Memoria de la práctica 4 de

Arquitectura de ordenadores

EJERCICIO 0: información sobre la topología del sistema

```
e342222@localhost:~$ cat /proc/cpuinfo | grep 'siblings\|core'
siblings        : 4
core id         : 0
cpu cores       : 4
siblings        : 4
core id         : 1
cpu cores       : 4
siblings        : 4
core id         : 2
cpu cores       : 4
siblings        : 4
core id         : 3
cpu cores       : 4
```

Tenemos 4 cores reales (*cpu_cores*) y 4 virtuales (*siblings*), por lo que no hay *hyperthreading*.

EJERCICIO 1: Programas básicos de OpenMP

```
e342222@14-26-66-229:~/Desktop/materialP4$ ./omp1 4
Hay 4 cores disponibles
Me han pedido que lance 4 hilos
Hola, soy el hilo 0 de 4
Hola, soy el hilo 2 de 4
Hola, soy el hilo 3 de 4
Hola, soy el hilo 1 de 4
```

1. **¿Se pueden lanzar más threads que cores tenga el sistema?¿Tiene sentido hacerlo?**

Se pueden lanzar tantos hilos como uno quiera, pero lanzar más del número de cores disponibles no sale rentable porque los hilos tendrán que compartir recursos entre ellos y al final no se estarán ejecutando de forma simultánea.

2. **¿Cuántos threads debería utilizar en los ordenadores del laboratorio?¿y en su propio equipo?**

En los equipos del laboratorio deberíamos utilizar como mucho 4 hilos. En nuestro propio equipo deberemos de usar 2 hilos ya que disponemos de dos núcleos.

3. **¿Cómo se comporta OpenMP cuando declaramos una variable privada?**

```
e342222@14-26-66-229:~/Desktop/materialP4$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7fff3aea8d34,x   &b = 0x7fff3aea8d38,    &c = 0x7fff3aea8d3c

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7fff3aea8d08,    &b = 0x7fff3aea8d38,    &c = 0x7fff3aea8d04
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 1]-1: a = -1,      b = 2,  c = 3
[Hilo 1]         &a = 0x7f11f74cde58,    &b = 0x7fff3aea8d38,    &c = 0x7f11f74cde54
[Hilo 1]-2: a = 22,      b = 6,  c = 4
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f11f64cbe58,    &b = 0x7fff3aea8d38,    &c = 0x7f11f64cbe54
[Hilo 3]-2: a = 27,      b = 8,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f11f6ccce58,    &b = 0x7fff3aea8d38,    &c = 0x7f11f6ccce54
[Hilo 2]-2: a = 33,      b = 10,         c = 3

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7fff3aea8d34,    &b = 0x7fff3aea8d38,    &c = 0x7fff3aea8d3c
```

Podemos observar en la captura que para cada variable privada crea 4 copias (una por hilo) con direcciones distintas. Dentro de la región paralela las variables privadas pueden tener valores distintos para cada hilo, pero cuando se sale se descartan los cambios y la variable queda con el valor inicial.

4. **¿Qué ocurre con el valor de una variable privada al comenzar a ejecutarse la región paralela?**

Si se declara como *firstprivate* las copias de la variable se inicializan con el valor que tenían antes de entrar en la región paralela. Si por el contrario se declara como *private,* se inicializan a 0.

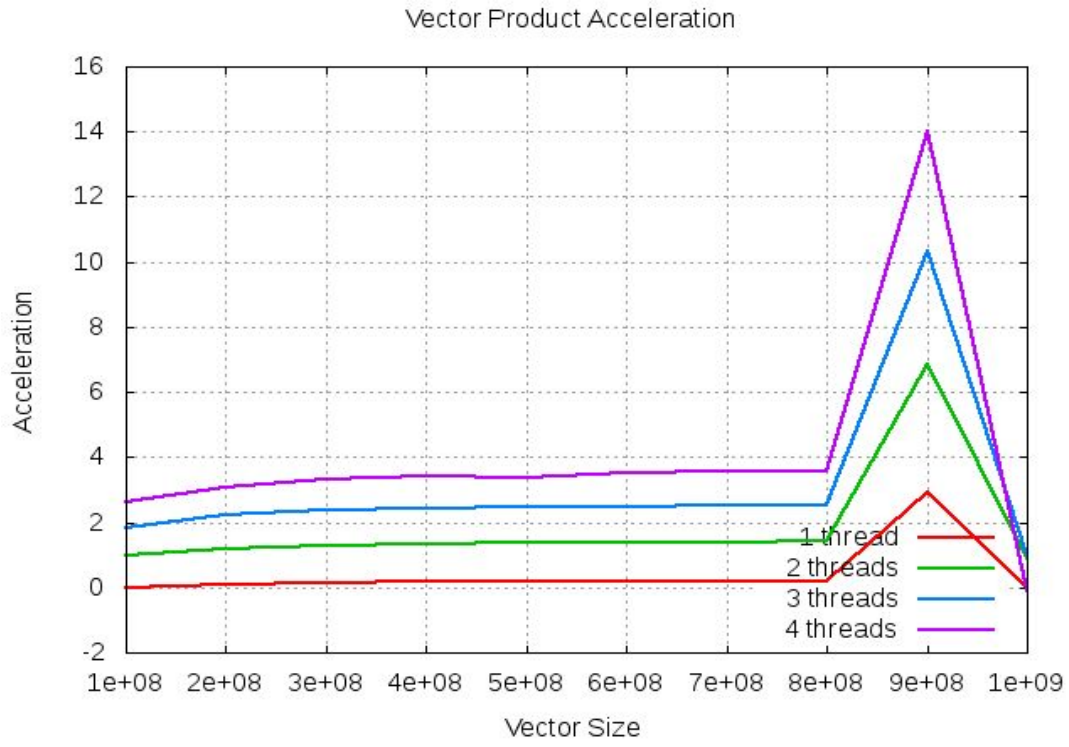
5. **¿Qué ocurre con el valor de una variable privada al finalizar la región paralela?**

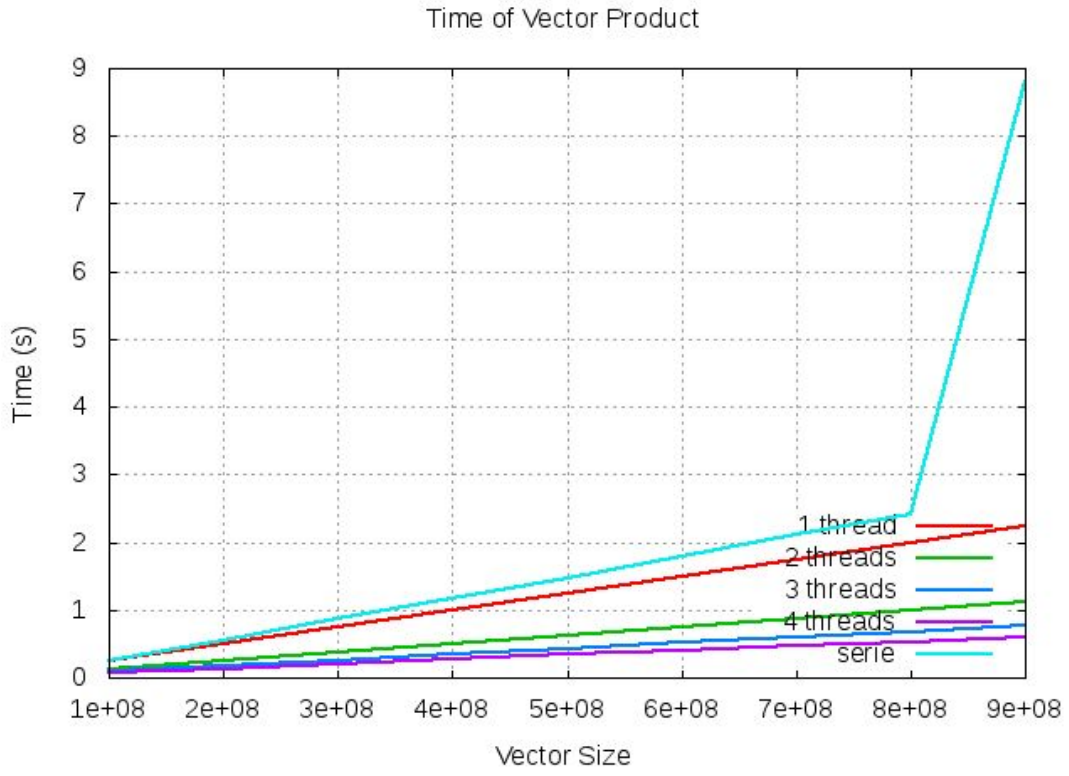
El valor de una variable privada al salir de la región paralela sigue teniendo el mismo valor que antes de entrar (porque lo que modifican los hilos son las copias en otras posiciones de memoria).

6. **¿Ocurre lo mismo con las variables públicas?**

No, si la variable es pública, todos los hilos comparten la misma dirección para ella y, por tanto, modifican la misma posición de memoria.

<u>EJERICIO 2:</u>Paralelizar el producto escalar


Vector Product Acceleration
Acceleration
Vector Size
16
14
12
10
8
6
4
2
0
-2
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
7e+08
8e+08
9e+08
1e+09
1 thread
2 threads
3 threads
4 threads


Time of Vector Product
Time (s)
Vector Size
9
8
7
6
5
4
3
2
1
0
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
7e+08
8e+08
9e+08
1 thread
2 threads
3 threads
4 threads
serie

1. **¿En qué caso es correcto el resultado?**

Sabemos que el programa de producto en serie nos va a dar el resultado correcto (33.319767). La primera versión en paralelo nos da un resultado completamente diferente (13.103340), y la segunda versión en paralelo da como resultado lo mismo que la versión serie (33.329441; varía algún decimal pero no es muy significativo).

2. **¿A qué se debe esta diferencia?**

Lo único que cambia entre las dos versiones en paralelo es que en la segunda se añade la cláusula *reduction(+:sum)*. Su función es asegurarse de que ningún hilo acceda al valor de la variable mientras otro esté actualizándolo. Al no estar presente en la primera versión, pueden darse situaciones del tipo:

1. El hilo 0 lee el valor de sum, que es 0.
2. El hilo 3 lee el valor de sum, que aún es 0.
3. El hilo 0 suma 0 + *a*, y guarda *a* en sum.
4. El hilo 3 suma 0 + *b*, y guarda *b* en sum, sobrescribiendo el resultado del hilo 0.

De esta manera, el resultado obtenido es considerablemente menor que el esperado, ya que la mayoría de los resultados de las sumas parciales son sobrescritos por los de otros hilos.

3. y 4. **En términos del tamaño de los vectores,¿compensa siempre lanzar hilos para realizar el trabajo en paralelo, o hay casos en los que no?Si compensara siempre,¿en qué casos no compensa y por qué?**

Para tamaños de vectores relativamente pequeños generar múltiples hilos y mantenerlos es más costoso que hacer la propia operación de vectores en serie, así que no merece la pena.

5. y 6. **¿Se mejora siempre el rendimiento al aumentar el número de hilos a trabajar?Si no fuera así,¿a qué debe este efecto?**

Para el número de hilos que representamos en las gráficas, la respuesta es sí, aunque no siempre tanto como sería deseable (caso de cuatro hilos). Sin embargo, sabemos que si seguimos aumentando el número de hilos hasta superar el de hilos virtuales que soporta el procesador, no conseguiríamos un rendimiento mayor.

7. **Valore si existe algún tamaño del vector a partir del cual el comportamiento de la aceleración va a ser muy diferente del obtenido en la gráfica.**

Observando el resultado de la gráfica se puede decir que el rendimiento va a ser estable ya que todas las funciones son no decrecientes, es decir, parece que van a seguir creciendo o, al menos, mantenerse estables sin llegar a decrecer. Esto se poduce a que el tamaño del vector crece de manera lineal y los correspondientes cálculos que lleva a cabo el programa crece a ese mismo ritmo.

EJERCICIO 3:

Tiempos de ejecución de 10 segundos:

Version\#hilos	1	2	3	4
Serie	8.337139			
Paralela-bucle1	7.776786	7.419696	4.181896	4.146556
Paralela-bucle2	6.666281	3.349639	2.27683	1.870316
Paralela-bucle3	6.787529	3.416436	2.331513	1.821752

Aceleración para 10 segundos(tomando como referencia la versión serie):

Version\#hilos	1	2	3	4
Serie	1			
Paralela-bucle1	1.072054	1.123649	1.993626	2.010617
Paralela-bucle2	1.250643	2.4889066	3.661731	4.457609
Paralela-bucle3	1.228302	2.440302	3.575849	4.57644

Tiempos de ejecución de 60 segundos:

Version\#hilos	1	2	3	4
Serie	58.550052			
Paralela-bucle1	52.504249	32.01989	23.893813	28.936271
Paralela-bucle2	46.995931	23.774293	16.12669	13.184709
Paralela-bucle3	47.896655	24.371354	16.479295	12.838173

Aceleración para 60 segundos (tomando como referencia la versión serie):

Version\#hilos	1	2	3	4
Serie	1			
Paralela-bucle1	1.115148	1.828552	2.450427	2.023413
Paralela-bucle2	1.245853	2.462746	3.63063	4.440754
Paralela-bucle3	1.2248	2.402412	3.552946	4.560621

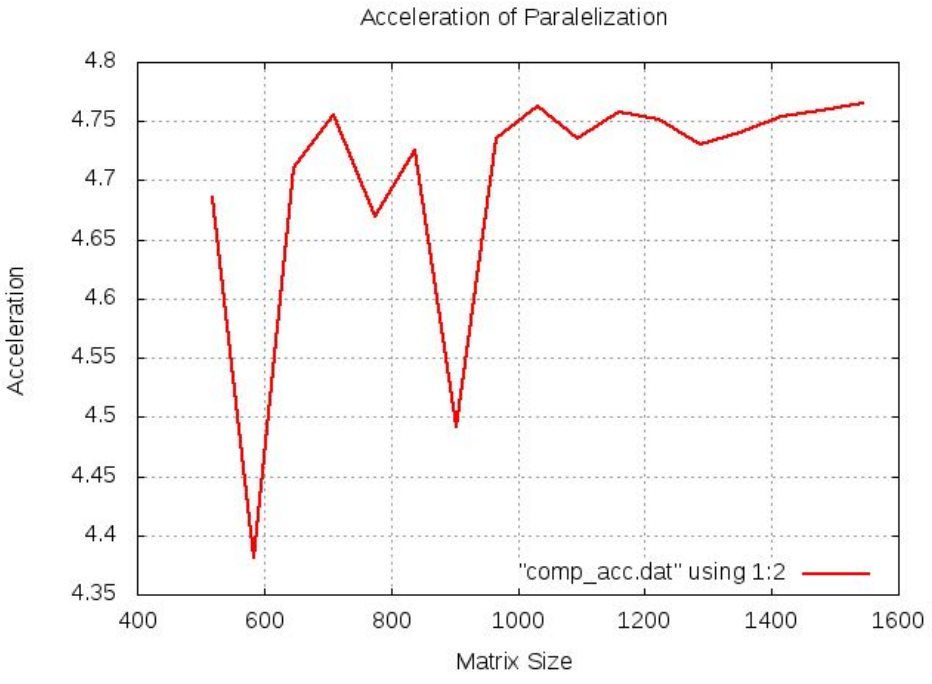
1. y 2. **¿Cuál de las tres versiones obtiene peor rendimiento?¿A qué se debe?¿Cuál de las tres versiones obtiene mejor rendimiento?¿A qué se debe?**

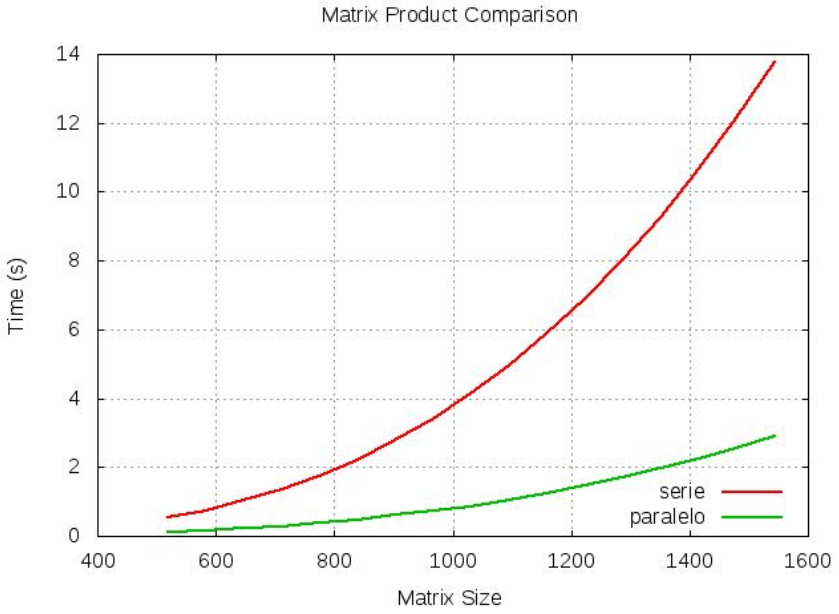
La versión que obtiene peor rendimiento es la paralela-bucle 1, mientras que la que obtiene el mejor rendimiento es la palabra-bucle 3.

Si paralelizamos uno de los bucles internos, tendremos que lanzar los hilos al comienzo de cada iteración del bucle superior, y unirlos al final de la iteración. Como sabemos, unir los hilos es costoso ya que se tiene que esperar al más lento, y por eso tarda más la paralelización de los bucles interiores.

Al paralelizar el bucles exterior se despliegan los hilos una vez al comienzo del bucle y no se unen hasta que termina el bucle completo. Por esta razón, es mucho más eficiente.

Tomando como referencia los tiempos de ejecución de la versión serie y el de la mejor versión paralela obtenida anteriormente(la mejor combinación entre las tres versiones de código,y las 4 posibilidades para el número de hilos paralelos).Tome tiempos en un fichero y realice una gráfica de la evolución del tiempo de ejecución y la aceleración(de la versión paralela vs serie) al ir variando el tamaño de las matrices de NxN para N entre 512+P y 1024+512+P (con incrementos en N de 64).Tras incluir estas dos gráficas en la memoria, incluya un párrafo describiendo y justificando el comportamiento observado en las mismas.





Podemos observar que cuando el tamaño de la matriz aumenta, así lo hacen también los tiempos, sin embargo, la versión en serie tiene un crecimiento mucho mayor, ya que va de los 0.2 segundos a los 14, la versión en paralelo no llega a alcanzar los 4 segundos con el mayor tamaño analizado. Sin embargo, fijándonos en la gráfica de la aceleración nos damos cuenta de que al principio crece muy rápido, es decir, para pasar de una aceleración de 2 a una de 3, simplemente necesita un incremento en la matriz de 400x400, pero a partir del tamaño 900x900 aproximadamente, la aceleración se mantiene más o menos estable alrededor de una aceleración de 4.75, esto ocurre porque el tiempo de ejecución de un producto de matrices tiene un crecimiento exponencial respecto a la dimensión de la matriz. Por otra parte, la reducción de tiempo que supone utilizar varios hilos no crece tan rápido, por este motivo, la diferencia de tiempo se estabiliza.

3. **Si en la gráfica anterior no obtuvo un comportamiento de la aceleración en función de N que se estabilice o decrezca al incrementar el tamaño de la matriz,siga incrementando el valor de N hasta conseguir una gráfica con este comportamiento e indique para que valor de N se empieza a ver el cambio de tendencia.**

El valor de N para el que se consigue estabilizar la aceleración es 900.

EJERCICIO 4: Ejemplo de integración numérica

1. **¿Cuántos rectángulos se utilizan en la versión del programa que se da para realizar la integración numérica?**

El programa utiliza 100.000.000 (10^8) rectángulos.

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par1
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.518666
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.336258
```

2. **¿Qué diferencias observa entre estas dos versiones?**

En el programa *pi_par1.c* cada hilo guarda las sumas parciales en una posición diferente del mismo array.
En *pi_par4.c* cada hilo guarda las sumas parciales en una variable privada.

3. **Ejecute las dos versiones recién mencionadas.¿Se observan diferencias en el resultado obtenido?¿Y en el rendimiento?Si la respuesta fuera afirmativa,¿sabría justificar a qué se debe este efecto?**

El resultado obtenido al ejecutar los dos programas es el mismo, pero el programa *pi_par4.c* se ejecuta mucho más rápido que *pi_par1.c*.

Esta diferencia se debe al efecto conocido como *false sharing*. Dado que en *pi_par1.c* el array de sumas parciales sólo va a tener cuatro posiciones, lo más probable es que esté completamente contenido en un bloque de caché (de forma que no quede parte al final de un bloque y parte al comienzo del siguiente). Por tanto, cada vez que un hilo modifique un valor del array los demás hilos tendrán que descartar su versión del array y cargar la nueva con el valor actualizado. Como esto va a ocurrir muchas veces, se pierde mucho tiempo recargando el mismo bloque. En *pi_par4.c,* por el contrario, cada hilo ,modifica una variable privada independiente y por lo tanto no se da este efecto.

4. **Ejecute las versiones paralelas 2 y 3 del programa.¿Qué ocurre con el resultado y el rendimiento obtenido?¿Ha ocurrido lo que se esperaba?**

De nuevo, el resultado es el mismo en las dos versiones, y también hay diferencia en el rendimiento.

La versión *pi_par2* hace privada para cada hilo la variable *sum*, pero esto no tiene ningún efecto ya que sólo almacena el puntero al array que se modifica, por lo que se produce *false sharing* y el tiempo es similar al obtenido con *pi_par1*.

En cambio, el cambio de *pi_par3* es que en lugar de reservar espacio en el array para 4 doubles (uno para las sumas de cada hilo), se reserva un array de una longitud de 4 bloques de caché, de tal manera que hay un bloque entero para cada hilo. Si bien esto no asegura que el primer cuarto del array esté íntegro en un bloque de caché, al utilizar sólo el comienzo (los primeros *sizeof(double)* bytes de cada cuarto), es bastante poco probable

que uno de los sumadores quede dividido entre dos bloques de caché. Esto elimina el *false sharing* por completo, y por eso el tiempo se parece al de *pi_par4*.

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par2
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.457994
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.328488
```

5. **Abra el fichero pi_par3.c y modifique la línea 32 del fichero para que tome los valores fijos 1, 2, 4, 6, 7, 8, 9, 10 y 12. Ejecute este programa para cada uno de estos valores.¿Qué ocurre con el rendimiento que se observa?**

Lo que esperaríamos que ocurriera es que el tiempo de ejecución fuera decreciendo según aumenta el padding, y que quedara estable al pasar de 8 (el número de *doubles* que caben en un bloque de caché). El padding representa el número de *doubles* que se le asigna al fragmento del array de cada hilo. Si es 1, entonces estamos en la situación de *pi_par1*, y si es 8 estamos en la situación de *pi_par3*. Según aumentamos este número, estamos aumentando la distancia entre las posiciones en las que realmente escribe cada hilo, y con eso reducimos la probabilidad de que estén todas en el mismo bloque de caché y se dé *false sharing*. Si lo aumentamos a más de 8, entonces le estaremos dando más de un bloque a cada hilo, y esto dejará de tener efecto en el rendimiento, porque los hilos siempre escribirán en bloques distintos.

En líneas generales, se cumplen nuestras predicciones, aunque hay algunas variaciones. Con un padding menor que 8, es posible que se dé *false sharing*. En cierto modo, el rendimiento también va a variar ligeramente dependiendo de la “suerte” que tengamos al asignar la memoria al array. Las posibilidades serían cautro:

- Que todas las posiciones donde escriben los hilos entren en el mismo bloque.
- Que las posiciones donde escriben los tres primeros (o últimos) hilos caigan en el mismo bloque, y la del otro hilo en el siguiente.
- Que las posiciones donde escriben los dos primeros hilos caigan en un bloque y las de los dos últimos hilos en el siguiente.
- Que la posición en la que escribe el primer hilo caiga sola en un bloque, las del segundo y tercer hilo caigan en el siguiente bloque, y la del cuarto hilo en otro distinto.

En el primer caso todos los hilos se estorban entre ellos. En el segundo, son tres los que se estorban y el cuarto se ejecuta por libre sin obstrucciones. En el tercer caso, los hilos se estorban por parejas: los dos primeros entre sí, y los dos últimos entre sí. En el último caso, sólo un par de hilos se estorban entre ellos, y los otros dos se ejecutan libremente.

Este puede ser el motivo de las variaciones que observamos respecto al resultado esperado.

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 1 elementos
Resultado pi: 3.141593
Tiempo 0.518128
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 2 elementos
Resultado pi: 3.141593
Tiempo 0.489373
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 4 elementos
Resultado pi: 3.141593
Tiempo 0.576541
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 6 elementos
Resultado pi: 3.141593
Tiempo 0.372536
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 7 elementos
Resultado pi: 3.141593
Tiempo 0.520574
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.465656
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 9 elementos
Resultado pi: 3.141593
Tiempo 0.450229
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 10 elementos
Resultado pi: 3.141593
Tiempo 0.388733
```

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 12 elementos
Resultado pi: 3.141593
Tiempo 0.332499
```

EJERCICIO 5: Uso de la directiva critical y reduction

1. **Ejecute las versiones 4 y 5 del programa. Explique el efecto de utilizar la directiva critical.¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?**

La cláusula *critical* hace que el código siguiente no se pueda ejecutar en más de un hilo a la vez. Con esto se evitan problemas de sincronización entre los hilos.
El programa *pi_par5,* a parte de tardar considerablemente más que *pi_par4*, da un resultado distinto.

Al principio, no entendíamos del todo por qué ocurre ninguna de las dos cosas. Las variables *x* y *sum* son privadas, por lo que no deberían dar ningún problema. La única variable compartida es *pi*, pero sólo se modifica bajo la cláusula *critical*, y por tanto tampoco debería presentar ningún problema. Además, esta cláusula no debería añadir demasiado tiempo a la ejecución, ya que cada hilo sólo pasa por ella una vez. La única explicación que se nos ocurre para que el resultado sea erróneo es que la variable *pi* no está inicializada, y eso podría modificar el resultado final.

Sin embargo, después nos fijamos en que la variable *i* sobre la que se itera en el bucle también es compartida. Al ser así, todos los hilos están escribiendo sobre la misma variable, y esto hace que algunos hilos hagan iteraciones correspondientes a otros, o que haya iteraciones que simplemente no se realizan. Por esta razón el resultado es incorrecto. Además, al escribir sobre la misma variable, de nuevo se da un efecto similar al *false sharing* que hace que se pierda tiempo.

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.344722
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par5
Resultado pi: 3.024308
Tiempo 1.449689
```

2. **Ejecute las versiones 6 y 7 del programa. Explique el efecto de utilizar las directiva utilizadas.¿Qué diferencias de rendimiento se aprecian?¿A qué se debe este efecto?**

En *pi_par6* se utiliza la cláusula *for*. Esta cláusula gestiona por nosotros el reparto de las iteraciones de un bucle entre los hilos. Sin embargo, se está volviendo a utilizar un array de *floats* para almacenar las sumas parciales de cada hilo y esto produce *false sharing*, por eso el rendimiento no es tan bueno.

En *pi_par7* se usa *for*, y además *reduction*. Esta cláusula se encarga de aplicar la operación que se le indica (en este caso la suma) para reducir a un sólo valor los contenidos en cada hilo de la variable que se le indica (en este caso *sum*). Esta implementación no da ningún problema, ya que *for* se encarga del reparto de iteraciones correctamente y *reduction* hace que no haya colisiones entre las *sum* de distintos hilos, y se encarga de sumarlas todas al final.

```
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par6
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.497429
e336543@localhost:~/Desktop/LUIA/materialP4$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.334329
```