Informe práctica 4

Zdravko Dimitrov Arnaudov
10/01/2021

Prefacio

El propósito de esta práctica es abordar el paradigma de paralelismo funcional asíncrono, con la directiva 'task' que dispone OpenMP. Esta herramienta favorece el paralelismo cuando no sabemos con exactitud las dimensiones del problema a computar. De este modo, se generan unidades de trabajo independientes que serán computadas por el Thread pool disponible.
En mi opinión, esta ha sido de las prácticas más interesantes y motivantes entre las realizadas. Inicialmente, supuso un reto leer el código y comprender su funcionalidad pero después, la evolución de la actividad fue súper fluida. La parte que más me ha llamado la atención fue conseguir paralelizar el problema. Es imprescindible entender el comportamiento del programa y sus recursos para aprovecharlos y sacar el máximo potencial a la hora de paralelizar. Con más tiempo, me hubiese gustado entrar más en detalle con la paralelización mediante pipelines, pero estoy bastante contento con el resultado alcanzado. No creo que hubiese cambiado nada de esta práctica, contempla todos los contenidos necesarios para completarla y alcanzar los objetivos propuestos.

Índice

1. Sistema

Con intención de obtener una descripción del sistema lo más detallada posible, buscaremos una serie de comandos equivalentes a '*cpuinfo*' o '*lscpu*', puesto que disponemos de Mac OS y dichas instrucciones no son reconocidas.

Uno de ellos se corresponde con '*system_profiler SPHardwareDataType'*, el cual nos muestra los aspectos hardware más generales y relevantes a tener en cuenta:

```
Hardware Overview:

      Model Name: MacBook Pro
      Model Identifier: MacBookPro17,1
      Processor Speed: 2,4 GHz
      Number of Processors: 1
      Total Number of Cores: 8
      L2 Cache (per Core): 4 MB
      Memory: 8 GB
```

Otro de los comandos encontrados es '*sysctl -a* | *grep machdep.cpu* ' y este en particular nos aporta información adicional sobre el número de cores físicos y lógicos del procesador:

```
machdep.cpu.cores_per_package: 8
```

```
machdep.cpu.core_count: 8
machdep.cpu.logical_per_package: 8
machdep.cpu.thread_count: 8
machdep.cpu.brand_string: Apple M1
machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 APIC SEP MTRR
PGE MCA CMOV PAT PSE36 CLFSH DS ACPI MMX FXSR SSE SSE2 SS HTT TM PBE
SSE3 PCLMULQDQ DTSE64 MON DSCPL VMX EST TM2 SSSE3 CX16 TPR PDCM SSE4.1
SSE4.2 AES SEGLIM64
machdep.cpu.feature_bits: 151121000215084031
machdep.cpu.family: 6
```

Por último, emplearemos un tercer comando '*sysctl hw*' para observar más en detalle el reparto de cores y también información adicional asociada a la memoria cache:

```
hw.ncpu: 8
hw.byteorder: 1234
hw.memsize: 8589934592
hw.activecpu: 8
hw.physicalcpu: 8
hw.physicalcpu_max: 8
hw.logicalcpu: 8
hw.logicalcpu_max: 8
hw.cputype: 7
hw.cpusubtype: 4
hw.cpu64bit_capable: 1
hw.cpufamily: 1463508716
hw.cpusubfamily: 0
hw.cacheconfig: 8 1 1 0 0 0 0 0 0 0
hw.cachesize: 3612786688 65536 4194304 0 0 0 0 0 0 0
hw.pagesize: 4096
hw.pagesize32: 16384
hw.busfrequency: 100000000
hw.busfrequency_min: 100000000
hw.busfrequency_max: 100000000
hw.cpufrequency: 2400000000
hw.cpufrequency_min: 2400000000
hw.cpufrequency_max: 2400000000
hw.cachelinesize: 64
hw.l1icachesize: 131072
hw.l1dcachesize: 65536
hw.l2cachesize: 4194304
hw.tbfrequency: 24000000
```

Teniendo en cuenta las especificaciones de nuestra máquina y que la arquitectura es ARM, para poder hacer uso de las herramientas que ofrece OpenMP es necesario usar una compilación distinta para nuestros programas. Inicialmente, ha sido imprescindible emplear una consola (Terminal, en Mac) corriendo un software específico que traduce instrucciones de x86 a ARM (Rosetta). Después, para compilar nuestro programa deseado en C y emplear tanto las directivas como funciones de OpenMP, el comando de compilación será: '*gcc -Xpreprocessor -fopenmp file.c -o file -lomp*'

2. Desarrollo de las preguntas planteadas

Para empezar, analizaremos todos aquellos errores que hemos identificado a lo largo del código y que nos impiden comenzar comenzar con la práctica. Empezando con el programa generator.c, distinguimos que en la línea 48 del código se pretende hacer 'strlen(sfooter)..'. Esta sentencia es incorrecta porque 'sfooter' es un array de caracteres y precisamente al final no se añade el carácter nulo, siendo este el límite de la función strlen. Para solucionarlo, tenemos dos opciones: añadir el caracter nulo o sustituir la función por sizeof, determinando correctamente el número de caracteres y símbolos en el array. Siguiendo con el fichero video_task.c, encontramos dos fallos más.
El primero de ellos tiene que ver con la matriz empleada en la función gauss para filtrar los píxeles. Concretamente, estaba mal inicializada porque no se acotaban las filas cada 5 elementos usando corchetes. El segundo y último fallo a corregir se encuentra el la misma función de filtrado. En particular, estábamos haciendo 'pixels[(x+dx-2),(y+dy-2)]'. Para solucionarlo, hemos tenido que sustituir la coma por && para adoptar ambas condiciones a la vez.

Una vez compilado nuestro programa, nuestro primer paso fue entender el comportamiento de la ejecución secuencial y después comenzar a paralelizarla. Explicaremos ordenadamente las pautas que hemos tomado hasta el diseño del algoritmo y posteriormente continuaremos contestando el resto de preguntas planteadas.

Como ya hemos dicho, lo primero que hicimos fue leer el código. En resumen, hemos apreciado que los vídeos elaborados por el programa generator se constituyen en base a 3 propiedades: anchura de las imágenes, altura de las imágenes y número de imágenes. Al ejecutar generator, se establece un fichero que será el vídeo de entrada, el cual procesará video_task.c para generar el correspondiente video filtrado de salida.
El código video_task.c, para cumplir su función conforma dos arrays, uno para leer las imágenes de los vídeos y otro donde almacenar las imágenes filtradas y posteriormente escribirlas en el fichero de salida. Si prestamos atención, el cuerpo de video_task se reduce a un bucle do while, en el que lee una imagen al tiempo, la procesa y la escribe en el vídeo filtrado ordenadamente, mientras queden imágenes por leer.
Dando un par de vueltas al código, nos damos cuenta de que estamos reutilizando la misma posición, tanto de pixels como de filtered para filtrar las imágenes. Cuando se ha terminado de procesar una, se sobreescribe la siguiente, si existe, en la misma posición. Por lo tanto, consideramos que este es el motivo por el que nos conviene buscar una solución paralela.
Si en lugar de leer una imagen al tiempo leyésemos n, podríamos llegar a aprovechar los cores que dispone nuestra máquina para al menos procesar una imagen a la vez. Es evidente que alcanzaremos un speedup considerable si se dan unas condiciones óptimas, en cuanto a dimensiones del vídeo y número de imágenes que procesamos cada vez.

A continuación, analizaremos el código implementado. El objetivo de la paralelización es, dentro del bucle do while, leer secuencialmente una serie de imágenes, procesarlas en paralelo con la directiva 'task' y escribir ordenadamente en paralelo y vuelta a empezar. Cabe destacar que si paralizásemos de forma conjunta el filtrado y la escritura, no respetaríamos el orden de las imágenes y por esta razón es necesaria la separación. Para poder leer, escribir y levantar taras secuencialmente, es imprescindible que solo compute un solo hilo. Por ello, tendremos una región paralela, una región single o máster y en el momento de filtrar las imágenes levantaremos tareas y al finalizar el bucle sincronizaremos todo el cómputo de tareas con taskwait.

```
//inicialización variables
 i = 0;
 int contador_iter = 0;

 start = omp_get_wtime(); //medimos tiempo inicial
 #ifdef PARALLEL  //paralelo
 #pragma omp parallel shared (pixels, height, width, in, out, filtered, seq, i, contador_iter)
 {
        #pragma omp single
                {

                bool termina = false;

                do {

                        while (i < seq && termina == false) {

                                size = fread(pixels[i], (height+2) * (width+2) * sizeof(int), 1, in);

                                if (size){
                                        i++;
                                } else {
                                        termina = true;
                                }

                        }


                        for (int j = 0; j < i; ++j) {
                                #pragma omp task
                                {
                                        fgauss (pixels[j], filtered[j], height, width);

                                        #ifdef IMPRIME
                                        int n_t = omp_get_thread_num();
                                        printf ("Soy el thread: %d ejecutando la imagen: %d\n", n_t,
contador_iter + j);
                                        #endif
                                }

                        }

                        #pragma omp taskwait //sincronización

                        for (int j = 0; j < i ; ++j){
                                fwrite(filtered[j], (height+2) * (width + 2) * sizeof(int), 1, out);

                                #ifdef DEBUG
                                printf ("Imagen terminada.\n");
                                #endif

                        }

                        //inicialización
                        contador_iter = contador_iter + i;
                        i = 0;
                        termina = false;

                } while (!feof(in));
```

} //fin single

} //fin parallel

Antes de comenzar respondiendo a las preguntas, mencionaremos cómo llegamos a imprimir el identificador de la imagen que filtra cada thread. Hemos dado con una variable 'contador_iter', que incrementa el número de imágenes leídas por cada iteración de bucle, para que solo tengamos que sumar de forma añadida su posición en el bucle de filtrado '…contador_iter +j'. Seguidamente, avanzaremos con las preguntas.

Una de las nuevas directivas que hemos empleado en este caso es 'omp_get_thread_num()', para poder imprimir por pantalla el thread que ha filtrado cada imagen. En cuanto a las directivas, hemos empleado tres nuevas. Una de ellas es '#pragma omp single' para delimitar una región de cómputo secuencial. La segunda y más novedosa es #pragma omp task. Esta sentencia genera tareas individuales a computar y las almacena en una cola. Cuando los threads que disponemos en la región paralela están libres, van consumiendo dichas tareas hasta terminarlas, pero el orden de ejecución de las mismas es incierto. En este caso, nos interesa crear tantas tareas como imágenes haya leídas. Para asegurarnos de que todas las tareas han acabado, realizamos una sincronización al final de este bucle con la directiva '#pragma omp taskwait'.

Respecto de la región paralela, es evidente que algunas variables serán compartidas y otras privadas. Las variables serán compartidas si su mismo valor debe ser usado en común por varios threads. Partiendo de esta regla, las variables 'height' y 'width' se compartirán porque su valor es invariable y es necesario tanto para leer las imágenes como para procesarlas por los distintos threads. Los arrays 'pixels' y 'filtered' también serán compartidos por múltiples threads, aunque acaben procesando imágenes en distintas posiciones de los mismos. Los ficheros 'in' y 'out', a pesar de ser empleados solo por un array para leer o escribir secuencialmente, los mantenemos compartidos por escalabilidad. Si en algún momento decidimos leer o escribir con varios hilos a la vez, conviene que todos puedan acceder a los ficheros de entrada y salida. Otra de las variables compartidas, puede ser el número de 'seq' establecido, del que deberá acceder el hilo single como los threads que computen las tasks. Las dos últimas variables compartidas que nos quedan por comentar son la 'i', siendo este el iterador que recorre las imágenes leídas y 'contador_iter', que modificará el hilo single y del que accederán el resto de hilos computando las tasks para imprimir el identificador de la imagen que han filtrado. En cuanto a las variables privadas, hemos encontrado dos, la primera es el booleano 'termina' que inicializamos dentro de la región single y que por lo tanto solo emplea un hilo. La segunda y última variable privada que hemos utilizado es 'j' a la hora de iterar los dos bucles for. En el primer bucle, es relevante porque cada hilo precisa de una iteración distinta, también podríamos haber empleado 'firstprivate(j)'. En el último bucle for, solo accede el hilo single pero en caso de haber más hilos para cumplir la función de escritura, también debería ser privada.

Para calcular la ganancia, emplearemos las dimensiones de vídeo que vienen indicadas por defecto : 1920 ancho, 1440 alto y 80 imágenes. En este caso, probaremos únicamente con las dimensiones establecidas y seq 8, siendo también este nuestro número de hilos en la región paralela. La ganancia que hemos obtenido es de 5,41. Es una mejora destacable, reduce los tiempos de 19,68s en secuencial a 3,63 en paralelo.

En el apartado siguiente de pruebas, entraremos en más detalle calculando las ganancias que obtenemos según las dimensiones del vídeo, el número de threads operando en la región paralela y el tamaño de la secuencia de imágenes.

Finalmente, la única optimización con la que hemos dado en la función fgauss() es usar pro-incremento (++i) en lugar de post-incremento(i++) en los bucles. El motivo de este cambio es incrementar la variable directamente en lugar de mantener su antiguo valor. Calculando de nuevo los tiempos de ejecución, vemos que la ganancia respecto de la versión paralela original no supone mejora, 0,9992. Incluso, empeora el resultado pero con mínima diferencia.

3. Benchmarking

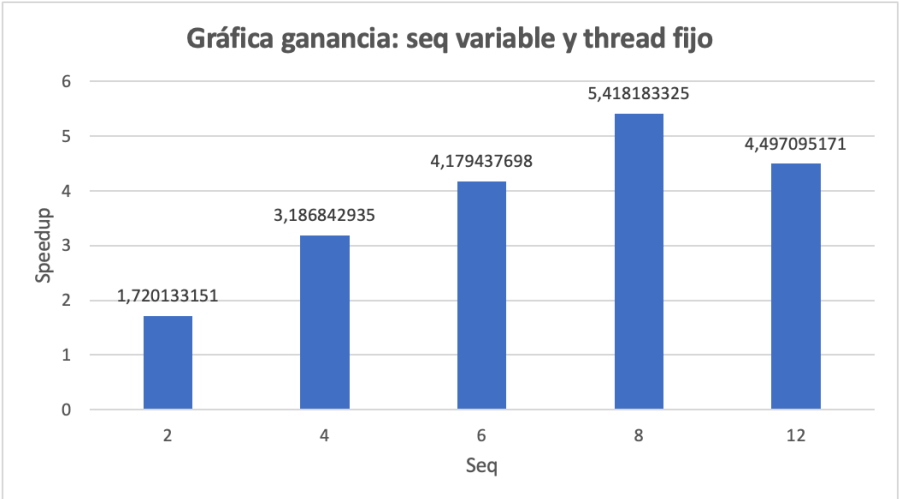
En este punto de la memoria, analizaremos los resultados obtenidos en base al benchmarking realizado. Nuestro interés es estudiar el comportamiento al paralelizar problemas con diferentes dimensiones y valorar bajo qué casos se obtiene mayor ganancia. Para ello, realizaremos dos pruebas: modificando las secuencias de imágenes que se procesan para ver el impacto en el speedup y alterando el número de threads que operan en la región paralela con diferentes dimensiones de video y secuencia.

Para agilizar nuestras ejecuciones, implementaremos un script sencillo 'VideoTaskScript.sh' que realizará 11 ejecuciones secuenciales del programa video_task, la primera de calentamiento, con una parada de 1s por activación.

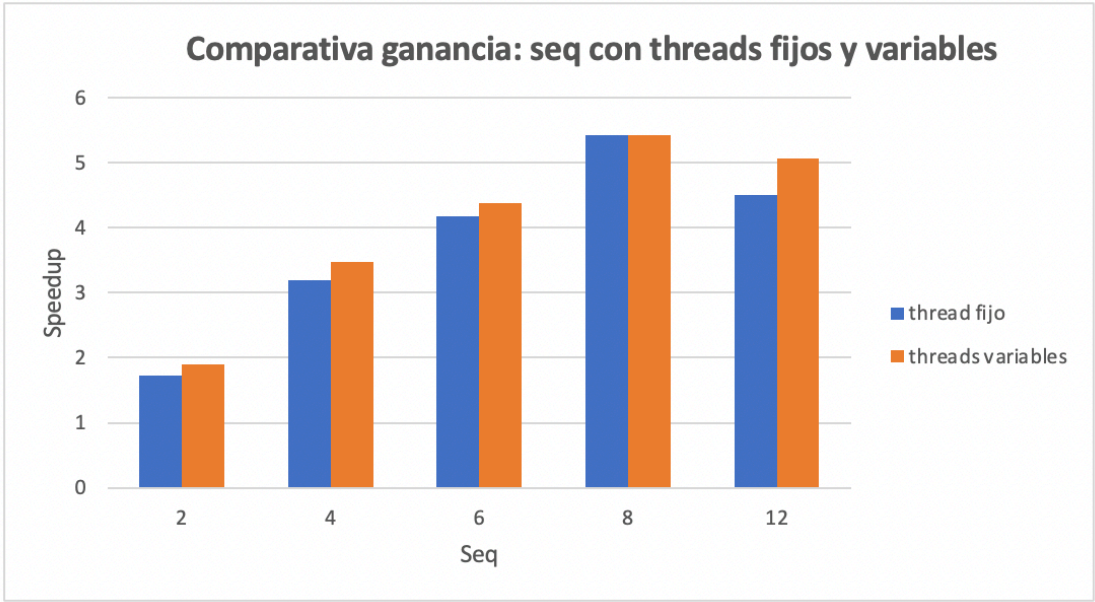
```
#! /bin/bash
for i in {0..10}
do
        sleep 1
        ./video_task
done
```

Antes de ejecutar el programa, hemos compilado el fichero 'generator.c' con 'gcc generator.c -o generator' y después compilado 'video_task.c' con 'gcc -03 -Xpreprocessor -fopenmp video_task.c -o video_task -lomp'.

Anteriormente, la ganancia fue calculada comparando la ejecución secuencial con la paralela, a partir de un número establecido de secuencias de imágenes que se procesan en paralelo. Nuestra prueba consistirá en probar múltiples tamaños de secuencia para las mismas dimensiones del vídeo y el mismo número de threads en la región paralela.
En particular, probaremos con secuencias de 2, 4, 6, 8 y 12 con 8 hilos.

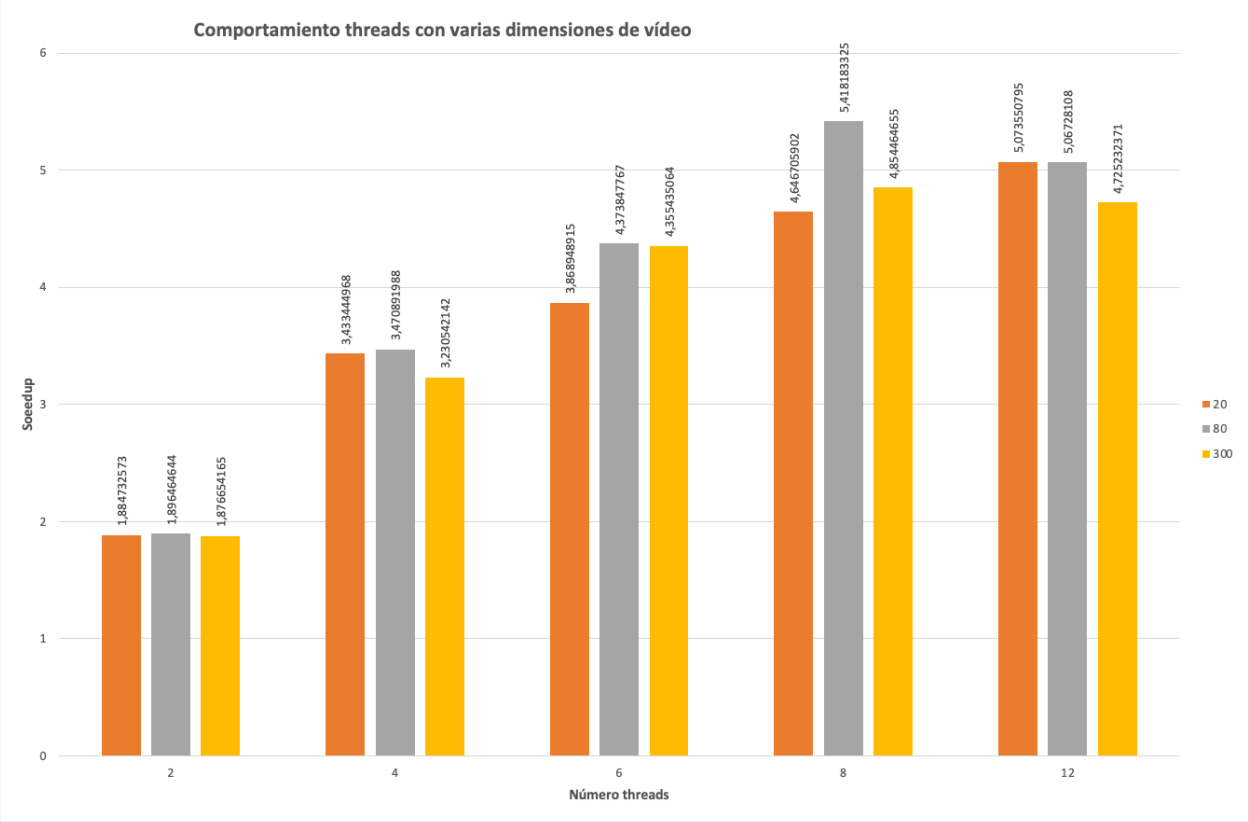


Como vemos, la ganancia tiende a incrementar a medida que procesamos bloques de imágenes mayores paralelamente. Con tamaño de secuencia 8, se edifica un Speedup máximo que se corresponde con el calculado anteriormente. Observando atentamente la gráfica, podemos formularnos una serie de preguntas. Si a mayor número de seq obtenemos más ganancia, ¿por qué con 12 desciende tanto? Además, la ganancia progresa equilibradamente hasta 6 de secuencia. Entonces, ¿por qué asciende tan rápido cuando llega a 8? Bien, para contestar es preciso tener en cuenta el número de threads computando en la región paralela. En todos los casos de seq, siempre empleamos 8 hilos. Por este motivo, cuando seq es 12, asignamos más tareas que hilos tenemos para computar y el resultado puede empeorar. El tiempo de ejecución también puede verse desfavorecido si el número de tareas es menor que el número de hilos, tendremos un grupo de hilos desocupados y desperdiciando tiempo solo por la gestión que conllevan. La conclusión que podemos extraer es que lo ideal sería tener tantas tareas, imágenes a procesar, como hilos haya en la región paralela y cuanto mayor sea esta proporción, mayor rendimiento obtendremos.
Para verificar si se alcanza mejor rendimiento modificando el número de threads según el tamaño de la secuencia de imágenes, generamos una gráfica comparativa para visualizar los resultados.

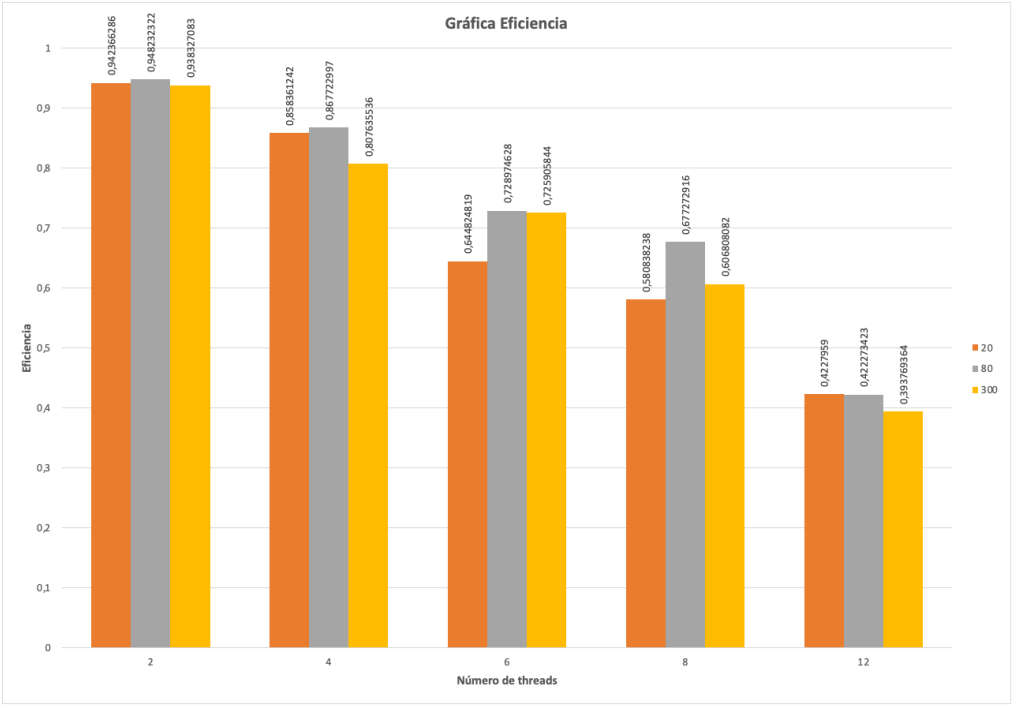


Es evidente que en cualquier situación, obtenemos mejor ganancia salvo con 8 hilos porque no hemos repetido las ejecuciones. Podemos destacar que con seq 12, se ha conseguido superar la barrera de speedup 5. La tendencia de la gráfica sigue siendo la misma y seguiría escalando si nuestra máquina tuviese más cores.

Para la segunda prueba a realizar, mantendremos la relación que acabamos de explicar con el número de secuencia y número de threads, pero ejecutaremos con distintas dimensiones de video. En concreto, la altura y la anchura de las imágenes se mantendrán para favorecer la consistencia y respetar un aspecto homogéneo con las anteriores pruebas. El único atributo que modificaremos será el número de imágenes del video, Max, que adoptará los valores 20, 80 y 300. El propósito es visualizar si a mayor o menor tamaño de vídeo con las especificaciones dadas la ganancia mejora o empeora. También se detallará la eficiencia resultante.



Es distinguible que las ganancias dictan la misma tendencia antes explicada, a mayor secuencia con igual número de threads, mejor resultado. El vídeo con 20 imágenes únicamente, presenta buen rendimiento con 2 y 4 threads. Después, la ganancia crece progresivamente sin estabilizarse. En cuanto al vídeo con 80 imágenes, se consigue mayor speedup con todos los números de threads y por este motivo, se encuentra en la región de mejores vídeos a procesar con las dimensiones de imagen dadas. Por último, el video más extenso que hemos filtrado refleja una aspecto similar al previo pero con ganancias menores. A estas alturas puede comenzar reflejándose el overhead que produce paralelizar, aunque siempre devolverá un resultado mejor que el secuencial. Finalmente echaremos un vistazo a las eficiencias registradas.



Aunque hayamos obtenido mejores tiempos con mayor número de threads, con 2 y 4 threads se establecen comportamientos mucho más eficientes que el resto.
A media que incrementamos los threads en la región paralela, el tiempo de ejecución también se ve penalizado por el overhead que implica el reparto de trabajo. Ya hemos contemplado en la primera prueba que los hilos mantenidos suponen un coste.

4. Valoración

Personalmente, esta práctica para mí ha sido única entre todas las realizadas. Ha supuesto un proceso inmersivo, beneficioso y a la vez desafiante. Podría quedarme con algún momento en particular que me haya gustado pero todo el desarrollo ha sido muy interesante. Desde averiguar el funcionamiento del código hasta implementar una solución, fue muy entretenido.
Me hubiese gustado enterar en detalle con la paralelización mediante pipelines y ver hasta dónde se podrían mejorar los tiempos. De la misma forma, me interesaba haber encontrado alguna optimización mejor para la función de gauss. Sin embargo, estoy bastante contento con los resultados obtenidos y las cosas aprendidas.