + [\, T_{sobrecarga} + T_{espera} \,]$$

En general, la sobrecarga es función de P, lo que puede limitar el número de núcleos que se pueden utilizar.



- Un apunte en relación al tiempo de ejecución.

  La memoria cache es el principal acelerador en un procesador. Para que se eficiente los datos (bloques) que se cargan deben reutilizarse.

  Ejemplo: en un bloque de 8 palabras están los elementos [0..7] de **A**

  - acceso en serie: un fallo y 7 aciertos
  - repartida en 8 núcleos: ¡todos los accesos son fallos! ¡la ejecución será más lenta!

  Para que la ejecución paralela sea eficiente hay que analizar muy bien **cómo se reparten los datos**.

- Parámetros para expresar el rendimiento

2. **Factor de aceleración** (*speed-up*). Se comparan dos tiempos de ejecución: serie y paralelo.

   fa = T_s / T_p ideal: P

   bueno: crecimiento **lineal** con P

3. **Eficiencia** (*efficiency*). Porcentaje (entre 0 y 1) del factor de aceleración conseguido respecto al teórico (P)

   ef= af / P ideal: 1

   bueno: independiente de P

- Parámetros para expresar el rendimiento

4. **Escalabilidad** (*scalability*). Analiza cómo evoluciona la eficiencia con el número de procesadores y el tamaño del problema.
   - escalabilidad **fuerte** (*strong*)

     >> ¿Se mantiene la eficiencia al aumentar P, sin aumentar el tamaño del problema (N), esto es, cuando los procesos paralelos tienen menos carga de trabajo?

     Si es así, la aplicación tiene una fuerte escalabilidad:

     $$ef(\mathbf{k \times P}, N) \cong ef(P, N) \qquad [\text{ Amdhal }]$$

     Es una muy buena característica, dado que brinda la oportunidad de tener un número elevador de procesadores (por desgracia, pocas aplicaciones tienen esta característica).

- Parámetros para expresar el rendimiento

4. **Escalabilidad** (*scalability*). Analiza cómo evoluciona la eficiencia con el número de procesadores y el tamaño del problema.

   - escalabilidad **débil** (*weak*)

     >> En este caso, al aumentar el número de procesadores, la eficiencia se mantiene si se aumenta el tamaño del problema.

     $$ef\ (k1 \times P, \mathbf{k2 \times N}) \cong ef\ (P, N) \qquad [\ \text{Gustafson}\ ]$$

     Esta característica también es buena, dado que permite un uso eficiente de muchos procesadores ejecutando aplicaciones más grandes (más iteraciones de una simulación, con mayor número de datos…). Muchas aplicaciones científicas tienen este comportamiento.

- Siendo el objetivo la velocidad de cálculo, tenemos dos opciones:

  1. **Ejecutar en paralelo el mismo programa serie**.
  2. Ejecutar en paralelo un **programa más grande** (más iteraciones de una simulación, con un mayor número de datos de entrada…) en el mismo tiempo que el programa original

En ambos casos, seguramente sólo se podrá ejecutar una fracción del código original en paralelo.

En general, la parte que debe ejecutarse en serie tiene un efecto negativo muy importante en el rendimiento.

1 Ejecución del mismo programa serie.

Ley de **Amdahl**.

En la mayoría de los casos, no puede ejecutarse todo el código de forma paralela (dependencias de datos, sincronización…).
En general, una parte se ejecutará en paralelo y otra en serie:

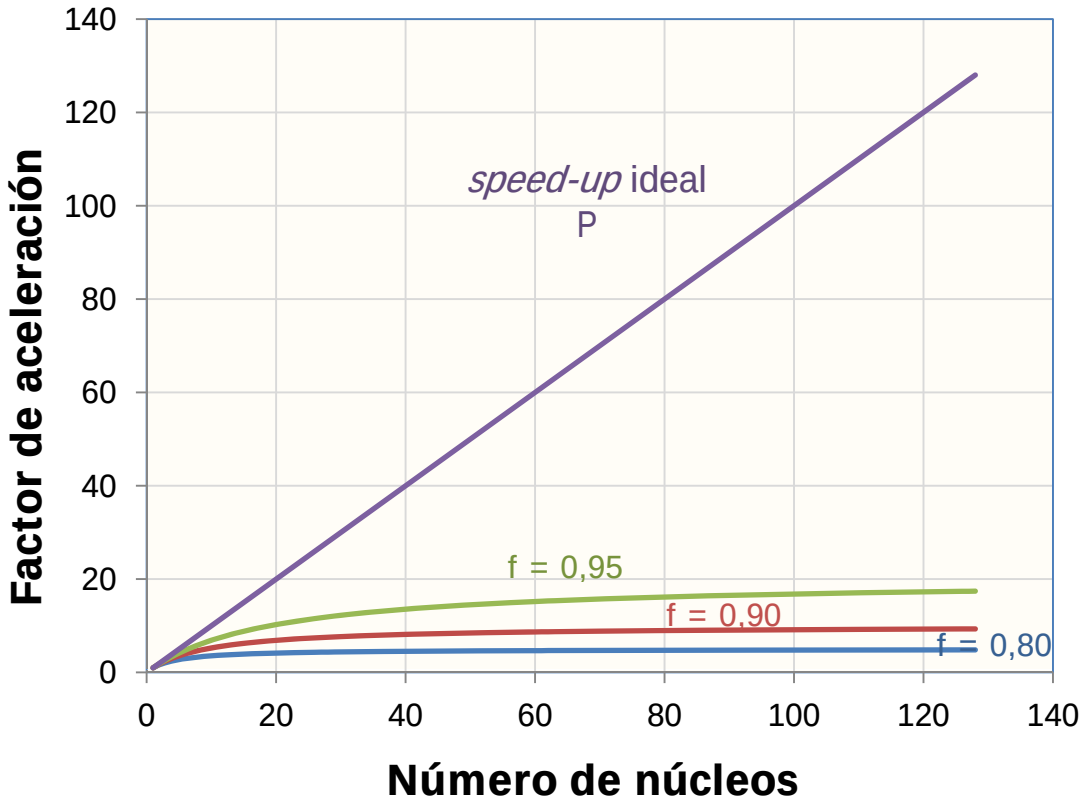
$$T_{sp} = f\,T_p + (1-f)\,T_s = f\,T_s/P + (1-f)\,T_s$$

Por tanto, sin tener en cuenta otro tipo de problemas, éste será el factor de aceleración que se conseguirá (ley de **Amdahl**):

$$fa = T_s / T_{sp} = T_s / [f\,T_s/P + (1-f)\,T_s]$$

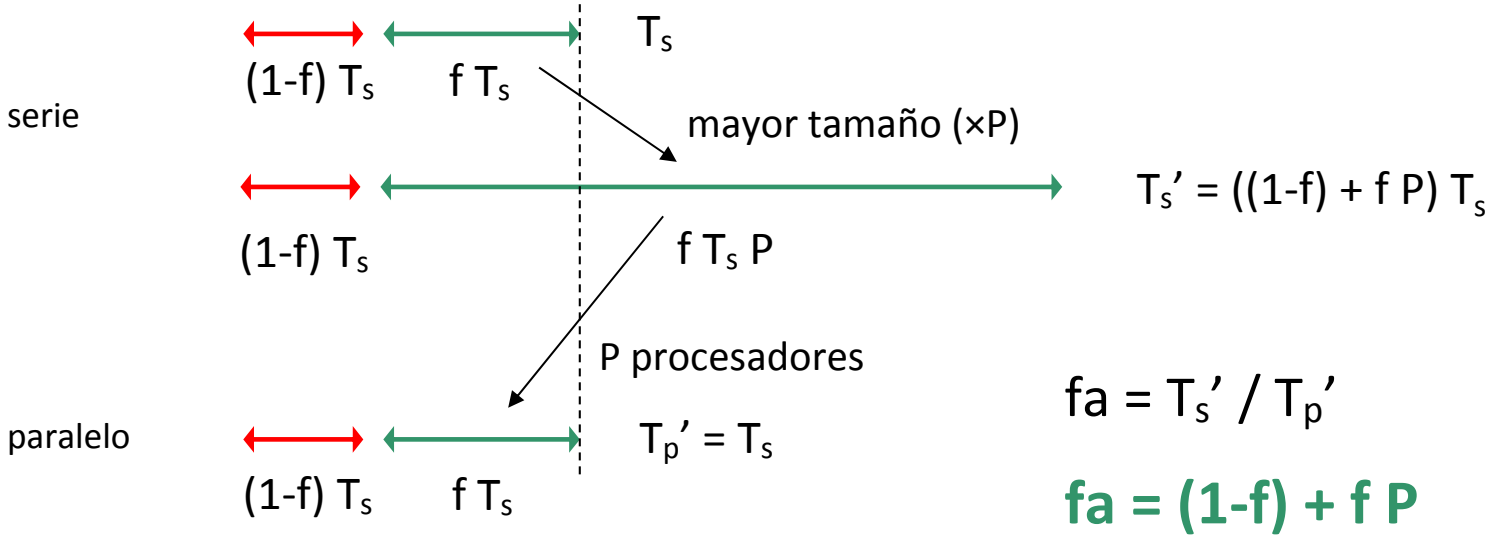
$$fa = P / [f + (1-f)\,P] \qquad \text{límite} \rightarrow 1 / (1-f) \;\; !!$$

Ley de **Amdahl**: **fa** = P / (f + (1-f) P) → 1 / (1-f)



2 Ejecución de un programa más grande en el tiempo del programa original. Ley de **Gustafson**.

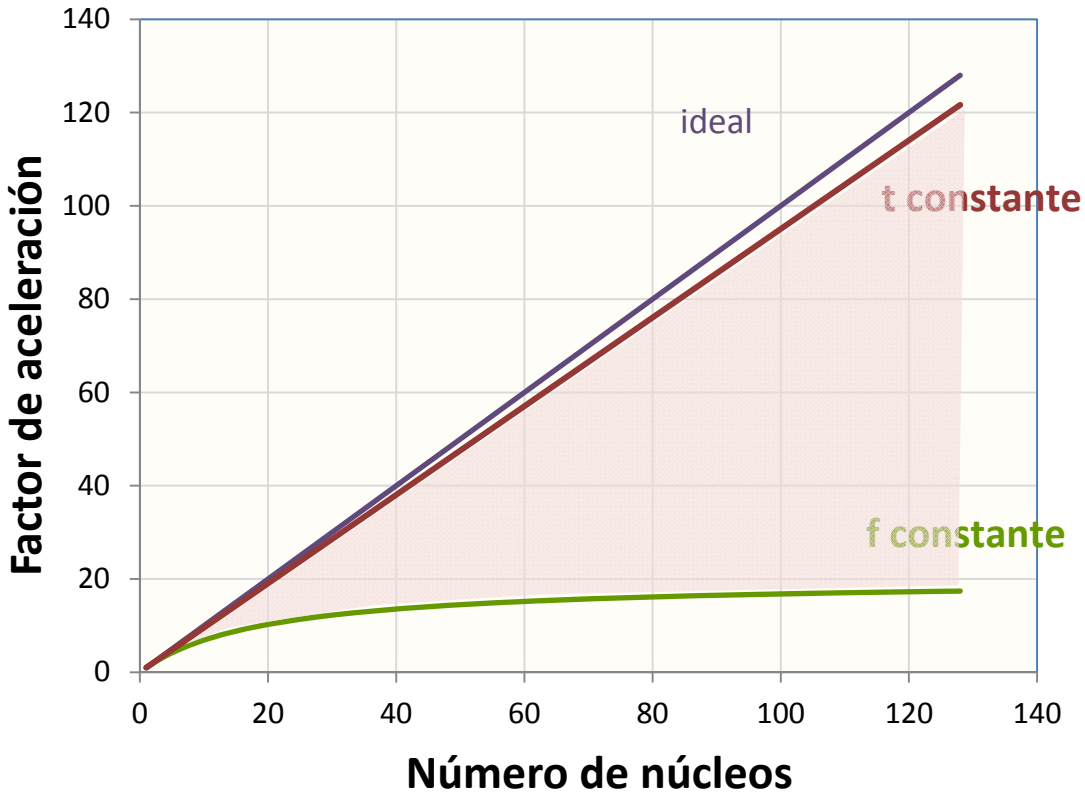
El paralelismo se utiliza también para **ejecutar tareas de mayor tamaño**. En general, en estos casos, la fracción de código que debe ejecutarse en serie no aumenta con el tamaño.



$fa = T_s' / T_p'$

$fa = (1-f) + f\ P$

Ley de **Gustafson** $fa = (1-f) + f\,P$



Resumen

Salvo en los casos más sencillos, al ejecutar un programa en paralelo, una fracción del programa debe ejecutarse en serie.

Si la fracción serie se mantiene constante con el tamaño del problema, el nivel de paralelismo que se puede conseguir está limitado, de acuerdo a la ley de Amdhal: 1 / (1-f).

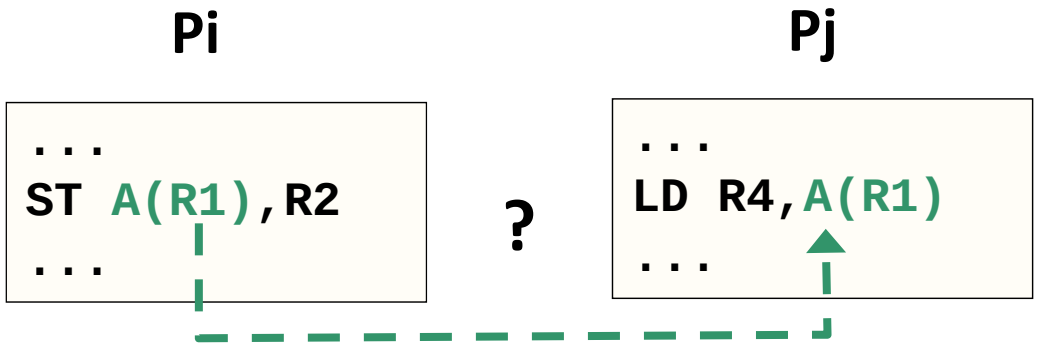
Sin embargo, en muchas aplicaciones, la fracción de código que debe ejecutarse en serie no aumenta al aumentar el tamaño del problema: ley de Gustafson. En estos casos, se utiliza el paralelismo para resolver problemas más grandes, que no pueden ejecutarse en serie (tiempos de ejecución muy elevados).

Sin tener en cuenta otros problemas: $T_p = f\,T_s/P + (1-f)\,T_s$

- En los sistemas SMP, los procesos paralelos comparten la memoria: **el intercambio de datos se realiza mediante el uso de variables compartidas**.

  Las variables privadas se pueden utilizar sin problemas, pero hay que tener cuidado al utilizar variables compartidas. Obviamente, **la ejecución paralela debe obtener los mismos resultados que la ejecución serie**.

  Por ello, en algunos momentos habrá que sincronizar la ejecución de los procesos paralelos. Por ejemplo, ¿qué debe leer Pj en este caso?



- La sincronización puede ser de tres formas:
  - **Secciones críticas.** Fragmentos especiales del código paralelo; todos los procesadores ejecutan esa parte de código pero de **uno en uno**, nunca en paralelo (exclusión mutua, *mutual exclusion*).
  - Eventos (*events*).
    - Punto a punto (de un proceso a otro): el dato que necesitas está listo (esquemas productor/consumidor).
      Es suficiente utilizar un *flag* (una variable binaria compartida) : está / no está.
    - **Barreras** (*barrier*). **Sincronización entre todos los procesos**.
      Todos los prcesos quedan esperando en un punto del programa, hasta que todos los procesos lleguen a ese punto.

- Ejemplo:
  suma de los elementos
  de un vector

```
for (i=0; i<N; i++)
  x = x + A[i];

for (i=0; i<N; i++)
  if (B[i]>0) fun (B[i], x);
```

Esa operación se ejecuta de forma sencilla en paraleo; por ejemplo, entre **cuatro** procesos, cada uno procesa un cuarto del vector.

Pero, ¿de esta forma?

P0

```
for (i=0; i<N/4; i++)
  x = x + A[i];

for (i=0; i<N/4; i++)
  if (B[i]>0) fun (B[i], x);
```

...

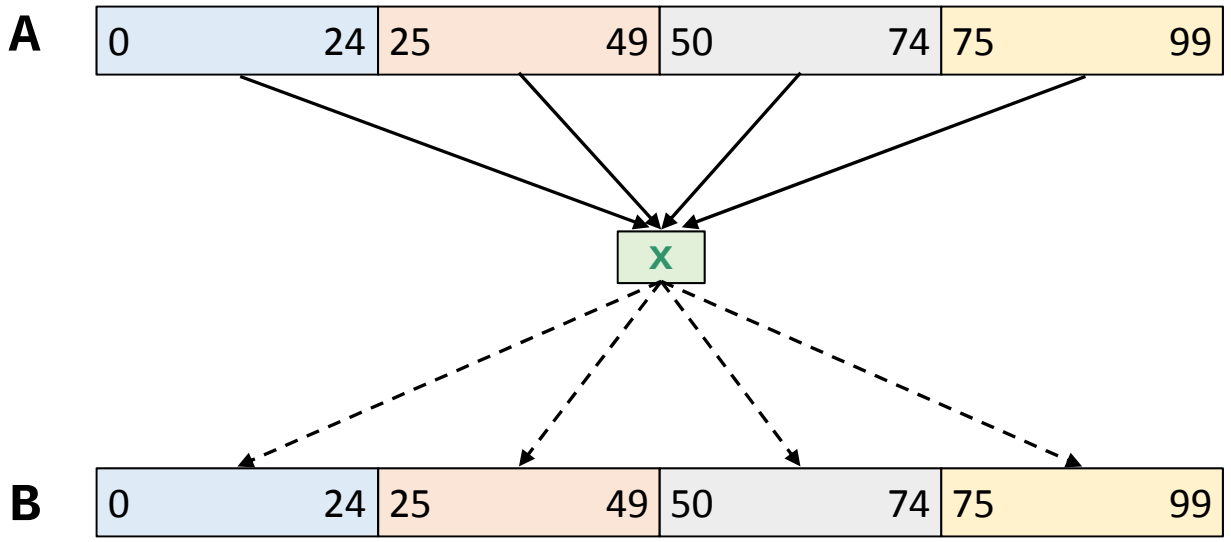
P3

```
for (i=3*N/4; i<N; i++)
  x = x + A[i];

for (i=3*N/4; i<N; i++)
  if (B[i]>0) fun (B[i], x);
```

Problemas

Los cuatro procesos utilizan la variable **X**: en el primer bucle, para calcular la suma de los elementos del vector **A** y, a continuación, para actualizar mediante la función los valores del vector **B**



¿Cómo se pueden realizar esas operaciones?

Problemas: primer bucle

En cada iteración, los procesos **leen** la variable **x**, hacen la suma y **escriben** en la variable **x**:

operación *read-modify-write* (RMW)

```
LD   R1,A(i)
LD   R2,x
ADD  R3,R2,R1
ST   x,R3
```

La ejecución paralela entre distintos procesos no está ordenada: las lecturas y escriturs pueden realizarse en cualquier orden… por lo que **el resultado puede no ser correcto.**

```
P0: LD..ADD..ST
P1:             LD....ADD..ST
P2:       LD....ADD....ST
P3:     LD...ADD....ST
```

P0, P2 y P3 han leído el mismo valor

Problemas: primer bucle

Para solucionar el problema hay que utilizar una **sección crítica**, para realizar la suma de forma ordenada, una a una.

```
LD   R1,A(i)
LD   R2,x
ADD  R3,R2,R1
ST   x,R3
```

Las secciones críticas se protegen mediante **variables cerrojo**.

Cerrojo: **1** - cerrado: esperar, no se puede entrar a la SC.

0 - abierto: se puede entrar a la SC.

Para gestionar el cerrojo se necesitan **instrucciones máquina específicas**, que garantizan el acceso **atómico** a memoria.

Acceso atómico: se lee y se escribe una posición de memoria sin interferencia de otros procesos (se verá en la asignatura PAR).

- Para proteger una sección crítica son necesarias dos funciones atómicas con la variable cerrojo: **`lock`** y **`unlock`**.

```
LOCK (cer)
{
   analizar el cerrojo:
     0 (abierto):  cerrar el cerrojo (1) y entrar en la SC;
     1 (cerrado): esperar hasta que el cerrojo esté abierto (0);
}

UNLOCK (cer)
{
   abrir cerrojo (0);
}
```

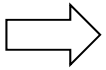
```
LOCK (cer);
x = x + A[i];
UNLOCK (cer);
```

ATENCIÓN

Una sección crítica se ejecuta en serie (proceso a proceso). Por tanto, debe ser un trozo de código pequeño, para no perder rendimiento.

```
for (i=0; i<N/4; i++)
  x = x + A[i];
...
```

??



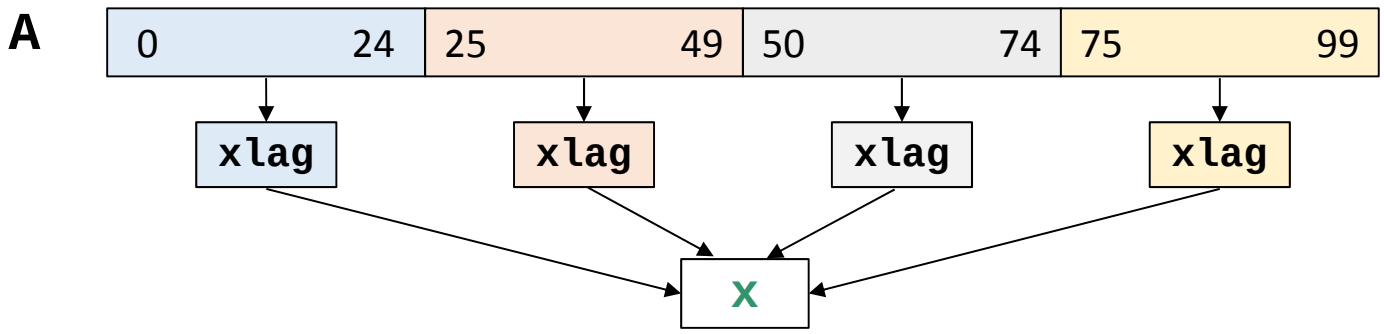
```
for (i=0; i<N/4; i++)
{
  LOCK (cer);
  x = x + A[i];
  UNLOCK (cer);
}
```

El código es correcto… pero ¿es eficiente? **No se ejecuta nada en paralelo**. Todo se ejecuta dentro de la sección crítica, en **serie**.

- Existe una solución más eficiente para este caso.

  Dado que se trata de una operación conmutativa, podemos definir una **variable privada** para cada proceso, **`xlag`**, y calcular las sumas parciales.

  Al final, hay que sumar en la variable **x** todas las sumas parciales; eso sí, dentro de una **sección crítica**.



```
for (i=0; i<N/4; i++)
  x = x + A[i];
...
```

!!

```
for (i=0; i<N/4; i++)
{
  xlag = xlag + A[i];
}
LOCK (cer);
 x = x + xlag;
UNLOCK (cer);
```

P0 <---------- cálculo ----------> [SC]

P1 <-------- cálculo ----------> [SC]

P2 <---------- cálculo --------> [SC]

P3 <---------- cálculo ----------> [SC]

La paralelización es correcta y eficiente. Sólo se ha ejecutado una sección crítica por cada proceso, no una por cada iteración.

Barreras de sincronización

Todos los procesos se quedan esperando, hasta que todos lleguen a la barrera.

Por ejemplo, alguna variable calculada entre todos los procesos, que debe volver a utilizarse en todos ellos. Aunque un proceso haya finalizado la primera fase, debe esperar a que todos hayan finalizado.

El primer proceso cierra la barrera y se queda esperando; los siguientes incrementan el contador de la barrera y se quedan esperando; **el último abre la barrera** y libera a todos los procesos.

Para crear una barrera, son necesarios:

- **un flag**: barrera cerrada (0) o barrera abierta (1).
- **un contador**, para contar cuántos procesos han llegado a la barrera; controlado dentro de una sección crítica.

Una barrera simple

B, un `struct` con tres variables: cerrojo, contador y estado

P: número de procesos

El primero cierra la barrera; el último la abre.

```
BARRERA(B,P)
{
   LOCK (B.cer);
     if (B.cont == 0) B.estado = 0;
     B.cont++;
     mi_cont = B.cont;
   UNLOCK (B.cer);

   if (mi_cont == P) {
     B.cont = 0;
     B.estado = 1;
   }
   else while (B.estado == 0) { };
}
```

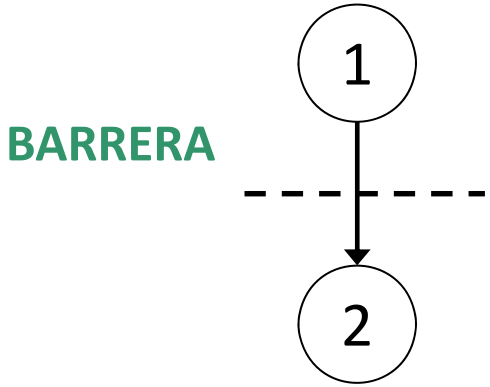
- Barreras de sincronización

```
for (i=0; i<N; i++)
  x = x + A[i];

for (i=0; i<N; i++)
  if (B[i]>0) fun(B[i], x);
```

En el ejemplo hay dos bucles que no son independientes.

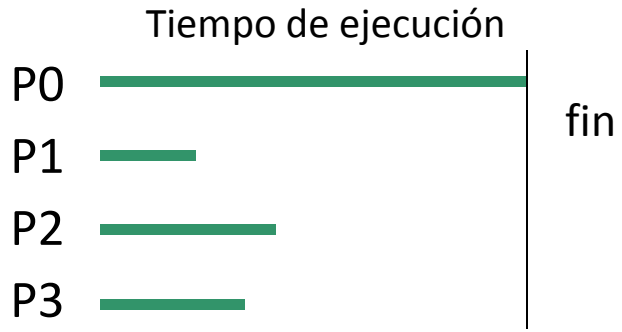
Hay una dependencia del primer bucle hacia el segundo: no se puede ejecutar el segundo sin haber acabado el primero.



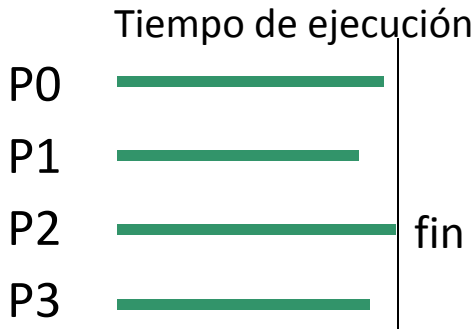
Hay que **sincronizar** la ejecución del segundo bucle, utilizando una barrera entre ambos: hay que esperar a que todos los procesos acaben el primer bucle.

```
for ...
BARRERA(B,P);
for ...
```

- ¿Cómo se reparten las iteraciones de un bucle (en general, las tareas) entre los procesos? El objetivo debe ser conseguir tiempos de ejecución similares entre todos los procesos (***load balancing***).



Reparto **no equilibrado**



Reparto **equilibrado**

- Dos estrategias:

  reparto estático: en tiempo de compilación.

  reparto dinámico: en tiempo de ejecución.

- En el ejemplo anterior, hemos realizado una planificación estática: cada procesador trabaja con un cuarto del bucle.

P0

```
for (i=0; i<N/4; i++)
  x_l = x_l + A[i];
...
for (i=0; i<N/4; i++)
  if (B[i]>0) fun (B[i], x);
```

...

P3

```
for (i=3*N/4; i<N; i++)
  x_l = x_l + A[i];
...
for (i=3*N/4; i<N; i++)
  if (B[i]>0) fun (B[i], x);
```

¿Se ha repartido la misma carga a cada procesador?

- **Primer bucle**: seguramente **SÍ**, dado que todas las iteraciones tienen el mismo coste.
- **Segundo bucle**: quizá **NO**; no sabemos cúal va a ser el cose de cada iteración, dado que se conocerá en tiempo de ejecución, en función del valor de cada elemento del vector **B**.

Planificación estática

Al escribir el programa paralelo se decide qué parte ejecutará cada proceso (**`pid`** = identificador del proceso). Dos opciones básicas:

- **Consecutiva**

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

```
inicio = pid * N/P;
fin = (pid+1) * N/P;
for (i=inicio; i<fin; i++)
{
   ...
}
```

- **Entrelazada**

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

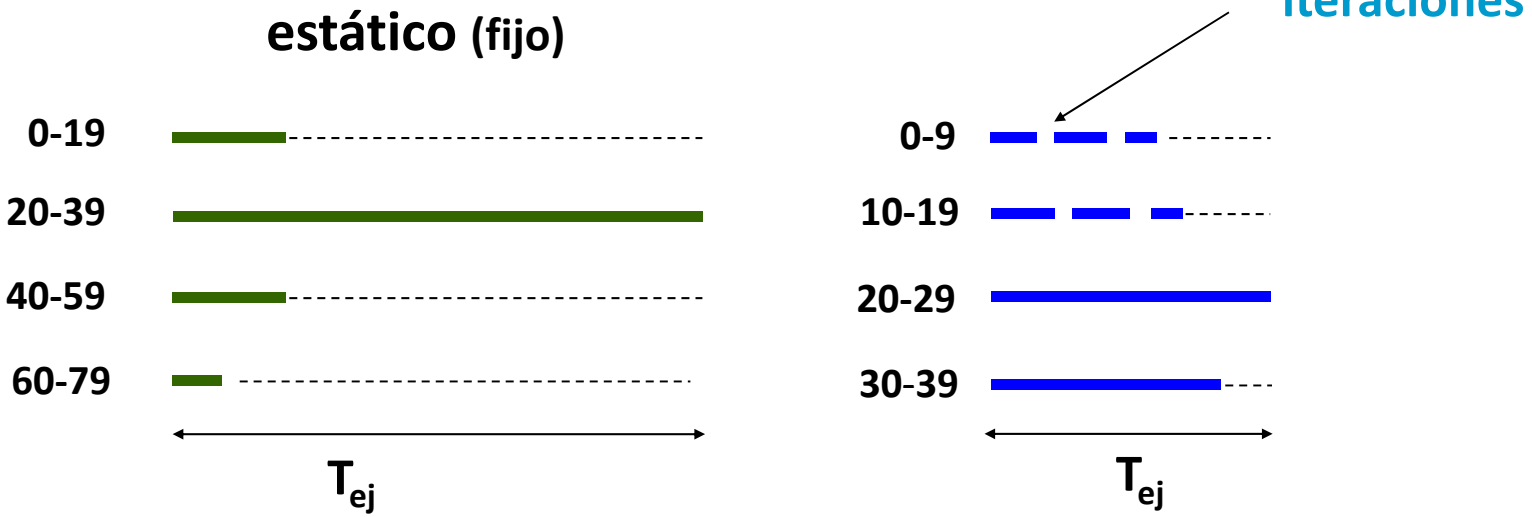
```
for (i=pid; i<N; i=i+P)
{
   ...
}
```

- **No se añade sobrecarga a la ejecución de los procesos**.
- Pero **no se asegura un reparto equilibrado** del trabajo entre todos los procesos.

Reparto equilibrado

```
for (i=0; i<80; i++)
  if (A[i] > 0) calcular();
```

Asignación dinámica de las iteraciones



Reparto dinámico

Para mantener equilibrado el reparto de trabajo, se organiza una "cola de tareas". Cuando un proceso finaliza una tarea (parte de un bucle), coge una nueva tarea de la "cola".

Dos opciones básicas:

a. los **trozos** del bucle son **del mismo tamaño**.

b. según se va ejecutando el programa, **los trozos son cada vez más pequeños**.

Self /Chunk scheduling

Las iteraciones se reparten de una en una o en bloques (*chunk*).

ATENCIÓN: supone una **sobrecarga** a la ejecución del bucle.

Hay que valorar el tiempo de ejecución y el tiempo de reparto.

```
LOCK (cer);
  mia = i;
  i = i + Z;    // Z = 1 → self
UNLOCK (cer);

while (mia < N) {
  for (j=mia; j<mia+Z; j++)
  {
    ...
  }
  LOCK (cer);
   mia = i;
   i = i + Z;
  UNLOCK (cer);
}
```

***Guided** scheduling*

Los **trozos** de bucle que se reparten son son siempre del mismo tamaño, sino que son **cada vez más pequeños**.

Por ejemplo, el trozo que se reparte tiene es una parte proporcional al número de iteraciones que faltan por procesar:

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

N: número total de iteraciones; i: iteración en la que se encuentra el bucle.

o repartos similares,

$$Z_s = Z_{s-1} (1 - 1/P)$$

- Ejemplo (1.000 iteraciones independientes, 4 procesadores):
  - ***chunk scheduling*** (Z = 100)

    100 100 100 100 100 100 100 100 100 100

    10 operaciones de reparto

  - ***guided scheduling***

    por repartir: 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 operaciones de reparto

Resumen

- En los computadores paralelos SMP, los núcleos comparten la memoria principal; la comunicación entre procesos se realiza mediante variables compartidas.
- La parte que hay que ejecutar en serie en un programa paralelo (o en un número reducido de procesadores), condiciona el **rendimiento** que se puede obtener (leyes de Amdhal/Gustafson). Parámetros de rendimiento: factor de aceleración y eficiencia.
- Hay que sincronizar el uso de variables compartidas. La **sincronización** se consigue mediante instrucciones máquina específicas, atómicas. Dos mecanismos: **secciones críticas** (trozos de código que los procesos ejecutan de forma secuencial), y **barreras** (para sincronizar todos los procesos).
- **El reparto de tareas** tiene que ser equilibrado, para no perder tiempo. El reparto puede decidirse en tiempo de compilación (conocemos el coste de las tareas), o en tiempo de ejecución (el coste de las tareas es desconocido).

3. Sistemas con más de un procesador: arquitecturas de memoria compartida

Apartado B: OpenMP (introducción)

- Introducción.
- Regiones paralelas. Tipos de variables.
- Paralelización de bucles. Reparto de iteraciones.
- Sincronización entre procesos.

OpenMP - Introducción

- Para programar aplicaciones paralelas en máquinas de memoria compartida (**SMP**).

- Qué ofrece
  - identificación de los procesos paralelos.
  - declaración de variables: compartidas, privadas…
  - definición de regiones paralelas (bucles, funciones…).
  - mecanismos de sincronización de procesos.

- Qué no ofrece
  - análisis de dependencias entre tareas (responsabilidad del programador).

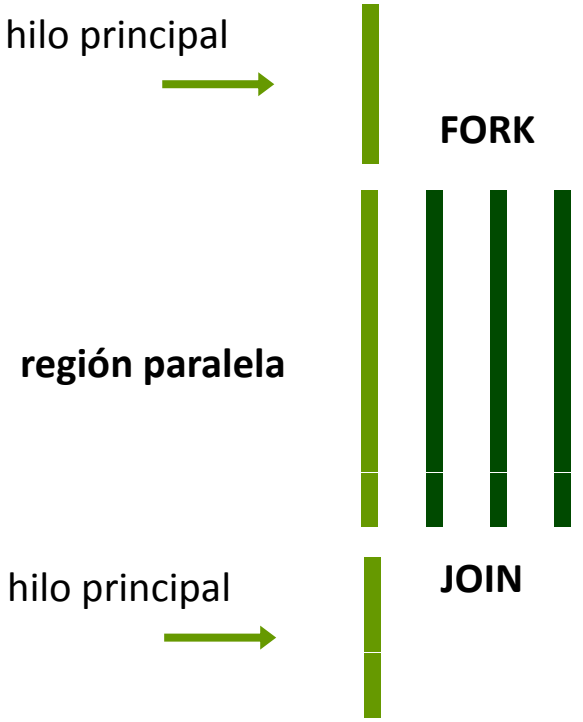
- No es un lenguaje de programación, sino una **API** (*application programming interface*) con los siguientes componentes (`include <omp.h>`):
  - **Directivas** para el compilador
  - Funciones de librería
  - Variables de entorno
- Las directivas facilitan la portabilidad y el paralelismo "incremental" (paralelizar una parte, a continuación otra...). En los entornos en los que no se utiliza OpenMP, no se tienen en cuenta las directivas.

```
C:        #pragma omp <directiva>
Fortran:  !&omp <directiva>
```

- El modelo de programación paralela que se utiliza es **Fork – Join**.

  En un determinado momento, el hilo principal genera P hilos que se ejecutan en paralelo.

  Al finalizar la región paralela, desaparecen todos los hilos, salvo el principal: la ejecución continúa en serie.



- Todos los hilos ejecutan **el mismo código** (una copia). Se asigna un identificador a cada hilo (**`tid`**).

  Para diferenciar las tareas ejecutadas por cada hilo:

  - **`if (tid == 0) then ... else ...`**
  - directivas específicas para el reparto de tareas.

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

- A tener en cuenta a la hora de paralelizar el código

1. Partiendo del código en serie, hay que determinar qué partes se pueden ejecutar en paralelo (tras analizar las dependencias).
   - estructuras de control paralelo
   - reparto de trabajo

2. Determinar el uso de cada variable: privadas, compartidas…

3. En caso de necesidad, hay que sincronizar los hilos: exclusión mutua (secciones críticas) o sincronización por eventos (por ejemplo, sincronización global por medio de barreras).

- **Región paralela**: parte del código que ejecutan todos los procesos en paralelo.

  Se indica mediante la siguiente directiva:

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

  La parte de código que define la región paralela debe ser un bloque básico del programa.

- Para indicar el **número de hilos** que se quieren generar en la región paralela, podemos utilizar estas tres opciones:

  a. De forma estática, antes de ejecutar el programa, utilizando una variable de entorno (línea de comandos):

```
export OMP_NUM_THREADS=8
```

  b1. En tiempo de ejecución, utilizando esta función de librería antes de la región paralela:

```
omp_set_num_threads (8);
```

  b2. En tiempo de ejecución, utilizando esta claúsula de la directiva `pragma parallel`:

```
#pragma omp parallel num_threads (8)
```

¿Qué hilo soy? ¿Cuántos hilos somos?

Cada hilo paralelo tiene su identificador.

El identificador 0 se corresponde con el hilo principal (*master*) .

Existen estas dos funciones de librería para identificar a los hilos:

```
tid = omp_get_thread_num ();
```

devuelve la identificación del hilo

```
nth = omp_get_max_threads ();
```

devuelve el número de hilos

Un ejemplo simple

```
...
#define N  12
int     i, tid, nth, A[N];
void 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 (“hilo %d(%d) en ejecución", tid, nth);
  A[tid] = 10 + tid;
  printf (“ha acabado el hilo %d\n", tid);
 }
 for (i=0; i<N; i++) printf (“A(%d) = %d\n”, i, A[i]);
}
```

barrera

- El hilo principal (master) tiene en cuenta todas las variables del programa y está activado durante toda la ejecución.

- Entre los hilos las variables pueden ser **privadas** o **compartidas** (*shared*). El comportamiento de las variables de indica mediante las correspondiente cláusulas.

- Éstas son las principales cláusulas para definir el comportamiento de las variables:

- **`shared (X)`** (lista de variables: a, b, c…)

  La variable **X** es compartida. Todos los hilos utilizan la misma variable (no hay copias).

  En caso de indicar lo contrario, las variables son compartidas.

- **`private (Y) / firstprivate (Y)`**

  La variable **X** es privada. Cada hilo crea su propia variable, sin inicializar. Al finalizar la ejecución de un hilo, se borra la copia de la variable en ese hilo.

  En el caso `firstprivate`, se mantiene el valor inicial a la variable.

El comportamiento se define para toda la variable; por ejemplo, no se pueden definir algunos elementos de un vector privados y otros compartidos.
Cada hilo utiliza una pila privada, por lo que todas las variables creadas dentro de una región paralela son privadas.

Ejemplo

```
X = 2; Y = 1; Z = 3;
#pragma omp parallel
  private (X) shared (Y) firstprivate (Z)
{
  Y = X*X + 3;
  X = Y*3 + Z;
}
printf ("X = %d \n", X);
```

X no tiene valor!

Z = 3

X no tiene un nuevo valor

- **`reduction (op:Z)`**

  La variable **Z** es compartida, y sobre ella se realiza una operación de reducción (op) de tipo RMW, de forma atómica. Por ejemplo, suma de los elementos de un vector, cálculo del máximo…

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

La propia cláusula indica la operación a realizar.

ATENCIÓN: no se puede saber en qué orden se realizarán las operaciones

--> la operación debe ser conmutativa (cuidado con los redondeos).

- Otro tipo de variables

  - **`#pragma omp threadprivate (X)`**

    La variable es privada, pero no desaparece al finalizar la región paralela (si no cambia el número de hilos). Mantiene el valor para las siguientes regiones paralelas. La variable debe ser global (antes del `main`).

    (También se puede utilizar al llamar a una función, para mantener la variable privada, sin necesidad de pasarla como parámetro)

  - **`copyin (X)`** cláusula

    Un hilo no puede acceder a las variables privadas de otro hilo. Mediante la claúsula **`copyin`** se puede copiar, en el resto de hilos, el valor que el master tiene de la variable al comienzo de la región paralela.

- Otras cláusulas

  - `default (none / shared)`

    none: hay que definir el carácter de las variables (útil para no olvidarse ninguna variable).

  - `if (valor)`

    Se creará la región paralela en función del valor (1/TRUE); en otro caso, se ejecutará en serie.
    Útil para no generar regiones paralelas “muy pequeñas”, teniendo en cuenta el coste de generar los hilos, sincronizarlos, etc.

  - `num_threads (valor)`

    Número de hilos a utilizar en la región paralela.
    Prioridad: variable de entorno >> función >> cláusula

- Resumen: cláusulas de las regiones paralelas
  - `shared, private, firstprivate (var)`
    `reduction (op:var)`
    `default (shared/none)`
    `copyin (var)`
  - `if (valor)`
  - `num_threads (valor)`

- Nota

  Salvo indicar lo contrario, **dentro de una región paralela no se puede crear otra región paralela** (*nested parallelism*).

  Para esa opción (¡después de analizar muy bien qué se quiere hacer!), hay que indicarlo explícitamente a través de una variable de entorno o función (se verá en la asignatura PAR).

OpenMP – Reparto de trabajo

- En general, es sencillo conseguir paralelismo en los bucles. **No hay que replicar el bucle**, sino **repartirlo**.

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

?

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

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

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

!

- El reparto de trabajo puede hacerse a “mano” o de forma “automática” utilizando algunas directivas.

  Éstas son las principales opciones:

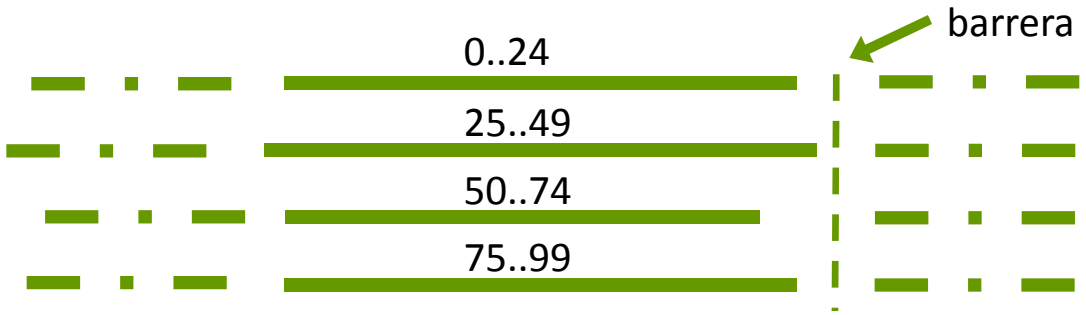
1. `for` — reparto de las iteraciones de un bucle.
2. `sections` — reparto de las funciones de una región paralela.
3. `single` — indica un trozo de código que será ejecutado por un único hilo.
4. `master` — indica un trozo de código que será ejecutado sólo por el hilo principal (*master*).

1. `pragma omp for [cláusulas]`

Se reparten las iteraciones de un bucle entre los hilos.
Todos los hilos se sincronizan al finalizar el bucle.

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

tipo de variables
reparto de iteraciones
sincronización



- Las directivas **`parallel`** y **`for`** pueden agruparse en una única en el caso de una región paralela con un único bucle

```
#pragma omp parallel for
```

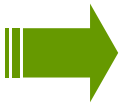
```
for (i=0; i<N; i++)
  Z[i] = a * X[i] + b;
```

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

El bucle puede paralelizarse sin problemas, dado que las iteraciones son independientes. La directiva **`parallel for`** creará varios hilos y repartirá la ejecución del bucle entre ellos.

El índice del bucle, **`i`**, es una variable privada (no es necesario indicarlo, pero merece la pena indicar el carácter de todas las variables).

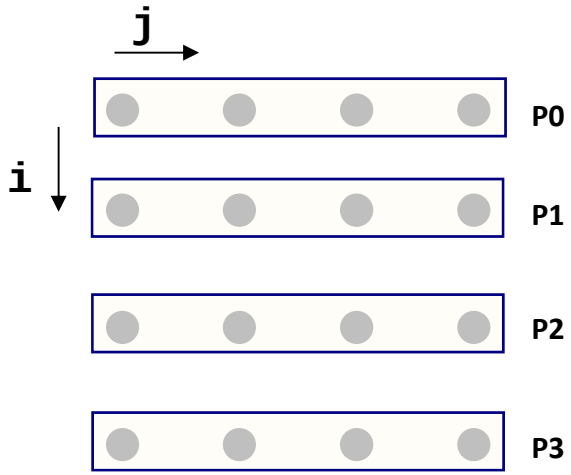
```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
 {
   x = B[i][j] * B[i][j];
   A[i][j] = A[i][j] + x;
   C[i][j] = x * 2 + 1;
 }
```



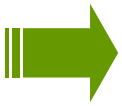
```
#pragma omp parallel for
                private(i,j,x)
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
 {
   x = B[i][j] * B[i][j];
   A[i][j] = A[i][j] + x;
   C[i][j] = x * 2 + 1;
 }
```

El bucle externo, `i`, se ejecutará en paralelo; los hilos ejecutarán el bucle `j`, para distintos `i`. Paralelismo de grano medio.

Las variables **`i`**, **`j`** y **`x`** tiene que ser **privadas**.



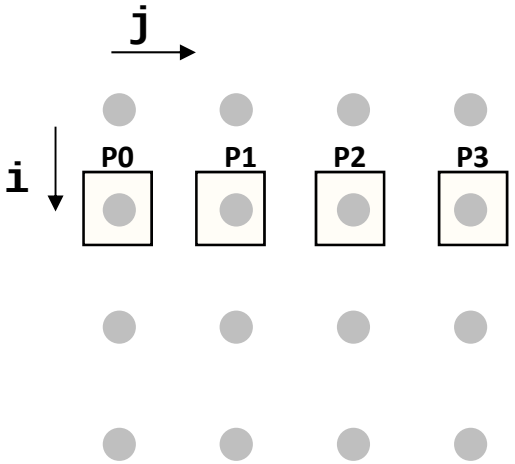
```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
 {
   x = B[i][j] * B[i][j];
   A[i][j] = A[i][j] + x;
   C[i][j] = x * 2 + 1;
 }
```



```
for (i=0; i<N; i++)
  #pragma omp parallel for
                private(j,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;
  }
```

El bucle externo, `i`, se ejecutará en serie. **En cada iteración, los hilos ejecutarán el bucle interno de forma paralela**.

Las variables `j` y `x` tienen que ser **privadas**.



Cláusulas de la directiva `for`

✓ **Carácter** de las variables

Al igual que en la región paralela (`shared`, `private`...), además de **`lastprivate`**: la variable seguirá en el hilo principal con el valor de la última iteración.

✓ **Planificación**

Para indicar cómo hacer el reparto de las iteraciones. Muy importante para conseguir un reparto de **trabajo equilibrado** (**rendimiento**).

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

Recuerda

- estático (compilación): sin coste, no garantiza equilibrio
- dinámico (ejecución): sobrecarga; reparto equilibrado

- Cláusulas de la directiva **`for`**

  - ✓ **`schedule (tipo[, tamaño_trozo])`**

`static,k`	**estático**, k iteraciones consecutivas (*round robin*; k=0 > N/P)
`dynamic,k`	**dinámico**, k iteraciones en cada asignación
`guided,k`	**dinámico**, número de iteraciones cada vez más pequeño (al final, k) $k_{i+1} = k_i (1 - 1/P)$ ($k_1 = N/P$ en general)
`runtime`	indicado en la variable de entorno **`OMP_SCHEDULE`** Ejemplo: **`$ export OMP_SCHEDULE="dynamic,1"`** Adecuado para hacer pruebas, evitando compilar el código cada vez que se quiere hacer una prueba. Una vez decidida la planificación final, es mejor evitar esta opción.

- Cláusulas de la directiva **`for`**
  - ✓ **`nowait`**

    Elimina la barrera existente al final del bucle.
    Adecuado para eliminar esa sincronización cuando no es necesaria.
    ¡**Cuidado**! Si se elimina la barrera en caso de ser necesaria, el resultado no será correcto.

  - ✓ `ordered`

    Ejecutar las iteraciones en orden.

  - ✓ `collapse (n)`

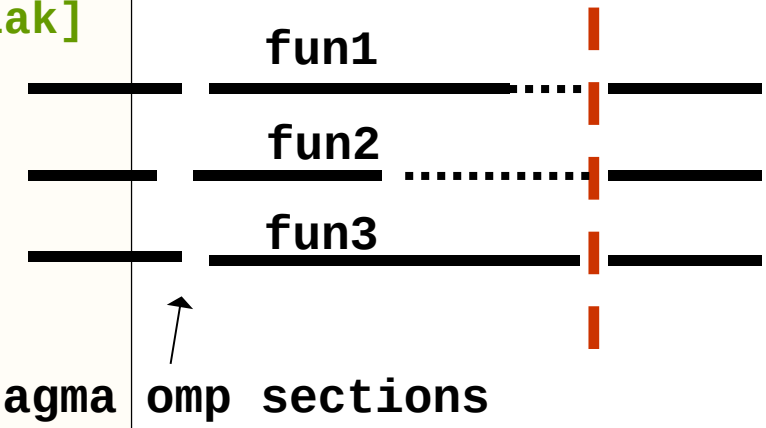
    Para agrupar los bucles pequeños (n) en un bucle mayor.

2. pragma omp **sections [cláusulas]**

Se reparten partes de código independientes (funciones) entre los hilos. Cada sección será ejecutada por un único hilo. Los hilos se sincronizan al final mediante una barrera.

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

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



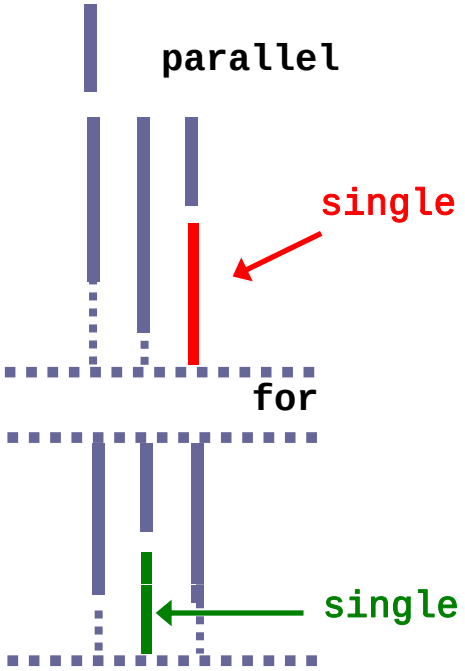
3. pragma omp **single [cláusulas]**

Indica una **parte del código que será ejecutada por un único hilo** (por ejemplo, una operación de E/S). No se sabe qué hilo ejecutará ese código: el primero en llegar. Al finalizar se sincronizan todos los hilos mediante una barrera implícita.

```
#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);
}
```



4. `pragma omp master`

Define una parte de código que sólo ejecutará el hilo principal, master (`pid` = 0).

Es similar a la directiva `single`, pero no tiene barrera al final, y el hilo que ejecutará esa parte de código es conocido: el hilo principal.

- El reparto (`for`, `sections`…) se puede indicar en la región paralela o en las funciones utilizadas.

  Por ejemplo:

```
#pragma omp parallel private (x)
{
  x = ...
  init (x);
  ...
}
```

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

Resumen

Hay que **repartir** las **tareas** entre los hilos de forma que el reparto sea lo más equilibrado posible.

El reparto puede ser "manual", en función del identificador del hilo, o bien a través de las opciones que ofrece OpenMP.

- Iteraciones del bucle (paralelismo de datos) mediante la directiva `pragma for` y la cláusula `schedule`. El reparto puede ser **estático** o **dinámico** (*self-scheduling* o de tipo *guided*).
- Secciones (paralelismo de función), mediante la directiva `pragma sections`.
- Dos opciones si queremos que un trozo de código sea ejecutado por un único proceso: `pragma single` y `pragma master`.

- Cuando las tareas no son independientes y hay que utilizar variables compartidas, es necesario sincronizar la ejecución de los hilos.

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

1. Secciones críticas

Se trata de un trozo de código que no puede ser ejecutado por más de un hilo al mismo tiempo. Todos los hilos ejecutarán ese trozo de código, pero de uno en uno.

Dos opciones principales: directivas `critical` y `atomic`.

`pragma omp critical`

Define una sección crítica. Existen dos opciones:

a. sin cerrojo explícito.

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

ATENCIÓN: una sección crítica tiene que ser lo más breve posible

`pragma omp critical`

b. definir distintas secciones críticas con variables cerrojo ("con nombre").

```
#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]; }
}
```

- Para gestionar las secciones críticas se utilizan cerrojos. OpenMP tiene varias funciones para trabajar con cerrojos: **`lock, unlock`**…

  Mediante dichas funciones se pueden gestionar todas las secciones críticas que se deseen (lo veremos en la asignatura PAR).

- **`pragma omp atomic`**

  Caso particular de una sección crítica para realizar una operación atómica RMW simple (sólo para unas pocas operaciones).

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

Para esas operaciones es más eficiente que una sección crítica normal.

OpenMP – Barreras

- Otro mecanismo de sincronización habitual es la sincronización por eventos: esperar hasta que algo ocurra.

  La sincronización puede ser punto a punto (de un hilo a otro) mediante una variable compartida o **global, mediante una barrera**.

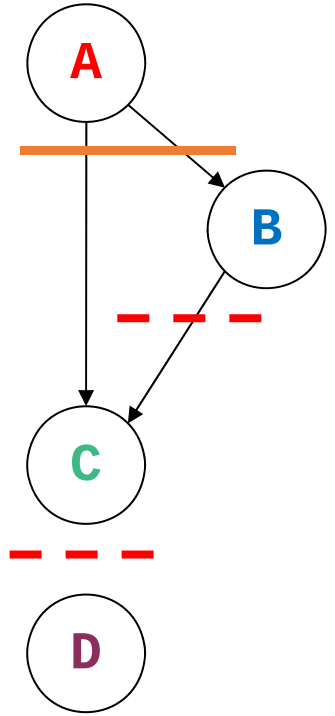
```
#pragma omp barrier
```

Barrera de sincronización entre todos los hilos: todos los procesos se quedan esperando hasta que todos lleguen a ese punto.

Recuerda: las directivas `parallel`, `for`... tienen una barrera implícita al final (en algunos casos puede eliminarse).

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);
}
```



OpenMP – Resumen

- **OpenMP:** API para paralelizar aplicaciones en los sistemas paralelos de memoria compartida (SMP).

  Directivas / funciones / variables de entorno

  - generar regiones paralelas.
  - reparto de tareas: iteraciones de bucles (for), secciones (sections).
  - sincronización de hilos (secciones críticas, barreras, cerrojos...).

- Dos funciones para medir el tiempo de ejecución:

  `omp_get_wtime ( );`

  `omp_get_wtick ( );`

```
...
t1 = omp_get_wtime ();
...
t2 = omp_get_wtime ();
Tej = t2 - t1;
...
```

Algunos errores habituales

- **definir de forma incorrecta las variables**: shared, private, reduction...
- ejecutar en paralelo tareas que no son independientes, sin la necesaria sincronización: **carreras** (*race*).
- sincronizar mal los hilos*:* **bloqueos** (*deadlock*).

Problemas con el rendimiento

- pérdida de la localidad al acceder a la cache, falsa compartición.
- reparto desequilibrado; pérdida de tiempo en las secciones críticas o barreras.
- Paralelismo de bajo nivel.
- apertura/cierre continuado de regiones paralelas sin ser necesario.

Libros

R. Chandra et al.
Parallel Programming in OpenMP. Morgan Kaufmann, 2001.

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

WEB

www.openmp.org (especificación, software…)