Escuela superior de informática

Diseño de infraestructuras de Red

Practica 2

Cristian Trapero Mora

13 de mayo de 2016

Índice

1. Sistema distribuido de renderizado de gráficos

1.1. 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 pixels 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 ünsigned charçorrespondiendo al valor R, G y B de cada uno de los colores primarios. Estos valores se pueden usar para la función dibujaPunto.

Para compilar el programa para openMPI en linux el comando es:

```
$ mpicc pract2.c -o pract2 -lX11
```

1.2. Planteamiento de la solución

Para el correcto desarrollo de la solución, se nos ha proporcionado un programa en C que contiene los métodos necesarios para la **creación de la ventana gráfica** usando el **servidor X11**, que contendrá la imagen renderizada, el método necesario para **dibujar** el gráfico y el fichero de prueba **datos.dat**.

Figura 1: Fichero datos.dat

Partiendo del código proporcionado, estructuraremos el programa en **dos tipos de proceso** o roles, los cuales tienen funciones específicas:

- **Maestro** o rank 0:

  Es el proceso encargado de **inicializar la ventana** de gráficos.

  Es el proceso encargado de **lanzar los procesos trabajadores** encargados de leer el fichero.

  Es el proceso encargado de **recibir los datos** enviados por los trabajadores y **pintarlos** en la ventana.

- **Trabajadores** o demás rank:

  Son los procesos encargados de **leer el fichero datos.dat** y procesar su información.

Son los procesos encargados de **aplicar filtros** a la imagen que será posteriormente dibujada.

Son los procesos encargados de **enviar los datos** al maestro.

1.3. Diseño del programa

Antes de centrarnos en el diseño del programa, debemos de conocer la estructura del fichero datos.dat para poder trabajar de forma correcta con los datos que este contiene. En nuestro caso, la representación de un pixel de la imagen viene dado como una tripleta de **tres *char unsigned***, correspondientes al **valor RGB** de los colores primarios.

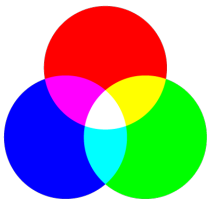

Figura 2: Colores primarios RGB

La imagen que vamos a procesar, tiene un tamaño de **400 por 400 píxeles**. Esto lo tenemos que tener muy en cuenta a la hora del desarrollo de la solución puesto que parametrizaremos la lectura del fichero en base al tamaño de la imagen, de tal forma que podamos escalar el programa para renderizar imágenes de distinto tamaño.

Una vez que conocemos la estructura del fichero datos.dat, nos centraremos en el desarrollo de la solución partiendo por el **maestro**:

- **Inicializar ventana**: Primeramente debemos de crear la ventana gráfica en la cual renderizaremos la imagen una vez procesado el fichero datos.dat. Para realizar este trabajo, se nos ha proporcionado la función **initX()**.
- **Lanzar procesos trabajadores**: Los procesos trabajadores se deben de lanzar en **tiempo de ejecución**. El numero de procesos viene definido como una macro denominada *NUMHIJOS* para parametrizar la lectura concurrente del fichero de una manera mas fácil. La función necesaria para ejecutar dichos procesos, en MPI2 se denomina

MPI_Comm_spawn. Una vez que levantamos los NUMHIJOS procesos, dicha función nos **devuelve un intercomunicador** por el cual se pueden comunicar todos los procesos creados. Dicho intercomunicador lo utilizaremos posteriormente para poder dibujar los puntos en la ventana gráfica.

- **Recibir los puntos**: Una vez que hemos levantado los procesos trabajadores, estos empezaran a procesar el fichero y a enviar al maestro toda la información, por tanto el maestro se debe de encargar de recibir dichos datos para poder renderizarlos. En nuestro caso en concreto hemos definido la función **recibirPuntos()** a la cual le pasamos el **intercomunicador** que comunica los procesos trabajadores, de tal forma que podamos obtener la información enviada por los mismos. En dicha función simplemente tenemos un bucle que realizará tantas iteraciones como píxeles tiene el fichero, de tal forma que recibamos los puntos en un buffer auxiliar, para posteriormente pintarlos mediante la función **dibujarPunto()**.

Hasta aquí hemos definido todas las funciones que debe llevar a cabo el proceso maestro, seguidamente nos centraremos en las de los trabajadores:

- **Abrir fichero**: Una vez que hemos comenzado la ejecución de los procesos trabajadores, estos se encargarán de leer los datos contenido en el fichero datos.dat. Para ello primeramente debemos de abrir el fichero con **permisos de lectura**. Para llevar a cabo dicha tarea haremos uso de la función **MPI_File_open**, a la cual debemos de pasar el nombre del archivo a leer y un manejador de archivo del tipo MPI_file.

- **Acceder a los datos**: Una vez que hemos abierto el fichero con permisos de lectura, cada proceso obtendrá una vista limitada a dicho fichero, de tal forma que podamos leer de forma concurrente dicho archivo sin que los procesos interfieran entre sí. Para ello haremos uso de la funcion **MPI_File_set_view**, a la cual debemos de pasar como argumento el **desplazamiento** a partir del cual queremos que tenga visibilidad dicho proceso. Para calcular el desplazamiento que tiene cada uno de los procesos es necesario **multiplicar el rank** propio del proceso **por el tamaño de datos** que va a leer, de tal forma que si lanzamos 10 procesos, cada uno de ellos solo vera accesible una décima parte del archivo.

- **Leer fichero**: Tras obtener acceso a una vista limitada del fichero, ahora debemos de leer los datos en si que este contiene. Para ello utilizaremos la funcion **MPI_File_read**, con la cual iremos leyendo pixel a pixel todas las **tripletas de *unsigned char*** contenidas en el fichero.

- **Aplicar filtro**: La información de cada pixel de la imagen se representa como una tripleta de los colores básicos: **Rojo, Verde y Azul**, tomando valores desde **0 hasta 255**. Para poder aplicar un filtro a la imagen, debemos de modificar dichos valores en base a formulas matemáticas. En nuestro caso simplemente hemos implementado dos filtros, uno en **escala de grises** y otro en color **sepia**. Para la selección del filtro a aplicar hemos utilizado un ***switch*** y una macro denominada FILTRO. En el caso de que FILTRO tome el valor 0, no se aplicará ningún filtro, si es 1 se aplicará el filtro de escala de grises y en el caso que sea 2, se aplicará un filtro sepia. Los valores que deben tomar, vienen definidos por las siguientes formulas matemáticas:

```
/* Sin filtro */
case 0:
    /*Rojo*/ puntoAPintar[2]=(int)color[0];
    /*Verde*/puntoAPintar[3]=(int)color[1];
    /*Azul*/ puntoAPintar[4]=(int)color[2];
    break;

/* Filtro en blanco y negro */
case 1:
    /*Rojo*/ puntoAPintar[2]=(((int)color[0])*0.393)+(((int)color[1])*0.769)+(((int)color[2])*0.189);
    /*Verde*/puntoAPintar[3]=(((int)color[0])*0.393)+(((int)color[1])*0.769)+(((int)color[2])*0.189);
    /*Azul*/ puntoAPintar[4]=(((int)color[0])*0.393)+(((int)color[1])*0.769)+(((int)color[2])*0.189);
    break;

/*Filtro en sepia */
case 2:
    /*Rojo*/ puntoAPintar[2]=(((int)color[0])*0.393)+(((int)color[1])*0.769)+(((int)color[2])*0.189);
    /*Verde*/puntoAPintar[3]=(((int)color[0])*0.349)+(((int)color[1])*0.686)+(((int)color[2])*0.168);
    /*Azul*/ puntoAPintar[4]=(((int)color[0])*0.272)+(((int)color[1])*0.534)+(((int)color[2])*0.131);
    break;
```

- **Enviar datos**: Una vez que hemos procesado el pixel en base al filtro, debemos de enviar dicho dato al proceso maestro, con las coordenadas x e y donde debe de pintarse. Para ello haremos uso de la función **MPI_Send**, de forma que enviamos el dato de manera síncrona.

Con esto ya hemos cubierto todos los aspectos de diseño correspondientes a los procesos trabajadores y al proceso maestro.

1.4. Flujo de datos MPI

Para el desarrollo de la solución hemos utilizado un total de 7 primitivas de MPI 2. La mayoría de ellas corresponden al acceso a ficheros. A continuación detallaremos brevemente el uso de cada una de las primitivas y su estructura.

La primitiva **MPI_Comm_spawn** nos permite lanzar un numero de procesos que ejecutarán el **mismo binario**. En nuestro caso, levantaremos tantos procesos como le indiquemos mediante la variable *NUMHIJOS* y todos ellos ejecutarán el **binario *pract2***:

```
MPI_Comm_spawn("pract2", MPI_ARGV_NULL, NUMHIJOS, MPI_INFO_NULL,
    0, MPI_COMM_WORLD, &commPadre, codigosError);
```

La primitiva **MPI_File_open** permite abrir un archivo de forma colectiva, de forma que todos los procesos lanzados tengan acceso al mismo. Al tratarse de una operación colectiva, podemos seleccionar el comunicador de forma que todos los procesos incluidos en el mismo, tengan acceso a dicho archivo. En nuestro caso utilizaremos el comunicador MPI_COMM_WORLD.

```
MPI_File_open(MPI_COMM_WORLD, IMAGEN, MPI_MODE_RDONLY,
    MPI_INFO_NULL, &manejadorArchivo);
```

La primitiva **MPI_File_set_view** permite obtener una vista parcial de los datos de un archivo. Nosotros lo hemos utilizado para limitar la vista que cada proceso trabajador tiene al fichero datos.dat, de forma que todos los procesos puedan leer concurrentemente el archivo sin interferirse entre ellos.

```
MPI_File_set_view(manejadorArchivo, rank*bloquePorTrabajador,
    MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, "native", MPI_INFO_NULL
    );
```

Como podemos observar la primitiva tiene como argumentos el manejador del archivo, una vez que lo hemos abierto, y el desplazamiento o bloque al partir del cual tendrá acceso.

La primitiva **MPI_File_read** permite leer datos del fichero. Nosotros lo hemos usado para ir leyendo pixel a pixel la información del fichero.

```
MPI_File_read(manejadorArchivo, color, 3, MPI_UNSIGNED_CHAR, &
    status);
```

La primitiva **MPI_File_close** permite cerrar el acceso al fichero.

```
MPI_File_close(&manejadorArchivo);
```

Por otro lado tenemos las funciones **MPI_Send** y **MPI_Recv**, las cuales permiten el envio y recepción de datos de forma síncrona. Las primitivas la siguiente estructura:

```
MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoDerecho, i,
    MPI_COMM_WORLD);

MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, vecinoIzquierdo, i,
    MPI_COMM_WORLD, &status);
```

En ambas primitivas definimos la **variable** que vamos a enviar, el **numero de datos** que enviamos, el **tipo de variable** que enviamos, el **destinatario** de envío, un **tag** o comentario y el **intercomunicador** al que afecta, y en la primitiva de recepción además incluimos una variable de estado.

1.5. Fuentes del programa

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

#define NIL (0)
//#define NUMHIJOS 8
//#define FILTRO 0
#define IMAGEN "foto.dat"
#define TAMANIOIMAGEN 400


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

```
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, whiteColor, whiteColor);
        XSelectInput(dpy, w, StructureNotifyMask);
        XMapWindow(dpy, w);
        gc = XCreateGC(dpy, w, 0, NIL);
        XSetForeground(dpy, gc, blackColor);

        for(;;) {
                XEvent e;
                XNextEvent(dpy, &e);
                if (e.type == MapNotify)
                        break;
        }

        mapacolor = DefaultColormap(dpy, 0);

}

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);

}


void recibirPuntos(MPI_Comm commPadre){

        int puntoADibujar[5];
        int i;

        for(i=0; i<TAMANIOIMAGEN*TAMANIOIMAGEN; i++){
```

```
                MPI_Recv(&puntoADibujar, 5, MPI_INT,
                    MPI_ANY_SOURCE, MPI_ANY_TAG, commPadre,
                    MPI_STATUS_IGNORE);
                dibujaPunto(puntoADibujar[0], puntoADibujar[1],
                    puntoADibujar[2], puntoADibujar[3],
                    puntoADibujar[4]);
        }
}



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

        int rank, size;
        MPI_Status status;
        MPI_Comm commPadre;
        int codigosError[NUMHIJOS];
        int puntoAPintar[5];

        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)){

                initX();
                MPI_Comm_spawn("pract2", MPI_ARGV_NULL, NUMHIJOS
                    , MPI_INFO_NULL, 0, MPI_COMM_WORLD, &
                    commPadre, codigosError);
                recibirPuntos(commPadre);
                printf("Imagen dibujada. Sera mostrada durante 5
                     segundos.\n");

/* Esperamos 5 segundos para poder visualizar la imagen */
                sleep(5);

        }else {

                int filasPorTrabajador=TAMANIOIMAGEN/NUMHIJOS;
                int bloquePorTrabajador=filasPorTrabajador*
                    TAMANIOIMAGEN*3*sizeof(unsigned char);
                int inicioLectura=rank*filasPorTrabajador;
```

```
                int finalLectura=inicioLectura+
                    filasPorTrabajador;

                MPI_File manejadorArchivo;
                MPI_File_open(MPI_COMM_WORLD, IMAGEN,
                    MPI_MODE_RDONLY, MPI_INFO_NULL, &
                    manejadorArchivo);
                MPI_File_set_view(manejadorArchivo, rank*
                    bloquePorTrabajador, MPI_UNSIGNED_CHAR,
                    MPI_UNSIGNED_CHAR, "native", MPI_INFO_NULL);

                unsigned char color[3];

                int i;
                int j;

                if(rank==NUMHIJOS-1){
                        finalLectura=TAMANIOIMAGEN;
                }

                for(i=inicioLectura; i<finalLectura; i++){
                        for(j=0; j<TAMANIOIMAGEN; j++){

                                MPI_File_read(manejadorArchivo,
                                    color, 3, MPI_UNSIGNED_CHAR,
                                    &status);
                                puntoAPintar[0]=j;
                                puntoAPintar[1]=i;

/* Aplicamos un filtro a la imagen */
                                switch(FILTRO){

/* Sin filtro */
                                        case 0:
                                        puntoAPintar[2]=(int)
                                            color[0];
                                        puntoAPintar[3]=(int)
                                            color[1];
                                        puntoAPintar[4]=(int)
                                            color[2];
                                        break;

/* Filtro en blanco y negro */
                                        case 1:
```

```
                                        puntoAPintar[2]=(((int)
                                            color[0])*0.393)+(((
                                            int)color[1])*0.769)
                                            +(((int)color[2])
                                            *0.189);
                                        puntoAPintar[3]=(((int)
                                            color[0])*0.393)+(((
                                            int)color[1])*0.769)
                                            +(((int)color[2])
                                            *0.189);
                                        puntoAPintar[4]=(((int)
                                            color[0])*0.393)+(((
                                            int)color[1])*0.769)
                                            +(((int)color[2])
                                            *0.189);
                                        break;

/*Filtro en sepia */
                                        case 2:
                                        puntoAPintar[2]=(((int)
                                            color[0])*0.393)+(((
                                            int)color[1])*0.769)
                                            +(((int)color[2])
                                            *0.189);
                                        puntoAPintar[3]=(((int)
                                            color[0])*0.349)+(((
                                            int)color[1])*0.686)
                                            +(((int)color[2])
                                            *0.168);
                                        puntoAPintar[4]=(((int)
                                            color[0])*0.272)+(((
                                            int)color[1])*0.534)
                                            +(((int)color[2])
                                            *0.131);
                                        break;
                                }

                                if(puntoAPintar[2]>255){
                                        puntoAPintar[2]=255;
                                }

                                if(puntoAPintar[3]>255){
                                        puntoAPintar[3]=255;
                                }
```

```
                                    if(puntoAPintar[4]>255){
                                            puntoAPintar[4]=255;
                                    }

                                    MPI_Send(&puntoAPintar, 5,
                                        MPI_INT, 0, 1, commPadre);
                            }
                    }

                    MPI_File_close(&manejadorArchivo);
            }

            MPI_Finalize();
            return 0;
}
```

1.6. Instrucciones de compilación y ejecución

Para compilar los programas simplemente debemos de ejecutar las siguientes instrucciones en un terminal en el directorio donde se encuentran los archivos fuente y el Makefile:

1. **Compilar**:

```
$ make compilar
```

2. **Compilar con filtro escala de grises**:

```
$ make compilarBYN
```

3. **Compilar con filtro sepia**:

```
$ make compilarSepia
```

Si queremos ejecutar el programa simplemente tenemos que ejecutar la instrucción:

1. **Ejecutar**:

```
$ make ejecutar
```

Si queremos limpiar el directorio del programa simplemente ejecutamos la siguiente instrucción:

1. **Limpiar directorio**:

```
$ make clean
```

2. Conclusiones

Como conclusiones puedo decir que me ha parecido muy interesante poder acceder de forma **concurrente** a un mismo fichero por parte de varios procesos, pudiendo limitar la vista que tienen estos, ya que esto nos permitiría poder leer un fichero de gran tamaño utilizando diferentes computadores, de tal forma que distribuyamos la carga de trabajo de forma equilibrada. Por otra parte he de decir que he tenido complicaciones en poder desarrollar una solución que envíe y reciba los datos de forma asíncrona, así que he optado por la opción **síncrona**. Esto ha supuesto que a la hora de levantar un numero grande de procesos, se relentize de manera sustancial el procesamiento de la imagen, al contrario que cabria esperar puesto que la carga de trabajo se distribuye a más procesos.

3. Bibliografía

Referencias

[1] Open MPI

[2] Filtro imagen