

Práctica 4

Explotar el Potencial de las Arquitecturas Modernas

Junco de las Heras y Marta Vaquerizo

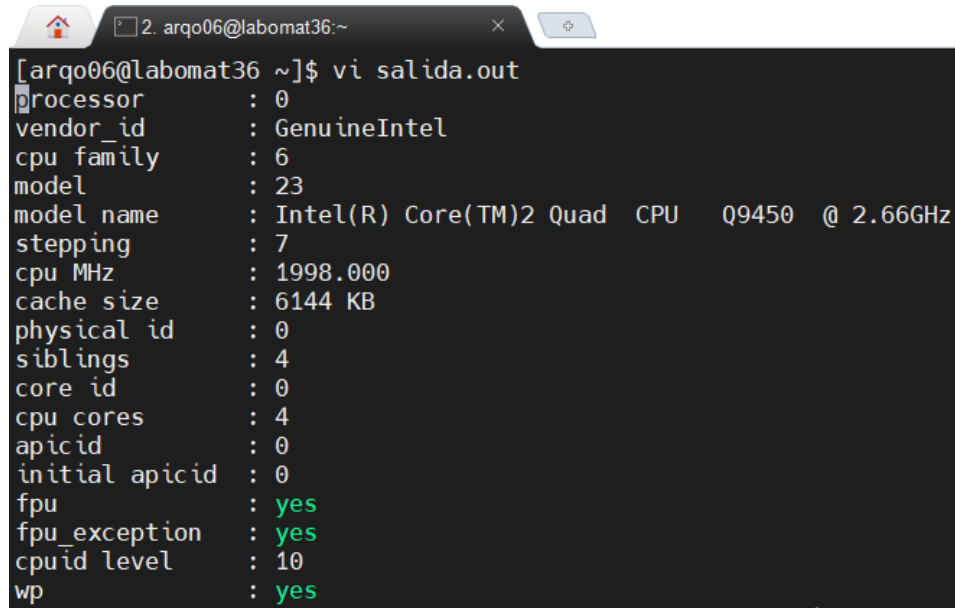
Índice

Ejercicio 0	2
Ejercicio 1	3
Ejercicio 2	6
Ejercicio 3	9
Ejercicio 4	14
Ejercicio 5	17

Ejercicio 0

La captura de pantalla se encuentra en la carpeta `ejercicio0`.

A continuación se muestra la ejecución de `cat /proc/cpuinfo` en el clúster:

A screenshot of a terminal window with a dark background. The window title bar shows a home icon, a tab labeled '2. arqo06@labomat36:~', and a close button. The terminal content shows a user prompt '[arqo06@labomat36 ~]\$' followed by the command 'vi salida.out'. Below this, the output of 'cat /proc/cpuinfo' is displayed in a key-value format. Most values are in white, but 'fpu', 'fpu_exception', and 'wp' are highlighted in green. The output indicates an Intel Core(TM)2 Quad CPU Q9450 with 4 cores and 4 siblings.

```
[arqo06@labomat36 ~]$ vi salida.out
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 23
model name     : Intel(R) Core(TM)2 Quad  CPU   Q9450  @ 2.66GHz
stepping       : 7
cpu MHz        : 1998.000
cache size     : 6144 KB
physical id    : 0
siblings       : 4
core id        : 0
cpu cores      : 4
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 10
wp             : yes
```

Figura 1: Ejecución de `cat /proc/cpuinfo` en el clúster.

En este caso, se puede observar que el campo `cpu cores` tiene el valor 4, y el campo `siblings` tiene el valor 4. Esto indica que no hay htpertreading. Podemos ver también que el nodo que ha ejecutado nuestro script es un nodo Intel (por el número de cores que tiene y por el `vendor_id`).

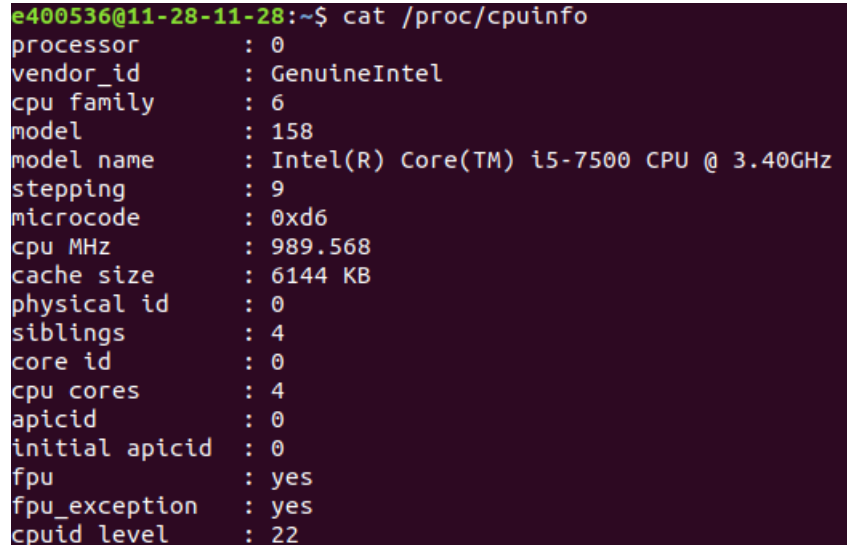
Ejercicio 1

El código y el *makefile*, para este apartado se encuentra en la carpeta `ejercicio1`.

Para todos los ejercicios, nuestro número P es 6, ya que somos la pareja 5 y $5 \% 8 + 1 = 6$.

1. Sí se pueden lanzar más hilos que cores tenga el sistema, pero no tiene sentido hacerlo. Esto se debe a que en cada core se puede ejecutar como máximo un hilo a la vez, entonces si se tiene x cores físicos (y si el hyperthreading está activado, $2 * x$ cores), se podrá ejecutar en paralelo x (o $2 * x$) hilos. Aunque, en un programa de cálculo intensivo, si hay más de estos x hilos, no se producirá el paralelismo buscado, ya que los hilos que sobran tendrían que esperar a que terminase alguno de los otros hilos.
2. El número optimo de hilos a lanzar para un programa de cálculo intensivo es el número de cores físicos, ya que si hubiese más estos hilos tendrían que estarse alternando, no produciendo eficiencia. Para un programa que requiera de entrada y salida, se pueden lanzar más hilos (con ayuda del hyperthreading por ejemplo) ya que cuando un hilo se bloquee otro podrá estarse ejecutando.

A continuación se muestra la ejecución en un ordenador del laboratorio:

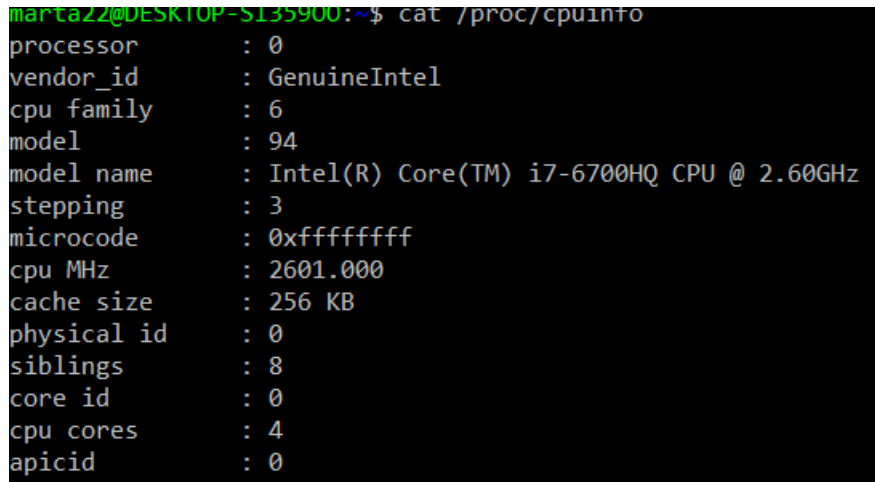


```
e400536@11-28-11-28:~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 158
model name     : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
stepping       : 9
microcode      : 0xd6
cpu MHz        : 989.568
cache size     : 6144 KB
physical id    : 0
siblings       : 4
core id        : 0
cpu cores      : 4
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 22
```

Figura 2: Ejecución de `cat /proc/cpuinfo` en el ordenador del laboratorio.

Se puede observar que el campo `cpu cores`, que se refiere a los cores físicos, tiene el valor 4, y el campo `siblings`, que se refiere a los cores virtuales, tiene el valor 4. Esto indica que no hay posibilidad de hyperthreading. El número óptimo de hilos a lanzar sería 4, el número de cores físicos.

A continuación se muestra la ejecución en un equipo local:



```
marta22@DESKTOP-S135900:~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 94
model name     : Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
stepping       : 3
microcode      : 0xffffffff
cpu MHz        : 2601.000
cache size     : 256 KB
physical id    : 0
siblings       : 8
core id        : 0
cpu cores      : 4
apicid         : 0
```

Figura 3: Ejecución de `cat /proc/cpuinfo` en el ordenador de uno de los integrantes.

Se puede observar que el campo `cpu cores` tiene el valor 4, y el campo `siblings` tiene el valor 8. Esto indica que hay posibilidad de activar el hyperthreading. El número óptimo de hilos a lanzar es 4, el número de cores físicos.

Como hemos comentado en el ejercicio 0, el clúster tiene 4 cores físicos y 4 cores virtuales, así que el número óptimo de hilos a lanzar es 4, el número de cores físicos.

3. Para deducir la prioridad entre las tres formas de elegir el número de hilos, se ha modificado el programa tal que se eligen un par de esas tres formas, a una se le pone el número de hilos que el usuario pasa por pantalla, y a otra ese número más 1. De esta manera se ve cual tiene prioridad sobre la otra en función del número de hilos que se lanzaban en la región paralela. Con estas modificaciones, se llegó a la conclusión que el orden de las prioridades (de mayor a menor) es el siguiente:

- 1) Con la clausula `#pragma omp parallel num_threads(numthr)`
- 2) Con la función `omp_set_num_threads(int num_threads)`
- 3) Con variable de entorno (`OMP_NUM_THREADS`)

Tiene sentido, ya que las variables de entorno son variables generales, que no dependen del `.c` que vayamos a ejecutar, y si en el `.c` en concreto queremos cambiar el número de threads a lanzar se pueda cambiar. Luego dentro del `.c` también pasa lo mismo, la función `omp_set_num_treads` es general y afecta a todo el `.c`, pero si queremos cambiar el número de threads en un bloque en concreto podamos.

4. `OpenMP` distingue dos tipos de variables privadas (`private` y `firstprivate`). En ambas `OpenMP` hace una copia de esa variable a cada hilo, así que si un hilo modifica una variable privada, no va a tener efecto en la misma variable para otro hilo. Cuando acabe la región paralela, las variables privadas contendrán el mismo valor con el que entraron.

5. El valor de la variable **private**, al comenzar la región paralela está indefinido, aunque normalmente toma el valor 0. Compilando con gcc nos salta el warning: -Wuninitialized. El valor de la variable **firstprivate** es el valor que tenía la variable antes de entrar a la región.
6. Al finalizar la región paralela, el valor, tanto de la variable **private** como de la **firstprivate**, es el que tenía antes de entrar en ésta.
7. No, no ocurre lo mismo con las variables públicas, ya que éstas son comunes a todos los hilos, y no cada uno tiene su propia copia. Además, al comenzar la región paralela, ya están inicializadas con el valor que tenía la variable antes de la región, y cualquiera que sea la operación que se realiza con ellas, el resultado será su valor al finalizar la región.

Ejercicio 2

El código y el *makefile*, para este apartado se encuentra en la carpeta `ejercicio2`.

1. El resultado va a ser el tamaño del vector `M`, ya que se trata de la suma de los productos de cada posición de los vectores (del mismo tamaño, `M`), que tienen un 1 en cada posición, por lo que es sumar tantos unos como posiciones tengan los vectores. En `arqo4.h` `M` es una macro definida como 1 millón.
2. El resultado es distinto al obtenido en el serie, y esto no debe ser así, deberían dar el mismo resultado. El problema que tiene el programa `pescalar_par1.c` es que todos los hilos escriben en la variable `sum` y no hay nada que controle el acceso a la variable, lo que provoca que el resultado no sea el mismo (data races).
3. Para este apartado se ha implementado el programa `pescalar_par2.c`.
Sí se puede realizar con ambos pragmas, tanto con `#pragma atomic` como con `#pragma critical`. Y el cambio que se ha realizado (para ambos pragmas) es añadir el pragma antes de la sentencia `sum = sum + A[k] * B[k]`. La diferencia entre ellas es que usando `critical` se puede hacer crítica un bloque entero (varias líneas) mientras que con `atomic` solo se puede hacer crítica una sentencia atómica (una suma en este caso) por lo que si se va a utilizar para solo una suma es mejor ya que está optimizado. Se ha elegido `#pragma atomic`.
4. Para este apartado se ha implementado el programa `pescalar_par3.c`.
Comparando con el punto anterior, la opción elegida será la de este apartado `#pragma omp parallel for reduction`, ya que al ejecutar ambas, dan el mismo resultado, pero la elegida tarda en ejecutar bastante menos que la del apartado anterior (de hasta 1 orden de magnitud). Esto es debido a que usando una región crítica, solo un hilo puede ejecutar la suma, mientras que usando `reduction` varios pueden ejecutar la suma, combinando el resultado con la operación puesta en `reduction`, siendo más paralelo que en el `pescalar_par2.c`.
5. Para este apartado, se ha implementado el programa `pescalar_par4.c`, que es el programa `pescalar_par3.c` (se considera que es la mejor opción), pero se ha añadido que se le pueda pasar los argumentos: tamaño del vector y el número de hilos a lanzar. En la región crítica se ha añadido una cláusula `tamano > MIN_TAMANIO`, la cual solo paraleliza en caso de que el tamaño del vector sea mayor que `MIN_TAMANIO`, en nuestro caso una constante con valor a 1000. Esto optimiza el caso de cuando el vector tiene un tamaño pequeño, ya que inicializar los hilos tiene un coste, no despreciable cuando el tamaño del vector es pequeño.

A partir de este programa, se han creado dos scripts:

- `pescalar.sh`
- `pescalar_graphics.sh`

El primer script, calcula los tiempos de ejecución al ejecutar el programa `pescalar_par4.c` variando el tamaño de vector entre 300000000 y 1500000000, con un salto de 300000000 unidades, y variando el número de hilos entre 1 y $2 \cdot C$ de una unidad en una unidad, donde C es el número de cores del clúster, ya que es donde se va a ejecutar, por lo que $C = 4$, ya que el hyperthreading no está activado.

El segundo script se encarga de generar las siguientes dos gráficas:

- *pescalar_serie_graphic.png*
- *pescalar_par_graphic.png*

la primera es la gráfica de los tiempos de ejecución del programa `pescalar_serie.c`, y la segunda es la gráfica de los tiempos del programa `pescalar_par4.c` variando el número de hilos de 1 a $2 \cdot C$, donde $C = 4$ (por lo que hay 8 líneas en la gráfica). A continuación se muestran las gráficas:

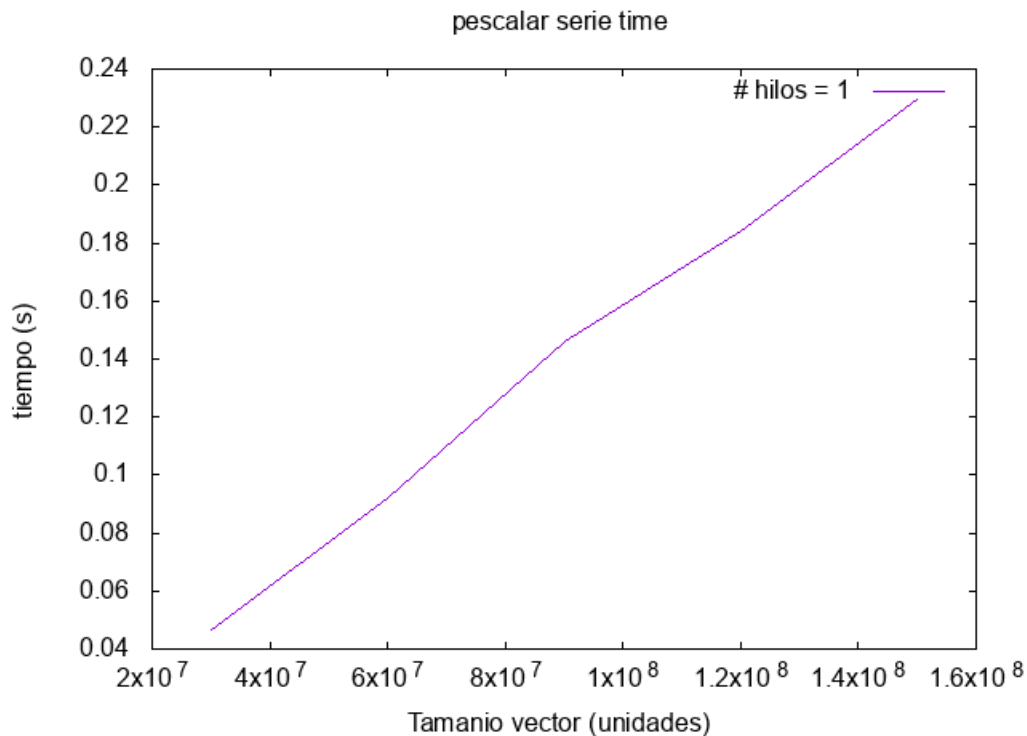


Figura 4: `pescalar_serie_graphic.png`

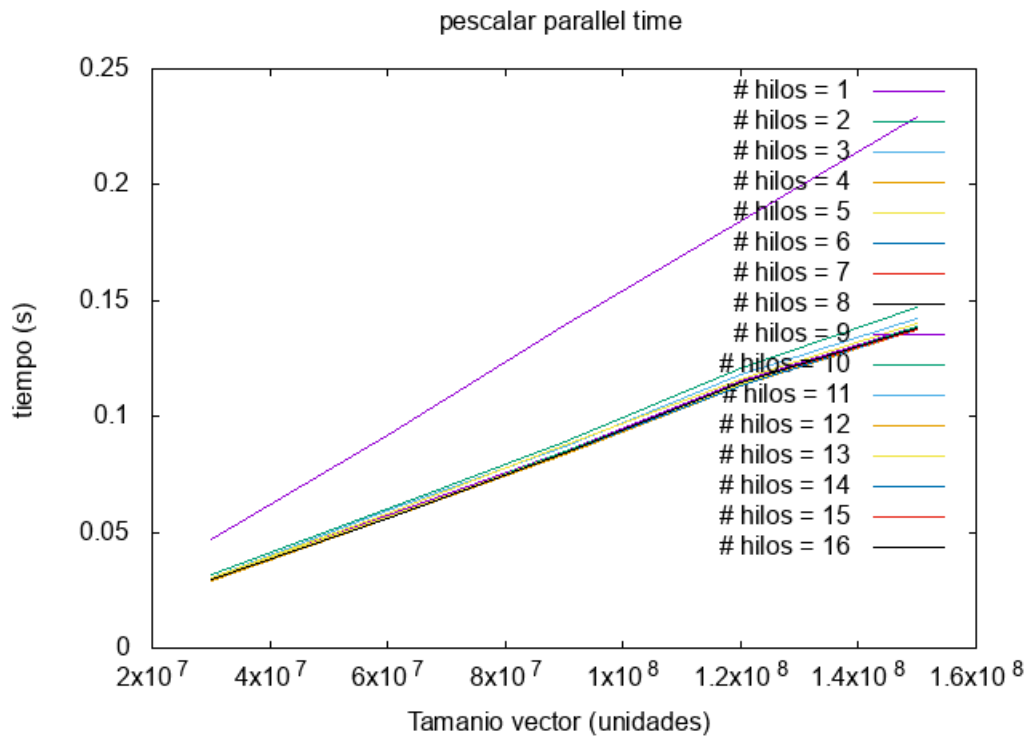


Figura 5: `pescalar_par_graphic.png`

Se puede ver que la gráfica `pescalar_serie_graphic.png` es la misma que la gráfica `pescalar_par_graphic.png` al ejecutar el programa `pescalar_par4.c` con 1 hilo de ejecución en la región. Esto es lo que se esperaba, ya que en el programa `pescalar_serie.c` solo hay 1 hilo de ejecución.

En nuestro caso hemos ejecutado con tamaños de vectores muy grandes (más de 1 millón de elementos), y es un programa de cálculo intensivo. Sabemos que compensa lanzar hilos siempre que no supere el número físico de cores, una vez superado (y como podemos ver en la gráfica) la mejora casi no es apreciable. Para tamaños muy pequeños (menor que 1000 elementos, por ejemplo), el coste de lanzar un hilo implica una carga de reserva de recursos nada despreciable, lo que implica un aumento del tiempo de ejecución considerable frente al tiempo de ejecución del algoritmo para un solo hilo, así que para tamaños menores que 1000 sería mejor usar la versión serie.

Podemos comprobar con la gráfica y con la ayuda de la terminal que hemos ejecutado el script, que hay 3 gráficas (con 1, 2, y 3 hilos) separadas, y luego a partir de 4 hilos están todas juntas. Esto es debido a que se han ejecutado en un solo procesador que tenía 4 cores físicos, y a partir del cuarto hilo ya no se nota mejora.

Ejercicio 3

El código y el *makefile*, para este apartado se encuentra en la carpeta `ejercicio3`.

En este ejercicio se han implementado los siguientes cuatro programas:

- *multiplication_slow.c* (de la práctica anterior)
- *multiplication_slow_par1.c* (paralelización del bucle más interno)
- *multiplication_slow_par2.c* (paralelización del bucle intermedio)
- *multiplication_slow_par3.c* (paralelización del bucle más externo)

Para poder completar las tablas que se piden, y automatizar el proceso, se ha creado un script *tablas.sh*, que calcula los tiempos de ejecución para los cuatro programas, y las aceleraciones, tomando como referencia la versión serie. Los datos se guardarán en un fichero *multiplication_time_N.dat* y *multiplication_speed_N.dat*, donde N es el tamaño de matriz.

TABLAS:

Para un tamaño de matriz $N = 1000$:

<div># hilos</div> <div>Versión</div>	1	2	3	4
Serie	12,850608	12,850608	12,850608	12,850608
Paralela-bucle1	10,841369	7,282123	7,995398	5,201010
Paralela-bucle2	15,213245	6,980360	8,409289	3,547559
Paralela-bucle3	14,201083	9,175650	6,908171	3,785163

Cuadro 1: **Tiempos de ejecución (s).**

<div># hilos</div> <div>Versión</div>	1	2	3	4
Serie	1	1	1	1
Paralela-bucle1	1,185330	1,764678	1,607250	2,470790
Paralela-bucle2	0,844698	1,840966	1,528144	3,622380
Paralela-bucle3	0,904903	1,400512	1,860204	3,394994

Cuadro 2: **Speedup.**

Para un tamaño de matriz $N = 2000$:

Versión \ # hilos	1	2	3	4
Serie	118,018862	118,018862	118,018862	118,018862
Paralela-bucle1	108,188828	64,791838	47,105276	39,925859
Paralela-bucle2	174,017335	92,025022	61,228575	45,905626
Paralela-bucle3	160,149591	78,053119	53,276837	40,707765

Cuadro 3: **Tiempos de ejecución (s).**

Versión \ # hilos	1	2	3	4
Serie	1	1	1	1
Paralela-bucle1	1,090859	1,821508	2,505427	2,955950
Paralela-bucle2	0,678201	1,282464	1,927512	2,570901
Paralela-bucle3	0,736928	1,512032	2,215200	2,899173

Cuadro 4: **Speedup.**

1. Como se puede observar en las tablas, para el tamaño de matriz $N = 1000$, la versión con peor rendimiento está claro que es la de la paralelización del bucle más interno, pero la versión con mejor rendimiento no se ve claro cuál es para este tamaño de matriz. Ahora bien, si se observan las tablas para el tamaño de matriz $N = 2000$, la versión con peor rendimiento es la misma que para el tamaño $N = 1000$, pero la que mejor rendimiento tiene es la de la paralelización del bucle más externo, superando a la versión de la paralelización del bucle intermedio en todas las ejecuciones. Con estos datos, se puede concluir que la versión con peor rendimiento es la del bucle interno, y la de mejor rendimiento es la del bucle externo.

Estos resultados se deben a que en la paralelización del bucle más interno, se tiene que crear la región paralela $N * N$ veces, por lo tanto los hilos, siendo N el número de iteraciones tanto del bucle externo como del intermedio. Además, debido a la sincronización de los hilos en la región, el programa tiene que esperar a que todos los hilos acaben hasta la siguiente iteración.

Por otro lado, en la paralelización del bucle más externo, la región paralela, y por tanto los hilos, solo se crean una vez, y los hilos no tienen que esperar a que acaben los otros hilos para poder hacer las iteraciones más internas (de los bucles intermedio e interno). Se puede destacar que, en la paralelización del bucle intermedio, la región paralela, y por tanto los hilos, se crean N veces, que son bastantes menos que en la del bucle interno, pero más que en la del bucle externo, por lo que su rendimiento, si el tamaño de matriz es suficientemente grande, es intermedio respecto a los rendimientos de las otras dos versiones, pero más cercano al rendimiento de la versión del bucle exterior.

2. En base a los resultados, se puede concluir que, en general, la mejor paralelización va a ser la de grano grueso, cuando se trata de paralelizar bucles anidados, debido a lo comentado en el apartado anterior. El hecho de que en la paralelización de grano fino se tenga que crear la región paralela, y por tanto los hilos, tantas veces como iteraciones haya en los bucles más externos, así como que los hilos tengan que esperar a que acaben todos antes de hacer la siguiente iteración, hace que su rendimiento disminuya lo suficiente para que la paralelización de grano grueso tenga mejor rendimiento.
3. Para este apartado, se ha creado el script *multiplication.sh*, que calcula los tiempos de ejecución de los programas `multiplication_slow.c` y `multiplication_slow_par3.c`, y la aceleración (*speedup*) entre ambos tiempos de ejecución $\frac{T_{TiempoSerie}}{T_{TiempoParalelo}}$ para tamaños de matriz N entre 1 y $124+248 \cdot P$, y 4 hilos a lanzar en la región paralela en el programa `multiplication_slow_par3.c`, ya que en las tablas se ve que es la mejor paralelización.

Este script genera las siguientes dos gráficas a partir de los datos que calcula anteriormente:

- *multiplication_time.png*
- *multiplication_speed.png*

A continuación se mostrarán las gráficas:

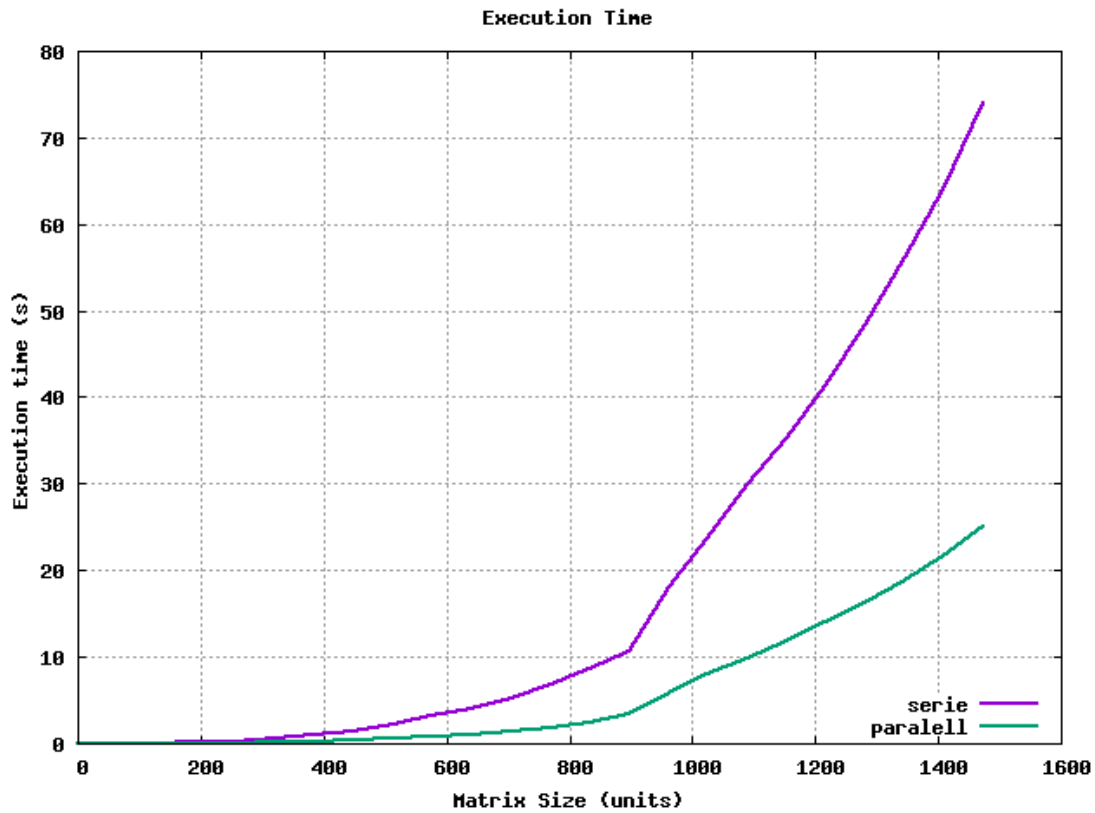


Figura 6: `multiplication_time.png`

Se puede observar en la gráfica que para tamaños de matriz relativamente pequeños los tiempos de ejecución son bastante parecidos para ambas versiones (serie y paralelo), pero a medida que el tamaño de matriz aumenta, la diferencia de tiempos es significativa. Mientras que con el programa `multiplication_slow_par3.c`, con 4 hilos en la región, el tiempo de ejecución más alto, es casi los 30 segundos, con el programa `multiplication_slow.c` es casi a los 80 segundos, hay una diferencia de casi 50 segundos para un tamaño de matriz suficientemente grande.

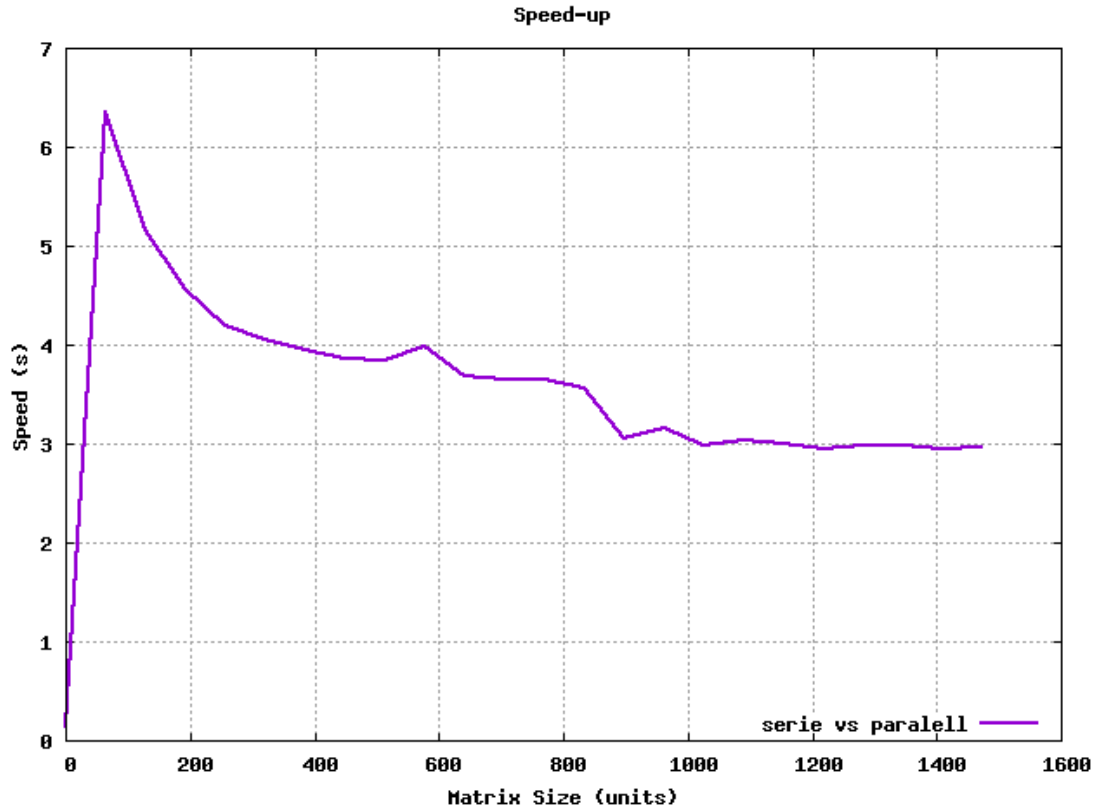


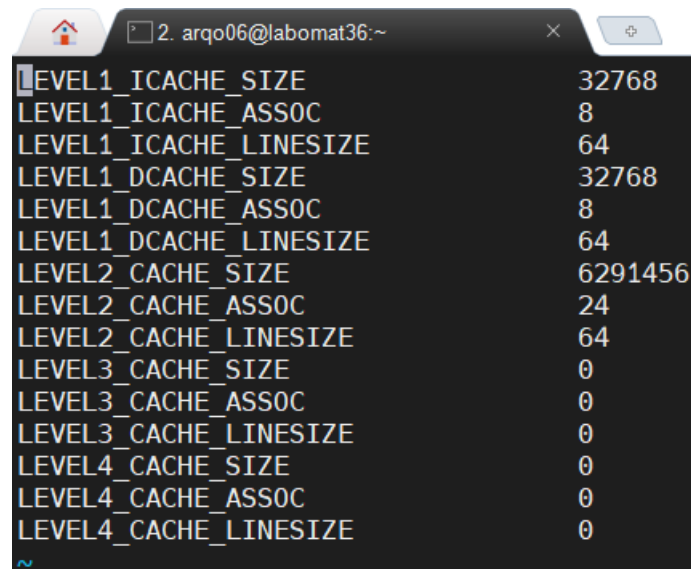
Figura 7: multiplication_speed.png

En esta segunda gráfica, se puede observar que para un tamaño de matriz muy pequeños, hay una “gran” aceleración ($\sim 6,3$), y va disminuyendo, a medida que se aumenta el tamaño de la matriz, hasta que se estabiliza en una aceleración de 3, aproximadamente, a partir del tamaño de matriz $N = 1000$.

Ejercicio 4

El código y el *makefile*, para este apartado se encuentra en la carpeta `ejercicio4`.

Para este ejercicio hay que tener en cuenta que en cada bloque de caché en el clúster, cuyo tamaño es 64 Bytes, puede haber hasta $\frac{64B}{8B} = 8$ elementos de tipo `double`, sabiendo que un elemento de tipo `double` ocupa 8 Bytes; y que el clúster tiene 4 cores, en el equipo en el que se ha ejecutado este ejercicio, ya que el hyperthreading está desactivado.



```
2. arqo06@labomat36:~  
LEVEL1_ICACHE_SIZE      32768  
LEVEL1_ICACHE_ASSOC      8  
LEVEL1_ICACHE_LINESIZE   64  
LEVEL1_DCACHE_SIZE      32768  
LEVEL1_DCACHE_ASSOC      8  
LEVEL1_DCACHE_LINESIZE   64  
LEVEL2_CACHE_SIZE       6291456  
LEVEL2_CACHE_ASSOC      24  
LEVEL2_CACHE_LINESIZE   64  
LEVEL3_CACHE_SIZE        0  
LEVEL3_CACHE_ASSOC        0  
LEVEL3_CACHE_LINESIZE    0  
LEVEL4_CACHE_SIZE        0  
LEVEL4_CACHE_ASSOC        0  
LEVEL4_CACHE_LINESIZE    0
```

Figura 8: Información sobre la caché de un equipo del clúster

Se ha ejecutado en el clúster el comando `getconf -a | grep cache`, y ha salido la información anterior. Se observa que el tamaño de bloque en cualquier caché (independientemente del nivel) es 64 Bytes. Se va a usar esta información para justificar los apartados de este ejercicio, ya que este ejercicio se ha ejecutado en ese equipo del clúster.

1. En el programa `pi_serie.c` se utilizan 100000000 rectángulos (lo que se llama `n` en el programa). Esto se sabe ya que el intervalo $[0, 1]$, donde se quiere integrar, se divide por `n`, para sacar la anchura de los rectángulos. Además, `n` es el número total de iteraciones en el bucle `for`, equivaliendo una iteración por rectángulo.
2. La diferencia entre los programas `pi_par1.c` y `pi_par4.c` es que en el primer programa, en la región paralela, se realiza la suma de cada hilo accediendo a la variable compartida `sum`, mientras que en el segundo programa, se declara una variable dentro de la región, y la suma se realiza con esa variable, y luego se guarda en la posición del número de hilo en la variable compartida `sum`. Esto provoca que en el primer programa vaya a haber muchos más fallos de caché que en el segundo, debido al efecto de *false sharing*.

3. Al ejecutar ambas versiones (`pi_par1.c` y `pi_par4.c`), el resultado es el mismo, pero el primer programa tarda más que el segundo. Esta diferencia de rendimiento se debe al *false sharing*. En `pi_par1.c`, cada hilo que se lanza en la región paralela accede a la variable compartida `sum` (con false sharing) $1 + \frac{n-tid}{numThreads}$ veces, donde el *tid* es el número de hilo, y el *numThreads*, el número de hilos que se han lanzado; mientras que en `pi_par4.c`, cada hilo accede una única vez a esta variable compartida.
4. Al ejecutar ambas versiones (`pi_par2.c` y `pi_par3.c`), el resultado es el mismo, pero el primer programa tarda más que el segundo. Esto es debido a que en el segundo programa cada hilo tiene su propia línea de caché, lo que implica que el efecto *false sharing* desaparece, mientras que en el primer programa dos hilos pueden tener posiciones de memoria en la misma línea de caché, así que cada vez que un hilo escriba, le producirá un *false sharing* al otro.
5. Al ejecutar el programa `pi_par3.c` con un `padding` de valores: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, y 12, se observa que el tiempo de ejecución va disminuyendo a medida que se aumenta el valor del `padding`. Esto ocurre debido a que cuando se aumenta el `padding`, el vector `sum` reserva memoria para `nprocs*padding` elementos, donde `nprocs` es el número de cores del equipo, y las sumas de cada hilo se guardan en la posición `tid*padding`, donde `tid` es el número de hilo. Esto produce que cada vez haya menos sumas de hilos por bloque en la caché, lo que reduce el efecto de *false sharing*, hasta que casi desaparece, cuando el `padding` es lo suficientemente grande para que en cada bloque de caché solo haya una suma de algún hilo.

Para el caso en el que se ejecuta el programa en el clúster, teniendo en cuenta los datos sobre el clúster mencionados al principio del ejercicio, se puede sacar cuántos bloques son necesarios para albergar el vector `sum`, y cuántos elementos hay en cada bloque. Para un valor `x` de `padding`, el número de bloques para guardar el vector `sum` es $\lceil \frac{x*4}{8} \rceil$, donde 4 es el número de cores del equipo donde se está ejecutando, y la cantidad de elementos por bloque es $\lceil \frac{8}{x} \rceil$. Para los valores mencionados anteriormente:

Padding	1	2	3	4	5	6	7	8	9	10	11	12
# Bloques para guardar el vector	1	1	2	2	3	3	4	4	5	5	6	6
# Elementos por bloque	8	4	3	2	2	2	2	1	1	1	1	1

Cuadro 5: Tabla de valores.

6. Al usar *critical* en el programa `pi_par5.c`, solo un hilo puede acceder a la vez a la variable `pi`, esto produce que mientras que un hilo accede a la variable, los demás están esperando a acceder a ella, que disminuye el rendimiento, pero, por otra parte, en el programa `pi_par4.c`, se produce el efecto *false sharing*, que también disminuye el rendimiento, por lo que se “compensan”, y es por ellos que tienen más o menos el mismo rendimiento.
7. En `pi_par6.c`, las directivas utilizadas son: `#pragma omp parallel default(shared) private(numThreads)` y dentro de ésta `#pragma omp for`. Estas directivas tienen el siguiente efecto: cuando el hilo máster llega a la primera de ellas, lanza tanto hilos como se le haya indicado anteriormente, de manera paralela, y cuando pasan por la segunda directiva, a cada uno de ellos se le asigna un número de iteraciones del `for` que hay tras la segunda directiva.

Por otro lado, se tiene el programa `pi_par7.c`, en el que se usa una sola directiva: `#pragma omp parallel for reduction(+: sum) private(i,x) default(shared)`. Con esta directiva, se lanzan tantos hilos como se le haya indicado anteriormente, asignándoles iteraciones del `for` que sigue a la directiva. Además, en la variable `sum`, mediante reducción, se van sumando los distintos resultados de cada uno de los hilos.

Se aprecia que el primer program (`pi_par6.c`) tarda más que el segundo (`pi_par7.c`), y esto se debe a que en el segundo programa se utiliza la cláusula `reduction(+: sum)`, que agiliza el proceso de cálculo. Además, el primero tiene *false sharing*, lo que puede producir un aumento del tiempo de ejecución, debido a que se tienen que escribir en memoria bloques a los que un hilo quiere acceder, después de que otro hilo haya accedido.

Ejercicio 5

El código y el *makefile*, para este apartado se encuentra en la carpeta `ejercicio5`. En esta carpeta también se han incluido las imágenes para las pruebas.

1. Este no es el bucle óptimo para paralelizar en el programa. Es mejor ir procesando una imagen por vez, y por cada imagen paralelizar alguno de los bucles que tiene el algoritmo, a procesar varias imágenes a la vez.
 - a) Si se pasan menos argumentos que número de cores, entonces como mucho va a haber un número de hilos igual al número de argumentos de entrada ejecutándose, por lo que habrá algún core que no esté ejecutando nada, perdiendo productividad.
 - b) Esta es otra de las desventajas de paralelizar el bucle de los argumentos de entrada. Como los hilos tienen memoria compartida, y cada hilo tiene que procesar una imagen, cada hilo necesita tener un espacio de *6GB* para poder tener en memoria toda la imagen, por lo que va a haber varias imágenes cargadas en memoria principal y eso ocupa mucho espacio, en vez de ir procesando imagen por imagen ocupando como mucho hasta *6GB*.
2. Sí, se han encontrado bucles que no accedían de forma óptima.

El primero es el bucle 1:

```
for (int i = 0; i < width; i++)  
    for (int j = 0; j < height; j++)  
        grey_image[j * width + i].
```

Se puede ver que está accediendo por columnas, pero se modifican cambiando el orden de iteración, primero iterando sobre *j* y luego sobre *i*.

Una cosa similar ocurre con el bucle 4, primero se itera sobre la variable *p1* y luego la variable *p2*, pero se accede a la posición `kernel[p1 + p2 * (2 * radius + 1)]`, es decir, que se itera sobre las columnas. La modificación para este bucle es cambiar el orden de iteración, primero sobre *p2* y luego sobre *p1*.

- a) Como lo de dentro de los bucles que hemos intercambiado el orden, no dependía del orden, el resultado sigue siendo correcto. Aún así hemos vuelto a ejecutar con todas las imágenes el código para ver que seguía dando el mismo resultado.
- b) Se entiende que preguntan por qué el orden no es correcto buscando la eficiencia (ya que el orden sí es correcto en el sentido que da un resultado correcto). No es correcto debido a que se accede en ambos casos al array por columnas, en el caso de `j * width + i` incrementando la *i* 1 unidad, se accede a la dirección de memoria contigua, pero incrementando la *j* se incrementa la dirección de memoria `width` unidades.

3. Para este apartado, se ha ido probando la paralelización de los distintos bloques de bucles anidados para mejorar el rendimiento, y mientras se hacía, se iba ejecutando el programa, y probando que el resultado fuese el mismo que al principio. Una vez que se han optimizado los distintos bloques, se ha vuelto a generar todas las imágenes para comprobar que funcionase el programa correctamente.

Los distintos bloques paralelizados, se han nombrado en el código del 1 al 4. En nuestro caso, se ha paralelizado en todos los bloques el segundo bucle empezando por el más externo, ya que esta paralelización es la más óptima. Por un lado, sabemos, por el ejercicio 3, que optimizar los bucles más externos proporciona un mejor rendimiento. Y por otro lado, en nuestro caso, las imágenes eran todas horizontales, es decir, que son más anchas que altas, por lo que paralelizar la altura nos da mejores resultados que paralelizar la anchura. Esto se debe a que como tenemos un número finito de hilos, a mayor número de iteraciones, a cada hilo se le asignarán más iteraciones, por lo que tardará más.

Se ha creado un `.c` para el programa con las paralelizaciones y las mejoras de acceso a los datos, se ha llamado `edgeDetector_par.c`.

4. Para este apartado, se ha creado un script, llamado `tablas_edgeDetector.sh`, que genera los datos para completar la tablas que se piden. Este script genera los siguientes ficheros:

- `edgeDetector_time.dat`
- `edgeDetector_speed.dat`

En el primer fichero, se guardan los tiempos de ejecución de ambas versiones del programa para cada imagen; y en el segundo fichero, se guardan las aceleraciones del programa paralelizado para cada imagen respecto a la versión serie.

A la tabla de tiempos de ejecución, se le añade una columna, la tasa de fps de cada programa, que se calcula directamente. Para calcularla, como se calcula el tiempo de 5 imágenes, será $\frac{5}{TiempoTotal}$, siendo el *TiempoTotal* la suma de tiempos de ejecución de las 5 imágenes.

TABLAS:

Imagen Versión	SD	HD	FHD	4k	8k	fps
Serie	0,079054	0,322083	0,735048	3,029175	12,787320	0,294938
Paralela	0,022671	0,070197	0,1430437	0,517378	1,993275	1,822189

Cuadro 6: **Tiempos de ejecución (s).**

Imagen Versión	SD	HD	FHD	4k	8k
Serie	1	1	1	1	1
Paralela	3,487009	4,588273	5,138650	5,854858	6,415231

Cuadro 7: **Speedup.**

Como se puede observar en la tabla de las aceleraciones, al paralelizar los bloques de bucles anidados, se ha mejorado bastante el rendimiento de la versión serie del programa, teniendo aceleraciones del orden de 3, 4, 5 y 6.

Esta mejora también se puede observar en la tabla de los tiempos de ejecución, en la tasa de fps, ya que en la versión serie, la media de *frames per second* es de 0,3 imágenes, es decir, ni una imagen por segundo; mientras que en la versión paralela, el programa puede procesar casi 2 imágenes por segundo.

5. Para este apartado, se ha creado un script, llamado `tablas_edgeDetector_03.sh`, que genera los datos para completar la tablas que se piden.

TABLAS:

Imagen Versión	SD	HD	FHD	4k	8k	fps
Serie	0,039712	0,168779	0,401620	1,612481	7,128492	0,534901
Paralela	0,013984	0,041741	0,076236	0,261074	0,939798	3,751407

Cuadro 8: **Tiempos de ejecución (s).**

Imagen Versión	SD	HD	FHD	4k	8k
Serie	1	1	1	1	1
Paralela	2,839816	4,043482	5,268114	6,176336	7,585132

Cuadro 9: **Speedup.**

Como se puede observar en la tablas, al paralelizar los bloques de bucles anidados y usar en la compilación la bandera `-O3`, los tiempos de ejecución han mejorado respecto a los tiempos de ejecución sin usar la bandera `-O3` en la compilación. También se ha mejorado la tasa fps, respecto a la anterior tabla, en ambas versiones, pero sobre todo se nota en la versión paralela, que, para este apartado, puede procesar casi 4 imágenes por segundo (más o menos el doble que del anterior apartado).

En cuanto a las aceleraciones, para las imágenes que menos pesan, la aceleración en este apartado es menos, pero para las imágenes más pesadas, la aceleración es bastante mayor que en el anterior apartado.

Se ha mirado en el man qué tipo de optimizaciones usaba `-O3`, y se ha comprobado que sí ha implementado algún tipo de vectorización. Está activada en la optimización `-O2`, así que en la `-O3` que es una versión más agresiva que la anterior todavía más. También se utiliza un desenrollado de bucles. Esta técnica es muy efectiva, ya que aunque hace que el ejecutable pese más, disminuye notablemente el número de saltos que tiene que dar.

Compilado con `-O3` la salida sigue siendo la misma, las imágenes generadas son las mismas que sin la bandera, lo que confirma que el programa sigue ejecutándose correctamente.