Práctica 4: Explorar el potencial de las arquitecturas modernas

Franciasco de Vicente Lana (francisco.vicentel@estudiante.uam.es)
Ricardo Riol González (ricardo.riol@estudiante.uam.es)

Arquitectura de Ordenadores
Universidad Autónoma

12 de diciembre de 2017

Ejercicio 0

Para este ejercicio introductorio simplemente se nos pide obtener información relativa a la arquitectura de la máquina sobre las que se ejecutarán el resto de ejercicios de la práctica. Para ello, ejecutamos el comando **cat /proc/cpuinfo** obtiendo los siguiente resultados:
El número de cores físicos de la máquina son 4 y lo extraemos de la línea
cpu cores : 4
El número de cores virtuales son 4 y lo vemos en la línea
siblings : 4
Como el número de cores físicos y virtuales son el mismo, podemos concluir que o bien hyperthreading está desactivado o bien esta funcionalidad no está implemetanda en la arquitectura de la máquina empleada. Si buscamos las características del procesador del equipo usado en la escuela (**model name : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz**) en la página oficial de intel, nos damos cuenta de que dicho procesador no soporta la funcionalidad de hyperthreading.
Por último en este ejercicio se nos pide la frecuencia de los procesadores. Cada uno de los cuatro procesadores de nuestra máquina tiene una frecuencia de:
cpu MHz : 800.000

Ejercicio 1

Este ejercicio se trata de una breve introducción a la librería omp para familiarizarnos con este método de paralelización de código; el cual hace de forma automática la separación en distintos threads de una sección de código bien preparada para ello.

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

Sí, podemos lanzar tantos threads como queramos. Sin embargo, no tiene sentido hacerlo. En el ejercicio 0 observamos las máquinas de las escuela tienen 4 cores físicos y virtuales (no hyperthreading), por lo que el máximo rendiemiento del procesador lo alcanzaremos con 4 hilos, que correrán por cada uno de los cores de la máquina. Por tanto, lanzar más hilos que cores virtuales tiene nuestra máquina no tiene sentido, ya que en el procesador solo podrán ejecutrase simultáneamente tantos hilos como cores viruales tanga la máquina.

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

En el caso de los ordenadores de la escuela que utilizamos para esta práctica tiene sentido utilizar un máximo de 4 threads ya que, como vimos en el ejercicio 0, tiene un total de cuatro núcleos (tanto físicos como virtuales). En caso de usar menos en general obtendríamos tiempos de ejecución mayores pues no se aprovecha el sistema en su totalidad y si empleamos más hilos no encontraremos una mejora significativa del rendimiento ya que estos competirán entre sí siendo contraproducentes.
En cuanto a nuestro equipo personal, al contar con un procesador de dos núcleos e hyperthreading (intel core i5-4300U), será más eficiente usar 4 hilos; aunque cabe mencionar que para programas de alto rendimiento para realizar cómputos el hyperthreading no es una ventaja con lo que convendría usar 2 hilos.

1.3 ¿Cómo se comporta OpenMP cuando declaramos una variable privada?
Al declarar una variable con la sentencia *#pragma omp parallel private(variable)* omp automáticamente asigna una copia de ella en cada uno de los hilos en los que se paraleliza un programa. Cabe destacar que esta varible no se inicializa al valor anterior.

1.4 ¿Qué ocurre con el valor de una variable privada al comenzar a ejecutarse la región paralela?
Al comenzar una región paralela el procesador asigna una zona de memoria para las variables privadas, por lo que tomarán valores completamente arbitrarios. En la mayoría de los casos estas toman el valor 0.

1.5 ¿Qué ocurre con el valor de una variable privada al finalizar la región paralela?
Cuando termina esta región la variable privada se pierde ya que solo es utilizada dentro de cada hilo y no en el flujo principal

1.6 ¿Ocurre lo mismo con las variables públicas?
No, las variables públicas mantienen su valor al terminar la región paralela.

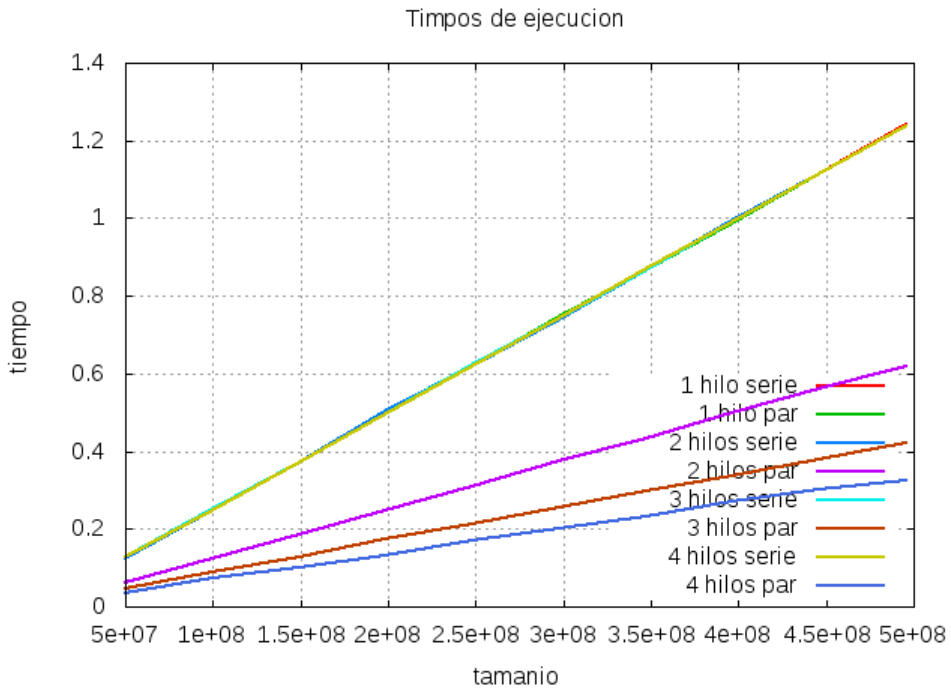
Ejercicio 2

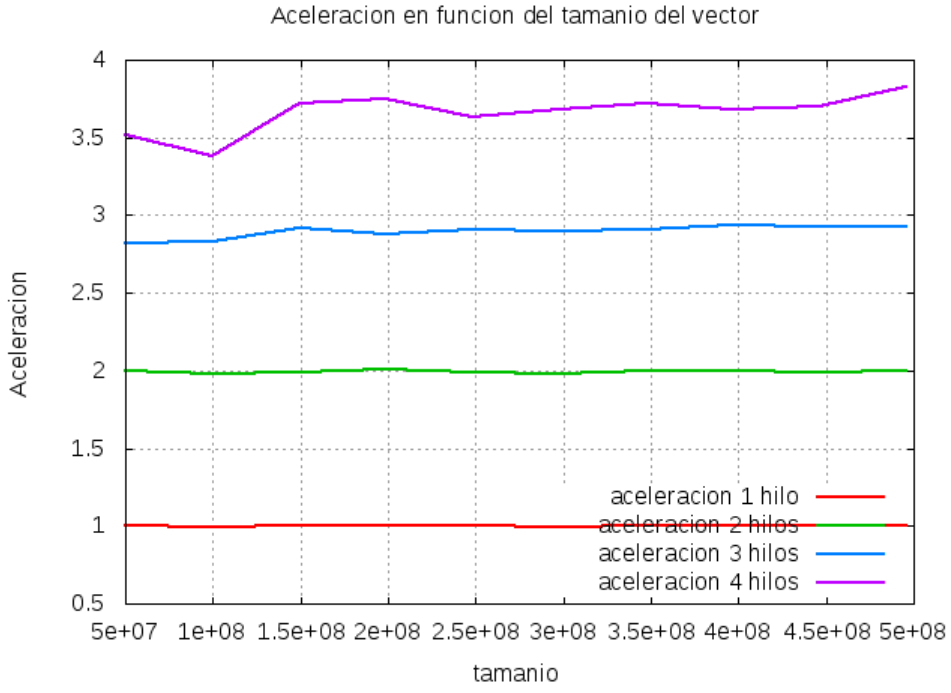
En este ejercicio se nos pide observar como OpenMP, con secciones paralelas, mejora el rendimiento del producto escalar de vectores.

2.1 ¿En qué caso es correcto el resultado?
Es obvio que el resultado correcto es el que nos da la ejecución de pescalar_serie.c, ya que implementa el algoritmo del producto escalar (sin paralelizar) de dos vectores de tamaño n. Si ejecutamos pescalar_par1.c y pescalar_par2.c observamos que en el caso del primero el resultado no coincide con el que cabría esperar, mientras que con pescalar_par2.c sí que lo hace.

2.2 ¿A qué se debe esta diferencia?
La diferencia se debe a que pescalar_par1.c utiliza una variable (sum), compartida para todos los hilos, donde suma los resultados de las multiplicaciones coordenada a coordenada. Al ser una variable compartida por todos los hilos que se ejecutan de forma simultánea, si un hilo toma el valor de esta variable para sumarle una cantidad y otro hilo en ese instante actualiza ($suma = suma + a \quad b$) el valor de la variable compartida, cuando el hilo anterior vaya a actualizar el valor de esta variable pisará el trabajo del otro hilo, obteniéndose así un resultado erróneo.

Una vez realizado el estudio del rendiemiento de las distintas versiones del producto escalar de vectores en función del tamaño, hemos obtenido los siguientes resultados:





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

No, para vectores de dimensión reducida no sale rentable paralelizar el cómputo, ya que las ventajas

que presenta no compensan el gasto de tiempo en la creación de los hilos y los cambios de contexto que suponen.

2.4 Si compensara siempre, ¿en qué casos no compensa y por qué?
No compensa realizar trabajo en paralelo cuando el tamaño de los vectores es pequeño, ya que la definición de la región paralela consume casi mas tiempo que realizar el producto escalar normal.

2.5 ¿Se mejora siempre el rendimiento al aumentar el número de hilos a trabajar?
Sí, como se puede osbservar en la primera gráfica. (Considerando hasta un máximo de 4 hilos).

2.6 Si no fuera así, ¿a qué debe este efecto?
El tiempo de ejecución se mentiene más o menos constante a partir de 4 hilos debido a que ya se están aprovechando los recursos del sistema al máximo y generar más hilos implica que compitan entre ellos.

2.7 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.
Como se observa en la gráfica los valores de la aceleración se mantienen más o menos constantes para tamaños entre 5exp(7) y 5exp(8). La gráfica nos da la intuición de que esos valores se mantendrán para tamaños de vector grandes. Sin embargo, para vectores pequeños estos datos pueden verse perturbados. En estas circunstancias, la definición de la zona paralela hace que el programa tarde incluso más que realizar el producto escalar en serie. Por esta razón, la aceleración de los programas que ultilizan OpenMP con vectores pequeños puede disminuir a valores entre 0 y 1 (serán como mucho igual de buenos que la versión serie del producto escalar).

Ejercicio 3

Para el tercer ejercicio se nos pide analizar el efecto que tiene la parelización de código sobre el programa de multiplicación de matrices desarrollado en la práctica anterior. En este caso se nos plantea la duda de qué bucle conviene paralelizar ya que hay tres anidados. Probamos cada uno de ellos, incluida la versión en serie, obteniendo los resultados mostrados en las siguientes tablas para matrices de dimensión 1000x1000 (representando la primera los tiempos de ejecución y la segunda la aceleración):

Hilos	1	2	3	4
Serie	3.974814	3.772763	3.955959	3.747654
Par 1	5.341551	3.519411	2.715940	2.486014
Par 2	4.705071	2.333059	1.685491	1.210445
Par 3	4.413628	2.341295	1.603538	1.201865

Hilos	1	2	3	4
Serie	1	1	1	1
Par 1	0.7441310585633273	1.0719870455596121	1.4565708373528133	1.5074951307595210
Par 2	0.8447936279813843	1.6170885519826116	2.3470662258060114	3.0960960638442886
Par 3	0.9005774841015146	1.6114001012260308	2.4670191788407883	3.1181987993659853

A continuación se muestra los mismos resultados que antes pero para un tamaño de matriz superior (2000):

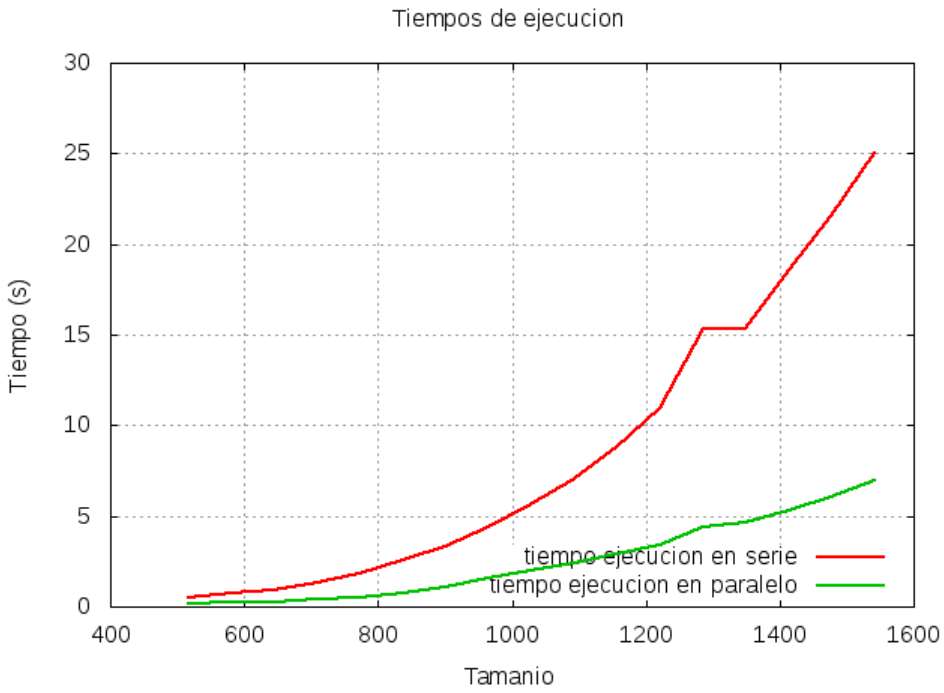
Hilos	1	2	3	4
Serie	34.990581	34.883165	35.239391	34.799495
Par 1	44.266136	24.329557	19.189060	17.202798
Par 2	41.726263	21.352960	14.914455	11.568614
Par 3	40.355197	19.966429	13.354696	10.309489

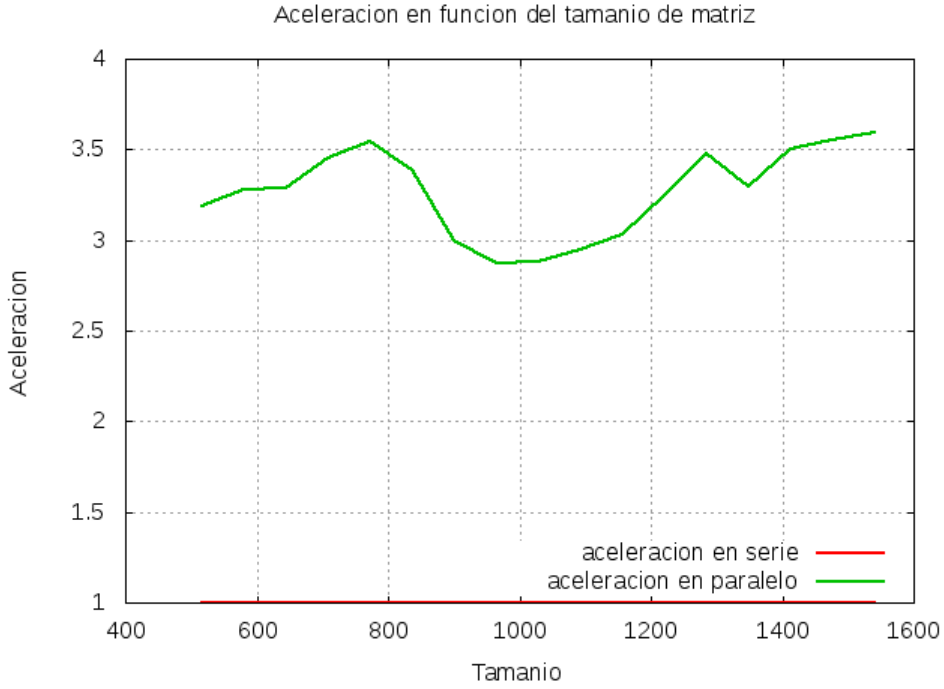
Hilos	1	2	3	4
Serie	1	1	1	1
Par 1	0.7904593479765209	1.4337772364700269	1.8364313311855817	2.0228973798332108
Par 2	0.8385745208000055	1.6336454056018463	2.3627675969386745	3.0080954382262213
Par 3	0.8670650523648788	1.7470908293115408	2.6387265573098781	3.3754820437753995

3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe?
Observando los tiempos obtenidos se puede ver que para matrices de un tamaño suficientemente grande es óptimo paralelizar el bucle más externo, la versión 3 según la nomenclatura empleada. Intuitivamente llegamos a la conslusión de que este efecto se debe a que paralelizando el bucle externo, se generan una cantidad significativamente inferior de hilos en comparación con los otros bucles por lo que se pierde menos tiempo tanto en el cambio de contexto entre hilos en el procesador como al crear y asignar memoria a cada uno de ellos.

3.2 ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?
Por otra parte, el peor resultado corresponde con la versión que paraleliza el bucle interno (versión 1). Su justificación es la misma que en el apartado anterior. Al tratarse del bucle más interno se va a generar una cantidad excesiva de hilos, lo cual resulta menos eficiente ya que cada hilo realizará tareas muy cortas y concretas por lo que los tiempos de generación y cambios entre hilos suponen demasiado tiempo extra a la ejecución. Es mejor tener menos hilos y que sean más independientes para un mejor aprovechamiento del multiprocesador.

Una vez realizado este primer análisis nos damos cuanta de que la mejor versión paralela es sin duda la 3 (bucle externo) haciendo uso de 4 threads. Con esto claro, comparamos su rendimiento con la versión en serie en las siguientes gráficas siendo la primera la que muestra la evolución de los tiempos de ejecución y la segunda la correspondiente a la aceleracion; ambas para matrices de dimensión entre 517 y 1531 con un salto de 64 unidades. Con el fin de obtener unas gráficas más suaves realizamos los cáculos repetidas ocasiones y de forma alternada para luego hallar la media de los resultados, que será considerablemente más fiable. Dado que la manipulación de matrices de dimensiones tan elevadas resulta costosa, en esta ocasión nos conformamos con 5 repeticiones.





3.3 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.

En la gráfica inmediatamente superior podemos ver que la aceleración permanece relativamente estable al aumentar el tamaño de la matriz desde 500 hasta 1500. Concretamente, presenta un valor comprendido entre 3 y 3.5, por lo que la aceleración que supone la paralelización es considerable.

Ejercicio 4

El ejercicio propuesto en esta ocasión ya no pide implementar código sino que, a partir de un programa dado que calcula el número pi mediate integración numérica, debemos analizar el efecto que tiene la paralelización de código y en concreto el *false sharing*.

4.1 ¿Cuántos rectángulos se utilizan en la versión del programa que se da para realizar la integración numérica?
Tras haber observado con detenimiento el método empleado para realizar la estimación de pi, vemos que se utilizan 100000000 rectángulos.

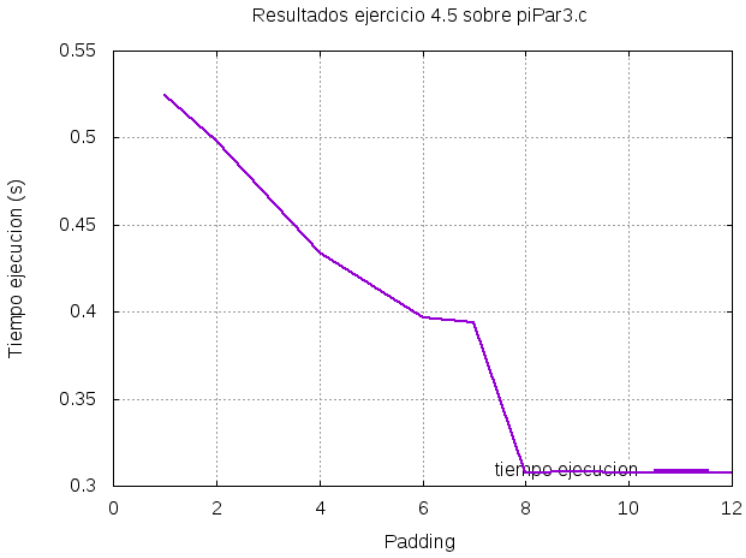
4.2: ¿Qué diferencias observa entre estas dos versiones?
La diferencia entre ambos reside en como se tratan las variables en el bucle a paralelizar. En la primera versión, debido a que se define la variable suma como compartida por defecto y se actualiza en cada iteración, se produce el efecto conocido como *false sharing*; este consiste en que cada vez que se modifica el valor de dicha variable en cada hilo, obliga a que el resto de ellos lo tengan que actualizar también, lo cual conlleva una disminución importatne en el rendimiento. Por otro lado, la versión número cuatro está mejor implementada de forma que solo actualiza el valor de esta variable compartida al final del cómputo mediante el uso de una variable privada y local para cada hilo evitando así el indeseado *false sharing*.

4.3: 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?
En efecto, como ya hemos justificado en la cuestión anterior, nos encontramos con una diferencia de rendimiento notoria. Se obtiene una tiempo de ejecución reducido casi a la mitad mediante la eliminación del *false sharing* que supone la versión pi_par4.c (0.517359 para la primera versión frente a 0.278871 segundos de la mejorada).

4.4: Ejecute las versiones paralelas 2 y 3 del programa. ¿Qué ocurre con el resultado y el rendimiento obtenido? ¿Ha ocurrido lo que se esperaba?
Estas versiones intentan mejorar la primera con dos enfoques distintos. En el primer caso simplemente se define la varible *sum* como privada para que en la zona paralela no sea compartida con otros hilos. Para la otra modificación se realiza un cambio en el padding de la memoria caché para acelerar el cómputo a realizar. Podemos ver como ambas versiones mejoran la original (pi_par2 apenas supone una mejora en el tiempo de ejecución con 0.500350 segundos, mientras que pi_par3 se acerca al reducido tiempo obtenido en la cuarta versión con 0.316755 segundos).

4.5: 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?

Para esta cuestión hemos realizado un script para facilitar el análisis sobre el efecto que tiene ir modificando el padding entre los valores indicados. La siguiente gráfica muestra estos resultados.



Como se puede observar a la perfección, para valores menores de 8 el tiempo de ejecución disminuye constantemente a medida que aumenta el número de elementos del padding. A partir de dicho valor los tiempos se estabilizan en torno a los 0.31 segundos como en el caso del código original de esta versión, por lo que deducimos que con este enfoque no podemos aspirar a una mejora mayor.

Ejercicio 5

En este ejercicio entra en juego la directiva *critical* de OpenMP, cuya funcionalidad es la de un mutex; es decir hacer que una región crítica (zona de código compartida entre n hilos) solo pueda ser ejecutanda por un único hilo al mismo timepo. El objetivo fundamental de esta directiva es la protección de variables de tal forma que los hilos no se pisen valores y puedan llegar a modificar el resultado final. Cabe mencionar que hemos llevado a cabo los ejercicios 4 y 5 en máquinas distintas, por lo que los tiempos de ejecución varían de uno a otro.

5.1: 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?

Ejecutando repetidas veces cada versión vemos inmediatamente que la versión 4 sigue siendo la más eficiente. Tan solo requiere de unos 0.34 segundos de media para realizar el cómputo; mientras que la quinta versión resulta 5 veces más lento, obteniendo tiempos superiores al segundo y media que además son mucho más variables que en el primer caso. Esto nos lleva a pensar que la definición de secciones críticas en un código a paralelizar mediante OMP no es una práctica eficiente. El razonamiento a este suceso es que las zonas críticas ralentizan notablemente el programa, ya que la ejecución de cada uno de los hilos en los que se divide debe detenerse en dicha región si cualquiera

de los otros la está ejecutando. Cuantos más hilos se generen y mayor sea la zona crítica, mayor será el impacto negativo del uso de la directiva *critical*.

5.2: : 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?

Por último, nos disponemos a analizar la sexta y sépitma versión para la estimación de pi. Al igual que en el resto de ocasiones, primero procedemos a comparar tiempos de ejecución para darse cuenta a simple vista de la implementación más eficiente. En este caso la sexta versión presenta tiempos en torno a medio segundo, en contraste con la número siete que iguala los mejores tiempos obtenidos hasta ahora (proporcionados por la versión 4) de 0,34 segundos. Estas modificaciones se basan en paralelizar el bucle interno por medio de los comandos de omp específicos para paralelizar bucles for. La última versión resulta más veloz debido a que aprovecha la directiva *reduction*, que en este caso es muy útil ya que evita tener que sumar los valores parciales obtenidos a posteriori. Esta diferencia entre ambos programas supone un incremento importante en la velocidad de ejecución, además de que simplifica el código notablemente.