Arquitectura de Ordendores, Práctica 4:
Explotar el Potencial de las Arquitecturas Modernas

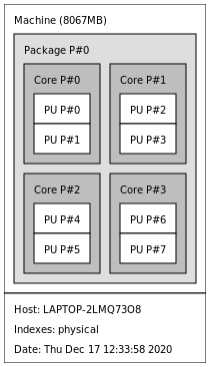
Leandro García y Fabián Gutiérrez (Grupo 1301_08)

18 de diciembre de 2020

En primer lugar, $P = (8 \mod 8) + 1 = 1$ será el número utilizado a lo largo de la práctica.

Ejercicio 0

En la figura 1 se puede observar el diagrama de la arquitectura del equipo generado por el comando `lstopo` y la información del procesador dada por `cat /proc/cpuinfo`. En particular, el equipo en el que se ejecutan las pruebas (que dispone de Windows 10 con el bash de Linux) consta de 4 *cores* con el *hyperthreading* habilitado, lo que da lugar a 8 procesadores lógicos, todos ellos a una frecuencia mínima de $1,8$ GHz. Este procesador cuenta además con las memorias caché mostradas a continuación.




```
leandro@LAPTOP-2LMQ73O8:/mnt/c/Users/leand/Documents/GIT/ARQO_2020/P4$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
stepping        : 10
microcode       : 0xffffffff
cpu MHz         : 1992.000
cache size      : 256 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
```

```
leandro@LAPTOP-2LMQ73O8:~$ getconf -a | grep -i "cache"
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                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Figura 1: Salida de `lstopo` y `cat /proc/cpuinfo` e información de la caché

En ejercicios posteriores se hará uso del clúster de docencia provisto para la medición de tiempos y rendimientos.

Ejercicio 1

1. El programa permite el lanzamiento de más hilos que *cores*. Sin embargo, el objetivo de utilizar OpenMP es lanzar hilos que paralelicen la ejecución del programa, con lo que lanzar más hilos que *cores* logra la misma paralelización que lanzar tantos hilos como *cores*. Por poner un ejemplo trivial, si el procesador del equipo cuenta con un único *core*, lanzar más de un hilo no supone paralelización alguna.

2. Como se discutió anteriormente, el número máximo de hilos que tiene sentido lanzar es igual al número de *cores* del sistema. Los equipos del laboratorio cuentan con 4 *cores*, por lo que tiene sentido lanzar hasta 4 hilos en paralelo. Por otra parte, el clúster cuenta con nodos Intel de 16 *cores* y nodos AMD de 12 *cores*, que se corresponden con un máximo de 16 y 12 hilos en paralelo, respectivamente (luego el máximo si se usa la cola general es de 16 hilos), pues OpenMP trabaja con *cores* que accedan a una misma memoria compartida, por lo que solo puede ejecutarse en un único nodo cada vez. Finalmente, el equipo propio consta de 8 *cores*, así que como máximo se pueden ejecutar 8 hilos en paralelo.

3. Al asignar a la variable de entorno `OMP_NUM_THREADS` el valor 1, ejecutar el programa `omp1.c` con el argumento 2 y añadir en la región paralela la cláusula `num_threads(3)`, se lanzan 3 hilos. Repitiendo el experimento anterior, omitiéndose en este caso la cláusula `num_threads(3)`, se lanzan 2 hilos. Por tanto, la sentencia con mayor prioridad es la cláusula `num_threads`, seguida por la función `omp_set_num_threads` y finalmente la variable de entorno `OMP_NUM_THREADS`.

4. Al declarar una variable privada, tanto con la cláusula `private` como `firstprivate`, OpenMP crea una para cada hilo, es decir, reserva para cada hilo una dirección en la que almacenar su variable privada.

5. Si se usa `private`, la variable no estará inicializada (como es el caso de `a` en el ejemplo), mientras que al usar `firstprivate` la variable creada almacenará el valor que tenía originalmente antes de la paralelización (como es el caso de `c` en el ejemplo).

6. El valor de las variables privadas en los hilos se desecha y se recuperan los valores previos a la paralelización (en el ejemplo, `a` recupera su valor original: 1).

7. A diferencia de las privadas, las variables públicas son comunes a todos los hilos. Por tanto, no se crea una copia por hilo, su valor inicial es el que tuviera antes de la paralelización y su valor final tras la conclusión de la región paralela se conserva. Cabe destacar que las variables públicas pueden ser modificadas por cada hilo, con lo que han de emplearse mecanismos que eviten condiciones de carrera provocadas por las escrituras concurrentes.

Ejercicio 2

2. Se observa que el resultado generado por `pescalar_par1.c` es incorrecto y, además, distinto para cada ejecución. En el código no hay mecanismo alguno que proteja la variable compartida `sum` y las operaciones sobre esta no son atómicas, por lo que muy probablemente se producen lecturas y escrituras concurrentes que resultan en sumar y almacenar valores erróneos.

3. Se pueden evitar las condiciones de carrera con ambas directivas:

   - `#pragma omp critical`: ha de incluirse el código crítico en el bloque de la directiva (al ser una única línea pueden omitirse las llaves).
   - `#pragma omp atomic`: la línea en la que se incrementa `sum` debe estar inmediatamente después de la directiva.

   La directiva `critical` es más potente ya que garantiza la exclusión mutua en el bloque de código que protege, pero esta potencia supone un mayor coste. Por su parte, la directiva `atomic` tiene un menor *overhead*, pero está limitada a un conjunto reducido de operaciones y solo protege una única asignación. Afortunadamente, el incremento de `sum` sí es soportado por `atomic`, por lo que es suficiente para que el resultado sea correcto y tiene un menor coste que `critical`. Por tanto, la solución elegida es la que utiliza `atomic`.

4. En comparación con las opciones dadas en el apartado anteior, la directiva `reduction` es la más eficiente en este caso. Esto se debe a que las otras alternativas declaran bien una región crítica o bien una instrucción atómica, que no pueden ser ejecutadas paralelamente en varios núcleos para evitar condiciones de carrera. Por contra, `reduction` opera con variables privadas para el acumulador en cada hilo que posteriormente se combinan para dar el resultado final, lo que permite paralelizar totalmente los distinitos hilos sin errores.

5. Dependiendo del tamaño de los vectores que se estén multiplicando, se puede observar que no siempre es conveniente paralelizar la ejecucuión. En el caso de operaciones con vectores por debajo de cierto tamaño, el *overhead* que supone crear los hilos paralelos es mayor que el tiempo que estos ahorran. Si en cambio el tamaño de los vectores es suficientemente grande, paralelizar supone mejores rendimientos, como se observa en las gráficas obtenidas.

   En cuanto al número de hilos lanzados, una mayor cantidad no siempre supone una mejora del tiempo de ejecución. En el caso de la imagen, se dispone de doce *cores* para utilizar (se utiliza un nodo AMD del clúster). El programa se optimiza al aumentar el número de hilos hasta alcanzar el máximo disponible. En la gráfica se puede apreciar cómo los mejores tiempos se dan para doce hilos, que es el número disponible en este procesador. Por encima de esta cantidad, aunque el rendimiento se parece, es ligeramente peor, ya que lanzar más hilos tiene un coste pero ningún beneficio, como se discutió en el Ejercicio 1. De hecho, en la figura 3 la aceleración se incrementa a mayor cantidad de hilos hasta llegar al máximo, doce, y en adelante las ejecuciones con más hilos tienen una aceleración comparable con haber lanzado menos de doce hilos, incurriendo sin embargo en mayores costes.

   Tras realizar varias ejecuciones de los programas, se ha llegado a la conclusión de que el umbral a partir del cual se rentabiliza la paralelización se encuentra entorno a 30 000. Esto implica que para vectores de dimensión menor se obtienen aceleraciones menores a 1, es decir, empeora el rendimiento. Se ha añadido a la versión multihilo un control para que se ejecute en serie en caso de no alcanzarse este umbral. Cabe destacar que esta medida ha sido tomada de nuevo en los procesadores AMD del clúster.

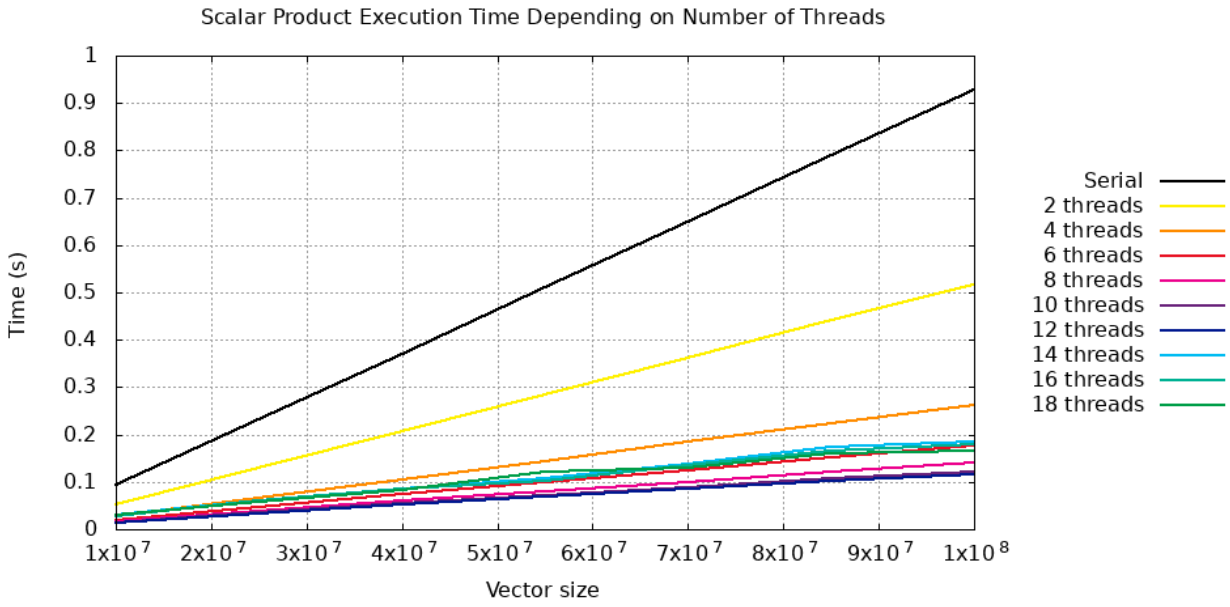


Figura 2: Tiempos de ejecución del producto escalar en función del número de hilos

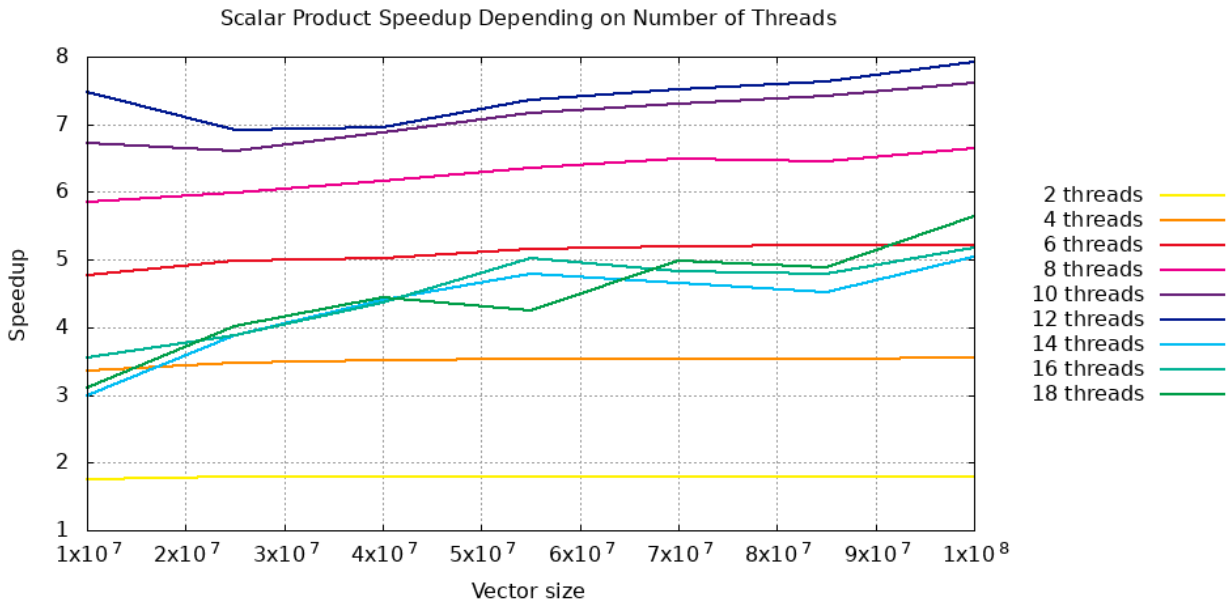


Figura 3: Aceleración del producto escalar en función del número de hilos (respecto versión serie)

Ejercicio 3

Tiempos de ejecución (s)				
Versión \n.º de hilos	**1**	**2**	**3**	**4**
Serie	106,8	-	-	-
Paralela – bucle 1	122,9	70,39	48,83	41,75
Paralela – bucle 2	109,6	62,50	39,62	30,82
Paralela – bucle 3	107,2	56,67	38,52	29,39

Tabla 1: Tiempos de ejecución para la multiplicación de matrices 2000×2000

***Speedup* (tomando como referencia la versión serie)**				
Versión \n.º de hilos	**1**	**2**	**3**	**4**
Serie	1	-	-	-
Paralela – bucle 1	0,869	1,517	2,188	2,558
Paralela – bucle 2	0,974	1,709	2,696	3,466
Paralela – bucle 3	0,996	1,885	2,773	3,634

Tabla 2: Aceleración para la multiplicación de matrices 2000×2000

1. En la tabla 2 se observa que la versión que paraleliza el bucle 1 (versión 1) tiene el peor rendimiento de las tres, mientras que la que paraleliza el bucle 3 (versión 3) tiene el mejor rendimiento. En particular, esto implica que, valga la obviedad, la versión que paraleliza el bucle 2 (versión 2) se encuentra en segundo lugar, lo que parece indicar una tendencia en relación al grano de la paralelización.

   Una posible explicación puede ser que la versión 1 lanza sus hilos $N \times N$ veces con los costes que cada lanzamiento acarrea. Además, las tareas que realiza cada hilo (en este caso, multiplicar una fila de la primera matriz por una columna de la segunda) pueden no ser tan provechosas como para compensar el coste de paralelizar. En contraste, la versión 3 solo lanza sus hilos una única vez, y cada uno se encarga de más tareas que los hilos de las demás versiones.

2. Estos resultados sugieren que la paralelización de grano grueso tiene un mejor rendimiento que la de grano fino. Es razonable generalizar y afirmar que la paralelización de grano grueso tiene un mejor rendimiento para otros algoritmos, al menos cuando sus bucles tienen una estructura similar.

3. Tras concluir que la versión con mejor rendimiento es la que paraleliza el bucle más exterior con cuatro hilos, se estudia cuánto mejor es respecto a la versión secuencial. En la figura 4 se observa que la versión paralela es bastante mejor que la versión en serie, especialmente para matrices de mayor tamaño. Se aprecia que los tiempos de la versión serie crecen más deprisa que los de la versión paralela en función del tamaño de los operandos.

   Por otra parte, pese a haber realizado los mejores esfuerzos para evitarlo (tomar varias veces[1] las medidas y calcular medias, tener cuidado de no evaluar sucesivamente las mismas matrices para evitar posibles mejoras gracias a las cachés, evaluar tamaños mayores[2] a los dados por

[1] Se tomaron 10 veces las medidas.
[2] Se evalúan tamaños desde 512 hasta 1856.

la fórmula $1024 + 512 \cdot P$), la gráfica de la figura 5 es muy irregular[3]. Se puede afirmar que la aceleración oscila entre 3,3 y 4, lo que supone una mejora de más del 200 % respecto de la versión serie. Lo que se espera *a priori* es que para matrices de dimensiones reducidas paralelizar no suponga mayores ventajas, incluso que pueda ser peor que no paralelizar puesto que es posible que el coste de lanzar los hilos supere los beneficios de hacerlo; mientras que para matrices de dimensiones mayores se obtenga una muy buena aceleración al paralelizar por ser el coste de lanzar los hilos despreciable en comparación.

En cuanto a un cambio de tendencia para matrices de dimensiones mayores, no se se aprecia en las figuras 4 y 5 (en la que tampoco se aprecia tendencia alguna como para decidir que cambia), en las que se evalúan tamaños entre 512 y 1856. Lo que podría esperarse es que, para matrices suficientemente grandes, la aceleración que supone paralelizar se vea disminuida ya que los recursos que cada hilo requiere son demasiado elevados. Específicamente, que para matrices muy grandes los recursos necesarios por hilo aumentan más rápidamente que antes. Por ejemplo, puede ocurrir, dada la arquitectura de los nodos del clúster, que las cachés de cada *core* sean comparativamente más pequeñas que la memoria necesaria para matrices grandes, particularmente las de niveles 1 y 2.

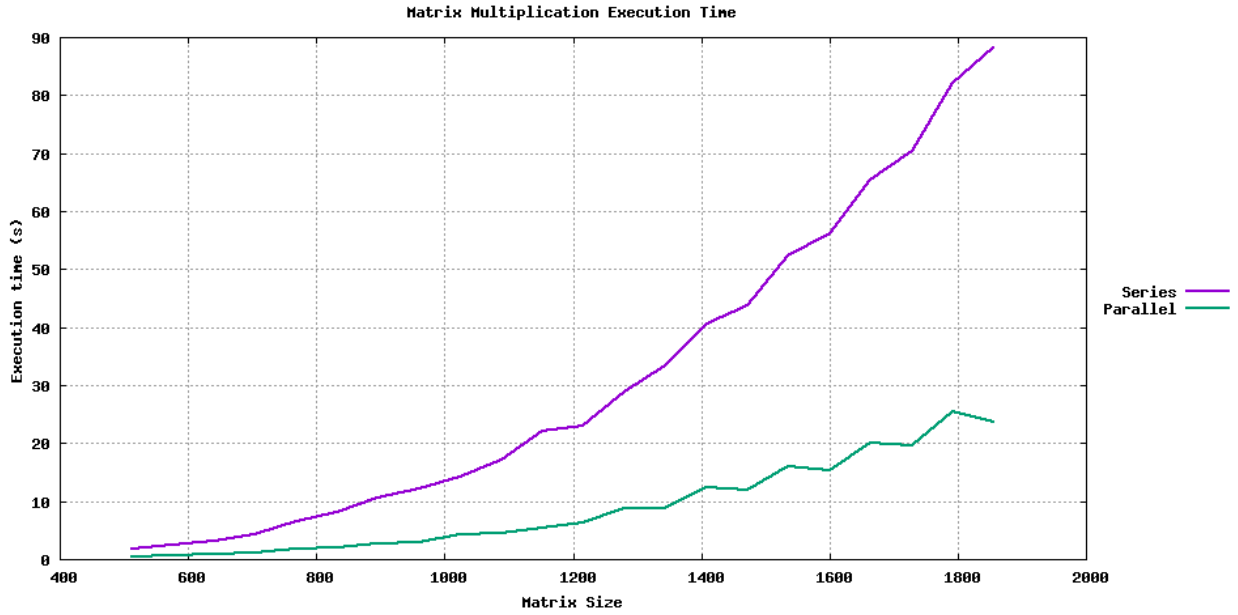


Figura 4: Tiempos de ejecución para el producto de matrices

[3]Quizás sea por la utilización del clúster por otros usuarios durante la ejecución. Es posible que otra tarea en el mismo nodo acabe utilizando alguno de los *cores* reservados para la tarea propia.

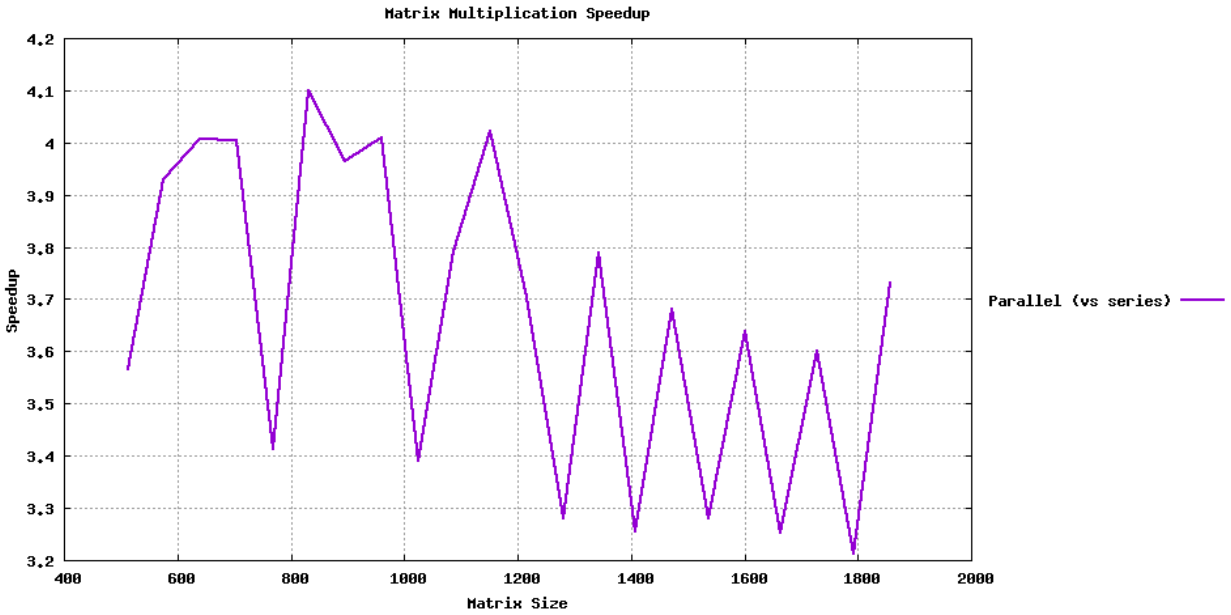


Figura 5: Aceleración para el producto de matrices (versión paralela respecto versión serie)

Ejercicio 4

1. Se usan 100 000 000 rectángulos, es decir, se hace una partición del intervalo $(0, 1)$ en 100 millones de componentes equiespaciadas para la integración numérica.

2. El programa `pi_par1` escribe directamente en el arreglo de sumas compartido (en su correspondiente entrada) en cada paso del algoritmo, mientras que `pi_par4` utiliza un acumulador privado y solo al final escribe en el arreglo compartido.

3. Ambas versiones obtienen el mismo resultado (que además es el correcto), pero `pi_par4` tiene un mejor rendimiento. Este fenómeno se explica por el *false sharing*.

   En ambas versiones se crea un arreglo con una entrada por hilo lanzado, y cada hilo escribe su suma parcial en la entrada correspondiente a su identificador. Si bien no se producen *data races* ya que cada hilo escribe en direcciones distintas, se hace presente el problema del *false sharing*: cada actualización de una entrada del arreglo en un *core* invalida el bloque de caché en el que reside en los demás *cores* aunque estos no accedan a esa entrada del arreglo en particular, lo que supone fallos de caché. Como `pi_par1` va modificando en cada paso del algoritmo su entrada del arreglo, provoca más fallos de caché en los *cores* vecinos que `pi_par4`, ya que este solo escribe en el arreglo una única vez al final.

4. Como era de esperar `pi_par2` tiene un rendimiento similar a `pi_par1`, pues lo único que hace es crear para cada hilo una copia privada de la variable `sum` (con su valor original, se usa `firstprivate`), pero esta es un puntero al arreglo y solamente se lee, con lo que no hace nada por solucionar el *false sharing*.

   Por su parte, `pi_par3` tiene un rendimiento claramente superior al de `pi_par2`. El motivo de esto es que se dejan índices a 0 en el vector para que así cada entrada significativa de este se encuentre en un bloque distinto y se evite así el *false sharing*. En el caso de la máquina empleada, por cada acumulador utilizado es necesario añadir otros 7 que no sean utilizados. Pese a que este programa mejora claramente el rendimiento, hay que tener en cuenta que empeora mucho en el uso de memoria. En particular, la máquina empleada necesita que el arreglo de variables *double* utilizado sea 8 veces mayor.

5. Tras realizarse pruebas con distintas separaciones entre los datos útiles del vector se confirman los resultados que se espera obtener. Si se utiliza un *padding* de 1, el programa se comporta igual que `pi_par1` (ya que son el mismo programa en realidad). A partir de este punto, cuanto mayor sea la separación, mejor es el rendimiento. Esto se debe a que una mayor separación entre los datos supone que haya menos datos útiles por bloque de caché, reduciendo así el *false sharing*. En el caso de la máquina empleada, al llegar a 8 (como ocurría en la versión original de `pi_par3`) se emplea un solo dato significativo por bloque, por lo que separar más los datos por encima de esta cifra no mejorará el rendimiento, ya que las entradas útiles estarán igualmente separadas en distintos bloques, que es el objetivo de esta versión del programa.

6. A diferencia de los anteriores, `pi_par5` no hace uso de un arreglo compartido, si no que utiliza directamente una variable compartida `pi` a la que cada hilo añade su suma parcial final (es decir, los hilos hacen un único acceso a `pi`). El resultado sigue siendo el correcto porque se usa la directiva `critical` para proteger el acceso a esta variable compartida. Específicamente se crea una zona de exclusión mutua que contiene la línea en la que se accede a `pi`, como se discutió anteriormente.

Comparando `pi_par4` y `pi_par5` se observa que tienen un rendimiento similar, lo que era de esperar puesto que ambas versiones se exponen al *false sharing* en el mismo número de ocasiones: cuando cada hilo accede a las direcciones de memoria compartidas, bien sea el arreglo en la primera o la variable en la segunda. Cabe comentar que la directiva `critical` supone un coste adicional para `pi_par5`, pero se ahorra realizar el bucle en el que se suman las entradas del arreglo de las versiones anteriores, por lo que parece ser que se compensan.

7. La versión `pi_par6` introduce la directiva `for` de OpenMP, que hace que las iteraciones del bucle se repartan entre los hilos. Es muy similar a `pi_par1`, salvo que en este se repartían manualmente los valores de `i` que cada hilo utilizaría.

   Por otra parte, `pi_par7` hace uso de la directiva `for` junto a una nueva directiva: `reduction`. Esta se encarga de que cada hilo haga su acumulación en una copia privada de la variable `sum` y al final combina los resultados de los hilos. También ha de indicarse la operación a realizar, en este caso `+`, que por lo explicado anteriormente ha de ser conmutativa para que el resultado no dependa del orden.

   Al ejecutar ambas versiones se observa claramente que `pi_par7` tiene un mejor rendimiento que `pi_par6`. Como se discutió anteriormente, `pi_par6` es idéntica a `pi_par1` salvo la división del bucle entre los hilos, lo que en particular implica que no hay ningún cambio que evite el *false sharing*. Por su parte, `pi_par7` evita el *false sharing* gracias al uso de la directiva `reduction`, por lo que es de esperar que tenga un mucho mejor rendimiento.

Ejercicio 5

1. Tras lo discutido en el Ejercicio 3, lo intuitivo sería paralelizar el bucle más externo. Sin embargo, analizando el código provisto se observa que el bucle 0 no forma parte de los cómputos en sí, sino que es una forma de procesar varias imágenes lanzando una única vez el programa (es decir, no tener que lanzar una ejecución por imagen). Luego las conclusiones obtenidas en el ejemplo previo no parecen extrapolarse a este caso.

   a. Si se pasan menos argumentos que *cores*, paralelizar el bucle 0 supone dejar un número de *cores* ociosos; no hay suficientes imágenes como para que a cada *core* le corresponda alguna. Por ejemplo, paralelizar el bucle 0 para procesar una única imagen es equivalente a no paralelizar, incluso podría tener un coste adicional por ejecutar las instrucciones de OpenMP para acabar de todos modos en un único hilo.

   b. Asumiendo 8 bits por píxel ($2^8 = 256$ colores, solo un canal), una imagen de 6 GB consta de unos $6{,}44 \cdot 10^9$ píxeles, o 6 gigapíxeles binarios. El programa almacena 3 arreglos con tantos bytes como píxeles aproximadamente[4]. Por tanto, reserva 6 GB de memoria por arreglo, 18 GB en total. Esto implica que al paralelizar el bucle 0 son necesarios 18 GB de memoria por hilo, lo que no es conveniente en el mejor de los casos e inaceptable en el peor.

   En definitiva, lanzar pocos hilos supone una mejora muy limitada en el rendimiento, y lanzar muchos supone un coste de almacenamiento excesivo. Por tanto, no parece ser este el bucle óptimo para ser paralelizado.

2. Hay cuatro bucles que acceden a los datos en un orden subóptimo. En todos se acceden a las matrices que almacenan los píxeles por columnas, es decir, acceden sucesivamente a elementos que distan entre sí el ancho de la imagen en píxeles. Es suficiente en cada caso intercambiar las cabeceras de los bucles `for` para que el recorrido se haga por filas y se aproveche así la localidad. Se adjunta el programa modificado en la entrega, indicando los bucles cambiados.

3. Una vez descartada en apartados anteriores la pararlelización del bucle 0 (el cual recorre todas las imagenes a procesar) no quedan muchas opciones más para la optimización. Cada iteración de este recorre tres grupos de bucles `for` en su ejecución (siempre hay un cuarto bucle que no se ejecuta), uno por cada imagen modificada que se quiere generar. Puesto que todos deben recorrer los píxeles de la imagen, todos son al menos dobles. En algunos casos hay bucles menores dentro de estos pero no se ejecutan suficientes iteraciones como para rentabilizar su paralelización. Dicho esto, sólo queda determinar qué bucle se puede optimizar más, si el exterior o el interior. Para modificar lo menos posible el diseño del programa y dado que en ejercicios anteriores ha aparecido como una de las mejores alternativas en cunato eficiencia, se utilizará la directiva `for`.

   Como cabía esperar, resulta más efectivo paralelizar el bucle externo, como ya se había visto anteriormente. Aunque ambos cambios mejoran el rendimiento, la elección del primer bucle (que recorre los índices de las columnas, generandose así hilos que recorren las filas) es claramente mejor.

4. En la tabla 3 se observan los resultados de la ejecución del programa `edgeDetector` para la versión en serie (con los accesos a los datos optimizados como se discutió anteriormente) y para la versión paralelizada, en este caso con 8 hilos.

[4]Se consideran despreciables las sustracciones que se hacen al quitar 2 o 4 píxeles a los bordes.

Imagen	Tiempos		Aceleración	Fps	
	Serie	Paralelo		Serie	Paralelo
SD	0,128	0,014	9,141	7,804	71,343
HD	0,519	0,054	9,521	1,924	18,323
FHD	1,176	0,122	9,570	0,850	8,135
UHD-4k	4,718	0,492	9,572	0,211	2,028
UHD-8k	18,901	1,962	9,632	0,052	0,509

Tabla 3: Resultados obtenidos en la ejecución de `edgeDetector`

Cabe comentar que las aceleraciones de la versión paralela respecto de la secuencial son superiores a 9 en todas las ejecuciones, y van aumentando a medida que se procesan imágenes de mayor resolución. Por otra parte, considerando que procesar imágenes a tiempo real equivale a hacerlo a 30 fps, únicamente se procesa a tiempo real imágenes SD con la versión paralela (que procesa más del doble de fotogramas por segundo necesarios). Para el resto de resoluciones no es posible procesar a tiempo real, siquiera por la versión en paralelo, aunque admitiendo el procesamiento a 20 fps, el estándar cinematográfico de la década de 1920, la versión en paralelo está muy cerca de procesar satisfactoriamente imágenes HD.

5. Ejecutando el comando `gcc -c -Q -O3 -help=optimizers | grep ‘‘unroll’’` en el *frontend* del clúster, como sugiere el manual de GCC, se observa que las banderas `-funroll-loops` y `-funroll-all-loops` están deshabilitadas, por tanto compilar con la bandera `-O3` no hace desenrrollado de bucles. Ejecutando de nuevo el comando pero con `grep ‘‘vector’’` se observa que la bandera `-ftree-vectorize` sí se encuentra habilitada, por lo que compilar con `-O3` realiza vectorización sobre árboles.

   Tras realizar la misma ejecución del apartado anterior con los programas optimizados por `-O3` se pueden observar resultados claramente mejores. Mientras que antes apenas era posible alcanzar la cota propuesta de 30 fps, en esta nueva ejecución se ve cómo los tiempos para imágenes HD superan claramente este umbral si se paraleliza su procesado. Incluso las imágenes en FHD superan los 20 fps que se hubieran considerado aceptables en estándares previos. En este caso la aceleración de la versión paralela es ligeramente menor, pues la versión secuencial también mejora su rendimiento, pero sigue suponiendo una ejecución aproximadamente 7 veces más rápida.

Imagen	Tiempos		Aceleración	Fps	
	Serie	Paralelo		Serie	Paralelo
SD	0,0364	0,006	6,598	27,508	181,501
HD	0,147	0,021	7,046	6,781	47,779
FHD	0,333	0,047	7,149	3,007	21,498
UHD-4k	1,332	0,191	6,979	0,751	5,241
UHD-8k	5,330	0,744	7,161	0,188	1,345

Tabla 4: Resultados obtenidos en la ejecución de `edgeDetector` optimizado con `-O3`