# Memoria Arquitectura de Ordenadores (Práctica 4)

Miguel Arconada Manteca
miguel.arconada@estudiante.uam.es

Alberto González Klein
alberto.gonzalezk@estudiante.uam.es

12 de diciembre 2018

# Introducción

Antes de resolver las cuestiones relacionadas con los ejercicios, vamos a explicar la organización de las carpetas, así como los scripts realizados. Tenemos una carpeta para los ejercicios 2 y 3, ya que son para los que hemos desarrollado scripts y hemos obtenido datos y gráficas, y una carpeta **src**, además de los scripts *prepararTodo.sh* y *ejecutarTodo.sh*. El script *prepararTodo.sh*, como su nombre indica, prepara todos los ejercicios para su ejecución, es decir, hace *make clean* para limpiar los viejos ejecutables, *make* para generar nuevos y da permisos de ejecución (con *chmod +x*) a los scripts de todos los ejercicios. Por otro lado, *ejecutarTodo.sh* ejecuta todos los ejercicios en orden. Por tanto, si queremos ejecutar la práctica adecuadamente y de forma sencilla, llamaremos primero a *prepararTodo.sh* y posteriormente a *ejecutarTodo.sh*.

La carpeta **src** contiene el makefile, todos los .c y .h (carpeta **c**), los ejecutables de estos .c (carpeta **exes**) y el script de *Gnuplot* que hemos creado, *plotScript.sh* (carpeta **scripts**). Este script funciona con los siguientes argumentos:

- Argumentos obligatorios:
  - $-f$ : Nombres de los ficheros de donde se extraen los datos.
  - $-o$ : Columna del fichero que es la variable independiente.
  - $-d$ : Columnas del fichero que son las variables dependientes.
  - $-p$ : Nombre del archivo donde se guarda la gráfica.
- Argumentos opcionales:
  - $-x$ : Nombre del eje X.
  - $-y$ : Nombre del eje Y.
  - $-t$ : Título de la gráfica.
  - $-s$ : Estilo de la gráfica. Puesto con *lines* por defecto.
  - $-k$ : Posición de la leyenda. Puesto en left top por defecto.
  - $-w$ : Anchura de línea. Puesto a 2 por defecto
  - $-l$ : Etiquetas de los elementos graficados

En cada una de las carpetas de los ejercicios se encuentra el script que realiza la funcionalidad definida en el enunciado de la práctica, así como dos carpetas: **dat** y **png**. En la carpeta **dat** se encuentran los archivos en los que se han volcado los datos pedidos en el ejercicio correspondiente, mientras que en la carpeta **png** se encuentran las gráficas generadas.

## Ejercicio 0

```
$ cat /proc/cpuinfo

processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 1
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
```

```
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 2
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 1
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 3
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 1
```

```
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 4
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 2
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 5
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
```

```
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 2
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 6
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
processor       : 7
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
```

```
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2000.000
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 23
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
bogomips        : 4000.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

*A partir de la información expuesta, indique en la memoria asociada a la práctica los datos relativos a las cantidad y frecuencia de los procesadores disponibles en el equipo empleado, y a si la opción de hyperthreading está o no activa.*

Este equipo tiene 8 procesadores de 2000 MHz cada uno. No observamos la bandera *ht* en el campo flags, por lo que el *hyperthreading* no está activo.

# Ejercicio 1

## Apartado 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. Sin embargo, esto carece de sentido puesto que el objetivo de los threads es la ejecución en paralelo de varios programas o fragmentos de un mismo programa. Si lanzamos más hilos que cores, los hilos tendrán que competir por los recursos entre ellos, aumentando el tiempo de ejecución.

## Apartado 2

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

Deberíamos utilizar tantos threads como cores tenga el equipo que utilicemos. Mediante el uso del comando indicado en el ejercicio 0, podemos observar que el equipo del laboratorio tiene 4 cores, el cluster tiene 12 y el equipo propio 8. Por tanto, para el equipo del laboratorio deberemos lanzar hasta 4 threads, para el nuestro 8, mientras que para el cluster podremos usar hasta 11.

***Se pide ejecutar el programa omp2 e incluir la salida en la memoria de la práctica.***

```
Inicio: a = 1,    b = 2,   c = 3
&a = 0x7fffc27b3e44 ,x    &b = 0x7fffc27b3e48 ,     &c = 0x7fffc27b3e4c
[Hilo 0]-1: a = 0,        b = 2,   c = 3
[Hilo 0]          &a = 0x7fffc27b3de0 ,     &b = 0x7fffc27b3e48 ,
&c = 0x7fffc27b3ddc
[Hilo 0]-2: a = 15,       b = 4,   c = 3
[Hilo 3]-1: a = 0,        b = 2,   c = 3
[Hilo 3]          &a = 0x7fefa456fe20 ,     &b = 0x7fffc27b3e48 ,
&c = 0x7fefa456fe1c
[Hilo 3]-2: a = 21,       b = 6,   c = 3
[Hilo 1]-1: a = 0,        b = 2,   c = 3
[Hilo 1]          &a = 0x7fefa558fe20 ,     &b = 0x7fffc27b3e48 ,
&c = 0x7fefa558fe1c
[Hilo 1]-2: a = 27,       b = 8,   c = 3
[Hilo 2]-1: a = 0,        b = 2,   c = 3
[Hilo 2]          &a = 0x7fefa4d7fe20 ,     &b = 0x7fffc27b3e48 ,
&c = 0x7fefa4d7fe1c
```

```
[Hilo 2]-2: a = 33,        b = 10,          c = 3
Fin: a = 1,        b = 10,          c = 3
&a = 0x7fffc27b3e44,     &b = 0x7fffc27b3e48,     &c = 0x7fffc27b3e4c
```

## Apartado 3

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

Para las variables privadas cada thread tiene la suya propia y no comparte su valor con el resto de threads, es decir, que los cambios en la variable en el thread 1 no afectan al valor de la "misma"variable en el thread 2, por ejemplo. Ponemos misma entre comillas puesto que lo que hace OpenMP es realizar copias de la variable por lo que tecnicamente no son la misma variable sino dos variables distintas con el mismo nombre.

## Apartado 4

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

Si es una variable *private* estará sin inicializar. En cambio, si es de tipo *firstprivate* estará inicializada con el valor que tuviese antes de la sentencia *pragma*, es decir, el valor que tuviese antes de lanzar los hilos.

## Apartado 5

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

El valor que tuviese la variable privada antes de entrar a la región paralela no se ve afectado. Podemos verlo como pasar un argumento por valor a una función. El valor va cambiando dentro de la función pero no se ve afectado el valor de la región que llame a dicha función.

## Apartado 6

*¿Ocurre lo mismo con las variables públicas?*

No, las variables públicas guardan siempre el valor que tuviese antes de la región paralela. Además los cambios del valor de dicha variable desde un thread afecta al valor de la variable en el resto. En este caso se trata de la misma variable como tal. Siguiendo la analogía anterior, podemos verlo como un paso de argumentos por referencia. Todos los hilos comparten la dirección de la variable y modifican el contenido de dicha dirección y por ello afectan al resto de hilos. Por ello, al final de la región paralela la variable tendrá último valor que haya obtenido en los hilos.

# Ejercicio 2

## Apartado 1

*¿En qué caso es correcto el resultado?*

La versión en serie y la versión paralela 2 dan el resultado correcto, aunque con un ligero error causado por la clausula *reduction* del for.

## Apartado 2

*¿A qué se debe esta diferencia?*

La versión 1 del producto escalar paralelizado no contiene la clausula *reduction* y la versión 2 sí. Reduction lo que hace es que todas las variables de los thread con un cierto nombre realicen la operación que se indique. En este caso, tenemos la variable **sum** y la operación +. OpenMP crea una copia de sum por cada hilo inicializada a la identidad de la operación (en este caso 0 por ser una suma). Realiza las operaciones sobre esa copia y cuando todos los hilos acaban las iteraciones fusiona los resultados de las copias en la variable sum original usando el operador indicado (+ en este caso). Por el contrario, en la versión 1 **sum**, al no ser especificada de otra manera en el comienzo de la región pragma, es una varible compartida entre todos los threads. Esto al no ser coordinado de ninguna forma puede provocar un comportamiento errático, como por ejemplo, realizar una operación con un valor de **sum** que no es el último obtenido.

El error causado por *reduction* mencionado anteriormente, se debe a que el orden de las operaciones no es el mismo en la versión en serie y en la versión 2 paralelizada. Como tenemos un número de decimales limitado y, por tanto, una precisión imperfecta, a cada operación podremos estar "perdiendo o ganando"valores.

## Apartado 3 y Apartado 4

*En términos del tamaño de los vectores, ¿compensa siempre lanzar hilos para realizar el trabajo en paralelo, o hay casos en los que no? ¿En qué casos no compensa y por qué?*

Para tamaños muy pequeños, es decir, pocas iteraciones de los bucles no compensa. Esto se debe a que lanzar los hilos y unirlos después tiene un coste de tiempo. Para que paralelizar sea rentable, este coste tiene que ser inferior al de las iteraciones extra por ejecutar secuencialmente.

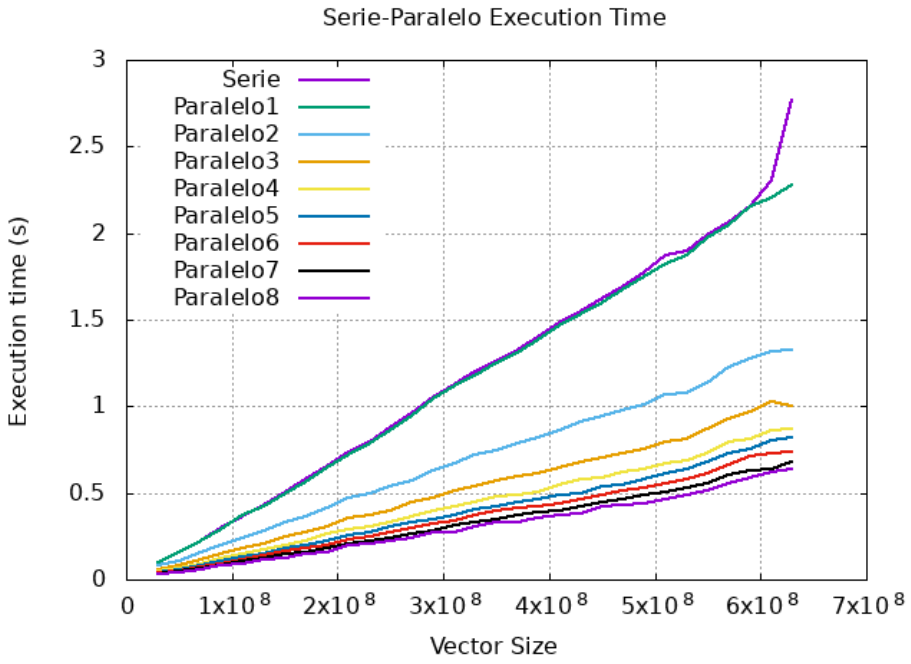


Figura 1: Tiempos de ejecución

En esta figura, al ser los tamaños muy grandes no se aprecia lo mencionado anteriormente, pero observando la tendencia de las rectas se observa que para tamaños inferiores a un cierto punto es más eficiente la ejecución en serie.

Además, también hay que tener en cuenta el efecto del posible *false sharing*. Sobre este tema profundizaremos en el **Ejercicio 3** y en el **Ejercicio 4**.

## Apartado 5 y Apartado 6

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

No siempre es el caso. En la figura 1, si nos fijamos en las líneas **Paralelo7** y **Paralelo8**, por ejemplo, observamos que para los primeros tamaños no hay una notable diferencia de tiempo debido a que lanzar un hilo más, al no haber suficientes iteraciones y debido a su coste, no aporta una mejora de rendimiento. Cuanto más aumenta el tamaño, observamos comienza a haber una cierta diferencia en los tiempos de ejecución. En esta gráfica hacemos zoom sobre los datos recogidos acerca de la paralelización con 7 y 8 hilos para observar lo mencionado:

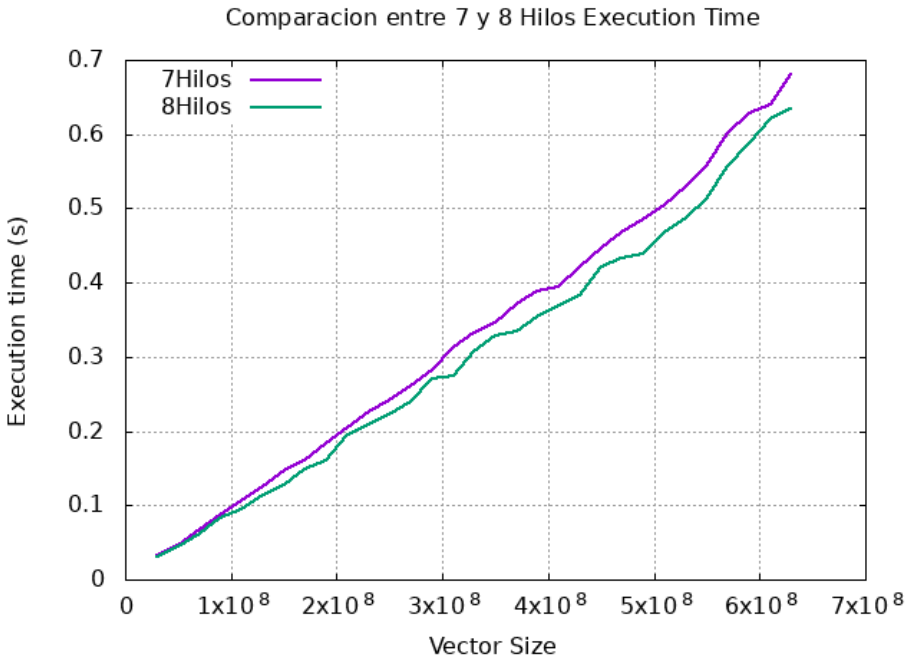


Figura 2: Tiempos de ejecución

Por ello, siempre hay que tener en cuenta la magnitud del trabajo a paralelizar y el coste de hacerlo para discernir de si es rentable la ejecución en paralelo o en serie y, en caso, de que paralelizar sea mejor, ser capaces de encontrar el número de hilos a lanzar óptimo.

De nuevo, también habría que evaluar el impacto del *false sharing* puesto que, a mayor número de hilos, mayor *false sharing* habrá.

## Apartado 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.*

La gráfica de aceleración obtenida es la siguiente:

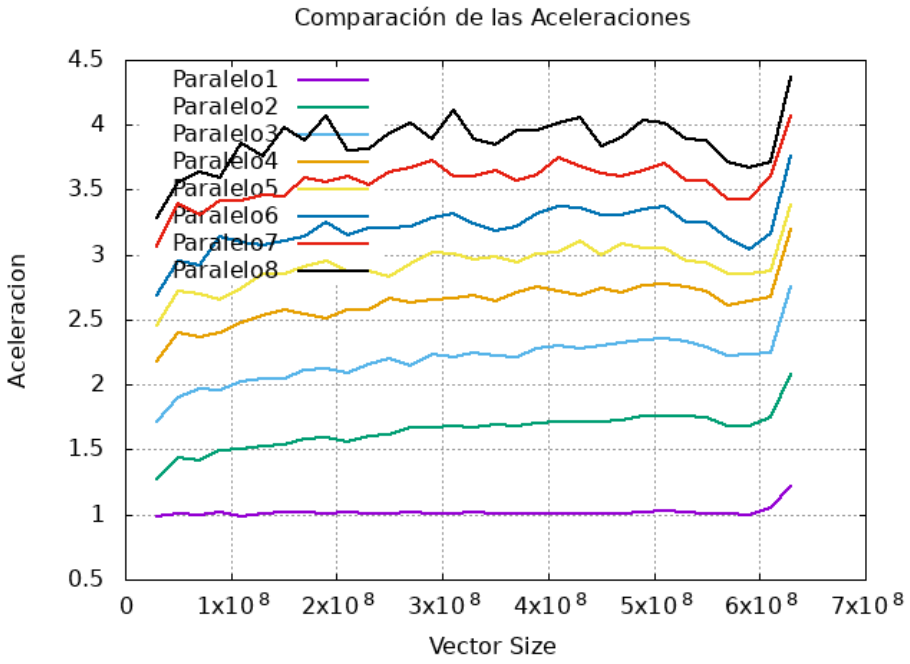


Figura 3: Aceleración

No pensamos que vaya a haber nunca una variación en el comportamiento de la aceleración, puesto que el crecimiento de los tiempos de ejecución para todas las versiones es de grado n, es decir, son rectas con pendiente constante. La aceleración representa la diferencia entre estas pendientes que, como son rectas con la misma pendiente siempre, va a tener siempre el mismo comportamiento, aproximadamiente.

# Ejercicio 3

Hemos rellenado las tablas para n=1000, n=1500 y n=3000:

## n = 1000

| Versión\# hilos | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 3.460449 | 3.460449 | 3.460449 | 3.460449 |
| Paralela – bucle1 | 4.069520 | 2.594466 | 2.485112 | 2.644982 |
| Paralela – bucle2 | 3.518227 | 10.252857 | 11.026664 | 11.105125 |
| Paralela – bucle3 | 3.461590 | 10.084728 | 11.006559 | 11.101285 |

Figura 4: Tiempos de ejecución n=1000

| Versión\# hilos | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 1 | 1 | 1 | 1 |
| Paralela – bucle1 | 0.850333 | 1.333780 | 1.392472 | 1.308307 |
| Paralela – bucle2 | 0.983577 | 0.337510 | 0.313825 | 0.311608 |
| Paralela – bucle3 | 0.999670 | 0.343137 | 0.314398 | 0.311716 |

Figura 5: Aceleración n=1000

## n = 1500

| Versión\# hilos | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 15.031540 | 15.031540 | 15.031540 | 15.031540 |
| Paralela – bucle1 | 17.195228 | 10.298523 | 8.983014 | 8.778434 |
| Paralela – bucle2 | 16.517506 | 34.598904 | 36.541597 | 37.521400 |
| Paralela – bucle3 | 16.501864 | 35.047034 | 36.605200 | 37.989000 |

Figura 6: Tiempos de ejecución n=1500

| Versión\# hilos | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 1 | 1 | 1 | 1 |
| Paralela – bucle1 | 0.874169 | 1.459582 | 1.673329 | 1.712325 |
| Paralela – bucle2 | 0.910036 | 0.434451 | 0.411354 | 0.400612 |
| Paralela – bucle3 | 0.910899 | 0.428896 | 0.410639 | 0.395681 |

Figura 7: Aceleración n=1500

### n = 2000

| Versión\# hilos | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 35.495556 | 35.495556 | 35.495556 | 35.495556 |
| Paralela – bucle1 | 36.099010 | 20.253538 | 18.073090 | 17.441690 |
| Paralela – bucle2 | 36.905864 | 81.174290 | 87.853927 | 88.642788 |
| Paralela – bucle3 | 36.972664 | 83.431883 | 87.704298 | 87.856890 |

Figura 8: Tiempos de ejecución n=2000

| Versión\# hilos | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 1 | 1 | 1 | 1 |
| Paralela – bucle1 | 0.983283 | 1.752560 | 1.964000 | 2.035098 |
| Paralela – bucle2 | 0.961786 | 0.437275 | 0.404029 | 0.400433 |
| Paralela – bucle3 | 0.960048 | 0.425443 | 0.404718 | 0.404015 |

Figura 9: Aceleración n=2000

## Apartado 1 y Apartado 2

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

Aunque con los valores de las tablas seríamos capaces de determinar qué versiones son más eficientes y cuáles son menos eficientes, hemos hecho una gráfica para ilustrarlo:

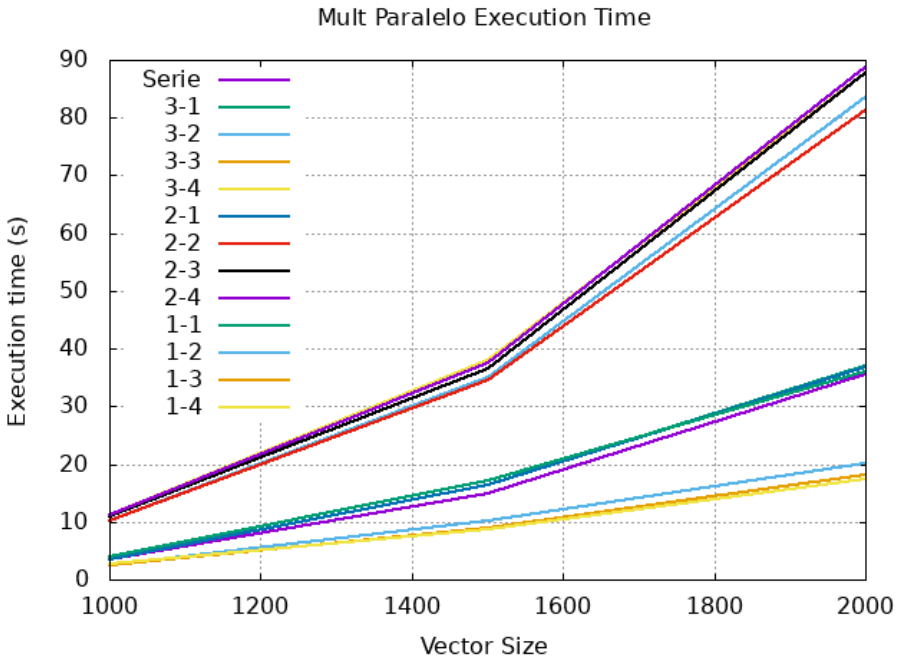


Figura 10: Tiempos de ejecución para n=1000, 1500, 2000

En esta gráfica se aprecia claramente que el bucle 1 es el que da un mejor rendimiento, es decir, el bucle interno. Esto se debe a que la escritura en la matriz la realizamos en el bucle intermedio, por lo que paralelizar el bucle 2 o el 3 causará *false sharing*, el cual, debido al gran tamaño de las matrices, causará un gran impacto sobre el tiempo de ejecución. En cambio, paralelizar el bucle 1 no produce este fenómeno, por lo que es el más eficiente.

Para poder discernir que versión es la menos eficiente con claridad tenemos la siguiente gráfica en la que excluimos el bucle 1:

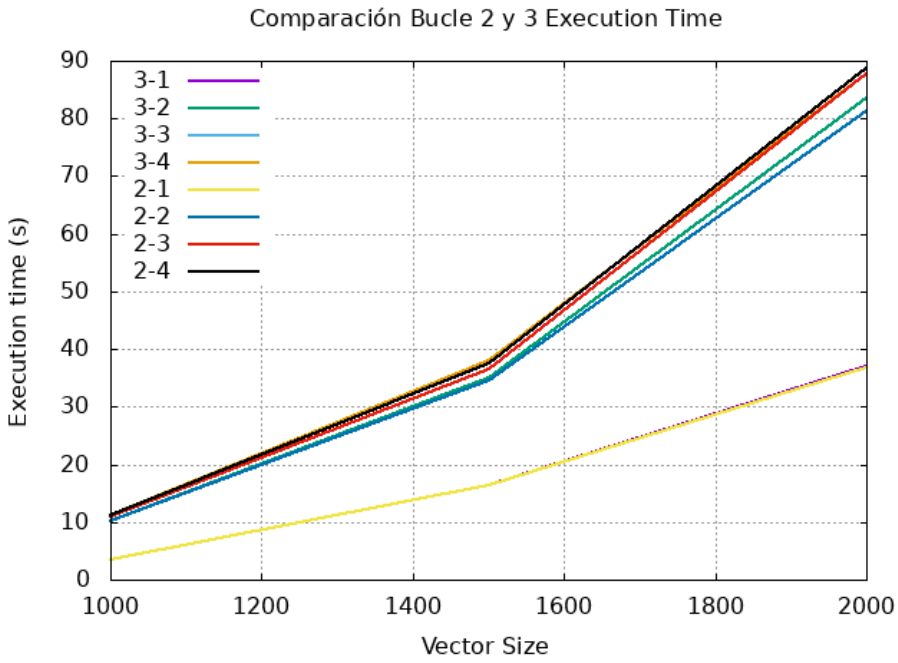


Figura 11: Tiempos de ejecución para Bucle 2 y 3

Aún no apreciamos qué versión es mejor. Lo que si obervamos es que cuantos más hilos peor es la eficiencia, lo cual tiene sentido ya que cuantos más hilos más *false sharing* habrá. Por ello, realizamos la comparativa entre el bucle 2 y 3 para 4 hilos:

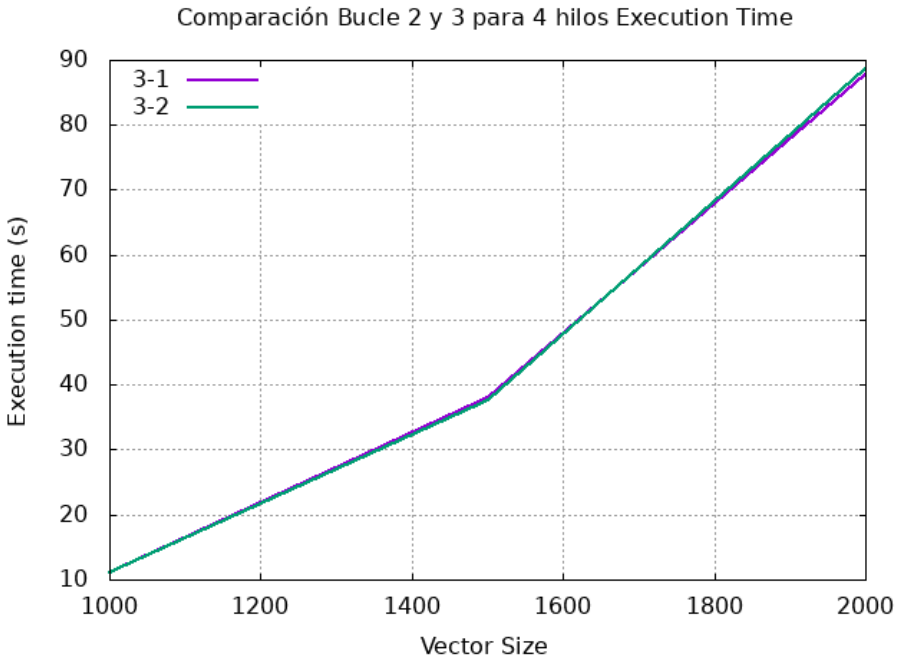


Figura 12: Tiempos de ejecución para Bucle 2 y 3 con 4 hilos

Observamos que prácticamente son iguales. Esto ocurre también para los otros números de hilos. Por tanto son igual de ineficientes. Esto se debe a que da igual cuál de los dos bucles se paralelice, el efecto del *false sharing* va a ser el mismo.

## Apartado 3

*Realice una gráfica de la evolución del tiempo de ejecución y la aceleración.Incluya un párrafo describiendo y justificando el comportamiento observado en las mismas. 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.*

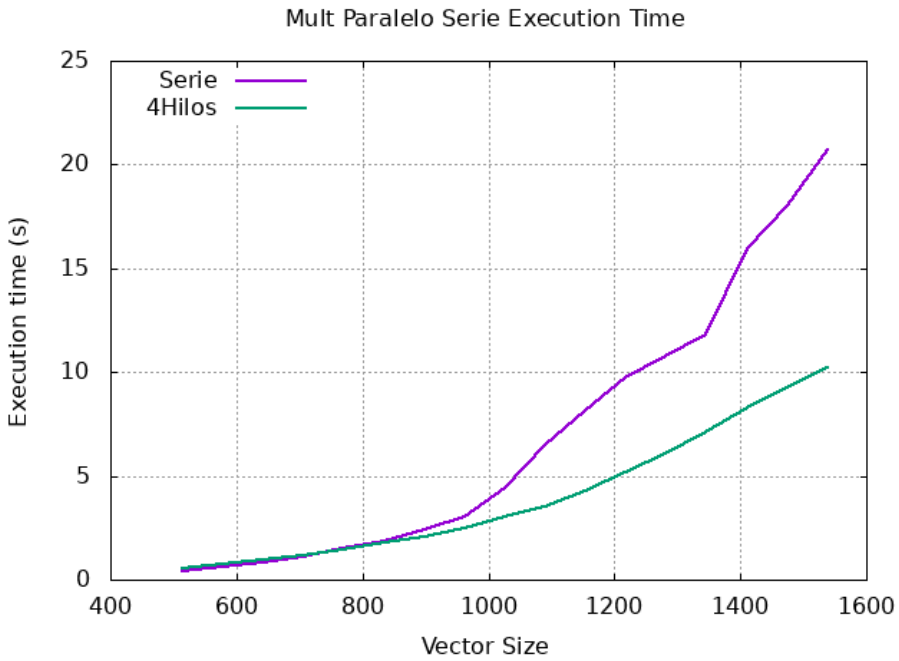


Figura 13: Tiempos de ejecución

Observamos que la versión serie tiene un crecimiento muy acelerado comparado con la versión paralela. Esto se debe a que el crecimiento de la versión serie es cúbico ($n*n*n = n^3$), mientras que la versión paralela tiene el mismo crecimiento que la versión serie pero dividida entre 4, al paralelizar el último bucle en 4 hilos y a efectos de contar el tiempo de ejecución se realizan $\frac{n}{4}$ iteraciones ($n*n*\frac{n}{4} = \frac{n^3}{4}$). Si observamos la velocidad a la que crecen $n^3$ y $\frac{n^3}{4}$ vemos que coincide con la tendencia de nuestra gráfica.

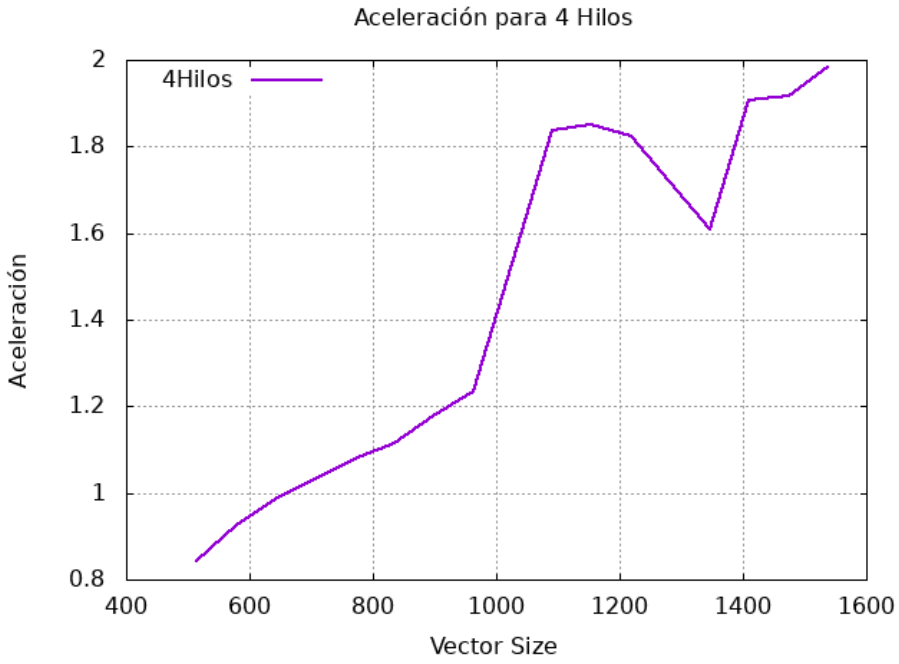


Figura 14: Aceleración

Para los tamaños indicados en el enunciado, observamos que crece. Aumentamos la N para intentar estabilizar la gráfica o incluso observar un decrecimiento. Para hacernos a la idea de qué tamaños deberíamos escoger usamos $Ninicio = 1400$, $Nfinal = 2000$, $Npaso = 200$ y $Niteraciones = 30$. Obtenemos:

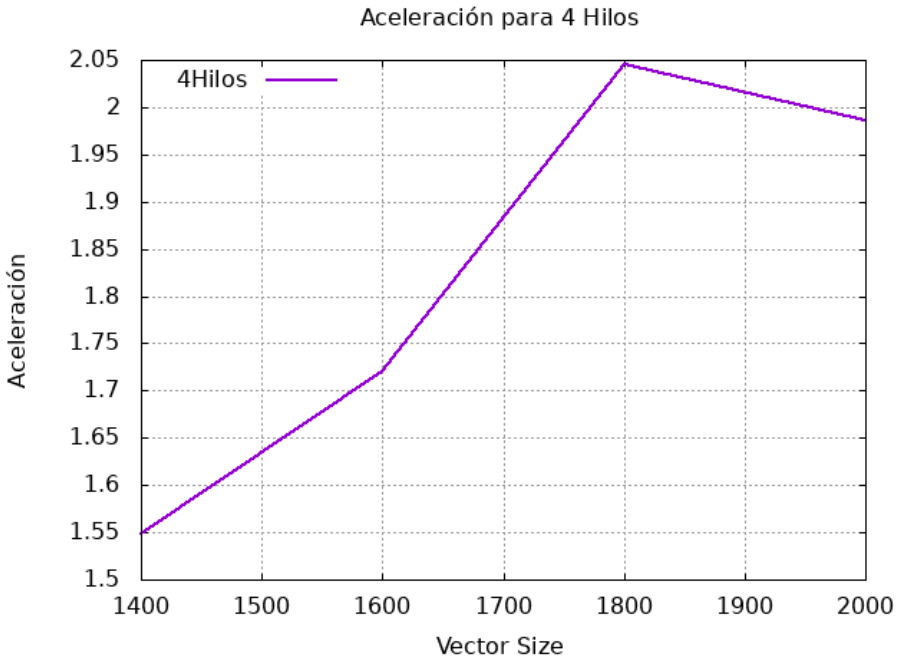


Figura 15: Aceleración

En está gráfica que, como podemos observar, no es muy precisa, apreciamos un cambio de comportamiento en torno al valor 1800. Por tanto, repetimos la ejecución en torno a ese valor. Los parámetros utilizados son: $Ninicio = 1800$, $Nfinal = 2400$, $Npaso = 50$ y $Niteraciones = 10$.

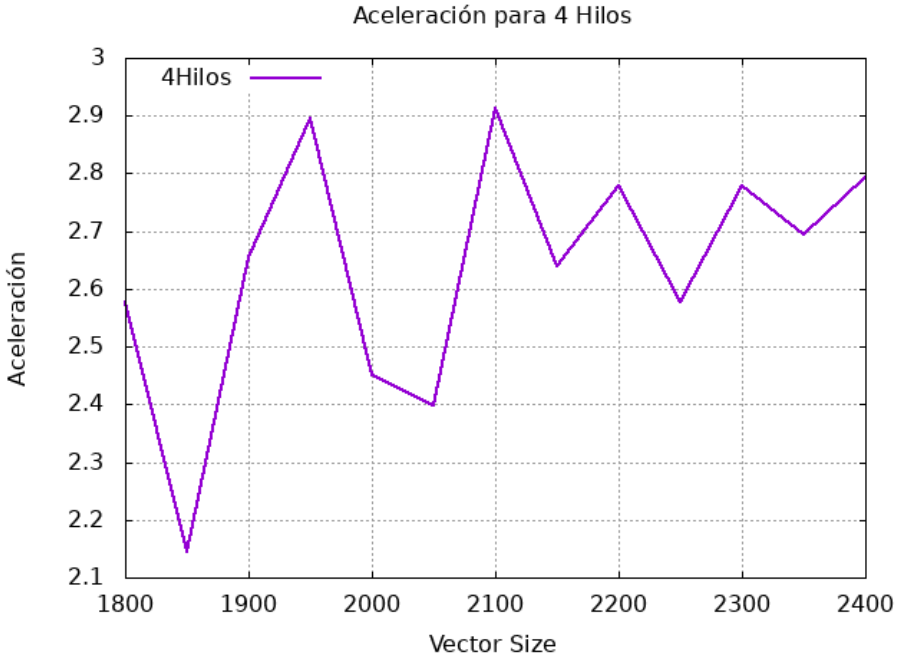


Figura 16: Aceleración

Observamos que es una gráfica con muchos picos. Esto se debe a que hemos elegido valores que están muy juntos y además, por motivos de tiempo, hemos utilizado pocas iteraciones. Aún así, estos picos tampoco suponen una variación muy grande y podemos afirmar que la aceleración se mantiene en torno al valor 2.7, aproximadamente. Para tratar de justificar este comportamiento, observamos la gráfica de los tiempos de ejecución asociados con esta aceleración:


Figura 17: Tiempos de ejecución

De nuevo, gráfica con picos causada por los factores mencionados con anterioridad. Aún así lo que podemos observar es que a mayor tamaño menor diferencia de pendientes de la recta tangente en el punto entre las gráficas de ambas versiones. Realmente, la aceleración lo que representa es la diferencia entre estas pendientes. Por ello, cuanto más aumente el tamaño menor diferencia de pendientes y la aceleración se estabilizará. En nuestro caso, en torno al valor $N = 2000$ observamos el cambio de comportamiento.

# Ejercicio 4

## Apartado 1

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

Se utilizan $10^8$ rectángulos, el valor de la variable $n$.

## Apartado 2

*¿Qué diferencias observa entre estas dos versiones?*

En la versión 1 la suma se realiza directamente en el array **sum**, mientras que en la versión 4 se declara una variable auxiliar, **priv_sum**, sobre la cual se realizan las sumas y posteriormente guarda su valor en el array.

## Apartado 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?*

Se obtiene el mismo resultado, pero el rendimiento es mucho mejor el de la versión 4. En nuestro caso es unas 5 veces mejor. Se debe al *false sharing*. En la versión 1 observamos que los hilos están accediendo a posiciones de memoria contiguas, lo que provoca que en la caché los datos estén posiblemente en el mismo bloque provocando este fenómeno. En cambio, en la versión 4 se utiliza una variable auxiliar por cada hilo para hacer la suma. Cada variable auxiliar puede estar o no (posiblemente no) en el mismo bloque de caché. Al realizar las operaciones sobre esta variable, se evita el *false sharing*.

## Apartado 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?*

El resultado obtenido es el mismo y el rendimiento es mejor en la versión 3 que en la versión 4. Era lo esperado, puesto que en la versión 2 no se realiza ninguna mejora significativa. Únicamente cada hilo genera una copia de sum inicializada (al ser de tipo *firstprivate*). Como **sum** es un puntero, lo que se copia es la dirección de memoria a la que apunta. De esta forma, cada hilo tiene su propio puntero a la dirección de memoria de **sum[0]** en lugar de compartir la variable.

En cambio, en la versión 3 lo que se hace es calcular cuántos doubles caben en una línea de la caché del equipo que se esté ejecutando. De esta forma, nos podemos asegurar que cada acceso de caché sea a una línea distinta, al desplazarse al menos ese número de posiciones de memoria de **sum**. De esta forma se evita el *false sharing*.

## Apartado 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?*

En el caso del ordenador en el que estamos ejecutando la práctica, sin cambiar la línea 32 tenemos $padsz = 8$. Esto quiere decir que tenemos 8 doubles por línea de caché. Por tanto, minimizar **padsz** empeora el rendimiento, pero aumentar el valor no lo mejora, ya que con $padsz = 8$ ya nos aseguramos que cada elemento del array se encuentré en líneas de caché distintas y, por tanto, un valor más grande no nos trae ningún beneficio. De esta forma, el rendimiento para los valores 8, 9, 10 y 12 es el mismo.

En cuanto a los valores menores de 8, vamos a analizarlos uno por uno. El valor 1 es simplemente análogo a ejecutar *pi_par1*. De hecho el rendimiento observado es el mismo aproximadamente. Para el valor 2, obtenemos un rendimiento aproximadamente el doble de mejor respecto al valor 1. Esto tiene sentido, puesto que si lo pensamos estamos evitando la mitad de los *false sharing*, aproximadamente. Lo mismo ocurre con los valores 4, 6 y 7. Tardan $\frac{1}{4}$, $\frac{1}{6}$ y $\frac{1}{7}$ respecto al valor 1 aproximadamente, respectivamente.

# Ejercicio 5

## Apartado 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?*

Hemos detectado lo que pensamos que es un error en *pi_par5.c*. La variable **i** estaba como una variable de tipo *shared* y esto provocaba que se ejecutase un menor número de bucles y ,por tanto, que las sumas parciales fuesen erróneas. Hemos solucionado este error declarando **i** en el interior de la región pragma para que cada hilo tenga su propia variable.

Dicho esto, no se aprecian cambios en el rendimiento. Intuitivamente, la versión 5 debería ser algo menos eficiente que la 4, puesto que la región crítica, al permitir solo que uno de los hilos ejecute el código de su interior, debería aumentar el tiempo de ejecución. Sin embargo, a esta región crítica solo se accede una vez por hilo, puesto que se calculan las sumas parciales fuera de la misma y solo se accede para guardar la suma calculada por cada uno de los hilos. Por tanto, el efecto es casi despreciable. Notaríamos el efecto de la región crítica si esta fuese muy grande (muchas líneas de código en su interior) o si se ejecutase un gran número de veces (como por ejemplo en un bucle). Además, con esta región crítica nos ahorramos el bucle externo utilizado en la versión 4 para calcular la suma de las sumas parciales.

## Apartado 2

La versión 7 es mucho más eficiente que la versión 6. Esto se debe a que, de nuevo, en la versión 6 se produce *false sharing* al usar un array para operar. Sin embargo, en la versión 7 se utiliza una variable privada por hilo que luego suman sus valores gracias a la claúsula *reduction*. Por tanto, no se produce *false sharing*. De hecho, la versión 7 y la 5 hacen exactamente lo mismo, ya que el funcionamiento de *reduction* es básicamente el de la región crítica usada en la versión 5.