MEMORIA
PRÁCTICA 4 ARQO

Potencial de las arquitecturas modernas

Jesús Daniel Franco López

ÍNDICE

Jesús Daniel Franco López
PT44 - 2019/2020

Ejercicio 0

Para obtener la información sobre la topología del sistema ejecutamos el comando cat /proc/cpuinfo en uno de los ordenadores del laboratorio 16, que son i7.
La información obtenida es la siguiente: existen 8 procesadores con 4 CPUs físicas y 8 CPUs virtuales. Hay hyperthreading activa ya que 8/4 = 2. Todos los procesadores tienen una frecuencia del orden de 900.000 MHz.

Adjuntamos captura de la salida de terminal para el primer procesador.

```
e338009@16-11-64-233:~/Downloads/materialP4$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
stepping        : 9
microcode       : 0x84
cpu MHz         : 900.126
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdp
e1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor d
s_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dno
wprefetch cpuid_fault epb invpcid_single pti ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invp
cid rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 7200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

Ampliamos la parte que nos da la información requerida

```
e338009@16-11-64-233:~/Downloads/materialP4$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
stepping        : 9
microcode       : 0x84
cpu MHz         : 900.126
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
```

Ejercicio 1

Captura de la salida del programa omp1

```
e338009@16-11-64-233:~/Downloads/materialP4$ ./omp1 5
Hay 8 cores disponibles
Me han pedido que lance 5 hilos
Hola, soy el hilo 0 de 5
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
```

1.1¿Se pueden lanzar más threads que cores tenga el sistema?
¿Tiene sentido hacerlo?
Sí se pueden lanzar más threads que cores tenga el sistema y tiene sentido hacerlo, ya que así los distintos hilos accederán a los recursos del sistema de manera compartida. Haciendo esto, las tareas se llevan a cabo con un consumo menor de recursos. Si ejecuto este comando en mi ordenador personal se observa como, efectivamente, se permite el lanzamiento de más threads que cores del sistema.

```
jesus@odin:~/Documentos/repositorios git/arqo_2019_2020/p4/EJERCICIO 2$ ./omp1 5
Hay 4 cores disponibles
Me han pedido que lance 5 hilos
Hola, soy el hilo 2 de 5
Hola, soy el hilo 1 de 5
Hola, soy el hilo 4 de 5
Hola, soy el hilo 3 de 5
Hola, soy el hilo 0 de 5
jesus@odin:~/Documentos/repositorios git/arqo_2019_2020/p4/EJERCICIO 2$
```

1.2¿Cuántos threads debería utilizar en los ordenadores del laboratorio? ¿y en su propio equipo?
Se deberían usar 8 threads en los laboratorios de los ordenadores, ya que a cada núcleo físico le corresponden dos virtuales. En mi ordenador personal, como dispone de 4 cores, se deberían lanzar 4 threads. El equipo en el que hemos trabajado es un pc de los laboratorios, por lo que los threads son los mismo. En la práctica, vamos a separar las tareas en 4 hilos.

Captura de la salida del programa omp2

```
e338009@16-11-64-233:~/Downloads/materialP4$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7ffd6c2a47b4,x   &b = 0x7ffd6c2a47b8,    &c = 0x7ffd6c2a47bc

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7ffd6c2a4750,    &b = 0x7ffd6c2a47b8,    &c = 0x7ffd6c2a474c
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 1]-1: a = 0,       b = 2,  c = 3
[Hilo 1]         &a = 0x7f0b7fcb3e20,    &b = 0x7ffd6c2a47b8,    &c = 0x7f0b7fcb3e1c
[Hilo 1]-2: a = 21,      b = 6,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f0b7f4b2e20,    &b = 0x7ffd6c2a47b8,    &c = 0x7f0b7f4b2e1c
[Hilo 2]-2: a = 27,      b = 8,  c = 3
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f0b7ecb1e20,    &b = 0x7ffd6c2a47b8,    &c = 0x7f0b7ecb1e1c
[Hilo 3]-2: a = 33,      b = 10,         c = 3

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7ffd6c2a47b4,    &b = 0x7ffd6c2a47b8,    &c = 0x7ffd6c2a47bc
```

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

Cuando declaramos una variable, se le asigna un valor aleatorio en cada llamada a la misma. Esto se puede evitar declarándola como first private. En este caso, toma el valor que se la haya especificado.

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

En cada una de las regiones paralelas la variable toma el valor que le corresponde. Este valor no se ve alterado por los que tenga la variable en las otras regiones.

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

Cuando trabajamos con hilos y variables privadas, se "crea" una nueva variable para trabajar dentro del mismo. Es a esta variable a la que se le modifica el valor, por lo que, al acabar, vuelve a adquirir el valor que tenía antes de comenzar el hilo.

1.6 ¿Ocurre lo mismo con las variables públicas?

En este caso, a trabajar con hilos y variables públicas, no se hace una copia de la variable para trabajar dentro de la región paralela. Esto quiere decir que el valor de la variable se irá modificando a la vez que se va modificando dentro del hilo.
En definitiva, las variables públicas continúan con el valor que se le ha asignado en el hilo.

Ejercicio 2

Captura de las salidas de los programas pescalar_serie, pescalar_par1 y pescalar_par2.

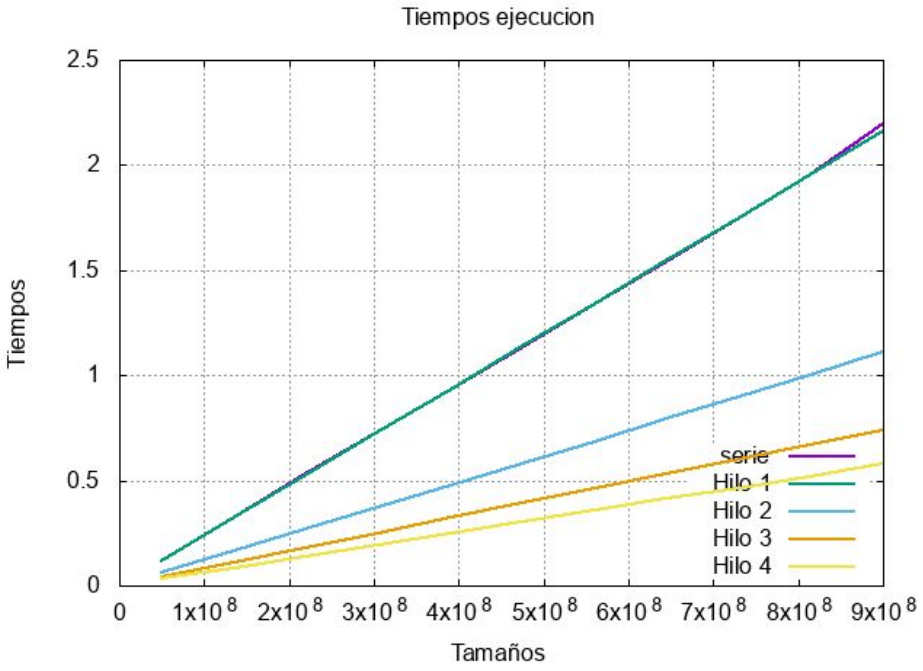
```
e338009@16-11-64-233:~/Downloads/materialP4$ ./pescalar_serie
Resultado: 33.319767
Tiempo: 0.002455
e338009@16-11-64-233:~/Downloads/materialP4$ ./pescalar_par1
Resultado: 4.774291
Tiempo: 0.004822
e338009@16-11-64-233:~/Downloads/materialP4$ ./pescalar_par2
Resultado: 33.330212
Tiempo: 0.003596
```

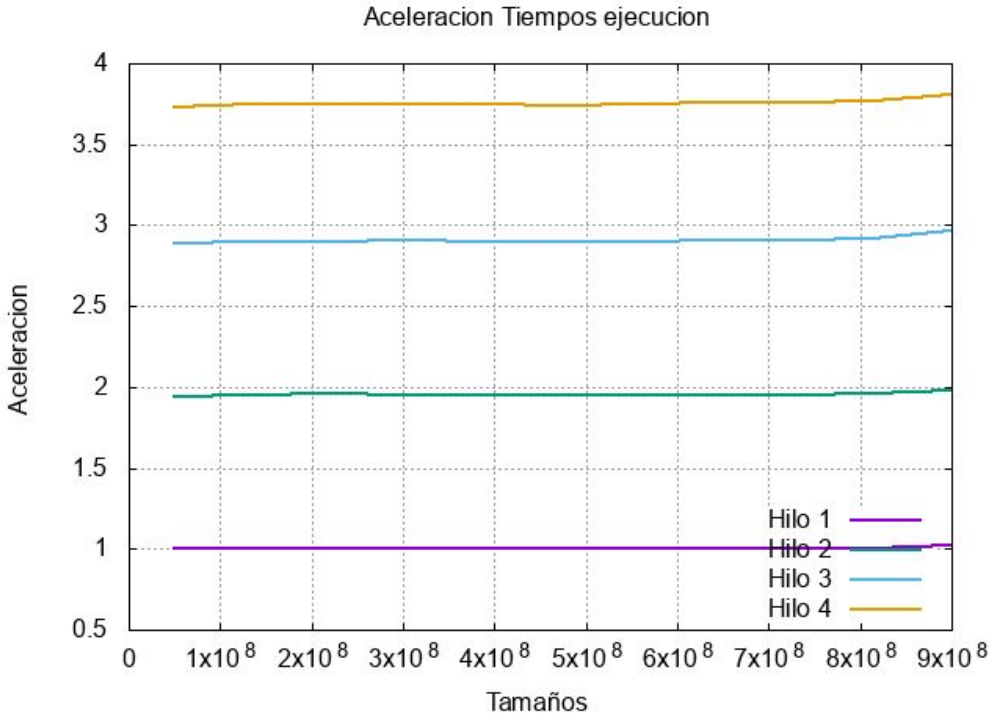
2.1 ¿En qué caso es correcto el resultado?

Observamos que los programas que devuelven el resultado correcto son el que realiza la operación sin paralelizar (pescalar_serie) y la segunda versión paralela (pescalar_par2).
Esta última se divide en tareas, de manera que cada hilo realiza una multiplicación y al final se suman.

2.2 ¿A qué se debe esta diferencia?

La diferencia entre las dos versiones paralelizadas es que, en la primera versión. la variable sum se actualiza en cada hilo, pero no se suma a los valores de los otro hilos. De esta manera la variable sum nos mostrará, finalmente, el resultado obtenido en el último hilo, y no el total.
La segunda versión, por el contrario, suma los resultado parciales del producto escalar, mediante el uso de "reduction (+:sum)".





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?

Observamos que cuando ejecutamos el programa lanzando un hilo, este tarda lo mismo que cuando se ejecuta el programa en serie. Esto lo vemos en ambas gráficas, las líneas de tiempos se solapan y la aceleración es siempre uno (o valores muy muy cercanos).

Cuando lanzamos más de un hilo sí que vemos que los tiempos disminuyen, y la aceleración se aproxima a cero, lo que nos indica que el tiempo de ejecución al lanzar los hilos es menor. Con esto podemos afirmar que, si lanzamos hilos, el rendimiento mejora en general.

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

Hay casos en los que no compensa lanzar hilos. Si trabajamos con vectores muy pequeños se tardaría más tiempo en crear los hilos y dividir las tareas en diferentes threads que en ejecutar el propio programa en serie.

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

El rendimiento del programa mejora cuando el número de hilos aumente, siempre y cuando este no supere el número de núcleos virtuales que tenga el ordenador. Los ordenadores de los laboratorios en los que hemos trabajado tenían 8, con lo que podían haber soportado 8 hilos. Como hemos lanzado 4, el rendimiento se ha visto mejorado.

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

Si lanzamos más hilos y vemos que el rendimiento no mejora, nos encontraríamos en el caso antes expuesto de que hemos superado el número de hilos que el ordenador puede manejar.

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

Si observamos nuestra gráfica de aceleración, nos damos cuenta de que la aceleración se mantiene constante (salvo ligeras desviaciones) para todos los tamaños que hemos usado. Con esto podemos afirmar que no va a haber ningún valor a partir del cual eso cambie.

Sin embargo, como explicamos en el punto 2.4, para tamaños muy pequeños de los vectores, la versión en serie podría tardar menos, por lo que la aceleración sería mayor que uno.

Ejercicio 3

Tiempos de ejecución (N=1000 y N=2000)

Versión\#hilos	1	2	3	4
Serie	7.15807	-	-	-
Par - Bucle 1	7.41457	4.8451	4.70983	4.77597
Par - Bucle 2	7.10497	3.74667	2.63817	2.34024
Par - Bucle 3	7.07757	3.96897	2.85973	2.07427

Versión\#hilos	1	2	3	4
Serie	67.8733	-	-	-
Par - Bucle 1	68.9923	36.828	34.743	35.045
Par - Bucle 2	68.169	35.2937	24.346	20.5711
Par - Bucle 3	67.8283	34.7683	23.4971	18.5017

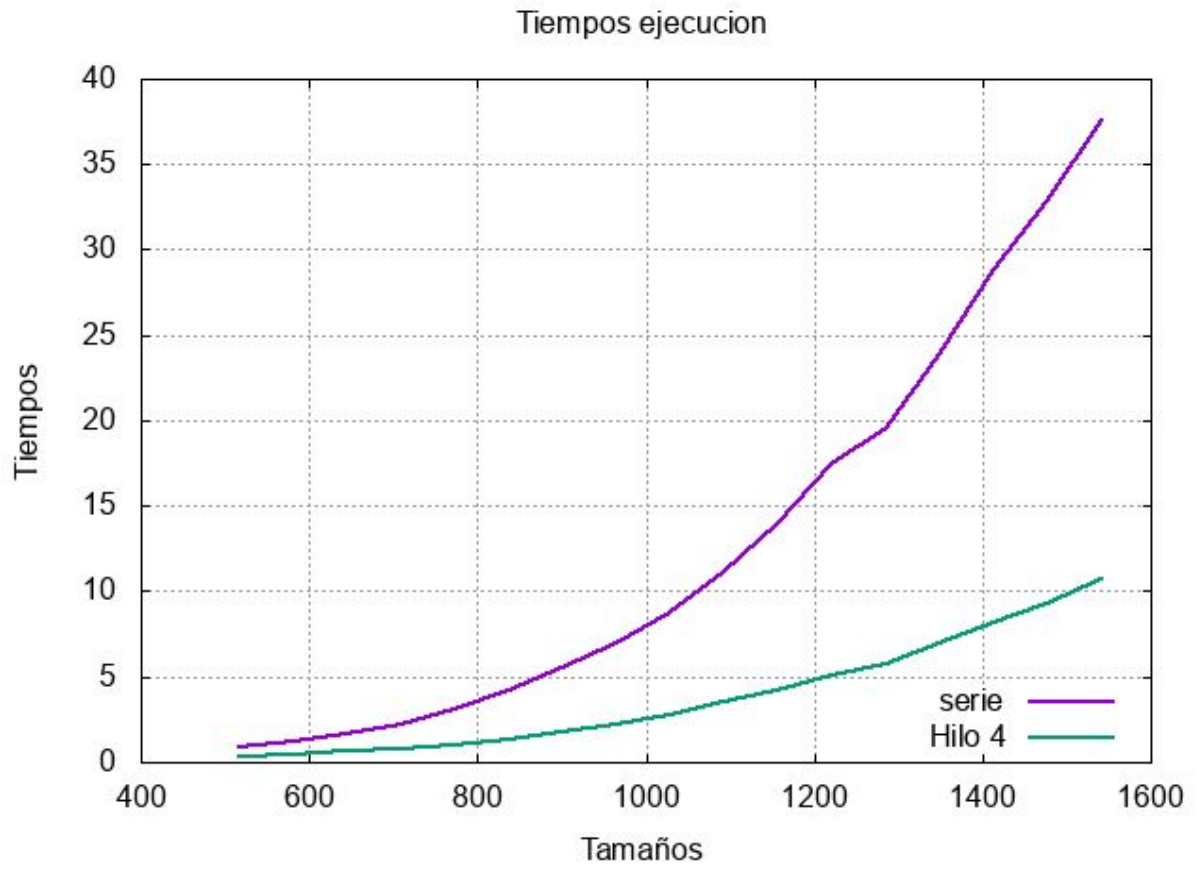
Aceleración (N=1000 y N=2000)

Versión\#hilos	1	2	3	4
Serie	1,000000	-	-	-
Par - Bucle 1	0.965406	0.492461	0.506605	0.49959
Par - Bucle 2	1.00747	0.636839	0.904424	1.01956
Par - Bucle 3	1.01137	0.60117	0.834352	1.1503

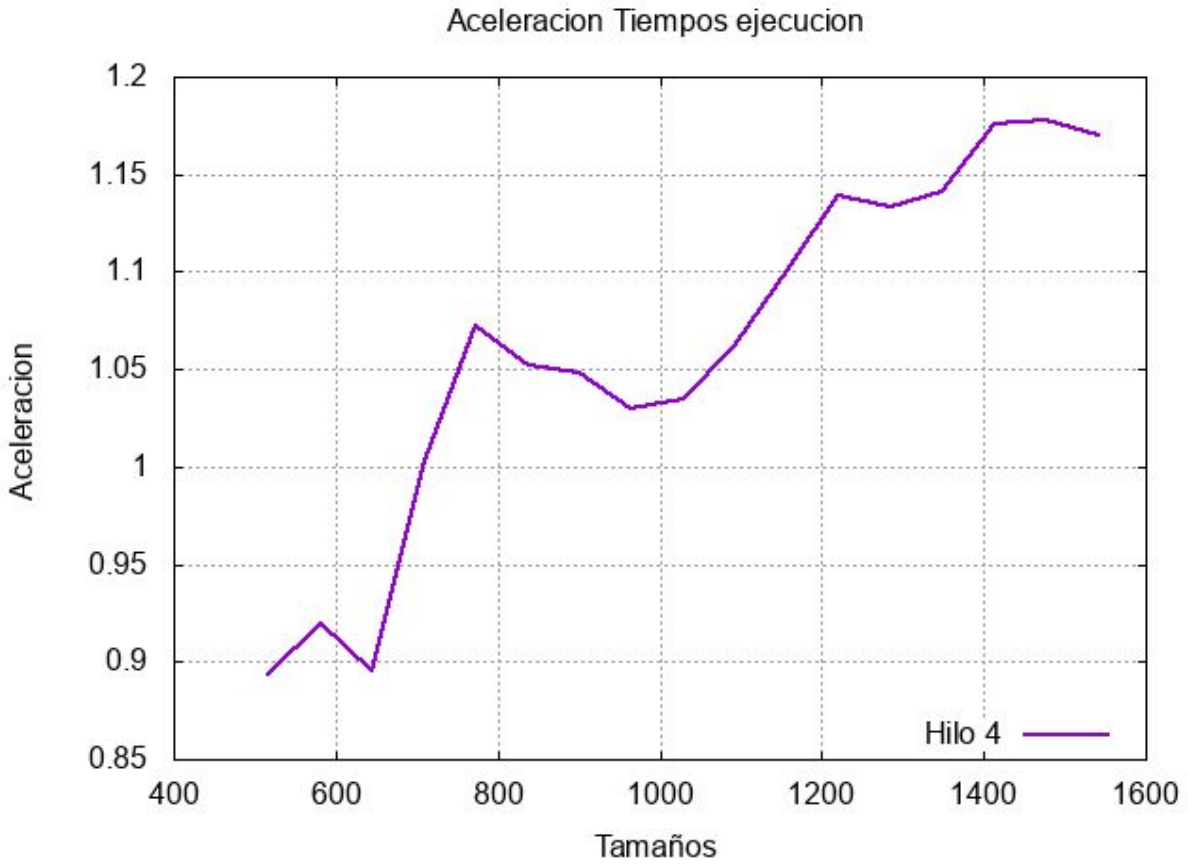
Versión\#hilos	1	2	3	4
Serie	1,000000	-	-	-
Par - Bucle 1	0.983781	0.614327	0.651194	0.645582
Par - Bucle 2	0.995663	0.641034	0.929287	1.09982
Par - Bucle 3	1.00066	0.65072	0.962861	1.22283

3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe?
La versión en la que paralelizamos el bucle 1 (el interno) es la que obtiene un peor rendimiento. La razón es sencilla, se están utilizando más recursos de los que realmente son necesarios debido a que lanzamos más hilos de los que en verdad necesitamos.

3.2 ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?
La versión en la que realizamos la paralelización del bucle 3 (bucle externo) es la que mejor rendimiento obtiene. El motivo es el inverso al de la pregunta anterior. En este caso lanzamos los hilos necesarios, normalmente el mismo número de hilos que de cores disponga nuestro terminal, y accedemos así a los recursos que son estrictamente necesarios.



Si observamos la gráfica, nos damos cuenta de que en las dos versiones representadas el tiempo de ejecución aumenta al hacerlo el tamaño de las matrices que multiplicamos. Aún así, vemos que el tiempo cuando paralelizamos el bucle externo y usamos 4 hilos aumenta a un menor ritmo que la versión de serie. Esto se debe a que las tareas se realizan al mismo tiempo. Como ya habíamos dicho en los apartados anteriores, la versión en la que paralelizamos tarda mucho menos que la versión serie.



A simple vista podríamos decir que la gráfica del aceleramiento es muy irregular (presenta muchos picos) pero si nos fijamos en el rango en el que varía la aceleración (0,9 - 1,2) nos damos cuenta de que sí que obtenemos la estabilidad buscada.

Ejercicio 4

```
gcc -g -fgomp -Wall -D_GNU_SOURCE -fopenmp -o
e362981@6B-21-64-217:~/materialP4$ ./pi_serie
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.319954
e362981@6B-21-64-217:~/materialP4$
```

Ejecución del programa que calcula pi en serie.

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

En la línea 11 del código de pi_serie.c (la versión en serie), se declara la variable n, número de rectángulos que utiliza el programa como 10^8.

```
e362981@6B-21-64-217:~/materialP4$ ./pi_par1
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.290621
e362981@6B-21-64-217:~/materialP4$ ./pi_par4
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.143813
e362981@6B-21-64-217:~/materialP4$
```

Ejecución del programa que calcula pi para versiones 1 y 4.

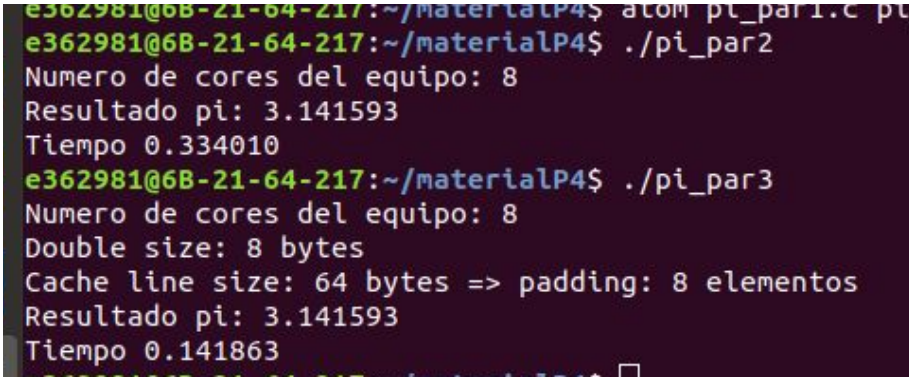
4.2 ¿Qué diferencias observa entre estas dos versiones?

La diferencia principal es que la versión pi_par4 utiliza una variable privada priv_sum en cada uno de los hilos en los que separa la tarea para hacer la operación 4.0 / (1.0 + x*x), y luego lo suma a la variable compartida sum(tid) (fuera del bucle), accediendo una sola vez a esta variable compartida. La versión pi_par1 va sumando dentro del bucle en la variable compartida sum(tid), accediendo tantas veces como hilos haya.

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?

	Serie	pi_par1	pi_par4
Rendimiento	1	$\frac{0.319954}{0.290621}$ = 1.10093	$\frac{0.319954}{0.143357}$ = 2.23187

El rendimiento de la versión pi_par4 es mucho mejor que el de la versión pi_par1. Esto se debe que la versión pi_par4, como habíamos explicado en el apartado anterior, accede una única vez a la variable compartida sum. pi_par1 accede a esta variable tantas veces como hilos haya, haciendo su ejecución más lenta.
Sobre el resultado de la pi obtenida observamos que coincide en ambos programas y los dos están bien.



Ejecución del programa que calcula pi para las versiones 2 y 3.

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?

	Serie	pi_par2	pi_par3
Rendimiento	1	$\frac{0.319954}{0.334010}$ = 0.95792	$\frac{0.319954}{0.141863}$ = 2.255373

El resultado en ambas versiones coincide y es correcto.
Observamos que el rendimiento de pi_par3 es mayor que el de pi_par2. Esto se debe a que. como hemos aumentado el tamamaño del array y no cabe en caché, cada hilo guarda en este array solamente la parte que modifica de dicho array, accediendo a menos podiciones de memoria y disminuyendo el tiempo de ejecución.

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 ejecución	Rendimiento
serie	0.319954	$\frac{0.319954}{0.319954} = 1$
1	0.324506	$\frac{0.319954}{0.324506} = 0.985972$
2	0.266293	$\frac{0.319954}{0.266293} = 1.201511$
4	0.174799	$\frac{0.319954}{0.174799} = 1.830411$
6	0.175808	$\frac{0.319954}{0.175808} = 1.819906$
7	0.169413	$\frac{0.319954}{0.169413} = 1.888603$
8	0.146932	$\frac{0.319954}{0.146932} = 2.177565$
9	0.150524	$\frac{0.319954}{0.150524} = 2.125601$
10	0.149563	$\frac{0.319954}{0.149563} = 2.139259$
12	0.150045	$\frac{0.319954}{0.150045} = 2.132387$

Vemos que en general, según vamos aumentando el valor, el rendimiento va mejorando. Observamos que el que mejor rendimiento tiene es en el que le asignamos el valor 8. Esto era de esperar, puesto que el ordenador en el que ejecutamos las distintas versiones del programa tiene 8 cores, haciendo muy fácil dividir las tareas en 8 hilos diferentes.

Ejercicio 5

```
e362981@6B-21-64-217:~/materialP4$ ./pi_serie
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.319954
```

Ejecución del programa que calcula pi en serie

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?

```
e338009@16-10-64-232:~/Downloads/materialP4$ ./pi_par4
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.143357
e338009@16-10-64-232:~/Downloads/materialP4$ ./pi_par5
Resultado pi: 3.424368
Tiempo 0.456526
```

Tabla de rendimientos:

	Serie	pi_par4	pi_par5
Rendimiento	1	$\frac{0.319954}{0.143357}$ = 2.23187	$\frac{0.319954}{0.456526}$ = 0.70084

Lo primero que debemos observar es que el cálculo de π a partir de la versión de paralelización 5 se aleja bastante del resultado real.
Aparte de esto, vemos que debido a la orden “critical” su rendimiento es mucho peor que el de la versión 4. Esto se debe a que estamos restringiendo una zona de código para que solo pueda ser ejecutada por un hilo a la vez, con lo que producimos tiempos de espera (de acceso a esta zona crítica) en los hilos.

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?

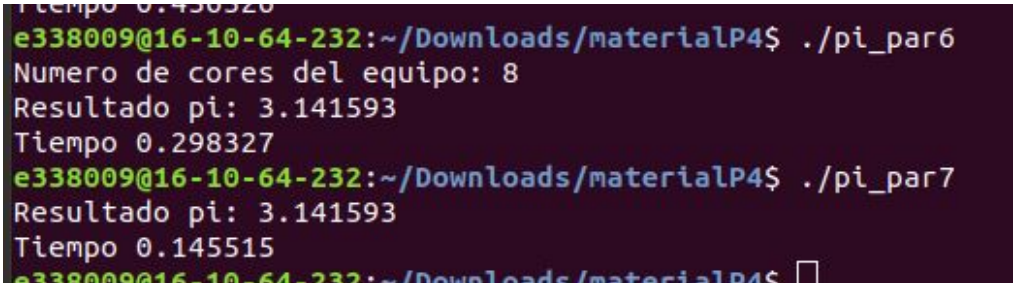


Tabla de rendimientos:

	Serie	pi_par6	pi_par7
Rendimiento	1	$\frac{0.319954}{0.298327} = 1.07249$	$\frac{0.319954}{0.145515} = 2.19877$

Sobre el resultado que obtenemos no podemos decir nada, en ambos se calcula el valor real de π.
Lo que sí observamos es que el rendimiento de la versión 7 es bastante mejor que el de la versión 6 (cerca del doble). Esto se debe al uso de la directiva reduction, lo que convierte en variables privadas la i y la j utilizadas en los bucles, causando un efecto parecido al que obtuvimos en el ejercicio 3 con el bucle 3.