MEMORÍA PRÁCTICA 4

Paralelización con OpenMP

Ejercicio 0

En particular, si nos fijamos en la información que nos da el fichero cpuinfo podemos deducir cuántos cores tiene nuestro equipo y si tiene o no habilitado la opción de hyperthreading (el parámetro cpu_cores indica cuántas CPUs físicas hay en equipo en total, y el parámetro siblings hace referencia al número de CPUs virtuales), si se da el caso de que el valor de siblings sea el doble que cpu_cores.

En los ordenadores del laboratorio (máquina virtual de la asignatura) concluimos que nuestro equipo tiene solo 1 CPU física. En el parámetro siblings encontramos el valor 1 por lo que hay 1 CPU virtual gestionada por un hilo. Con el parámetro physical_id podemos deducir que los 2 cores están en los procesadores 0 y 2.

En el cluster, se da la siguiente configuración:

AMD: Hay 2 procesadores de 8 cores cada uno, sin hyperthreading.

Intel: Hay 2 procesadores de 4 cores cada uno, con hyperthreading.

Los dos pueden ejecutar 16 hilos.

En la máquina Ubuntu que hemos empleado para la ejecución de los ejercicios tenemos 4 cores físicos y está habilitada la opción de hyperthreading. Por tanto, tenemos 8 cores virtuales.

Ejercicio 1

Apartado 1.1.

Sí, no tiene sentido hacerlo ya que solo se podrá ejecutar en paralelo el número tantos hilos como hilos totales (en caso de tener hyperthreading será el doble del número de cores) tenga el sistema.

Apartado 1.2.

En los ordenadores del laboratorio se deberían ejecutar 4 hilos dado que hay 4 cores y no presenta hyperthreading.
En el cluster, se deberían utilizar 16 hilos para las máquinas AMD e Intel.
Para nuestro ordenador, deberíamos utilizar 8 hilos.

Apartado 1.3.

#pragma omp parallel private(tid) num_threads(num) es la más prioritaria. Después, va *omp_set_num_threads(num)* y por último #define OMP_NUM_THREADS 5.

Apartado 1.4.

La variable es independiente de cada hilo de ejecución en la región ejecutada en paralelo.

Apartado 1.5.

Se inicializa a 0.

Apartado 1.6.

La variable privada solo tiene efecto en la región de código bajo la cláusula en paralelo. Al terminar la región en paralelo la variable no deja rastro. La variable mantiene el valor que tenía antes de la ejecución.

Apartado 1.7.

Las variables compartidas se modifican por cada hilo (son compartidas) y tienen un valor de entrada del valor que tenía la variable declarada antes de la región *pragmaomp*. Las variables privadas aparecen sin inicializar y son independientes de cada hilo.

Ejercicio 2

Apartado 2.1.

Ambos vectores tienen todas sus componentes a 1 y el producto escalar es igual al tamaño de los vectores. De forma evidente, al aumentar el tamaño de los vectores aumenta el tiempo total de ejecución ya que se ejecutan más operaciones.

Apartado 2.2.

El resultado no es correcto. El resultado es un número aleatorio entre [M/Nthreads, M] debido a que la variable sum no se protege y se actualiza independientemente en cada ejecución de los hilos; se solapan las ejecuciones de los hilos.

Apartado 2.3.

Se puede resolver con ambas directivas, ya que ambas contribuyen a la exclusión mutua de una sección crítica. Se añade la cláusula de *pragma* correspondiente antes de realizar la suma en la variable sum. La opción óptima es atomic dado que es una región breve de código - una sola instrucción - que corresponde a operación atómica, y es más eficiente que empleando critical.

Cambios realizados:

```
#pragma omp parallel for
        for(k=0;k<M;k++)
        {
                #pragma omp atomic
                sum = sum + A[k]*B[k];
        }
```

```
#pragma omp parallel for
        for(k=0;k<M;k++)
        {
                #pragma omp critical
                sum = sum + A[k]*B[k];
        }
```

Apartado 2.4.

La ejecución es mucho más rápida que utilizando las cláusulas anteriores y es correcta.

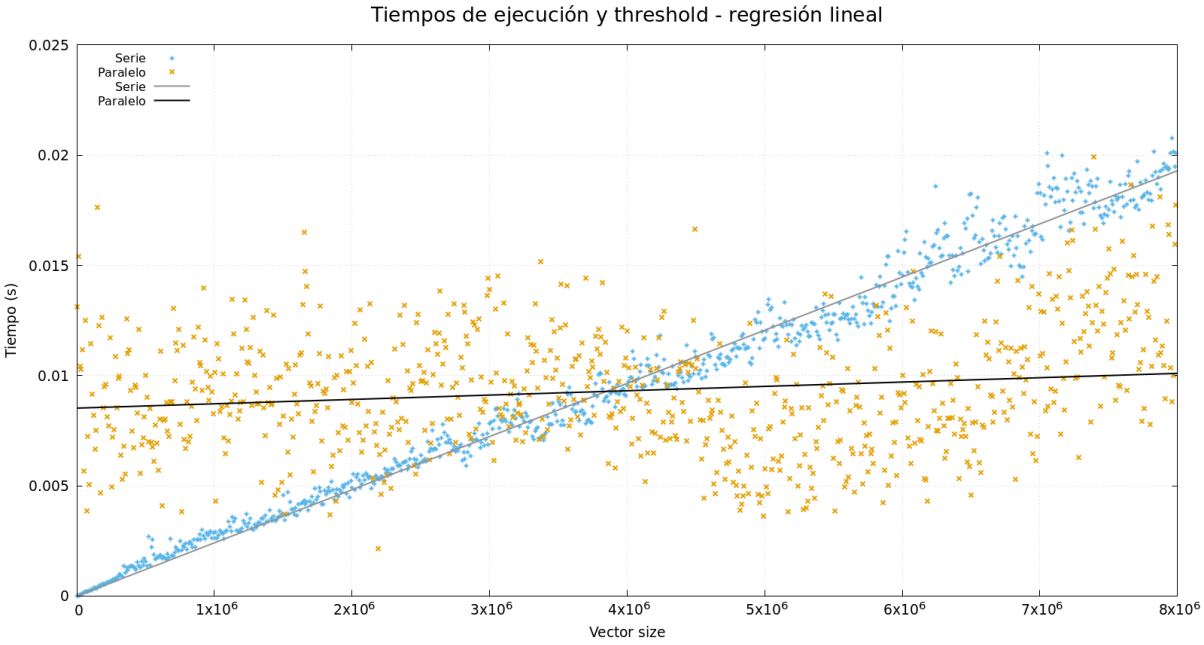
Cambios realizados:

```
#pragma omp parallel for reduction(+:sum)
        for(k=0;k<M;k++)
        {
                sum = sum + A[k]*B[k];
        }
```

Apartado 2.6.

Hemos desarrollado el script script_estimacion_manual.sh para, dado un valor de T, calcula una media de un número de repeticiones indicado y muestra por pantalla los valores de ejecuciones de ambos programas para 0.8*T y 1.2*T, de forma que se puede probando el valor de threshold que satisface ambas condiciones. Cerca de 0.8 * 4M (aprox. 3.200.000) se cumple que las medidas de los tiempos del programa en serie son inferiores. Asimismo, cerca de 1.2 * 4M (aprox. 4.800.000), las medidas tomadas para el producto escalar con paralelización resultan ser inferiores.

Apartado 2.7.



Hemos optado por una aproximación más visual y menos susceptible a datos atípicos. El principal problema que encontramos es que operando por prueba y error puede dar el caso de que se obtenga un valor de T significativamente alejado del real. Por esta razón, hemos optado por tomar datos de tiempos de ejecución en ambos programas variando el tamaño de vector entre 10^3 y 8*10^6. Haciendo una regresión lineal, el valor de T - de 4.000.000 según nuestro estudio - se encontraría en la intersección de ambas rectas, ya que es el punto en que el producto escalar paralelizado comienza a ser más eficiente que el de serie. Este método es, en esencia, una ampliación del método de prueba y error propuesto.

Ejercicio 3

Tamaño 1000 x 1000

Versión\# hilos	1	2	3	4
Serie	3.087691			
Paralela - bucle1	3.020166	2.084969	1.804682	1.846270
Paralela - bucle2	2.959633	1.552770	1.077845	0.935006

Paralela - bucle3	3.094017	1.830846	1.000337	0.893814

Speedup (tomando como referencia la versión serie)

Versión\# hilos	1	2	3	4
Serie	1			
Paralela - bucle1	1.0224	1.4809	1.7109	1.6724
Paralela - bucle2	1.0433	1.9885	2.8647	3.3023
Paralela - bucle3	0.9980	1.6865	3.0867	3.4545

Tamaño 1500 x 1500

Tiempos de ejecución (s)

Versión\# hilos	1	2	3	4
Serie	13.938234			
Paralela - bucle1	13.135171	9.336151	8.101746	8.031703
Paralela - bucle2	14.670947	7.372350	5.600984	5.337786
Paralela - bucle3	14.054810	7.523842	5.105005	5.211390

Speedup (tomando como referencia la versión serie)

Versión\# hilos	1	2	3	4
Serie	1			

Paralela - bucle1	1.0611	1.4929	1.7204	1.7354
Paralela - bucle2	0.9501	1.8906	2.4885	2.6112
Paralela - bucle3	0.9917	1.8525	2.7303	2.6746

Tamaño 2400 x 2400

Tiempos de ejecución (s)

Versión\# hilos	1	2	3	4
Serie	69.685487			
Paralela - bucle1	69.355288	40.345324	33.637817	31.159301
Paralela - bucle2	68.246849	35.736995	27.264857	24.356654
Paralela - bucle3	60.619205	32.383221	20.275543	19.343712

Speedup (tomando como referencia la versión serie)

Versión\# hilos	1	2	3	4
Serie	1			
Paralela - bucle1	1.0048	1.7272	2.0716	2.2364
Paralela - bucle2	1.0211	1.9500	2.5559	2.8610
Paralela - bucle3	1.1496	2.1519	3.4369	3.6025

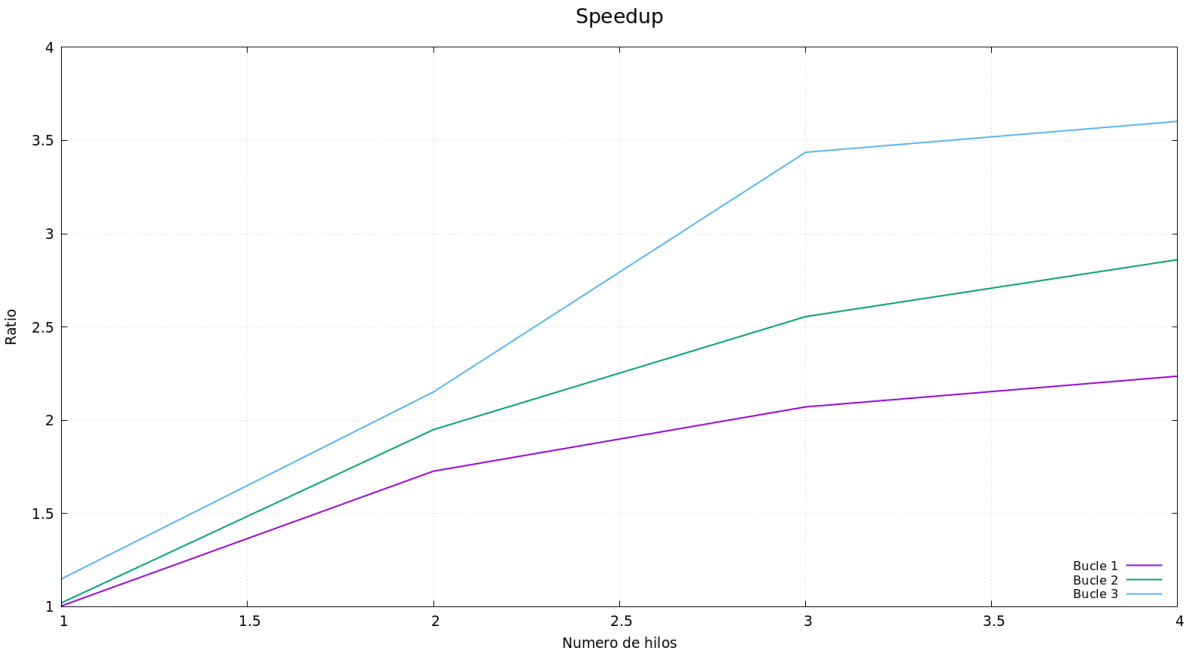
Apartado 3.1.

El peor rendimiento es el de la versión en serie. Se debe a que solo un hilo ejecuta todo el código del programa, mientras que en el resto de programas se produce una paralelización de las tareas.

Entre las tres versiones de paralelización, la de peor rendimiento es la del bucle interno (bucle 1), ya que se deben realizar N^2 lanzamientos del número de hilos especificado, por lo que se pierde mucho tiempo en la creación y recogida de hilos.
El de mejor rendimiento es del bucle exterior (bucle 3), siguiendo la lógica de minimizar el número de hilos lanzados, ya que realiza un único lanzamiento del número de hilos especificados. En todos los casos, la totalidad del programa se encuentra paralelizada, aunque de diferentes formas.

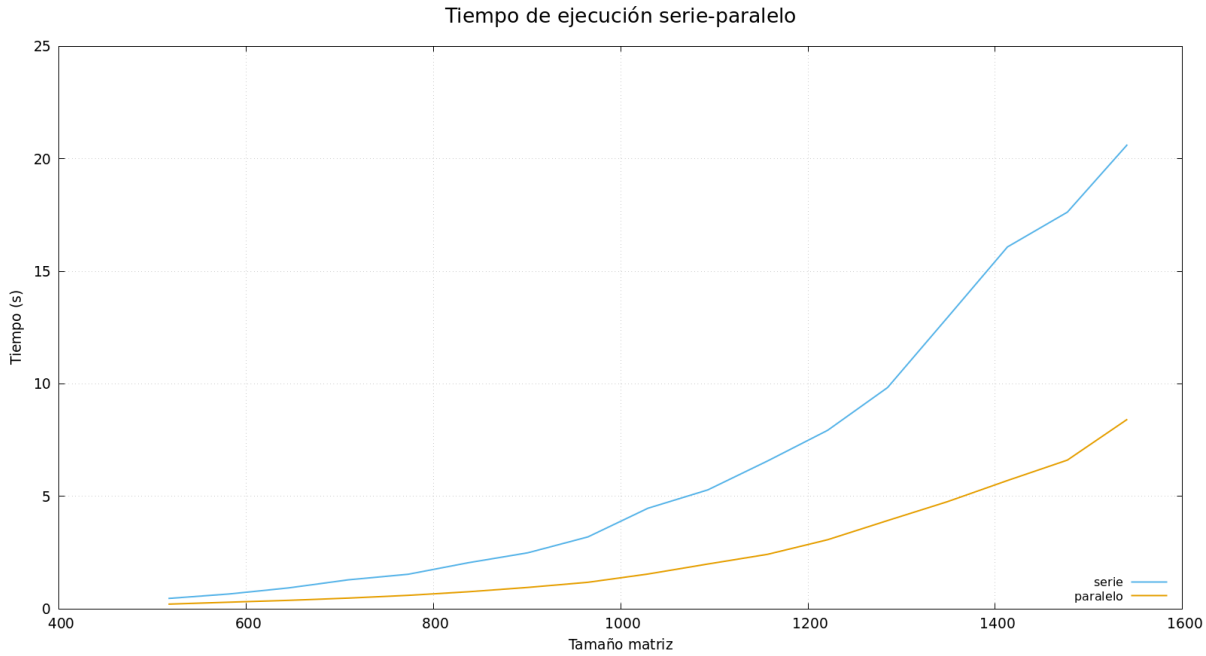
Apartado 3.2.

Paralelización de grano grueso (bucle más externo) ya que supone un menor lanzamiento total de hilos. Además, se da el caso de que las tareas repartidas entre los hilos es la misma, ya que el cálculo - y por ende el tiempo de cálculo - de los elementos de la matriz resultado es igual para cualquier posición, a excepción de los datos de las matrices iniciales. Por esta razón, no se producen desigualdades en la repartición de tareas. Además, requiere menos interacción entre hilos.

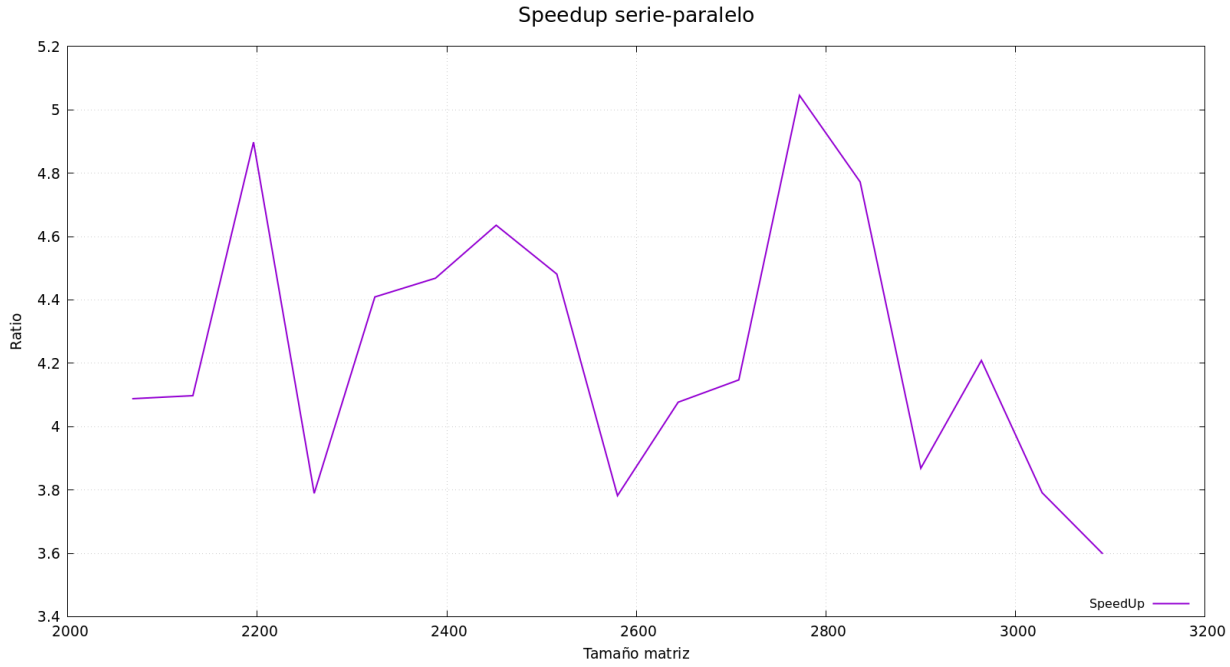


Se puede ver que el mejor rendimiento para matrices de tamaño 2400 x 2400 es el del bucle 3.

Apartado 3.3.



Como cabe esperar, el tiempo de ejecución asociado a la multiplicación paralelizada (crecimiento exponencial) es menor que el tiempo de la multiplicación en serie. La explicación es evidente: una repartición de la tarea entre los cuatro hilos, que se ejecutan de forma concurrente da lugar a un tiempo de ejecución total menor.



La gráfica está generada para valores de N en el rango [2068,3092]. Se observa que la aceleración se mantiene estable en un rango de [3.6, 5.1] a medida que crece el tamaño de matriz, este valor está relacionado con el número de cores disponibles y empleados para la ejecución de los programas (4 cores en nuestro caso).

Ejercicio 4

Apartado 4.1.

El programa utiliza 10^8 rectángulos diferentes con un valor de h de 10^(-8).

Apartado 4.2.

Versión/#hilos	8
Serie	1.164570
Paralela - par1	0.373870
Paralela - par2	0.379765
Paralela - par3	0.287946
Paralela - par4	0.285819
Paralela - par5	0.277279
Paralela - par6	0.396168
Paralela - par7	0.274538

Speedup (tomando como referencia la versión serie)

Versión/#hilos	8
Serie	1
Paralela - par1	3,114906251
Paralela - par2	3,066554316
Paralela - par3	4,044404159
Paralela - par4	4,074501695
Paralela - par5	4,199993508
Paralela - par6	2,939586236
Paralela - par7	4,241926436

El resultado es correcto en todos los programas.

Apartado 4.3.

No tiene sentido declarar la variable sum como privada, ya que al tratarse de un puntero se inicializa dentro de la sentencia pragma omp con el valor 0 y apuntaría a una posición de memoria inaccesible por lo que genera violación de segmento. En general y por estas razones, no es lógico declarar una variable de tipo puntero como private.

Apartado 4.4.

pi_par3 es una versión optimizada de pi_par1. Ambas se basan en un array para almacenar el resultado de la suma del bucle. La diferencia radica en que pi_par3 reserva más memoria de la necesaria de forma que los datos se guarden en bloques de caché distintos, separados por el padding. De esa forma evita los problemas de false sharing, ya que en pi_par1 por cada acceso por parte de un hilo a la variable sum, que se encuentra en el mismo bloque que las otras, se genera un fallo de memoria. Esto conlleva un mayor tiempo de ejecución, ya que es necesario llevar el bloque modificado de caché a memoria y al revés.
En el caso de pi_par5 solo utiliza una variable sum para almacenar el resultado de la suma y establece las secciones críticas por medio de la cláusula critical, fomentando la exclusión mutua de los elementos de la suma.
False sharing se produce cuando un core modifica algún dato correspondiente a un bloque de caché al que algún otro hilo accede posteriormente. El bloque está marcado como inválido y es necesario acceder a memoria para cargar el bloque con los datos nuevos modificados, generando un fallo de memoria por acceder a datos del bloque pese a que sean distintos (y distintas posiciones de memoria).
El tamaño de línea de caché es relevante ya que determina el número de datos que pueden estar en un mismo bloque de caché. Esto tiene gran importancia para evitar problemas de false sharing dado que elementos de esa línea son los que conllevan fallos de memoria: al escribir los datos en memoria y al traerlos cuando queremos modificar un nuevo datos perteneciente al mismo bloque.

Apartado 4.5.

La sentencia critical genera una sección de código que solo puede ser accedida por un hilo simultáneamente (sección crítica). El ejercicio pi_par5.c utiliza la cláusula critical para generar una sentencia de código de suma independiente para cada hilo y así evitar condiciones de carrera. De esta forma no se producen fallos en memoria dado que no utiliza vectores para almacenar los datos de la suma y cada hilo accede a la sección crítica adecuadamente, por tanto el tiempo de ejecución es menor y, consecuentemente el speedup es mayor que programas que utilizan arrays para almacenar sumas de elementos (se someten a un mayor número de fallos en memoria).

Apartado 4.6.

El programa pi_par6.c realiza dos paralelizaciones del código mediante sentencias pragma. La primera se realiza antes del bucle de forma que se reparte la tarea entre todos los hilos disponibles en el sistema. Sin embargo, al paralelizar el bucle for ya no hay más capacidad de concurrencia. De esta manera, se pierde la paralelización, ya que no hay suficientes hilos como para que cada hilo creado en la primera sentencia genere a su vez más. Por esta razón, el rendimiento de esta versión es menor, con una aceleración aproximada de 3, mientras que otras versiones - pi_par3, pi_par4, pi_par5 y pi_par7 - tienen una aceleración de 4.

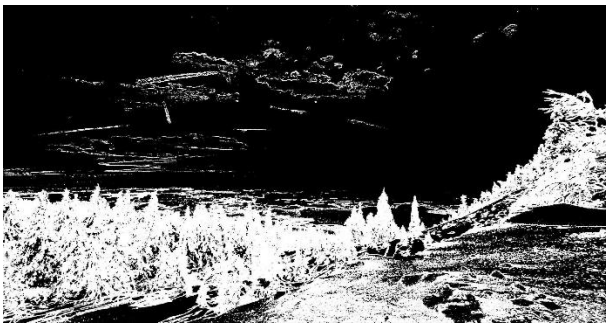
Apartado 4.7.

La versión óptima es pi_par7.c que utiliza la cláusula reduction para paralelizar y restringir las secciones críticas de los hilos en la suma de elementos del vector. De esta forma el programa utiliza variables a las que acceden todos los hilos y efectúan una operación de acumulación de forma atómica (suma de elementos del vector), derivando en una mejora en el tiempo de ejecución respecto a los otros programas. Así pues, es mejor que emplear una sección crítica.

Ejercicio 5

Apartado 5.0.

El programa edgeDetector.c genera 3 ficheros imagen a partir de la imagen seleccionada. Un fichero con la imagen en negro resaltando los bordes y la silueta en blanco. Un fichero con esa imagen anterior sin ruido. Una última imagen que representa la imagen pasada como argumento convertida a tonos de color blanco y negro.

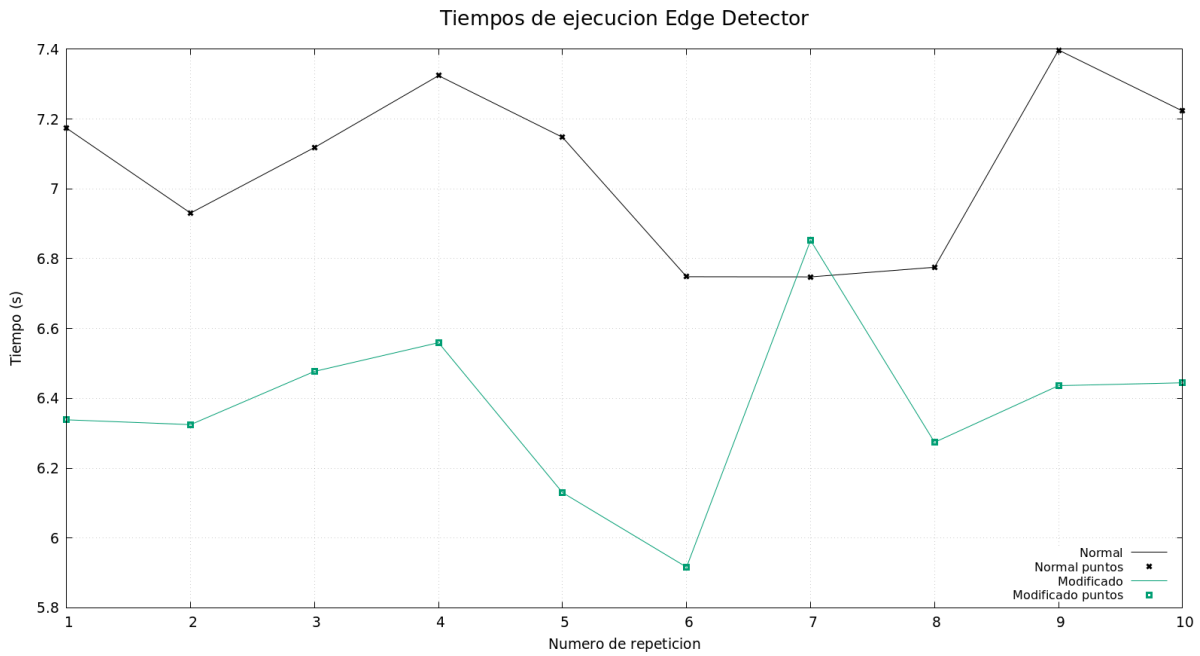



Apartado 5.1.

El bucle 0 no es el bucle óptimo para ser paralelizado. La paralelización de grano grueso es eficiente cuando se lleva a cabo una repartición justa del trabajo entre todos los hilos, condición que no se satisface en este caso. Así pues, en la línea de la pregunta a), si se paraleliza el bucle externo - bucle 0 - pero se piden procesar menos imágenes que cores virtuales tiene el equipo, no se estará empleando la totalidad de la capacidad de paralelización, ya que se lanza un hilo por cada imagen. En relación a la pregunta b), cada hilo emplea del orden de 4 veces el tamaño original de la foto, una para cargar la imagen y tres para las imágenes de resultado. Si la imagen ocupa 6GB, al lanzar todos los hilos a la vez, la cantidad de memoria total ocupada se multiplica por la cantidad de hilos lanzados. El programa deja de ser funcional, ya que habría fallos por falta de memoria disponible.

Apartado 5.2.

El orden en todos los bucles - salvo el primer bucle del programa, en la función *gaussiankernel* - del programa que acceden a vectores de variables no era correcto dado que no accedía a los elementos del vector en orden. De esta forma accedían a bloques distintos cada vez y eso suponía un mayor número de fallos de caché. Cambiando el orden de los iteradores en los bucles conseguimos acceder a los elementos del array de forma consecutiva y generando un número menor de fallos de caché. Por lo tanto, el tiempo de ejecución se reduce considerablemente como podemos observar en la gráfica proporcionada.



Apartado 5.3.

Tras probar diferentes opciones en el programa edgeDetector_mod.c, hemos llegado a la conclusión que la paralelización de grano grueso en los bucles internos es la

óptima y la que mejor tiempo de ejecución resulta. En comparación a su homónimo de grano fino, esta segunda conlleva la necesidad de crear 8 hilos, en nuestro caso, por cada iteración del bucle externo, de forma que el tiempo empleado en la generación de hilos es excesivo. Otro tipo de paralelizaciones como la creación de una región paralela para todo el código del interior del bucle 0, y la pertinente repartición de la tarea de los bucles for en función del número de hilo, conlleva la necesidad de sincronizar los procesos, ya que el resultado de un bucle anterior es empleado en el siguiente bucle. Dentro de los bucles for, no es necesaria ningún tipo de exclusión mutua para ninguna variable a excepción del último, que incluye una suma. En este caso, la solución se caracteriza por el uso de la cláusula reduction.

Apartado 5.4.

Resolución	Tiempo serie (s)	Tiempo paralelo (s)	Speedup	Tasa FPS
SD 720 x 480	0.037809	0.019898	1.9001	50.26 -> 50
HD 1280 x 720	0.098321	0.046630	2.1085	21.45 -> 21
FHD 1920 x 1080	0.226099	0.121226	1.8651	8.25 -> 8
UHD-4k 4096 × 2160	1.178974	0.344459	3.4227	2.90 -> 2
UHD-8k 7680 × 4320	4.224342	1.305665	3.2354	0.77 -> 0

Apartado 5.5.

Resolución	Tiempo serie (s)	Tiempo paralelo (s)	Speedup	Tasa FPS
SD 720 x 480	0.009382	0.028839	0.3253	34.68 -> 34
HD 1280 x 720	0.024344	0.065260	0.3730	15.32 -> 15
FHD 1920 x 1080	0.057545	0.094255	0.6105	10.61 -> 10
UHD-4k 4096 × 2160	0.460323	0.383900	1.1991	2.60 -> 2
UHD-8k 7680 × 4320	1.496586	1.390388	1.0764	0.72

Las imágenes generadas, a simple vista, parecen idénticas a las generadas en el apartado anterior.

En relación a los tiempos de ejecución, se aprecia que los tiempos de la versión serie se ven reducidos considerablemente y el speedup es menor con esta optimización. Sin embargo en la versión paralela se mantienen con valores similares. Al ejecutar el programa paralelizado con la bandera -O3 podemos apreciar una ligera reducción en los tiempos de ejecución. Nuestro programa ya paralelizado y vectorizado con Openmp no requiere de optimizaciones de mejoras de eficiencia y rendimiento en bucles grandes.

La inclusión de la flag -O3 para optimizar la ejecución conlleva la habilitación de una serie de banderas. En primer lugar, **ftree-loop-distribution** mejora el rendimiento de la caché para bucles grandes como los que iteran sobre el tamaño de la imagen, y activa otras optimizaciones como la paralelización y vectorización. Asimismo, la bandera **fpeel-loops** activa el desenrollado de bucles, de forma que se eliminan bucles con una cantidad constante de iteraciones pequeñas.
Fuente: Optimize Options (Using the GNU Compiler Collection (GCC))