

Arquitectura de Ordenadores

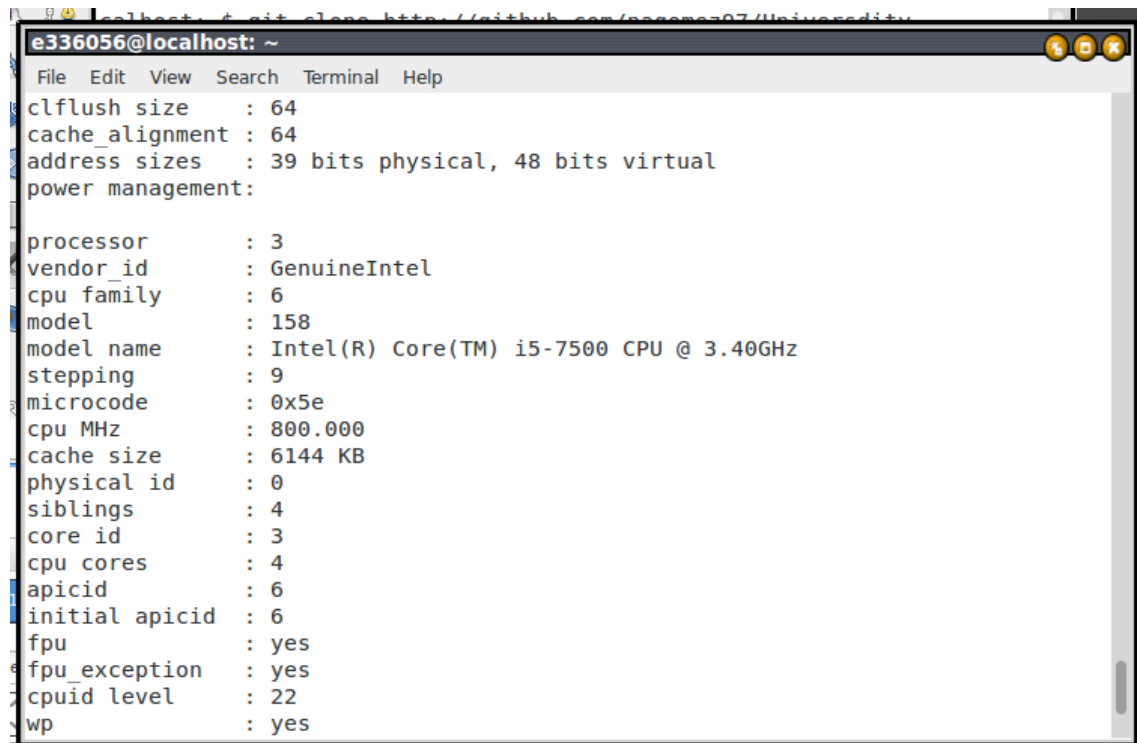
Práctica 4: Explotar el potencial de las arquitecturas modernas

UAM 2017-2018

Óscar Gómez Borzdynski
José Ignacio Gómez García

Ejercicio 0: Información sobre la topología del sistema

En este ejercicio queremos obtener información sobre el sistema usando los comandos `cat /proc/cpuinfo`. Cuya salida es la siguiente:



```
e336056@localhost: ~  
File Edit View Search Terminal Help  
clflush size      : 64  
cache_alignment  : 64  
address sizes     : 39 bits physical, 48 bits virtual  
power management:  
  
processor         : 3  
vendor_id        : GenuineIntel  
cpu family       : 6  
model            : 158  
model name       : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz  
stepping         : 9  
microcode        : 0x5e  
cpu MHz          : 800.000  
cache size       : 6144 KB  
physical id      : 0  
siblings         : 4  
core id          : 3  
cpu cores        : 4  
apicid           : 6  
initial apicid   : 6  
fpu              : yes  
fpu_exception    : yes  
cpuid level      : 22  
wp               : yes
```

Podemos ver que nuestro ordenador cuenta con un procesador i5-7500 con una frecuencia máxima de 3.40 GHz, 4 **cpu cores** y 4 **siblings**. Como tiene el mismo número de cores que de procesos virtuales (siblings), deducimos que se le asigna un proceso a cada core, por lo que no se está produciendo *hyperthreading*.

También podemos ver que el core número 3 está corriendo a una frecuencia de 800MHz (mucho más pequeña del máximo indicado). Esto se debe a que los procesadores modernos pueden adaptar su frecuencia de trabajo para ahorrar energía. Para entender mejor su funcionamiento, pusimos nuestra máquina a trabajar con un programa grande, obteniendo el siguiente resultado:

```
e336056@localhost:~$ cat /proc/cpuinfo | grep MHz  
cpu MHz          : 2300.000  
cpu MHz          : 3401.000  
cpu MHz          : 3401.000  
cpu MHz          : 3401.000
```

Como se puede ver, las frecuencias de trabajo aumentan significativamente hasta alcanzar su máximo (3.4 GHz).

Ejercicio 1: Programas básicos de OpenMP

En este ejercicio vamos a familiarizarnos con programas basados en OpenMP. Para ello contamos con *omp1.c* y *omp2.c*.

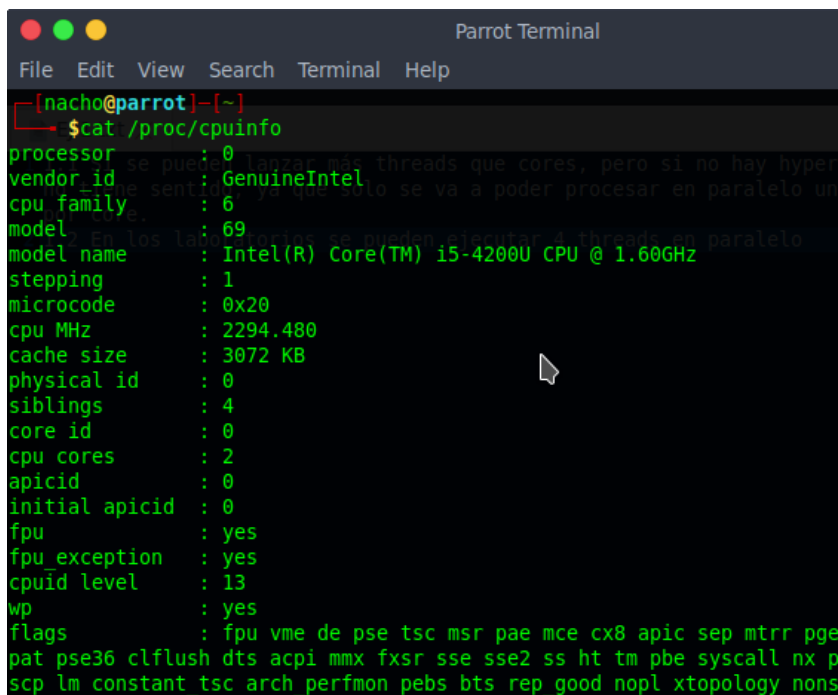
Tras correr el primer programa, contestamos a las siguientes preguntas.

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

Sí se pueden lanzar más threads que cores, pero si no hay hyperthreading no tiene sentido, ya que sólo se va a poder procesar en paralelo un thread por core.

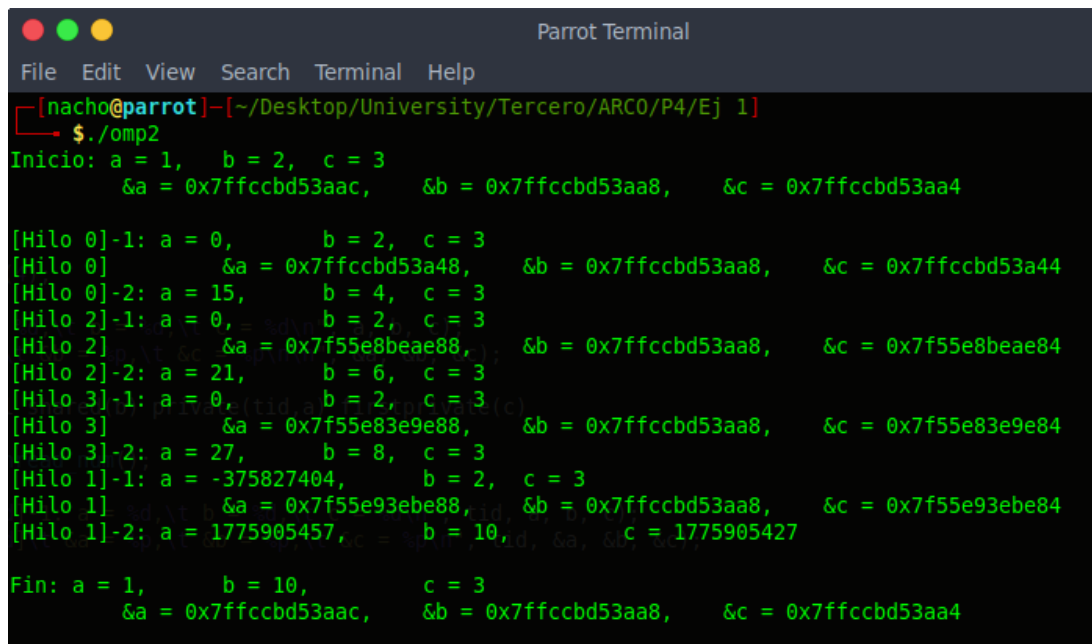
1.2 ¿Cuántos threads debería utilizar en los ordenadores del laboratorio? ¿y en su propio equipo?

En los laboratorios se pueden ejecutar 4 threads en paralelo. En nuestro ordenador, se pueden ejecutar 4 threads en 2 cores, por lo que sí admite un hyperthreading de dos threads por core.



```
Parrot Terminal
File Edit View Search Terminal Help
[nacho@parrot]~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 69
model name     : Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz
stepping       : 1
microcode      : 0x20
cpu MHz        : 2294.480
cache size     : 3072 KB
physical id    : 0
siblings       : 4
core id        : 0
cpu cores      : 2
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 13
wp             : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx p
scp lm constant tsc arch perfmon pebs bts rep good nopl xtopology nons
```

A continuación, ejecutamos el segundo programa y obtenemos la siguiente salida:



```
Parrot Terminal
File Edit View Search Terminal Help
[nacho@parrot]-[~/Desktop/University/Tercero/ARC0/P4/Ej 1]
$ ./omp2
Inicio: a = 1,    b = 2,    c = 3
      &a = 0x7ffccbd53aac,    &b = 0x7ffccbd53aa8,    &c = 0x7ffccbd53aa4

[Hilo 0]-1: a = 0,    b = 2,    c = 3
[Hilo 0]    &a = 0x7ffccbd53a48,    &b = 0x7ffccbd53aa8,    &c = 0x7ffccbd53a44
[Hilo 0]-2: a = 15,   b = 4,    c = 3
[Hilo 2]-1: a = 0,    b = 2,    c = 3
[Hilo 2]    &a = 0x7f55e8beae88,    &b = 0x7ffccbd53aa8,    &c = 0x7f55e8beae84
[Hilo 2]-2: a = 21,   b = 6,    c = 3
[Hilo 3]-1: a = 0,    b = 2,    c = 3
[Hilo 3]    &a = 0x7f55e83e9e88,    &b = 0x7ffccbd53aa8,    &c = 0x7f55e83e9e84
[Hilo 3]-2: a = 27,   b = 8,    c = 3
[Hilo 1]-1: a = -375827404,    b = 2,    c = 3
[Hilo 1]    &a = 0x7f55e93ebe88,    &b = 0x7ffccbd53aa8,    &c = 0x7f55e93ebe84
[Hilo 1]-2: a = 1775905457,    b = 10,    c = 1775905427

Fin: a = 1,    b = 10,    c = 3
     &a = 0x7ffccbd53aac,    &b = 0x7ffccbd53aa8,    &c = 0x7ffccbd53aa4
```

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

Al declarar una variable privada en el pragma, ésta es única para cada hilo. Se le asigna un valor aleatorio dentro del pragma (no mantiene el contenido previo) y no se va a almacenar su contenido tras salir de la región de paralelización. El programa *omp2.c* declara *a* como una variable privada del pragma, por lo que podemos ver que cada hilo le asigna a la variable una posición de memoria diferente, pero al finalizar, ésta mantiene la posición de memoria inicial, con el mismo valor que se le asigna (*a=1*).

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

Como hemos explicado en el apartado anterior, al entrar en la región paralela se le asigna una nueva posición de memoria a la variable privada, por lo que su valor es desconocido (el que tuviese esa posición de memoria previamente), y no el que le habíamos asignado.

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

La variable vuelve a recuperar la posición que se le asigna al inicializarse, por lo que vuelve a recuperar el valor previo al pragma.

1.6 ¿Ocurre lo mismo con las variables públicas?

En el caso de las variables públicas, la posición de memoria no se altera, por lo que si un hilo modifica su contenido este cambio se va a reflejar al salir del pragma. Si lo modifican varios hilos, se va a mantener el valor de la última modificación.

Ejercicio 2: Paralelizar el producto escalar

En este ejercicio vamos a trabajar con la paralelización del producto escalar. Para ello contamos con tres programas: *pescalar_serie.c*, *pescalar_par1.c* y *pescalar_par2.c*

Tras ejecutarlos, contestamos a las siguientes preguntas:

2.1 ¿En qué caso es correcto el resultado?

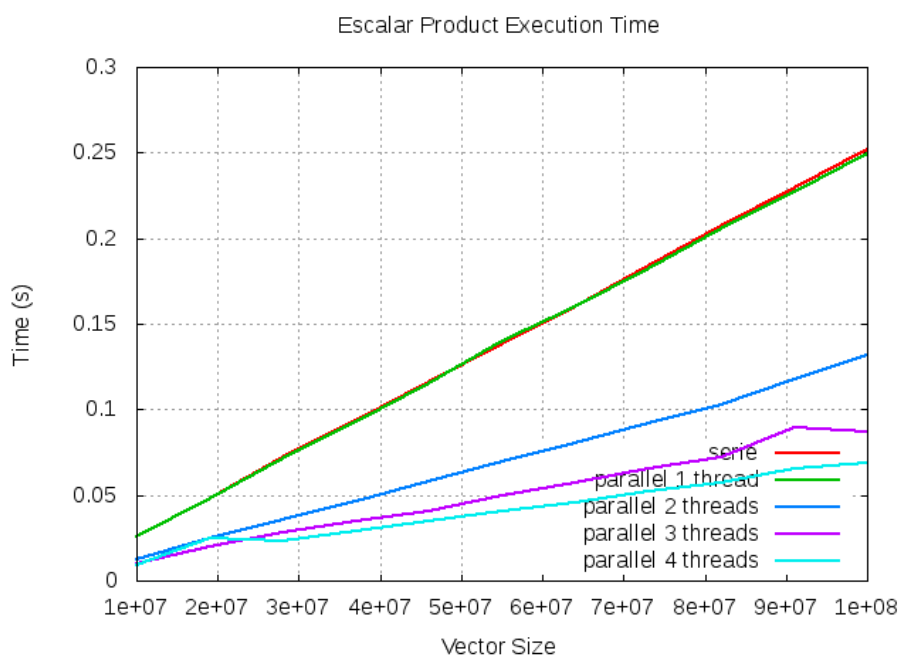
El resultado es correcto en par2, ya que es igual que el obtenido en el procesamiento en serie (que sabemos que es válido).

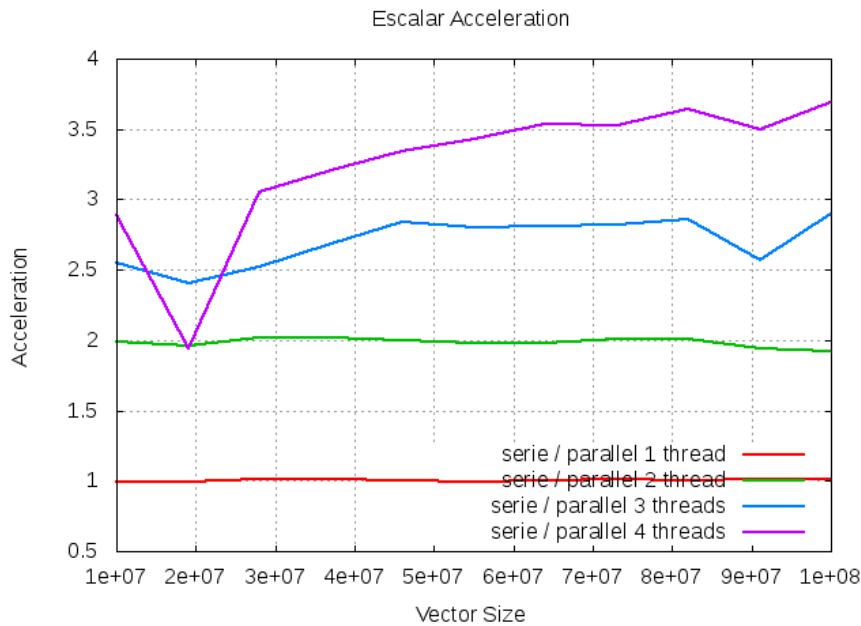
2.2 ¿A qué se debe esta diferencia?

Esto se debe a que par1 emplea la variable sum de forma compartida entre los threads del bucle, por lo que se dan situaciones en la que dos threads cogen el mismo valor inicial de sum, y lo operan sobre el mismo sum, por lo que uno de ellos va a sobrescribir el resultado del anterior. Nos encontramos ante un fallo del tipo WAW (Write After Write).

Los resultados obtenidos con los ficheros originales no eran exactamente iguales entre los programas en serie y en paralelo(2). Esto se debía a que los datos los tratábamos con precisión float, y al cambiar el orden de cálculo mediante threading, aunque debería dar el mismo resultado, se truncaba de distinta forma. Por ello cambiamos el tipo de la variable sum para que almacenase doubles, evitando así los problemas derivados del truncamiento.

Fijándonos ahora únicamente en el programa en serie, y en la paralelización correcta, desarrollamos un script que haga una gráfica de tiempos de ejecución:





A continuación contestamos a las preguntas siguientes:

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?

Podemos apreciar que para vectores pequeños (Menores de 20000000) crear 4 hilos ralentiza el tiempo de ejecución respecto a crear 2 hilos. Esto se puede deber a que la creación de los hilos es más lenta que la ejecución en sí. Por tanto, pensamos que si el trabajo a realizar es de un tiempo menor al que tardaríamos en lanzar los hilos no valdrá la pena.

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

Creemos que en caso de que tengas un procesador *mononúcleo* y sin *hyperthreading*, no compensaría ya que todo el trabajo lo realizaría el mismo núcleo. Como la creación de los hilos requiere de cierto tiempo, lo que haremos es ralentizarlo.

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

Tal y como hemos dicho en los apartados anteriores, el rendimiento para vectores muy grandes sí mejora significativamente al aumentar el número de *threads*. Para tamaños menores creemos que el rendimiento es muy similar. Por ello, en caso de un proyecto real, estaremos gastando tiempo de producción para paralelizar el programa sin tener un rendimiento mayor.

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

Esto se debe a el tiempo tardado en crear los *threads* (y cerrarlos) o debido a la arquitectura del ordenador, donde con un solo núcleo no vamos a paralelizar de forma efectiva.

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

Por las razones que hemos dado antes, pensamos que en vectores pequeños, la gráfica de la aceleración cambiará, pudiendo mostrar una aceleración menor que 1 para los paralelos.

Ejercicio 3: Paralelizando la multiplicación de matrices

En este ejercicio vamos a estudiar la paralelización usando los programas de multiplicación de matrices desarrollados en prácticas anteriores. Elegimos el programa de multiplicación de traspuestas, ya que el más eficiente.

Se nos pide alterar la región de paralelización, de modo que:

- Bucle 1 se corresponde con la paralelización del bucle externo
- Bucle 2 se corresponde con la paralelización del bucle intermedio
- Bucle 3 se corresponde con la paralelización del bucle interno

Al ejecutar los programas, obtenemos los siguientes resultados (con matrices de tamaño 3000)

Tabla de tiempos

Version/Hilos	1	2	3	4
Serie	148.551018			
Bucle 1	142.015291	293.591228	264.627745	364.192596
Bucle 2	142.189859	103.969002	315.717396	304.656156
Bucle 3	152.904785	84.120909	60.631733	56.547607

Tabla de speedup

Version/Hilos	1	2	3	4
Serie	1			
Bucle 1	1.04602	0.505979	0.561358	0.407891
Bucle 2	1.04474	1.4288	0.470519	0.487602
Bucle 3	0.971526	1.76592	2.45005	2.62701

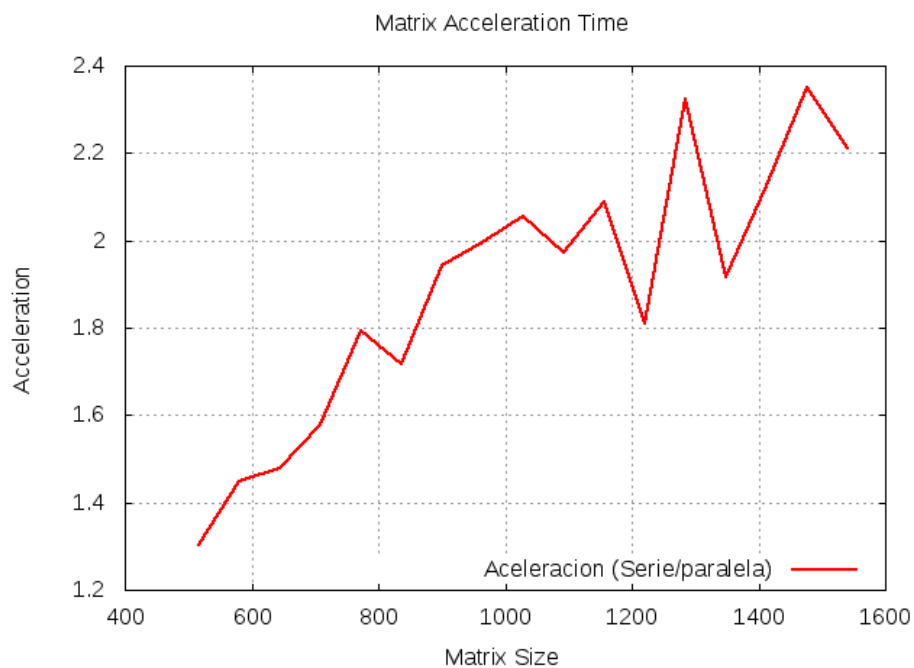
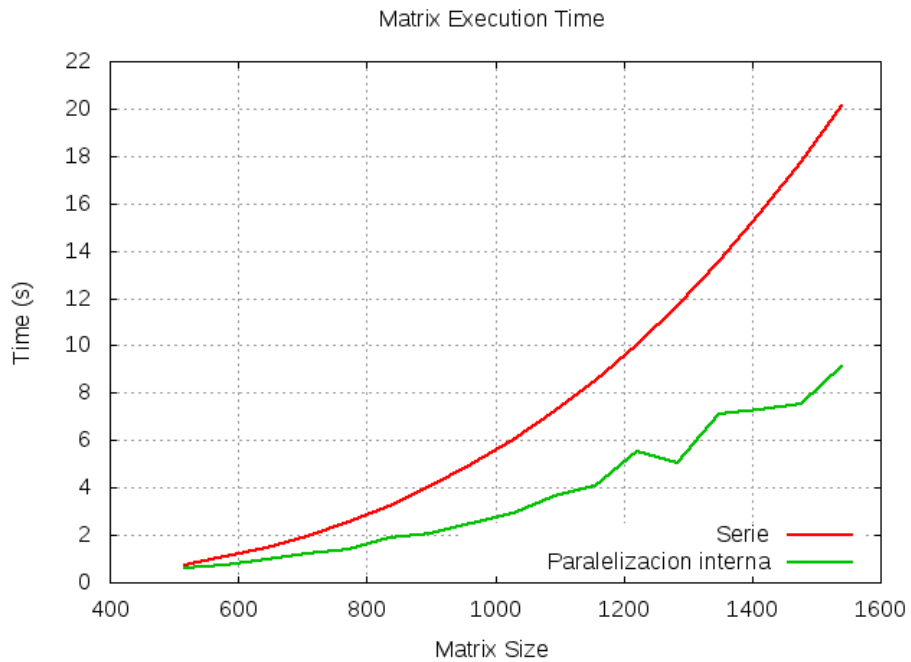
Podemos apreciar que la forma más eficiente de paralelizar este programa es creando la región pragma entorno al bucle más interno, alcanzando su mayor eficacia con 4 *threads*.

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

El peor rendimiento se obtiene al paralelizar el bucle externo. Esto se debe a que cada hilo va a tratar de realizar cálculos sobre los mismos campos de la matriz, produciéndose un *false sharing* que va a ralentizar notablemente la ejecución.

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

El mejor rendimiento se obtiene al paralelizar el bucle interno. Esto se debe a que es el bucle que realiza los cálculos como tal, de forma que si lo paralelizamos vamos a ahorrar mucho tiempo de ejecución, ya que cada hilo realiza sus cálculos por separado y luego se unen.



3.3 Si en la gráfica anterior no obtuvo un comportamiento de la aceleración en función de N que se establezca 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 qué valor de N se empieza a ver el cambio de tendencia.

Se puede ver que a partir del N=1200 se producen cambios de tendencia, que indican una posible estabilización de la gráfica.

Ejercicio 4: Ejemplo de integración numérica

En este apartado vamos a trabajar con una serie de programas que nos han dado. Todos ellos sirven para aproximar π de forma numérica, mediante el cálculo de áreas bajo la gráfica de una ecuación. A mayor número de áreas, mayor precisión obtendremos.

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

Usa 100000000 rectángulos.

4.2 ¿Qué diferencias observa entre estas dos versiones?

El programa *pi_par1* utiliza la variable *sum* (*shared*) para almacenar los datos de la suma, mientras que *pi_par4* los almacena en una variable privada, y sólo actualiza *sum* cuando termina el cálculo. Sin embargo, como *pi_par1* utiliza *sum[tid]*, siendo *tid* una variable privada y única de cada *thread* (está ligada al *thread_id*), no se va a producir *false sharing*, ya que no se va a acceder al mismo sector del array *sum* desde distintos hilos. Sin embargo, como cada hilo tiene que apoderarse del array múltiples veces para poder acceder a la posición correspondiente y actualizarla, va a ser más lento que *pi_par4*.

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?

```
oscar@ubuntu:~/University/Tercero/ARCO/P4$ ./pi_par1
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.227200
oscar@ubuntu:~/University/Tercero/ARCO/P4$ ./pi_par4
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.134025
```

Los resultados son los mismos, pero tarda menos *pi_par4* (como ya hemos explicado)

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?

En este caso, *pi_par2* emplea la variable *sum* como *firstprivate*, por lo que va a mantener el valor asignado antes del *pragma*, pero no se van a mantener los cambios realizados durante la región paralela. Sin embargo, *sum* es un puntero, por lo que al guardar datos en la región de memoria a la que apunta, esos datos sí que siguen ahí tras el *pragma*.

En el caso de *pi_par3*, el programa le asigna a cada *thread* un tamaño de memoria equivalente al tamaño de una línea de caché. De este modo, cuando se cargue la región necesaria para escritura, se reducirán el número de accesos a memoria, ya que se cargará la línea entera de una sola vez.

Por ello, el resultado esperado es el mismo en ambos casos, pero mucho más rápido para *pi_par3*.

```
oscar@ubuntu:~/University/Tercero/ARCO/P4$ ./pi_par2
Numero de cores del equipo: 4
Resultado pi: 3.141593
Tiempo 0.211516
oscar@ubuntu:~/University/Tercero/ARCO/P4$ ./pi_par3
Numero de cores del equipo: 4
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.170730
```

En efecto, al ejecutar los programas vemos que *pi_par3* es más rápido que *pi_par2*

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?

En este caso lo que hemos hecho es modificar la línea 32 de pi_par3 de manera que si el programa no recibe ningún argumento use el valor por defecto. Si recibe algún argumento, lo usará como *padding*. De este modo podremos elaborar un script para este apartado, obteniendo la siguiente salida:

```
Padding habitual: 8
Padding: 1 Tiempo de ejecución: 0.201561
Padding: 2 Tiempo de ejecución: 0.31059
Padding: 4 Tiempo de ejecución: 0.237021
Padding: 6 Tiempo de ejecución: 0.137437
Padding: 7 Tiempo de ejecución: 0.136052
Padding: 8 Tiempo de ejecución: 0.143626
Padding: 9 Tiempo de ejecución: 0.135327
Padding: 10 Tiempo de ejecución: 0.143053
Padding: 12 Tiempo de ejecución: 0.135102
```

Podemos observar que, si el *padding* insertado es menor que el habitual, obtenemos un resultado similar a *pi_par1*. Esto se debe a la misma razón: el abuso en los cambios del dato entre cachés. Ahora bien, para valores mayores o iguales al *padding* habitual obtenemos resultados muy similares entre sí, con una aceleración significativa y un tiempo de ejecución similar a *pi_par4*.

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

Para este último apartado nos ayudaremos de las versiones 4, 5, 6 y 7 del programa de aproximación numérica. Tras ejecutarlos, contestamos a las siguientes preguntas:

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?

Al ejecutar las dos versiones para el cálculo de π apreciamos que el valor devuelto por pi_par5 no es el correcto. Pensamos que el problema viene de que la variable *i* se declara en el exterior de la zona paralelizada y toma carácter compartido. Esto provoca una situación de carrera donde varios *threads* intentan acceder a dicha variable simultáneamente, generando un error en el cálculo.

En cuanto al rendimiento cabe destacar que pi_par5 es considerablemente más lento que pi_par4. Pensamos que se debe a que al declarar una sección crítica, los *threads* que llegan allí tienen que comprobar si ya hay otro *thread* ejecutando esa instrucción y esperar, provocando un cuello de botella que ralentiza toda la ejecución.

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?

Ejecutando las versiones 6 y 7 observamos que la última tarda aproximadamente la mitad de tiempo que la otra. Esto se debe a que la directiva *for reduction* optimiza la suma entre los *threads*, creando un árbol binario inverso que suma los valores intermedios en *threads* diferentes.