Alberto Vázquez Martínez

Documentación práctica 2 – Diseño de Infraestructura de red

Enunciado del problema

Utilizaremos las primitivas pertinentes MPI2 como acceso paralelo a disco y gestión de procesos dinámico:

Inicialmente el usuario lanzará un solo proceso mediante mpirun -np 1 ./pract2. Con ello MPI lanza un primer proceso que será el que tiene acceso a la pantalla de gráficos, pero no a disco. Él mismo será el encargado de levantar N procesos (con N definido en tiempo de compilación como una constante) que tendrán acceso a disco, pero no a gráficos directamente. Los nuevos procesos lanzados se encargarán de leer de forma paralela los datos del archivo foto.dat. Después, se encargarán de ir enviando los pixeles al primer elemento de proceso para que éste se encargue de representarlo en pantalla.

Usaremos la plantilla pract2.c para comenzar a desarrollar la práctica. En ella debemos completar el código que ejecuta el proceso con acceso a la ventana de gráficos (rank 0 inicial) y la de los procesos "trabajadores".

Se proporciona el archivo foto.dat. La estructura interna de este archivo es 400 filas por 400 columnas de puntos. Cada punto está formado por una tripleta de tres "unsigned char" correspondiendo al valor R,G y B de cada uno de los colores primarios. Estos valores se pueden usar para la función dibujaPunto.

Planteamiento y diseño de la solución

Lo primero que debemos tener en cuenta es que el primer y único proceso lanzado es el que se encargará de iniciar el resto de los procesos, que serán lo que "leerán" el archivo *foto.dat*. Para iniciar los procesos, se utilizará la función *MPI_Comm_Spawn*, indicando el archivo ejecutable correspondiente a los procesos, el número de procesos que se ejecutarán y otros parámetros propios de la función. El número de procesos se define, al comienzo del código, con una constante llamada *NUM_WORKERS_PROCESS*. Estos procesos pasarán a ser procesos hijos del proceso 0, y tendremos dos comunicadores que serán necesarios para el envío y la recepción de los mensajes.

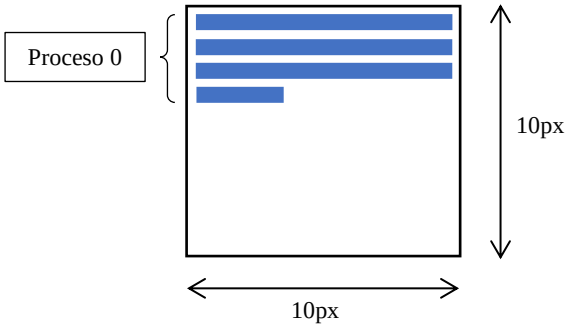
A continuación, se solicitará al usuario un número entero que indicará el tipo de filtro que se le aplicará a la imagen. Una vez sabemos el tipo de filtro, se le enviará a cada uno de los *procesos trabajadores* dicho valor para que así todos los procesos apliquen el mismo filtro.

Por parte del proceso 0, solo quedaría esperar a que los procesos trabajadores le envían cada uno de los píxeles y este los dibuje utilizando la función *dibujaPunto*. Tanto la recepción del mensaje como la función de dibujo se encuentran dentro de un bucle *for* que va de 0 hasta 160000, ya que la imagen que vamos a imprimir es de 400x400 pixeles.

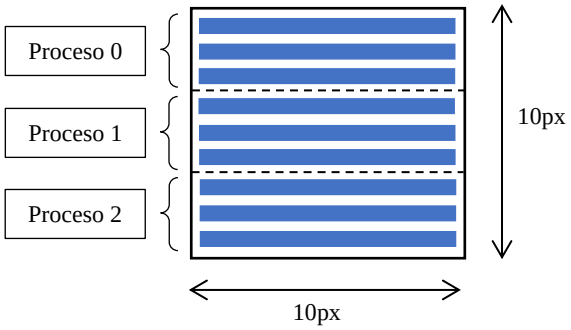
Por otro lado, los *procesos trabajadores* lo primero que harán será leer el archivo *foto.dat* con la función *MPI_File_open* y además obtendremos el tamaño del archivo con la función *MPI_File_get_size*, y esto lo hacemos para conocer el tamaño en bytes total del archivo. En este caso el tamaño del archivo es de 480000 bytes, ya que, como se ha mencionado anteriormente, esta imagen es de 400x400 pixeles, pero además se almacena cada píxel como una tripleta con los códigos de color *RGB* (Rojo, verde y azul). Por tanto, nos quedaría: $400 \times 400 \times 3 = 480000$

Además, dividimos el tamaño del archivo entre el número de *procesos trabajadores* para establecer el tamaño de archivo que le pertenece a cada proceso. Justo después de establecer el tamaño del bloque que leerá cada proceso, se comprobará que los bloques sean exactamente N filas de 400 pixeles cada una. Esto se hace para que cada proceso complete un bloque exacto de pixeles, y no deje sin completar un bloque, para así, no tener problemas con la posición de los píxeles del siguiente proceso.

Ejemplo de comprobación de píxeles: Si tenemos por ejemplo un archivo de 10x10 píxeles, y utilizamos 3 procesos, entonces, a cada proceso le correspondían 33.33 pixeles, o lo que es lo mismo, 3 filas de 10 pixeles más 3.33, por lo que quedaría algo así:



Esta situación causaría problemas a la hora de dibujar los siguientes pixeles por parte del siguiente proceso, ya que la posición de los píxeles en el eje de la *X*, no sería 0. Por tanto, lo que se hace para solucionar este problema es que cada proceso complete un bloque de pixeles perfecto, en otras palabras, que el tamaño del bloque del archivo asignado debe ser un número tal que al dividirlo entre 1200 sea un número entero. La división es entre 1200 debido a que tenemos 400 píxeles de ancho y cada píxel está representado por 3 valores (*RGB*). Por lo que quedaría algo tal que así:



Utilizaremos la función que nos proporciona *OpenMPI* de *MPI_File_set_view* para asignar la vista correspondiente a cada *proceso trabajador*. Una vez establecida la vista a cada proceso, se procederá con su lectura utilizando la función *MPI_File_read*.

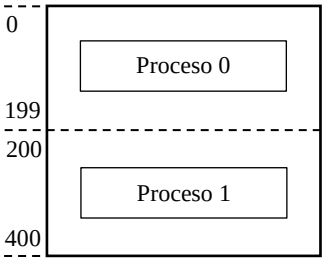
Por último, cada uno de los procesos seleccionarán el tipo de filtro que el usuario a introducido por teclado anteriormente y realizará el envío al proceso 0 del comunicador padre. Tanto la selección del filtro como el envío se encuentran dentro de un doble bucle *for*. El recorrido de la imagen se realiza de la misma manera que la de una matriz o vector de dos dimensiones. El primer bucle *for* se tiene que calcular, en píxeles, el inicio y el final del bloque correspondiente al vector (inicio < final), para ello se utiliza la siguiente fórmula:

$$inicio = \frac{tamaño\ buffer \times rank}{3 \times 400} \qquad final = \frac{tamaño\ buffer \times rank}{3 \times 400} + \frac{tamaño\ buffer}{3 \times 400}$$

El tamaño del buffer es el tamaño del bloque del archivo asignado a cada proceso. Por ejemplo, tenemos 2 procesos y una imagen de 400x400 pixeles, sus inicios y finales de las filas, en pixeles, serían:

Tamaño del buffer = $\frac{400 \times 400 \times 3}{2} = 240000$

- Proceso 0: inicio = $\frac{240000 \times 0}{3 \times 400} = 0$ final = $\frac{240000 \times 0}{3 \times 400} + \frac{240000}{3 \times 400} = 200$
- Proceso 1: inicio = = $\frac{240000 \times 1}{3 \times 400} = 200$ final = $\frac{240000 \times 1}{3 \times 400} + \frac{240000}{3 \times 400} = 200 + 200 = 400$



El segundo bucle *for* sería un bucle de 0 a 400 recorriendo cada una de las columnas de la imagen. Ejemplo de ejecución del programa para 4 procesos:

También hay que tener en cuenta el caso de que el número de procesos trabajadores sea impar, ya que la división de pixeles de la imagen no sería exacta y por tanto faltarían al menos 1 fila de 400 pixeles sin completar. *Para* solucionar esto, el último proceso se encargará de completar los píxeles restantes. Para calcular cuantos píxeles faltan por completar tendríamos que restar el tamaño total del archivo, menos la multiplicación del tamaño de imagen asignado a cada proceso por el número de procesos trabajadores.

Aplicación de los filtros

Para aplicar los filtros a la imagen se realizan ciertas operaciones distintas para cada filtro, que se aplican a cada píxel decodificado de la imagen.

- Filtro blanco y negro:

Se establece el mismo valor para cada uno de los valores *RGB* del píxel. Viene dado por la siguiente formula:

$$R, G, B = R \times 0.333 + G \times 0.333 + B \times 0.333$$

- Filtro sepia:

Para el filtro sepia se utilizan una serie de valores para cada uno de los valores *RGB*. Además, para calcular cada valor se necesitan todos los valores *RGB*. Por último, hay que comprobar que el valor resultante no sea superior a 255, en caso de que lo sea, se establecerá el valor 255.

$$R = R \times 0.393 + G \times 0.769 + B \times 0.189$$

$$G = R \times 0.349 + G \times 0.686 + B \times 0.168$$

$$B = R \times 0.272 + G \times 0.534 + B \times 0.131$$

- Filtro color invertido:

Para este filtro lo único que hay que hacer es restar a cada valor *RGB* el valor de 255.

$$R = 255 - R$$

$$G = 255 - G$$

$$B = 255 - B$$

Fuentes del programa

Pract2.c

```
#include <openmpi/mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <assert.h>
#include <unistd.h>

#include "filters_values.h"

#define NIL (0)
#define NUM_WORKERS_PROCESS 4
#define FILENAME "data/foto.dat"
#define HEIGHT 400
#define WIDTH 400

/* Global variables */
XColor colorX;
Colormap mapacolor;
char cadenaColor[]="#000000";
Display *dpy;
Window w;
GC gc;

/* Auxiliar functions */
void initX() {
    dpy = XOpenDisplay(NIL);
    assert(dpy);

    int blackColor = BlackPixel(dpy, DefaultScreen(dpy));
    int whiteColor = WhitePixel(dpy, DefaultScreen(dpy));

    w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0,
                            400, 400, 0, blackColor, blackColor);
    XSelectInput(dpy, w, StructureNotifyMask);
    XMapWindow(dpy, w);
    gc = XCreateGC(dpy, w, 0, NIL);
    XSetForeground(dpy, gc, whiteColor);
    for(;;) {
        XEvent e;
        XNextEvent(dpy, &e);
        if (e.type == MapNotify)
            break;
    }
    mapacolor = DefaultColormap(dpy, 0);
}

/* Draw a pixel into de window */
void dibujaPunto(int x,int y, int r, int g, int b) {
```

```
    sprintf(cadenaColor,"#%.2X%.2X%.2X",r,g,b);
    XParseColor(dpy, mapacolor, cadenaColor, &colorX);
    XAllocColor(dpy, mapacolor, &colorX);
    XSetForeground(dpy, gc, colorX.pixel);
    XDrawPoint(dpy, w, gc,x,y);
    XFlush(dpy);
}

/* Select filter */
int get_num_filter() {
    int num_filter = 0;

    printf("\nFiltros:\n");
    printf("- (1) Sin filtro\n");
    printf("- (2) Blanco y negro\n");
    printf("- (3) Sepia\n");
    printf("- (4) Color invertido\n");
    printf("Introduzca un número de filtro: \n");
    scanf("%d", &num_filter);

    return num_filter;
}

/* Normal image, without filter */
void set_no_filter(int *buffer, unsigned char *buf, int cnt) {
    buffer[2] = buf[cnt];
    buffer[3] = buf[cnt+1];
    buffer[4] = buf[cnt+2];
}

/* Set grayscale filter for each pixel */
void set_bw_filter(int *buffer, unsigned char *buf, int cnt) {
    buffer[2] = buf[cnt]*BLACK_WHITE_VALUE + buf[cnt+1]*BLACK_WHITE_VALUE +
buf[cnt+2]*BLACK_WHITE_VALUE;
    buffer[3] = buf[cnt]*BLACK_WHITE_VALUE + buf[cnt+1]*BLACK_WHITE_VALUE +
buf[cnt+2]*BLACK_WHITE_VALUE;
    buffer[4] = buf[cnt]*BLACK_WHITE_VALUE + buf[cnt+1]*BLACK_WHITE_VALUE +
buf[cnt+2]*BLACK_WHITE_VALUE;
}

/* Set sepia filter for each pixel */
void set_sepia_filter(int *buffer, unsigned char *buf, int cnt) {

    buffer[2] = buf[cnt]*RED_SEPIA_R + buf[cnt+1]*RED_SEPIA_G +
buf[cnt+2]*RED_SEPIE_B; /* Red */
    if (buffer[2] > 255) {
        buffer[2] = 255;
    }

    buffer[3] = buf[cnt]*GREEN_SEPIA_R + buf[cnt+1]*GREEN_SEPIA_G +
buf[cnt+2]*GREEN_SEPIA_B; /* Green */
    if (buffer[3] > 255) {
```

```
        buffer[3] = 255;
    }

    buffer[4] = buf[cnt]*BLUE_SEPIA_R + buf[cnt+1]*BLUE_SEPIA_G +
buf[cnt+2]*BLUE_SEPIA_B; /* Blue */
    if (buffer[4] > 255) {
        buffer[4] = 255;
    }
}

/* Set inverted colors filter */
void set_inverted_filter(int *buffer, unsigned char *buf, int cnt) {
    buffer[2] = 255-buf[cnt];
    buffer[3] = 255-buf[cnt+1];
    buffer[4] = 255-buf[cnt+2];
}

/* Switch that calls the filter selected */
void select_filter(int *buffer, unsigned char *buf, int cnt, int num_filter) {
    switch (num_filter) {
    case 1:
        set_no_filter(buffer, buf, cnt);
        break;
    case 2:
        set_bw_filter(buffer, buf, cnt);
        break;
    case 3:
        set_sepia_filter(buffer, buf, cnt);
        break;
    case 4:
        set_inverted_filter(buffer, buf, cnt);
        break;
    default:
        set_no_filter(buffer, buf, cnt); /* No filter by default */
        break;
    }
}

int check_pixels_division(int bufsize) {
    int truncanted;
    double result;

    result = (double) bufsize/(3*HEIGHT);
    truncanted = (int) result;
    //printf("%d %f\n",truncanted, result);
    while (result != truncanted) {

        bufsize--;
        result = (double) bufsize/(3*HEIGHT);
        truncanted = (int) result;
        //printf("%d\n",bufsize);
    }
```

```
    return bufsize;
}

/* Main function */
int main (int argc, char *argv[]) {

    int rank, size, tag, num_filter;
    MPI_Comm commPadre, intercomm;
    MPI_Status status;
    int buffer[5];
    int errcodes[NUM_WORKERS_PROCESS];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_get_parent( &commPadre );

    if ((commPadre==MPI_COMM_NULL)
        && (rank==0) )  {

        MPI_Comm_spawn("exec/pract2", MPI_ARGV_NULL, NUM_WORKERS_PROCESS,
MPI_INFO_NULL, 0, MPI_COMM_WORLD, &intercomm, errcodes);

        /* Set and send the number of filter */
        num_filter = get_num_filter();
        for (int i = 0; i < NUM_WORKERS_PROCESS; i++) {
            MPI_Send(&num_filter, 1, MPI_INT, i, i, intercomm);
        }

        initX();

        for (int i = 0; i < HEIGHT*WIDTH; i++) {
            MPI_Recv(&buffer, 5, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, intercomm,
&status);
            dibujaPunto(buffer[0], buffer[1], buffer[2], buffer[3], buffer[4]);
        }

        printf("Presiona una tecla para continuar...\n");
        getchar();getchar();
    }

    else {
        int bufsize, nrchar, cnt = 0;
        unsigned char *buf;  /* Buffer for reading */
        MPI_Offset filesize;
        MPI_File myfile;    /* Shared file */
        MPI_Status status;  /* Status returned from read */
        MPI_Request request;

        MPI_Recv(&num_filter, 1, MPI_INT, 0, MPI_ANY_TAG, commPadre, &status);
        MPI_Comm_get_parent( &commPadre );
```

```
        MPI_File_open (MPI_COMM_WORLD, FILENAME, MPI_MODE_RDONLY,
MPI_INFO_NULL, &myfile); /* Open the file */
        MPI_File_get_size(myfile, &filesize);  /* Get the size of the file */

        filesize = filesize/sizeof(unsigned char); /* Calculate how many elements that is */
        bufsize = filesize/NUM_WORKERS_PROCESS; /* Calculate how many elements each
processor gets */

        bufsize = check_pixels_division(bufsize);

        int diff = 0;
        if (rank == NUM_WORKERS_PROCESS-1) {
            if (bufsize*NUM_WORKERS_PROCESS != filesize) {
                diff = filesize - (bufsize*NUM_WORKERS_PROCESS);
                 }
            //printf("%d - %d\n",diff, diff+bufsize);
        }

        buf = (unsigned char *) malloc((bufsize+diff)*sizeof(unsigned char)); /* Allocate the buffer
to read to, one extra for terminating null char */
        //printf("%d (%d)\n",bufsize,filesize);
        MPI_File_set_view(myfile, rank*bufsize*sizeof(unsigned char), MPI_UNSIGNED_CHAR,
MPI_UNSIGNED_CHAR,
                "native", MPI_INFO_NULL); /* Set the file view */
        MPI_File_read(myfile, buf, bufsize+diff, MPI_UNSIGNED_CHAR, &status); /* Read from
the file */
        MPI_Get_count(&status, MPI_UNSIGNED_CHAR, &nrchar); /* Find out how many
elemyidnts were read */

        buf[nrchar] = (unsigned char)0; /* Set terminating null char in the string */
        MPI_File_close(&myfile); /* Close the file */
        //printf("[%d] %d - %d (%lld)\n",rank,nrchar,bufsize+diff,filesize);

           int begin = (bufsize*rank)/(3*HEIGHT);
           int end =  (((bufsize+diff)*rank)/(3*HEIGHT))+((bufsize)/(3*HEIGHT));

           for (int y = begin; y < end; y++) {
             for (int x = 0; x < WIDTH; x++) {
                buffer[0] = x;
                buffer[1] = y;
                select_filter(buffer, buf, cnt, num_filter);
                MPI_Send(&buffer, 5, MPI_INT, 0, x*y, commPadre);
                cnt+=3;
             }
        }
    }

    MPI_Finalize();
}
```

Definitions.h

```
/* SEPIA FILTERS VALUES */
#define RED_SEPIA_R 0.393
#define RED_SEPIA_G 0.769
#define RED_SEPIE_B 0.189
#define GREEN_SEPIA_R 0.349
#define GREEN_SEPIA_G 0.686
#define GREEN_SEPIA_B 0.168
#define BLUE_SEPIA_R 0.272
#define BLUE_SEPIA_G 0.534
#define BLUE_SEPIA_B 0.131

/* BLACK AND WHITE FILTERS VALUES */
#define BLACK_WHITE_VALUE 0.333
```

Instrucciones para compilar y ejecutar

La estructura de archivos y directorios es la siguiente:

- Src: código fuente en C.
- Include: archivos cabecera con constantes de filtros aplicados a la imagen.
- Exec: archivos ejecutables generados en la compilación.
- Data: contiene el archivo foto.dat.
- Makefile: utilizado para compilar y ejecutar el código.

Para compilar el programa basta con ejecutar el comando **make**. Se compilarán todos los archivos C del directorio *src*.

Para ejecutar el programa: **make run_pract2**

Para limpiar el directorio *exec* de ejecutables utilizar el comando **make clean**.