Renderizado Distribuido

1. Enunciado del problema:

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

Lanzaremos un único proceso que 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.

2. Planteamiento de la solución:

Para la solución he optado como dice el enunciado por un único proceso que es el encargado de crear a los demás procesos hijos, estos son los que tienen acceso al fichero "foto.dat" y con una constante definida como "FILTRO" podemos elegir que filtro queremos aplicar a la foto.

Una vez que los procesos leen el valor del pixel y le aplican el filtro que seleccionemos, estos le mandan los valores al proceso "rank 0" que es el encargado de imprimir los pixeles en una "Xwindow".

3. Diseño del programa:

En el programa lo primero que hago es un `MPI_Comm_Spawn` para crear los hijos que van a ser los encargados de acceder a "foto.dat", a continuación, el proceso "rank 0" se espera a recibir los distintos pixeles.

En los procesos hijos creo las variables necesarias para controlar que parte del fichero y cuantos datos tiene que procesar cada uno. Después creo un `MPI_File` para poder abrir el fichero con `MPI_File_open` y poder asignarle cada una parte a cada proceso con `MPI_File_set_view`.

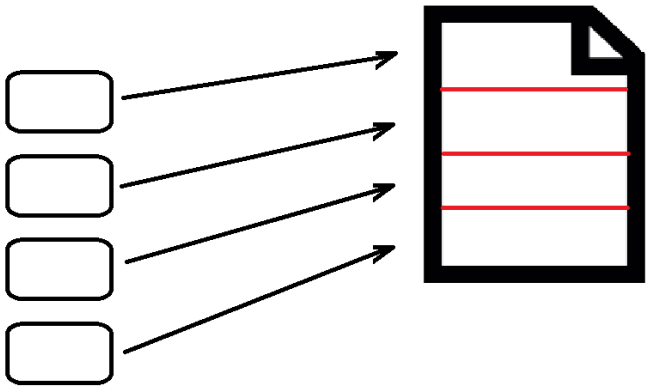
Una vez hecho esto, cada proceso lee los datos que les corresponde con sus correspondientes coordenadas y con la opción marcada de filtro le aplican un filtro u otro al pixel.

Finalmente mandan tanto el pixel ya procesado como las coordenadas del mismo al proceso "rank 0". Cuando este proceso recibe los datos, manda a la función `dibujaPunto` que es la encargada de ir colocando los pixeles en la pantalla.

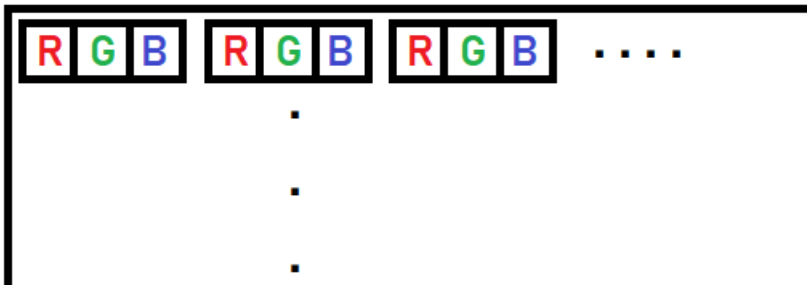
4. Explicación del flujo de datos:

Por ejemplo, supongamos que nuestro programa una vez lanzado, crea 4 hijos para realizar el renderizado de la imagen.

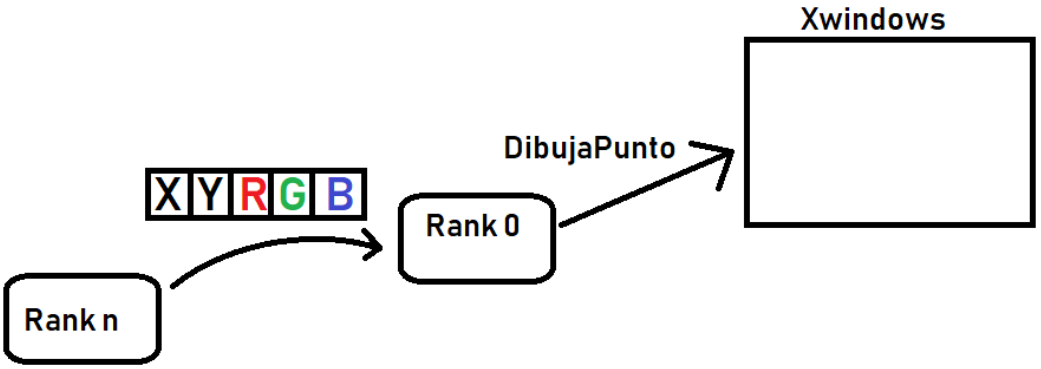
Cada uno de los hijos se encargará de acceder a una parte del fichero "foto.dat".



Una vez que cada uno de los procesos tenga su parte asignada, comenzaran a leer cada una de las tripletas de colores que representan a cada pixel.



Una vez que leen el valor de cada uno, le aplican el filtro seleccionado y se lo mandan junto con las coordenadas X e Y al proceso "rank 0" para que los ilumine en una "Xwindow" a través de la función `dibujaPunto`.



5. Fuentes del programa:

El código del programa es el siguiente:

```
/* Pract2  RAP 09/10    Javier Ayllon*/
/*Código modificado por Juan Mena*/

#include <openmpi/mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <assert.h>
#include <unistd.h>
#define NIL (0)
#define NUM_HIJOS 4
#define FILTRO 0
#define MIN(a, b) (((a) < (b)) ? (a) : (b))

/*Variables Globales */

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

/*Funciones auxiliares */

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);

}

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);
}

/* Programa principal */

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

  int rank,size;
  MPI_Comm commPadre;
  int tag;
  MPI_Status status;
  int buf[5];
  int errcodes[NUM_HIJOS];
  int seguir = 1;

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

  /*Compruebo si hay procesos para realizar el renderizado*/
  if(NUM_HIJOS == 0){
    seguir = 0;
    fprintf(stderr, "ERROR, debe haber al menos 1 proceso hijo para el
                                                    renderizado\n");
  }

  if ( (commPadre==MPI_COMM_NULL)
        && (rank==0) )  {
    if(seguir != 0){
      initX();
      /* Codigo del maestro */

      /*En algun momento dibujamos puntos en la ventana algo como
      dibujaPunto(x,y,r,g,b);  */

        MPI_Comm_spawn("pract2",MPI_ARGV_NULL,NUM_HIJOS,MPI_INFO_NULL,
                        0, MPI_COMM_WORLD, &commPadre, errcodes);

        for(int i=0;i<400*400;i++){
        MPI_Recv(&buf,5,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,commPadre,
                                              MPI_STATUS_IGNORE);
```

```
        dibujaPunto(buf[0],buf[1],buf[2],buf[3],buf[4]);
        }
        sleep(3);
    }
        } else {
                if(seguir != 0){
        /* Codigo de todos los trabajadores */
        /* El archivo sobre el que debemos trabajar es foto.dat */

        int filasPorProceso=400/NUM_HIJOS;
        intelementosPorProceso=filasPorProceso*400*sizeof(unsigned
                                                            char)*3;

        int inicioLectura=filasPorProceso*rank;
        int finalLectura=inicioLectura+filasPorProceso;

        MPI_File archivo;

        /*Abrimos el fichero con permisos de lectura*/
        MPI_File_open(MPI_COMM_WORLD,"foto.dat",MPI_MODE_RDONLY,
                                    MPI_INFO_NULL, &archivo);

        MPI_File_set_view(archivo,elementosPorProceso*rank,
        MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, "native", MPI_INFO_NULL);

        /*Array para guardar los valores de RGB*/
        unsigned char rgb[3];

        /*Se controla el tamaño del último proceso*/
        if(rank == NUM_HIJOS-1)
           finalLectura=400;

        for(int i=inicioLectura;i<finalLectura;i++){
           for(int j=0;j<400;j++){
                MPI_File_read(archivo,rgb,3,MPI_UNSIGNED_CHAR,&status);
                        buf[0]=j;       /*Coordenada x*/
                        buf[1]=i;       /*Coordenada y*/

                        switch(FILTRO){
                          case 0: /*Sin filtro*/
                            buf[2]=(int)rgb[0]; /*Rojo*/
                            buf[3]=(int)rgb[1]; /*Verde*/
                            buf[4]=(int)rgb[2]; /*Azul*/
                          break;

                          case 1: /*Filtro negativo*/
                            buf[2]=255 - (int)rgb[0]; /*Rojo*/
                            buf[3]=255 - (int)rgb[1]; /*Verde*/
                            buf[4]=255 - (int)rgb[2]; /*Azul*/
                          break;

                          case 2: /*Filtro blanco y negro*/
                            buf[2]=((int)rgb[0]+(int)rgb[1]+
                                                (int)rgb[2])/3; /*Rojo*/

                            buf[3]=((int)rgb[0]+(int)rgb[1]+
                                                (int)rgb[2])/3; /*Verde*/

                            buf[4]=((int)rgb[0]+(int)rgb[1]+
                                                (int)rgb[2])/3; /*Azul*/
                          break;
```

```
                        case 3: /*Filtro sepia*/
                          buf[2]=MIN(((.3*rgb[0]+.6*rgb[1]+.1*rgb[2])
                                      +40),255);      /*Rojo*/

                          buf[3]=MIN(((.3*rgb[0]+.6*rgb[1]+.1*rgb[2])
                                      +15),255);      /*Verde*/

                          buf[4]=.3*rgb[0]+.6*rgb[1]+.1*rgb[2];
                                                      /*Azul*/
                        break;
                        }
                    MPI_Bsend(&buf, 5, MPI_INT, 0, 0, commPadre);
                }
        }
        MPI_File_close(&archivo);
        }
  }
        MPI_Finalize();
        return 0;
}
```

6. Instrucciones de como compilar y ejecutar:

Para ejecutar el programa deberemos utilizar el comando "`mpirun -np 1 ./pract2`" para hacer uso de las primitivas MPI.

Para ejecutar el programa usaremos el comando "`mpicc pract2.c -o pract2 -lX11`".

Para hacer uso del Makefile, basta con escribir "`make compileRender`" para realizar la compilación y con "`make runRender`" ejecutamos el programa.

7. Conclusiones:

La principal conclusión es que al realizar este programa de renderizado de imágenes con MPI y de forma paralela podemos reducir el tiempo de ejecución y podemos aprovechar más las características de la máquina. Además, el programa tiene una complejidad menor y un mejor rendimiento que si lo realizásemos de forma secuencial.