Arquitectura de Computadores
Práctica de programación paralela con OpenMP

Realizada por:
Beatriz Cifuentes Fernández; NIA: 100383230; Grupo 80
Javier Diaz Leyva; NIA: 100383310; Grupo 80
Eduardo de Andrés Tabernero; NIA: 100363553; Grupo 80
Marcelino Tena Blanco; NIA: 100383266; Grupo 80

Índice

1. Introducción

El presente documento trata sobre el modelado e implementación de un programa que realiza copias y máscaras en ficheros BMP. El programa tendrá dos versiones: una será secuencial y la otra será paralela mediante el uso de la librería openMP. El objetivo de la práctica es reconocer mecanismos de optimización de recursos (entre ellos el tiempo) y rendimiento mediante la comparación entre las dos versiones del programa más una versión sin optimizar. El siguiente documento se estructura de la siguiente forma:

- **Versión secuencial**: explica al principio como funciona el código tanto secuencial como paralelo al principio. Después, mediante diagramas de flujo, explica cómo funcionan los métodos realizados. Los métodos Gauss y Sobel documentados en esta parte son solo para la versión secuencial. Los demás métodos son similares en ambas versiones.
- **Versión paralela**: explica cómo se han diseñado e implementado los métodos Gauss y Sobel de forma paralelizada.
- **Evaluación de rendimiento**: mediante gráficos y uso de tres imágenes de distintos tamaños evaluamos el rendimiento de cada una de las versiones añadiendo la versión sin optimizar.
- **Pruebas**: esta parte dispone de una tabla donde se especifica la prueba, se comenta cómo se va a realizar, se especifica que debe salir y por último la salida obtenida. De esta forma comprobamos tanto la versión secuencial como la versión paralela para comprobar su perfecto funcionamiento.
- **Conclusiones**: parte final donde contamos si cumplimos los objetivos de la práctica, la dificultad de la misma o los diversos obstáculos que hemos tenido que superar para poder lograrla.

Equipo utilizado:		
SO	Ubuntu 20.04.1 LTS	
Compilador	g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0	
Procesador	Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz	
	Velocidad del reloj	3.00GHz
	Memoria Caché	L1: 256 KiB
		L2: 1 MiB
		L3: 6 MiB
	Número de Núcleos/hilos	Cores = 4; threads = 4
	Tamaño de palabra	64 bits
Memoria	8GB DIMM DDR4 Synchronous 2133 MHz (0,5 ns), tamaño de palabra de 64bits	
Placa Base	Gigabyte Technology B250M-DS3H-CF	
Disco Duro	KINGSTON SA400S3, SSD 111GiB	

2. Versión secuencial

Todo programa consta de tres partes: entrada, proceso y salida:
Entrada: Para la entrada, tenemos diversas partes

- Argumentos de entrada: **Operador Ruta_de_entrada Ruta_de_salida**
  - **Operador**: debe ser copy, gauss o sobel
  - **Ruta_de_entrada**: ruta donde se obtendrán los ficheros (debe ser una carpeta)
  - **Ruta_de_salida**: ruta donde se añadirán los nuevos ficheros (debe ser una carpeta)
- Ficheros de la ruta de entrada: son los ficheros que están en **Ruta_de_entrada**.

Salida: para la salida, tenemos:

- Salida por pantalla: obtenemos información sobre los argumentos pasados por entrada y sobre cómo ha ido la ejecución del programa:
  - En caso de que sea **correcta** aparecerá el tiempo que ha tardado en realizar el **Operador** designado, en leer la imagen y en escribirla, además de otras salidas.
  - En caso de que sea **incorrecta**, aparecerá un mensaje sobre qué error ha ocurrido, dando lugar al cierre del proceso.

  El programa avanza fichero por fichero, es decir, si el primer fichero que recoge hace que el funcionamiento del programa sea correcto, aparecerá salida correcta a pesar de que el segundo fichero recogido diera error, donde entonces saldría el error del segundo y se cerraría el proceso. De esta forma, conocemos qué está ocurriendo en el programa en el procesado de cada fichero.
- Ficheros de la ruta de salida: son los ficheros que fueron procesados mediante **Operador** y guardados en la ruta elegida por el usuario: **Ruta_de_salida**

Proceso: un resumen del proceso que realiza el programa (más tarde hablaremos más en profundidad) es el siguiente:

1. Comprueba que todos los argumentos pasados son correctos.
   1.1. Si son incorrectos, el proceso acaba.
2. Comprueba que las carpetas existen, abriéndose.
   2.1. Si no existen o no pueden abrirse, el proceso acaba.
3. Consigue un fichero de la carpeta.
   3.1. Si no quedan más ficheros por leer, el proceso acaba.
4. Abre el procesamiento de la imagen.
   4.1. Abre el fichero pasado.
   4.2. Lee el fichero pasado, comprobando la cabecera..
   4.3. En caso de que Operador sea gauss o sobel:
      4.3.1. Pasa el fichero leído para hacerle el filtro de gauss.
      4.3.2. En caso de que Operador sea sobel
         4.3.2.1. Pasa el fichero obtenido por gauss y le aplica la transformación de sobel.
   4.4. Escribe el nuevo fichero generado.
5. Imprime por pantalla el tiempo guardado.
6. Vuelve al paso 3

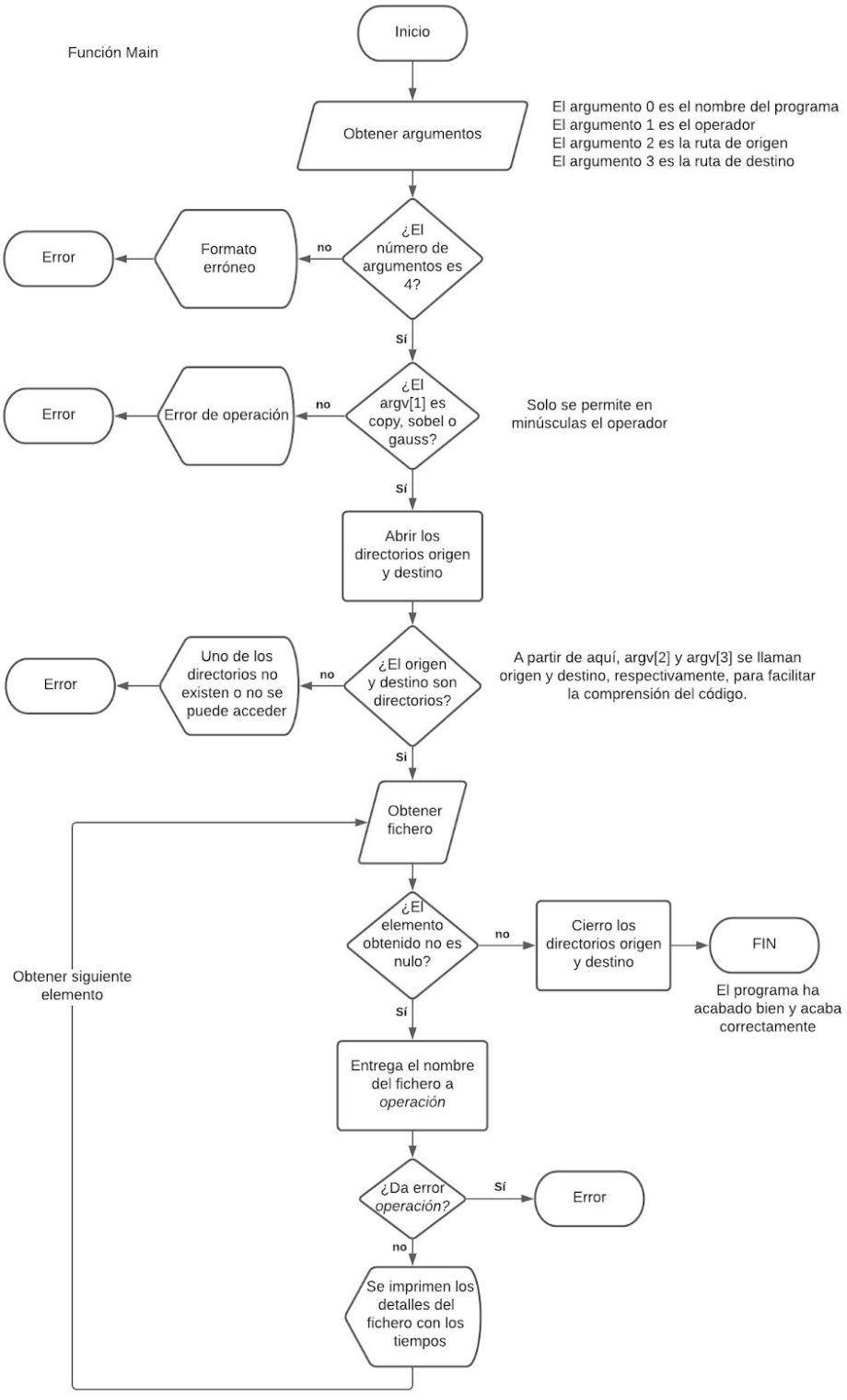
Los pasos 4, 4.2, 4.3.3, 4.3.2.1 y 4.4 guardan el tiempo transcurrido.

2.1 Implementación

El programa cuenta con seis partes, las cuales son: *main, operación, lectorBMP, escritorBMP, gauss y sobel*. A continuación, explicamos cada una:

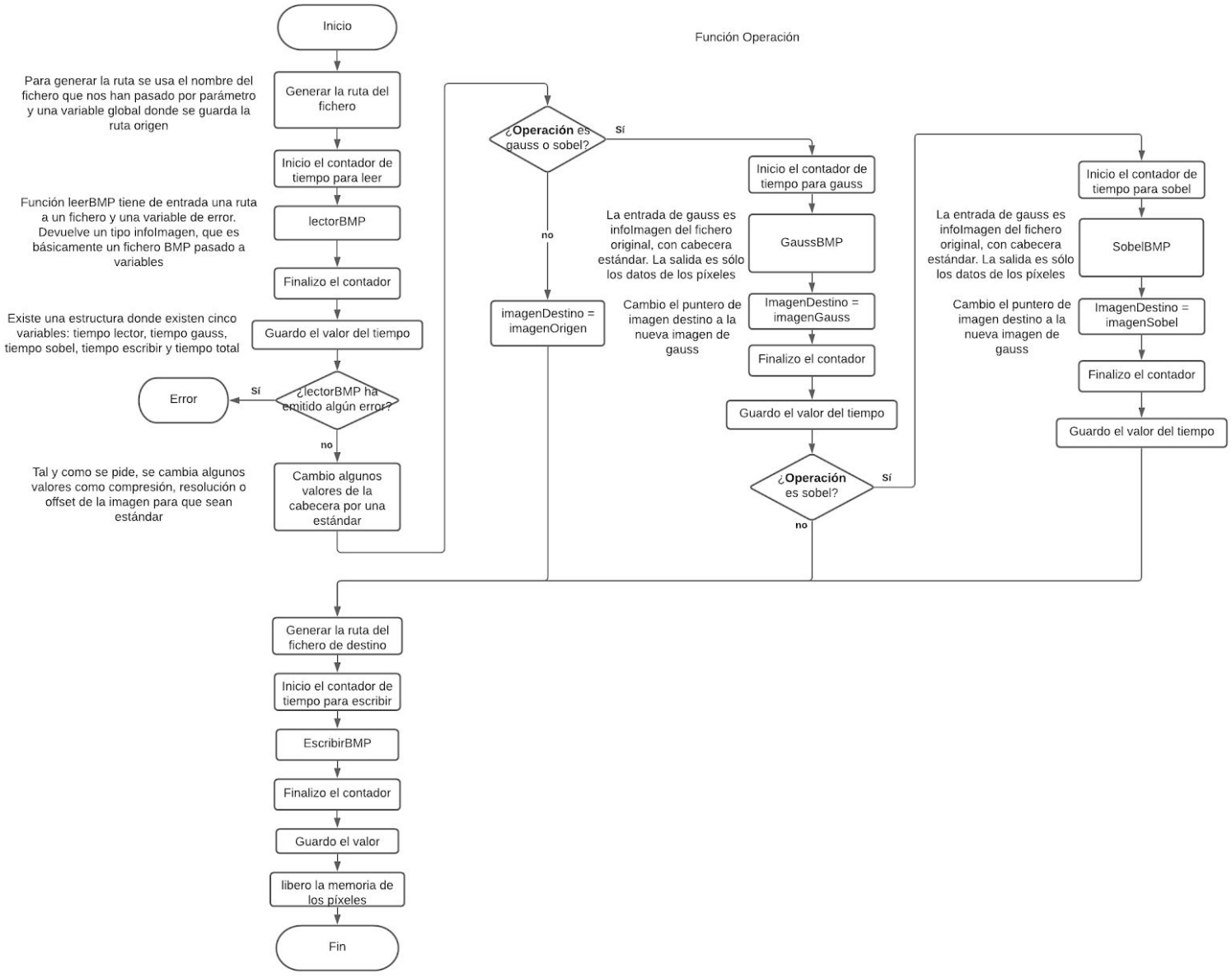
2.1.1 *Main*

El main es la función principal del programa. Se encarga de comprobar si los argumentos pasados son correctos, si las carpetas existen y de conseguir ficheros, los cuales se pasarán como parámetro a *operación.*



2.1.2 *Operación*

La función operación realiza la lectura y escritura de un fichero BMP siempre y sobel y gauss cuando se especifica, teniendo en cuenta que gauss se realiza también en sobel.

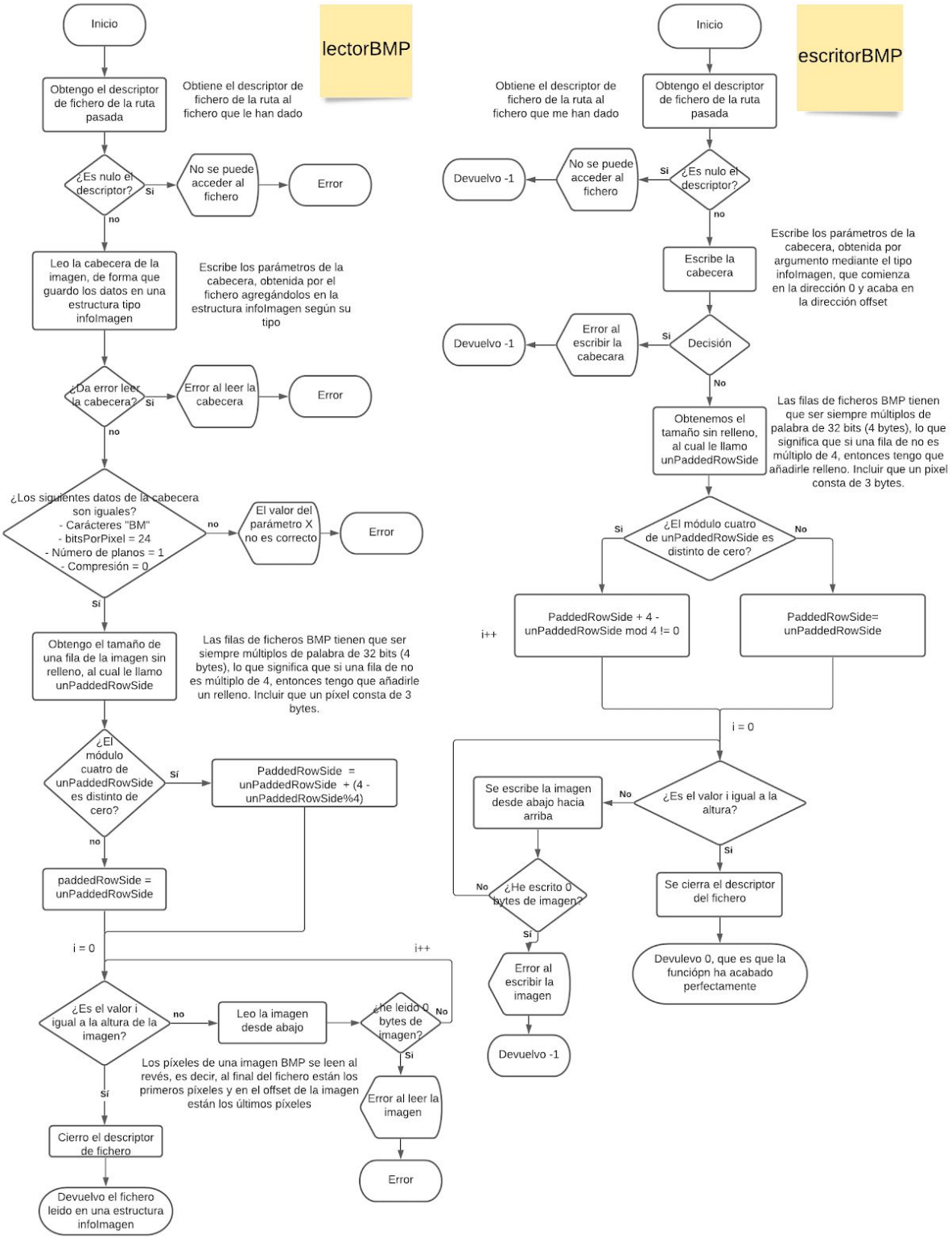


2.1.3 *LectorBMP*

El lector (o leerBMP, que es el nombre de la función) obtiene como entrada una ruta a un fichero y una variable de error, que se usará para acabar el proceso en caso de que obtenga errores. La salida es un fichero infoImagen, el cual contiene la cabecera de imagen en variables según su tamaño y los píxeles almacenados en caracteres sin signo. El diseño implementado se basa en la lectura total del fichero. Para solucionar que el programa no guardara el relleno de la imagen, obtuvimos el relleno que genera la imagen para que sea múltiplo de palabra y lo sumamos al total de la fila en bytes. Luego al leerla, ignoramos ese relleno, para solo obtener la imagen.

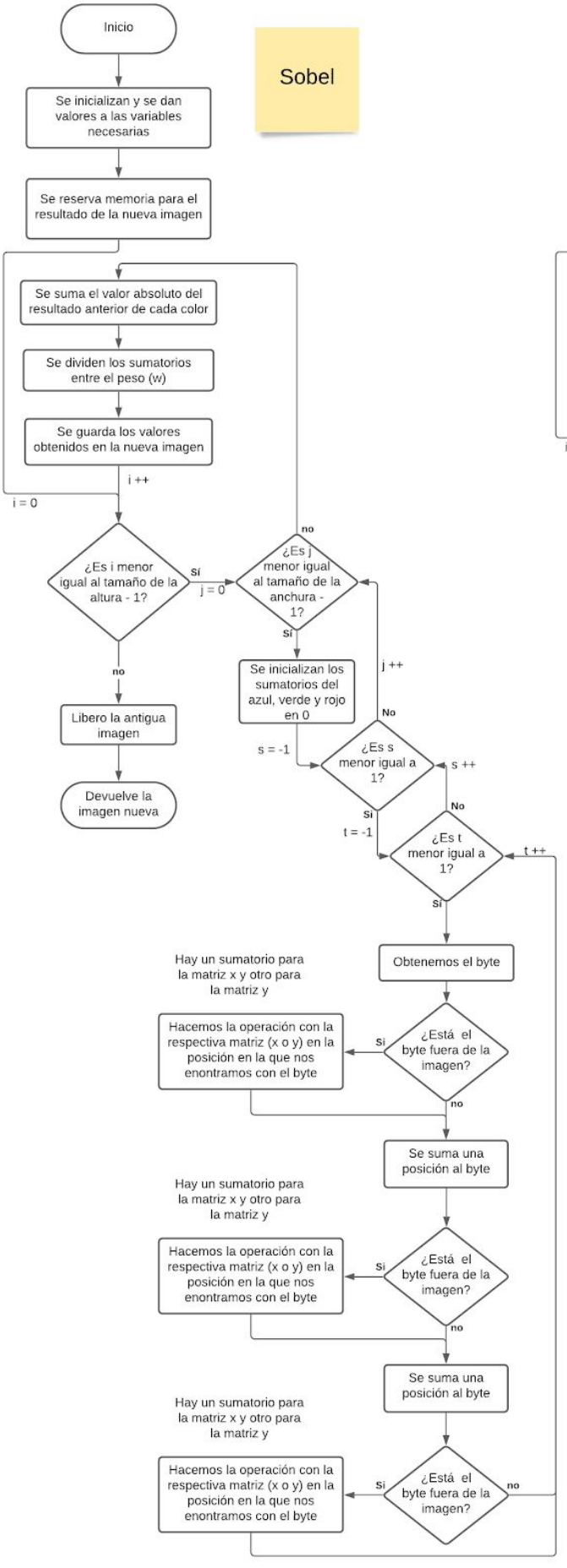
2.1.4 *EscritorBMP*

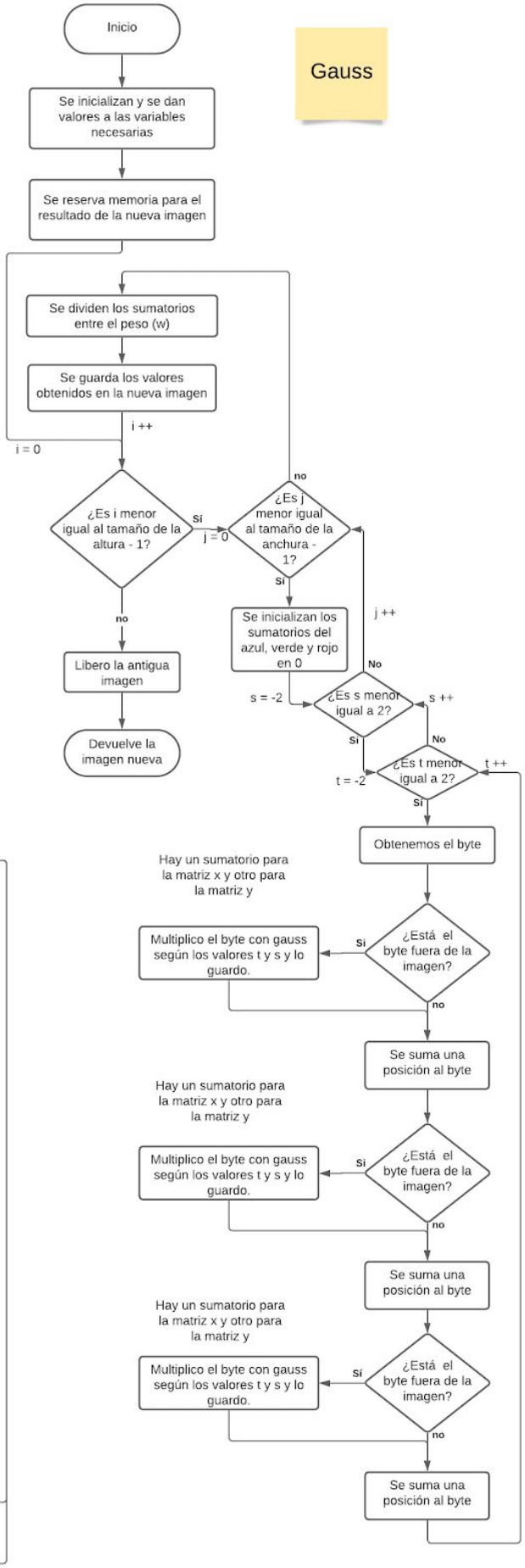
El escritor realiza lo mismo que el lector, nada más que para escribir. Hace con el relleno lo mismo para poder almacenarlo en el fichero.



2.1.5 *Gauss y Sobel*

Se obtienen a través de las fórmulas obtenidas del enunciado. En evaluación del rendimiento explicaremos sobre cómo fueron optimizados.


Sobel
Inicio
Se inicializan y se dan valores a las variables necesarias
Se reserva memoria para el resultado de la nueva imagen
Se suma el valor absoluto del resultado anterior de cada color
Se dividen los sumatorios entre el peso (w)
Se guarda los valores obtenidos en la nueva imagen
i ++
i = 0
¿Es i menor igual al tamaño de la altura - 1?
Sí
j = 0
¿Es j menor igual al tamaño de la anchura - 1?
no
Sí
Se inicializan los sumatorios del azul, verde y rojo en 0
j ++
s = -1
¿Es s menor igual a 1?
No
s ++
Si
t = -1
¿Es t menor igual a 1?
t ++
Sí
Obtenemos el byte
Hay un sumatorio para la matriz x y otro para la matriz y
Hacemos la operación con la respectiva matriz (x o y) en la posición en la que nos enontramos con el byte
¿Está el byte fuera de la imagen?
Si
no
Se suma una posición al byte
Libero la antigua imagen
Devuelve la imagen nueva


Gauss
Inicio
Se inicializan y se dan valores a las variables necesarias
Se reserva memoria para el resultado de la nueva imagen
Se dividen los sumatorios entre el peso (w)
Se guarda los valores obtenidos en la nueva imagen
i ++
i = 0
¿Es i menor igual al tamaño de la altura - 1?
Sí
j = 0
¿Es j menor igual al tamaño de la anchura - 1?
no
Sí
Se inicializan los sumatorios del azul, verde y rojo en 0
j ++
s = -2
¿Es s menor igual a 2?
No
s ++
Si
t = -2
¿Es t menor igual a 2?
t ++
Sí
Obtenemos el byte
Hay un sumatorio para la matriz x y otro para la matriz y
Multiplico el byte con gauss según los valores t y s y lo guardo.
¿Está el byte fuera de la imagen?
Si
no
Se suma una posición al byte
Libero la antigua imagen
Devuelve la imagen nueva

3. Versión paralela

La versión paralela centra la acción de su código en la parte del **proceso** justamente en los puntos anteriormente mencionados **4.3.1** y **4.3.2**.

La paralelización consiste en la creación de hilos y cada uno de estos se va a encargar de realizar la operación (ya sea de gauss o Sobel) sobre toda una fila de píxeles (anchura de la fotografía). Dependiendo del número de hilos que queramos utilizar (1,2,4,8 o 16), se leerán esa cantidad de filas en paralelo, siempre y cuando el computador tenga esa cantidad de hilos. Si es menor a la designada, los hilos que no entren en proceso paralelo se tienen que esperar a que acaben los primeros que llegan al procesador.

Suponiendo que tenemos una imagen de 1000x1000 píxeles a escala 1:100 se vería de la siguiente manera con una cantidad estipulada de 4 hilos:

Hilo N	0-99*	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 1	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 4	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 3	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 2	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 1	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 4	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 3	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 2	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
Hilo 1	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999

*Cada casilla representa un cuadro de 100x100 píxeles, funcionando con 4 hilos en paralelo

En este ejemplo, se puede ver cómo funciona la paralelización de nuestro programa. Lo primero es asignar a cada uno de los hilos una fila, después cada hilo realizará el bucle pero con los valores del for otorgados. En nuestro proyecto hemos decidido usar un modelo paralelo dinámico, de forma que a cada hilo se le asigna una valor de forma secuencial (igual que en la tabla anterior).
En el momento que un hilo termine una acción, comenzará inmediatamente con la fila consecutiva correspondiente y así en bucle hasta que se haya aplicado la función a todas las filas del archivo.
Estos hilos contienen atributos privados para que solo puedan ser accedidos mediante la memoria privada de cada núcleo para evitar que se cometa un fallo de datos. Para agregar los bytes en el fichero final no hay problema, debido a que cada byte solo se tiene que hacer por un hilo y que se lee siempre del fichero original que no se puede modificar.

4. Evaluación de rendimiento

Realizaremos tres partes de evaluación: una parte donde evaluamos la parte secuencial mediante las versiones optimizadas y no optimizadas del código. Una segunda parte donde evaluaremos la versión paralela según los hilos que utilicemos y una última donde evaluaremos la versión paralela con la versión secuencial optimizada.

4.1 Evaluación secuencial

Para la evaluación secuencial vamos a utilizar dos versiones del código: una versión sin optimizar, la cual ha sido modificada expresamente para poder comprobar como el código puede ser exageradamente lento si no se realizan las optimizaciones necesarias y la versión la cual explicamos anteriormente que está optimizada. Las optimizaciones que se realizaron fueron:

- Optimización espacial, de forma que leo los bytes seguidos y a la vez, haciendo que el vector de anchura de bytes se desplace de forma más eficiente en memoria caché, obteniendo menos fallos.
- Optimización de desenrollamiento de bucles, de forma que hemos eliminado un bucle donde obtenemos cada uno de los bytes de un pixel por una versión similar sin bucle que nos ha disminuido el tiempo considerablemente.
- Hemos optimizado la tarea de operaciones de variables, de forma que en vez de tener que realizar una suma cada vez que entres en un if, por ejemplo, esta operación será realizada solo una vez y guardada en una variable.

Los experimentos los vamos a realizar con tres fotos:

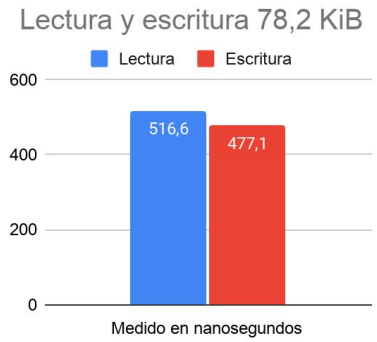
Peso	Altura	Anchura
78,2 KiB	200 píxeles	133 píxeles
961,9 KiB	469 píxeles	700 píxeles
51,3 MiB	3456 píxeles	5184 píxeles

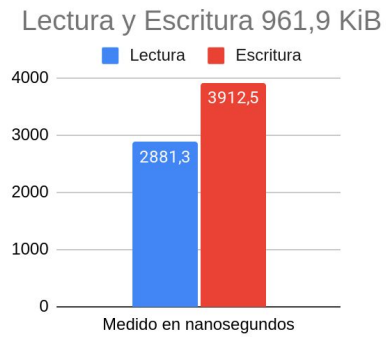
De esta forma podemos comprobar entre una imagen pequeña, una imagen típica y una imagen exageradamente grande.

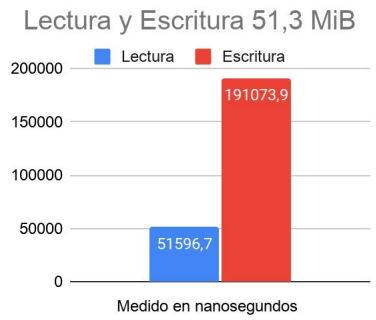
4.1.1 *Comprobar lectura y escritura*

Lo primero que vamos a evaluar son la lectura y escritura de archivos. Esto es debido a que en las tres versiones se utilizan las mismas funciones sin modificaciones, por lo tanto la lectura y escritura funcionará de la misma forma debido a que no cambia. Como se puede comprobar, el tiempo aumenta según el tamaño de las filas y de las columnas. Hay una curiosidad: como hemos explicado anteriormente, las funciones leerBMP y escribirBMP son muy similares, cambiando que una hace la función fread, y la otra realiza la función fwrite. Pues como podemos ver en las gráficas, el tiempo de lectura de un fichero crece cuantas más filas y columnas tiene (obvio), pero crece mucho más el tiempo en la escritura del mismo fichero. Si observamos la segunda gráfica, la escritura tiene un 25% menos de rendimiento que la lectura, pero este porcentaje escala casi a un 75% en el caso de la imagen más pesada. Esto puede ser debido a que tiene que leer en memoria y escribir en disco, acción que lleva mucho más tiempo que leer de disco y escribir en memoria ya que se necesita tener que acceder al

disco y encontrar bloques libres, en cambio cuando lees ya existe de antemano una zona habilitada para escribir el archivo.







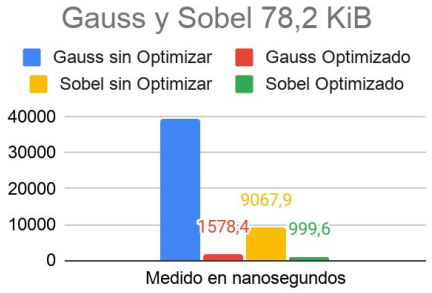
4.1.2 *Gauss y Sobel*

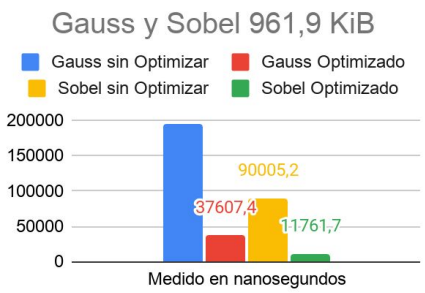
Para Gauss y Sobel tenemos que tener en cuenta que tenemos dos versiones del código: una versión la cual está mal optimizada y otra versión que está optimizada como explicamos anteriormente. En las gráficas se puede comprobar que la realización de Gauss tarda más que la realización de Sobel y eso es debido a que Gauss tiene que acceder a más bytes que Sobel, lo que provoca que seguramente de más fallos de caché y genere aumentos de tiempo. Esto es porque Gauss necesita poder acceder a veinticinco bytes, los cuales son los elementos de la máscara de Gauss y, en definitiva, los bytes a los que tiene que acceder en la imagen. En cambio, Sobel solo tiene que acceder a 9 bytes tanto en la máscara horizontal como en la máscara vertical, lo que provoca que para la segunda máscara no existan fallos debido a que ya tiene los bytes en memoria caché.

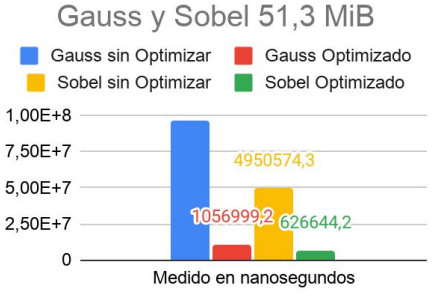
La optimización del código se puede ver en las gráficas que mejora por mucho el rendimiento. Esto debido, a:

- La optimización de espacio hizo que en caché se diera menos fallos de lectura
- Guardar en variables las operaciones que más se realizan hizo que tuvieran que pasar menos por la unidad aritmético-lógica, ahorrando ciclos y obteniendo esos valores desde la caché debido a que se utilizaban todo el rato.
- El desenrollado de bucles hizo que tuviéramos un aumento de rendimiento ya que disminuyó los retrasos debido al paralelismo a nivel de instrucción.

Por último, hablando de las gráficas y de los tiempos, se puede comprobar que el rendimiento ha aumentado más en imágenes más grandes que en las imágenes más pequeñas, lo que nos da en conclusión que optimizar es imprescindible si utilizas una gran cantidad de información.

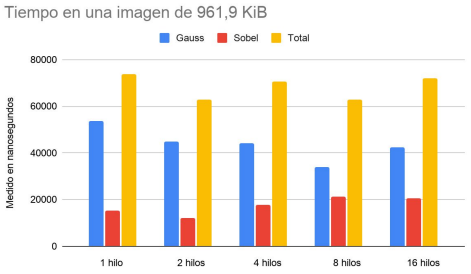


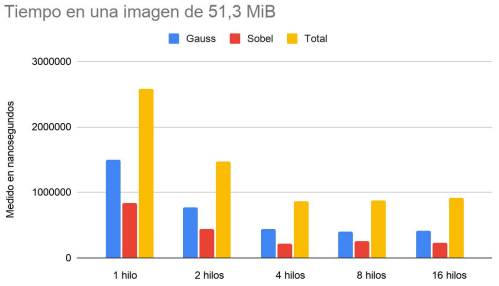


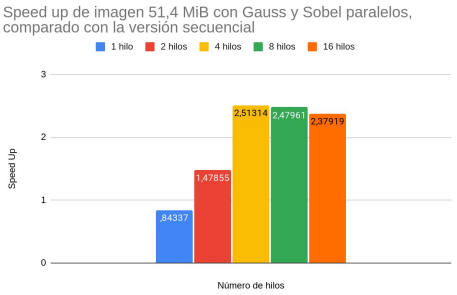


4.2 Evaluación paralela

La versión paralela se ha evaluado mediante el número de hilos. Como se puede comprobar en la primera gráfica, el número de hilos en una imagen normal de alrededor de 1 MiB de tamaño no importa porque siempre obtiene el mismo tiempo. En cambio, se puede observar en la segunda gráfica, la cual se realizó con una imagen de 51,3 MiB, existe una tendencia de mejora de rendimiento hasta los 4 hilos. A partir de ahí se estanca y no importa el número de hilos que pongas. Esto es debido a que la cpu utilizada posee 4 hilos, por lo tanto solo puede procesar 4 hilos en el momento. Si añades más al programa tienes que esperar a que los anteriores hilos acaben para procesar los siguientes. Por último, tenemos el Speed up, que se obtiene dividiendo el tiempo total obtenido de la versión secuencial entre el tiempo total de la versión paralela. Como se puede comprobar, la mejora empieza a partir de los dos hilos, llega a su clímax con 4 hilos y a partir de ahí no mejora.





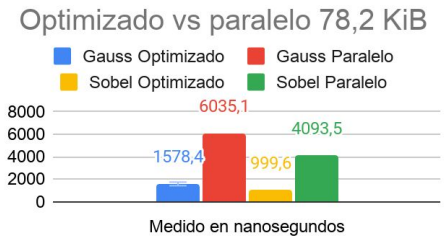


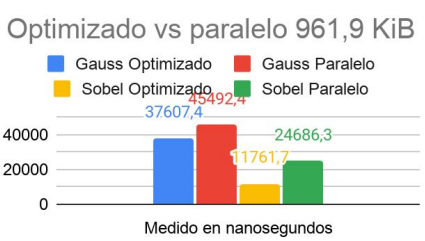
4.3 Evaluación paralela y secuencial

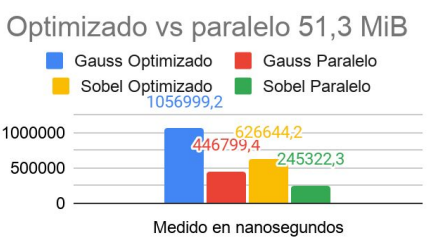
Para este apartado hemos realizado el experimento con cuatro hilos y hemos recogido los valores de Gauss y Sobel optimizados del apartado 4.1.2. No hemos querido agregar los datos de Gauss y Sobel no optimizados debido a que no son útiles para encontrar si está mejor optimizado un código secuencial o paralelo.

A partir de las gráficas, podemos deducir que según el fichero el rendimiento es mejor en código secuencial o paralelo. Esto es debido a que en ficheros pequeños, se tarda más en realizar las acciones pertinentes de paralelizar el código que en realizar los filtros. Cuanto más grande el fichero, más cambia el rendimiento por la parte paralela, debido a que realizar el reparto de filas ahora sí ayuda a obtener un muchísimo mejor rendimiento.

Por lo tanto, en esta parte podemos sacar en conclusión que en un futuro (o incluso ahora en el presente) es muy necesario paralelizar el código ya que las imágenes, los vídeos o propiamente el procesamiento de imagen de una gpu cada vez más tiene más resolución. Esto provocará que si no se incluye paralelización se generará una pérdida de rendimiento significativa. Por ello, muchas tarjetas gráficas ya están abandonando la idea de añadir una velocidad de reloj alta a cambio de tener más microprocesadores que sean capaces de procesar por sí mismos una parte en específico de la imagen, obteniendo un mayor rendimiento.







5. Pruebas realizadas

Objetivo	Procedimiento	Salida esperada	Salida obtenida
Funcionamiento correcto de Copy.	Comando de “copy” correcto con una directorio de origen y uno de destino.	Éxito, se copian correctamente las imágenes de la carpeta “origen” en “destino”.	Éxito.
Funcionamiento correcto de Gauss.	Comando de “Gauss” correcto con una directorio de origen y uno de destino.	Éxito, se aplica correctamente la función gauss en las imágenes de la carpeta “origen” y se almacenan en “destino”.	Éxito.
Funcionamiento correcto de Sobel.	Comando de “Sobel” correcto con una directorio de origen y uno de destino.	Éxito, se aplica correctamente la función sobel en las imágenes de la carpeta “origen” y se almacenan en “destino”.	Éxito.
No hay imágenes en la carpeta origen.	La carpeta origen está vacía.	Éxito, no copia nada.	Éxito.
Pasarle una dirección de destino sin / al final.	El segundo argumento no tiene una / al final del directorio de destino.	Éxito, el programa añade un / en caso que falte.	Éxito.
Pasarle una dirección de origen sin / al final.	El primer argumento no tiene una / al final del directorio de origen.	Éxito, el programa añade un / en caso que falte.	Éxito.
Pasarle una dirección de destino y de origen sin / al final.	El primero y segundo argumento no tiene una / al final del directorio de origen y de destino.	Éxito, el programa añade un / en caso que falte.	Éxito.
Pasarle en origen una ruta en el que el programa no tiene permisos.	Para ello, intentamos acceder a una carpeta del sistema, donde sea necesario ser superusuario.	No se puede abrir el directorio de origen.	La salida corresponde con la esperada: Cannot open directory.
Pasarle en destino una ruta en el que el programa no tiene permisos.	Para ello, intentamos acceder a una carpeta del sistema, donde sea necesario ser superusuario.	No se puede abrir el directorio de destino.	La salida corresponde con la esperada: Error de escritura:Output directory [directorio] does not exist.

Pasar una gran cantidad de imágenes BMP.	El directorio de origen tiene una gran cantidad de imágenes.	Éxito, se realizan las operaciones correctamente para las distintas imágenes.	Éxito.
Pasar como argumento de destino el mismo directorio que el de origen.	Los dos argumentos de origen y de destino son la misma carpeta.	Se realiza la operación en el mismo directorio.	Éxito.
Se pasan menos de cuatro argumentos.	Comando en el que sólo pasamos img-seq sin acompañarlo de más argumentos, en el caso de pasar dos y tres pasa lo mismo.	Formato incorrecto, no se puede realizar la operación porque se necesitan los cuatro parámetros.	Formato incorrecto.
Se pasan más de cuatro argumentos.	Comando en el que pasamos más de cuatro argumentos.	Formato incorrecto, no se puede realizar la operación porque espera que entren cuatro parámetros.	Formato incorrecto.
Se pasa el segundo argumento erróneamente.	Escribimos mal el segundo argumento (en vez de “sobel”, “sobell”).	Operación inesperada, no se puede realizar la operación porque no se encuentra en una de las posibles operaciones.	Operación inesperada.
Formato de la imagen incorrecto (BMP).	Introducimos una fotografía de formato jpg en vez de BMP.	El archivo no es BMP, no se puede realizar la operación debido a que la imagen no está en el formato adecuado.	La salida es la esperada.
Uno de los directorios pasados por parámetro no existe.	Un directorio pasado por parámetro no existe.	No se puede abrir el directorio[], no se puede abrir el directorio que debería ser el origen porque no existe.	No se puede abrir el directorio[].
Pasar una imagen BMP sin datos.	La imagen no contiene píxeles.	Debe dar error de lectura.	Da error de lectura.
Pasar una imagen BMP de gran tamaño (300 mb).	La imagen que se encuentra en el origen es de un gran tamaño.	Escribe y lee la imagen perfectamente, siempre que exista memoria.	Lee y escribe la imagen correctamente.

Utilizar un path absoluto como home en el origen.	Pasar como argumento de origen un path absoluto como home.	Debe funcionar correctamente.	Éxito.
Insertar una carpeta dentro de la carpeta de origen.	Dentro de la carpeta de origen hay otra carpeta "Nueva".	Error de escritura: es un directorio, debido a que el programa no está diseñado para que actúe sobre directorios.	La salida corresponde con la esperada: Error de escritura: es un directorio.
Un fichero en la carpeta destino que se llama igual que la de origen no se puede reemplazar.	Para ello le quito permisos a una imagen o archivo que se llama igual al fichero obtenido.	Error de escritura, debido a que no se puede escribir sobre un fichero sin permisos.	La salida corresponde con la esperada: Error debido a que el acceso a la fichero es denegado.
Un fichero de la carpeta de origen no se puede leer.	Para ello le quito permisos a una imagen o archivo que se llama igual al fichero obtenido.	Error de escritura, debido a que no se puede escribir sobre un fichero sin permisos.	La salida corresponde con la esperada: Error debido a que el acceso a la fichero es denegado.

6. Conclusiones

Antes de finalizar, quiero añadir el estudio de los modelos de planificación static, dynamic y guided en cuatro y ocho hilos. Static es el que peor resultado obtiene, debido a que particiona el bucle y le da una bloque a cada hilo. Esto genera más fallos de caché debido a que no se puede llevar la imagen completa a la memoria caché. En cambio, guided y dynamic obtienen resultados similares, debido a que realizan una acción similar, es decir, dynamic asigna hilos de forma secuencial, de forma que el primer hilo hace el bucle con el valor 0, el segundo con el 1... y cuando el primer hilo acaba, se le da un valor que va después del hilo n.

Desde el punto de vista de la asignatura, este proyecto nos ha servido para entender de manera más técnica el funcionamiento de los hilos y los procesadores. Aprendes a cómo desarrollar código de manera eficiente para obtener los mejores resultados en tu programa, aprovechando las posibilidades que nos brindan los multiprocesadores. Todo esto sin olvidarnos de que hemos aprendido técnicas de procesamiento de imágenes, que nos muestra una utilidad real de todo lo visto en las clases.

Pero tampoco ha sido tarea fácil. Los múltiples cambios de código y la dificultad para encontrar ayuda útil para el desarrollo del proyecto en internet, han aumentado bastante el coste en horas.

La mayoría de problemas que nos han ido surgiendo durante la programación han sido los comienzos de cada fase, es decir, hacer el copia y pega de la imagen, conseguir obtener Gauss o Sobel, la paralelización de unos de los dos, etc. Una vez consigues la realización de uno, el otro salía con mayor facilidad.

Hemos comprendido la importancia que tiene la paralelización y por qué es necesario comprenderla. Al final siempre llegamos a la misma conclusión: divide y vencerás.