PRÁCTICA 2. SISTEMA DISTRIBUIDO DE RENDERIZADO DE GRÁFICOS

DISEÑO DE INFRAESTRUCTURA DE RED

JOSE MIGUEL CABANILLAS MANSILLA

Contenido

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 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.

Para compilar el programa para openMPI en Linux el comando es: **mpicc pract2.c -o pract2 -IX11.**

2. Planteamiento de la solución

Tenemos que tener en cuenta que ya se nos proporciona una estructura de código sobre la cual hemos trabajado.

En resumidas cuentas, nuestra tarea consiste en lo siguiente:

Como se nos especifica en el enunciado, el proceso principal (Rank 0) será el encargado de levantar un número de procesos que nosotros queramos y luego tras esto esperará a que los procesos que ha creado le manden los datos necesarios. Una vez recibidos estos datos ejecutará la función que se nos ha proporcionado para dibujar la foto que tenemos que renderizar.

Por otro lado, una vez que el Rank 0 ha creado el número de procesos que nosotros le indicamos para trabajar en paralelo con la imagen, tendremos que asignar a cada proceso una zona sobre la que trabajar en paralelo. Cada proceso tendrá una única zona específica.

Mientras que un proceso lee su zona, irá almacenando en un buffer 5 datos, que son los que enviaremos al proceso padre para poder ejecutar la función que la imagen a puntos. Estos datos son la posición en x e y del pixel y los valores que toman como R, G y B.

En cada pasada del bucle mandaremos estos valores, pero antes, tendremos que aplicar un filtro, para ver la imagen con distintos colores y tendremos que controlar, por ejemplo, que a la hora de aplicar un filtro los valores RGB no superen el valor 255.

3. Diseño

Solo comentaré aquellos métodos o implementaciones realizadas por mí:

- Void filtros (int filtro,int *buf,int i, int j, unsigned char *rgb). Este método es el utilizado para poder aplicar un tipo de filtro a nuestra imagen. La variable filtro es una variable definida //define Filtro cuyo valor podemos cambiar en tiempo de compilación para así poder elegir el filtro que queramos. Le pasamos el buffer que enviaremos al Rank 0. Las coordenadas x e y que no cambian y una matriz o array de unsigned char que es donde hemos ido almacenando los valores RGB que ha ido leyendo cada proceso.
  Con este método no solo aplicamos el filtro, sino que también dejamos preparado el array de 5 buff para que cada proceso lo mande al proceso padre (Rank 0). Para asignar el filtro basta con sumar o multiplicar una serie de valores a los valores leídos de la imagen por defecto.

```
void filtros(int filtro,int *buf,int i, int j,unsigned char *rgb){

 buf[0]=j; //Almacenamos los valores de la posicion x y del pixel
  buf[1]=i; //leido

        switch(FILTRO){
          case 0: //Mostrar la imagen de manera normal (SIN FILTROS)
            buf[2]=(int)rgb[0];
            buf[3]=(int)rgb[1];
            buf[4]=(int)rgb[2];
          break;

          case 1: //Mostar la imagen on un filtro sepia
            buf[2]=(int)rgb[0]*0.6;
            buf[3]=(int)rgb[1]*0.28;
            buf[4]=(int)rgb[2]*0.12;
          break;

          case 2: //Mostrar la imagen con un filtro Verde
            buf[2]=(int)rgb[0]+0;
            buf[3]=(int)rgb[1]+145;
            buf[4]=(int)rgb[2]+80;
          break;

          case 3: //Mostrar la imagen con un filtro Azul
            buf[2]=(int)rgb[0]+0;
            buf[3]=(int)rgb[1]+112;
            buf[4]=(int)rgb[2]+184;
          break;

          case 4: // Mostrar la imagen con un filtro Rojo
            buf[2]=(int)rgb[0]+230;
            buf[3]=(int)rgb[1]+0;
            buf[4]=(int)rgb[2]+38;
          break;
        }
```

- Void control(int *buf). Con este método controlamos que una vez que hemos aplicado el filtro, los valores del buffer de tamaño 5, correspondientes a RGB (posiciones 2,3 y 4 del buffer) no superen el valor 255, que es el máximo que puede tener cada punto en R, G y B.

```
void control(int *buf){ // Este metodo lo aplicamos para en caso de que
  // añadamos un filtro, los valores no superen el 255 distorsionando así la imagen
if(buf[2]>255){
            buf[2]=255;
          }
          if(buf[3]>255){
            buf[3]=255;
          }
          if(buf[4]>255){
            buf[4]=255;
          }

}
```

Una vez que hemos comentado los métodos que he implementado vamos a explicar como funciona el main.

El main podemos dividirlo en dos partes; Una es la que realiza el proceso que lanzamos al ejecutar el programa, que es el que se encarga de levantar el número de procesos que nosotros queremos por medio de MPI_Comm_spawn y luego le dejamos esperando a que los procesos que levantan le manden los datos para dibujar la imagen.

La otra parte es la relacionada con los procesos lanzados. Calculamos la zona de trabajo de cada uno de ellos. Abrimos el archivo con MPI_File_open, modificamos la vista con MPI_File_set_view. Luego cada proceso se encargará de leer su zona con MPI_File_read, aplicar el filtro que hayamos elegido y enviar el buffer resultante en cada pasada del bucle.

Dentro del mismo mundo, tenemos dos comm, uno es el comm del padre, y otro es el intercomunicador para poder relacionar al padre con los hijos. Por medio del intercomunicador mandamos y recibimos los datos.

```
if ( (commPadre==MPI_COMM_NULL) && (rank==0) )  {
    initX();
  MPI_Comm_spawn("pract2",MPI_ARGV_NULL, NUMEROPROCESOS, MPI_INFO_NULL,0,MPI_COMM_WORLD,&interComm,errcodes);

  //Realizamos un bucle desde 0 a 400*400 que es la dimension de la imagen
  for(i=0;i<(LONLADO*LONLADO);i++){
    MPI_Recv(&buf,5,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,interComm,&status);

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

    dibujaPunto(buf[0],buf[1],buf[2],buf[3],buf[4]);
  }


  sleep(5); //dejamos 5 segundos para que muestre el gato
}
```

Esta es la primera parte del main correspondiente al proceso padre

```
else {
  int tamanioLeerProceso,desplazamiento,filaInicio,filaFinal;
  tamanioLeerProceso=LONLADO/NUMEROPROCESOS;
  desplazamiento=tamanioLeerProceso*LONLADO*3*sizeof(unsigned char);
  //Será las filas que lee cada proceso por las columnas que son 400 por la tripleta que leerá en cada punto
  //que es del tipo unsigned char
  unsigned char rgb[3]; //La tripleta de colores por puntos

  filaInicio=rank*tamanioLeerProceso; //Fila en la cual empezamos a leer la imagen, dependerá del rank o identificador
  filaFinal=filaInicio+tamanioLeerProceso; //Así y teniendo en cuenta lo anterior variará la filaFinal
  //Tendremos para cada proceso que se ejecute un marco en el que moverse


  /* Codigo de todos los trabajadores */
  /* El archivo sobre el que debemos trabajar es foto.dat */

  MPI_File_open(MPI_COMM_WORLD,FOTO,MPI_MODE_RDONLY,MPI_INFO_NULL,&fphoto);
  MPI_File_set_view(fphoto,desplazamiento*rank,MPI_UNSIGNED_CHAR,MPI_UNSIGNED_CHAR,"native",MPI_INFO_NULL);
  //Por medio de MPI_File_set_view.
  //Cambiamos la vista del proceso sobre el archivo
  //La vista la definimos por medio de tres parametos :Desplazamiento,tipo de datos elemental

  for (i=filaInicio;i<filaFinal;i++){
    for(j=0;j<LONLADO;j++){
      MPI_File_read(fphoto,rgb,3,MPI_UNSIGNED_CHAR,&status);


      filtros(FILTRO,buf,i,j,rgb); //Llamamos al metodo de aplicacion de filtros

      control(buf); // Controlamos si aumentan los valores
      // a más de 255, que es el valor maximo para cada valor rgb
      //Si no lo controlasemos, al aplicar un filtro se vería una imagen errónea

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


  }
  MPI_File_close(&fphoto); //Cerramos el fichero, como si se tratase de un fichero
}
```

Esta es la segunda parte, correspondiente a los procesos.

Para controlar que pueda ejecutar el programa con un número de procesos no divisibles por el tamaño de la imagen de manera correcta.

```
    if(rank==NUMEROPROCESOS-1){
    filaFinal=LONLADO;      //Controlo que si el número de procesos no es divisible entre
                // el tamaño del bloque, se ejecute el programa correctamente
}
```

Para asegurarnos de cada proceso empieza y acaba en un punto único y ningún proceso solapa con la parte de otro, mostramos su Rank, el punto de inicio, final y el total del tamaño del bloque que recorre.

```
printf("Hola soy el proceso %d y voy a empezar a leer en el punto x=%d , y= %d \n",rank,inicio,fin);
printf("Soy el proceso %d y el tamaño de bloque que he recorrido es: %d \n",rank, fin-inicio);
```

4. Explicación del flujo de datos de los comandos MPI

- **int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[]).**
  Esta primitiva intenta iniciar "maxprocs" copias idénticas del programa indicado en "command". Además, se le pueden indicar argumentos de entrada al programa y otra información como por ejemplo el padre de los nodos que van a crear. La sentencia devuelve el intercomunicador con el que el nodo que ha ejecutado la función podrá comunicarse con los nodos que serán creados. En el programa lo utilizamos para spawnear el número de procesos que hemos definido.
- **Int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fb):**
  Abre el archivo especificado mediante "filename". Como es una rutina colectiva, todos los nodos que ejecuten el comando y que estén en el mismo comunicador, recibirán el descriptor de archivo del fichero. En nuestro caso el descriptor en nuestro caso se trata de fphoto y el filename se trata de nuestra foto.
- **Int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info).**
  Esta sentencia cambia la vista que un proceso tiene sobre un archivo. El inicio de esta vista es "disp" que es el desplazamiento. El tipo de datos se define con etype y la distribución de los datos se da en "filetype". Además, se resetea el puntero de manera independiente de cada proceso a 0, haciendo que cada proceso maneje su propio fragmento como un fichero único. Asignamos a cada proceso un inicio marcado por el desplazamiento hasta el final de la foto.
- **Int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status).**
  Esta función intenta leer desde el archivo asociado con el descriptor de archivo fh, el número de elementos igual al especificado en "count" y del tipo "datatype". Los datos leídos se almacenan en el array "rgb" y la función almacena en "status" información adicional. Una vez que leemos los puntos de la imagen uno a uno los vamos enviando por medio de MPI_Send al proceso principal
- **Int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm).**
  Se envía "buf" según el "count" y el "datatype" especificados a "dest" dentro del comunicador "comm". Además, podemos indicar con tag cualquier tipo de etiqueta de identificación para el mensaje. Mandamos el buffer en el cual tenemos almacenados los tres puntos rgb y la posición x,y de cada pixel.
- **Int MPI_File_close(MPI_File *fh).**
  La función sincroniza el estado del fichero y después cierra el fichero asociado al descriptor "fh"
- **Int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**.
  El nodo que llama a esta función obtendrá los datos que tiene el "buf" según el "count"(tamaños) y el "datatype". También se puede especificar el origen del dato(source) e identificar el mensaje a recibir(tag). Recibimos cada punto de cada uno de los procesos para que el proceso principal dibuje la imagen poco a poco.

5. Código fuente

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

#define NIL (0)
#define NUMEROPROCESOS 4 //Numero de procesos que lanzara el rank 0
#define FOTO "foto.dat"
#define LONLADO 400 // La longitud de un lado de la imagen, la imagen es 400x400
//#define FILTRO //A la hora de compilar elegiremos un valor para ver que filtro aplicamos a la
imagen

/*Variables Globales */
void filtros(int filtro,int *buf,int i, int j,unsigned char *rgb);
void control(int *buf);
XColor colorX;
Colormap mapacolor;
char cadenaColor[]="#000000";
Display *dpy;
Window w;
GC gc;
int errcodes[NUMEROPROCESOS];
/*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,interComm;
  MPI_Status status;
  int buf[5]; //Buffer para almacenar los 5 elementos que necesito
  //coordenada en x , coordenada en y, así como la tripleta de
  //los colores, r, g ,b
  MPI_File fphoto; //descriptor Fichero MPI
  int i,j;

  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, NUMEROPROCESOS,
MPI_INFO_NULL,0,MPI_COMM_WORLD,&interComm,errcodes);

    //Realizamos un bucle desde 0 a 400*400 que es la dimensión de la imagen
    for(i=0;i<(LONLADO*LONLADO);i++){
      MPI_Recv(&buf,5,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,interComm,&status);

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

      dibujaPunto(buf[0],buf[1],buf[2],buf[3],buf[4]);
    }


    sleep(5); //dejamos 5 segundos para que muestre el gato
  }
  else {
    int tamanioLeerProceso,desplazamiento,filaInicio,filaFinal;
    tamanioLeerProceso=LONLADO/NUMEROPROCESOS;
    desplazamiento=tamanioLeerProceso*LONLADO*3*sizeof(unsigned char);
```

```
   //Será las filas que lee cada proceso por las columnas que son 400 por la tripleta que leerá en cada punto
   //que es del tipo unsigned char
   unsigned char rgb[3]; //La tripleta de colores por puntos

   filaInicio=rank*tamanioLeerProceso;
   filaFinal=filaInicio+tamanioLeerProceso;

   /* Codigo de todos los trabajadores */
   /* El archivo sobre el que debemos trabajar es foto.dat */

   MPI_File_open(MPI_COMM_WORLD,FOTO,MPI_MODE_RDONLY,MPI_INFO_NULL,&fphoto);

MPI_File_set_view(fphoto,desplazamiento*rank,MPI_UNSIGNED_CHAR,MPI_UNSIGNED_CHAR,"native",MPI_INFO_NULL);
   //Por medio de MPI_File_set_view. Cambiamos la vista del proceso sobre el archivo
   //para poder trabajar en paralelo
   //La vista la definimos por medio de tres parametos :Desplazamiento,tipo de datos elemental
          if(rank==NUMEROPROCESOS-1){
                filaFinal=LONLADO;    //Controlo que si el número de procesos no es divisible entre el tamaño del bloque, se ejecute el programa correctamente
          }
int inicio,fin;
inicio=filaInicio*LONLADO;
fin=filaFinal*LONLADO;
   printf("Hola soy el proceso %d y voy a empezar a leer en el punto x=%d , y= %d \n",rank,inicio,fin);

printf("Soy el proceso %d y el tamaño de bloque que he recorrido es: %d \n",rank, fin-inicio);

   for (i=primeraFila;i<ultimaFila;i++){
    for(j=0;j<LONLADO;j++){
      MPI_File_read(fphoto,rgb,3,MPI_UNSIGNED_CHAR,&status);


      filtros(FILTRO,buf,i,j,rgb); //Llamamos al metodo de aplicacion de filtros

      control(buf); // Controlamos si aumentan los valores
      // a más de 255, que es el valor maximo para cada valor rgb
      //Si no lo controlasemos, al aplicar un filtro se vería una imagen errónea

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

   }
   MPI_File_close(&fphoto); //Cerramos el fichero, como si se tratase de un fichero
 }

 MPI_Finalize();

}
```

```
void control(int *buf){ // Este metodo lo aplicamos para en caso de que
  // añadamos un filtro, los valores no superen el 255 distorsionando así la imagen
if(buf[2]>255){
  buf[2]=255;
 }
if(buf[3]>255){
  buf[3]=255;
 }
if(buf[4]>255){
  buf[4]=255;
}

}

void filtros(int filtro,int *buf,int i, int j,unsigned char *rgb){

 buf[0]=j; //Almacenamos los valores de la posicion x y del pixel
  buf[1]=i; //leido

       switch(FILTRO){
         case 0: //Mostrar la imagen de manera normal (SIN FILTROS)
           buf[2]=(int)rgb[0];
           buf[3]=(int)rgb[1];
           buf[4]=(int)rgb[2];
         break;

         case 1: //Mostar la imagen on un filtro sepia
           buf[2]=(int)rgb[0]*0.6;
           buf[3]=(int)rgb[1]*0.28;
           buf[4]=(int)rgb[2]*0.12;
         break;

         case 2: //Mostrar la imagen con un filtro Verde
           buf[2]=(int)rgb[0]+0;
           buf[3]=(int)rgb[1]+145;
           buf[4]=(int)rgb[2]+80;
         break;

         case 3: //Mostrar la imagen con un filtro Azul
           buf[2]=(int)rgb[0]+0;
           buf[3]=(int)rgb[1]+112;
           buf[4]=(int)rgb[2]+184;
         break;

         case 4: // Mostrar la imagen con un filtro Rojo
           buf[2]=(int)rgb[0]+230;
           buf[3]=(int)rgb[1]+0;
           buf[4]=(int)rgb[2]+38;
         break;
       }
}
```

6. Instrucciones de cómo compilar y ejecutar

Utilizaremos el makefile que hemos implementado. Las opciones que este nos proporciona son las siguientes y tendremos que ejecutarlas sobre el directorio en el que se encuentre el código fuente:

- **make clean**: Ejecutamos este comando para poder limpiar el ejecutable de anteriores compilaciones.
- **make practica2**: Compilamos el programa y lo ejecutamos sin aplicar ningún filtro.
- **make practica2Sepia**: Compilamos el programa y lo ejecutamos con un filtro en sepia.
- **make practica2Verde**: Compilamos el programa y lo ejecutamos con un filtro verde.
- **make practica2Azul**: Compilamos el programa y lo ejecutamos con un filtro azul.
- **make practica2Rojo**: Compilamos el programa y lo ejecutamos con un filtro rojo.

7. Conclusiones

A medida que vamos aumento el número de procesos para trabajar la imagen en paralelo, más rápido obtenemos la misma. Distribuir la carga de trabajo entre distintos procesos, es a la larga más eficaz que asignarle un trabajo pesado a un único proceso. Esto con una imagen pequeña como la que se nos ha proporcionado en el ejercicio, es aparentemente complicado de verlo, pero en grandes sistemas que tengan que tratar datos gigantescos día a día, puede suponer una gran mejora y ahorro de tiempo.

Además, nosotros hemos realizado está práctica en un mismo ordenador lanzando distintos procesos, enfocando esto hacia un gran sistema distribuido, en vez de lanzar un proceso, podríamos cargar toda esa tarea en una máquina o nodo de un clúster que este dedicada a un trabajo en especial, consiguiendo un gran rendimiento.