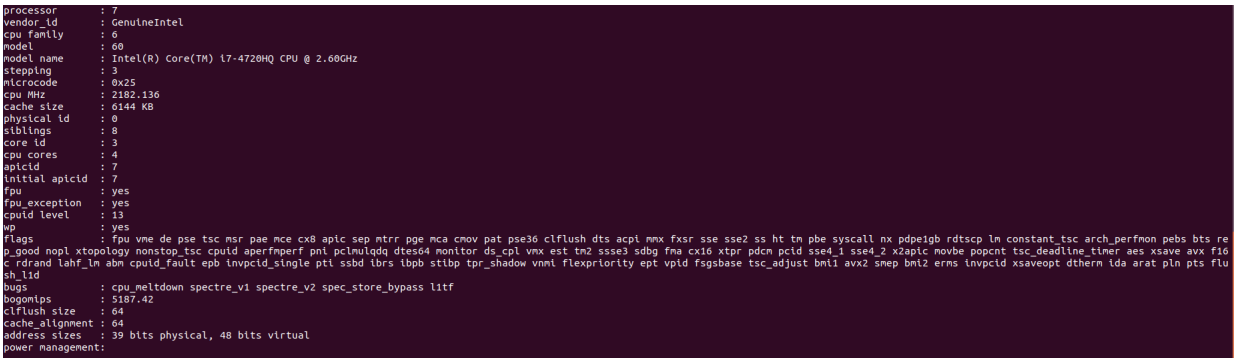
Práctica 4

Explotar el potencial de las arquitecturas modernas

Alejandro Santorum y David Cabornero

Ejercicio 0



```
processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz
stepping        : 3
microcode       : 0x25
cpu MHz         : 2182.136
cache size      : 6144 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 7
initial apicid  : 7
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
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 pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx 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 cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts flush_l1d
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf
bogomips        : 5187.42
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

Número de núcleos: 8
Frecuencia de procesadores: 2.6 GHz
El hyperthreading está activado en los 8 núcleos, lo cual hemos comprobado con el comando `cat /proc/cpuinfo | grep 'flags' | grep 'ht'`, y viendo que la flag ht está activa en todos los núcleos.

Ejercicio 1: Programas básicos de OpenMP

1.1

Efectivamente, como se muestra en la imagen a continuación, se pueden tener más threads que cores en el programa.

Sin embargo, no tiene demasiado sentido hacerlo, ya que se ejecuta un thread por núcleo. La misión principal consiste en dividir una tarea en varios threads para que los núcleos trabajen en paralelo, pero si hay threads de los que no puede ocuparse ningún core no se avanza demasiado en esta tarea.

1.2

Dicho lo anterior, optimizaremos al máximo nuestro programa cuando usemos correctamente todos los núcleos. Esto significa que en el laboratorio deberíamos usar 4 threads, si estamos en un cluster Intel de la EPS 16, si estamos en un cluster AMD de la EPS 12 y en casa, como hemos comprobado anteriormente, 8.

1.3

```
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./omp2
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7fff49f98814,x   &b = 0x7fff49f98818,    &c = 0x7fff49f9881c

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7fff49f987b0,    &b = 0x7fff49f98818,    &c = 0x7fff49f987ac
[Hilo 0]-2: a = 15,      b = 4,  c = 3

[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f5186148e20,    &b = 0x7fff49f98818,    &c = 0x7f5186148e1c
[Hilo 3]-2: a = 21,      b = 6,  c = 3

[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f5186949e20,    &b = 0x7fff49f98818,    &c = 0x7f5186949e1c
[Hilo 2]-2: a = 27,      b = 8,  c = 3

[Hilo 1]-1: a = 0,       b = 2,  c = 3
[Hilo 1]         &a = 0x7f518714ae20,    &b = 0x7fff49f98818,    &c = 0x7f518714ae1c
[Hilo 1]-2: a = 33,      b = 10,         c = 3


Fin: a = 1,      b = 10,         c = 3
        &a = 0x7fff49f98814,    &b = 0x7fff49f98818,    &c = 0x7fff49f9881c
```

Para cada thread se crea una variable distinta, y por lo tanto si modificamos una variable en cierto thread, el resto de threads no se verán afectados.

1.4

La variable privada no está inicializada, el programa por defecto nos indica que el valor de la variable es 0. Si en cambio no referimos a las *firstprivate*, su valor corresponde al valor que tenía la variable con el mismo nombre antes de paralelizar.

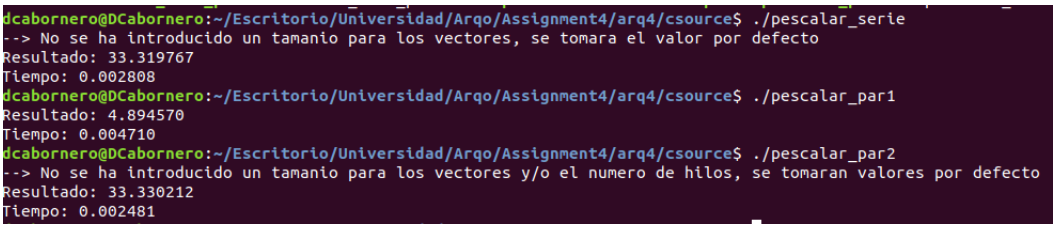
1.5

Al finalizar la región paralela la variable es eliminada, así que no adopta ningún valor. En este ejercicio concreto, tanto la variable a como la variable c (*private* y *firstprivate* respectivamente) tienen el valor asignado antes de que empezaramos la paralelización. Esto se debe a que estas variables solo existen en los threads, y al volver al thread master el programa se comporta como si estas variables eliminadas nunca hubieran existido.

1.6

Las variables públicas, en cambio, sí que cambian, ya que estas variables son compartidas para todos los threads, y cada vez que un thread la modifica esta variable es modificada en todos los threads. Podemos observar que cada thread incrementa en 2 el valor de la variable pública, y que es incrementada 2*(numero de threads) finalmente. La variable privada también es distinta, pero como consecuencia de que se le asigna un valor que depende de la variable pública.

Ejercicio 2: Paralelizar el producto escalar



2.1

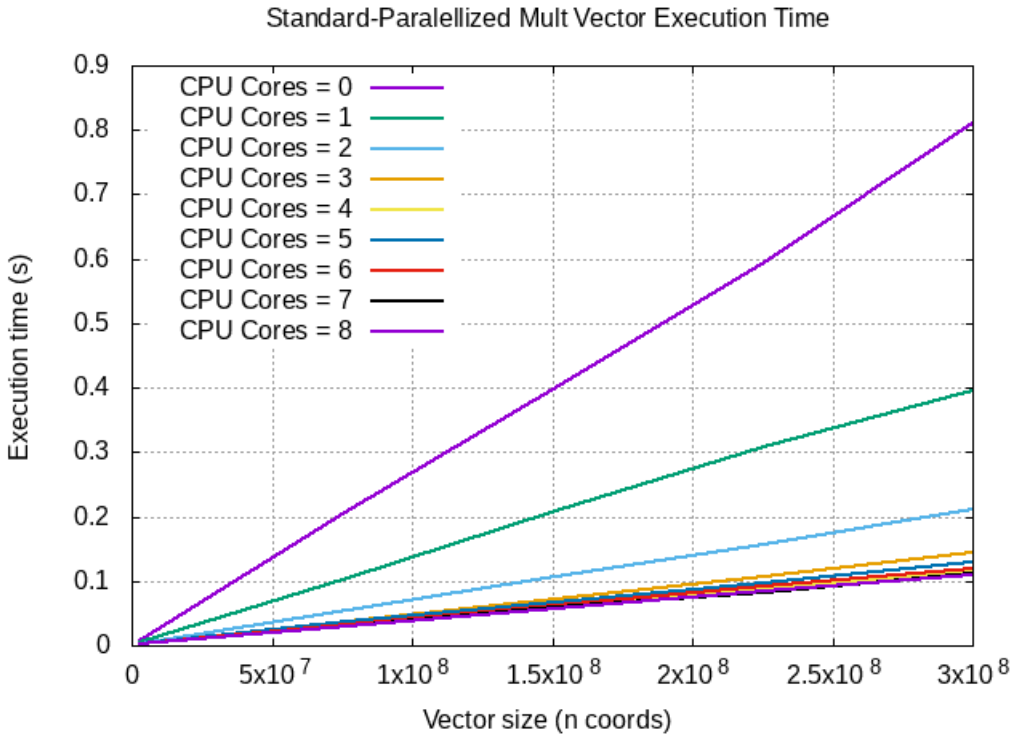
El resultado es correcto cuando se hace el producto escalar en serie (no se utiliza ningún tipo de paralelización, solo realizamos el producto escalar sin mayores complicaciones). Por ende, el segundo producto escalar paralelo tampoco está equivocado, pues da el mismo resultado.

2.2

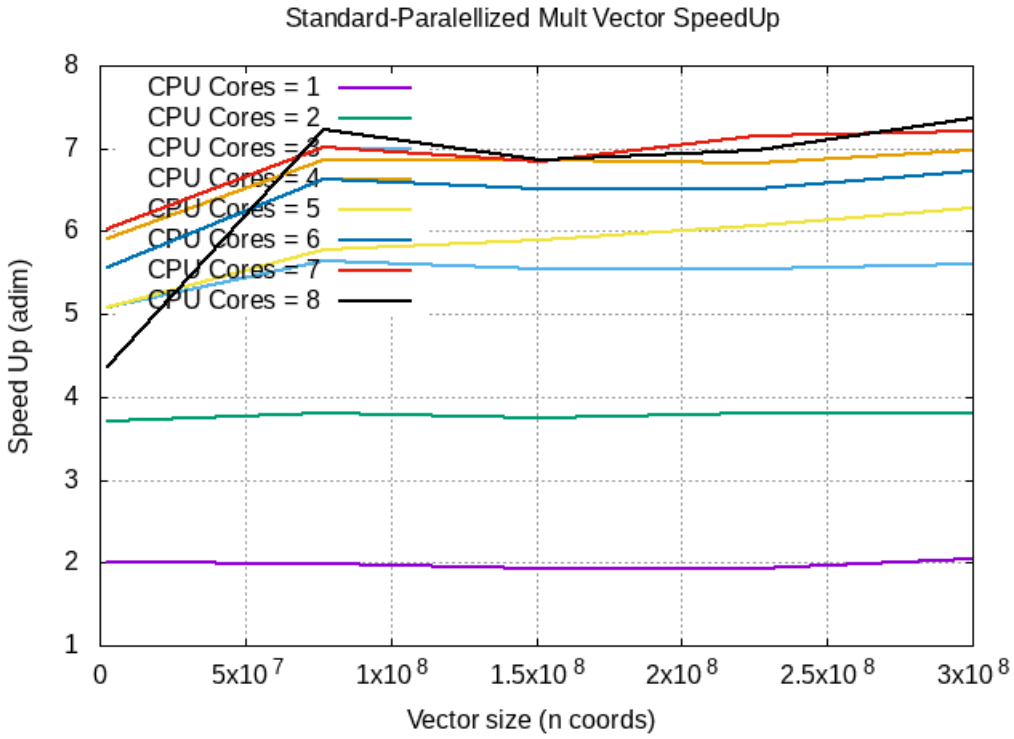
Es correcto el segundo paralelo, ya que estamos indicando mediante un reduction que la variable *sum* va a estar compartida y va a ser modificada en cada iteración del bucle.
En cambio, al no tener esta consideración en el primer escalar, la variable *sum* se modifica en ciertos momentos sin estar actualizada, por lo que no alcanza el valor que debería.

2.3

Hemos realizado dos gráficas, en la primera vemos la velocidad con la que se ejecuta el programa con distintos tamaños, dependiendo de los núcleos utilizados:



CPU Cores = 0 significa que estamos trabajando en serie. En la segunda gráfica, comparamos las aceleraciones con respecto al programa en serie:



Como vemos, paralelizar con 2 o 3 núcleos acelera mucho más el programa (respecto a la ejecución en serie), por lo que compensa muchísimo, y esa compensación se va viendo reducida cuando ya vamos aumentando el número de núcleos una vez paralelizado el programa con suficientes núcleos.
Sin embargo, parece concluyente que compensa bastante paralelizar nuestro programa. De todas formas, no compensa lanzar más hilos cuando este supera al máximo de núcleos.

2.4

Si en algún momento puede no compensar, ese momento es cuando nuestro programa es lo bastante pequeño como para que complejizarlo con paralelización no sea necesario, ya que el esfuerzo necesario no se va a ver retribuido en una mejora sustancial.
Para pequeños programas, puede no llegar a compensar el lanzar hilos, ya que el tiempo invertido en lanzar los hilos puede superar al que ganamos con la paralelización.

2.5

Por lo general, sí. Sin embargo, podemos encontrar momentos puntuales en los que las gráficas de aceleración se cruzan, dejando entrever que, en ciertos momentos, ha ido mejor con uno o más núcleos menos el mismo programa. Aún así, cabe destacar de nuevo que son momentos puntuales.
Todo esto es coherente siempre y cuando no se supere el número de núcleos que tiene el equipo, ya que a partir de ese momento, o bien no ayuda a mejorar el rendimiento del programa, o bien en el peor de los casos ralentiza el programa.

2.6

Como hemos dicho, estos cambios son puntuales y perfectamente pueden deberse al estado de la máquina, además de que, como también hemos apuntalado anteriormente, las mejoras en el programa una vez que se trabaje con muchos núcleos son poco sustanciales. Todo ello contribuye a que no sea extraño que, puntualmente, parezca que un núcleo menos es más provechoso. Si es el caso en que tenemos más hilos que núcleos, evidentemente no estamos acelerando el programa, como hemos mencionado anteriormente.

2.7

La gráfica no experimenta grandes cambios a partir de grandes valores. Si a lo que se refiere la pregunta es a la cantidad de núcleos utilizados, si siguiéramos aumentando los núcleos no se experimentaría una mejora.
Como se ha comentado anteriormente, los tamaños muy pequeños no confieren un aumento destacable con la paralelización, llegando incluso a retrasar el programa.

Ejercicio 3: Paralelizar la multiplicación de matrices

Tabla de tiempos de ejecución:

Size=1220	1	2	3	4	5	6	7	8
Serie	12.8471056	-	-	-	-	-	-	-
Bucle1	14.0502524	5.5812704	4.2027878	3.4551392	4.025613	3.8437454	3.7697566	3.9559044
Bucle2	3.0431754	1.6134918	1.0142372	0.8164612	1.2029944	0.9895728	0.8957572	0.7378412
Bucle3	2.984827	1.5082654	1.185273	0.7510784	0.8378694	0.7097376	0.6924318	0.6944052
Size=2020	1	2	3	4	5	6	7	8
Serie	68.7848692	-	-	-	-	-	-	-
Bucle1	71.9685826	29.847346	20.384959	16.1010406	22.4634052	20.5444662	18.4845306	17.7026608
Bucle2	18.2429312	9.2232986	7.6534886	4.8244574	6.4823496	5.507357	4.8338446	4.6128064
Bucle3	18.1956584	9.2964006	6.2050118	4.860676	5.0419848	4.2322864	3.909769	3.8345616

Tabla de aceleración de ejecución:

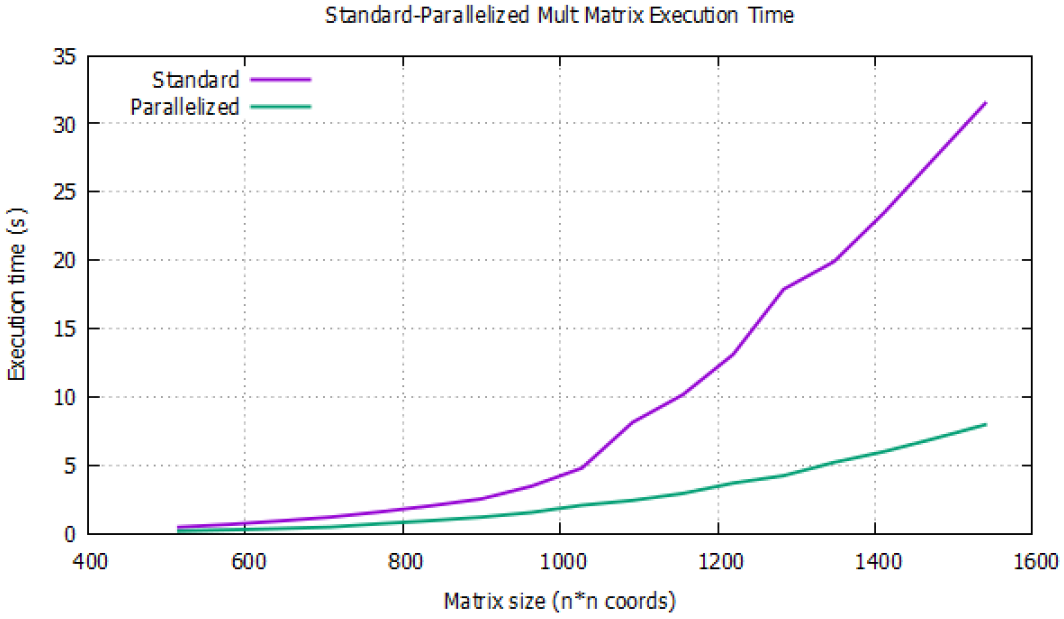
Size=1220	1	2	3	4	5	6	7	8
Serie	1	-	-	-	-	-	-	-
Bucle1	0.914368314	2.301824617	3.056805676	3.718259918	3.191341443	3.342340416	3.407940343	3.247577368
Bucle2	4.221611939	7.962299901	12.66676631	15.73510854	10.67927299	12.98247648	14.34217397	17.41174876
Bucle3	4.304137426	8.517801708	10.83894225	17.1048796	15.33306456	18.10120473	18.55360427	18.50087759
Size=2020	1	2	3	4	5	6	7	8
Serie	1	-	-	-	-	-	-	-
Bucle1	0.955762455	2.304555628	3.374295195	4.272076005	3.062085583	3.348097173	3.721212656	3.885566694
Bucle2	3.77049436	7.457729841	8.987387686	14.25753479	10.61110144	12.48963327	14.22984702	14.91171821
Bucle3	3.780290204	7.39908618	11.08537283	14.1512969	13.64241899	16.25241364	17.59307754	17.93813123

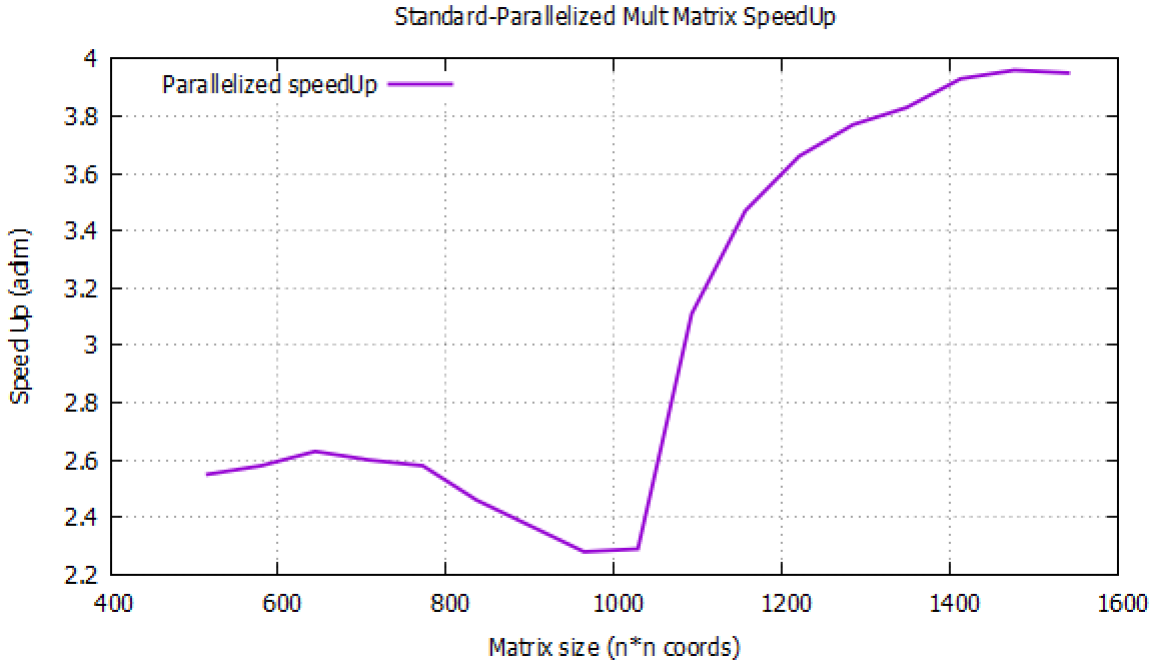
3.1

El bucle 1, ya que hay sobrecarga de creación de hilos, ya que para cada iteración se está creando un nuevo hilo, que hace un trabajo demasiado pequeño para la carga que supone crear el hilo, ralentizando así el programa.

3.2

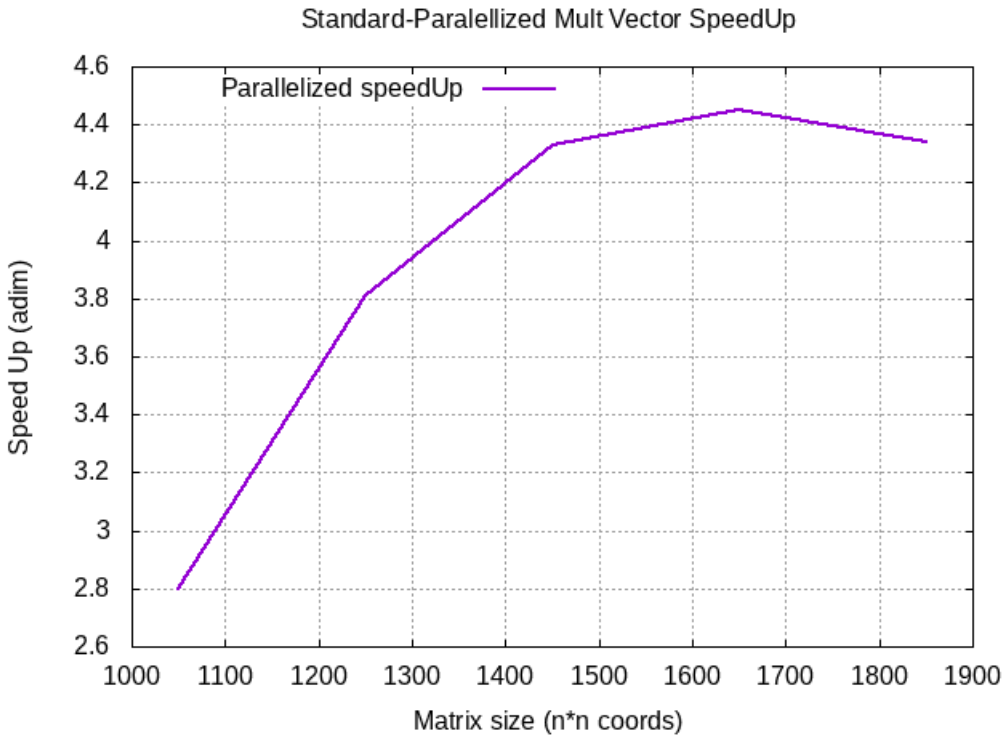
El bucle 3, ya que al ser el bucle más externo no se malgasta demasiado tiempo creando un número excesivo de hilos, a la vez que se aprovecha la paralelización del programa. Tampoco hay tanta diferencia con el bucle 2, ya que aunque se crean más hilos innecesarios, no se crean hilos compulsivamente como en el bucle 1.





Como se puede ver, los tiempos se corresponden con las expectativas, ya que la versión paralelizada es bastante más óptima que la versión en serie. Fijándonos en la tabla de aceleraciones, aumenta la aceleración según au-

menta el tamaño de la matriz, estabilizándose a partir de tamaños grandes (1300 x 1300). Como curiosidad, cerca de las potencias de dos se nota una aceleración menor, dado a que en la versión estándar se optimiza algo más el programa. Esto último se puede observar en la siguiente gráfica en la cual no hay muestras de potencias de dos:



Ejercicio 4: Ejemplo de integración numérica

4.1

La función se divide en 100000000 rectángulos, donde cada rectángulo tiene de longitud 100000000^{-1} y de altura la imagen del punto medio del intervalo.

4.2

La diferencia radica en que, a la hora de calcular el valor de cada rectángulo, el primer programa utiliza una variable compartida para los cálculos inter-

medios, mientras que el cuarto programa utiliza una variable privada para los mismos cálculos intermedios, y una vez finalizados el valor definitivo es asignado a la variable compartida.

4.3

```
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par1
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.372788
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par4
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.284700
```

No se observan diferencias en el resultado obtenido, de hecho, los dos son correctos, ambos se aproximan al número pi.
En cambio, sí que hay una clara diferencia en cuanto a tiempo empleado: el cuarto programa tarda menos. Esto es razonable, ya que, a diferencia del primero no se produce el efecto del *false sharing* 100000000 veces, sino 8 veces, ya que en la primera versión la suma se realiza sobre el array sum y en la cuarta versión estos cálculos se realizan en una variable privada, y finalmente sí que se asigna a la variable compartida. Solamente accede a memoria compartida cuando se han realizado todos los cálculos y debemos modificar nuestra variable.
El *false sharing* produce que, una vez que un hilo cambia un bloque, esté marcado como sucio en la memoria caché, y el siguiente thread que quiera leerlo y modificarlo tendrá que ir a por él a la memoria principal.

4.4

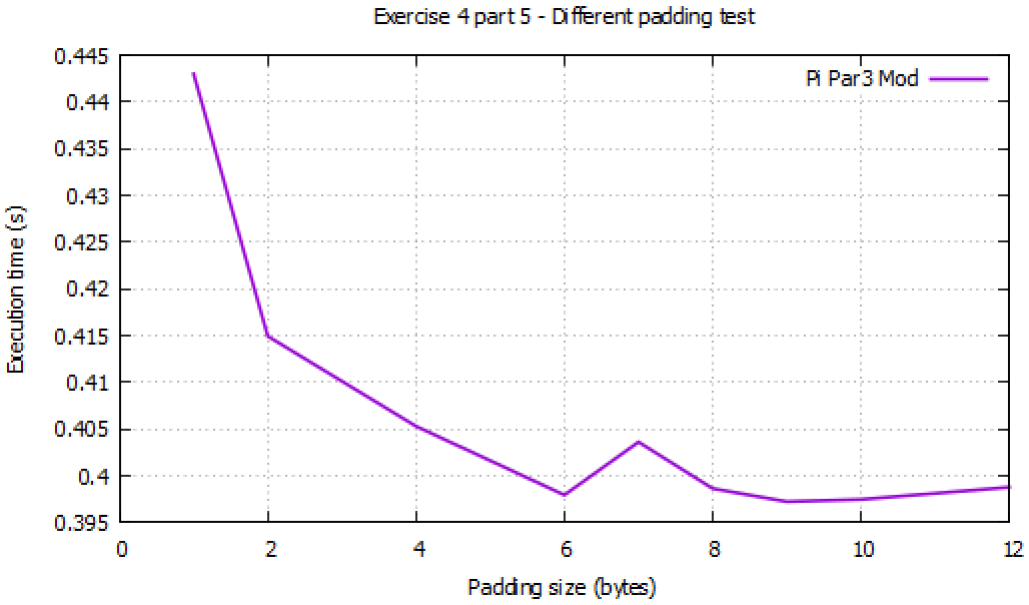
```
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par2
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.408054
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par3
Numero de cores del equipo: 8
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.195726
```

El resultado sigue siendo correcto, pero el 2 empeora su rendimiento, mientras que el 3 mejora aún más.
El 2 está cometiendo el mismo error que la primera versión, porque cuando se declara un puntero como *firstprivate*, todas las variables privadas creadas

apuntan al mismo sitio de memoria, por lo que sigue estando presente el mismo fenómeno de *false sharing*. En cambio, en la versión 3, se hace un array más grande para que cada hilo tenga su propio bloque de memoria, eliminando así el fenómeno de *false sharing*.

4.5

Hemos modificado el fichero pi_par3.c con el nombre pi_par3_mod.c. El tiempo de ejecución baja considerablemente aumentando el padding hasta 8, pero una vez llegados a ese punto la gráfica se estabiliza. Encontramos el sentido a esto en que el tamaño de línea del PC es de 64 bits, y por tanto en cuanto el padding supere el 8, estaremos trabajando con más de 8 Bytes, y por lo tanto no se compartirá ninguna zona entre los hilos. Aquí tenemos la muestra gráfica de lo explicado:



Ejercicio 5: Uso de la directiva critical y reduction

5.1

```
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par5
Resultado pi: 3.142388
Tiempo 0.534647
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par4
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.194343
```

La directiva critical se utiliza para que únicamente pueda acceder un único hilo a la vez a una cierta zona; es lo que también se conoce como un semáforo. Como vemos, la versión 4 es bastante más eficiente que la versión 5.
Esto se debe a que, el array compartido de la versión 4 tiene un array formado por varios bloques, donde cada hilo puede ir trabajando en un bloque distinto. En cambio, en la versión 5 la sección crítica reduce la eficiencia de la paralelización, ya que solo puede acceder un hilo concurrentemente.

5.2

```
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.190186
dcabornero@DCabornero:~/Escritorio/Universidad/Arqo/Assignment4/arq4/csource$ ./pi_par6
Numero de cores del equipo: 8
Resultado pi: 3.141593
Tiempo 0.369606
```

En la versión 6, utilizamos un array con tantos espacios como hilos haya, y en cada espacio vamos haciendo la suma parcial de pi que corresponde a ese hilo, realizadas mediante work sharing. Frente a esta idea encontramos la versión 7, donde utilizamos el método reduction, por lo que cada hilo trabaja con una variable privada distinta, realizando sumas parciales. Después, todas estas sumas parciales serán sumadas, para obtener el resultado final.
Como vemos, es más eficiente la versión 7, ya que sigue pecando de *false sharing* la versión 6 (bastante menos que en la versión 1 o 2), donde un double no es lo suficientemente grande como para rellenar un bloque entero, por lo que puede darse el caso de que dos hilos intenten acceder a un mismo bloque, ralentizando el proceso. Por el contrario, la versión 7 realiza un repartimiento del trabajo más eficiente gracias al compilador y la claúsula *reduction*.