MEMORIA

Práctica 4: Explotar el Potencial de las Arquitecturas Modernas

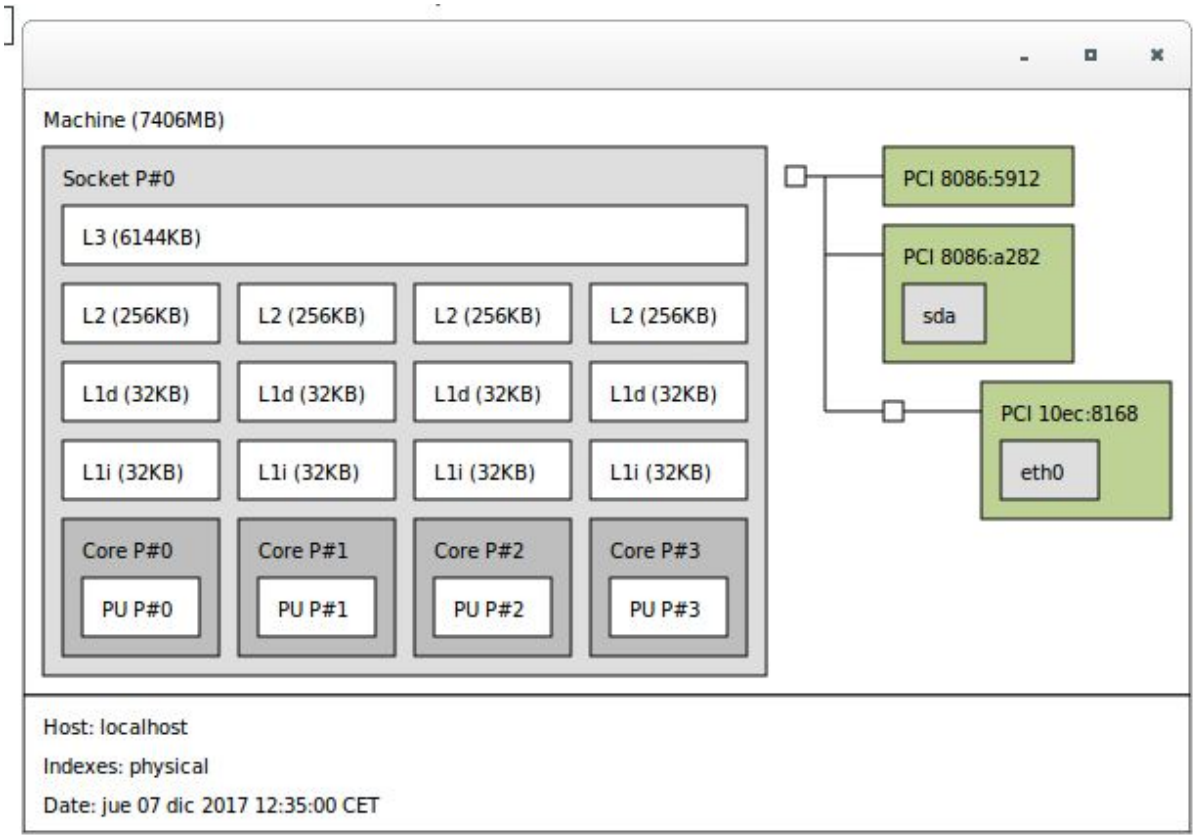
Aitor Arnaiz del Val
Andrés Salas Peña

PM29 - Gr1301 - Práctica 4
Arquitectura de Ordenadores

Ejercicio 0

INFORMACIÓN SOBRE LA TOPOLOGÍA DEL SISTEMA

Indique en la memoria asociada a la práctica los datos relativos a la cantidad y frecuencia de los procesadores disponibles en los equipos del laboratorio, y a si la opción de hyperthreading está o no activa.



Tal y como se puede observar en la captura de pantalla adjuntada con la práctica, el número de procesadores disponibles en el laboratorio es de 4. En cuanto a la frecuencia de los procesadores, los tres primeros procesadores (0,1,2) poseen una frecuencia de 800 MHz y el último de 1200 MHz (visible en las capturas adjuntas). En cuanto a la opción de hyperthreading, podemos decir que en todos los procesadores se encuentra disponible (flag -hp) pero en ninguno de ellos está activada ya que las CPU's virtuales (dado por el parámetro siblings) coinciden con las CPU's físicas (cpu_cores) en número, luego hay tantas CPU's virtuales como físicas y por tanto no hay hyperthreading.
Aquí se puede observar una visión más gráfica tras ejecutar el comando lstopo. No obstante, la información más detallada se encuentra en las capturas adjuntas.

Ejercicio 1

PROGRAMAS BÁSICOS DE OPENMP

```
e336041@localhost:~/UnidadH/FALSO/materialP4$ ./omp1 6
Hay 4 cores disponibles
Me han pedido que lance 6 hilos
Hola, soy el hilo 0 de 6
Hola, soy el hilo 4 de 6
Hola, soy el hilo 1 de 6
Hola, soy el hilo 2 de 6
Hola, soy el hilo 3 de 6
Hola, soy el hilo 5 de 6
e336041@localhost:~/UnidadH/FALSO/materialP4$ ./omp1 5
Hay 4 cores disponibles
Me han pedido que lance 5 hilos
Hola, soy el hilo 4 de 5
Hola, soy el hilo 3 de 5
Hola, soy el hilo 1 de 5
Hola, soy el hilo 2 de 5
Hola, soy el hilo 0 de 5
e336041@localhost:~/UnidadH/FALSO/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 1 de 4
Hola, soy el hilo 2 de 4
Hola, soy el hilo 3 de 4
e336041@localhost:~/UnidadH/FALSO/materialP4$ ./omp1 3
Hay 4 cores disponibles
Me han pedido que lance 3 hilos
Hola, soy el hilo 0 de 3
Hola, soy el hilo 2 de 3
Hola, soy el hilo 1 de 3
```

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

Tal y como se puede apreciar en la captura de pantalla adjunta “omp1.png” se pueden lanzar más hilos que cores disponibles tiene el sistema pero no tiene ningún sentido hacerlo puesto que el lanzamiento de hilos busca un máximo aprovechamiento de todos los cores del sistema. Una vez aprovechados todos los cores del sistema, crear un hilo más solo va a conllevar una pérdida de tiempo puesto que el programa tendrá que crear ese hilo y esperar a que termine alguno de los cores para mandarle el hilo creado.

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

Se deben lanzar tantos hilos como cores tenga el ordenador, en el caso de los laboratorios al poseer cuatro cores tiene sentido crear hasta 4 hilos.
En mi equipo, dispongo de cuatro cores luego el número de hilos que debería utilizar sería 4, al igual que en los laboratorios.

```
e336041@localhost:~/UnidadH/FALSO/materialP4$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7fff21fe7684,x   &b = 0x7fff21fe7688,    &c = 0x7fff21fe768c

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7fff21fe7658,    &b = 0x7fff21fe7688,    &c = 0x7fff21fe7654
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f52eb5b8e58,    &b = 0x7fff21fe7688,    &c = 0x7f52eb5b8e54
[Hilo 3]-2: a = 21,      b = 6,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f52ebdb9e58,    &b = 0x7fff21fe7688,    &c = 0x7f52ebdb9e54
[Hilo 2]-2: a = 27,      b = 8,  c = 3
[Hilo 1]-1: a = -1,      b = 2,  c = 3
[Hilo 1]         &a = 0x7f52ec5bae58,    &b = 0x7fff21fe7688,    &c = 0x7f52ec5bae54
[Hilo 1]-2: a = 34,      b = 10,         c = 4

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7fff21fe7684,    &b = 0x7fff21fe7688,    &c = 0x7fff21fe768c
```

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

En el programa dado de ejemplo, la variable definida como privada es **a**. Tal y como se observa, lo que hace OpenMP con esta variable es crear una copia de la misma en cada hilo (con una dirección distinta para cada uno) de tal forma que la copia de la variable solo sea accedida desde ese mismo hilo.

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

Siguiendo la ejecución del programa y su salida, vemos que **a** toma distintos valores para cada hilo perteneciente a la región paralela(Hilo 0 a=0; Hilo 3 a=0; Hilo 2 a=0; Hilo 1 a=-1). Esto es debido a que al ser declarada como privada, la variable no es inicializada y toma valores aleatorios. Si seguimos la ejecución dentro de cada hilo, observamos que las operaciones que efectúan los hilos son sobre su respectiva variable privada.

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

Una variable privada al finalizar la región paralela es eliminada perdiendo todos sus datos. El objetivo de declarar una variable privada es hacer que esta sea utilizada solo dentro de la región paralela sin que el resto de la ejecución serializada se de cuenta de los cambios de la variable. En el programa, se observa que cuando finalizan los hilos y se obtiene el valor de **a** de la región serializada, el valor es el mismo que antes de la paralelización tal y como se esperaba

1.6 ¿Ocurre lo mismo con las variables públicas?

Las denominadas como variables públicas son las que se declaran con la claúsula shared y en nuestro programa la variable shared es **b**. Estas son completamente distintas a las privadas. Lo primero es que no se genera una copia por hilo si no que, al ser pública, la dirección a la que acceden todos los hilos es la misma y se corresponde con la dirección de b en su ejecución serializada.

En cuanto a su valor, como todos los hilos modifican el valor de b de la misma dirección y cada hilo suma dos a la variable, el primer hilo ejecutado, el hilo 0, cambia el valor de b a 4, el siguiente hilo, el 3, coge ese último valor de b y le vuelve a sumar 2 dando a b el valor de 6. Así siguen el resto de hilos hasta que b acaba la región paralela con el valor de 10. Ese valor, a diferencia de las variables privadas, se conservará en la siguiente parte serializada porque todos los hilos accedieron a la misma dirección de memoria para modificar b (la dirección de b serializada).

1.7 Aunque no se contemple en las preguntas a responder, en este apartado se realiza un estudio de la diferencia entre firstprivate y private.

En el programa dado, la variable firstprivate es **c**, tal y como se observa la única diferencia que posee con una variable private como a es que el valor de la copia de c en cada hilo es el mismo que el de la parte serializable del código. Es decir, si la variable es first-private, el valor que se generará en cada hilo no será aleatorio si no que será el correspondiente a c en el momento antes de paralelizarse el programa.

Ejercicio 2

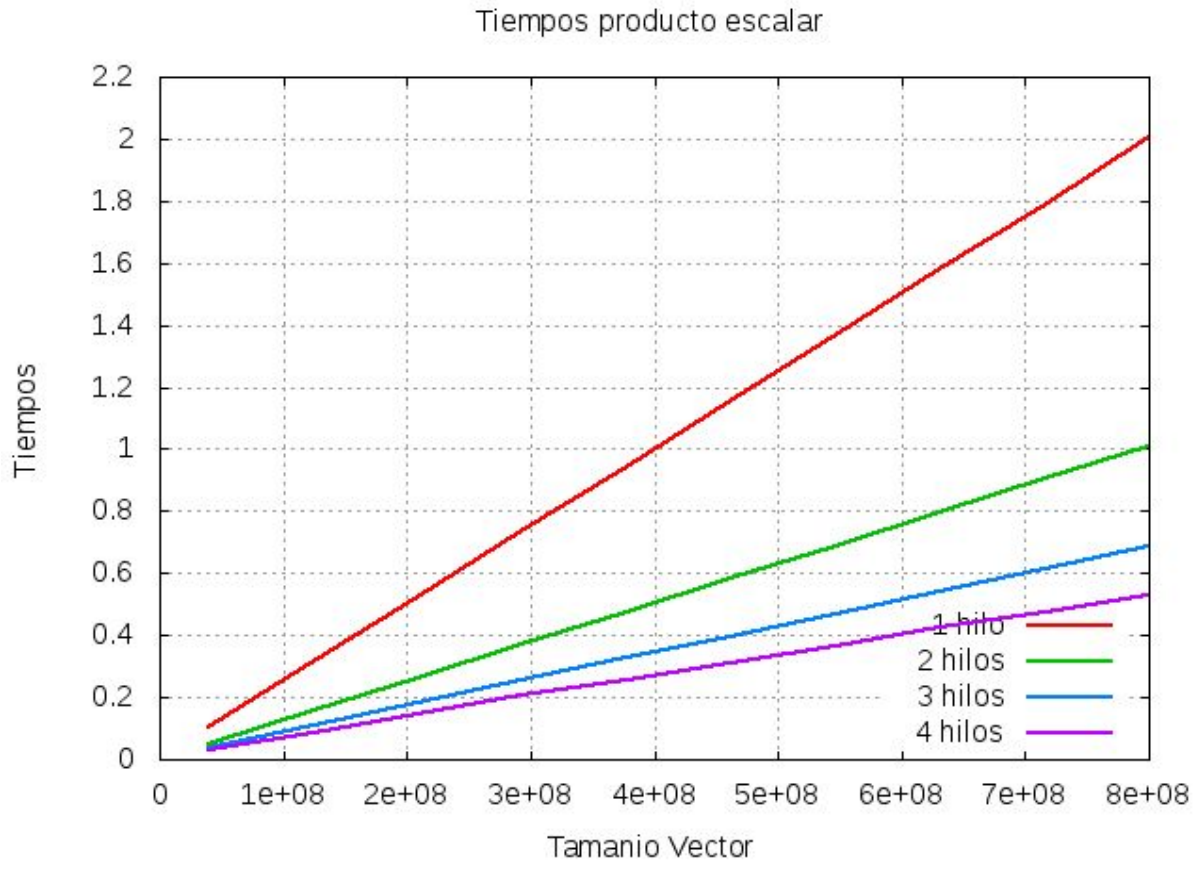
PARALELIZAR EL PRODUCTO ESCALAR

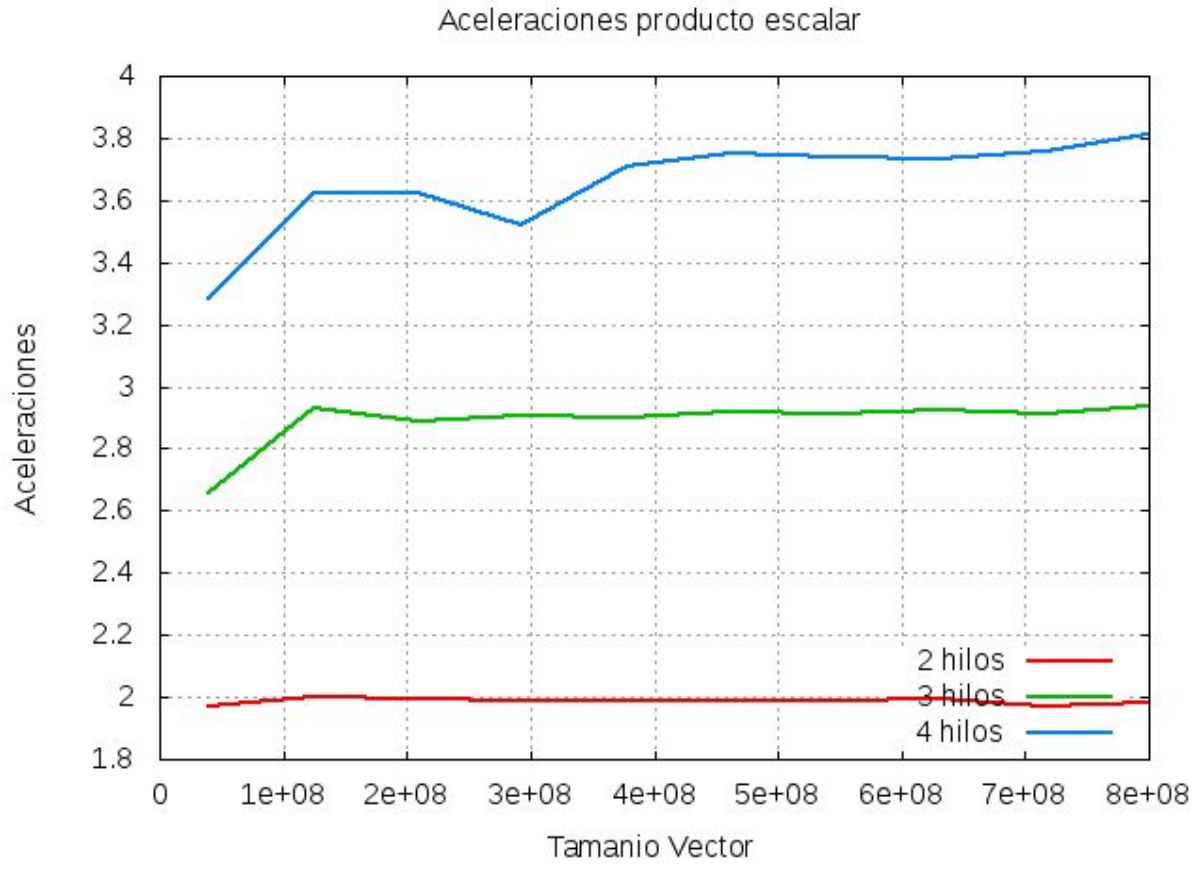
2.1 Ejecute pescalar_serie, pescalar_par1 y pescalar_par2. ¿En qué caso es correcto el resultado?

El resultado es correcto, tal y como se puede ver en la captura, para el programa pescalar_par2 porque no varía repitiendo la ejecución y coincide con el de pescalar_serie mientras que el de pescalar_par1 varía y además no coincide su resultado con las otras dos salidas.

2.2 ¿A qué se debe esta diferencia?

La diferencia se debe a la paralelización del cálculo, en pescalar_par1 se lanzan los hilos y cada uno ejecuta sus sumas correspondientes sin tener en cuenta al resto. En pescalar_par2 sin embargo, se usa la claúsula reduction(+:sum) que lo que hace es que cada hilo genera un resultado parcial que se recoge y se computa (con la operación “+”) para obtener el resultado final.


Tiempos producto escalar
Tiempos
Tamanio Vector
1 hilo
2 hilos
3 hilos
4 hilos
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
0
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
7e+08
8e+08


Aceleraciones producto escalar
Aceleraciones
Tamanio Vector
2 hilos
3 hilos
4 hilos
1.8
2
2.2
2.4
2.6
2.8
3
3.2
3.4
3.6
3.8
4
0
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
7e+08
8e+08

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?

Para tamaños de vectores relativamente pequeños la diferencia de tiempos es tan pequeña que prácticamente no compensa paralelizar el trabajo; sin embargo, cuando aumentamos considerablemente los tamaños se empieza a notar mucho más las diferencias entre hacer el mismo trabajo con 1 o múltiples hilos, con lo que para tamaños muy grandes es cuando más compensa.

Para tamaños extremadamente pequeños de vector, el tiempo que se pierde en paralelizar no compensa la rapidez que se gana en la ejecución y por tanto en estos casos no compensa.

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

Como se acaba de responder, cuando menos compensa es para los tamaños de vectores más pequeños, ya que los tiempos son tan pequeños que la mejora de rendimiento que nos proporciona la paralelización en hilos del trabajo es prácticamente despreciable.

2.5 ¿Se mejora siempre el rendimiento al aumentar el número de hilos a trabajar?

Tal y como se puede apreciar en ambas gráficas, sí. Cuanto mayor es el número de hilos, mayor es el rendimiento obtenido al hacer el producto escalar.
Analizando más en detalle los resultados, al utilizar 2 hilos se observa que la aceleración es de orden 2, al utilizar 3 hilos (aunque se pierde algo más de tiempo del esperado [en torno al 15%]) vemos que la aceleración se aproxima a 3. Al utilizar 4, se pierde también algo de tiempo, haciendo que la aceleración sea un 20% menor de lo esperado.
Observando más en detalle la gráfica de tiempos, podemos expresar que estos crecen de manera lineal. Las pendientes de las gráficas son más pronunciadas cuando tenemos menos hilos y esto es debido a que en tamaños más grandes, las diferencias entre el número de hilos usados se hacen más significativas.

2.6 Si no fuera así, ¿a qué debe este efecto?

Al aumentar el número de hilos al máximo que tiene sentido lanzar, se va a mejorar el rendimiento (salvo mala programación de la paralelización). Una vez sobrepasado este máximo, la mejora va a ser minúscula y por tanto no merecerá la pena lanzar más hilos.

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.

Parece que las gráficas de las aceleraciones se estabilizan a partir de $4*10^8$ y no se muestran indicios de que vayan a sufrir cambios para mayores tamaños, ya que los tiempos aumentarán de forma lineal con respecto del tiempo, y las aceleraciones se mantendrán

prácticamente constantes a partir de su estabilización en el valor citado. Concluyendo, no parece que haya un valor de vector a partir del cual las gráficas cambien de forma considerable.

Ejercicio 3

PARALELIZAR LA MULTIPLICACIÓN DE MATRICES

TAMAÑO 1000

Tiempos de Ejecución (s)				
Versión\# Hilos	1	2	3	4
Serie	7.015712			
Paralela - bucle1	7.189410	5.623031	6.342065	7.055736
Paralela - bucle2	7.259925	3.881327	2.938699	2.641877
Paralela - bucle3	8.070867	3.847926	2.655277	2.199403

Speedup (tomando como referencia la versión serie)				
Versión\# Hilos	1	2	3	4
Serie	1.000000			
Paralela - bucle1	0.955513	1.116748	1.335637	1.140819
Paralela - bucle2	1.001939	1.927921	2.570027	3.294271
Paralela - bucle3	1.004917	1.978285	2.669655	3.537817

TAMAÑO 1700

Tiempos de Ejecución (s)				
Versión\# Hilos	1	2	3	4
Serie	55.208836			
Paralela - bucle1	54.005379	30.487803	31.949482	32.432549
Paralela - bucle2	54.642525	29.215374	22.162165	17.688227
Paralela - bucle3	56.535015	29.871004	19.644930	16.630623

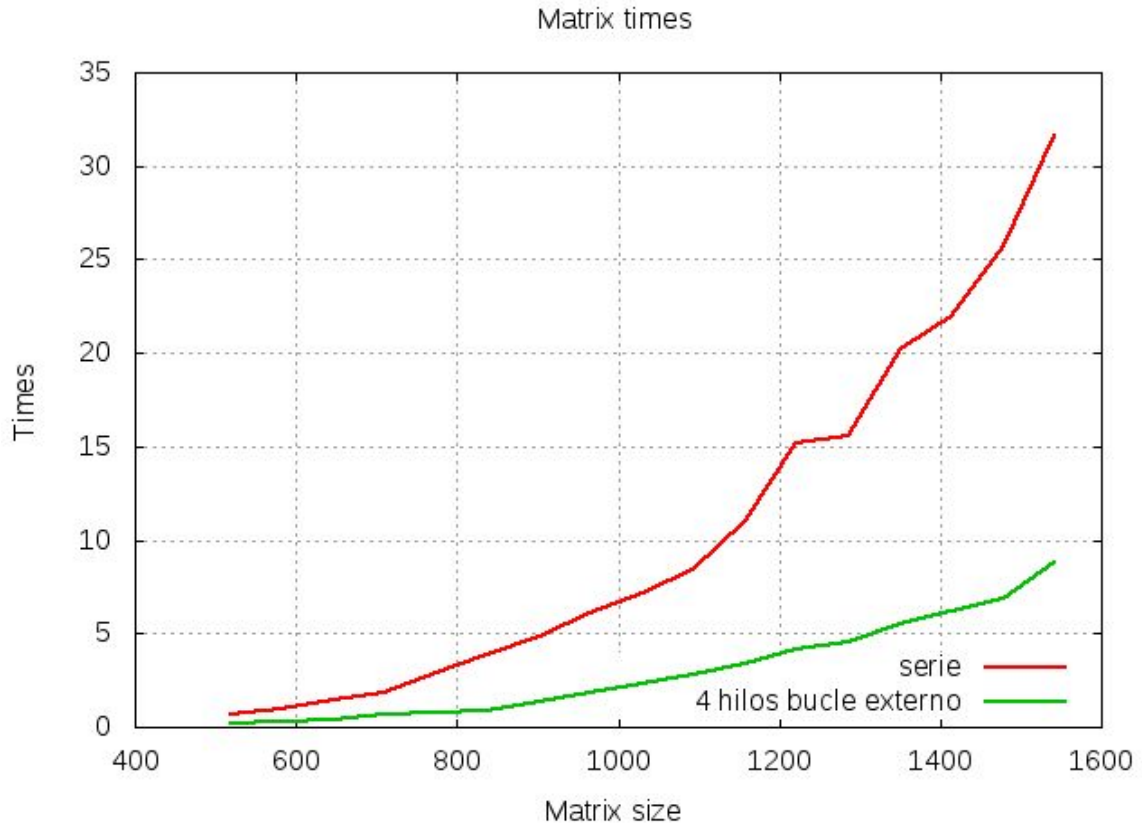
Speedup (tomando como referencia la versión serie)				
Versión\# Hilos	1	2	3	4
Serie	1.000000			
Paralela - bucle1	1.008096	1.855983	1.741592	1.730414
Paralela - bucle2	1.730414	1.918299	2.561886	3.210128
Paralela - bucle3	0.965550	1.957476	2.622323	3.356801

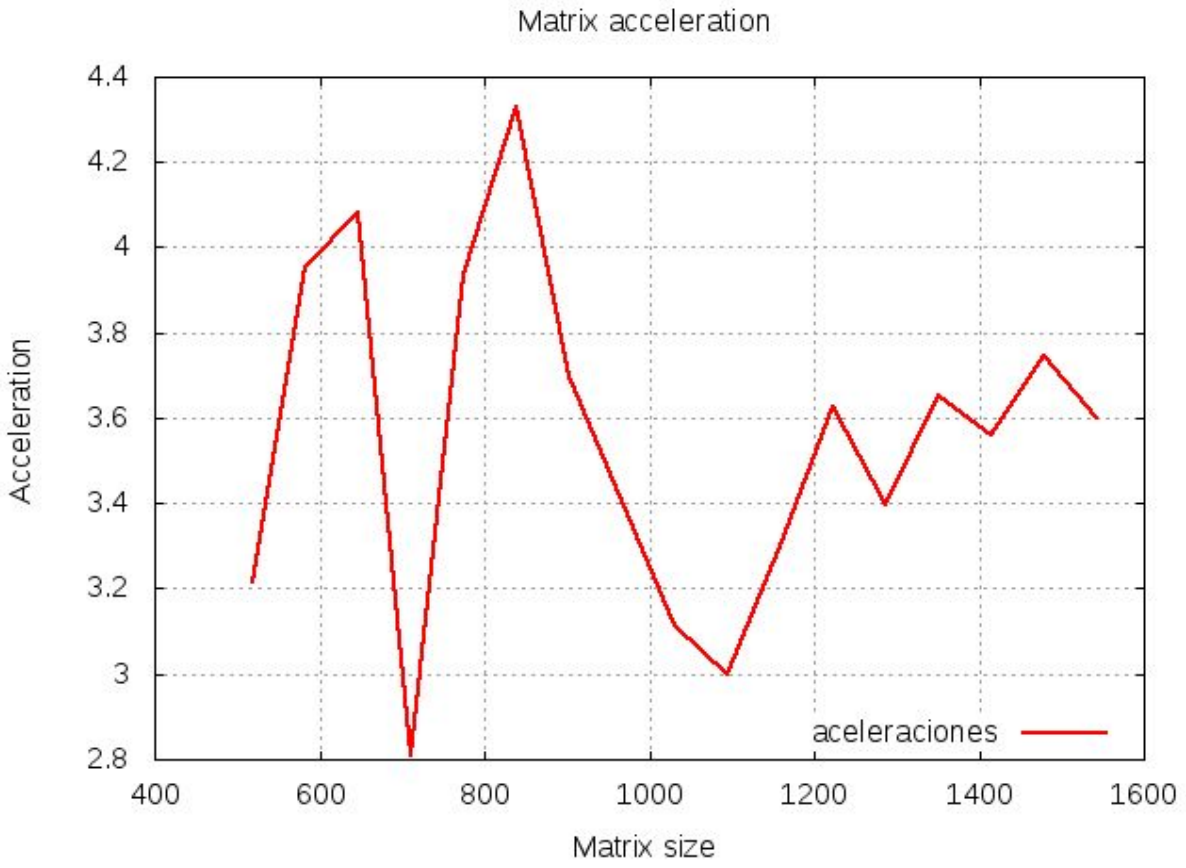
3.1 ¿Cuál de las tres versiones paralelas obtiene peor rendimiento? ¿A qué se debe?

Para responder a esta pregunta, nos fijamos en las tablas tanto de tiempos como de aceleración. Podemos observar que la versión más lenta (la que más tiempo tarda y la que menos aceleración tiene) es la paralela bucle 1. Esto es debido a que al ser el bucle más interno, paralelizas menos variables haciendo un menor uso de la paralelización que en el resto

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

Al igual que en el ejercicio anterior, nos fijamos en las tablas y observamos que la más rapida (la que menos tiempo tarda y la que más aceleración posee) es la paralela bucle 3. Esto es debido al efecto contrario de la pregunta anterior, paralelizamos todas las variables posibles haciendo que el uso de la paralelización y su aprovechamiento sea máximo.





Como se puede observar en la gráfica de tiempos adjunta, los tiempos de la multiplicación de matrices son mucho menores en la versión mejorada del programa (4 hilos, bucle externo) como era de esperar. En la gráfica de aceleraciones se refleja esta mejora con respecto a la versión serie que como se puede observar, presenta oscilaciones en los mismos lugares donde lo hace la gráfica de tiempos (pues la aceleración se obtiene a partir de los tiempos de la gráfica de tiempos), oscilando ésta en torno a valores próximos a 3 o 4, tendiendo a estabilizarse finalmente como podemos ver en torno a una aceleración de 3.6 aproximadamente.

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 qué valor de N se empieza a ver el cambio de tendencia.

En la explicación de las gráficas hemos hecho un estudio de las mismas. En dichas gráficas, vemos como la tendencia a estabilizarse se produce para valores de N alrededor de los 1300 y se estabiliza para una aceleración cercana a los 3,6. El porqué de este valor y no el valor 4 esperado, se discutió en el ejercicio anterior.

Ejercicio 4

EJEMPLO DE INTEGRACIÓN NUMÉRICA

4.1 ¿Cuántos rectángulos se utilizan en la versión del programa que se da para realizar la integración numérica (pi_serie)?

Tal y como se puede apreciar en el programa, el número de rectángulos usados es el correspondiente al valor n = 100000000.

4.2 ¿Qué diferencias observa entre las versiones pi_par1.c y pi_par4.c?

En la versión pi_par1.c se observa que el cálculo de la suma se hace directamente sobre la variable public sum[tid] mientras que en la versión pi_par4.c se crea una variable privada en cada hilo en la que se hará la suma y después estas son asignadas a la variable public sum[tid].

```
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par1
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.481404
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.327854
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par1
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.501091
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.327952
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par1
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.491818
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.328873
```

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?

Tras ejecutar las dos versiones, observamos que el resultado obtenido es el mismo tal y como era de esperar porque ambos programas están bien implementados y realizan bien las operaciones. Sin embargo, en cuanto a rendimiento, el cálculo es muy rápido pero se pueden observar diferencias. En pi_par1 se obtienen tiempos ligeramente superiores a pi_par4, estas diferencias serían más acusadas en programas más costosos y lentos. El efecto de esta ligera diferencia de tiempo es debido a las diferencias explicadas en la respuesta anterior: el programa pi_par1 ejecuta las operaciones dividiéndose en hilos pero al ser pública la variable, cada vez que se realizan las operaciones correspondientes, se va modificando la dirección de la variable pública y el resto de hilos tienen que estar

pendientes de estas modificaciones de la variable pública constantemente. Sin embargo, en el programa pi_par4, las operaciones se hacen sobre una variable privada y una vez se obtiene el resultado definitvo, este es guardado en la variable pública. Los accesos a variables privadas son mucho más rápidas y menos costosas que los accesos a variables públicas ya que en este último caso, todos los hilos deben de estar pendientes con más frecuencia de los cambios en las variables públicas.

```
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par2
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.459258
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.325572
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par2
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.502919
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.323863
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./pi_par2
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.468560
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.333344
```

4.4 Ejecute las versiones paralelas 2 y 3 del programa (pi_par2.c y pi_par3.c) . ¿Qué ocurre con el resultado y el rendimiento obtenido? ¿Ha ocurrido lo que se esperaba?

La modificación realizada en pi_par2.c con respecto a pi_par1.c consiste en declarar como firstprivate la variable sum pero esto no afecta prácticamente en nada al rendimiento.
En pi_par3 lo que se busca es que cada hilo ocupe el tamaño máximo del bloque. Ambos programas están bien codificados lo que hace que se consiga el resultado adecuado. En cuanto al rendimiento de ambos, observamos que el de pi_par3.c es algo mejor ya que aprovechas siempre el tamaño máximo de bloque y no genera bloques adicionales que conllevan una ligera pérdida de tiempo mayor. Por tanto, podemos concluir que los resultados obtenidos son los esperados.

```
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.490441
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.497475
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.367939
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.344361
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.351143
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.334900
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.331460

e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.314215
e336041@localhost:~/Downloads/FALSO/Ejercicio4/Codigo$ ./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 0.312964
```

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?

Se observa que cuanto más grande es el valor del padding más rápido es pi_par3.c sin llegar sin embargo a ser suficientemente rápido como pi_par4.c El motivo de la mejora de la rapidez de pi_par3.c al aumentar el tamaño es, como explicábamos en el apartado anterior, el mejor aprovechamiento del tamaño del bloque. Con más elementos que tamaño de bloque, se generan dos bloques, haciendo que esto mejore un poco más el rendimiento del programa a cambio de gastar más recursos, ya que el segundo bloque se encuentra casi vacío (en el caso de 10 solo tendrá 2 elementos de 8 disponibles y en el caso de 12, 4 de 8).

Ejercicio 5

USO DE LA DIRECTIVA CRITICAL Y REDUCTION

```
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.329950
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par5
Resultado pi: 2.723127
Tiempo 0.927608
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.325188
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par5
Resultado pi: 3.448238
Tiempo 1.813850
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.322943
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par5
Resultado pi: 3.379038
Tiempo 1.756863
```

5.1 Ejecute las versiones 4 y 5 del programa (pi_par4.c y pi_par5.c) . Explique el efecto de utilizar la directiva critical. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

La directiva critical como su nombre indica, lo que hace es definir una sección crítica dentro de una paralelización. Esto quiere decir que solo un único hilo podrá acceder a esta sección del código. Esto provoca en este programa que los resultados no se correspondan con el esperado ya que faltaría una cláusula reduction para recoger todos los resultados al final. La ejecución de pi_par5.c es aproximadamente el doble de lenta que pi_par4.c esto es debido a que todos los hilos lanzados deben esperar a que otro hilo termine en la sección crítica y esta quede libre para su uso.

```
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par6
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.488535
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.326523
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par6
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.496682
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.324820
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par6
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.502603
e336041@localhost:~/Downloads/FALSO/Ejercicio5/Codigo$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.327042
```

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

En estas dos últimas versiones del cálculo de pi se usan conceptos ya mencionados en otros ejercicios de la práctica, en pi_par6 se usa el concepto de work sharing para el bucle interno de la sección paralela mientras que en pi_par7 se usan a la vez el concepto de work sharing y la claúsula reduction. Aunque ligeramente, es mejor el rendimiento de pi_par7 que el de pi_par6 ya que al incluir la claúsula reduction se evita que los hilos estén pendientes de los demás, cada hilo hace su cálculo y después la claúsula reduction, una vez terminados todos los hilos, se encarga de recoger todas las sumas parciales.

Aclaraciones

Todas las ejecuciones se han realizado con P=6 ya que tal y como se indica, el valor de P se calcula con el número de la pareja módulo 8 y sumando 1 al resultado.

29 mod 8 = 5; 5+1 = 6

Adjuntas se encuentran distintas carpetas en las que además de los ficheros y scripts pedidos se encuentran las capturas realizadas para el posterior análisis de los resultados. Además de dichas capturas, se encuentran todos los programas necesarios para el apartado correspondiente, con su makefile propio.

En el guión de la memoria de la práctica facilitado para el ejercicio 2 no se hacía un estudio de las diferencias entre first-private y private. Hemos considerado interesante a la vez que oportuno (el programa ejecutado declaraba una variable también first-private) comentarlas y ese es el motivo de la existencia de un apartado 1.7.

Aunque para lograr unos resultados más realistas se debería haber realizado las ejecuciones como se explicaron en la práctica anterior, en los ejercicios 2 y 3 nos hemos ceñido estrictamente al resultado, haciendo 5 repeticiones para el ejercicio 2 pero sin preocuparnos por la alternancia de las ejecuciones y sin hacer repeticiones para el ejercicio 3. Los resultados son lo suficientemente parecidos a las ejecuciones con repeticiones y alternadas así que nos sirven ambas gráficas para explicar los fenómenos ocurridos.