UNIVERSIDAD AUTÓNOMA DE MADRID

ARQUITECTURA DE COMPUTADORES

(2019-2020)

PRÁCTICA 4

Alba Ramos Pedroviejo
Ana Roa González

Grupo 1363

Madrid, 17 de diciembre de 2019

Ejercicio 0

El ordenador del laboratorio tiene 4 procesadores, cada uno con 4 cores físicos y otros 4 virtuales (por lo tanto, no permite hyperthreading). La frecuencia de cada procesador es de aproximadamente 1200MHz.

Los ordenadores del cluster tienen 16 procesadores, cada uno con 4 cores físicos y otros 8 virtuales (por lo tanto, sí permite hyperthreading). La frecuencia de cada procesador es de 2268MHz.

Ejercicio 1

Pregunta 1

Sí, y tiene sentido porque con los hilos pretendemos paralelizar tareas dentro del mismo procesador/core (dentro de las limitaciones del planificador). Ahora bien, depende también de la tarea que queramos realizar: si la tarea consiste en varias subtareas, tiene sentido lanzar hilos dentro de esa tarea, sino no. Lo ideal sería que cada tarea se ejecutara en un core, y dentro de cada core, las tareas se dividieran en varios hilos.

Pregunta 2

Podemos lanzar un hilo por cada core y tener cada tarea en un core diferente. Esto depende del número de cores de cada equipo. En el caso de los del cluster, como hay 8 cores virtuales podemos lanzar hasta 8 hilos, pero en los del laboratorio no podemos, porque no soporta multihilo.

Pregunta 3

Ante una variable privada, cada hilo la verá como que no está inicializada y la verá en el espacio de direcciones asociado, esto es, con una dirección de memoria distinta a la que pueda tener esa variable en el programa principal (antes de lanzar la región paralela) y distinta a la del resto de hilos.

Pregunta 4

Al no estar inicializada, toma el valor 0 por defecto. Cada hilo actualiza su propia copia de la variable partiendo del valor 0.

Pregunta 5

Como cada hilo ha actualizado su propia copia de la variable privada (en su espacio de direcciones), al acabar el programa este no se entera de los cambios, al tratarse de un espacio de direcciones diferente. El programa verá el valor de la variable que le dio inicialmente.

Pregunta 6

No ocurre lo mismo, porque al ser públicas se consideran un recurso compartido con la misma dirección de memoria para todos los hilos. De esta forma, cuando uno de los hilos actualiza su valor, se guarda en la dirección de memoria que ven todos, por tanto todos se dan cuenta del cambio y trabajan sobre él. Al finalizar la región paralela, la variable tiene el valor actualizado por todos los hilos.
Sin embargo, cuando lanzamos todos los hilos a la vez, la variable se inicializa con el mismo valor en todos, pero a medida que van escribiendo, los valores actualizados son visibles para todos los hilos. Por eso podemos observar que, al imprimir los valores de la variable en el programa en cada hilo, vemos que el valor inicial es el mismo (2), pero la actualización es sobre los cambios que ya haya.

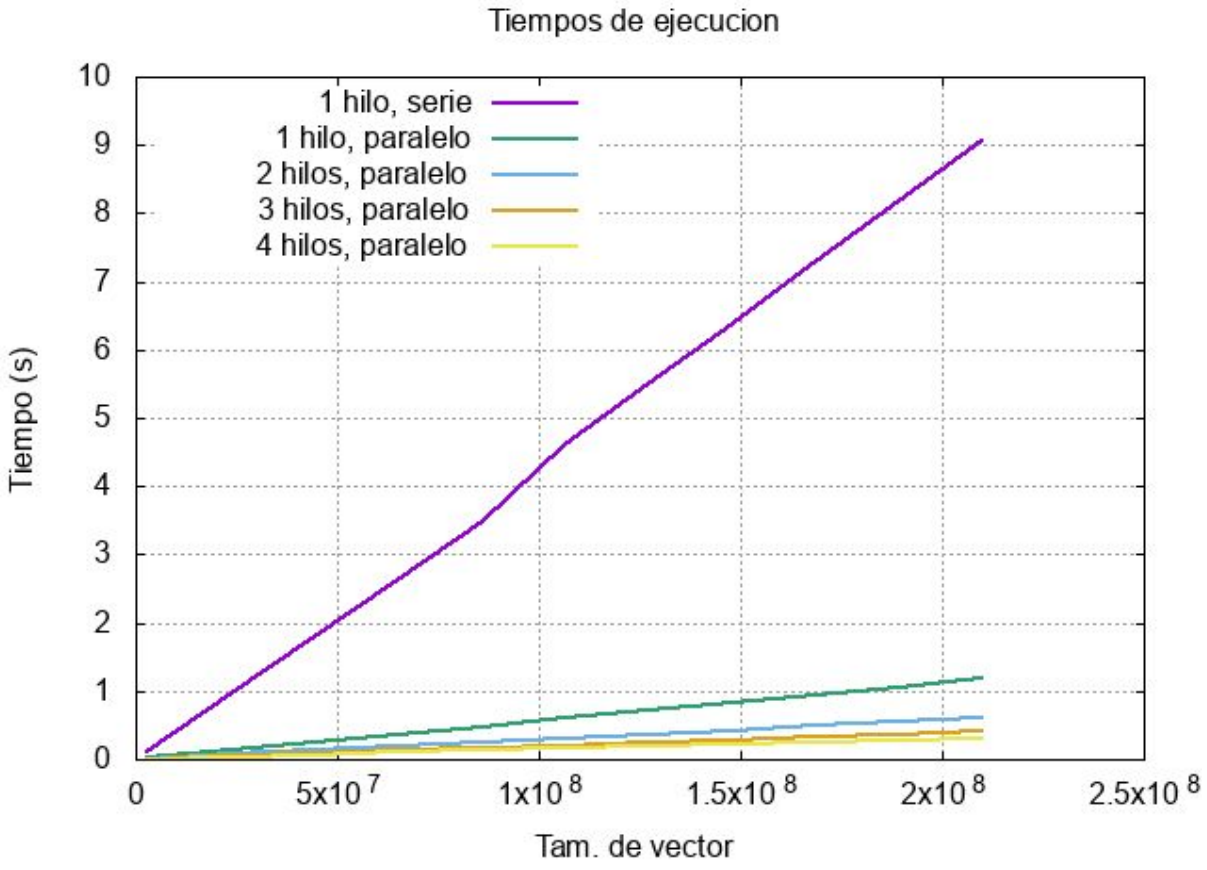
Ejercicio 2

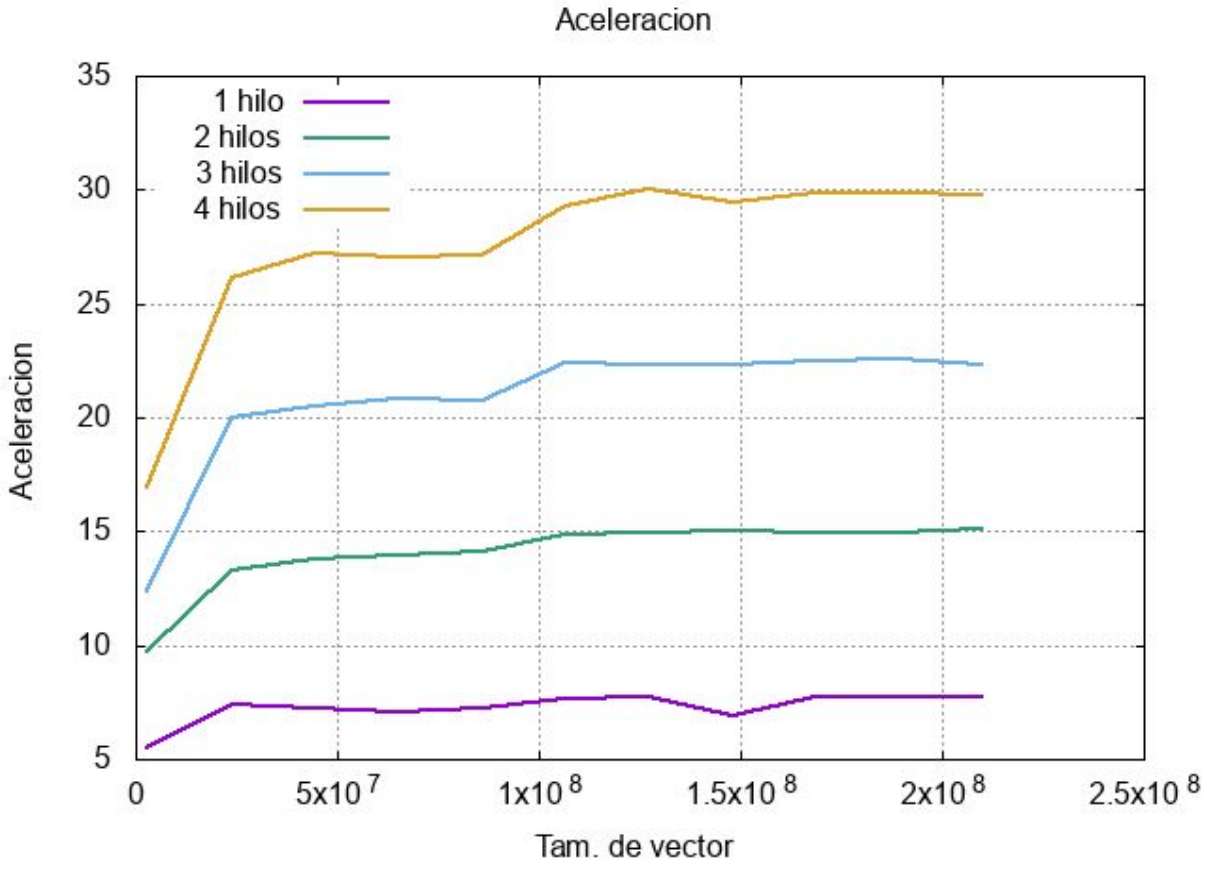
Pregunta 1

El caso correcto es el de los programas serie y paralelo versión 2. Las diferencias que se observan en el resultado son mínimas y es debido a la forma que tienen los hilos de realizar las aproximaciones cuando hay muchos decimales.

Pregunta 2

La diferencia es que en la versión 1 del programa paralelo no se está utilizando la función reduction para la suma de cada resultado y, por lo tanto, está habiendo una condición de carrera. Por eso, la variable sum no está siendo global y cada hilo ve su propia copia y trabaja sobre ella. El último hilo que acabe es el que da su valor a la variable global. En el segundo caso, con la función reduction, sí se está controlando la condición de carrera, accediendo los hilos a la variable global correctamente.


Tiempos de ejecucion
1 hilo, serie
1 hilo, paralelo
2 hilos, paralelo
3 hilos, paralelo
4 hilos, paralelo
Tiempo (s)
Tam. de vector
0
5x10 7
1x10 8
1.5x10 8
2x10 8
2.5x10 8
0
1
2
3
4
5
6
7
8
9
10


Aceleracion
1 hilo
2 hilos
3 hilos
4 hilos
Aceleracion
Tam. de vector
0
5x10 7
1x10 8
1.5x10 8
2x10 8
2.5x10 8
5
10
15
20
25
30
35

Preguntas 3 y 4

No compensa siempre, ya que para tamaños a partir de aproximadamente $1.7*10^8$ vemos que la aceleración obtenida tiende a ser una recta para todos los hilos. Con esto observamos que no compensa, a partir de estos tamaños, lanzar hilos respecto a ejecutar en serie la tarea.

Preguntas 5 y 6

No compensa siempre, porque hay tareas que si no se paralelizan no se van a beneficiar del lanzamiento de hilos, por ejemplo tareas de asignación o printf. Además, lanzar hilos es una tarea costosa, y manejar la paralelización también. En vista de esto, si el tiempo de ejecución de una tarea va a ser menor que el tiempo que se tarde en preparar el sistema con los hilos y en su ejecución paralela, entonces no compensa hacerlo.

Pregunta 7

Observamos un punto de inflexión en el tamaño (aproximadamente) de $2*10^7$, en el que el comportamiento pasa de crecer exponencialmente a estabilizarse en una recta.

Ejercicio 3

Versión/nº hilos	**1**	**2**	**3**	**4**
Serie	32.5589	24.5359	27.659	35.7091
Paralelo bucle1	35.205	17.9028	17.481	20.32075
Paralelo bucle2	38.2897	12.9973	13.1709	14.48
Paralelo bucle3	37.1905	14.8532	12.2480	**11.6710**

Tiempos de ejecución(s)

Versión/nº hilos	1	2	3	4
Serie	1	1	1	1
Paralelo bucle1	0.82125	1.38125	1.545	1.7337
Paralelo bucle2	0.84875	1.9525	2.7662	2.6737
Paralelo bucle3	0.845	1.64875	2.37	**2.81**

Aceleración(s)

Pregunta 1

El peor rendimiento lo obtiene la versión del bucle paralelo 1 (más interno) en cada caso. Esto se debe a que únicamente paraleliza el trabajo del bucle más interno, mientras que los externos deben realizar todo secuencialmente. Además, cada vez que se ejecute el bucle interno, debe volver a preparar la región paralela y los hilos, por lo que necesita más tiempo.

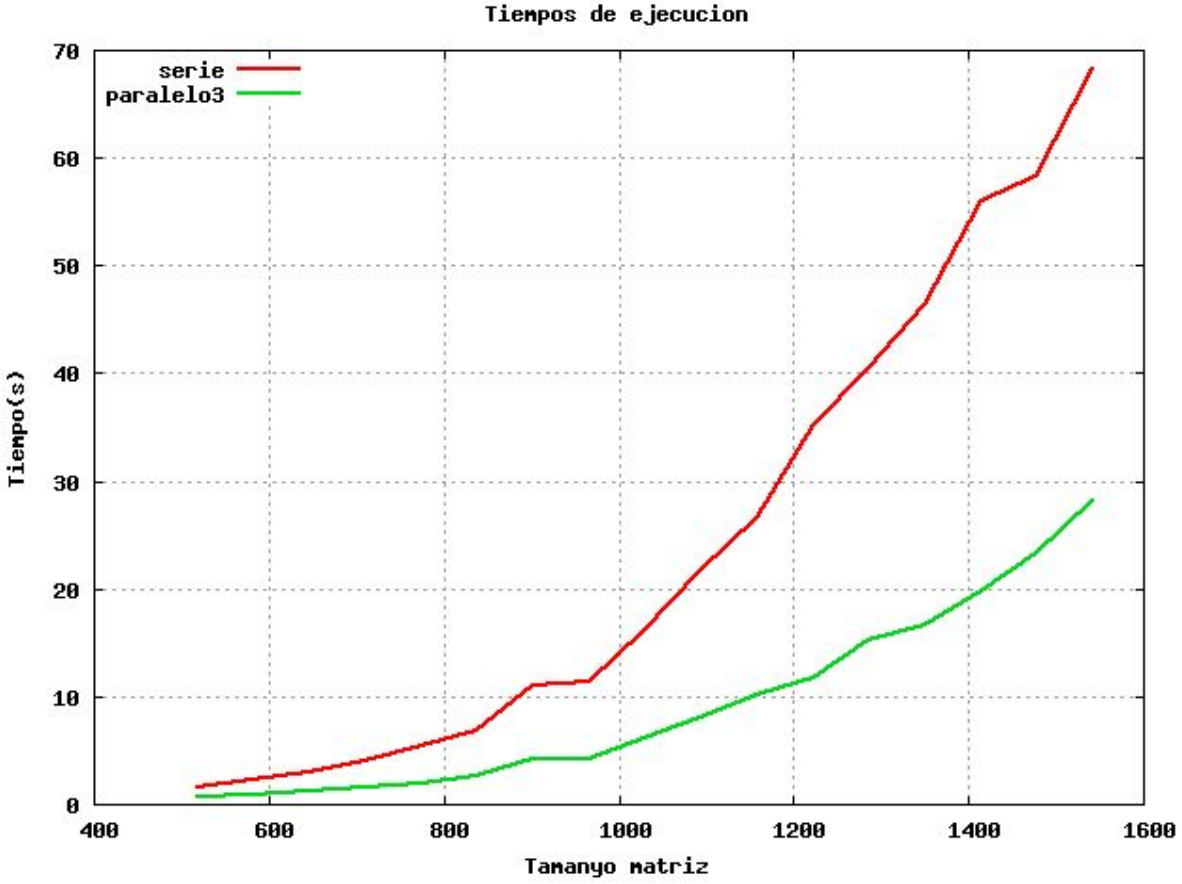
Pregunta 2

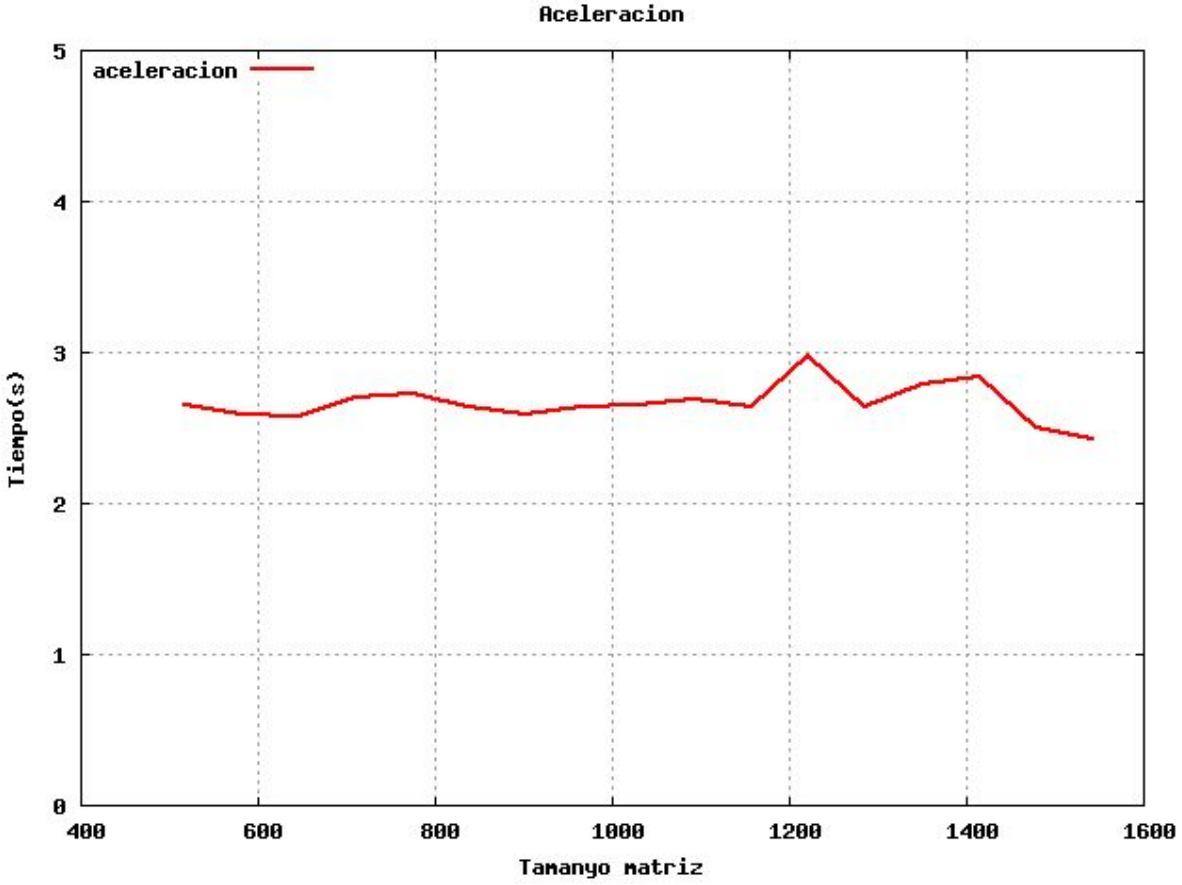
El mejor rendimiento lo obtiene la versión del bucle paralelo 3 (más externo) con 4 hilos. Esto se debe a que, cuantos más hilos haya realizando tareas en paralelo, esta versión del bucle paraleliza más tareas. Al estar en el bucle más externo, paraleliza todo el trabajo de los bucles internos. Además, realiza la preparación de los hilos una sola vez.

Pregunta 3

En las gráficas podemos observar que el comportamiento es el esperado para el programa. Observamos que el tiempo de ejecución ha sido notablemente más bajo para matrices de mayor tamaño en el caso del paralelo. Aquí es donde mejor se observa la potencia de los hilos y la paralelización, con diferencias de más de 30 segundos para el mayor tamaño.

En la gráfica del tamaño de la matriz observamos que ya se encuentra estabilizada en el valor 2’6. Esto se debe a que hemos comenzado a tomar valores desde un tamaño mayor de 500 y no en 0, en cuyo caso habríamos visto el crecimiento y después la estabilización, como ocurría en el ejercicio 2. Esto es por lo mismo que ocurría al multiplicar vectores en el ejercicio 2, ya que por muchos hilos que haya en paralelo, llegará un momento en el que al aumentar más el tamaño no haya compensación, como hemos explicado previamente.


Tiempos de ejecucion
serie
paralelo3
Tiempo(s)
Tamanyo matriz
0
10
20
30
40
50
60
70
400
600
800
1000
1200
1400
1600


Aceleracion
aceleracion
Tiempo(s)
Tamanyo matriz
0
1
2
3
4
5
400
600
800
1000
1200
1400
1600

Ejercicio 3

Pregunta 1

Se utilizan 100.000.000 de rectángulos.

Pregunta 2

El programa 1 va almacenando el resultado de la integral directamente en el propio array de sumas en su posición asociada a su PID. El programa 4, en cambio, primero almacena el resultado en una variable auxiliar y después, al finalizar el bucle, copia el valor de esa variable al array.

Pregunta 3

Se han obtenido los siguientes resultados:

PI_PAR1
Numero de cores del equipo: 12
Resultado pi: 3.141593
Tiempo 4.194066

PI_PAR4
Numero de cores del equipo: 12
Resultado pi: 3.141593
Tiempo 0.202437

Creemos que la diferencia en rendimiento entre ambos programas se debe a que el primero, para cada repetición, debe ir a la dirección de comienzo del array y desplazarse hasta su posición, leer su valor, realizar la suma y finalmente actualizarlo. En cambio, como el otro programa usa una variable auxiliar, sólo accede a la posición del array una única vez, al acabar el bucle, para actualizar su valor. Todo esto contribuye a que haya false sharing, es decir, cada vez que un hilo modifica su posición del array (bloque de caché), los demás son notificados de que el bloque ha sido modificado y van a tener que cargarlo de nuevo con la actualización (incluso si esta modificación no ha sido sobre la posición en la que ellos están trabajando).

Pregunta 4

Se han obtenido los siguientes resultados:

PI_PAR2

Numero de cores del equipo: 12
Resultado pi: 3.141593
Tiempo 4.191657

PI_PAR3
Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.204340

El resultado no ha variado, como era de esperar. La versión que utiliza las cachés es la que mejor rendimiento obtiene y más similar a pi_par4, mientras que la versión pi_par2 no ha mejorado, pues lo único que se ha hecho en ella es hacer que la suma obtenga el valor inicial de inicialización, mientras que la forma de almacenar los resultados es la misma.

La mejora en el programa pi_par3 se debe a que ahora el tamaño del bloque es mucho mayor debido a que multiplicamos por el padding obtenido. Con esto, se logra que el bloque no contenga el array entero, que era lo que probablemente ocurría antes. Ahora, cuando un hilo modifica su parte del array, a los demás que no tengan esa parte en el bloque no se les notifica del cambio y no deben actualizar su bloque. Aún así, puede ocurrir que en algunas ocasiones siga ocurriendo esto, pero menos que antes.

Pregunta 5

Observamos que, a mayor padding, mejor rendimiento obtenemos. Esto es debido a lo que hemos explicado en el apartado anterior: a mayor padding, más grande es el array, y menos influidos se ven los bloques asociados a cada hilo por los cambios en otro. Adjuntamos los resultados obtenidos:

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 1 elementos
Resultado pi: 3.141593
Tiempo 4.165310

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 2 elementos
Resultado pi: 3.141593
Tiempo 1.742980

Numero de cores del equipo: 12
Double size: 8 bytes

Cache line size: 64 bytes => padding: 4 elementos
Resultado pi: 3.141593
Tiempo 0.568567

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 6 elementos
Resultado pi: 3.141593
Tiempo 0.602916

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 7 elementos
Resultado pi: 3.141593
Tiempo 0.535398

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.208069

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 9 elementos
Resultado pi: 3.141593
Tiempo 0.208100

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 10 elementos
Resultado pi: 3.141593
Tiempo 0.204237

Numero de cores del equipo: 12
Double size: 8 bytes
Cache line size: 64 bytes => padding: 12 elementos
Resultado pi: 3.141593
Tiempo 0.204527

Ejercicio 5

Pregunta 1

Se observa la siguiente salida en ambos programas:

PI_PAR4
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.295635

PI_PAR5
Resultado pi: 1.017036
Tiempo 1.460116

Lo primero que observamos es que ahora los resultados son diferentes. Esto se debe a que pi no se está inicializando y se está utilizando directamente en el bucle que actualiza su valor. Aparte de esto, vemos que empeora el rendimiento al haber introducido la sección crítica. Esto ocurre porque al final esto se comporta secuencialmente, ya que un hilo no puede entrar a actualizar la sección crítica si ya hay otro dentro.

Pregunta 2

El resultado obtenido es el siguiente:

PI_PAR6
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.881444

PI_PAR7
Resultado pi: 3.141593
Tiempo 0.298649

Se observa una mejora porque en el pi_par6 primero se debe rellenar el array de sumas de cada proceso y, cuando acabe, recorrerlo entero para ir sumando todos los valores y calcular finalmente el valor de pi. En cambio, en el programa pi_par7, se hacen ambos pasos a la vez gracias a la cláusula reduction. En el mismo bucle, el proceso calcula su suma y este valor se añade a la suma de pi, y cuando el bucle acabe, OpenMP gestionará la puesta en común de todos los resultados.