Programación de Sistemas Distribuidos

Curso 2020/2021

Práctica 2

Filtrado de imágenes en un sistema distribuido utilizando MPI

1.- Introducción

Esta práctica consiste en diseñar e implementar un sistema de filtrado de imágenes en un sistema distribuido utilizando MPI. El objetivo de la práctica es poder distribuir el procesado de una imagen entre diferentes recursos de cómputo para incrementar su rendimiento.

En concreto, se va a realizar un filtrado sobre imágenes bitmap (BMP) en escala de grises a blanco y negro. Cada pixel de una imagen en escala de grises se representa mediante un valor entre 0 y 255. Sin embargo, en las imágenes en blanco y negro contamos únicamente con dos colores: blanco, representado con el valor 255 y negro, representado con el valor 0.

Para el desarrollo de esta práctica se proporciona un algoritmo secuencial de filtrado. El objetivo de la misma es desarrollar un programa en MPI para procesar de forma paralela el filtrado de imágenes. Aunque no es obligatorio, el alumno puede tomar el código fuente facilitado como base para realizar la práctica. El código fuente se encuentra en el fichero `PSD_Prac2_SolucionSecuencial.zip`. La siguiente porción de código (ver fichero `bmpBlackWhite.h`) contiene las cabeceras utilizadas por los ficheros BMP.

```
// BMP file header
typedef struct{
    unsigned short bfType;
    unsigned int bfSize;
    unsigned short bfReserved1;
    unsigned short bfReserved2;
    unsigned int bfOffBits;
} tBitmapFileHeader;


// BMP info header
typedef struct{
    unsigned int  biSize;
    unsigned int  biWidth;
    unsigned int  biHeight;
    unsigned short biPlanes;
    unsigned short biBitCount;
    unsigned int  biCompression;
    unsigned int  biSizeImage;
    unsigned int  biXPelsPerMeter;
    unsigned int  biYPelsPerMeter;
    unsigned int  biClrUsed;
    unsigned int biClrImportant;
} tBitmapInfoHeader;
```

Cada fichero bitmap cuenta con dos cabeceras. La primera cabecera, representada con la estructura `tBitmapFileHeader` cuenta con 5 campos:

- `bfType`: Tipo. Generalmente contiene los caracteres "BM" en ASCII.
- `bfSize`: Tamaño del fichero (en bytes).
- `bfReserved1`: Campo reservado. Generalmente tiene el valor 0.
- `bfReserved2`: Campo reservado. Generalmente tiene el valor 0.
- `bfOffBits`: Offset al primer byte de la imagen en el fichero.

La segunda cabecera, representada con la estructura `tBitmapInfoHeader`, cuenta con los siguientes campos:

- `biSize`: Tamaño de la cabecera.
- `biWidth`: Ancho de la imagen (en pixels).
- `biHeight`: Alto de la imagen (en pixels).
- `biPlanes`: Número de planos.
- `biBitCount`: Número de bits utilizados por pixel.
- `biCompression`: Tipo de compresión ó 0 cuando no hay compresión.
- `biSizeImage`: Tamaño de los datos de la imagen (en bytes). En algunos casos, puede no coincidir con la multiplicación de ancho por alto.
- `biXPelsPerMeter`: Resolución.
- `biYPelsPerMeter`: Resolución.
- `biClrUsed`: Número de colores.
- `biClrImportant`: Número de colores importantes.

Es importante remarcar que la cantidad de datos de la imagen no siempre va a coincidir con la multiplicación del ancho por el alto de la imagen. Esto sucede ya que el formato BMP utiliza *padding* (relleno) en las filas. De esta forma, los bytes necesarios para almacenar una fila de la imagen deben ser divisibles por 4. En caso contrario, se introducen bytes de relleno. La forma de calcular el tamaño (en bytes) de una fila se describe a continuación:

```
tBitmapInfoHeader headerInfo;
unsigned int rowSize;
…
rowSize = (((headerInfo.biBitCount * headerInfo.biWidth) + 31) / 32 ) * 4;
```

Los datos de la imagen se almacenan de forma secuencial en el fichero. Puesto que vamos a utilizar imágenes en escala de grises, cada pixel de la imagen se representará con un `unsigned char`. Así, cada pixel podrá tomar un valor entre 0 y 255. La siguiente figura muestra un ejemplo gráfico de cómo se representan los *pixels* de la imagen en fichero.

56	32	104	112	32	32	56	129	…	**210**
45	4	205	67	89	206	255	255	…	4
…	…	…	…	…	…	…	…	…	…
45	2	234	230	122	122	122	126	…	89

Aunque los *pixels* se almacenen de forma secuencial en el fichero, se debe hacer una distinción lógica entre las distintas filas que componen la imagen. De esta forma, no existen dos *pixels* que puedan ser vecinos si cada uno pertenece a una fila distinta de la imagen, aunque éstos estén almacenados de forma contigua en el fichero. En el ejemplo anterior, los *pixels* en negrita (210 y 45) se almacenan de forma contigua en fichero. Sin embargo, cada uno de estos *pixels* no interviene en el filtrado del otro. Esta división de la imagen en filas puede resultar muy útil al distribuir los datos entre procesos.

El filtrado de la imagen va a consistir en transformar cada pixel en escala de grises a un *pixel* en color blanco o negro. Para calcular el valor final del pixel hacemos uso de 2 parámetros. El primer parámetro es el *threshold* (umbral). El segundo parámetro es el vector que contiene tanto el *pixel* que se está procesando como sus vecinos. Básicamente, los *pixels* vecinos de un pixel *p* se corresponden con los *pixels* que *p* tiene a

su izquierda y a su derecha. Así, denotamos p_v como el vector que contiene a p y a sus *pixels* vecinos, es decir, los *pixels* involucrados en el filtrado de p.

El siguiente ejemplo muestra los *pixels* involucrados para el filtrado de un pixel p (borde negro). En este caso, el valor de p es 89. Sus vecinos, marcados en rojo, tienen valores de 67 y 206. Por lo tanto, p_v=[67,89,206]

45	4	205	**67**	**89**	**206**	255	255	…	4

El siguiente ejemplo muestra el vector de los *pixels* involucrados en el filtrado de p (borde negro). En este caso, siendo p el primer pixel de la fila, p_v=[45,4].

45	**4**	205	67	89	206	255	255	…	4

Para facilitar el proceso de filtrado, se facilitan varias funciones incluidas en el fichero `bmpBlackWhite.h`

```
void readHeaders (char* fileName,
                  tBitmapFileHeader *bmHeader,
                  tBitmapInfoHeader *bmInfoHeader);

void writeHeaders (char* fileName,
                   tBitmapFileHeader *bmHeader,
                   tBitmapInfoHeader *bmInfoHeader);

void printBitmapHeaders (tBitmapFileHeader *bmHeader,
                         tBitmapInfoHeader *bmInfoHeader);

unsigned char calculatePixelValue (tPixelVector vector,
                                   unsigned int numPixels,
                                   unsigned int threshold,
                                   int debug);
```

La función `readHeaders` lee las dos cabeceras del fichero `fileName`. De forma similar, la función `writeHeaders` escribe en el fichero `fileName` las dos cabeceras pasadas como parámetro. En ambas funciones, los ficheros se cierran al finalizar la operación correspondiente.

La función `printBitmapHeaders` imprime por pantalla el contenido de las dos cabeceras pasadas como parámetro.

Finalmente, la función `calculatePixelValue` se encarga de hacer la transformación de un *pixel* p. Esta función recibe 4 parámetros:

- `vector`: Vector de *pixels* involucrados en el filtrado de p, es decir, p_v.
- `numPixels`: Número de *pixels* involucrados en el filtrado de p.
- `threshold`: Umbral para calcular el valor del pixel filtrado.
- `debug`: Indica si se imprime por pantalla mensajes sobre el filtrado (1) o no (0).

El filtrado completo de la imagen consistirá en recorrer la matriz de *pixels* e invocar a la función `calculatePixelValue` con los parámetros correspondientes. Seguidamente se presentan algunos ejemplos[1] aplicando este filtro.

[1] Imágenes obtenidas de http://ilab.engr.utk.edu/ilabdocs/Epilog/BMP%20sample%20files/

Fig. 1 Original

Fig. 2 Threshold=60

Fig. 3 Threshold=120

En estos ejemplos, Fig.1 muestra la imagen original, mientras que Fig.2 y Fig. 3 muestran el filtrado utilizando los valores de *threshold* 60 y 120, respectivamente.

2. Filtrado distribuido utilizando MPI

En esta práctica se va a procesar, de forma distribuida y paralela, el filtrado de una imagen utilizando MPI. Para ello, se van a distinguir dos tipos de procesos. El proceso *master* se encargará de realizar todas las tareas de entrada y salida sobre los ficheros. Además, realizará la asignación de los datos a procesar. Por otro lado, los procesos *worker* realizarán el filtrado de la imagen. Básicamente, estos procesos deben recibir una porción de la imagen, realizar el filtrado de la misma y devolverla al proceso *master*.

No se permite enviar la imagen completa a todos los procesos *worker*. Dependiendo de la estrategia utilizada, se enviarán los datos correspondientes a los procesos *worker* mediante un determinado algoritmo, pero nunca la imagen completa.

El número mínimo de procesos, incluyendo el proceso *master*, para iniciar el proceso de filtrado, será de 3. En caso contrario, se mostrará un mensaje de error y se finalizará la ejecución del programa.

2.1 Modo estático

En este apartado se va a procesar, de forma distribuida y paralela, el filtrado de una imagen utilizando una estrategia de distribución de datos estática. Es decir, a cada proceso *worker* se le asignará una única porción de la imagen para que realice el filtrado, calculado al inicio del programa y antes de que el proceso *master* comience a distribuir los datos entre los procesos *worker*. Así, a cada proceso *worker* le corresponde la misma carga de trabajo, exceptuando el último proceso únicamente en aquellos casos donde la cantidad total de datos a procesar no sea divisible por el número de procesos *worker*.

La carga de trabajo puede asignarse, por ejemplo, dividiendo la imagen por filas y asignando a cada *worker* un número determinado de filas contiguas.

El diseño e implementación del algoritmo encargado de distribuir los datos de la imagen original, así como de recoger las porciones de la imagen filtrada, será tarea del alumno. Se tendrán en cuenta todas aquellas consideraciones que tengan un impacto positivo en el rendimiento del programa.

El código fuente de este apartado deberá estar contenido en el fichero `bmpFilterStatic.c`. La ejecución del programa se realizará de la siguiente forma:

```
mpiexec -hostfile machines -np numProc ./bmpFilterStatic im_o im_f thres
```

donde:

- `machines` es el fichero que contiene la dirección de las máquinas.
- `numproc` es el número de procesos que intervienen en el filtrado.
- `bmpFilterStatic` es el código fuente del programa pedido.
- `im_o` es el nombre de la imagen original.
- `im_f` es el nombre de la imagen filtrada.
- `thres` es el umbral.

2.2 Modo dinámico

En este apartado se va a procesar, de forma distribuida y paralela, el filtrado de una imagen utilizando una estrategia de distribución de datos dinámica. Para poder asignar carga de trabajo de forma dinámica, es necesario establecer el tamaño del grano, o lo que es lo mismo, el tamaño de datos máximo que cada proceso *worker* recibirá para realizar el filtrado. Por ejemplo, el grano podrá hacer referencia al número de filas que un proceso deberá filtrar, cada vez que el proceso maestro le asigne una porción de la imagen. Consecuentemente, cada proceso *worker* puede procesar más de una porción de la imagen.

De esta forma, cada vez que un proceso *worker* haya finalizado el procesado de una determinada porción de la imagen, el proceso *master* le podrá asignar una nueva porción que aún no haya sido filtrada. Así, la carga se distribuye de forma dinámica, asignando más cantidad de trabajo a los procesos ejecutados en las máquinas con mayor capacidad de cómputo.

El diseño e implementación del algoritmo encargado de distribuir los datos de la imagen original, así como de recoger las porciones de la imagen filtrada, será tarea del alumno. Se tendrán en cuenta todas aquellas consideraciones que tengan un impacto positivo en el rendimiento del programa.

El código fuente de este apartado deberá estar contenido en el fichero `bmpFilterDynamic.c`. La ejecución del programa se realizará de la siguiente forma:

```
mpiexec -hostfile machines -np numProc ./bmpFilterDynamic im_o im_f thres nrows
```

donde:

- `machines` es el fichero que contiene la dirección de las máquinas.
- `numproc` es el número de procesos que intervienen en el filtrado.
- `bmpFilterDynamic` es el código fuente del programa pedido.
- `im_o` es el nombre de la imagen original..es
- `im_f` es el nombre de la imagen filtrada.
- `thres` es el umbral.
- `nrows` es el número de filas máximo que el proceso master puede asignar, cada vez, a un proceso *worker*.

3.- Entrega de la práctica

Para entregar esta práctica se habilitará un entregador en la página de la asignatura del Campus Virtual. La fecha límite para entregarla será el **día 17 de diciembre de 2020 a las 18:00**.

No se permitirá la entrega de prácticas fuera del plazo establecido.

La entrega se realizará mediante un **único fichero con extensión .zip**. La estructura del fichero será la siguiente:

```
nombres.txt
bmpFilterStatic.c
bmpFilterDynamic.c
Makefile
bmpBlackWhite.h
bmpBlackWhite.o
```

donde el fichero `nombres.txt` contendrá el nombre y apellidos de los integrantes del grupo.

Se van a perseguir las copias y plagios de prácticas, aplicando con rigor la normativa vigente.