www.asesacademia.com

ASES

assessoria d'estudis superiors

PAR

DOSSIER INTENSIVO FINAL

ISMA Q.

Tema 1: Why parallel computing?

Ejecución serie vs paralela:

- **En ejecución serie:**

$$Tiempo\ ejecución\ =\ N\ *\ CPI\ *\ tciclo$$

Donde tciclo es el tiempo de un ciclo, CPI indica los ciclos por cada instrucción y N son el número de instrucciones en el programa

- **En ejecución paralela:**

$$Tiempo\ ejecución\ =\ \frac{N}{P}\ *\ CPI\ *\ tciclo$$

Aquí, P indica el número de procesadores sobre el que hemos paralelizado el programa.

OJO: Esa fórmula es muy idílica, ya que supone que hemos partido la carga de trabajo perfectamente entre los P procesadores y no hay dependencias en el programa ni ningún tipo de overhead. En la vida real siempre hay overheads y normalmente suelen haber dependencias. Ya hablaremos de ello más adelante!

¿Y qué es throughput computing o la concurrencia?
Es el hecho de ejecutar K aplicaciones distintas en P procesadores. El sistema operativo mete a cada aplicación (que no tienen nada que ver entre ellas) en un procesador distinto para que se puedan estar ejecutando todas. En caso de K > P, el sistema operativo multiplexa los tiempos de ejecución (sacando y volviendo a meter muy frecuentemente las aplicaciones en ejecución) para hacer parecer que se están ejecutando todas a la vez.

*Nota: En esta asignatura llamamos **procesador o CPU** a cada una de las unidades de cómputo que tiene un socket. Probablemente tendrás unas 8 en el ordenador de tu casa.*

Y ten en cuenta que si paralelizas un programa, su resultado debe ser siempre igual al de su ejecución serie. Ya veremos qué hacer para conseguirlo.

Procesos vs threads:

Un proceso es un programa, una aplicación. Por defecto, un proceso tiene un único thread (hilo) de ejecución. Pero si lo paralelizamos en un programa que ejecutarán P threads significa que cada uno de ellos se encargará de una porción del programa, y en el caso ideal podrán correr todos ellos en paralelo!

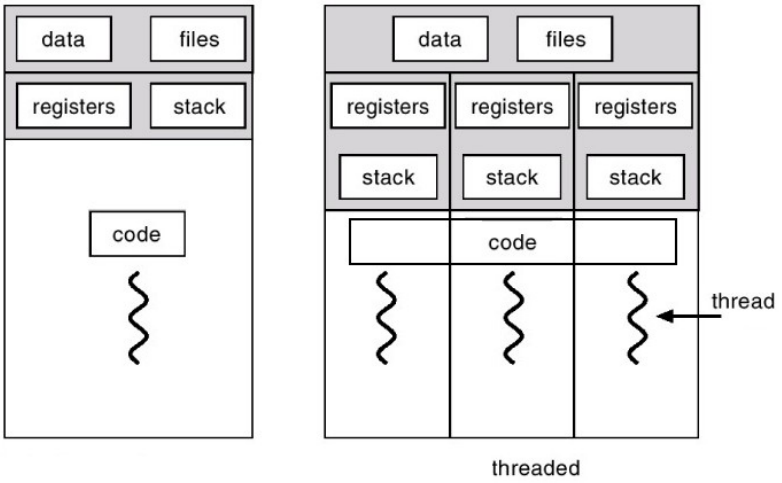


Figura 1. Procesos vs threads

Un proceso (programa) tendría la pinta que vemos a la izquierda de la figura 1. Tendría un único hilo de ejecución, que estaría ejecutando un cierto código, y ese proceso tendría todo lo que habéis visto en asignaturas anteriores: una pila, un código, usaría unos registros del procesador al ejecutarse…

En cambio, si partimos ese programa en P threads (donde todos esos threads siguen perteneciendo al mismo proceso), cada uno de ellos utilizará sus propios registros al ejecutarse en una CPU y utilizarán su propia pila.
Hay un único espacio de pila que puede usar ese proceso, pero cada uno de los threads utilizarán un espacio distinto de esa pila, de forma que cada uno puede guardar ahí sus propios datos!
El código, por otro lado, sí que será compartido por todos los threads del proceso. La gracia está en que probablemente querramos que cada uno de ellos esté ejecutando un trocito del código distinto.

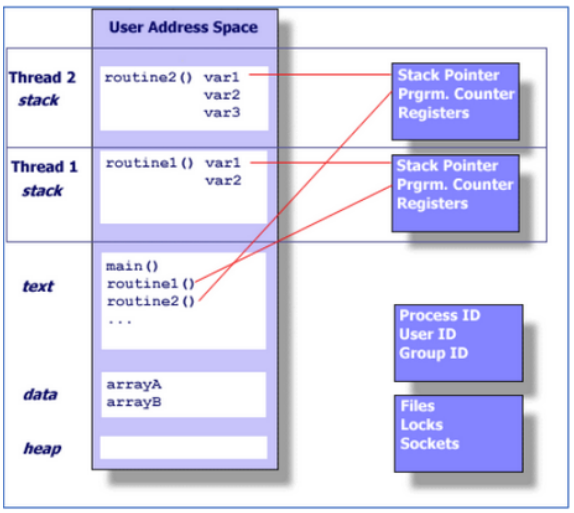


Figura 2. Un proceso con 2 threads

Problemas en la concurrencia / paralelización:

Hay 4 problemas principales que nos podemos encontrar al paralelizar un código o permitir concurrencia sobre una aplicación:

Data race:

Data race (en español, condición de carrera) ocurre cuando 2 threads están intentando hacer una modificación sobre una misma posición de memoria.

La forma de solucionarlo es incluir instrucciones / directivas de acceso exclusivo a variables. Veremos como hacerlo y cómo saber dónde colocarlas!
OJO: Es muy importante donde se colocan, porque colocarlas incorrectamente podría seguir haciendo que la ejecución fuese errónea o muy ineficiente.

Deadlock:

El deadlock (abrazo mortal) ocurre cuando colocamos las instrucciones de acceso exclusivo (locks) de forma que ambos threads quedan bloqueados.

Ejemplo: A quiere transferir dinero a B y B quiere transferir a A. Para que no hayan dataraces, ambos necesitan bloquear el acceso a su propia cuenta y a la ajena:

A	B
lock cuentaB	
	lock cuentaA
lock cuentaA	
	lock cuentaB

A puede coger el lock de cuenta B y B puede coger el lock de cuenta A. Pero luego en sus siguientes instrucciones de intentar coger un lock fallan porque lo tiene cogido otro thread o proceso. El resultado es que ambos se quedan bloqueados hasta esperar que el otro suelte el lock, pero nunca lo soltará.

Starvation:

Se produce cuando un thread no puede hacer ningún tipo de tarea útil por culpa de otro. Por ejemplo, cuando un thread tiene bloqueado mucho tiempo el acceso a un recurso y hay otro thread intentando acceder y no puede.

Livelock:

Ocurre cuando tenemos 2 threads haciendo trabajo completamente inútil. Los 2 threads están activos ocupando la CPU pero no avanzan.
Un ejemplo podría ser que, en el ejemplo del deadlock, se hace una solución muy mala en la que se dice “si tengo el lock A (o B) y detecto un bloqueo mútuo, suelto el lock A (o B) y al cabo de 10 segundos vuelvo a intentar cogerlo”: Aquí lo que ocurrirá es que, a los 10 segundos, ambos harán lo mismo y volverán a estar con el mismo problema.

Tema 2: Understanding Parallelism

Como ya vimos en el tema 1, es importante entender completamente el programa para saber si la computación puede ser dividida en varias tareas y si esas tareas se pueden ejecutar de forma paralela o hay dependencias entre ellas (una no puede empezar hasta que acabe la otra, porque necesita un resultado que la otra ha escrito).

Representaremos en un **TDG** las **tareas**, poniendo flechas indicando las **dependencias** de tipo Read after Write. Si entre 2 tareas no hay flechas, pueden ejecutarse paralelamente.

Cada tarea, que será un nodo del grafo, la representaremos con un número que indicará su **peso**, ya que no es igual una tarea que hace una suma que una tarea que hace la multiplicación de dos grandes matrices.

Camino crítico: Es el camino del grafo con más peso. Un camino está formado por los diferentes nodos que tienen dependencias entre ellos, y el camino más largo (considerando pesos de cada nodo) es el camino crítico.

Granularidad: cómo de grande es una tarea. Hacer pocas tareas muy pesadas tiene una mayor granularidad que hacer muchas tareas con poco contenido. Hay que buscar la granularidad perfecta!

Nomenclatura y definiciones:

T_1 : Suma de todos los tiempos que tardan todas las tareas. Es decir, es el tiempo que tardaría en ejecutarse en 1 procesador (donde se ejecutaría secuencialmente).

T_∞ : Suma de los tiempos de los nodos que forman el camino crítico. Es decir, es el tiempo que tardaría en ejecutarse en infinitos procesadores.

T_p : Tiempo que tardaría en ejecutarse el programa con P procesadores, teniendo en cuenta dependencias. $T_p >= T_\infty$

Pmin : Número mínimo de procesadores necesarios para que $T_p = T_\infty$

Parallelism : Parallelism = T_1 / T_∞. Indica el paralelismo ideal alcanzable con la aplicación. Cuanto mayor sea el número, más paralelizable será.

S_p : $S_p = T_1 / T_p$. SpeedUp conseguido con P procesadores.

E_p : $E_p = S_p / P$. Eficiencia del paralelismo con P procesadores.

Tseq : Tiempo de la parte no paralelizable.

Tpar : Tiempo (total) de la parte paralelizable.

Scalability: Cómo varía el speedUp en función del incremento de procesadores.

φ : φ = Tpar / T_1 . Fracción paralelizable del programa

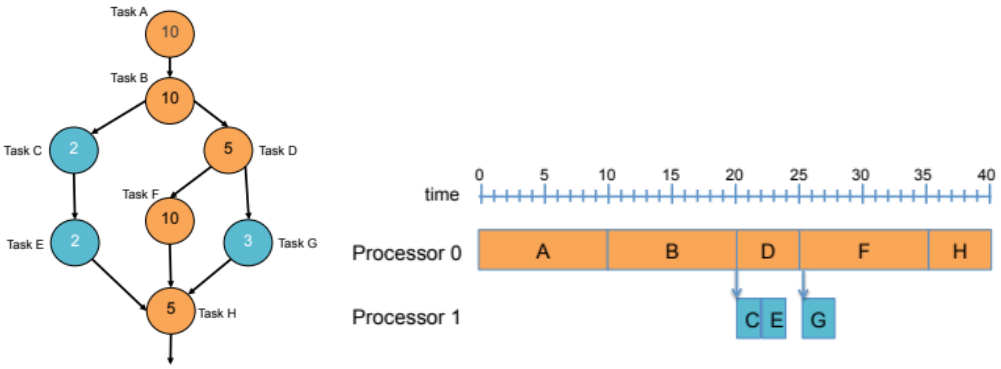


Figura 3. TDG de un programa. *Fuente: transparencias PAR*

Escalabilidad:

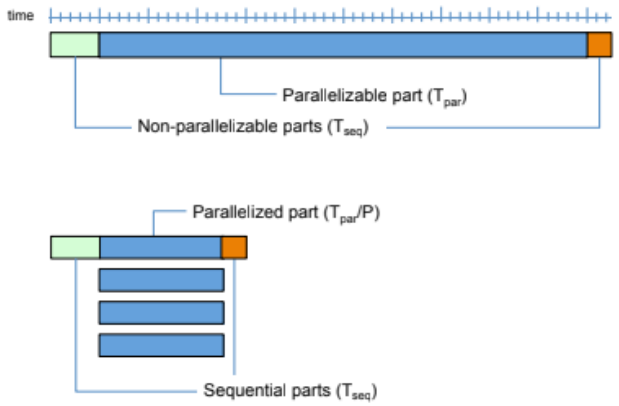
De una aplicación se puede medir su strong scalability y su weak scalability.

Si aumentamos P y el tiempo de ejecución se reduce linealmente (o cercano a linealmente) con el aumento de P, diremos que la aplicación tiene una gran **strong scalability**.

Si aumentamos el tamaño del problema P veces y nos ponemos a utilizar P procesadores y obtenemos un tiempo de ejecución igual o muy parecido al que teníamos inicialmente, diremos que la aplicación tiene una gran **weak scalability**.

No es excluyente tener strong y weak scalability, es decir, podría tener ambas, ninguna o solamente una de las dos.

Ley de Amdahl:



$$T_1 = T_{seq} + T_{par}$$
$$\varphi = T_{par}/T_1$$
$$T_1 = (1-\varphi) \times T_1 + \varphi \times T_1$$
$$T_P = T_{seq} + T_{par}/P$$
$$T_P = (1-\varphi) \times T_1 + (\varphi \times T_1/P)$$

Figura 4. Motivación Amdahl. *Fuente: transparencias PAR*

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1-\varphi) \times T_1 + (\varphi \times T_1/P)}$$

$$S_p = \frac{1}{((1-\varphi) + \varphi/P)}$$

Y esa es la fórmula de la ley de Amdahl aplicada al paralelismo.

En el caso de utilizar muchísimos procesadores, es decir P tiende a infinito, nos queda que:

$$S_p \to \frac{1}{(1-\varphi)}$$

Por tanto, tal como se puede apreciar, el SpeedUp que conseguiremos aumentando el número de procesadores viene completamente determinado por el peso que tenga sobre el programa la parte paralelizable.

Moraleja: No gastar esfuerzos en paralelizar partes que tienen un peso pequeño en el programa.

Pero que Amdhal no te deprima… Si tenemos un programa con poca parte paralela, y que por tanto tendrá una strong scalability muy mala (porque Sp crecerá poco en relación a P), podemos tener suerte de que podamos aumentar el tamaño del problema y así aumentemos justamente la parte paralelizable, de forma que estaríamos teniendo y aprovechando una buena weak scalability!

Overheads:

Paralelizar un programa no es ideal. Si un programa es perfectamente paralelizable y lo ejecutamos sobre 4 procesadores, no ocurrirá que $T_4 = T_1 / 4$, sino que tendremos $T_4 = T_1 / 4 + T(\text{overheads})$.

Los overheads más comunes son:

- **Overheads por creación de threads y tareas:** No se crean por arte de magia!
- **Overheads por acceso a recursos compartidos:** Uno tiene que esperar mientras el otro accede!
- **Overheads por sincronizaciones:** A veces tenemos que sincronizar a los threads, por ejemplo añadiendo barreras de las cuales no pueden pasar hasta que no hayan llegado todos allí, para asegurar el cumplimiento de dependencias y no acabar con un resultado erróneo.
- **Y más:**
    - Envíos de datos
    - "Idleness" (un thread no hace nada útil, debido a, por ejemplo, dependencias o desbalanceos de carga)
    - Replicación del mismo cómputo
    - Jerarquía de memoria

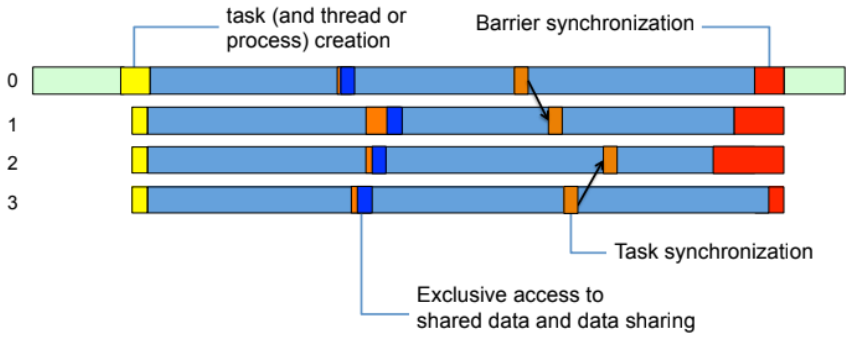


Figura 5. Ejecución de 4 threads mostrando algunos overheads. Fuente: transparencias PAR

Cuanto mayor es P, más grandes suelen ser los overheads.
Incluso hay un punto donde el rendimiento empieza a caer si se añaden nuevos procesadores. Fíjate:

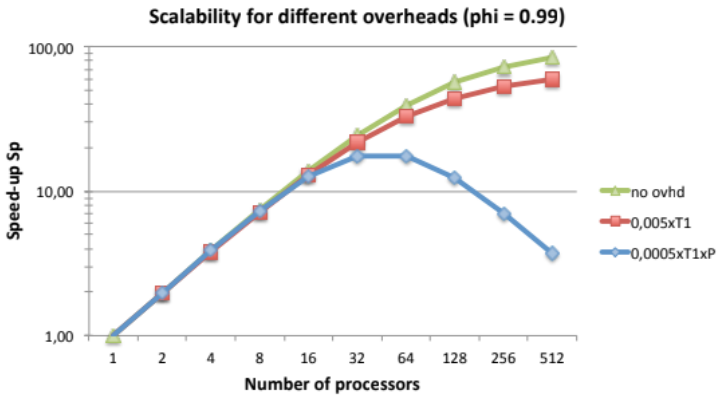


Figura 6. Overhead proporcional a P penaliza P altas. Fuente: transparencias PAR

Stencil usando Jacobi:

```
for (i = 1; i < n-1; i++) {
  for (j = 1; j < n-1; j++) {
    tmp = u[(i+1)][j] + u[(i-1)][j] + u[i][(j+1)] + u[i][(j-1)] - 4 * u[i][j];
    utmp[i][j] = tmp/4;
  }
}
```

Sin dependencias de read after write! Un chollo para paralelizarlo.

Lo primero que debes pensar es en la granularidad de las tareas que quieres tener:

Task is … (granularity)	Num. tasks	Task cost	T_1	T_∞	Parallelism
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	1
Each iteration of i loop	n	$n \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot t_{body}$	n
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{body}$	t_{body}	n^2
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot r \cdot t_{body}$	$n \div r$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$n^2 \cdot t_{body}$	$c \cdot t_{body}$	$n^2 \div c$
A block of r x c iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$n^2 \cdot t_{body}$	$r \cdot c \cdot t_{body}$	$n^2 \div (r \cdot c)$

Figura 7. Granularidad tareas al operar en una matriz. *Fuente: Transparencias PAR.*

Stencil usando Gauss-seidel:

```
for (i = 1; i < n-1; i++) {
  for (j = 1; j < n-1; j++) {
    tmp = u[(i+1)][j] + u[(i-1)][j] + u[i][(j+1)] + u[i][(j-1)] - 4 * u[i][j];
    u[i][j] = tmp/4;
  }
}
```

Aquí sí hay dependencias read after write.

Por tanto el grafo de dependencias sería este:

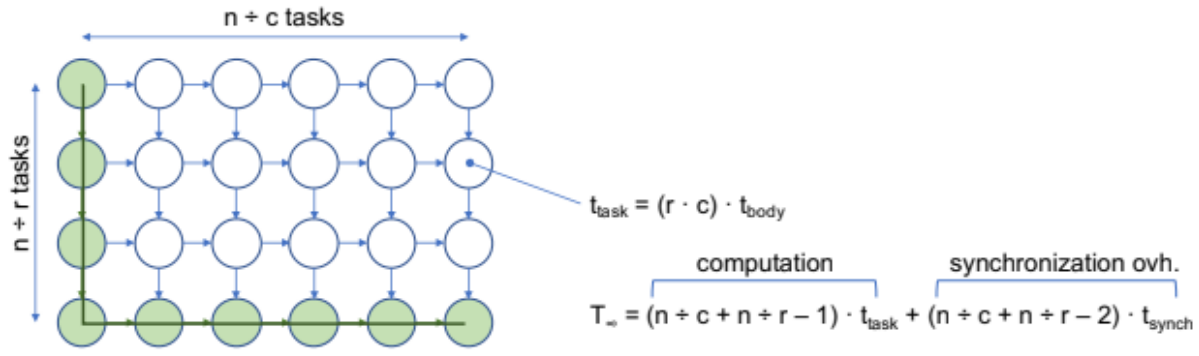


Figura 8. Dependencias Stencil usando Gauss-seidel. *Fuente: transparencias PAR.*

En la figura 8 se ve marcado en verde el camino crítico. Si te fijas en el código, una cierta posición de la matriz se debe calcular después de que estén correctamente calculadas las posiciones de arriba e izquierda, para respetar el resultado que habría en una ejecución secuencial donde se recorre la matriz por filas.

Cada arista del grafo es una sincronización, ya que hay que “avisar” cuándo está calculado un cierto nodo para poder ponerse a calcular el siguiente.

Aquí, la matriz de NxN ha sido partida en cuadraditos de r filas y c columnas, donde cada cuadradito es una task. El tiempo de ejecución en “infinitos” procesadores dependerá de su camino crítico, en el que hay (n/r) + (n/c) -1 tareas, y en el que hay (n/r -1) + (n/c -1) sincronizaciones aristas, es decir, sincronizaciones. Si cada sincronización tarda t_{synch} y cada tarea tarda t_{task}, su tiempo de ejecución es el que vemos en la parte derecha de la figura 8.

Memoria distribuida:

Threads de un proceso comparten memoria, pero 2 procesos distintos no, por lo que tendrán que transmitirse datos a través de un modelo de paso de mensajes.

Vamos a suponer que un proceso accede a algo que está en su memoria con tiempo 0, pero que si quiere algún dato que pertenece a la memoria de otro proceso (y que por tanto le va a tener que pedir y el otro proceso le va a tener que dar) va a tener que pagar el siguiente tiempo de acceso:

$$T_{access} = t_s + m \times t_w$$

Donde:
t_s : tiempo de preparar el acceso remoto
m : número de bytes a transferir
t_w : tiempo de un byte de esa transmisión

Suposiciones:

- En un momento dado, P_i solamente puede hacer 1 acceso remoto.
- En un momento dado, P_i solamente puede servir 1 acceso remoto a otro P_j
- En un momento dado, P_i solamente puede hacer 1 acceso remoto y servir 1 acceso remoto a P_k (se permite j = k)

Tema 3: Introduction to parallel architectures.

Coherencia de memoria (en arquitecturas **UMA**):

¿Por qué necesitamos un mecanismo de coherencia de memoria?

Suponed una caché **write back** (es decir, que cada vez que un core escribe en su caché, no se copia en memoria el dato escrito, hasta que la linea de caché sea reemplazada por otra), que es la más típica en los procesadores actuales. Aquí se ve de forma muy evidente que si otro core le pide ese mismo dato a memoria puede leer algo completamente obsoleto, ya que el dato que memoria le daría memoria no es válido.

Suponed una caché **write through** (es decir, que cada vez que un core escribe en su caché (salvo que estuviese en estado inválido) también se escribe en memoria). Aquí no se ve tan obvio por qué es incoherente, así que el ejemplo para mostrarlo es más complejo:
El core 0 (CPU0) lee el dato X (X=1 en este momento), y por tanto se almacena en su caché. El core 1 (CPU1) también lee X (X=1) y lo almacena en su caché. Entonces el core 0 escribe X=5 y esa información se escribe en su caché y también en memoria (por ser write through). Entonces el core 1 lee X, pero lee X=1 en lugar de X=5, ¿por qué? Porque ha sido un acierto de caché, y en la caché del core 1 la variable X aún valía 1 (solo valía 5 en la caché del core 0 y en memoria principal).

Write update snooping coherence:

Podemos conseguir coherencia a través de propagar todas las escrituras por un bus común que une todas las caché, de forma que cada caché pueda ver si ha habido alguna actualización de un dato que almacena. Así, cuando una caché escriba un dato, lo propagará por el bus, y el resto de controladores de las demás cachés (que estarán mirando constantemente el bus) verán si se ha escrito alguna dirección de memoria de las que están almacenando en su caché (y el valor del dato escrito). En caso de ser así, actualizarán su dato.

No es muy común encontrarse esto en los procesadores, porque introduce mucho tráfico en el bus (hay que propagar absolutamente todas las escrituras por el bus, indicando como mínimo la dirección de memoria afectada y el nuevo dato escrito, independientemente de si existe ese dato en los demás cores o no, porque no se sabe esa información).

Write invalidate snooping coherence:

El método más típico en los procesadores actuales es el de propagar ciertas escrituras por un bus común que une todas las caché, de forma que cada caché pueda ver si ha habido alguna actualización de un dato que almacena. Así, si un controlador de memoria ve que ha sido escrita una posición de memoria que estaba almacenando, convertirá esa línea de caché en inválida.

De esta forma solo hay que propagar las direcciones escritas (y no el dato). Además, no se propagarán todas las escrituras, ya que se almacenará la información de si esa línea de

memoria la tiene alguien más en caché (estado M indicará que somos los únicos con ella), por lo que solo se propagará la escritura si tenemos información de que alguien más está utilizando de forma válida ese dato.

La desventaja de este método es que provoca más misses en caché, dado que cada vez que se escribe en un lado se invalida en el resto de cachés que lo estaban usando.

El protocolo más sencillo de write invalidate es el **MSI**:

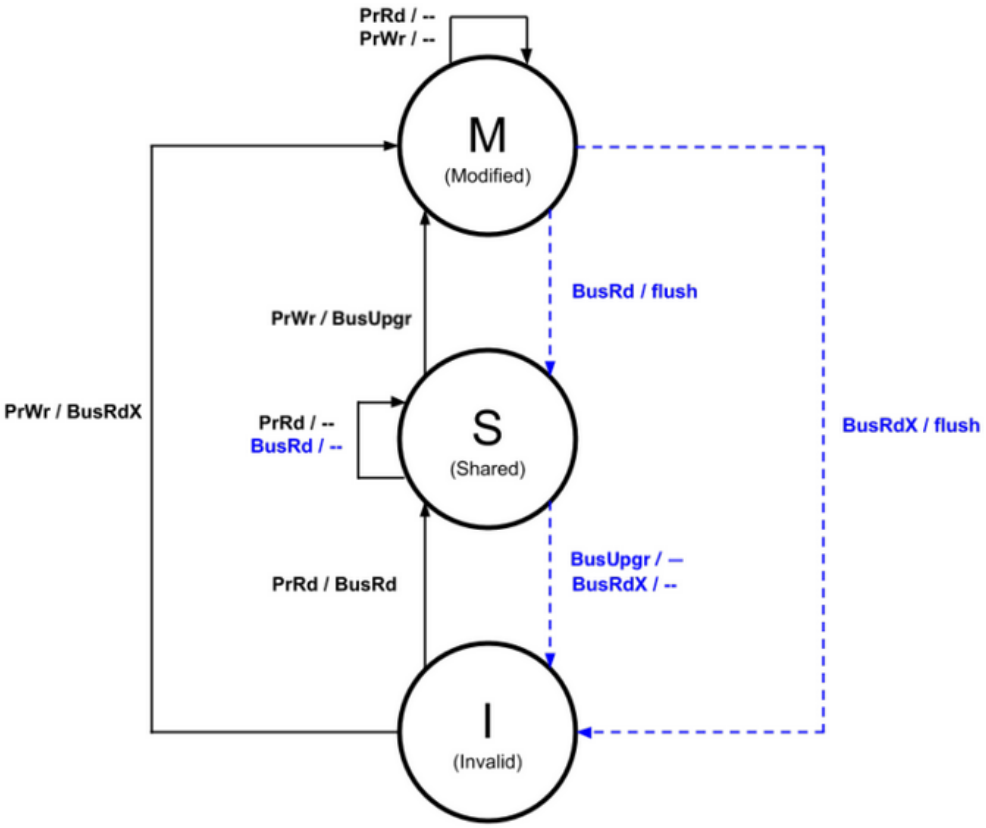


Figura 9. Protocolo MSI. Fuentes: transparencias PAR

Otros protocolos de write invalidate…

Los procesadores actuales utilizan protocolos algo más complejos (y eficientes) que el MSI, pero basándose en el MSI, algunos de ellos son:

- **MOSI:** Aquí hay un estado de "owner". Aquí cuando una caché que tenía una línea en estado M hace flush, no se actualiza memoria principal y la propia línea en esa caché pasa a estado O. Esto vendría a ser como un estado S pero en el que la caché será la responsable de servir el dato a otras caché (en lugar de memoria). Y ya por fin cuando esa línea en estado O desaparezca de la caché será cuando se actualice memoria (evidentemente, para no perder esos datos).

- **MESI:** Si una caché pide el dato para lectura, y es la única con ese dato, pasará de I a E en lugar de I a S. De esta forma, si luego escribe sobre el dato podrá pasar a M sin avisar a nadie (por tanto, menos tráfico en el bus). Si en algún momento estando

en E alguien pide el dato para lectura, pasará de E a S. Todas las demás transacciones no comentadas serán igual que en MSI.

- **MSIF:** Igual que MSI pero ahora cuando una caché quiera un dato, se lo servirá otra caché que lo tenga (si hay alguna). Cuál? Pues la última que hubiese pedido ese dato, que en lugar de estar en S estaría en F. Siempre la última que ha pedido un dato (para lectura) pasará a F en lugar de S, y la que estuviese en F se lo servirá y volverá a S.

- **Y más…** Como por ejemplo combinaciones entre ellas (MOESI, etc).

True y false sharing:

La compartición de datos por diferentes CPUs es lo que causa tráfico de coherencia (invalidaciones de líneas y demás). Esa compartición puede ser verdadera o falsa:

True sharing:

Ocurre cuando dos o más CPUs están compartiendo un mismo dato, por lo cual es inevitable cada uno de los mensajes de coherencia entre cachés que ocurra y cada una de las invalidaciones.

Aún así, muchas veces que hay true sharing se pueden minimizar las acciones de coherencia necesarias (y así hacer más rápido el programa). Por ejemplo:

```
int result = 0;
void dot_product(int *A, int *B, int n) {
   for (int i=0; i< n; i++)
       #pragma omp atomic
       result += A[i] * B[i];
}
```

The line containing variable `result` is subject to coherence actions at each iteration of `i`. It could be easily transformed into

```
void dot_product(int *A, int *B, int n) {
   int tmp = 0;
   for (int i=0; i< n; i++)
       tmp += A[i] * B[i];
    #pragma omp atomic
    result += tmp;
}
```

Figura 10. Mejorando el programa en casos de true sharing. Fuente: transparencias PAR

False sharing:

Ocurre cuando dos o más CPUs están compartiendo una misma línea de memoria pero cada una usa un dato distinto dentro de esa línea. Es decir:
Recordemos que lo del estado I, S y M hace referencia a una línea de caché entera (típicamente 64 bytes). Por lo que si dos procesadores están operando sobre diferentes variables (y las están escribiendo) que están sobre la misma línea de caché, se producirán invalidaciones estúpidas, puesto que, para mantener la coherencia, se irá invalidando esa línea en las demás caches cada vez que una caché quiera escribir en la variable que estaba

tocando, pero todas esas operaciones de coherencia son innecesarias porque ni siquiera están usando el mismo dato, así que “no hay necesidad de que nada sea coherente”.

Nota que para que se considere false sharing, tiene que haber escritura por parte de al menos uno de los procesadores que está operando sobre esa línea de caché en cuestión, ya que, si no, no habrían invalidaciones (solo escrituras producen invalidaciones).

Es muy fácil evitar el false sharing: Simplemente hay que hacer que las variables (distintas) sobre las que operaba cada CPU se situen en líneas distintas. Por ejemplo, añadiendo un ***padding*** entre ellas. Veamos:

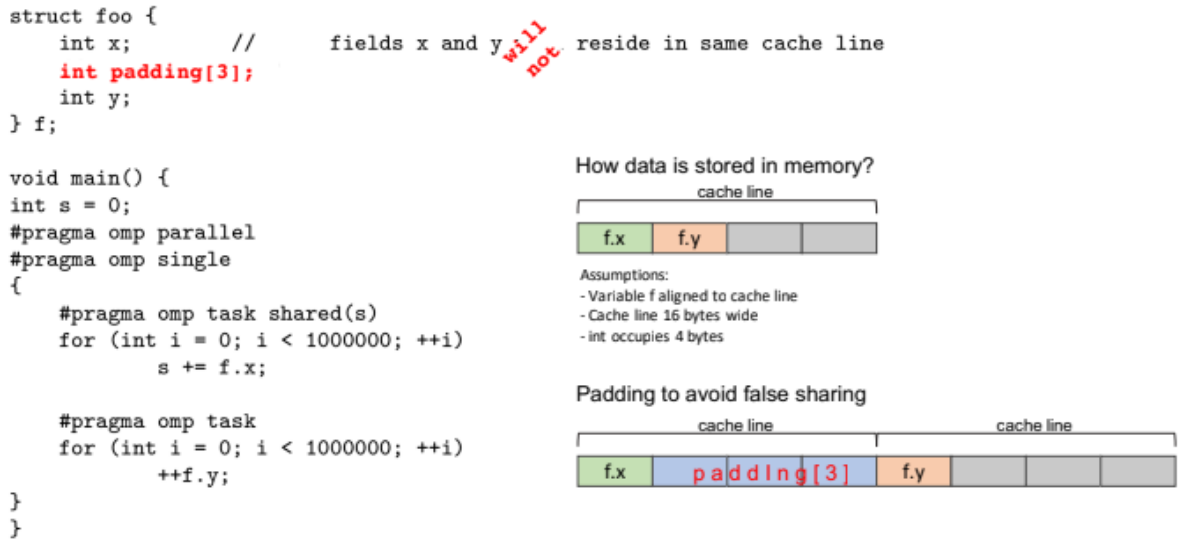


Figura 11. Evitando false sharing. Fuente: transparencias PAR

Fíjate en la figura 11 que si no hubiésemos añadido ese padding tendríamos f.x y f.y en la misma línea de caché, y como f.y se está escribiendo… habrían muchas invalidaciones innecesarias.
En cambio al añadir el padding no se va a producir ninguna invalidación innecesaria porque ahora cada procesador está actuando sobre una línea distinta de memoria.

Coherencia de memoria (en arquitecturas **NUMA**):

Directory Based Coherence:

En el caso de tener un bus común y hacer “snooping”, se genera muchísimo tráfico inútil (ya que todos los procesadores lo ven todo, en lugar de ver exclusivamente lo que les incumbe). Por tanto la otra alternativa, que utiliza comunicación punto a punto en lugar de comunicación broadcast por un bus, es el **directorio**. Un directorio es una estructura de datos que almacenará la información necesaria para mantener la coherencia de un cierto subconjunto de bloques de memoria. De esta forma, cuando un NUMAnode tenga que mandar un mensaje de coherencia relativo a un bloque X, se lo enviará al directorio que se ocupe de ese bloque X. Y luego, ese directorio, mandará otros mensajes de coherencia a los NUMAnode involucrados (si es que hace falta), con las respectivas respuestas de cada uno.

Es decir, supón que NUMAnode0 escribe un dato X. Lo que hará NUMAnode0 será comunicarle al directorio engargado de ese dato que ha hecho una escritura. Entonces, el directorio enviará mensajes de invalidación a otros NUMAnode que tuviesen presente el dato (el directorio conoce qué NUMAnodes son). En caso de que el dato estuviese dirty en algún otro NUMAnode, ese NUMAnode le tendrá que enviar el dato correcto al directorio para que el directorio se lo retorne a NUMAnode0.

Por cada bloque de memoria *(bloque de caché = bloque de memoria)* existirá una entrada de directorio. En ella se almacenará la siguiente información:

- **2 bits** que indicarán el **estado** de ese bloque (U si no está en la caché de ningún NUMAnode, S si sí que está en alguno, pero está limpio el dato, y M si sí que está en 1 NUMAnode y el dato está sucio.
- **P bits**, uno por cada NUMAnode del sistema, que indicarán si el bloque está presente en ese NUMAnode o no (0 si no está presente y 1 si sí lo está).

Cada NUMAnode del sistema contendrá un trozo de la memoria principal global del sistema, y el directorio asociado a ese trozo. Esos trozos serán disjuntos entre ellos y la unión de todos formará la memoria del sistema. Así que cuando hablemos de operaciones de coherencia entre NUMAnodes, distinguiremos estos 3 tipos:

- **Home:** Es el NUMAnode que contiene ese bloque de memoria en su memoria principal, y por tanto el directorio de ese NUMAnode es el encargado de gestionarlo. *OJO: No confundas memoria caché con memoria principal, porque un bloque de memoria está en un solo trozo de memoria (de un solo NUMAnode) pero puede estar en la cache de todos (siempre que esté en estado S).*
- **Local:** Es el NUMAnode que inicia la operación acerca de un bloque, si es necesario. Es decir, es el NUMAnode que le va a enviar una petición al nodo Home.
- **Remote:** Es el NUMAnode que tiene una copia del bloque en cuestión en su caché (o alguna de sus cachés, si tiene varias).

Flujo de comunicación:

Mensajes que llegan del nodo Local al nodo Home:

- **RdReq:** El nodo local no tiene el dato y lo pide para lectura.
- **WrReq:** El nodo local no tiene el dato y lo pide para escritura.
- **UpgrReq:** El nodo local pidió el dato para lectura pero ahora quiere escribirlo, así que envia el mensaje para notificar al nodo Home del cambio de estado.

El nodo Home, responde a Local en cada uno de esos mensajes, con estos otros mensajes:

- **Dreply:** El nodo local pidió un dato con RdReq o WrReq y, por tanto, el nodo Home le devuelve el dato solicitado.
- **Ack:** El nodo local avisó a Home de que se iba a escribir una cierta línea, así que el nodo Home le contesta con un ack (después de invalidar a los remotos que tuviesen esa línea).

El nodo Home, en algunos casos, debe mandar mensajes a nodos Remote:

- **Fetch:** Si Home debe responder con un dato (con Dreply) pero el bloque está en M (lo que significa que hay una copia dirty en otro nodo remoto), el nodo Home mandará un fetch a ese nodo remoto para que le sirva el dato (y, evidentemente, ese nodo remoto pasará de estado M a S).
- **Invalidate:** Si le han solicitado a Home escribir sobre un bloque, Home debe invalidar el bloque en nodos remotos. Para ello mandará un Invalidate.

Los nodos remotos también responderán al Fetch con un **Dreply** y al Invalidate con un **Ack**.

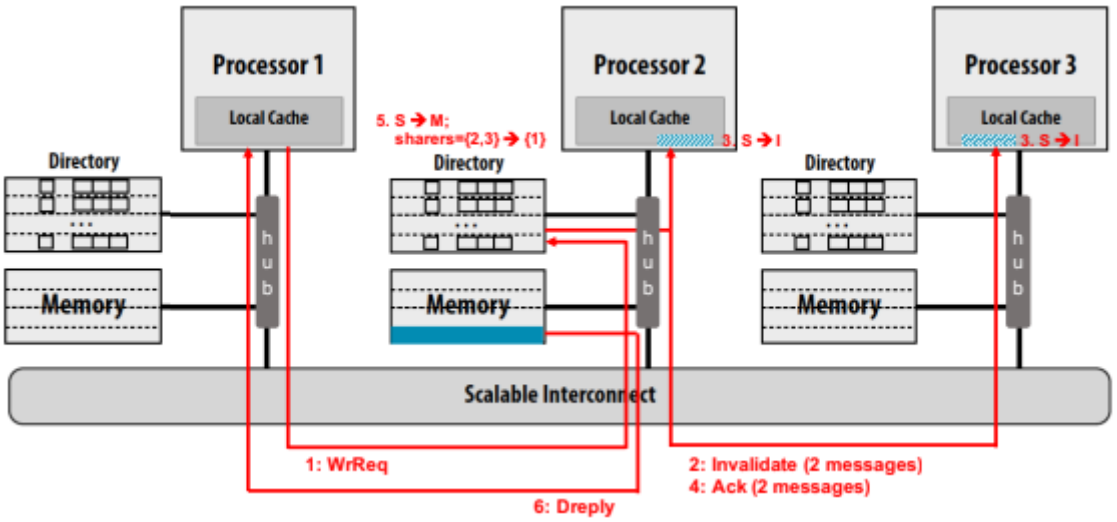


Figura 12. Ejemplo Numa. Fuentes: transparencias PAR

First touch allocation:

Normalmente un bloque estará en la memoria del primer numaNODE que llegase a utilizar ese bloque. Por tanto, piensa en un vector sobre el que van a actuar en partes iguales 4 numaNODE distintos. Si cada numaNODE inicializa la parte del vector sobre la que operará, será mucho mejor que si un solo numaNODE se encarga de inicializar todo el vector. Por qué? Porque si lo inicializa el mismo numaNODE todo el vector estará sobre la memoria de ese nodo, por lo que todos los mensajes de coherencia NUMA serán lanzados contra el mismo nodo, creando cuellos de botella.

Tema 4: Task decomposition strategies

Tipos de tasks decompositions:

Linear:

Son aquellas en las que la tarea es un trozo de código tal cual, un trozo "arbitrario" de nuestro programa, como podría ser el contenido de toda una función.

```
int main() {
   ...
   tareador_start_task("init_A");
   initialize(A, N);
   tareador_end_task("init_A");

   tareador_start_task("init_B");
   initialize(B, N);
   tareador_end_task("init_B");

   tareador_start_task("dot_product");
   dot_product (N, A, B, &result);
   tareador_end_task("dot_product");
   ...
}
```

Figura 1. Linear Task decomposition. Fuente: transparencias PAR.

Iterative:

Tareas que se forman en construcciones iterativas. Es decir, tareas creadas dentro de bucles.

Single loop iteration:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; ii++)
       tareador_start_task("singleit");
       C[i] = A[i] + B[i];
       tareador_end_task("singleit");
}
```

Chunk of loop iterations:

```
#define BS 16
void vector_add(int *A, int *B, int *C, int n) {
   for (int ii=0; ii< n; ii+=BS) {
       tareador_start_task("chunkit");
       for (i = ii; i < min(ii+BS, n), i++)
           C[i] = A[i] + B[i];
       tareador_end_task("chunkit");
       }
}
```

Figura 2. Iterative Task decomposition. Fuente: transparencias PAR.

Los bucles pueden ser **countable** o **uncountable**. Countable es cuando sabemos de antemano cuántas iteraciones se harán y uncountable cuando no. En general un compilador siempre considera los **while como incontables**.

- Ejemplo countable:

```
for (int i = 0; i < N; i++) { //y conocemos el valor de N
        …
}
```

- Ejemplo uncountable:

```
while (p != NULL) {
        …
```

p = p->next; }

Recursive:

Son las tareas encontradas en programas recursivos del tipo divide-and-conquer.

Las estrategias recursivas pueden ser tipo **leaf** o **tree**. En las estrategias leaf hay un thread que va recorriendo el "árbol recursivo" y, al llegar a una hoja (caso base), crea una tarea. En las estrategias tree se crean tareas también en los casos recursivos, es decir, habrán tareas que de lo único que se encargarán será de crear otras tareas y punto. **En una estrategia tree es posible que una tarea se encargue solo de crear otras tareas o es posible que se encargue solo de ejecutar el caso base.**

- Leaf strategy:

```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   }
   else
       {
       tareador_start_task("leaftask");
       vector_add(A, B, C, n);
       tareador_end_task("leaftask");
       }
}
void main() {
   ....
   rec_vector_add(a, b, c, N);
   ...
}
```

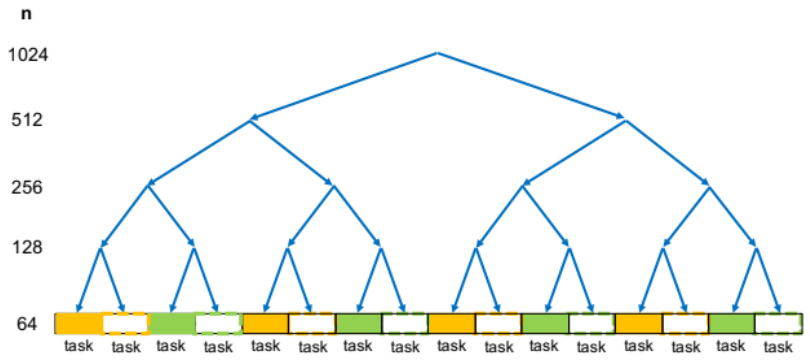


Figura 3. Leaf strategy task decomposition. Fuente: transpas PAR

- Tree strategy:

```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       tareador_start_task("treetask1");
       rec_vector_add(A, B, C, n2);
       tareador_end_task("treetask1");
       tareador_start_task("treetask2");
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
       tareador_end_task("treetask2");
   }
   else vector_add(A, B, C, n);
}
void main() {
   ....
   rec_vector_add(a, b, c, N);
   ...
}
```

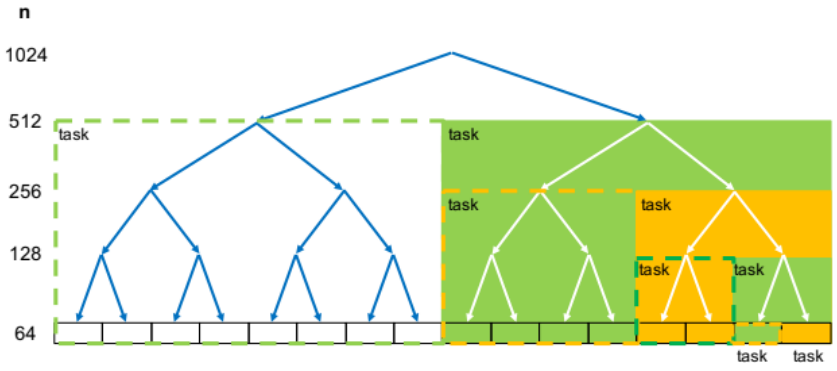


Figura 4. Tree strategy task decomposition. Fuente: transpas PAR

Date cuenta que en la estrategia tree se están creando tareas que se ocuparán de hacer simplemente otra ejecución de esa función recursiva. En esa ejecución de la función recursiva pueden pasar 2 cosas: Que vuelvan a entrar al caso recursivo, y por tanto su único cometido será crear otras tareas y punto, o que les toque entrar en un caso base, y por tanto harían lo mismo que una tarea de estrategia leaf.

En programas muy grandes, se nos puede ir de la mano la creación de tareas en estrategias recursivas, por ello en PAR normalmente tendremos un **control cut-off**.

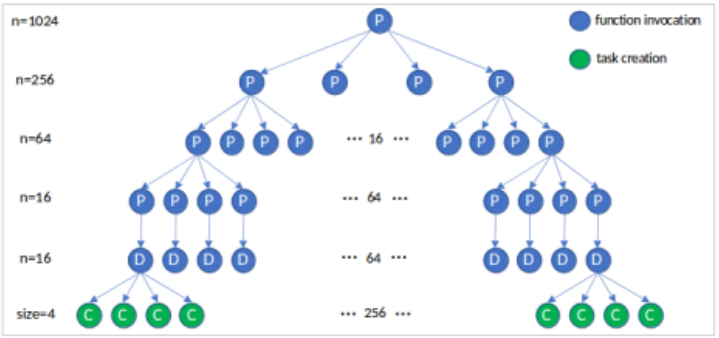
EJEMPLOS:

1) **Para este código, di cuantas tareas se acaban creando. Si definimos la granularidad de una tarea como el número de invocaciones a "compute", ¿cuál es la granularidad de una tarea aquí?**
   Y, asumiendo que cada tarea tarda 16ut y su creación tarda 2ut y que todo el resto del código tarda 0 (es decir, cada nueva llamada a una función tarda 0), ¿cuánto vale Tinf? *(a la derecha de la figura 5 está la respuesta).*

```
#define N 1024
#define MIN 16

void doComputation (int * vector, int n) {
   int size = n / 4;
   for (int i = 0; i < n; i += 4)
      tareador_start_task("compute");
      compute(&vector[i], size);
      tareador_end_task("compute");
}

void partition (int * vector, int n) {
   if (n > MIN) { // MIN is multiple of 4
      int size = n / 4;
      for(int i=0; i<4; i++)
         partition(&vector[i*size], size);
      }
   else
      doComputation(vector, n);
   return;
}
void main() {
   ...
   partition (vector, N); // N is multiple of 4
   ...
}
```



#tasks = 256, granularity = 1, T_∞ = (256 x 2) + 16 = 528

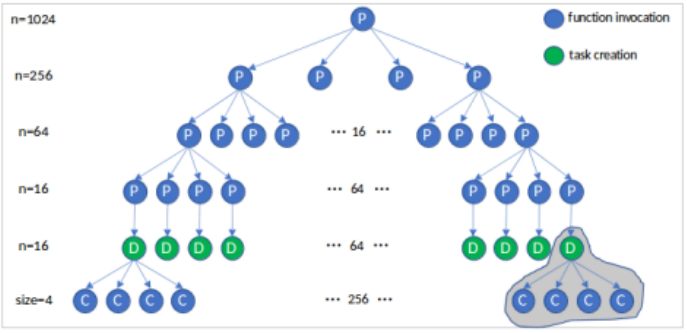
Figura 5. Ejemplo 1. Transparencias profe Dani Jiménez.

2) **Ahora el código se cambia un pelín. Si ahora decimos que la granularidad de una tarea es el número de invocaciones a "compute", ¿cuál es la granularidad de una tarea?**
   Y, asumiendo que cada ejecución de compute tarda 16ut y la creación de una tarea tarda 2ut y que todo el resto del código tarda 0, ¿cuánto vale Tinf?

```
#define N 1024
#define MIN 16

void doComputation (int * vector, int n) {
   int size = n / 4;
   for (int i = 0; i < n; i += 4)
      compute(&vector[i], size);
}

void partition (int * vector, int n) {
   if (n > MIN) { // MIN is multiple of 4
      int size = n / 4;
      for(int i=0; i<4; i++)
         partition(&vector[i*size], size);
      }
   else
      tareador_start_task("doComputation");
      doComputation(vector, n);
      tareador_end_task("doComputation");
   return;
}
void main() {
   ...
   partition (vector, N); // N is multiple of 4
   ...
}
```



#tasks = 64, granularity = 4, T_∞ = (64 x 2) + (4 x 16) = 192

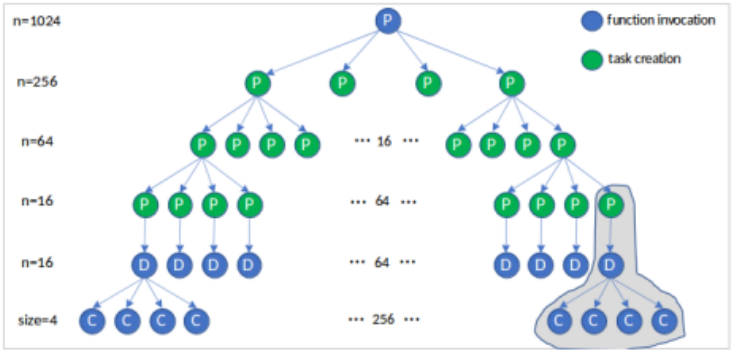
Figura 6. Ejemplo 2. Transparencias profe Dani Jiménez.

3) **Ahora pasamos de tener una estrategia leaf a una estrategia tree. El master thread crea 4 tareas y cada una de esas tareas se dedica a crear otras tareas que van creando tareas hasta que llegamos al punto en el que una tarea creada es la encargada de entrar en el caso base en lugar del caso recursivo.**
Con las mismas suposiciones que antes, contesta lo mismo que se ha preguntado antes.

```
#define N 1024
#define MIN 16

void doComputation (int * vector, int n) {
   int size = n / 4;
   for (int i = 0; i < n; i += 4)
      compute(&vector[i], size);
}

void partition (int * vector, int n) {
   if (n > MIN) { // MIN is multiple of 4
      int size = n / 4;
      for(int i=0; i<4; i++)
         tareador_start_task("partition");
         partition(&vector[i*size], size);
         tareador_end_task("partition");
      }
   else
      doComputation(vector, n);
   return;
}
void main() {
   ...
   partition (vector, N); // N is multiple of 4
   ...
}
```



#tasks=84, granularity=4, T_∞ = (3 x 4 x 2) + (4 x 16) = 88

Figura 7. Ejemplo 3. Transparencias profe Dani Jiménez.

OpenMP:

Directivas más importantes:

 # pragma omp parallel -> Se crea una implicit task para cada thread (y es inmediatamente ejecutada)

 # pragma omp single -> Su contenido lo ejecuta solamente el primer thread de la región paralela que entra en el single.

 # pragma omp task -> Se crea una explicit task que se va a la pool de threads y será ejecutada cuando algún thread del equipo la coja de esa pool de tasks.

 # pragma omp taskloop -> para que en un bucle se creen varias tareas. Implicitamente se privatiza la variable de inducción de ese bucle. Se le pueden especificar cosas como el número de tareas que queremos que se creen.

pragma omp [ALGO] **firstprivate(var)** -> Se crea una copia privada de *var* en la región en la que está ese firstprivate. La copia privada coge el valor que tuviese la variable (compartida) en ese instante.

pragma omp [ALGO] **private(var)** -> Se crea una copia privada de *var* en la región en la que está ese firstprivate. La copia privada NO coge el valor que tuviese la variable (compartida) en ese instante.

IMPORTANTE:

- Una variable declarada DENTRO del parallel, al crear una task, será firstprivate por defecto.
- Una variable declarada FUERA del parallel, al crear una task, será shared por defecto.
- Por tanto, en general, una variable declarada dentro de parallel es privada (y declarada dentro de task, obviamente, también, y con más razón). Y si es declarada fuera del parallel será compartida al usarse en cualquier parte de ese parallel.

 # pragma omp taskwait -> Línea para esperar a que se acabe la ejecución de las tareas que se acaban de crear. Es decir, si un thread crea 5 tareas y se encuentra esta línea, deberá esperar a que acaben de ejecutarse esas 5 tareas. **OJO: El thread solo esperará a las tareas que ha creado (o sea, a sus hijas), NO a otras tareas descendientes de esas.**

 # pragma omp taskgroup -> El thread espera a que acabe la ejecución de las tareas que ha creado en esa región **Y TAMBIÉN A SUS DESCENDIENTES**.

 # pragma omp barrier -> Sirve para esperar a que todos los threads acaben su trabajo y lleguen a esa barrera.

Atomic vs Reduction vs Critical vs Locks:

 # pragma omp atomic -> Para operaciones, en general, de lectura + modificación + escritura de forma atómica. También se puede hacer para solo escritura o solo lectura (pragma omp atomic write / pragma omp atomic read).
Las operaciones de modificación permitidas son solamente operaciones básicas:
+, *, -, /, &, ^, |, <<, >>.

pragma omp [ALGO] **reduction(operación : variable)** -> Sirve para acompañar a algunas directivas, como por ejemplo taskloop. Indica que la variable que se está modificando (sobre la que habría data race si no se controla) se privatizará sobre la ejecución de cada tarea y, al final, se hará la *operación* pertinente sobre la variable compartida (como si fuera un atomic). Es lo más eficiente de todo.
Ejemplo:

```
#pragma omp taskloop reduction(+: result) grainsize(4)
for (int i = 0; i < n; i++) {
        result += A[i]*B[i]
}
```

 # pragma omp critical -> Sirve para hacer que el código de la región crítica solo lo ejecute un thread a la vez. Bastante ineficiente, pero a veces no hay más remedio. Existen los "criticals con nombre". Si le damos dos nombres a 2 criticals distintos, puede entrar un thread en uno y otro en el otro. Si no les diésemos nombre, Solo se podría entrar en 1. **IMPORTANTE:** Ese nombre no se puede calcular en tiempo de ejecución. Es decir, no se puede conseguir la mejora que nos da omp_set_lock con lo de los nombres de los criticals.

omp_set_lock(&var_lock) -> Sirve para casi lo mismo que el critical y es igual de eficiente que el critical. Entonces, cuál es la diferencia? La veremos en clase comparando estos códigos:
(Imagina que v es un vector donde cada posición contiene una lista encadenada de elementos. Y function es una función donde se añade un nuevo elemento a la lista de v[index]. Por lo tanto debemos bloquear todo el acceso a v[index] porque varios threads podrían intentar añadir algo ahí a la vez)

```
**** Tenemos una región paralela ****
while (.....) { // bucle ejecutado por varios threads
        # pragma omp critical {
                function(v[index], …);
        }
}
```

```
**** Tenemos una región paralela ****
while (.....) { // bucle ejecutado por varios threads
        omp_set_lock(&v_locks[index]);
        function(v[index], …);
        omp_unset_lock(&v_locks[index]);
}
```

Tareas recursivas:

Ya se ha explicado que existen 2 tipos: Tree y leaf.
Para las tareas recursivas hay que tener en cuenta varias cosas:

1- Cuidado con data races! Es decir, si todos los casos base están haciendo, por ejemplo, result += algo_calculado, habrá que protegerlo con un atomic. Otra opción es convertir esas funciones void en funciones int que nos retornen ese “algo_calculado” y así evitar tener que poner una línea atómica de result += algo_calculado. En este caso, será el padre el que recoja el int que retornan sus hijos y hacer con ello lo que toque.
2- Si hemos hecho esa solución de pasar void a int… Cuidado! El padre no podrá hacer nada hasta que sus hijos hayan retornado algo correcto. Por tanto, habrá que añadir taskwaits! En la figura 8 se muestra.
Y no solo eso: También tendremos que poner explícitamente “shared” la variable que va a retornar el hijo en la tarea, para que el padre tenga acceso a ese valor! (Si no, cualquier variable creada en una tarea dentro de un parallel es firstprivate por defecto, por lo que el padre jamás se enteraría del valor). En la figura 8 se muestra.

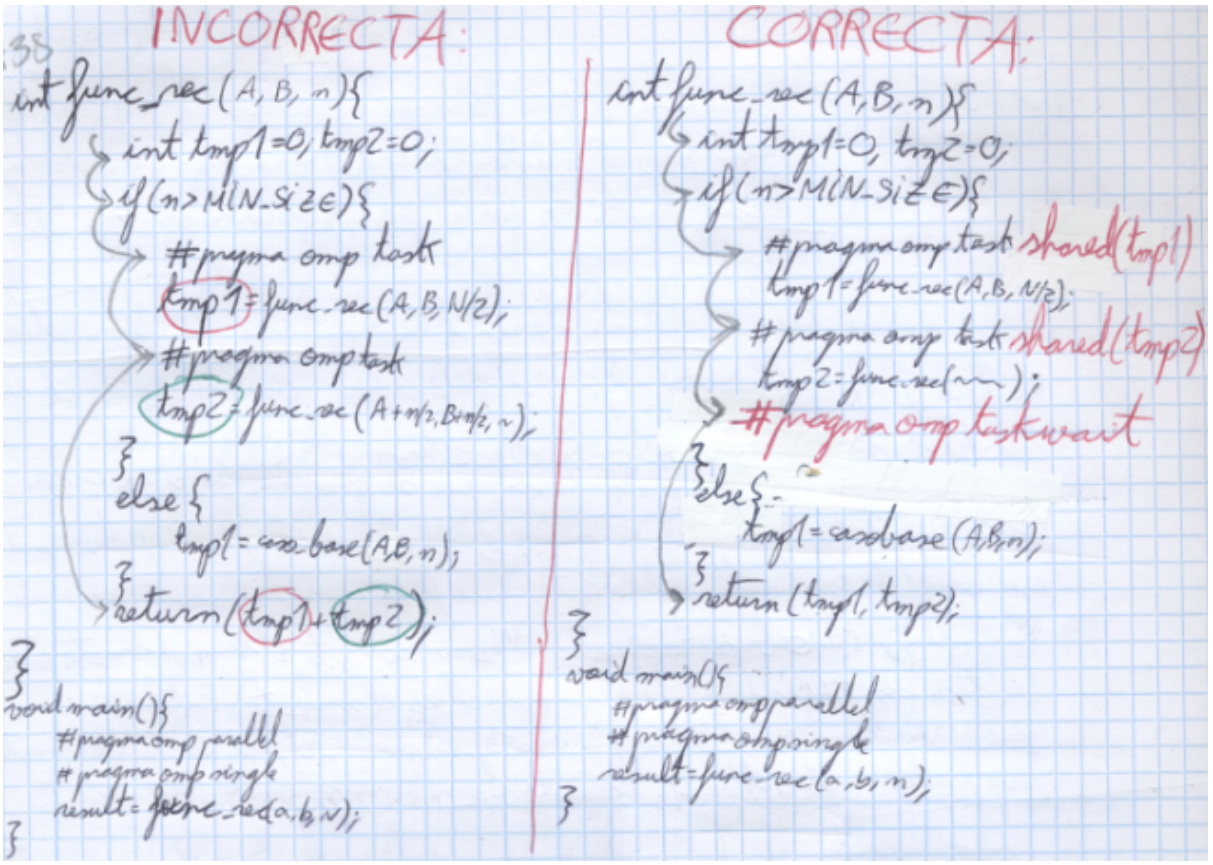


Figura 8. Estrategia tree recursiva con funciones que NO son void.

En resumen:
Si la hacemos void: Necesitaremos atomics como mínimo (y, depende del caso, pueden haber más necesidades).
Si la hacemos int: Necesitamos taskwait + shareds como mínimo (y, depende del caso, pueden haber más necesidades).

Lo que he dicho es válido para tree y para leaf. La figura 8 muestra para tree, pero en el caso de leaf es lo mismo solo que en el caso recursivo no hay tareas y en el caso base sí (y es allí donde pondríamos el shared y taskwait).

CUT-OFF:

Para evitar crear demasiadas tareas, necesitamos tener mecanismos de CUT-OFF. Esto es, dejar de crear tareas cuando tengamos demasiadas. **NOTA:** No confundas CUT-OFF con la variable que se mira para ver si se entra en el caso base o en el recursivo (que esa variable sería MIN_SIZE en la Figura 8).

Leaf strategy with **depth recursion control**

```
#define CUTOFF 2
...
void rec_dot_product(int *A, int *B, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth == CUTOFF)
            #pragma omp task
            {
                rec_dot_product(A, B, n2, depth+1);
                rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            }
        else {
            rec_dot_product(A, B, n2, depth+1);
            rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else    // if recursion finished, need to check if task has been generated
        if (depth <= CUTOFF)
            #pragma omp task
            dot_product(A, B, n);
        else
            dot_product(A, B, n);
}
...
```

Figura 9. CUT-OFF estrategia LEAF. Fuentes: transpas PAR

Tree strategy with **depth recursion control**

```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

int rec_dot_product(int *A, int *B, int n, int depth) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
           #pragma omp task shared(tmp1)
           tmp1 = rec_dot_product(A, B, n2, depth+1);
           #pragma omp task shared(tmp2)
           tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
           #pragma omp taskwait
        } else {
           tmp1 = rec_dot_product(A, B, n2, depth+1);
           tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
   }
   else tmp = dot_product(A, B, n);
   return(tmp1+tmp2);
}
```

Figura 10. CUT-OFF estrategia TREE (además con taskwait y shared por ser de tipo int). Fuentes: transpas PAR.

Para el CUT-OFF, además, se pueden usar la cláusula *final* y la función *omp_in_final()*:

Making use of omp_in_final:

```
#define MIN_SIZE 64
#define CUTOFF 3
...
int rec_dot_product(int *A, int *B, int n, int depth) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       if (!omp_in_final()) {
           #pragma omp task shared(tmp1) final(depth >= CUTOFF)
           tmp1 = rec_dot_product(A, B, n2, depth+1);
           #pragma omp task shared(tmp2) final(depth >= CUTOFF)
           tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
           #pragma omp taskwait
       } else {
           tmp1 = rec_dot_product(A, B, n2, depth+1);
           tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
       }
   }
   else tmp1 = dot_product(A, B, n);
   return(tmp1+tmp2);
}
...
```

Figura 11. final + Omp_in_final en CUT-OFF. Fuente: transpas PAR

El ejemplo que vemos en la figura 11 es para el caso de TREE. Lo que podemos ver es que se crea una tarea activando el flag de *final* si y solo si depth >= CUTOFF. Para las tareas que se hayan creado con ese flag activado, tendremos que cuando se ejecuten darán “false” en “!omp_in_final()”.

Depend:

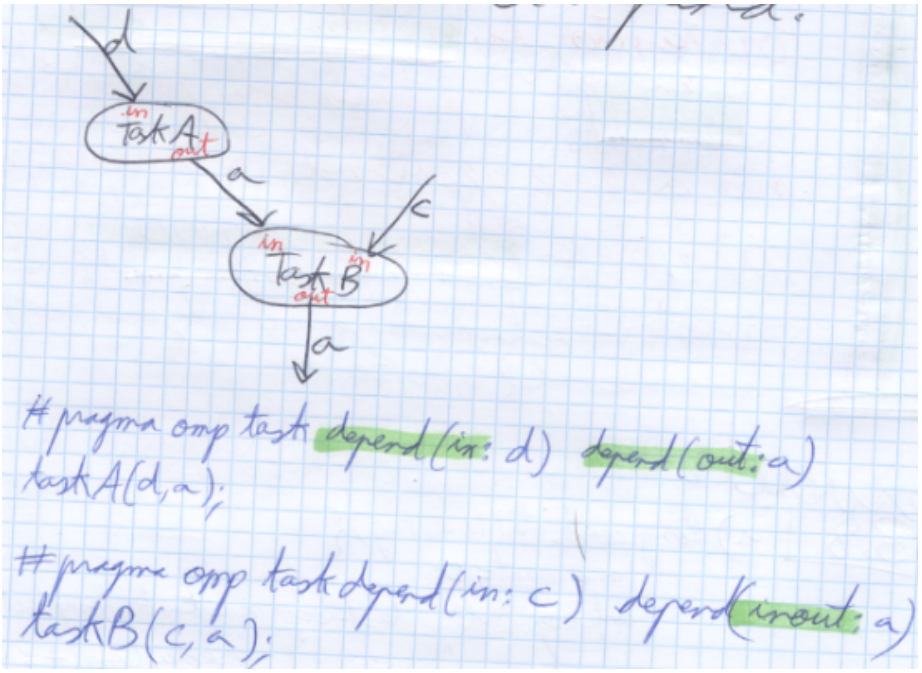


Figura 12. Depend.

Gracias al depend conseguiremos que OpenMP sepa en qué momento tiene que ejecutar cada cosa en runtime, por lo que el programador tendrá menos faena de ir controlándolo con barreras.

Reglas del depend:
IN: Se espera a que todas las tareas con depends de tipo INOUT y OUT sobre esa variable que hayan antes en el programa finalicen. De esta forma se cumplen las dependencias RaW (dependencias verdaderas).
OUT: Se espera a que todas las tareas con depends de tipo IN, OUT e INOUT sobre esa variable que hayan antes en el programa finalicen. De esta forma se cumplen las dependencias WaR (antidependencia) y WaW.
Por tanto la gestión de antidependencias.

IMPORTANTE: Para detectar Dependencias (ya sean RaW, WaR o WaW) el compilador solo mira que haya coincidencia en la dirección de las variables implicadas (y, en el caso de vectores, en la dirección inicial y la cantidad de elementos implicados). Por tanto:

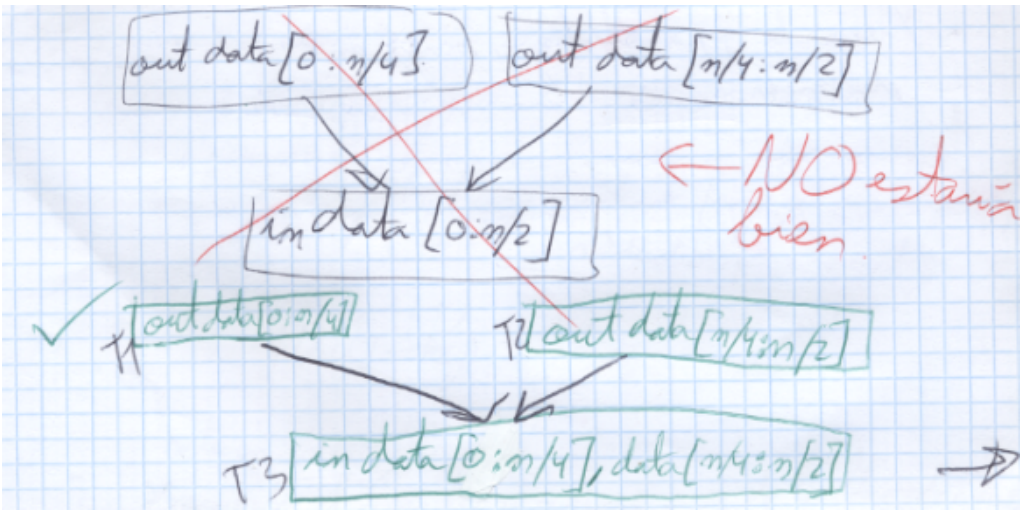


Figura 13. Depend.

Como vemos en la figura 13, si ponemos depend(in: data[0:n/2]) el compilador no será capaz de asociarlo ni con data[0:n/4] ni con data[n/4:n/2]. Y lo que sí es correcto (color verde de la figura 13) es equivalente a haber puesto “depend(out: data[0])”, “depend(out: data[n/4]” y “depend(in: data[0], data[n/4]) porque ahí sigue coincidiendo la dirección inicial de la dependencia y la longitud (1 elemento), así que el compilador entiende que hay una true dependency entre esas tareas.

Soporte HW para la sincronización:

Imagina el siguiente código (ejecutado por varias CPUs) para intentar tener acceso exclusivo a una variable:

```
            // omp_lock_t flag
init_lock:  st flag, 0
            ...
set_lock:   ld r1, flag
            bnez r1, set_lock
            st flag, #1
           ...
            // safe access
           ...
unset_lock: st flag, #0
            ...
```

Supón que flag=0 indica que NO hay nadie dentro de la región crítica que se quiere proteger (esa región crítica está marcada como "//safe access" en el código) y flag=1 indica que sí que hay que esperarse.

Lo que están haciendo todas las CPUs en ese código es:
"leo el contenido de flag, compruebo si es 0, en caso de que NO sea 0 vuelvo a saltar a la línea de leer el contenido de flag y en caso de que sí sea 0 avanzo hacia delante (poniendo el flag a 1 y accediendo a la región crítica); al salir de la región crítica pongo flag a 0".

¿Ves algún problema?

El problema es que la lectura de flag (y posterior escritura en caso de que valga 0) no se hace atómicamente, por lo que podría suceder que varias CPUs lean flag=0 y por tanto todas ellas hagan store flag 1 y entren a la región crítica.

¿Cómo lo solucionamos?

Tenemos soporte hardware para estas situaciones:

test-and-set

En la instrucción test-and-set se realiza atómicamente la lectura y escritura de flag, por lo que deja de existir el problema comentado anteriormente!
Ahora las líneas de la parte de "set_lock" del código anterior cambiarían por:
set_lock: // es una simple etiqueta de x86 (recordad AC).
t&s r1, flag // comprueba si la dirección de memoria "flag" contiene un 0. En caso afirmativo se pone un 1 dentro de "flag" y se hace r1 = 0. En caso negativo se deja "flag" tal como está (es decir, con un 1) y se hace r1 = 1. Para que lo veas más fácilmente: r1 se carga con el contenido de "flag" previo a la modificación del "set".
bnez r1, set_lock // si r1 vale distinto a 0 (o sea, 1), salta a la etiqueta set_lock para volver a intentar el t&s.

Exchange

Intercambia (atómicamente) el contenido de un registro y el de una posición de memoria:

```
            mov r2, #1
set_lock:   exch r2, flag
            bnez r2, set_lock
```

En el código puedes ver como se pone r2=1 (que es lo que queremos poner en flag para coger acceso excusivo) y se intercambia ese 1 por el contenido actual de flag. En caso de que flag ya valiese 1 (alguien tenía cogido el acceso) no habrán cambios sobre flag y volveremos a saltar a la etiqueta de set_lock. En caso de que flag valiese 0 estaríamos poniéndolo a 1 y pudiendo avanzar hacia la zona de exclusión mútua.

Fetch-and-op

Algunas operaciones del tipo lectura-modificación-escritura permiten hacer atómicamente. No hay una instrucción para eso sino que se pone un prefijo a la instrucción. Por ejemplo, si en x86 pones el prefijo “LOCK” antes de un add [mem] %eax, conseguirás que la operación de [mem] = [mem] + %eax se haga atómicamente. Con esto se implementa el **atomic**, y por eso el atomic solo permitía un conjunto pequeño de operaciones de tipo lectura-op-escritura.

Mejorar estrategias de test-and-set: test-test-and-set

Lo malo de estas instrucciones de acceso exclusivo es que no son nada eficientes: Ese “set” que se hace es una escritura con todo lo que conlleva: invalidaciones por el bus (que inducen tráfico e invalidan a quien tenga ya en su caché información de que test vale 1, por ejemplo). Esto es aplicable a t&s, exchange, etc… Por tanto, optaremos por estrategias del tipo test-test-and-set:

```
set_lock:   ld r2, flag

            bnez r2, set_lock
            t&s r2, flag
            bnez r2, set_lock
```

Se trata de hacer, primero de todo, una lectura de “flag” normal y corriente. Si flag=1 ni siquiera intentaremos el t&s porque no tendría ningún sentido. Si flag=0 sí intentaremos el t&s (o exchange o lo que queramos). Nota que como no hay atomicidad entre ese primer test y ese segundo test-and-set nos podría pasar que en el primer load veamos que flag=0 pero luego, al hacer t&s, veamos que flag=1. En ese caso no pasará nada malo más allá de invalidar la línea de caché por culpa de ese “set”, pero esto ocurrirá muchas menos veces que si siempre hiciéramos un test-and-set sin otro test previo.

Load linked store conditional (ll-sc)

“Load ligado a un store condicional”. **NO es una instrucción atómica.** La atomicidad puede hacer bajar mucho el rendimiento! el LL-SC consiste en lo siguiente:

- LL retorna el valor de una cierta posición de memoria (como un load normal) pero marca esa posición para indicar que alguien ha hecho un LL.
- SC hace store solamente si se cumple la condición de que esa posición sigue marcada como que alguien ha hecho un LL. Al completarse el store, se borra la información de que se había hecho LL sobre ella.

Por tanto, varias CPUs pueden hacer LL sobre la misma posición de memoria (todas leyendo lo mismo), pero solamente la primera que intente escribir en esa posición podrá completar su SC. De esta forma aseguramos atomicidad en la operación load-store de la CPU a la que le hemos permitido escribir, sin necesidad de haber tenido que usar una operación atómica (más ineficiente).
NOTA: Si alguien hace un store normal y corriente también se borra la información de que alguien había hecho LL.

Tema 5: Data-aware task decomposition strategies

Dividir el trabajo por tareas tiene un problema: no se puede controlar qué CPU ejecutará cada tarea. Por tanto, si tenemos una tarea que toca todo el rato las mismas posiciones de memoria (de un vector o matriz) nos interesa que se ejecute siempre en la misma CPU para aprovechar los datos de su caché. ¿Cómo podemos conseguir eso? Con tareas implícitas.

La estructura típica de tareas explícitas es:
pragma omp parallel
pragma omp single
pragma omp task (varias).

La estructura que utilizaremos ahora para tareas implícitas es:
pragma omp parallel, y punto.
De esta forma, sin un single, todos los threads ejecutarán todo el código de la región paralela. Y será allí dentro, en la región paralela, donde controlaremos los valores que tendrán las variables de inducción de los bucles para cada thread para así asegurarnos que un thread se encargue solamente de las posiciones de memoria que nosotros queramos.

Para ello vamos a utilizar muchísimo estas 2 funciones:
omp_get_num_threads(); //retorna número de threads en la región (NUM_THREADS).
omp_get_thread_num(); //retorna identificador de thread (de 0 a NUM_THREADS-1).

Formas de distribuir los datos:

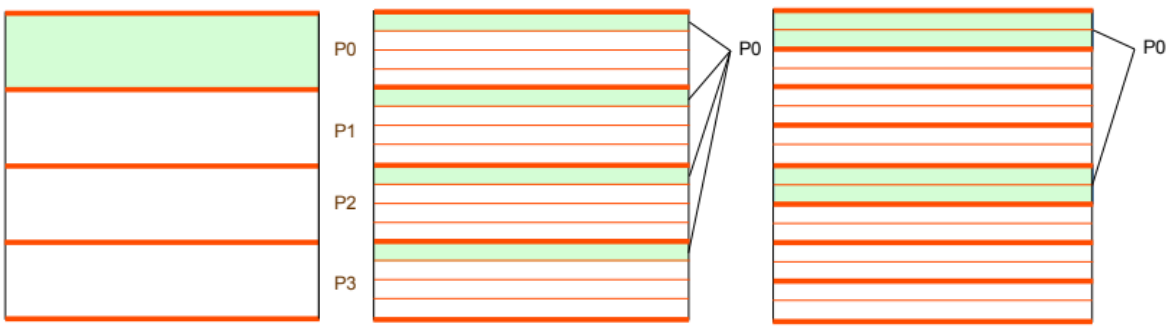


Figura 14. Distribuciones geométricas: block, cyclic y block-cyclic. Fuente: transpas PAR

Block

El código típico que haríamos para controlar las iteraciones de cada thread en una distribución por bloques es el siguiente:

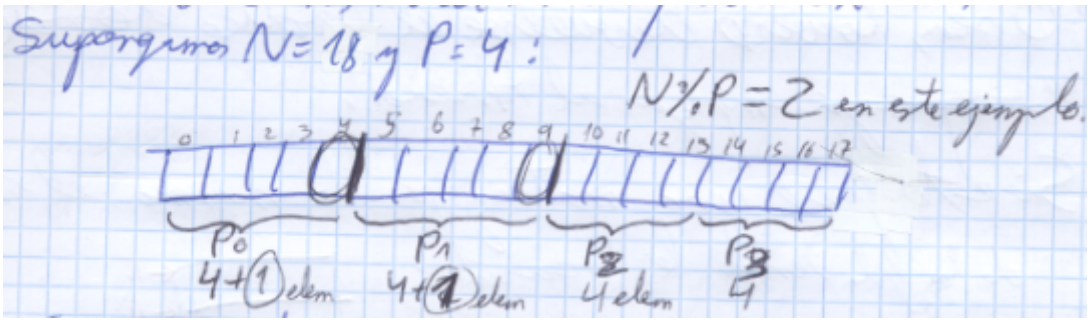
```
int myid = omp_get_thread_num();
int numprocs = omp_get_num_threads();
int i_start =  myid * (MATSIZE/numprocs);
int i_end = i_start + (MATSIZE/numprocs);
if (myid == numprocs-1) i_end = MATSIZE;
```

Pero ese código tiene un problema: El último thread puede tener un gran desbalanceo de carga. Imagínate que MATSIZE = 24 y numprocs = 5. Con ese código tendríamos que cada thread se encargaría de 4 filas mientras el último se encargaría de 8! (las 4 que de por sí le tocaban más las 4 que sobraron al hacer 24/5). Por tanto, el código que utilizaremos para conseguir un desbalanceo de, como máximo, 1 fila (o elemento, según el problema) es el siguiente:

```
int myid = omp_get_thread_num();
int numprocs = omp_get_num_threads();
int i_start =  myid * (MATSIZE/numprocs);
int i_end = i_start + (MATSIZE/numprocs);
int rem = MATSIZE % numprocs;
if (rem != 0) {
    if (myid < rem) {
        i_start += myid;
        i_end += (myid+1);
    } else {
        i_start += rem;
        i_end += rem;
    }
}
```

Ese código, más resumidamente, es:
i_start = id*(N/P) + (id < (N%P) ? id : N%P);
num_elem = N/P + (id < (N%P));
i_end = i_start + num_elem;

Esto lo que consigue es:



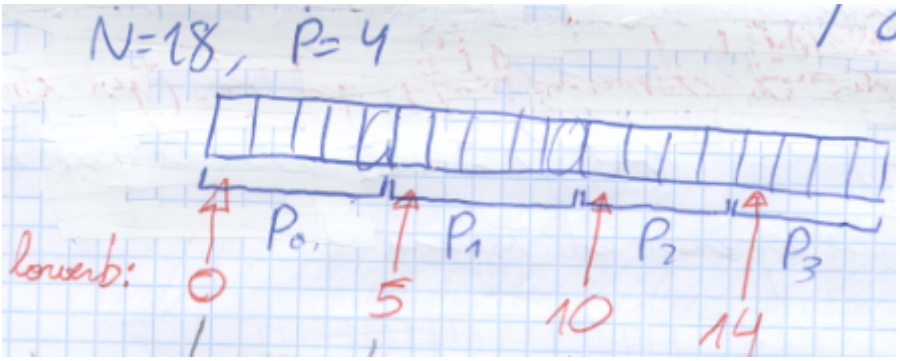
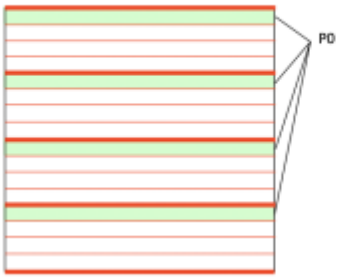


Figura 15. Descomposición block con balanceo mejorado

Cyclic

CYCLIC DATA DECOMPOSITION, by ROWS




```
#pragma omp parallel private(i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for(i=my_id; i<N; i+=howmany)
        for (j=0; j<N; j++) {
    ...
}
```

Descomposición cíclica viene muy bien para situaciones como esta:

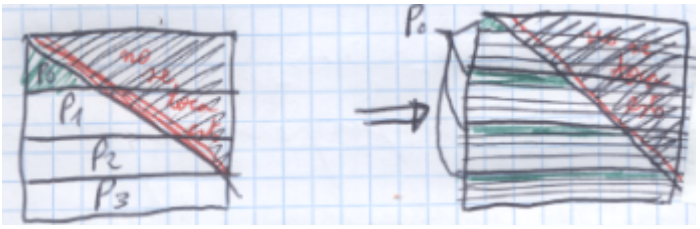
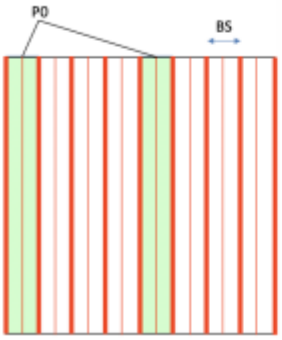


Figura 16. Mejor balanceo de trabajo gracias a cyclic

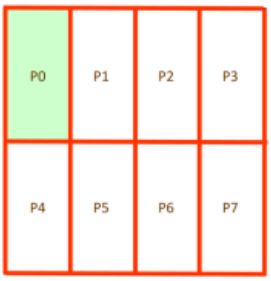
Block-cyclic

BLOCK-CYCLIC DATA DECOMPOSITION, by COLUMNS




```
#pragma omp parallel private(i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    for(i=0; i<N; i++)
        for (jj=my_id*BS; jj<N; jj+=howmany*BS) {
            for (j=jj; j<jj+BS; j++)
        ...
    }
}
```

Descomposición por bloques en 2D




```
sum = 0;
#pragma omp parallel private(ii,i,j) reduction(+:sum) num_threads(8)
{
   int my_id = omp_get_thread_num();
   int block_i = my_id/4; // 0 or 1 indicates which row of blocks
   int block_j = my_id%4; // 0,1,2,3 indicates which column of blocks
   int BSi = N/2;         // Assume N is multiple of 2
   int BSj = N/4;         // Assume N is multiple of 4
   int i_start = block_i * Bsi;  // where should thread start?
   int i_end = i_start + BSi;
   int j_start = block_j * Bsj;
   int j_end = j_start + BSj;
   for(i=i_start; i<i_end; i++)
     for (j=j_start; j<j_end; j++)
        sum+= f(Matrix_in[i][j])
}
```

Fíjate que las variables block_i y block_j son como el my_id pero en 2D.
Si N no fuese múltiple (es decir, la repartición de trabajo no pudiese ser perfecta) haríamos una repartición igual que vimos en la sección de block (al fin y al cabo esto sigue siendo distribución por bloques, pero en 2D). Básicamente habría que cambiar las líneas de BS (num_elem) y i/j_start:

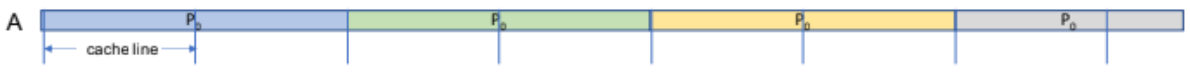
int BSi = N/2 + (block_i < (N%2));
int BSj = N/4 + (block_j < (N%4));
int i_start = block_i * N/2 + (block_i < N%2 ? block_i : N%2);
int j_start = block_j * N/4 + (block_j < N%4 ? block_j : N%4);

Evitar false sharing

A continuación se muestran diferentes ejemplos con sus posibles soluciones, sacadas de las transparencias de PAR:

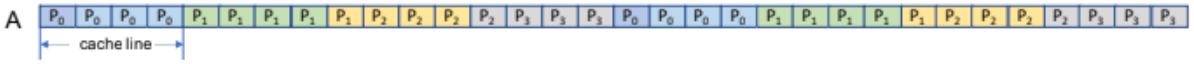
```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    int BS = n / omp_get_num_threads();
    for (i=myid*BS; i<(myid+1)*BS; i++) A[i] = foo(i*23);
}
```

Possible solution: introduce some load unbalance, so that BS corresponds with a number of elements that fit in a number of complete cache lines

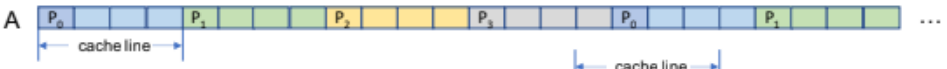


```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    for (i=myid; i<n; i+=howmany) A[i] = foo(i*23);
}
```

Possible solution: make larger chunk size (p.e. 4) → block-cyclic

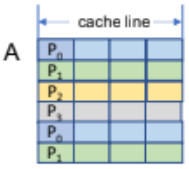


Alternative solution: Add padding – i.e. one element per cache line



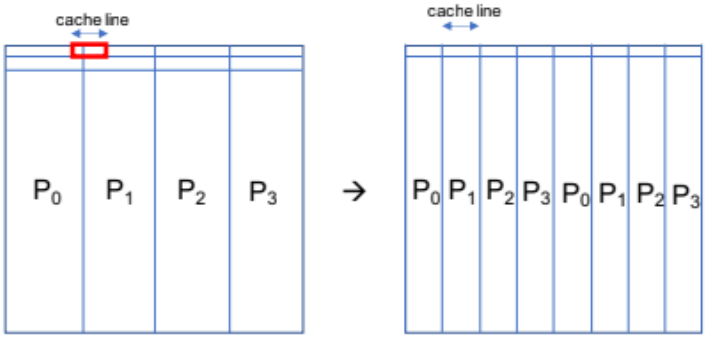
How? `int A[100];` → `int A[100][4];`

And the access needs to change …
`A[i][0] = foo(i*23)`



In 2D matrices we can also have false sharing problems … solutions?

- block → block-cyclic



- Add some padding