Practica 4 Arquitectura de Ordenadores

Lucía Asencio y Juan Riera

Diciembre 2017

0. Información sobre la topología del sistema

Con el comando `cpuinfo` pudimos comprobar que, en los ordenadores del laboratorio, disponemos de 4 cpu_cores y 4 siblings (4 reales y 4 virtuales). A partir de aquí, sabemos que, en caso de existir *hyperthreading*, este no está habilitado.

Para comprobar si existía, buscamos en internet y comprobamos que el procesador no disponía de *hyperthreading*.

De los 4 cores reales, 3 son de 800MHz y 1 de 1000MHz

1. Programas básicos de OpenMP

```
e337572@localhost:~/ArqO_molaconlusi/practica4/material$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7fff1dad6454,x   &b = 0x7fff1dad6458,    &c = 0x7fff1dad645c

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7fff1dad6428,    &b = 0x7fff1dad6458,    &c = 0x7fff1dad6424
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 1]-1: a = -1,      b = 2,  c = 3
[Hilo 1]         &a = 0x7f74f07dce58,    &b = 0x7fff1dad6458,    &c = 0x7f74f07dce54
[Hilo 1]-2: a = 22,      b = 6,  c = 4
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f74ef7dae58,    &b = 0x7fff1dad6458,    &c = 0x7f74ef7dae54
[Hilo 3]-2: a = 27,      b = 8,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f74effdbe58,    &b = 0x7fff1dad6458,    &c = 0x7f74effdbe54
[Hilo 2]-2: a = 33,      b = 10,         c = 3

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7fff1dad6454,    &b = 0x7fff1dad6458,    &c = 0x7fff1dad645c
```

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

El programa nos permite lanzar tantos hilos como queramos dentro de cierto margen (32037 es el máximo que nos ha permitido lanzar), pero como no disponemos de *hyperthreading*, no tendría sentido hacerlo ya que cada core no permitiría la ejecución de más de un thread a la vez.

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

A la vista de la salida de `cpuinfo`, sabemos que el número idóneo de hilos es 4.

No disponemos de un ordenador personal,y encontramos de un elitismo más característicos del feudalismo que de la Edad Contemporánea la incorrecta presuposición de que disponemos de tal equipo.

Era bromis, no hemos podido contenernos. Con `cpuinfo`, obtenemos que nuestro equipo consta de 4 cores reales y 8 virtuales, por lo cual el número ideal de threads a utilizar es 8.

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

Las variables privadas declaradas como "private" apuntarán a una dirección de memoria cualquiera que tendrá valores residuales. Por otro lado si están declaradas como `firstprivate` tendrán el valor de incialización de la variable con el mismo nombre presente fuera de los threads.

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

Las variables delcadaras con `private` se deben inicializar en cada hilo, si no, ocurrirá como ocurre en la ejecución del programa, que tendrán valores indeterminados. En este caso tienen 0 o -1 porque apuntan a direcciones que contienen todo 0's o todo F's. Si se declara con `firstprivate` no hay este problema, ya que la variable queda con el valor de inicialización original.

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

Como observamos, las direcciones de memoria de las variables privadas son distintas de las de las originales, tanto en el caso de las declaradas con "private" como en el de las declaradas con `firstprivate`. Por ello, su valor se pierde al terminar la ejecucion del thread.

6 ¿Ocurre lo mismo con las variables públicas?

En el caso de las variables públicas, sus direcciones de memoria son las mismas que las de las variables originales. Por esa razón cuando se editan en lo threads se edita la variable original, y por tanto cuando acaban los threads se mantiene el valor.

2. Paralelizar el producto escalar

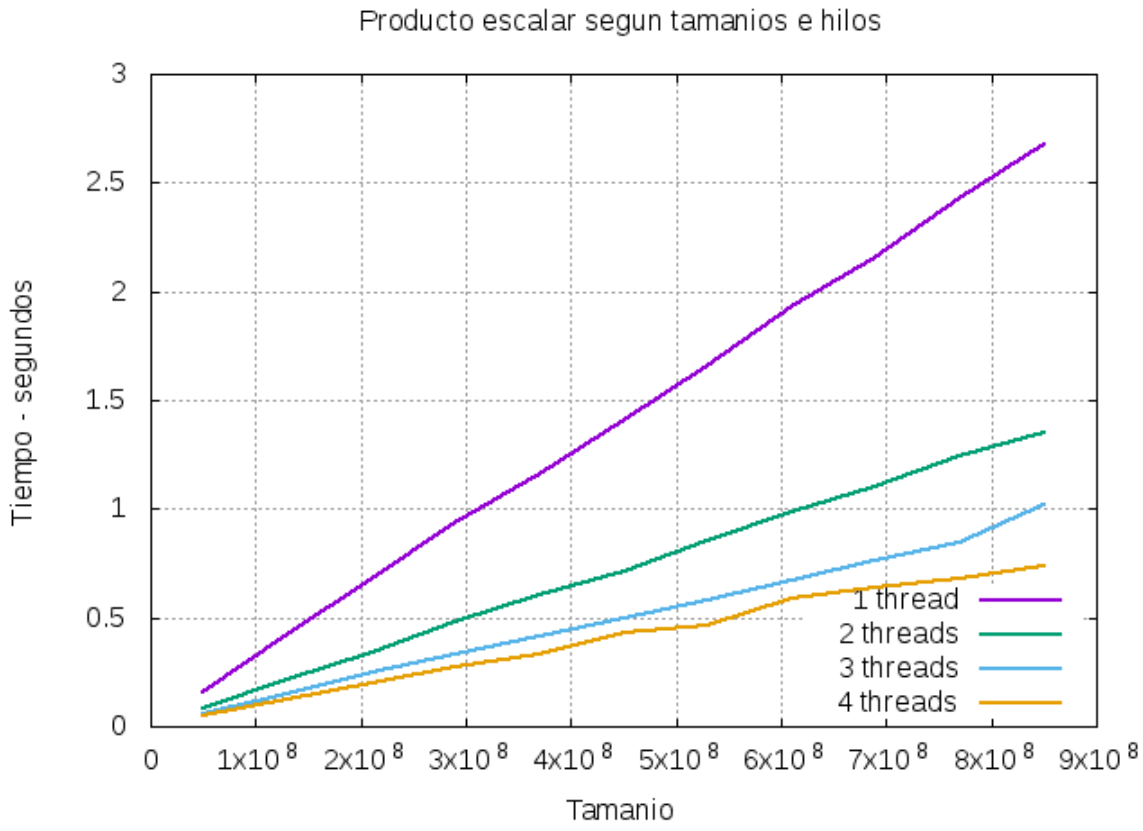
1 ¿En qué caso es correcto el resultado?

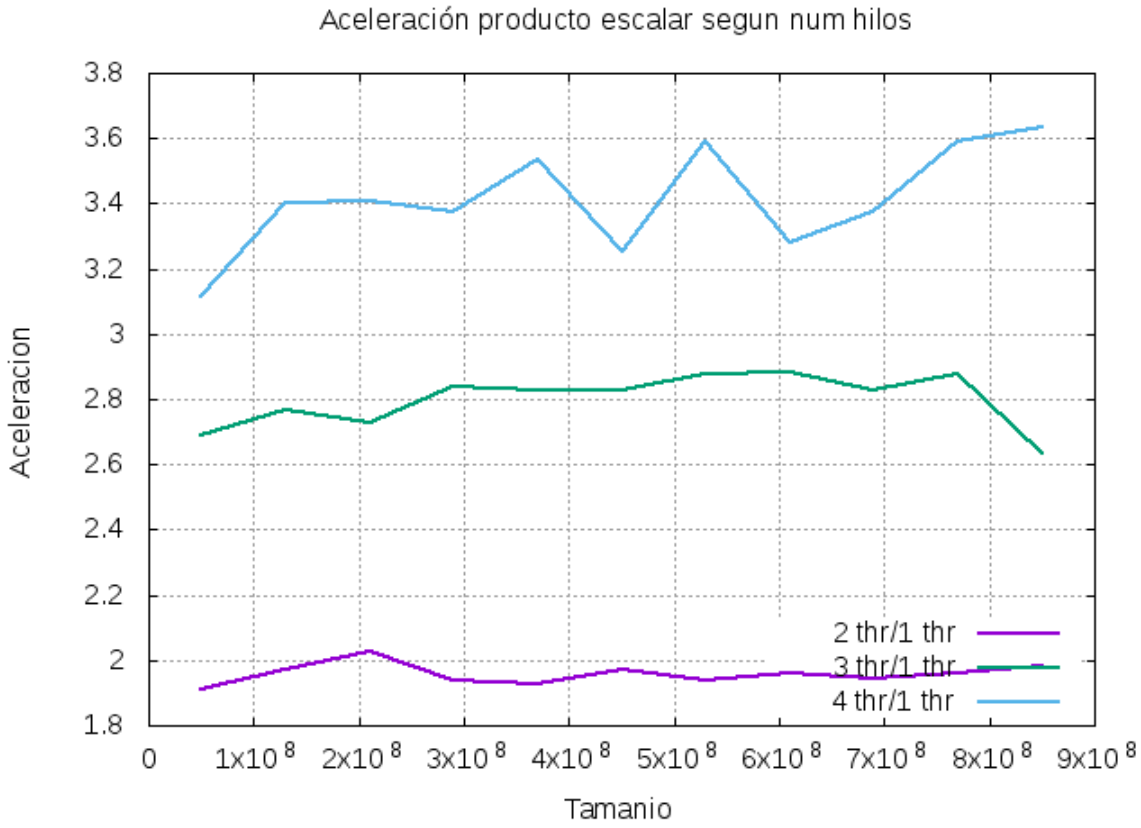
De las ejecuciones paralelas, el resultado sólo es correcto en la ejecución `par2`.

2 ¿A qué se debe esta diferencia?

Como en par1 la variable sum es pública, no se controla la concurrencia de operaciones sobre ella y pueden ocurrir distintos sucesos que den lugar a un error en el resultado. Por ejemplo: que el thread 1 coja el valor de sum, el thread 2 coge el mismo valor sum, ejecuta su operación y lo guarda, y al final el thread 1 ejecuta su operación con el valor antiguo de sum, guardando un valor erróneo en la variable. Sin embargo, con el uso de reduction, la segunda versión si controla el acceso a esta variable: cada hilo hace su propia suma y luego se suman todos los resultados.

Nota: *el enunciado indicaba que teníamos que usar vectores que tardaran em multiplicarse hasta 10 segundos. Las gráficas obtenidas no van a representar estos tiempos por lo siguiente: aunque la salida del programa nos daba tiempos de ejecución de apenas 2 segundos, añadiendo otro timeval para saber el tiempo total de ejecución, obteníamos ejecuciones de 20 segundos. Por tanto, aunque en la gráfica se reflejen tiempos bajos, los tiempos reales eran los que la práctica requería.*





3 En términos del tamaño de los vectores, ¿compensa siempre lanzar hilos para realizar el trabajo en paralelo, o hay casos en los que no?

Como puede deducirse de la primera gráfica, lanzar más hilos ha aumentado en todo caso la eficiencia del programa, y el único caso en que podemos encontrar alguna pega es para el lanzamiento de 4 hilos, que discutimos en la siguiente pregunta.

Para 2 hilos, el programa se ejecuta exactamente 2 veces más rápido (véase la gráfica de la aceleración), y para 3 hilos también trabajaba alrededor de 3 veces más rápido que el programa en serie.

4 Si compensara siempre, ¿en qué casos no compensa y por qué?

A pesar de que trabajar con 4 hilos sea más rápido que trabajar con 3, hemos encontrado una pequeña pega: esperaríamos que el rendimiento con fuera cuatro veces mayor que el del programa de producto escalar en serie, pero no es así. La aceleración medida ronda el 3,5, en lugar de la ideal, que sería 4.

Esto podría deberse a que el tiempo de inicialización y sincronización de hilos fuera demasiado alto como para conseguir el rendimiento óptimo del programa.

5 ¿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 representados en las gráficas, la respuesta es sí, aunque no siempre tanto como

sería deseable (caso de 4 hilos). Sin embargo, sabemos que si seguimos aumentando el número de hilos hasta superar el de los hilos virtuales que soporta el peocesador, no conseguiríamos un rendimiento mayor.

6 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.

Suponemos que para tamaños pequeños de vectores, la aceleración obtenida crecerá de manera muy rápida, pero que existirá un tamaño a partir del cual la aceleración se estabilice, y lo que refleja nuestra gráfica es la aceleración ya estabilizada.

3. Paralelizar la multiplicación de matrices

La salida de nuestro programa, que rellenaba las tablas del eneunciado y las mostraba por pantalla y sobre el cual hablaremos más en detalle un poco mas adelante, además de las gráficas, ha generado la siguiente salida:

```
Calculando tiempos

Tiempos de ejecución para tamaño 1000:

Versión 1 Hilo          2 Hilos         3 Hilos         4 Hilos

Serie:  5.151897        5.075940        5.062715        5.010849

Par 1:  7.044106        5.256404        6.583724        6.105344

Par 2:  6.953895        3.767271        4.614347        3.523721

Par 3:  6.225681        3.351287        3.261757        3.023674


Aceleracion (speedup) con respecto a la version serie para tamaño 1000

Version 1 Hilo          2 Hilos         3 Hilos         4 Hilos

Serie:  1.000000        1.014964        1.017615        1.028148

Par 1:  .7313769        .9801181        .7825201        .8438340

Par 2:  .7408649        1.367540        1.116495        1.462061

Par 3:  .8275234        1.537289        1.579485        1.703853
```

```
Calculando tiempos

Tiempos de ejecución para tamaño 6000:

Versión 1 Hilo          2 Hilos          3 Hilos          4 Hilos

Serie:  1078.551270     1068.962646      1068.968140      1068.563721

Par 1:  1420.567993     988.266724       1268.779907      1161.201416

Par 2:  1594.532349     798.315002       961.236328       751.089478

Par 3:  1305.288818     702.498169       685.212219       669.273743

Aceleracion (speedup) con respecto a la version serie para tamaño 6000

Version 1 Hilo          2 Hilos          3 Hilos          4 Hilos

Serie:  1.000000        1.008970         1.008964         1.009346

Par 1:  .7592394        1.091356         .8500696         .9288235

Par 2:  .6764060        1.351034         1.122045         1.435982

Par 3:  .8262931        1.535308         1.574039         1.611524
```

1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe?

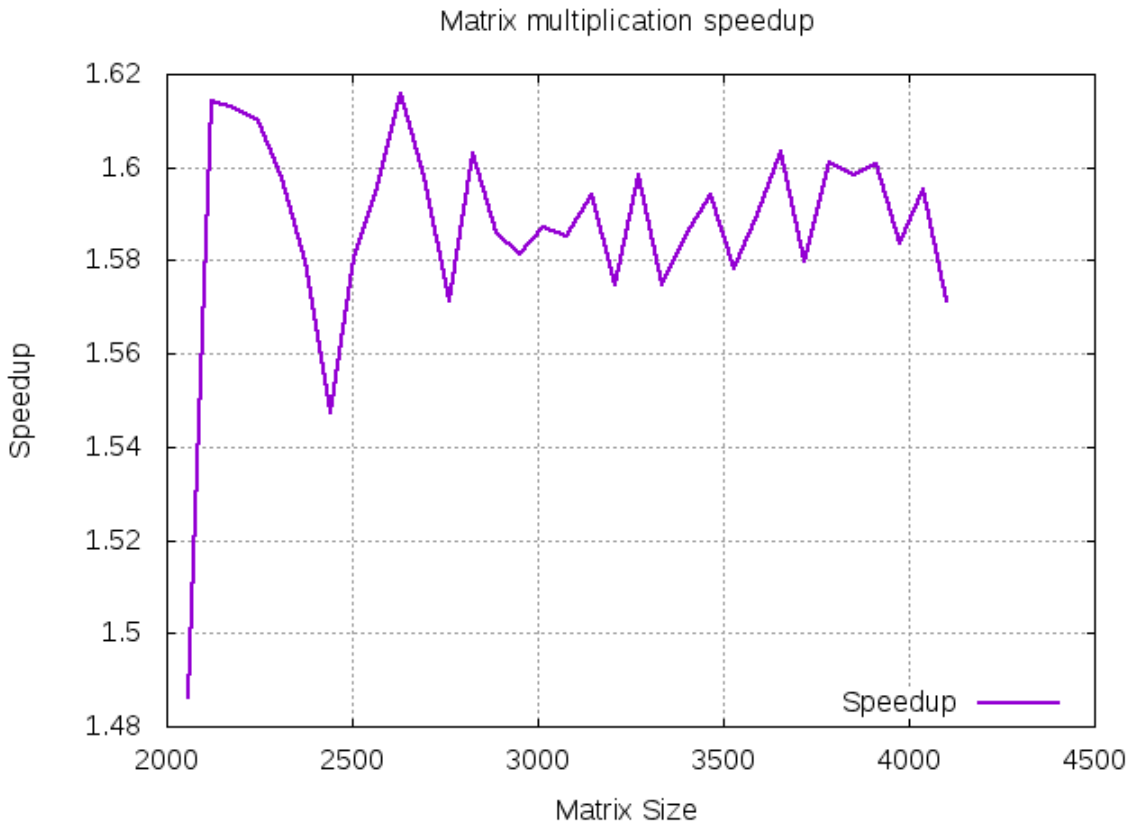
La versión que ha obtenido peor rendimiento ha sido la de paralelización del bucle con un único hilo. La razón de esto se halla en el hecho de que, por un lado, un hilo es el menor nivel de paralelización posible, por otro lado, el número de operaciones paralelizadas no es el suficiente para que salga rentable dicha paralelización en términos de eficiencia, y esto se ve potenciado por el hecho de que la paralelizacion se ejecuta en cada iteración del bucle superior. Esto se aprecia con claridad en las salidas de los programas que tenemos más arriba, para matrices de tamaño 1000 y 6000

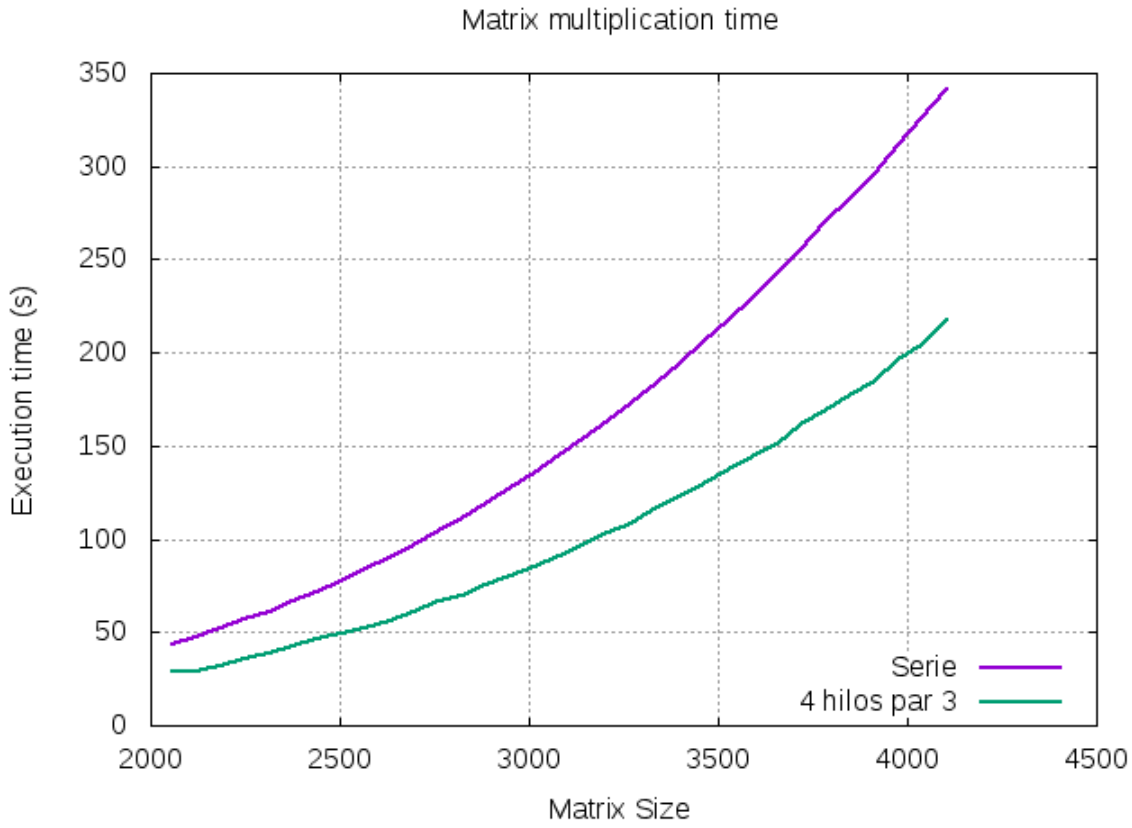
2 ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?

Tiene mejor rendimiento la version en la que se paraleliza el bucle 3, es decir, el bucle exterior, con 4 hilos. Esto tiene sentido, ya que a más hilos, mientras el número de éstos no supere el número de cores de la máquina, mayor es la paralelización, y mayor es, por tanto, el rendimiento. En cuanto a que el bucle que tiene más sentido paralelizar sea el exterior se debe a que esta es la manera de paralelizar más operaciones. Por ejemplo, el bucle interior ejecuta menos operaciones por cada iteración que cualquiera de los otros.

3 Gráficas y comentarios

Estas son las gráficas que ha generado nuestro programa. La primera es tiene el *speedup* en el eje Y y el tamaño de la matriz en el eje de las X. La segunda tiene el tiempo de ejecución en el eje Y y el tamaño de la matriz en el eje X.





Para generar la gráfica hemos utilizado un script que incluimos en esta entrega llamado *ej3.sh*, que primeramente rellenaba y mostraba las tablas del enunciado para matrices de tamaños *1000x1000* y *6000x6000*. Después generaba las gráficas para tamaños desde *2056* hasta *4014* aumentando de 64 en 64.

Se aprecia e la gráfica que el crecimiento de tiempo con el aumento del tamaño de la matriz es exponencial, tal y como abría de esperar (ya que el número de resultados a obtener es el numero de elementos de la matriz, que va aumentando de forma cuadrática sobre el aumento del tamaño de la matriz) También vemos como una paralelización adecuada aumenta el rendimiento, en este caso, lo aumenta en un 59% aproximadamente. En la gráfica de speedup, aunque aparentemente hay grandes irregularidades, basta con mirar la escala del eje Y para darse cuenta de que no es así realmente, casi todos los valores se mantienen en un intervalo muy pequeño, de tamaño 0,05 $(1,57 - 1,62)$.

4. Ejemplo de integración numérica Perdida de rendimiento por el efecto de falso compartir (False sharing) en OpenMP

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

La partición es de $n = 10^8$ rectángulos.

2 ¿Qué diferencias observa entre estas dos versiones?

En el caso de `pi_par1.c`, a cada hilo se le asocian ciertas h de la partición y un elemento de un array compartido. Para cada una de sus h, el hilo suma cierto valor a su correspondiente elemento del array.

En cambio, en `pi_par4.c`, cada hilo realiza su propia suma sobre una variable invisible a los demás hilos, y sólo cuando ha completado todos los cálculos que se le habían asignado escribe ese valor en el elemento del array que le corresponde.

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?

A pesar de que en ambos programas la salida indica `resultado` π`:  3.141593`, el rendimiento es muy distinto. En `./pi_par4` tenemos tiempo 0.121272 y en `./pi_par1` obtenemos 0.370529, que resulta tres veces más lento.

La razón es que, en `pi_par1.c`, por cada iteración del bucle los hilos actualizan diferentes posiciones del mismo array. Como cada procesador tiene su propia caché donde reside una copia del array, cada actualización de cualquier elemento del mismo debe propagarse hacia arriba, y luego llegar a las cachés distintos procesadores. En cambio, `pi_par4.c` sólo escribe en el array una vez por cada hilo (en lugar de una vez por cada hilo e iteración), por lo cual la actualización del mismo no supone un problema en el rendimiento del programa.

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?

Observamos que, por un lado, el resultado obtenido es el correcto de π en ambos casos. Por otro lado, la version `pi_par3.c` es alrededor de 0.8 segundos más rápida que la versiona `pi_par2.c`.

Esto es lo que cabría esperar, ya que la versión `pi_par2.c` está simplemente cogiendo *sum* como un puntero privado que apunta a la dirección original de *sum*, de esta manera, todos los threads editan la misma variable *sum*, pero siguen actualizando el mismo array, que sigue teniendo que acualizarse en todas las cachés, de ahí que el rendimiento sea similar al de `pi_par1.c`, esencialmente, caen en el mismo uso mejorable de la caché. Sin embargo, la versión `pi_par3.c` hace algo distinto: estudia el tamaño de la caché para llevar a cabo la técnica de "padding", consistente en mirar cuantos datos caben en la caché, cargarlos, y reutilizarlos todo lo posible, para cargar los siguientes despues. A pesar de ello, cada edición de la variable *sum* sigue provocando que en las cachés de los otros cores se tenga que recargar *sum* en cada uno de ellos por cada edición, por ello, aunque el rendimiento mejora, no llega a estar al nivel de `pi_par4.c`

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?

Observamos que para *padsz*=1 tenemos tiempos de ejecución similares a los de `pi_par1.c`, y conforme

vamos aumentando el valor, estos tiempos van disminuyendo hasta llegar a 8 (valor natural de la máquina, que es el que obtenía el programa antes de editarlo al ejecutarse), momento a partir del cual se estabilizan. Esto es ciertamente contraintuitivo y posiblemente debido a las características de la máquina en la que se ha ejecutado. Lo que ocurriría normalmente, sería una abrupta subida a partir del valor 8, debido a un desbordamiento de la caché que provocaría un aumento exponencial de los fallos de caché.

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?

Podemos observar que la version 5 no solo es mucho más lenta que la version 4, sino que además el resultado es erróneo y no determinista, es decir, que con cada ejecución es uno distinto. La lentitud se debe a que cuando un thread llega a la instrucción `critical` detiene la ejecución de todos los demás threads, a diferencia de un semáforo, que simplemente impide que varias ejecuciones entren en la zona crítica a la vez, esto provoca que cada thread pase muchísimo tiempo parado y tome mucho mas tiempo la ejecución total.

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?

Ambos resultados son correctos, aunque la versión 7 es notablemente más rápida que la versión 6, del orden de un 25%, debido al hecho de que la versión 6 sigue cometiendo *false sahring* con la variable *sum*. La versión 7 soluciona este problema con el uso de *reduction*. Cada hilo tiene su propia variable sum que utiliza por separado, y al final de la ejecución de los hilos, todas las variables *sum* de éstos se suman para formar una sola.