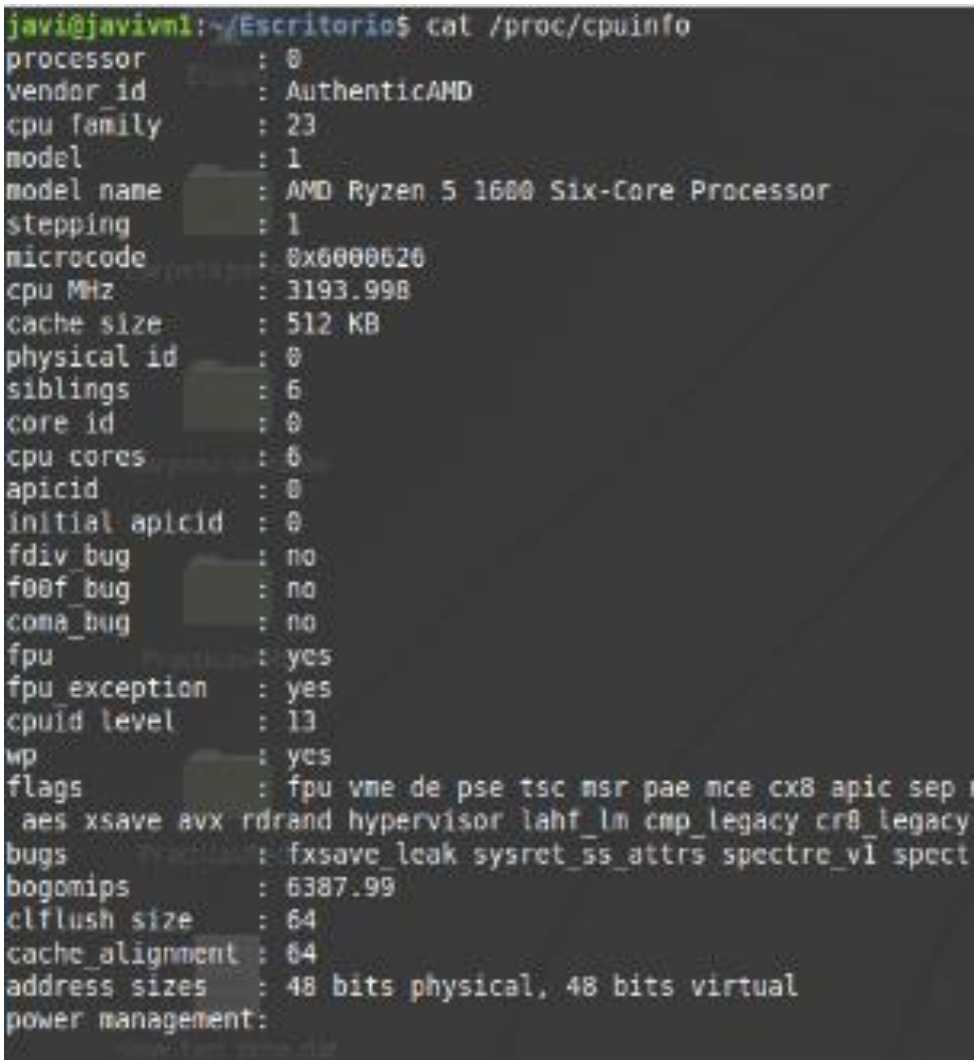
Práctica 4

Ejercicio 0

Al analizar el ordenador que vamos a emplear para ejecutar las diferentes partes de la práctica con el comando “cat /proc/cpuinfo” obtenemos en la salida una gran cantidad de información acerca del procesador del equipo.



Observando esta información vemos que se trata de un procesador con una frecuencia máxima de 3193,998 Mhz, es decir, aproximadamente 3,2Ghz, y sin hypertherading puesto que dispone de 6 cores físicos mientras que tiene 6 siblings, es decir, dispone de 6 threads. Como el número de threads es igual que el de cores, vemos que no tiene hyperthreading.

Ejercicio 1

1.1-1.2

Los ordenadores permiten lanzar más threads que cores tenga un sistema. Pero esto no siempre tiene sentido pues no todos los procesadores soportan hyperthreading y, por tanto, en los casos en los que no se soporte esto, no habría un aumento de rendimiento al lanzar más threads que cores, sin embargo, si un sistema sí soporta esta tecnología, sí que tendría sentido lanzar hasta 2 veces más threads que cores.

Como los ordenadores del laboratorio disponen de 4 cores sin hyperthreading, solo tendría sentido usar hasta 4 threads (tantos como cores), en el caso del cluster tendría sentido ejecutar hasta 128 threads pues este dispone de 16 procesadores cada uno de los cuales tiene 8 cores, lo que suma un total de 128 cores que, y como no soportan hyperthreading solo tiene sentido que ejecute un hilo cada uno, lo que supone que ejecutar más de 128 threads a la vez no tienen sentido.

En nuestro ordenador personal (el que vamos a emplear para ejecutar las distintas partes de esta práctica) tendría sentido ejecutar hasta 6 threads a la vez pues dispone de un único procesador con 6 cores sin hyperthreading.

1.3-1.5

Tras leer el código fuente de los programas omp1.c y omp2.c, los ejecutamos y obtenemos la siguiente salida:

```
javi@javivm1:~/Escritorio/PracticasARQO/Practica4/Ej2$ ./omp2
Inicio: a = 1,    b = 2,   c = 3
        &a = 0xbf975fc8,x       &b = 0xbf975fcc,        &c = 0xbf975fd0

[Hilo 3]-1: a = 12,       b = 2,   c = 3
[Hilo 3]          &a = 0xb6d492b4,        &b = 0xbf975fcc,        &c = 0xb6d492b0
[Hilo 3]-2: a = 159,      b = 4,   c = 147
[Hilo 2]-1: a = 8,        b = 2,   c = 3
[Hilo 2]          &a = 0xb754a2b4,        &b = 0xbf975fcc,        &c = 0xb754a2b0
[Hilo 2]-2: a = 85,       b = 6,   c = 67
[Hilo 0]-1: a = -1208512512,      b = 6,   c = 3
[Hilo 0]          &a = 0xbf975f44,        &b = 0xbf975fcc,        &c = 0xbf975f40
[Hilo 0]-2: a = 822083611,        b = 8,   c = 822083587
[Hilo 1]-1: a = 4,        b = 6,   c = 3
[Hilo 1]          &a = 0xb7d4b2b4,        &b = 0xbf975fcc,        &c = 0xb7d4b2b0
[Hilo 1]-2: a = 49,       b = 10,          c = 19

Fin: a = 1,       b = 10,          c = 3
        &a = 0xbf975fc8,        &b = 0xbf975fcc,        &c = 0xbf975fd0

javi@javivm1:~/Escritorio/PracticasARQO/Practica4/Ej2$
```

Observando la salida obtenida podemos deducir que cuando declaramos una variable privada, openMP crea una variable independiente con el mismo nombre para cada thread. Si uno de los threads modifica su variable, la variable del mismo nombre del resto de threads no se modifica pues son variables diferentes, que openMP inicializa a 0 el valor de las variables privadas cuando se inicia la región paralela y que, cuando esta región acaba, la variable vuelve a tomar el valor que tenía antes de la región paralela.

1.6

En el caso de las variables públicas, vemos que estas son compartidas por todos los threads, y que los cambios que realice en ellas un thread se verán reflejados en el resto. Al finalizar la región paralela, al contrario que con las variables privadas, esta variable mantendrá el valor que tenía al final de la región paralela (tras ser modificada por el último thread).

Ejercicio 2

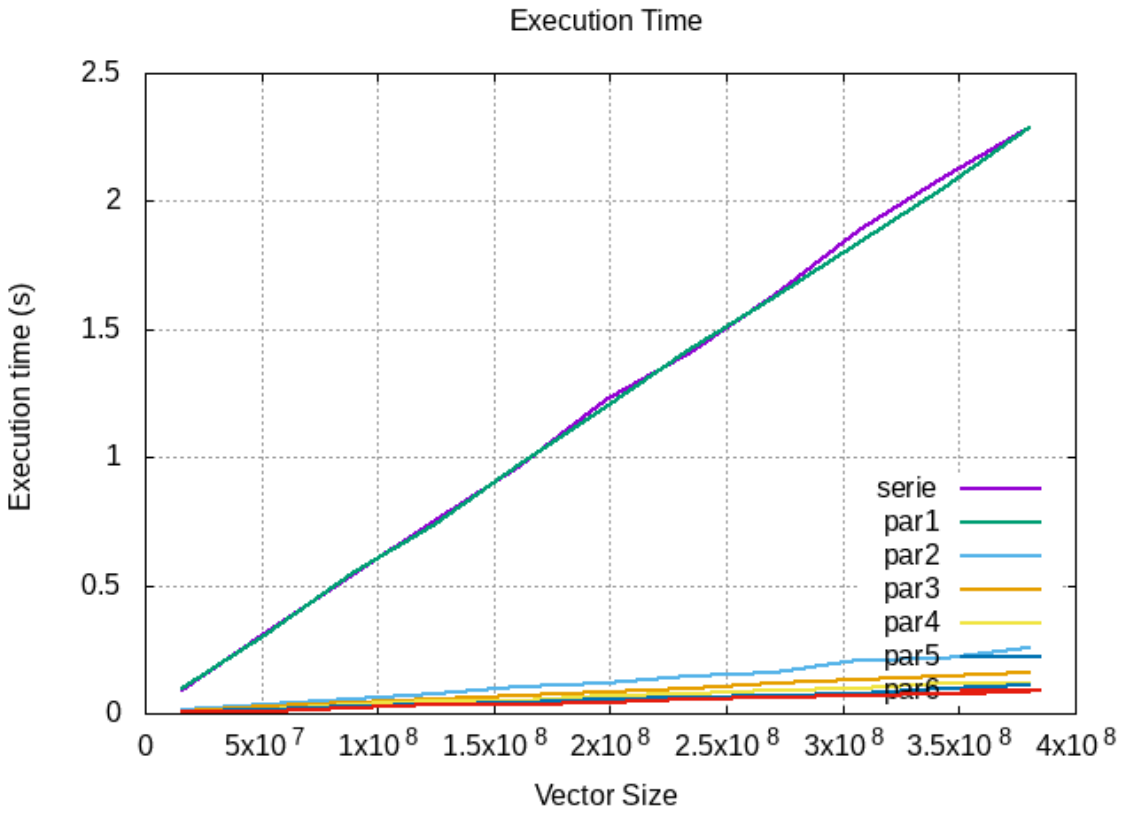
Para la realización de este ejercicio ejecutamos los programas pescalar_serie, pescalar_par1 y pescalar_par2 que se nos proporcionan en el material de la práctica. Estos programas generan 2 vectores del tamaño que se les indique y realizan su producto escalar.

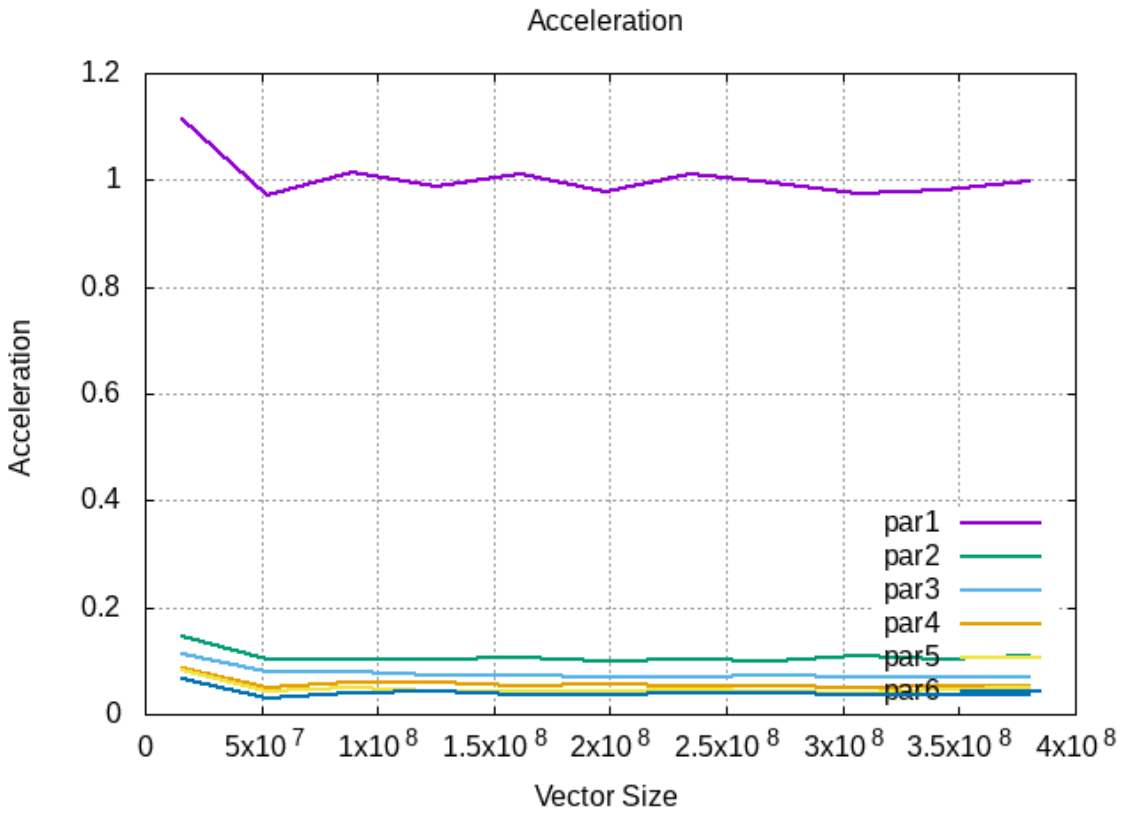
Ejecutamos estos programas y observamos que el resultado es correcto en el caso de pescalar_serie pues no emplea hilos por lo que no puede haber problemas de sincronización, y también es correcto en el caso pescalar_par2 pues al incluir en la declaración de la región de openMP el atributo "reduction(+: sum)" openMP crea una copia interna de la variable “sum” dentro de cada thread y al acabar la región paralela suma estas copias internas en una única evitando así problemas de sincronización. Esto no se realiza en el caso pescalar_par1 haciendo que surjan en este tipo de problemas de sincronización generando un resultado incorrecto.

2.1-2.2

Tras esto ejecutamos los 2 programas que generan la salida correcta con los tamaños de vectores y las repeticiones que se nos pide en la práctica. En este caso hemos escogido como tamaño mínimo 16.000.000 pues es el tamaño para el cual el tiempo de ejecución se aproxima más a 0,1s, y como tamaño máximo 380.000.000 que tarda aproximadamente 2,5s (no escogimos un tamaño más grande pues el ordenador en el que lo llevamos a cabo no nos permitía reservar memoria para vectores más grandes), con saltos de 36.640.000 para obtener un total de 10 tamaños; y realizamos un total de 5 repeticiones para cada tamaño y número de hilos (en nuestro caso el número máximo de hilos es 6).

Al ejecutar este script obtenemos una serie de ficheros con los datos y las 2 gráficas siguientes que los representan:


Execution Time
Execution time (s)
Vector Size
serie
par1
par2
par3
par4
par5
par6
0
0.5
1
1.5
2
2.5
5x10^7
1x10^8
1.5x10^8
2x10^8
2.5x10^8
3x10^8
3.5x10^8
4x10^8


Acceleration
Acceleration
Vector Size
par1
par2
par3
par4
par5
par6
0
0.2
0.4
0.6
0.8
1
1.2
5x10^7
1x10^8
1.5x10^8
2x10^8
2.5x10^8
3x10^8
3.5x10^8
4x10^8

2.3-2.3

Basándonos en los resultados obtenidos, parece que para los tamaños de los vectores seleccionados siempre es conveniente lanzar hilos al calcular el producto escalar para reducir el tiempo de ejecución. Sin embargo, si hubiésemos empleado tiempos más pequeños habríamos visto que esto no es tan eficiente pues cuanto más pequeño sea el tamaño del vector mayor será la aceleración (calculada como tiempo paralelo/tiempo serie) y por tanto menor será la diferencia de rendimiento entre lanzar hilos y no lanzarlos por lo que para tamaños muy pequeños es posible que no fuese rentable lanzar hilos.

2.4-2.5

Al aumentar el número de hilos para trabajar en general sí se mejora el rendimiento del programa, sin embargo, en tamaños muy pequeños podría darse el caso de que el tiempo que se tarde en lanzar los hilos sea muy similar al que se tarda en realizar la operación en sí, en cuyo caso no sería mayor el rendimiento cuanto mayor fuese el número de hilos lanzaos sino todo lo contrario.

2.6

Como hemos dicho anteriormente cabe destacar que si estuviésemos ante tamaños de vectores muy pequeños la aceleración se comportaría de forma muy diferente a lo que observamos en la gráfica puesto que cuanto menor sea el tamaño menor será la diferencia entre el tiempo en serie y el tiempo en paralelo y, por tanto, mayor será el valor de la aceleración. Por lo tanto, para tamaños pequeños, la gráfica mostraría una aceleración mucho más elevada.

Ejercicio 3

Para realizar este ejercicio partimos del código empleado para la multiplicación de matrices en la práctica anterior y lo modificamos creando 4 versiones distintas, una en serie (igual que la de la práctica anterior) matrix_mult.c y 3 que paralelizan cada una uno de los bucles de la multiplicación de matrices matrix_mult_1.c, matrix_mult_2.c y matrix_mult_3.c, donde la versión uno paraleliza el bucle más interno y la 3 el más externo.

Ejecutamos estas distintas versiones de la multiplicación de matrices para los tamaños 1150x1150 y 1850x1850 pues son los que más se aproximaban a 10s y 60s en serie en el ordenador donde lo ejecutamos. Para llevar a cabo esta ejecución empleamos el script Ej3.sh que genera varios archivos con las velocidades de las distintas versiones y sus aceleraciones correspondientes calculadas como tiempoParalelo/tiempoSerie.

Con los datos obtenidos rellenamos las siguientes tablas:

Tiempo de Ejecución (1150) (s)						
Versión/nº Hilos	1	2	3	4	5	6
Serie	10,36					
Paralelo-bucle1	6,72	5,24	4,97	5,13	5,85	5,87
Paralelo-bucle2	5,82	3,23	2,19	1,57	1,36	1,25
Paralelo-bucle3	5,87	2,85	1,98	1,46	1,18	1

Tiempo de Ejecución (1850) (s)						
Versión/nº Hilos	1	2	3	4	5	6
Serie	72,39					
Paralelo-bucle1	62,68	22,74	20,36	15,92	17,52	17,11
Paralelo-bucle2	57,95	27,77	18,95	13,93	11,21	9,43
Paralelo-bucle3	57,01	28,44	16,21	12,92	10,74	8,93

Speedup (1150)						
Versión/nº Hilos	1	2	3	4	5	6
Serie	1					
Paralelo-bucle1	0,65	0,51	0,48	0,49	0,56	0,56
Paralelo-bucle2	0,56	0,31	0,21	0,15	0,13	0,12
Paralelo-bucle3	0,57	0,28	0,19	0,14	0,11	0,1

Speedup (1850)						
Versión/nº Hilos	1	2	3	4	5	6
Serie	1					
Paralelo-bucle1	0,87	0,31	0,28	0,22	0,24	0,24
Paralelo-bucle2	0,8	0,38	0,26	0,19	0,15	0,13
Paralelo-bucle3	0,79	0,39	0,22	0,18	0,15	0,12

3.1-3.2

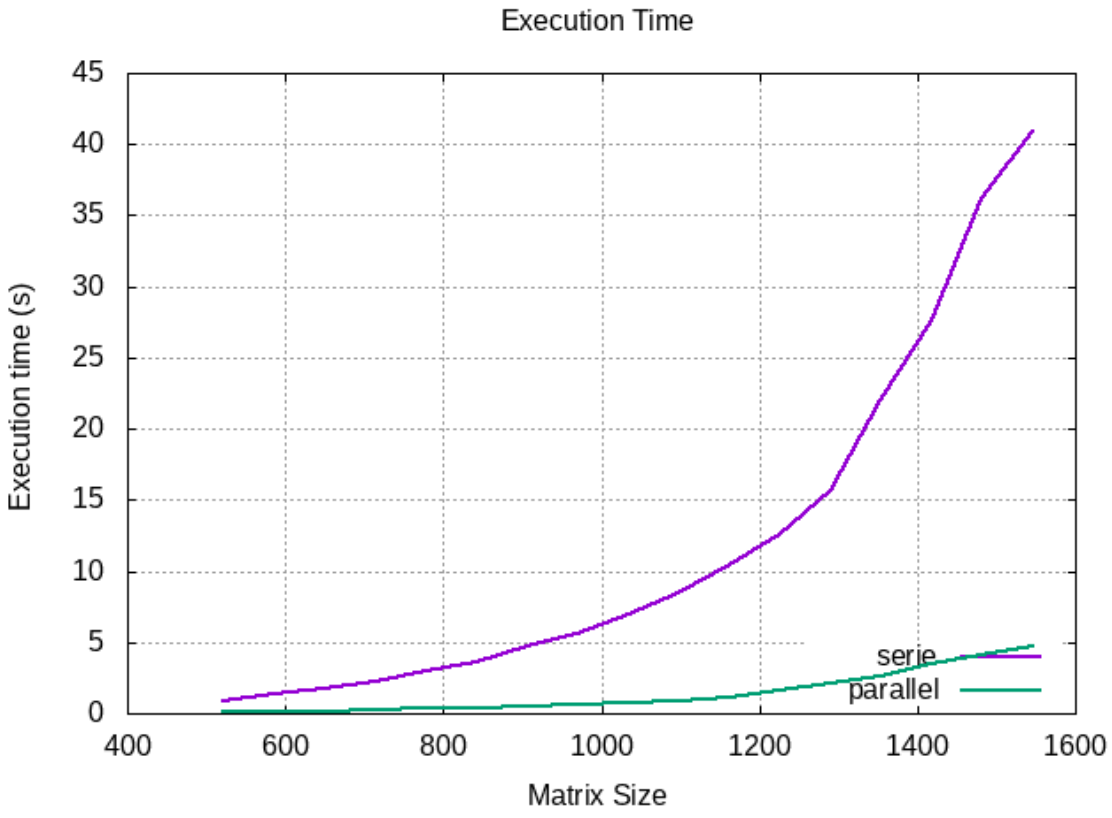
En las tablas se puede observar que todas las versiones paralelizadas obtienen un mejor rendimiento que la versión en serie incluso cuando estas últimas emplean tan solo 1 hilo. Esto se debe probablemente a que, al paralelizar los bucles hemos tenido que emplear la directiva "reduction(+: sum)" para evitar que varios hilos modificasen la misma posición de la matriz a la vez, haciendo de este modo que los hilos tengan que acceder al doble puntero de la matriz muchas menos veces (pues las sumas de los productos las realiza sobre la variable privada sum en vez de directamente sobre la matriz) que en la versión en serie, ya que en esta se tiene que acceder a la matriz cada vez que se lleva a cabo un producto, puesto que las sumas se realizan

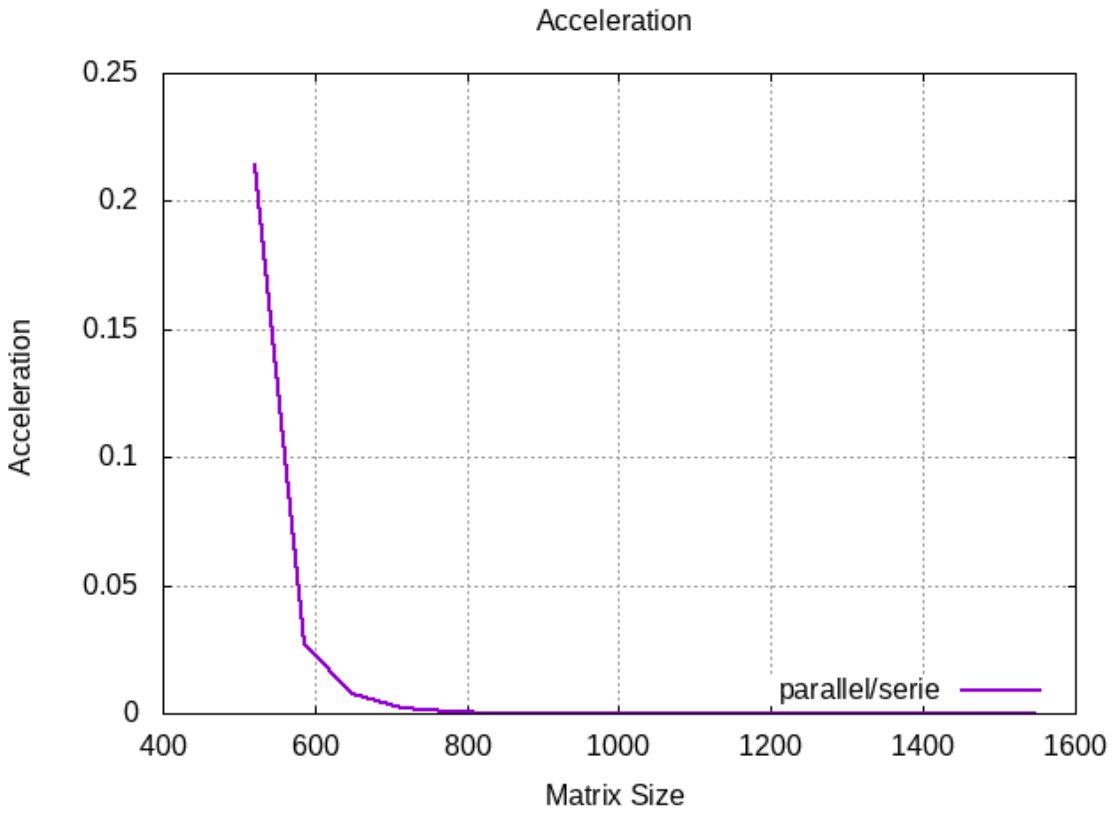
directamente sobre el elemento de la matriz. Esta diferencia a la hora de acceder a la matriz es la que causa esta diferencia en el tiempo de ejecución.

Es fácil observar a partir de los resultados de estas tablas que la versión paralelizada más eficiente es la que paraleliza el bucle 3, es decir, el más externo. Esto se debe a que al paralelizar el bucle más externo se consigue que cada hilo haga algunas de sus iteraciones, luego cada uno de los hilos estaría a su vez haciendo tan solo las iteraciones de los bucles más internos correspondientes a las iteraciones del bucle externo que le tocan, es decir, que, aunque tan solo el bucle externo esté paralelizado, esto hace que el trabajo de los bucles internos también se lo distribuyan entre todos los hijos ejecutados.

Del mismo modo podemos ver que la versión paralelizada menos eficiente es la que paraleliza el bucle 1, es decir, el bucle más interno. De forma opuesta a lo anterior, esto se debe a que al paralelizar tan solo el bucle más interno, los 2 bucles más externos se realizan en serie, de modo que su trabajo no se divide como en la versión en que se paraleliza el bucle más externo.

Para ejecutar tan solo las versiones en serie y la versión más eficiente de las paralelizadas (la del bucle 3) con el número de hilos más eficiente (6) y con las repeticiones y los tamaños solicitados hemos creado un script llamado Ej3_2.sh Este script genera 3 archivos con los datos pedidos, timesSerie.dat, timesPar.dat y accPar.dat, y 2 gráficas que los representan, times.png y acceleration.png, que son las siguientes:





En las gráficas se puede observar en primer lugar que el tiempo de ejecución es mucho menor en la versión en paralelo que en serie, como era de esperar, ya que la versión en paralelo divide el "trabajo" entre los distintos hilos que se ejecutan, y, también, se ve que en ambos casos el tiempo de ejecución es mayor cuanto mayor es el tamaño de la matriz puesto que cuanto más grande sea la matriz mayor será el número de iteraciones que se deberán realizar para llevar a cabo la multiplicación.

En el caso de la gráfica de la aceleración se observa claramente como esta disminuye conforme aumenta el tamaño de la matriz, esto se debe a que cuanto mayor sea el tamaño de la matriz mayor será la eficiencia del programa que realiza el proceso en paralelo (es decir, tardará menos en comparación con el de serie) mientras que mayor será el tiempo que tarda el programa en serie, y, como la aceleración se calcula como tiempoParalelo/tiempoSerie como el tiempo en serie aumenta más que el tiempo en paralelo, esta aceleración se aproxima a cero a medida que aumenta el tamaño. Cabe destacar que cuando las matrices comienzan a ser muy grandes, la aceleración comienza a estabilizarse en valores muy próximos a cero, esto se debe a que cuando el tiempo en paralelo es muy pequeño en comparación con el de serie, la aceleración se queda muy próxima a cero y por mucho que el tiempo en serie continúe aumentando la aceleración disminuirá muy poco notablemente ya que estará muy cerca de su mínimo que sería cero (nunca llega a él).

3.3

En la gráfica de aceleración se puede observar que el cambio de tendencia se da cuando las matrices tienen un tamaño alrededor de 600x600, que es mas o menos cuando la aceleración comienza a estabilizarse en valores muy cercanos a 0.

Ejercicio 4

4.1

Para realizar este ejercicio, en primer lugar, observamos en detalle el código fuente del programa pi_serie.c que obtiene un valor aproximado de pi por integración numérica y vemos que para llevar a cabo este cálculo emplea un total de 100.000.000 rectángulos.

4.2-4.3

Tras esto pasamos a analizar las versiones paralelas que se nos proporcionan de este programa, fijándonos en primer ligar en pi_par2 y pi_par4. La diferencia principal entre estos 2 programas es que el programa pi_par1 va actualizando constantemente el valor que ha calculado en el array compartido mientras que pi_par4 calcula primero el valor final y después lo guarda en la posición correspondiente del array compartido, por lo tanto, pi_par1 va accediendo constantemente a este array compartido mientras que pi_par4 solamente necesita hacer un acceso a este al finalizar el cálculo para guardar el dato.

En ambas versiones el resultado que se calcula es exactamente igual, sin embargo, pi_par4 lo calcula en mucho menos tiempo que pi_par1, aproximadamente 10 veces más rápido en el ordenador en que lo hemos probado. Esto probablemente se debe a que, como hemos comentado, en pi_par1 cada uno de los hilos ejecutados accede al array compartido y modifica el campo que le corresponde cada vez que calcula el valor del área de uno de los rectángulos, accediendo así un gran número de veces a este; sin embargo, en pi_par4 cada hilo accede tan solo 1 vez al terminar de calcular la suma de las áreas de todos los rectángulos que le corresponden en una variable propia. De este modo, como en pi_par4 se tiene que acceder mucho menos al array compartido se da menor competencia, y los hilos tienen menor probabilidad de tener que esperar a que otro termine de acceder al array compartido para poder acceder él a modificarlo, por lo que tarda menos.

4.4

Al ejecutar las versiones pi_par2 y pi_par3, que pretenden modificar levemente pi_par1 para obtener una eficiencia similar a la de pi_par4, vemos que, aunque ambas dan el resultado correcto, pi_par2 no obtiene la mejora esperada de tiempo, de hecho, prácticamente no se aprecia diferencia en el tiempo que tarda con respecto a pi_par1, sin embargo, pi_par3 si tarda el tiempo esperado aproximándose mucho al tiempo de pi_par4.

4.5

Al modificar la línea 32 de pi_par3 para que tome los valores 1, 2, 4, 6, 7, 8, 9, 10 y 12 se observa con bastante claridad que cuanto mayor es este valor, el tiempo que tarda el programa en calcular el valor de pi es menor, por lo tanto, el rendimiento mejora cuanto mayor es este campo. Esto se debe a que cuanto mayor es ese valor, mayor es el tamaño reservado en caché para cada uno de los campos del array compartido que emplean los hilos para sumar las áreas de los rectángulos, de modo que es menos probable que tengan que acceder a un mismo área del array por lo que se dan menos casos en los que tengan que esperar a que otro hilo termine de modificar el array para poder acceder ellos (false sharing), es decir, hay menor competencia por los recursos haciendo que el programa tarde menos en su ejecución.

Ejercicio 5

En este último ejercicio de la práctica se pide que nos fijemos en las 3 últimas versiones del programa que obtiene una aproximación de pi por integración numérica. Estas 3 últimas versiones son pi_par5, pi_par6 y pi_par7.

5.1

Si comparamos el programa pi_par5 con pi_par4 vemos que la principal diferencia es que pi_par5 emplea la directiva de openMP “critical”. Al usar esta directiva lo que se consigue es que ese fragmento de código solo lo pueda ejecutar 1 hilo a la vez de modo que no haya varios hilos que modifiquen a la vez el valor pi pues si lo hiciesen la suma acabaría siendo errónea, sin embargo esto hace que se tengan que esperarse unos hilos a otros haciendo que la ejecución tarde más, como ocurre en pi_par5, mientras que en pi_par4 tarda mucho menos pues los hilos no necesitan ir esperando a entrar en ninguna sección del código. Sin embargo lo que más destaca al ejecutar pi_par5 es que esta versión da resultados incorrectos y diferentes cada vez, esto se debe a que el parámetro i que utiliza cada uno de los hilos para calcular su parte de pi no es privado, y por tanto cuando uno lo modifica, se modifica en todos los hilos, haciendo que haya iteraciones de los bucles de los distintos hilos que no se ejecuten y por tanto acabando en un resultado erróneo.

5.2

Por último, al ejecutar las 2 últimas versiones pi_par6 y pi_par7 observamos que existe una gran diferencia de rendimiento entre ellas similar a la ya explicada entre pi_par1 y pi_par4. En este caso pi_par6 es el que tarda un tiempo superior mientras que pi_par7 tarda mucho menos en realizar el cálculo de pi. Esto se debe a que en pi_par6 cada uno de los hilos debe ir modificando un array compartido casi constantemente (como en pi_par1) por lo que se produce un cuello de botella ya que tan solo 1 hilo puede acceder al array a la vez, mientras que pi_par7 no emplea este array compartido de modo que no tiene este cuello de botella. En estas versiones además observamos 2 directivas que no habíamos visto en las anteriores. En primero lugar en pi_par6 vemos "omp for" que se emplea para que el bucle for que aparece justo después se ejecute con contadores "i" independientes a pesar de que la i no está declarada como variable privada. La otra directiva aparece en pi_par7 y es "omp parallel for reduction(+: sum)" Esta directiva sirve

para que el bucle for al que se refiere se realice de forma paralela por varios hilos y para que cada uno emplee una variable privada "sum" para ello y, tras terminar la ejecución todos los hilos, se unan todas estas variables "sum" privadas en una sola con la operación especificada en el reduction, que, en este caso es la suma; por lo tanto, tras la ejecución de todos los hilos se tendría una única variable "sum" cuyo valor es el de la suma de todas las variables "sum" privadas de cada uno de los hilos.