PRÁCTICA 2: SISTEMA DISTRIBUIDO DE RENDERIZADO DE GRÁFICOS

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

ASIGNATURA: DISEÑO DE INFRAESTRUCTURA DE RED

AUTOR: MIGUEL DE LAS HERAS FUENTES

Sumario

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 “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 -lX11
```

1.2. PLANTEAMIENTO DE LA SOLUCIÓN

Para la implementación de la solución se nos ha proporcionado una plantilla en C que contiene todos los métodos necesarios para levantar un servidor X11 que será donde se mostrará la imagen; y para dibujar todos los puntos de la misma. Además, también se nos ha proporcionado un fichero datos.dat que contiene los valor R,G,B de cada pixel de la imagen que queremos mostrar.

Tal y como se aprecia en el enunciado del problema, el programa tendrá dos tipos de procesos o roles:

- **rank 0 o maestro:**

  - Inicializará el servidor X11 con la ventana de gráficos donde se mostrará la imagen.

  - Lanzará el resto de procesos trabajadores

- Recibirá y pintará los datos enviados por el resto de procesos que contendrán los pixels de la imagen.

- **Resto de ranks o trabajadores:**

  - Leerán del fichero datos.dat la parte que se les haya asignado y creará los puntos en base a los datos leídos.

  - Enviará los datos al proceso maestro(rank 0).

1.3. DISEÑO DEL PROGRAMA

Antes de adentrarnos en la solución propuesta es necesario conocer la estructura del fichero datos.dat ya que gran parte de la lógica del programa depende del tamaño y estructura de los datos que este fichero contiene.

Este fichero contiene la imagen que debemos procesar y estará formado por un total de 400 filas y 400 columnas de puntos de manera que cada punto estará formado por una tripleta de datos de tipo unsigned char; por tanto, tendremos 480.000 datos de tipo unsigned char.

Una vez conocida la estructura del fichero que deberemos leer; podemos comenzar con la explicación de la solución.

Para diseñar la solución hemos utilizado un único archivo .c llamado pract2.c. Este archivo contiene un total de cuatro métodos; dos de los cuales fueron proporcionados por el profesor de la asignatura. Estos métodos son:

- **initX**: Método proporcionado por el profesor que permite levantar un servidor X11 con una ventana gráfica donde podremos renderizar la imagen formada por los datos del archivo datos.dat.
- **DibujaPunto**: Método proporcionado por el profesor que permite representar un punto en la ventana creada en el método anterior mediante la asignación de unas coordenadas x e y, y unos valores R,G,B.
- **ImprimirPunto**: Este método lo utilizaremos para que el proceso maestro reciba todos los puntos obtenidos por los procesos trabajadores. Para ello utilizaremos la función MPI_Recv que nos permitirá recibir el punto y la llamada al método “dibujaPunto” que nos permitirá dibujarlo en el servidor X11.
- **Main**: Este es el método principal del programa y el que contiene toda la lógica de la solución. Aquí podemos encontrar dos partes bien diferenciadas, una que realiza el proceso maestro y otra que realizarán el resto de trabajadores.

  El proceso maestro realizará las siguientes tareas:

  - En primer lugar, este proceso inicializará el servidor X11 para que más tarde podamos mostrar la imagen renderizada. Para realizar esta tarea, llamará al método initX proporcionado por el profesor.

- Una vez inicializado el servidor, creará los procesos trabajadores mediante la llamada a la función MPI_Comm_spawn. El número de procesos que debe lanzar el proceso maestro estará definido en tiempo de compilación en la macro NUMPROCESOS. Además, a estos procesos se les proporcionará el intercomunicador commPadre obtenido de la llamada a la función MPI_Comm_get_parent realizada por el proceso maestro.

- A continuación, el proceso maestro comenzará con la recepción y posterior representación de los puntos que serán enviados por los procesos trabajadores. Para realizar esta tarea, el proceso llamará al método imprimirPunto que hemos explicado anteriormente.

- Una vez representados todos los puntos, el proceso esperara 10 segundos mediante la función sleep(10). Este será el tiempo que podremos ver la imagen hasta que desaparezca.

Por su parte, cada proceso trabajador creado por el maestro deberá realizar lo siguiente:

- Lo primero que debe hacer este proceso es calcular la porción de documento que le ha sido asignada. Para ello, se deberán tener en cuenta la cantidad de procesos lanzados, la cantidad de filas y columnas y el tamaño de cada dato de tipo unsigned char.

- Una vez calculada la parte que le corresponde al proceso, este deberá abrir el fichero datos.dat con permisos de lectura para poder acceder a los datos de los puntos correspondientes a la porción de la imagen que le ha sido asignada. Esta tarea se llevará a cabo con la llamada a la función MPI_File_open.

- Como dijimos al comienzo del documento, el fichero datos.dat será accedido por todos los procesos trabajadores de forma simultánea. Por eso es importante utilizar algún mecanismo de sincronización entre los procesos que impidan que unos interfieran con otros en la lectura de los datos. Para esto, el proceso llamará a la función MPI_File_set_view con un argumento desplazamiento que le indicará la posición desde la cuál puede comenzar a leer datos.

- Una vez obtenido el acceso al fichero y a la vista correspondiente al rank del proceso, el siguiente paso es comenzar con la lectura de los datos. Para ello se realizará la llamada a la función MPI_File_read que irá leyendo una por una cada tripleta de char.

- Finalmente, después de cada lectura de las tripletas que contienen el fichero, se procederá al envío de los datos al proceso maestro. Para ello utilizaremos la función MPI_Send.

1.4. FLUJO DE DATOS MPI

Para el desarrollo de la práctica la mayoría de las primitivas MPI que se han utilizado son para la lectura del fichero datos.dat. Además de estas, también se han utilizado otras primitivas para el envío y recepción de datos como son MPI_Send y MPI_Recv. A continuación se explicará detalladamente para que sirve cada función y como se ha utilizado en nuestro programa:

- **MPI_Comm_spawn:** Esta función nos permite lanzar un número de procesos pasado como parámetro que ejecutarán un fichero binario indicado en el primer parámetro de la función. En esta práctica se lanzarán NUMPROCESOS trabajadores y todos ellos ejecutarán el fichero pract2.

  Esta primitiva tiene la siguiente sintaxis:

  ```
  int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs, MPI_Info info,
          int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
  ```

  donde command es el nombre del programa que se generará, argv son los argumentos para el command, maxprocs es el número máximo de procesos para iniciar, info es un conjunto de pares clave-valor que le indican al sistema en tiempo de ejecución dónde y cómo iniciar los procesos, root es el rank del proceso donde se examinan los argumentos anteriores, comm es el intracomunicador, intercomm es el intercomunicador entre el grupo original y el grupo generado, y errcodes es un array donde se almacenarán los códigos de error de los procesos creados.

- **MPI_File_open:** Esta función nos permite abrir un archivo de forma colectiva. En nuestro caso la hemos utilizado para que todos los procesos tengan acceso al fichero datos.dat.

  Esta primitiva tiene la siguiente sintaxis:

  ```
  int MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info,
     MPI_File *fh)
  ```

  donde comm es el intracomunicador desde el que se emite la llamada colectiva, filename es el nombre del archivo que queremos abrir, amode es el modo de acceso al fichero(permisos), info es la información que el usuario quiere proporcionar para optimizar la función, y fh es el manejador del fichero que queremos abrir.

- **MPI_File_set_view:** Esta función colectiva nos permite cambiar la vista del proceso de los datos en el archivo. En nuestro caso la utilizaremos para asignarle a cada proceso trabajador la porción de datos que debe leer del fichero.

  Esta primitiva tiene la siguiente sintaxis:

  ```
  int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype
  filetype, const char *datarep, MPI_Info info)
  ```

  donde fh es el manejador del fichero que queremos leer, disp es el desplazamiento desde el inicio del fichero, etype es el tipo de dato que contiene el fichero, filetype es el tipo de fichero, datarep es la forma de representación de los datos e info es la información que se quiera detallar de la operación.

- **MPI_File_read**: Esta función nos permite leer los datos de un fichero. En nuestro caso la utilizaremos para obtener los valores R,G,B de cada punto de la imagen.

  Esta primitiva tiene la siguiente sintaxis:

  ```
  int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  ```

  donde fh es el manejador del fichero, buf es el buffer donde almacenaremos los datos leidos, count es el número de elementos en el buffer, datatype es el tipo de datos que se van a leer, y status es el estado de la operación.

- **MPI_File_close:** Esta función permite cerrar un fichero abierto previamente.

  Esta primitiva tiene la siguiente sintaxis:

  ```
  int MPI_File_close(MPI_File *file);
  ```

  donde file es el manejador del fichero que queremos cerrar.

Por último tenemos las primitivas de transferencia de datos que utilizaremos para el envío y recepción de los datos leídos del fichero. Estas primitivas son MPI_Send y MPI_Recv:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

1.5. 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 IMAGEN "src/foto.dat"
#define TAMANIO 400

/*Variables Globales */

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, 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);

}

//Método para recibir y dibujar el punto
void imprimirPunto(MPI_Comm commPadre){
 int punto[5];
 for(unsigned i=0; i<TAMANIO*TAMANIO; i++){
   //recibimos el punto
   MPI_Recv(&punto,5,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,commPadre,
        MPI_STATUS_IGNORE);
   //imprimimos el punto
   dibujaPunto(punto[0],punto[1],punto[2],punto[3],punto[4]);
 }
}
```

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

 int rank,size;
 MPI_Comm commPadre;
 int tag;
 MPI_Status status;

 int buf[5]; //buffer del punto

 //codigos de error de los procesos trabajadores
 int codeError[NUMPROCESOS];


 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();

        /* Codigo del maestro */

  //Creacion de los procesos trabajadores

MPI_Comm_spawn("pract2",MPI_ARGV_NULL,NUMPROCESOS,MPI_INFO_NULL,0,MPI_
COMM_WORLD,&commPadre,codeError);

        //recibimos e imprimimos los puntos
  imprimirPunto(commPadre);
  printf("Imagen renderizada. Se mostrará durante 10 segundos\n");
  sleep(10);

 }else {

  /* Codigo de todos los trabajadores */
  int filasPorTrabajador = TAMANIO/NUMPROCESOS;
  int bloquePorTrabajador = filasPorTrabajador * TAMANIO * 3 * sizeof(unsigned char);

  //definimos la fila de inicio y final del rank
  int inicio = rank * filasPorTrabajador;
  int final = inicio + filasPorTrabajador;

  /* El archivo sobre el que debemos trabajar es foto.dat */

  //manejador del archivo foto.dat
  MPI_File fp;
  //Abrimos el archivo en modo solo lectura
  MPI_File_open(MPI_COMM_WORLD,IMAGEN,MPI_MODE_RDONLY,
       MPI_INFO_NULL, &fp);
```

```
//Le asinamos al rank su parte correspondiente del archivo
   MPI_File_set_view(fp,rank*bloquePorTrabajador,MPI_UNSIGNED_CHAR,
MPI_UNSIGNED_CHAR,"native",MPI_INFO_NULL);

   if(rank==NUMPROCESOS-1) final=TAMANIO;

   unsigned char color[3];

   //comenzamos la lectura de la vista asignada al rank
   for(unsigned i=inicio;i<final;i++){
     for(unsigned j=0; j<TAMANIO;j++){

       MPI_File_read(fp,color,3,MPI_UNSIGNED_CHAR,&status);

       //Creamos el punto
       //intercambiamos las posiciones de i y j para que la imagen no salga girada
       buf[0]=j;
       buf[1]=i;
       buf[2]=(int)color[0];
       buf[3]=(int)color[1];
       buf[4]=(int)color[2];

       //Enviamos el punto al proceso maestro
       MPI_Send(&buf,5,MPI_INT,0,1,commPadre);

     }
   }

   //Cerramos el archivo foto.dat
   MPI_File_close(&fp);

 }

 MPI_Finalize();

}
```

1.6. INSTRUCCIONES DE COMPILACIÓN Y EJECUCIÓN

Para la compilación y ejecución del programa se ha utilizado un archivo Makefile que hace más sencillo y rápido este proceso. Podemos ejecutar el programa directamente mediante el comando make all, o bien ir compilando y ejecutando paso a paso.

1. Compilar el programa pract2.c: make compilar
2. Ejecutar el programa: make run

Si queremos eliminar los archivos generados podemos utilizar el comando make clean

2. CONCLUSIONES

Esta práctica me ha parecido muy interesante y productiva ya que me ha permitido adquirir conocimientos como la lectura de ficheros de forma paralela en MPI y el renderizado de dibujos e imágenes en un servidor gráfico como X11.

3. REFERENCIAS

Material de ayuda para las primitivas MPI:

https://www.open-mpi.org/doc/v4.1/

https://docs.microsoft.com/es-es/message-passing-interface/microsoft-mpi