EJERCICIO 0 :

Para obtener la información del cluster hemos usado el comando cat /proc/cpuinfo:

```
[arqo49@labomat36 materialP4]$ clear

[arqo49@labomat36 materialP4]$  cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 16
model           : 9
model name      : AMD Opteron(tm) Processor 6128
stepping        : 1
cpu MHz         : 800.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 8
apicid          : 16
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext f
xsr_opt pdpe1gb rdtscp lm 3dnowext 3dnow constant_tsc rep_good nonstop_tsc extd_apicid amd_dcm pni monitor cx16 popcnt lahf_lm cmp_leg
acy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt nodeid_msr npt lbrv svm_lock nrip_save pausefilter
bogomips        : 4000.50
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate
```

De aquí se extrae que el cluster tiene dos procesadores de 8 cores. El hyperthreading no está activado,asi que estos cores son todos físicos, y se pueden lanzar un total de 16 hilos

La frecuencia de cada procesador es de 800 Mhz.

EJERCICIO 1 :

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

Si se pueden lanzar más hilos que cores, y tiene sentido hacerlo siempre que la tarea a paralelizar no demande mucho procesador, ya que en ese caso, si se lanzan muchos hilos, se puede saturar el sistema y no se alcanzará el máximo rendimiento.

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

Al ejecutar el comando “cat /proc/cpuinfo” podemos observar que los ordenadores de los laboratorios tienen 4 cores físicos y un total de 4 hilos, luego no tienen hyperthreading. Lo ideal para ellos, entonces sería 4 hilos.

En el cluster, como hemos comentado en el ejercicio 0, tiene 2 procesadores con 8 cores físicos cada uno, luego lo ideal sería utilizar 16 hilos. Finalmente, en nuestro equipo , hay 2 cores físicos, y un total de 4 hilos, por lo que tiene hyperthreading. Lo ideal seria por lo tanto utilizar 4 hilos.

Al ejecutar ./omp2 en el cluster nos ha salido lo siguiente:

```
[arqo49@labomat36 materialP4]$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7fff459faeec,x   &b = 0x7fff459faee8,    &c = 0x7fff459faee4

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7fff459faea4,    &b = 0x7fff459faee8,    &c = 0x7fff459faea8
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 1]-1: a = 32687,   b = 2,  c = 3
[Hilo 1]         &a = 0x7faf54a37e04,    &b = 0x7fff459faee8,    &c = 0x7faf54a37e08
[Hilo 1]-2: a = 1068439990,      b = 6,  c = 1068439972
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7faf53635e04,    &b = 0x7fff459faee8,    &c = 0x7faf53635e08
[Hilo 3]-2: a = 27,      b = 8,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7faf54036e04,    &b = 0x7fff459faee8,    &c = 0x7faf54036e08
[Hilo 2]-2: a = 33,      b = 10,         c = 3

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7fff459faeec,    &b = 0x7fff459faee8,    &c = 0x7fff459faee4
```

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

Las variables declaradas como private, tienen un valor propio para cada thread sin importar si estaba inicializada en el thread master, de modo que cada copia tendrá un valor distinto en cada hilo. Por otro lado, las variables firstprivate sí que cogen el valor inicial del thread master.

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

Independientemente de si estaba inicilizada o no una variable antes de comenzar la sentencia OpenMP, al empezar dicha sentencia, las variables declaradas como private son inicializadas a un valor, uno por cada hilo. Por ejemplo, al ejecutar ./omp2, la variable “a” es privada, en el thread master, tiene un valor de 1, mientras que en los hilos el valor cambia y es independiente uno de otros (en la mayoría de hilos se asigna el valor 0 y en el hilo 1 se le asigna un valor por encima de 30.000).

En segundo lugar, las variables declaradas como firstprivate, como la “c”, sí que son inicializadas dentro del hilo tomando el valor del thread master. Por ejemplo, la “c” se inicializa en todos los hilos a 3.

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

El valor de las variables privadas, una vez ha finalizado la sentencia paralela, se destruye y no influye al valor de la variable en el thread master, el cual sigue siendo el inicial. Por ejemplo, en el programa de la captura, se puede observar que tanto al principio como al final, “a” y “c” mantienen los valores “1” y “3”.

1.6 ¿Ocurre lo mismo con las variables públicas?

No, las variables públicas se comparten entre el thread master y los hilos, por lo tnto las modificaciones que se aplican a estas variables dentro de los hilos, afectan al valor final. Por ejemplo, en la captura de pantalla, “b” comienza valiendo “2” y termina valiendo “10”.

EJERCICIO 2 :

Al ejecutar los programas nos quedan estos resultados:

```
[arqo49@labomat36 materialP4]$ ./pescalar_serie
Resultado: 33.319767
Tiempo: 0.009939
[arqo49@labomat36 materialP4]$ ./pescalar_par1
Resultado: 4.554927
Tiempo: 0.036305
[arqo49@labomat36 materialP4]$ ./pescalar_par2
Resultado: 33.330593
Tiempo: 0.002549
```

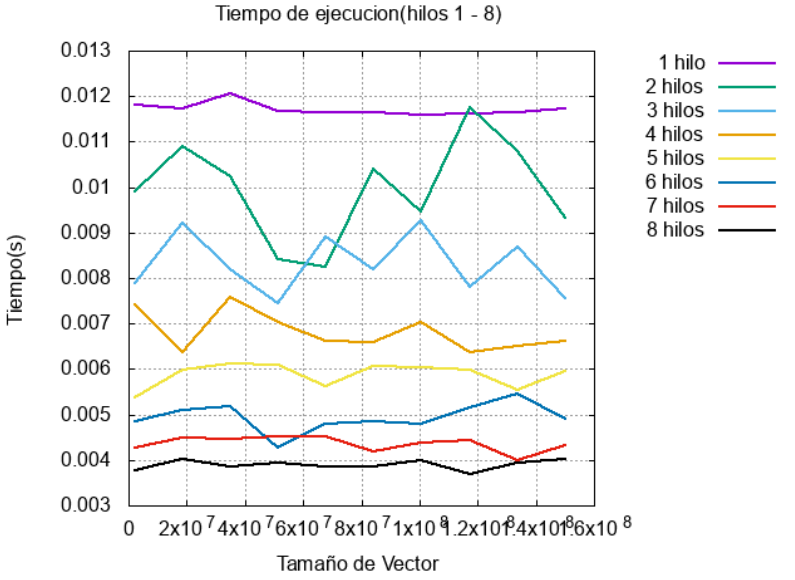
2.1 ¿En qué caso es correcto el resultado?

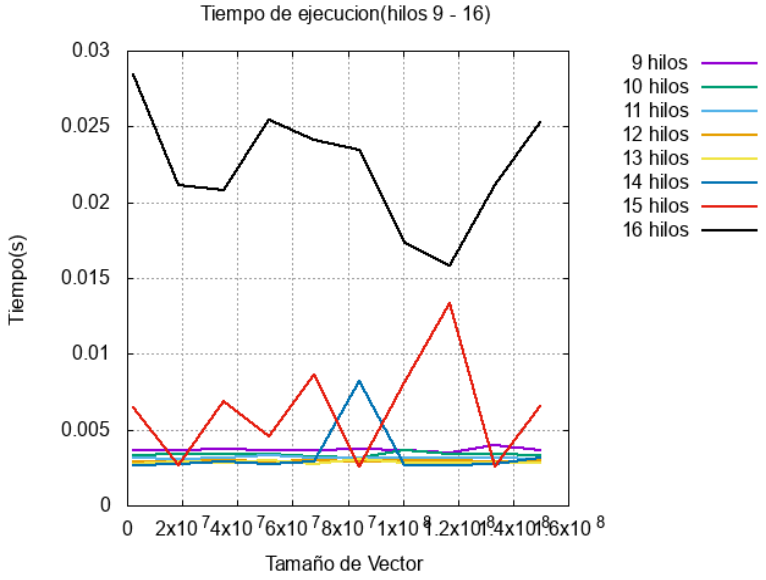
El resultado correcto es el que da pescalar_par2.c.

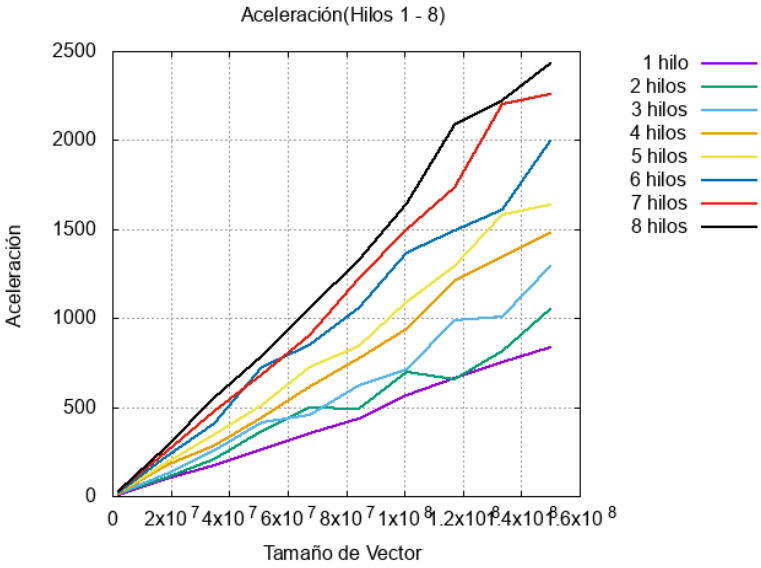
2.2 ¿A qué se debe esta diferencia?

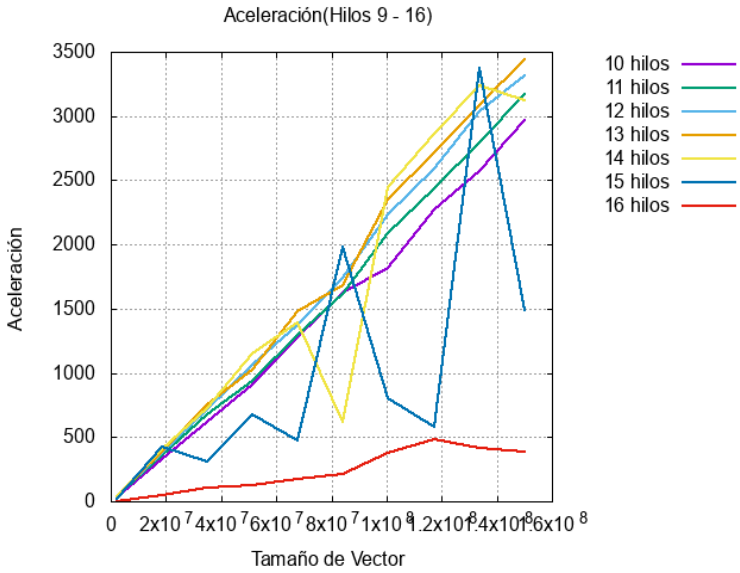
La diferencia se debe a las sentencias de OpenMP, en el programa pescalar_par1.c podemos observar que se utiliza la sentencia "#pragma omp parallel for" de modo que repartimos el trabajo de calcular el producto escalar entre los hilos en un orden aleatorio, de modo que se pueden ejecutar a la vez una lectura y una escritura de la variable "sum" y no ir acumulando correctamente el valor final. Sin embargo, el programa pescalar_par2.c usa la sentencia "#pragma omp parallel for reduction(+:sum)", la cual regulariza los hilos de manera que se va acumulando correctamente el valor de sum, pues sincroniza los hilos para no realizar a la vez una lectura y una escritura.

Seguidamente, hemos modificado pescalar_serie. C para que acepte como argumento de entrada el tamaño de los vectores a multiplicar, mientras que pescalar_par2.c ahora necesita dos argumentos de entrada, el tamaño de los vectores y el número de hilos. Además, hemos creado un script de bash llamado *ejercicio2.sh*, el cual ejecuta el proceso pedido en el enunciado, almacenando los tiempos de ejecución de los programas pescalar_serie y pescalar_par2 , variando uniformemente el tamaño de los vectores entre 2000000(tarda 0.1 segundos) y 150000000(tarda alrededor de 10 segundos). Para cada tamaño, ejecutamos la prueba 5 veces y el tiempo será la media de lo obtenido. Y además, para el caso paralelo, ejecutaremos todas estas mediciones variando el numero de hilos de 1 a 16. Los resultados de tiempo y aceleración(tserie/tparalelo)se ven reflejados en las siguientes gráficas(graficadas con el script *graficar2.sh*):









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

Estudiando los resultados obtenidos y observados en las gráficas, concluimos que compensa lanzar hilos,ya que se puede ver que los tiempos de ejecución disminuyen y la aceleración aumenta cuantos más hilos se usan, excepto en el caso de 14,15 y 16 hilos, donde los tiempos empeoran ligeramente respecto a los resultados justo anteriores.

2.5 ¿Se mejora siempre el rendimiento al aumentar el número de hilos a trabajar?
2.6 Si no fuera así, ¿a qué debe este efecto?

A raiz de nuestro estudio, no podemos concluir que siempre se mejore el rendimiento cuantos más hilos se usen, puesto que para procesos cortos, lanzar tantos hilos seguramente no compense y no produzca una mejora. Ya hablando de nuestros resultados, podemos concluir que cuantos más hilos, mejor rendimiento, mirando nuevamente lo presentado en las gráficas, y destacando de nuevo esa pequeña bajada del rendimiento cuando se usan 14, 15 y 16 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.

Analizando los resultados obtenidos, y teniendo en cuenta que para matrices no mucho más grandes de las utilizadas en nuestro estudio, el sistema se desborda, podemos aventurar que no existe un tamaño tal que la aceleración varíe en gran medida. Las matrices utilizadas en nuestra prueba son de un tamaño bastante grande, y no se aprecian variaciones destacables entre las distintas gráficas(una vez más, obviando los resultados de 14, 15 y 16 hilos, que reducen ligeramente el rendimiento).

EJERCICIO 3 :

Para este ejercicio, primeramente hemos desarrollado una script(tablas.sh) que ejecuta las pruebas pedidas para rellenar las tablas. Los resultados se ven reflejados en las mismas :

Tamaño = 1000

Tiempo de ejecución(s)

Versión\ # hilos	1	2	3	4
Serie	13.5162	-	-	-
Paralela – bucle 1	15.0394	9.5906	7.4691	6.8575
Paralela – bucle 2	13.4231	9.8820	4.4725	3.3549
Paralela – bucle 3	13.1447	6.6060	4.4109	3.3273

Speedup (tomando como referencia la versión serie)

Versión\ # hilos	1	2	3	4
Serie	1	-	-	-
Paralela – bucle 1	0.8987	1.4093	1.8096	1.9710
Paralela – bucle 2	1.0069	1.3677	3.0220	4.0287
Paralela – bucle 3	1.0282	2.0460	3.0642	4.0622

Tamaño = 2000

Tiempo de ejecución(s)

Versión\ # hilos	1	2	3	4
Serie	119.8862	-	-	-
Paralela – bucle 1	136.2030	73.579	52.7268	45.8836
Paralela – bucle 2	123.73	64.1421	40.0433	28.3069
Paralela – bucle 3	117.2340	62.6077	43.2442	26.3108

Speedup (tomando como referencia la versión serie)

Versión\ # hilos	1	2	3	4
Serie	1	-	-	-
Paralela – bucle 1	0.8802	1.6293	2.2737	2.6128
Paralela – bucle 2	0.9689	1.869	2.9939	4.2352
Paralela – bucle 3	1.0226	1.9148	2.7723	4.5565

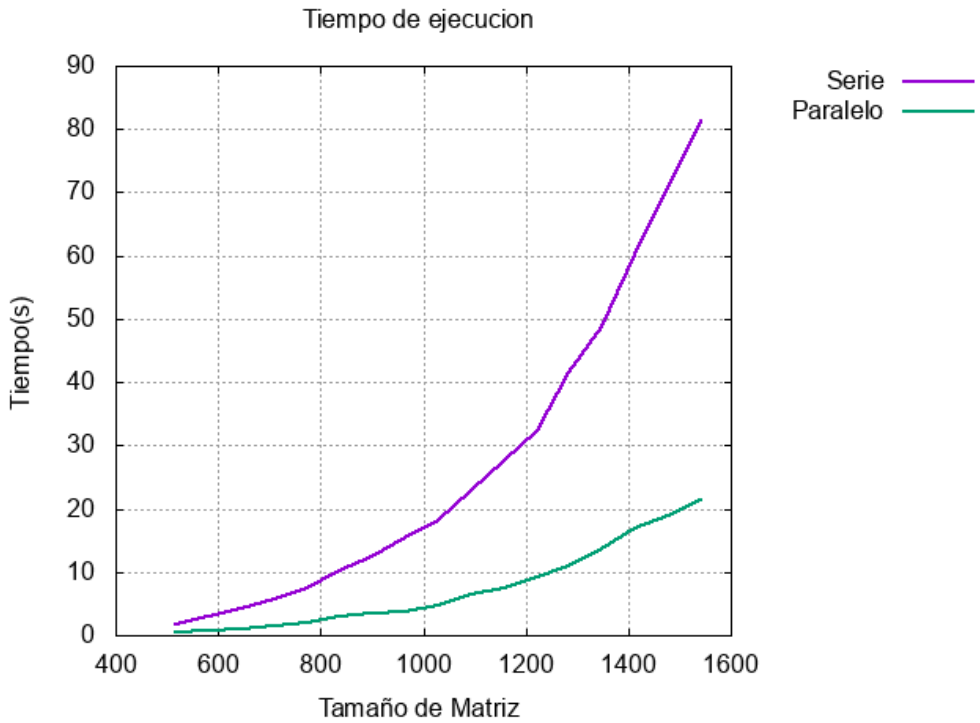
3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe?

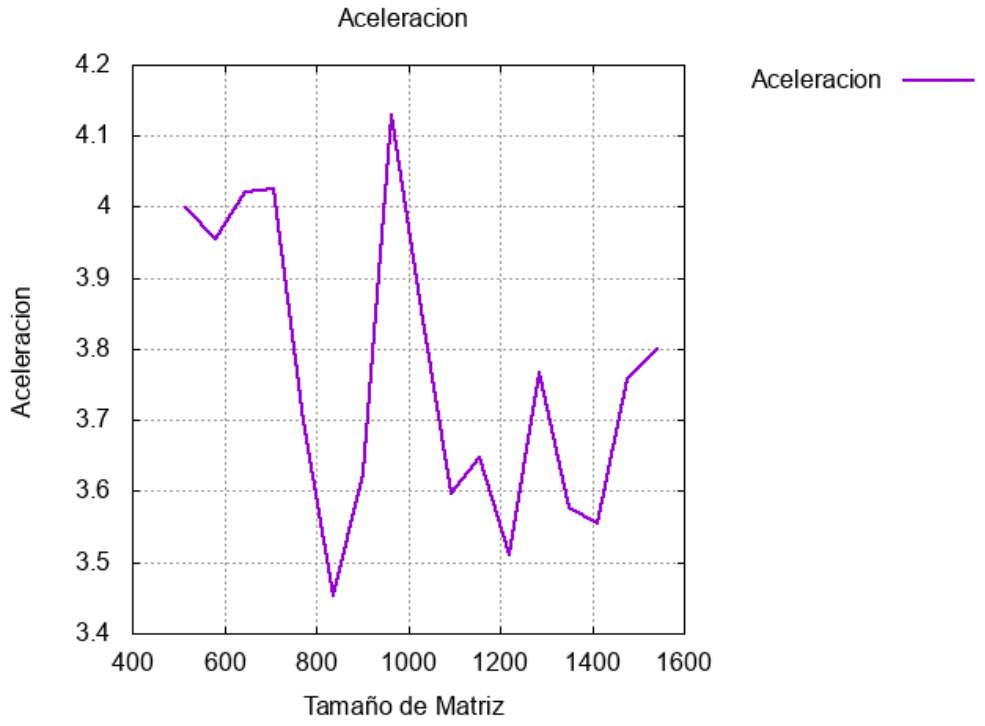
En las tablas se puede observar que la peor versión es la de 1 hilo , paralelizando el bucle más interno. Esto se debe a que estamos paralelizando un único bucle, y al ser el más interno, la creación del hilo que se usa para su paralelización se repite tantas veces como iteraciones hacen los bucles más externos, de manera que se pierde mucho tiempo en el proceso.

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

La mejor versión es aquela en la que paralelizamos el bucle más externo, haciendo uso de 4 hilos. Esto se debe a que estamos paralelizando una zona en la que se van a realizar muchas operaciones, y lo hacemos con un numero de hilos suficientemente grande como para que el reparto de tareas entre hilos alcanze una eficiencia que le hace mejor que el resto de casos.

A continuación, hemos creado un script(*ejercicio3.sh*) que realiza la prueba descrita en el enunciado , con la versión en serie de la multiplicación y la versión de 4 hilos paralelizando el bucle más externo, para tamaños de matriz entre 517 y 1541, con saltos de 64 unidades, y realizando la prueba 5 veces para cada tamaño. Los resultadosse ven reflejados en las siguientes gráficas :





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.

Se puede observar en la gráfica que, a pesar de los picos, la aceleración va disminuyendo. Debido al escaso número de tamaños analizados, no se puede observar con claridad donde se estabilizaría la gráfica o donde empieza a disminuir pronunciadamente la aceleración, pero como la mayor diferencia (el mayor pico) se observa alrededor del 1000, podemos concluir que la tendencia cambia cerca de tal valor.

EJERCICIO 4 :

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

En pi_serie.c se puede observar que el número de rectángulos utilizados en 100000000, es decir, el número de iteraciones del bucle.

4.2: ¿Qué diferencias observa entre estas dos versiones?

En cuanto al código, podemos observar que en el programa *pi_par1.c* , los hilos van acomulando sus resultaods en un array compartido, mientras que el programa *pi_par4.c* primero lo guarda en una variable privada y después del bucle lo almacena en el array compartido.

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?

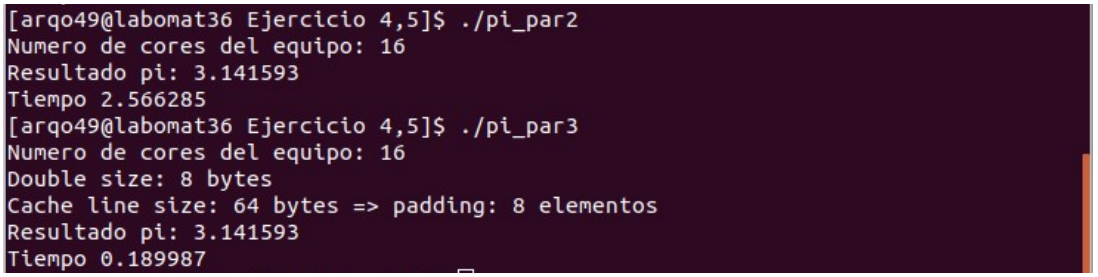
Ejecutamos ambos programas y obtenemos la siguiente salida :

```
[arqo49@labomat36 Ejercicio 4,5]$ ./pi_par1
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 2.102621
[arqo49@labomat36 Ejercicio 4,5]$ ./pi_par4
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.180019
```

En ambas se obtiene el mismo resultado para pi, mientras que en cuanto al rendimiento, se observa una diferencia considerable entre pi_par1 y pi_par4, donde el primero tarda bastante más que el segundo. Esto se debe a una constante reescritura del array compartido, que provocará sucesivos fallos en las cachés de los cores, que tendrán que acceder cada vez que falle a memoria para actualizar las cachés, provocando una pérdida de tiempo bastante consistente. Por ello, se puede observar esta diferencia de tiempo, ya que *pi_par1* escribe varias veces por hilo en el array, y *pi_par4* solamente una vez por hilo.

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?

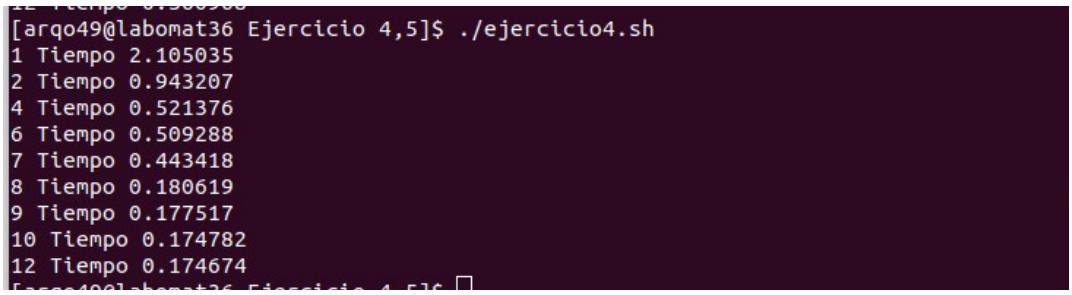
Ejecutamos y obtenemos lo siguiente:



El programa *pi_par2* declara la variable compartida sum como privada, pero esto no supone una mejora en el rendimiento, ya que se sigue teniendo el problema expuesto en el apartado anterior. Sin embargo, el programa *pi_par3* si evita este problema, como se puede observar al ver su rendimiento. Lo evita redimensionando el array compartido sum para que tenga un tamaaño tal que se eviten los fallos en la caché, y por lo tanto, se evite la consecuente pérdida de rendimiento.

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?

Hemos creado el script *ejercicio4.sh* para ejecutarlo y nos sale lo siguiente:



Tomando el valor 1 se obtiene un resultado parecido a *pi_par1*, esto se debe a que no estamos fragmentando el array sino que lo multiplicamos por 1, igual que en el programa indicado, luego no obtenemos ninguna mejoría. A medida que vamos aumentando el tamaño de la fragmentación, se va obteniendo una mejoría pues cada vez es mayor la fragmentación del bloque, y así es mejor el rendimiento. Este alineamiento de datos permite evitar el problema de *pi_par1*.

EJERCICIO 5:

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?

Ejecutamos ambos programas y nos queda:

```
[arqo49@labomat36 Ejercicio 4,5]$ ./pi_par4
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.237259
[arqo49@labomat36 Ejercicio 4,5]$ ./pi_par5
Resultado pi: 2.728239
Tiempo 5.117659
```

Podemos observar que *pi_par4* soluciona el problema de *pi_par1* utilizando variables privadas, obteniendo así una mejora en el rendimiento. Por otro lado, el programa *pi_par5* empeora el rendimiento pues al crear una sección crítica se pierde tiempo en la espera de los hilos.

Podemos ver también que en *pi_par5* el resultado de pi es incorrecto, esto se debe a que la variable *i* del bucle es pública, luego cada bucle alterará el valor de la variable donde se guarda el resultado.

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

Ejecutamos ambos programas y nos sale:

```
[arqo49@labomat36 Ejercicio 4,5]$ ./pi_par6
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 2.130963
[arqo49@labomat36 Ejercicio 4,5]$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.172558
```

En el programa *pi_par6* no se obtiene ninguna mejora pues paraleliza el bucle que calcula pi sin declarar *reduction*, luego no está bien paralelizado y por ello tarda más que *pi_par7,* el cual si que incluye el *reduction*, la cual, comentada en el primer ejercicio, evita el problema del *false sharing.*