# ARQUITECTURA DE ORDENADORES

# Práctica 4

# Explotar el Potencial de las Arquitecturas Modernas

Jorge Santisteban Rivas
Javier Martínez Rubio
Curso 2018-2019

## Ejercicio 0: Información sobre la topología del sistema

Tras ejecutar el comando cat /proc/cpuinfo y observar su salida, vemos que el equipo dispone de 4 cpu cores (físicas) y 8 siblings (virtuales). Por lo tanto, podemos deducir que hay hyperthreading, ya que 8/4=2.

## Ejercicio 1: Programas básicos de OpenMP

Ejecutamos el ejercicio omp1.c, y respondemos a las siguientes preguntas:

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

Sí que se pueden lanzar más threads que cores tiene el sistema, y además, al haber hyperthreading, tiene sentido hacerlo. Esto se debe a que, de esta manera, los threads accederán a los recursos del sistema de forma compartida, por lo que el consumo de recursos será inferior.

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

Los ordenadores del laboratorio, que es donde hemos realizado toda la práctica, deberían utilizar 8 threads, ya que es el número total de cpus virtuales del que disponen.

Tras ejecutar el programa omp2.c, obtenemos la siguiente salida:

```
e360104@16-17-64-239:~/UnidadH/ARQO P4/materialP4/materialP4$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7ffe41108914,x   &b = 0x7ffe41108918,    &c = 0x7ffe4110891c

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7ffe411088b0,    &b = 0x7ffe41108918,    &c = 0x7ffe411088ac
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7fb1886dde20,    &b = 0x7ffe41108918,    &c = 0x7fb1886dde1c
[Hilo 2]-2: a = 21,      b = 6,  c = 3
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7fb187edce20,    &b = 0x7ffe41108918,    &c = 0x7fb187edce1c
[Hilo 3]-2: a = 27,      b = 8,  c = 3
[Hilo 1]-1: a = 0,       b = 2,  c = 3
[Hilo 1]         &a = 0x7fb188edee20,    &b = 0x7ffe41108918,    &c = 0x7fb188edee1c
[Hilo 1]-2: a = 33,      b = 10,         c = 3

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7ffe41108914,    &b = 0x7ffe41108918,    &c = 0x7ffe4110891c
```

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

Al declarar una variable privada, esta será usada por cada hilo como una variable temporal, ya que se copia en el almacenamiento local de cada hilo.
En este programa se declaran como privadas dos variables, y cada hilo tendrá su propio valor de estas variables, independientemente del valor que tengan en los demás hilos. También se observa en las direcciones, ya que en cada hilo las direcciones de estas variables son distintas a las del resto de hilos.
También podemos observar la declaración firsprivate de la variable c. Esta variable se comporta de manera similar a las privadas, con la diferencia de que en este caso la variable se inicializa para todos los hilos con el valor que tenga en el momento de la declaración. Se ve que para todos los hilos esta variable comienza con el valor 3 y a partir de ahí cada hilo le acaba dando su propio valor.

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

Las variables privadas, como no se inicializan, no tomarán el valor asignado previo a su declaración como privada cuando empiece a ejecutarse la región paralela. Esto lo observamos con la variable a, que se inicializa con el valor 1, pero sin embargo toma el valor 0 en cada hilo al empezar a ejecutarse.

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

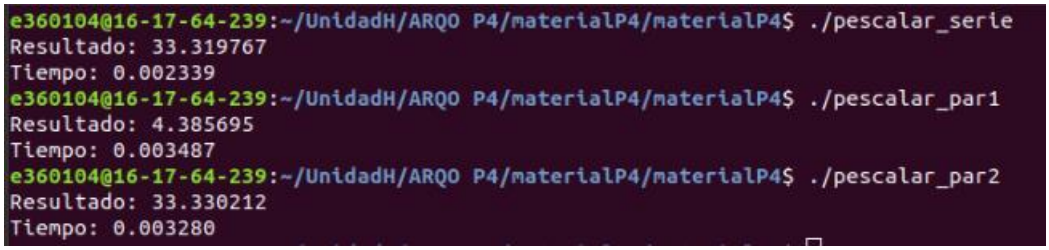
Cuando finaliza la región paralela, estas variables no conservan los cambios que habían tenido durante este periodo paralelo, si no que mantienen el valor con el que habían sido inicializadas antes de entrar en la región paralela.

**1.6 ¿Ocurre lo mismo con las variables públicas?**

Al contrario que con las variables privadas, las variables públicas comparten los datos son accesibles por todos los hilos. En este caso, la b es una variable privada, y podemos ver como se mantiene la misma dirección tanto dentro como fuera de los hilos. Además, las modificaciones en los hilos del valor de esta variable se mantienen, por lo que al finalizar el programa, su valor será el que tenía en la última modificación en la región paralela.

# Ejercicio 2: Paralelizar el producto escalar

Ejecutamos los programas pescalar_serie, pescalar_par1 y pescalar_par2, y obtenemos la siguiente salida:



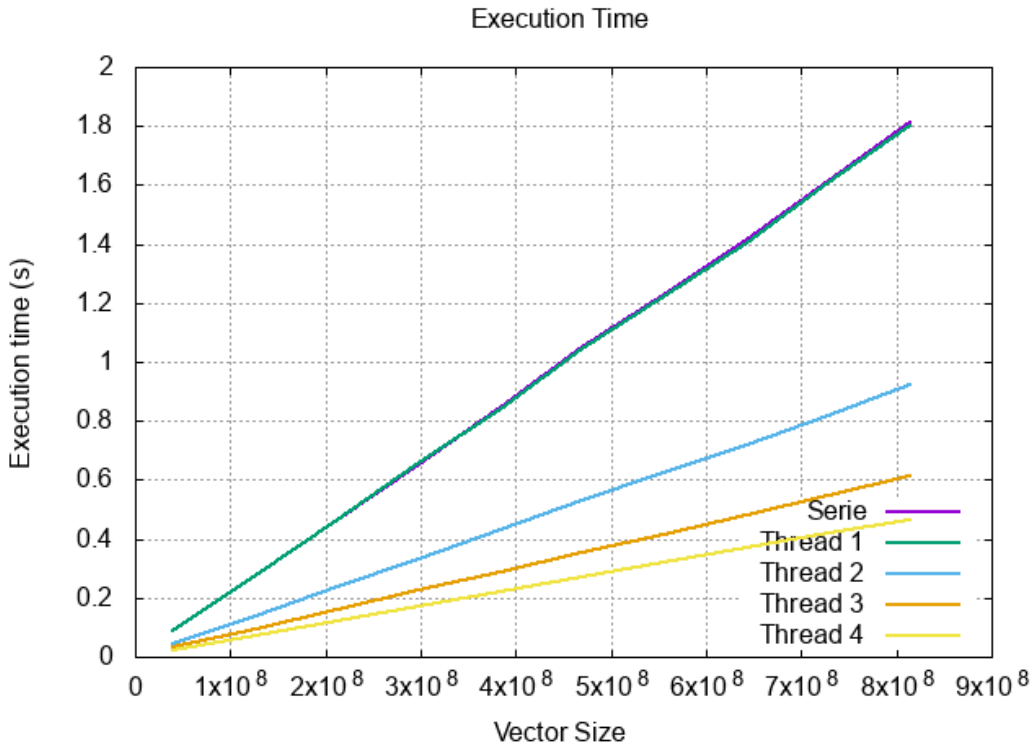

**2.1 ¿En qué caso es correcto el resultado?**

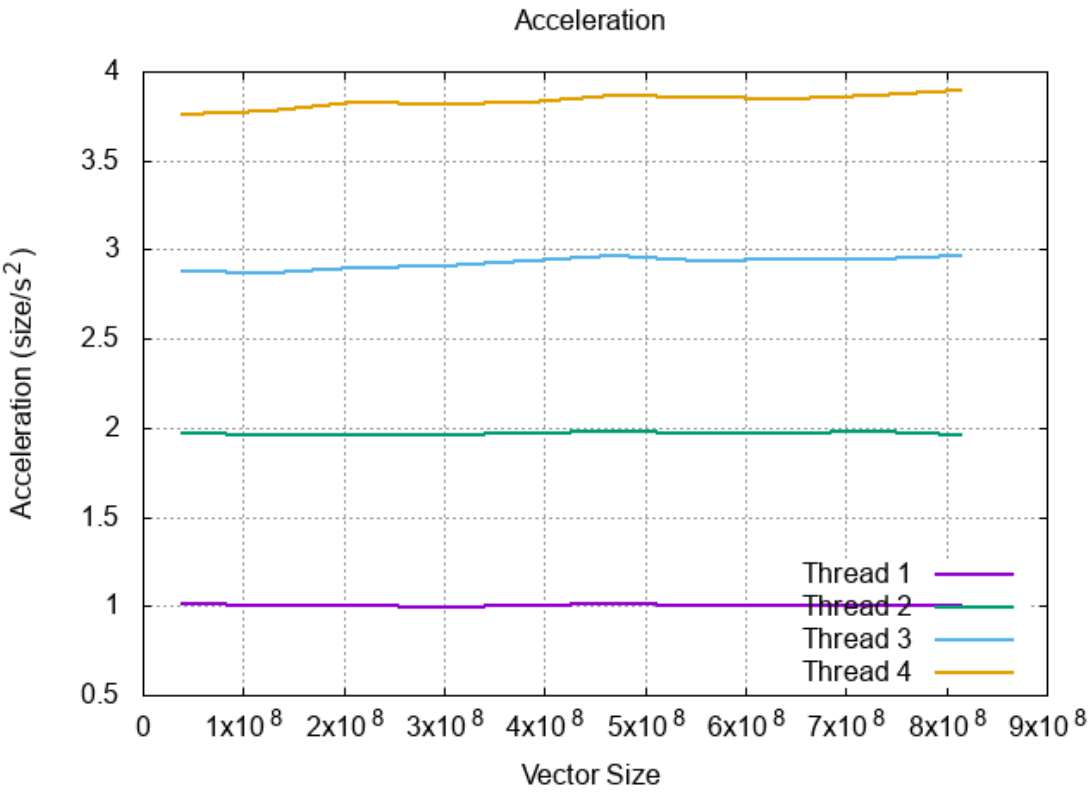
El resultado es correcto en los programas pescalar_serie, que no tiene paralelización y pescalar_par2, que lo paraleliza utilizando la cláusula reduction.

**2.2 ¿A qué se debe esta diferencia?**

En el programa pescalar_par2, que es correcto, cada hilo crea la variable sum, que será combinada y se asignará a la variable original global al final del bloque de hilo.
Por su parte, el programa pescalar_par1, al no utilizar la cláusula reduction, no realiza de forma adecuada la paralelización, ya que la variable sum toma un valor distinto para cada hilo, y finaliza con el valor que dejó el último hilo que realizó la ejecución.

Hemos decidido ejecutar los programas comenzando con una dimensión de vectores de 40000000 y finalizando con 900000000. Establecemos un rango a la hora de tomar medidas del tiempo de ejecución y aceleración para los dos programas que son correctos y obtenemos las siguientes gráficas:


Execution Time
Execution time (s)
Vector Size
Serie
Thread 1
Thread 2
Thread 3
Thread 4
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
1x10^8
2x10^8
3x10^8
4x10^8
5x10^8
6x10^8
7x10^8
8x10^8
9x10^8


Acceleration
Acceleration (size/s^2)
Vector Size
Thread 1
Thread 2
Thread 3
Thread 4
0.5
1
1.5
2
2.5
3
3.5
4
0
1x10^8
2x10^8
3x10^8
4x10^8
5x10^8
6x10^8
7x10^8
8x10^8
9x10^8

**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?**

Siempre que la aceleración esté por encima del 1, el programa será más rápido que en serie, por lo tanto compensará. En este caso, vemos como lanzar el programa pescalar_par2 con un solo hilo es prácticamente igual que lanzar el de serie. Sin embargo, a medida que aumentamos los hilos, la mejora se hace muy significativa.

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

No compensa para tamaños muy pequeños, ya que se tarda tiempo en lanzar los hilos y gestionarlos, mientras que en serie ese tiempo no es necesario y se empieza antes a realizar cálculos.

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

El rendimiento mejorará siempre y cuando el equipo disponga de los suficientes núcleos para soportar los hilos lanzados. En nuestro caso, como disponemos de hasta 8 , podríamos lanzar hasta 8 hilos y aumentaría el rendimiento. Sin embargo, si lanzáramos más, solo podrían ejecutarse simultáneamente esos 8, con lo que el rendimiento disminuiría.

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

Por lo comentado en el apartado anterior, el ordenador puede soportar tantos hilos como núcleos totales tenga, por lo que a partir de ahí, aumentar el número de hilos supone perder rendimiento ya que el ordenador no los puede gestionar de la manera más eficiente.

**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.**

Observando la gráfica vemos como la aceleración se mantiene constante para todos los tamaños. Esto nos hace pensar que, aunque aumentemos el tamaño, la aceleración seguirá constante. Sin embargo, por lo comentado en el apartado 2.4, si cogemos tamaños muy pequeños, veríamos una aceleración por debajo del 1, ya que se perdería rendimiento utilizando hilos.

## Ejercicio 3: Paralelizar la multiplicación de matrices

Tras ejecutar las tres versiones paralelas que realizan la multiplicación de matrices, obtenemos los siguientes resultados para dos tamaños distintos, 1000 y 2050:

Tamaño 1000x1000:

| Tiempos de ejecucion(s) | | | | |
|---|---|---|---|---|
| Version\# hilos | 1 | 2 | 3 | 4 |
| Serie | 4,8298 | | | |
| Paralela-Bucle 1 | 5,17534 | 3,19226 | 2,38033 | 2,02957 |
| Paralela-Bucle 2 | 4,8278 | 2,51109 | 1,64338 | 1,26528 |
| Paralela-Bucle 3 | 4,82079 | 2,45607 | 1,64557 | 1,26279 |

| Speedup | | | | |
|---|---|---|---|---|
| Version\# hilos | 1 | 2 | 3 | 4 |
| Serie | 1 | | | |
| Paralela-Bucle 1 | 0,93323 | 1,51297 | 2,02905 | 2,37972 |
| Paralela-Bucle 2 | 1,00041 | 1,92339 | 2,93895 | 3,81719 |
| Paralela-Bucle 3 | 1,00187 | 1,96648 | 2,93504 | 3,82472 |

Tamaño 2050x2050

| Tiempos de ejecucion(s) | | | | |
|---|---|---|---|---|
| Version\# hilos | 1 | 2 | 3 | 4 |
| Serie | 57,9352 | | | |
| Paralela-Bucle 1 | 59,8894 | 29,657 | 20,23 | 16,4731 |
| Paralela-Bucle 2 | 57,8729 | 29,3898 | 19,9465 | 14,9725 |
| Paralela-Bucle 3 | 57,8047 | 29,4978 | 20,2511 | 14,9953 |

| Speedup | | | | |
|---|---|---|---|---|
| Version\# hilos | 1 | 2 | 3 | 4 |
| Serie | 1 | | | |
| Paralela-Bucle 1 | 0,96737 | 1,95351 | 2,86382 | 3,51696 |
| Paralela-Bucle 2 | 1,00108 | 1,97126 | 2,90452 | 3,86944 |
| Paralela-Bucle 3 | 1,00226 | 1,96405 | 2,86085 | 3,86356 |

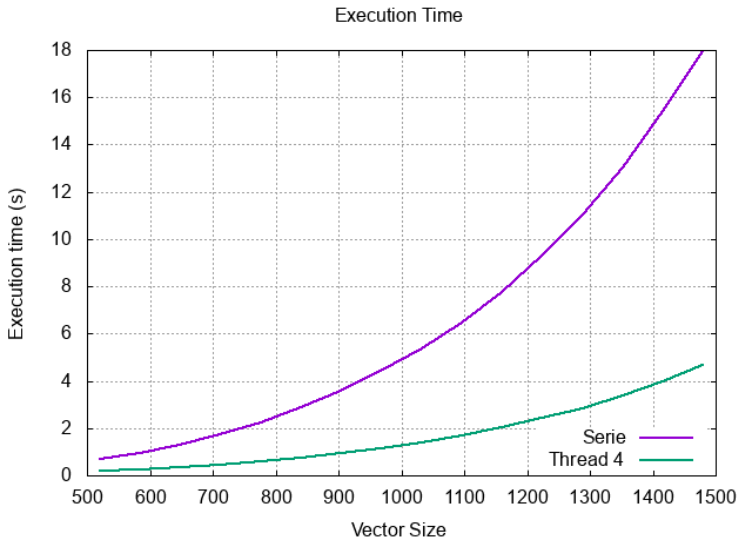
**3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe?**

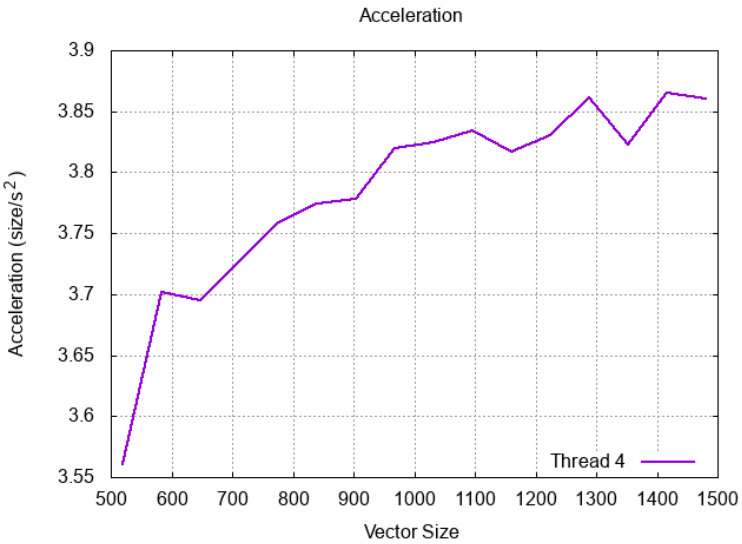
El peor rendimiento se obtiene paralelizando el bucle 1, es decir, el más interno. Esto se debe a que se lanzan más hilos de los que realmente se quieren y no se gestionan bien. Esto supone un gasto innecesario de recursos, que deriva en una falta de eficacia.

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

Aunque el rendimiento obtenido paralelizando los bucles externo e intermedio es muy parecido, sí que se aprecia una cierta mejora en el programa par3. Por lo tanto, se obtiene el mejor rendimiento paralelizando el bucle externo. Esto se debe a que paralelizar este bucle supone agrupar n tareas adaptándolas a los cores disponibles, con lo que los hilos estarán mucho más controlados que si se trata de las n*n*n tareas que hay que agrupar paralelizando el bucle más interno.

Obtenemos las gráficas comparando los programas serie y par3:




Observamos como el tiempo de ejecución del programa en serie crece mucho más rápido que el  par3 con 4 hilos, que lo hace de una manera mucho más suave. En cuanto a la aceleración, se aprecia cómo se estabiliza en un valor cercano a 4 para valores a partir de un tamaño de aproximadamente 1400.

# Ejercicio 4: Ejemplo de integración numérica

Ejecutamos la versión en serie:

```
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio4$ ./pi_serie
Resultado pi: 3.141593
Tiempo 1.037788
```

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

Se utilizan 100.000.000 rectángulos.

**4.2: ¿Qué diferencias observa entre las versiones par1 y par4?**

Podemos ver en el código que la versión pi_par4, cada hilo utiliza una variable privada priv_sum, de manera que esta guarda el valor de cada hilo y luego estos se pasan a la variable común. Por otra parte, en el pi_par1 el resultado se calcula directamente sobre la variable compartida.

**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?**

```
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio4$ ./pi_par1
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.330968
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio4$ ./pi_par4
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.151253
```

Para ambos programas vemos como el resultado final obtenido es exactamente el mismo, con los mismos decimales, pero sin embargo el programa pi_par4 emplea un tiempo mucho menor, por lo que la aceleración de este es mayor que para pi_par1. Esto se debe a que el pi_par4, el tener una única variable sum, esta sólo se modifica una vez cuando se escribe en un elemento del array. Sin embargo, el pi_par1, como hace la modificación dentro de los distintos hilos, es necesario modificarla en todos ellos, con lo que se consume más tiempo.

**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?**

```
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio4$ ./pi_par2
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.323991
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio4$ ./pi_par3
Numero de cores del equipo: 8
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.153241
```

Se obtiene más o menos el resultado esperado, ya que el pi_par2 es más lento que el pi_par3, y ambos se comportan de una manera muy parecida a pi_par1 y pi_par4. En pi_par3 se ha aumentado el tamaño del array donde se guardan los resultados de cada hilo. Sin embargo, se sigue necesitando recargar los datos en cada core por cada modificación de la variable suma.

**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?**

| | Tiempo de ejecucion | Aceleracion |
|---|---|---|
| serie | 1,037788 | |
| pi_par3_1 | 0,3408708 | 3,044520094 |
| pi_par3_2 | 0,279953 | 3,707007962 |
| pi_par3_4 | 0,187538 | 5,533747827 |
| pi_par3_6 | 0,175923 | 5,899103585 |
| pi_par3_7 | 0,172359 | 6,0210839 |
| pi_par3_8 | 0,140744 | 7,373586085 |
| pi_par3_9 | 0,138804 | 7,476643324 |
| pi_par3_10 | 0,14027 | 7,398502887 |
| pi_par3_12 | 0,142758 | 7,269561075 |

Vemos cómo va aumentando el rendimiento,  y se estabiliza a partir de pi_par8 en una aceleración de entre 7 y 7,5. El programa que menos tiempo emplea es el pi_par9, por lo que también es el que tiene el mejor rendimiento.

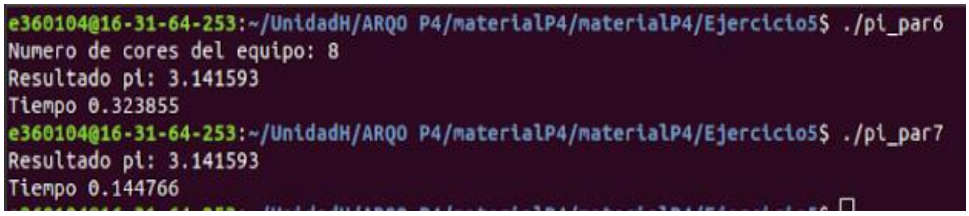
## Ejercicio 5: Uso de la directiva critical y reduction

**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?**

```
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio5$ ./pi_par4
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.144299
e360104@16-31-64-253:~/UnidadH/ARQO P4/materialP4/materialP4/Ejercicio5$ ./pi_par5
Resultado pi: 3.299086
Tiempo 0.426064
```

Cabde destacar que la versión pi_par5 utiliza la directiva critical, que restringe la ejecución asociada al bloque a un único hilo, por lo que el resultado obtenido no es el correcto. También disminuye el rendimiento, ya que se reduce el paralelismo y cada hilo pasa mucho tiempo en espera.

**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?**




Estos dos programas utilizan las directivas parallel, default(shared), for y private. Además, el programa pi_par7 utiliza también reduction.

La directiva parallel sirve para formar un equipo de hilos e iniciar la ejecución paralela, con default se controla el atributo por defecto de compartición de datos para variables referenciadas en el constructor de la región paralela, for se utiliza para especificar que las iteraciones se repartirán y ejecutarán por un equipo de hilos y con private se declaran variables privadas en el programa pi_par6. Además, el programa pi_par7 utiliza la cláusula reduction con lo que cada hilo tiene su propia variable que utiliza por separado, y al final de la ejecución de los hilos se suman todas para formar una sola.

Debido a la directiva reduction, el programa pi_par7 es mucho más eficiente que el pi_par6, ya que esta es muy potente.