Arquitectura de Computadores

GRADO EN INGENIERÍA INFORMÁTICA

OpenMP

Curso 2020/2021

Jorge Rodríguez Fraile, 100405951, Grupo 81, 100405951@alumnos.uc3m.es
Carlos Rubio Olivares, 100405834, Grupo 81, 100405834@alumnos.uc3m.es
Ivan Serrano García, 100405836, Grupo 81, 100405836@alumnos.uc3m.es
Francisco José Ruiz De La Cruz, 100405807, Grupo 81, 100405807@alumnos.uc3m.es

Índice

Introducción

En el siguiente documento se representa la memoria del proyecto. El proyecto consiste en la realización de un programa en el lenguaje de programación C++, tanto en secuencial como en paralelo, que permite la copia de fotos en bmp a otro directorio distinto y la aplicación de dos filtros. Con el primer filtro se aplica una difusión gaussiana a la foto que tiene como objetivo reducir la nitidez de la foto. El segundo filtro se trata del operador Sobel, que es un filtro de realce, y cuyo objetivo es ayudar a detectar los bordes de la foto.

El contenido de esta memoria se organizará en varios bloques. Estos bloques son: versión secuencial, versión paralela, evaluación del rendimiento del programa, las pruebas realizadas y por último las conclusiones.

En los bloques de versión secuencial y paralela del programa se explicará la implementación de las funciones y la optimización de código que se ha llevado a cabo en las dos versiones.

En el bloque de evaluación del rendimiento analizaremos la eficiencia del programa con y sin optimizaciones del compilador.

En el bloque de pruebas realizadas analizamos el correcto funcionamiento del programa con distintas pruebas que abarquen los diferentes casos en los que el programa debería funcionar, así como en los casos que no debería hacerlo.

Y por último en las conclusiones, analizaremos los problemas encontrados durante la realización de la práctica, la dificultad de la misma y que hemos aprendido durante su realización.

Versión secuencial

Implementación

Copy

Esta funcionalidad la hemos conseguido, abriendo la imagen de la carpeta raíz. Primero leemos la cabecera de tamaño fijo, que nos permite conocer cuál es el tamaño de la imagen para poder almacenarla localmente y operar con ella directamente.

El tiempo que comprende desde que se abre el fichero imagen bmp hasta que se almacena en las variables locales la cabecera y el cuerpo de la imagen es el tiempo de carga, dentro de este tiempo se hacen las comprobaciones del formato de la imagen.

En este caso no se aplican modificaciones sobre los datos, por lo que nos limitaremos a crear el fichero en la carpeta de salida con el mismo nombre que el original, y ahora si escribimos la imagen leída.

Primero la cabecera se va escribiendo parámetro a parámetro según las características de la imagen, y a continuación escribimos los datos de la imagen ajustándose a la nueva cabecera.

En este caso no hay tiempo de operación, ya que solo se carga y se almacena, el tiempo de almacenamiento es la creación y escritura del fichero de salida.

Gauss

Para esta implementación del código hemos decidido usar una función para luego poder usarla en Sobel, ya que debemos utilizar también Gauss en dicho apartado. Como parámetros de esta función tenemos dos ints que serán el alto y el ancho de la imagen, los datos de la imagen sin padding se almacenan en la variable data, y los datos de imagen que resultan, se denominan res.

A los datos de imagen se les quita los bytes de padding leyendo de width en width, y saltándose width%4 bytes que representan el rellenado al final de una línea.

Al inicio de la función, creamos la matriz m 5x5 que nos servirá de máscara. Una vez creada, empezamos a recorrer los bits de la imagen, en cada bit creamos tres variables que almacenarán los datos RGB resultantes del pixel.

Una vez situados en el pixel, comprobamos que no está fuera de la imagen o cualquier error del estilo, y aplicamos las transformaciones indicadas en gauss en cada uno de los valores RGB del pixel. Una vez terminados los sumatorios, accedemos a los 3 valores RGB del píxel y les asignamos los valores obtenidos en las iteraciones anteriores que han sido guardados nuestras variables, pero esta vez divididos por w, que en este caso es 273.

Una vez explicada la función, volvemos al código principal. Lo primero que se hace es crear la cabecera de res tal y como lo hace la función copy, inicializamos el timer para gauss y llamamos a la función gauss con sus parámetros necesarios.

Ahora que tenemos nuestro encabezado res con el resultado, finalizamos el timer de gauss para ver cuánto se ha tardado, y procedemos a iniciar el contador de store y copiar la imagen resultante añadiendo el padding que le hemos quitado durante la carga. Una vez terminado, cerramos el timer.

Por último, hacemos las diferencias de los timers para ver el tiempo de cada proceso y los sumamos para obtener el total. Al hacer esto, imprimimos todos los resultados obtenidos, y en la carpeta output tendremos todos nuestros resultados.

Sobel

Para implementar esta función utilizamos los resultados obtenidos de aplicar la función gauss.

Los parámetros introducidos en esta función tenemos el alto y el ancho de la imagen en dos ints, la cabecera de la imagen y también el resultado de hacer la función gauss.

Creamos dos matrices 3x3, que se usarán como máscaras, iniciadas con los valores aportados en el enunciado de la práctica. Una vez hecho esto, se crearán dos bucles for anidados, que se usarán para recorrer el ancho y el alto de la matriz, y para guardar en cada pixel los 3 colores RGB que lo componen, y seguidamente guardará cada pixel de la imagen. Después se inicializan las variables locales donde se realizarán las operaciones que almacenarán los valores de la imagen pixel a pixel, que después se almacenarán en la variable resultado. Dentro de los dos bucles, se crearán otros dos bucles for anidados que se usarán para realizar las operaciones necesarias que aplicarán los cambios de sobel a los pixeles, y se controlará la detección de los bordes de la imagen usando un condicional if.

En el tiempo que comprende desde que se abre el timer de la función sobel hasta que se cierra, se realizan los 4 bucles anidados que se han explicado previamente.

Para terminar, se escribe la cabecera correspondiente a la imagen que hemos modificado, después como hacíamos en gauss leemos el resultado de sobel de width en width y vamos añadiendo el padding correspondiente (width%4), de esta manera la imagen concuerda con el formato inicial.

Lectura y control de parámetros

Para la lectura y control de los parámetros de entrada hemos identificado uno por uno los diferentes errores que se pueden dar al pasar los parámetros por el comando de entrada. En primer lugar, hemos comprobado con un if si el número de parámetros introducidos en el comando es correcto, ya que este número debe ser 4.

Posteriormente hemos comprobado que el parámetro que indica la operación que se va a realizar sea una de las operaciones válidas. Para ello hemos usado un if que comprueba que el parámetro 1 de argv sea igual a “copy”, “gauss” o “sobel”.

También comprobamos que los directorios de entrada y salida existan, para ello comprobamos que la variable de tipo DIR que guarda el resultado de intentar abrir el directorio pasado por parámetro no sea “NULL”.

Si el directorio existe, lo abrimos y leemos las imágenes, empezando por sus cabeceras y comprobamos que sean correctas y las imágenes sean de formato bmp. Leyendo los bytes de la cabecera bmp extraemos la información útil a la hora de manejar los parámetros de las fotos, es decir, el tamaño de la foto, su altura y su ancho y también comprobamos que el bmp sea de 24 bits por punto leyendo los bytes 28 y 29, que tenga solo un plano leyendo los bytes 26 y 27 y que el byte de compresión sea 0 leyendo los bytes del 30 al 34. Por último, comprobamos que no se produce ningún error al leer el cuerpo de la imagen.

También se ha verificado que las imágenes de la carpeta de entrada sean .bmp, de tal manera que si no lo son sale un error y pasa a la imagen siguiente.

Optimización del código

General

Para optimizar la carga y almacenamiento de las imágenes, a la hora de leerlas y escribirlas en el directorio de salida, hemos diferenciado entre las imágenes que tienen padding y las que no. Para obtener el padding hemos hecho width%4. Cuando leemos una imagen, si tiene padding, procedemos a leer la imagen línea por línea ignorando el padding que se almacena al final de cada línea. Posteriormente, tras haber realizado gauss o sobel, escribimos la imagen resultado en el directorio de salida. Para ello, escribimos primero la cabecera de la imagen y por último escribimos la imagen resultado línea por línea, teniendo en cuenta que al final de cada línea tenemos que escribir el padding ignorado a la hora de leer la imagen.

En el caso de que la imagen no tenga padding la lectura y escritura de la imagen es más simple, ya que no se tiene en cuenta el padding que hay que ignorar y que posteriormente hay que reestablecer, lo que hace que sea mucho más rápido que escribir línea a línea.

En general hemos almacenado los datos en arrays en vez de matrices separadas, lo que ha conllevado que los 3 bytes de colores están consecutivos, que dificulta un poco a la hora de

coger elementos para modificarlos, pero da buenos resultados a la hora de almacenarla en memoria, lo podemos hacer directamente.

Todos los bucles recorren nuestro array de datos de imagen de la manera más eficiente posible, no merecería la pena hacer intercambio de bucles.

Gauss

Hemos optado por una optimización de fusión de bucle, donde hemos intentado organizar el código en un bucle y hacer todas las operaciones ahí, para evitar una complejidad mayor, además hemos intentado simplificar los valores resultado de la función, con el array res, que solo almacena los bytes de datos de la imagen resultado.

Sobel

Se ha seguido una optimización de fusión de bucle, al igual que en apartado anterior, donde toda la función se ha desarrollado dentro de un bucle for, y, además, todos los elementos necesarios están almacenados de manera ordenada para que se recorra con mayor facilidad, los elementos accedidos están espacialmente cercanos y al ser secuencial no se pierde de caché la imagen para acceder a otro bloque.

Versión paralela

Implementación

Para implementar la versión paralela hemos llevado al fichero image-par el contenido de image-seq para así poder ejecutar los comandos de OpenMP. Para desarrollar como se ha implementado el código se intentará explicar desde el punto de vista cronológico, ya que de esta manera será más fácil visualizar cómo se ha implementado esta parte de la práctica.

La primera idea que se tuvo fue hacer una división del bucle en los fors anidados que se encontraban en las soluciones de gauss y sobel, ya que la complejidad del código venía de ahí, por lo que se introdujo el comando #pragma omp parallel for al principio de los bucles de ambos procesos, de esta manera cada hilo se reparte un determinado número de píxeles que transformar; como era de esperar el resultado era el esperado y se mejoró el tiempo de ejecución. Un paso más para mejorar la paralelización en estos bucles sería cambiar el 'schedule', por lo que se prosiguió a estudiar cuál de los 3 tipos era el mejor, tras varias pruebas se llegó a la conclusión de que 'dynamic' era la opción correcta ya que la idea principal que se tenía para paralelizar era que cada hilo fuera 'recogiendo' iteraciones de una cola, repartiéndose así la imagen.

Después de implementar estos cambios el tiempo de ejecución mejoró notablemente, aunque se intentó mejorar la sección de copy sin mejor resultado, ya que la implementación de carga de imágenes es secuencial en el código, por lo que es inútil paralelizar esta parte. Aun así, se han estudiado diversas posibilidades para ver la mejor combinación de paralelización en el código, todas estas pruebas y combinaciones se explicarán en el apartado de pruebas con su debida explicación, ya que no han formado parte del código final.

Optimización del código

Todas las optimizaciones realizadas en esta parte vienen de la parte secuencial, no se ha realizado ninguna optimización adicional que no haya sido incluida previamente en la sección de secuencial.

Evaluación de rendimiento

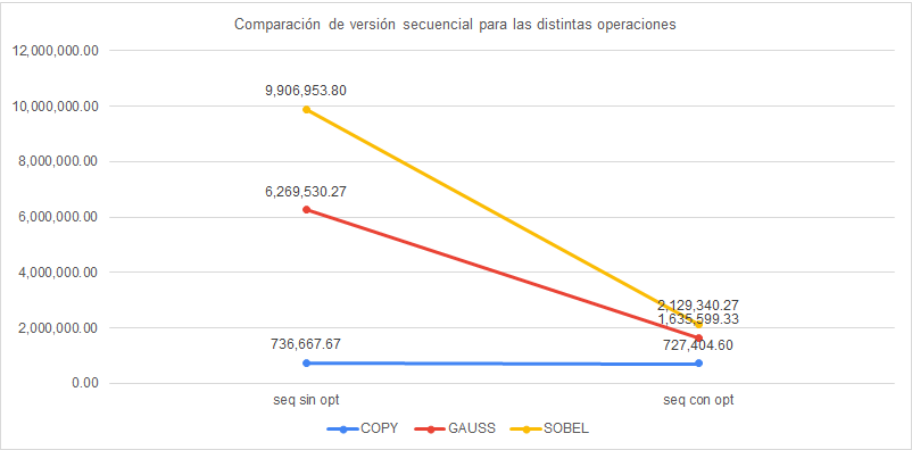
Características del procesador

Para la evaluación de esta aplicación se ha empleado un computador con un procesador i7-5820K, que cuenta con 6 cores sin Hyperthreading y con hilo por core. El orden de los bytes es Little-Endian y direcciones de 46 bits. En cuanto a la memoria caché, se tienen dos cachés de primer nivel, una para datos con 192 KiB y otra para instrucciones con la misma capacidad. Por último, se poseen otras dos cachés una L2 de 1.5 MiB y una L3 de 15 MiB.

Resultado de versión secuencial con y sin optimizaciones

En la gráfica que se muestra a continuación podemos observar los tiempos totales de ejecución del programa secuencial, con y sin optimizaciones, para cada función del programa. Como podemos ver en la gráfica la ejecución del programa secuencial compilado sin optimizaciones es mucho peor para las funciones de Gauss y Sobel que compilando la versión secuencial con optimizaciones. Esto se debe principalmente a la gestión del padding que hay que hacer para las fotos que tienen padding, que como hemos podido comprobar, son las fotos que más tardan en procesarse. Estas fotos se van escribiendo línea a línea para poder añadir el padding al final de cada línea y es por esto por lo que consumen un tiempo mayor. Copy no varía mucho ya que no tiene que realizar ningún proceso costoso como realizar un bucle for o bucles anidados.

La aplicación de optimizaciones al compilar hace que se mejore mucho las velocidades de recorrido de bucles, es por ello que observamos mucha mejora en Gauss y Sobel de la versión secuencial optimizada.



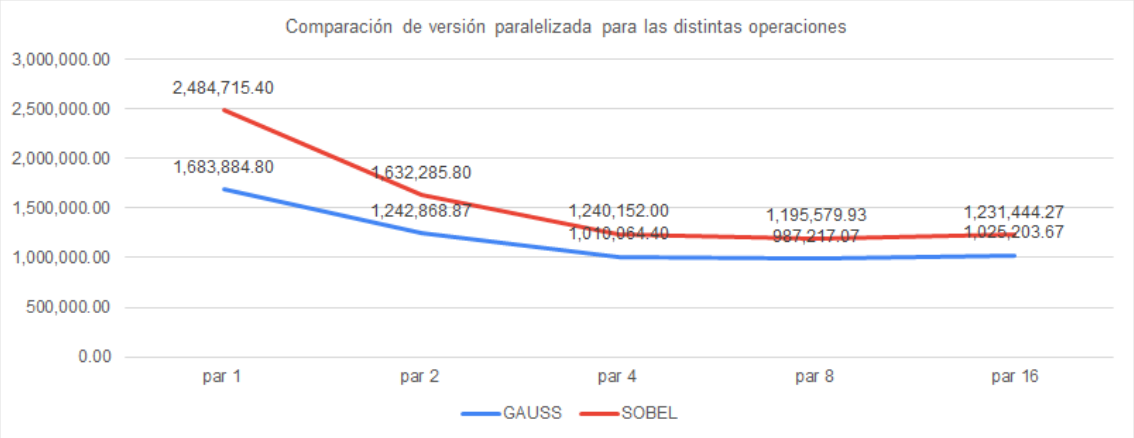
Resultado de versión paralela optimizada

En la versión paralelizada no se compara copy, ya que el tiempo de copy es siempre el mismo para cualquier número de hilos utilizados, debido a que la función copy no se ha paralelizado.

Para comparar los resultados obtenidos en la ejecución de la versión paralela, utilizando para ello un número distinto de hilos en cada uno, hemos creado la siguiente gráfica. En ella podemos observar que el tiempo empleado para ejecutar va reduciéndose a medida que vamos aumentando el número de hilos que entran en juego, tanto para la función gauss como para la función sobel.

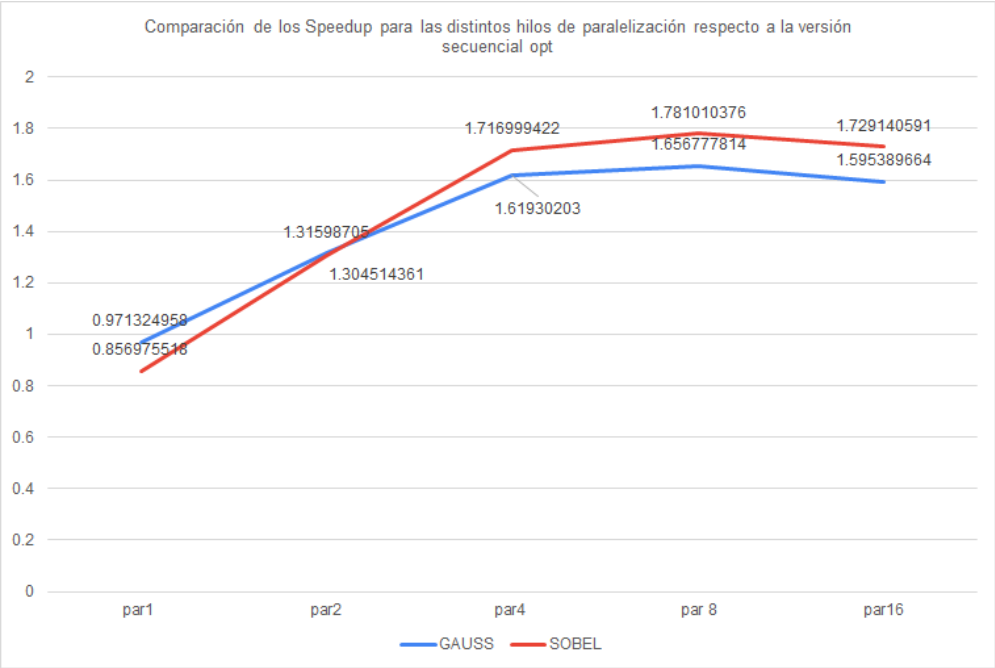
Si analizamos la gráfica, podemos observar que el tiempo empleado para la ejecución usando 4 y 8 hilos es muy parecido. Por esto mismo pensamos que no es demasiado rentable adquirir un procesador que cuente con un número superior a 4 hilos, ya que el poco tiempo que te ahorras al ejecutar, no compensa el coste técnico que esto requiere. Esto supone que lo más eficiente sería utilizar 4 hilos.

Si miramos el tiempo que tarda en ejecutar usando 16 hilos, vemos que es mayor que el que tarda 8 hilos. Esto se debe a que, para imágenes de un tamaño pequeño, no merece la pena dividir tanto el trabajo, ya que es más costoso la creación del doble de hilos que el procesar la imagen.



En la siguiente gráfica se muestra el speedup conseguido al paralelizar con diferente número de hilos respecto a la versión secuencial optimizada.

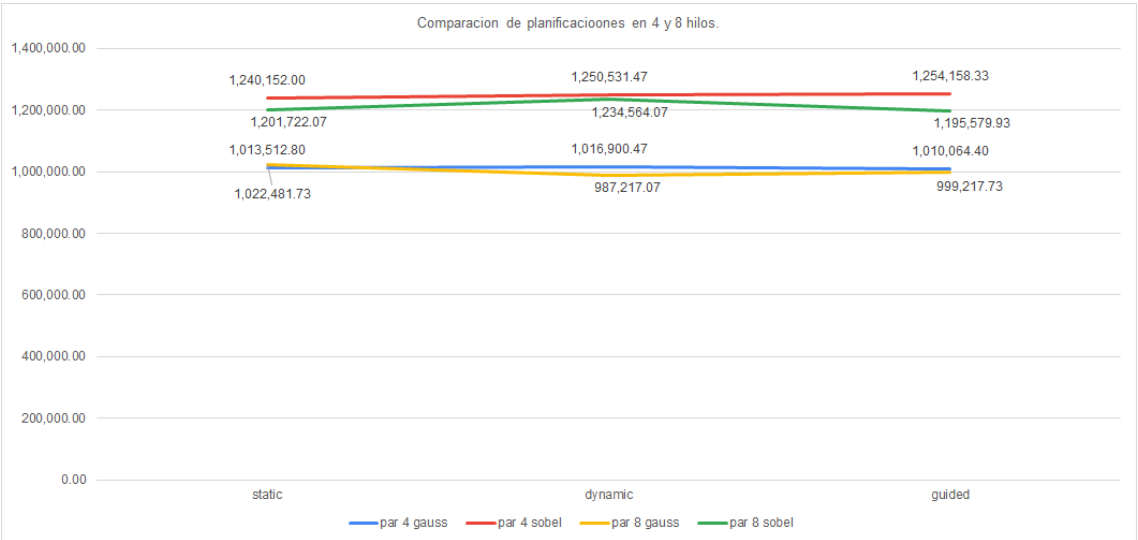
Como podemos observar la gráfica nos dice que paralelizar con 1 hilo gauss y sobel hace que empeore el rendimiento. Esto se debe a que paralelizar con un hilo es prácticamente igual a no hacerlo, con la diferencia de que hay que crear y destruir el hilo, es por ello que la versión secuencial es mejor en este caso, ya que no tiene que crear y destruir hilos. Para todas las demás versiones paralelizadas con distinto número de hilos se produce una mejora notable. Cabe destacar que el speedup decae para la versión de 16 hilos ya que como hemos mencionado anteriormente, no mejora el rendimiento crear y destruir tantos hilos si el programa no es tan complejo como para dividir tanto la carga de trabajo.



Impacto de la planificación

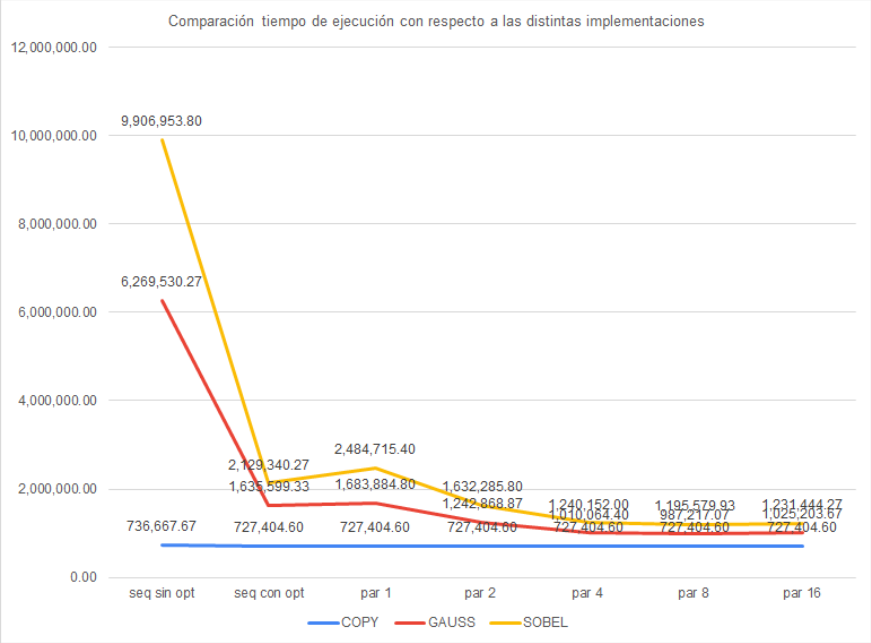
Para las distintas ejecuciones, hemos usado distintas planificaciones para cada una de ellas, dependiendo de los resultados que nos proporcionaban. En el caso de la ejecución usando 4 hilos en la función gauss usamos la planificación guided ya que es la que mejores resultados nos proporciona. En el caso de 4 hilos en la función sobel usamos la planificación static. En la de 8 hilos con la función gauss usamos dynamic, y en la de 8 hilos en la función 8 utilizamos la planificación guided.

Analizando la siguiente gráfica podemos observar que la planificación no influye lo suficiente para sacar una conclusión clara de cuál planificación es la mejor para cada caso. Esto se debe a que hemos cogido un valor de media para cada caso tras la realización de 15 ejecuciones y puede darse que en una de las ejecuciones el tiempo que tarda gauss con dynamic para 4 hilos sea menor que lo que tarda con la planificación static, pero en otra ejecución pase lo contrario. Es por ello que no se puede sacar una conclusión clara sobre que planificación es mejor.



Análisis de resultados

Como podemos observar en la gráfica de los resultados totales para todas las versiones, la versión más óptima de nuestro programa, es la paralelizada con 8 hilos. Con esta gráfica podemos concluir que las versiones secuenciales son peores que las paralelizadas, ya que no aprovechamos el rendimiento que nos puede dar el uso de varios hilos. Con las versiones paralelizadas podemos observar que a medida que aumenta el número de hilos, mejora el tiempo de ejecución, excepto para el caso de 16 hilos, ya que en este caso se están creando muchos hilos para una carga de trabajo para la que no merece la pena utilizar tantos hilos.



Pruebas realizadas

Comando con formato invalido:

Para esta prueba escribimos un argumento menos de los requeridos y comprobamos que nos muestre un error. Como podemos comprobar, se muestra un error diciendo que el formato del comando está mal introducido.

Hacer copy y comparar con la original:

Al aplicar la función copy a las imágenes que se encuentran en la carpeta input podemos observar que se copian de manera correcta en la carpeta output.

Hacer gauss y comparar con solución:

Para hacer esta prueba ejecutamos el script solución proporcionado y posteriormente duplicamos las imágenes resultantes en la carpeta output para facilitar la comparación entre nuestro resultado y el resultado del programa solución. Tras duplicar las fotos podemos ejecutar

nuestro programa y comparar nuestros resultados con los resultados del programa solución. Como se puede observar las dos imágenes son exactamente iguales por lo que la función gauss de nuestro programa funciona correctamente.

Hacer sobel y comparar con solución:

Para hacer esta prueba realizamos los mismos pasos que en la prueba anterior y ejecutamos nuestro programa con la función sobel y cómo podemos observar las fotos resultantes son iguales a las que se crean al ejecutar el programa solución por lo que interpretamos que nuestro programa ejecuta correctamente la función sobel.

Directorio inexistentes:

Para realizar esta prueba pasamos por parámetros un nombre de directorio no existente, ya sea de origen o destino, y comprobamos si nos muestra un error. Como podemos observar al realizar esta prueba efectivamente nos salta un error que nos dice que no se pudo abrir el directorio.

Imagen con otro formato:

Para realizar esta prueba hemos añadido una imagen que no es de formato bmp dentro de la carpeta input y hemos ejecutado el programa para comprobar que se ejecuta el programa para todas las demás fotos menos para la foto con otro formato. Como podemos observar se ejecuta el programa para todas las demás fotos excepto para la que no tiene el formato adecuado, en cuyo caso, muestra un error que dice que el formato de la foto no es el esperado.

Imagen con más de 1 plano:

Para realizar esta prueba hemos añadido una imagen que no cumpla con esta característica en la carpeta input. Al ejecutar el programa podemos observar que se muestra un error al intentar procesar la imagen que no es válida y el programa sigue ejecutando las imágenes que sí son válidas.

Imagen con bit count distinto de 24:

De la misma manera que en la anterior prueba, añadimos una imagen en el directorio input que no cumpla con la característica especificada. Como se puede observar, al ejecutar el programa, se procesan todas las imágenes menos la imagen que no es válida, para la cual muestra un error diciendo que el bit count no es 24.

Imagen con compresión:

Para realizar esta prueba hemos modificado la función copy para que, al escribir la cabecera, el byte de compresión sea distinto de 0. Una vez modificada la cabecera copiamos una foto en la carpeta input y ejecutamos el programa, esta vez sabiendo que el byte de compresión es distinto

de 0, por lo que el programa debe procesar todas las fotos menos la foto que tiene el byte de compresión cambiado. Como podemos observar, al ejecutar el programa se procesan todas las fotos menos la foto que no cumple con la característica mencionada en la que se muestra un error al intentar procesar diciendo que el byte de compresión es distinto de 0

Imagen con ancho no múltiplo de 4, es decir, necesita padding:

1 byte de padding: Para realizar esta prueba añadimos una imagen en input cuyo ancho módulo 4 sea igual a 1. Si comparamos la imagen procesada con la imagen que se muestra cuando ejecutamos el script de la solución, podemos observar que la imagen se ha procesado correctamente.

2 bytes de padding: Para esta prueba realizamos el mismo proceso que en la anterior, pero esta vez el ancho de la imagen módulo 4 debe ser 2. Si comparamos la imagen procesada con la imagen que se muestra cuando ejecutamos el script de la solución, podemos observar que la imagen se ha procesado correctamente.

3 bytes de padding: Para esta prueba realizamos el mismo proceso que en la primera prueba, pero esta vez el ancho de la imagen módulo 4 debe ser 3. Si comparamos la imagen procesada con la imagen que se muestra cuando ejecutamos el script de la solución, podemos observar que la imagen se ha procesado correctamente.

Pruebas Versión Paralela:

Antes de entrar en las diferentes pruebas que hemos realizado, cabe mencionar que se ha comprobado que no aparecen condiciones de carrera en ninguno de los puntos críticos que se han paralelizado. Además de volver a comprobar que se pasan todas las pruebas de la versión secuencial

División de las operaciones de Gauss en secciones: Se han intentado paralelizar las operaciones sobre los bytes en Gauss mediante secciones, pero el tiempo de ejecución era peor, ya que los hilos tenían que acceder al mismo píxel para una vez paralelizar ahí, lo que obviamente es bastante menos efectivo que hacer que cada hilo acceda a un pixel diferente.

División de las operaciones de Sobel en secciones: Para poder afirmar la teoría que se había planteado en los resultados de la prueba anterior, se han aplicado los mismos parámetros en la función Sobel, y como era de esperar los resultados también empeoraron respecto a nuestra paralelización inicial.

Planificación static de los bucles de recorrido de píxeles de Gauss: Se ha aplicado el valor static para el schedule de los bucles de recorrido de la imagen, el resultado es mejor que en la opción secuencial, pero al tener prefijados los píxeles sobre los que deben trabajar cada hilo es algo más lento que la opción dynamic, donde se reparte dinámicamente los píxeles en vez de cambiar siempre los mismos.

Planificación guided de los bucles de recorrido de píxeles de Gauss: El siguiente paso lógico era aplicar la condición guided a los bucles, el resultado es bastante parecido a static, ya que suponemos que el tamaño de bloques que se van asignando es al principio grande (seguramente parecido al aplicado en static) y después va decreciendo.

Planificación dynamic de los bucles de recorrido de píxeles de Sobel: En esta prueba hemos dejado la planificación dynamic para Sobel, pero empeora respecto al uso de static, creemos que porque al estar ya aplicado en Gauss, el efecto que produce en Sobel es mínimo.

Planificación guided de los bucles de recorrido de píxeles de Sobel: Teniendo en cuenta los resultados anteriores, guided no aporta un mejor resultado a static en esta implementación de código, como se ha explicado anteriormente.

Región ordered para incremento de las variables en las operaciones de Gauss: Una de las pruebas que vimos más adecuadas para los bucles que realizan las operaciones en los píxeles eran las de aplicar una región ordered para las variables que se incrementan, aunque sólo se consiguió aplicar la región ordered a una variable, no a las 3 con las que trabajamos, por lo que no hubo mejora alguna en cuanto al tiempo de ejecución.

Región ordered para incremento de las variables en las operaciones de Sobel: Se ha seguido el mismo proceso para Sobel, aunque de nuevo, sólo se puede aplicar a una sola variable, por lo que el resultado es similar a la prueba anterior.

Conclusiones

En esta práctica hemos podido aprender a trabajar con ficheros de imágenes, cosa que nunca antes hemos tenido la oportunidad de hacer, y hemos aplicado un gran tiempo de nuestro trabajo en intentar comprender de la mejor manera posible la versión secuencial de nuestro programa, para poder sacar el rendimiento máximo a la versión paralelizada.

En cuanto a OpenMP, nos ha resultado un conjunto de comandos bastante útil, ya que con una simple línea puedes paralelizar un bucle completo, por ejemplo. En definitiva, la herramienta es bastante accesible y fácil de usar, además las diapositivas que se nos han adjuntado también han servido de ayuda.

Por último, nos gustaría resaltar que hemos tenido diversos problemas a la hora de resolver el padding en las fotos, ya que al principio lo manejábamos dentro del bucle de gauss, pero acabamos resolviendo este problema aplicándolo fuera de gauss y sobel, en el hilo de ejecución principal.