

Sistema distribuido de Renderizado de Gráficos

PRACTICA II. MPI2.

Rosa María Sacedón Ortega – rosamaria.sacedon@alu.ulcm.es
DISEÑO DE INFRAESTRUCTURA DE RED | ESCUELA SUPERIOR DE
INFORMÁTICA (UCLM)

MPI2

TABLA DE CONTENIDOS

1. Enunciado del problema
2. Planteamiento de la solución
3. Diseño del problema
4. Explicación del flujo de datos
5. Compilación y ejecución del programa
6. Conclusiones

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 encargará de ir enviando los píxeles 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

Mediante un programa en C# que contiene los métodos necesarios para la creación de una ventana gráfica usando el servidor X11, la imagen renderizada, el método que dibuja el gráfico y el fichero *foto.dat*.

El usuario ejecutará el programa solo creando un único proceso que será el proceso principal. Este después creará una cantidad de procesos trabajadores que vendrá definido de la siguiente forma (*ilustración 1*):

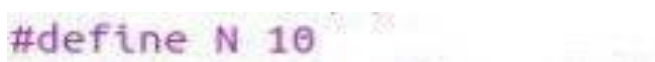


Ilustración 1

En el código podemos diferenciar dos tipos de procesos: maestro o *rank 0* y el de los trabajadores o *rank restantes*, los cuales tendrán unas funciones específicas. Si nos encontramos en el caso del proceso maestro, éste se encargará de inicializar la ventana de gráficos, lanzará los procesos trabajadores que leerá el fichero y, por último, recibirá los datos que los trabajadores han enviado y asignar los píxeles correspondientes a los datos en la ventana de gráficos. Sin embargo, si nos encontramos con los procesos trabajadores, se encargará de abrir, leer y procesar los datos del fichero *datos.dat*. También son los encargados de aplicar a la imagen los diferentes filtros que se hayan implementado, y para finalizar, enviaran la imagen con el filtro ya aplicado que hayamos asignado al proceso maestro.

3. DISEÑO DEL PROBLEMA

El proceso principal creará tantos procesos hijos como se haya indicado y esperará a que estos manden todos los puntos de la imagen que les haya tocado procesar. Los procesos trabajadores primero calcularán sobre cuántas filas de la imagen trabajarán. Una vez calculadas el número de filas, toca leer el fichero foto.dat píxel a píxel (tripleta de tres unsigned char que representan R, G, B de los colores primarios rojo, verde y azul respectivamente) y aplicará un filtro sobre estos valores para enviarlo al proceso principal. Cada proceso trabajador tendrá una vista sobre el fichero foto.dat de tal forma que se posicionará en la primera posición que le toque leer el fichero.

La espera de puntos será realizada por el proceso maestro, mientras que el reparto de la imagen, la apertura del fichero, la lectura píxel del fichero foto.dat y la aplicación de los filtros se realizará por parte de los procesos trabajadores.

```
/* Pract2 RAP 09/10 Javier Ayllon*/

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

#define NIL (0)
#define N 10
#define IMAGEN "foto.dat"
#define TAM 400

/*Variables Globales */

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

Ilustración 2

Podemos observar en la *ilustración 2*, como el proceso principal creará hasta 10 procesos trabajadores que son los que están definidos en: <#define N 10>

Además, se crean dos macros para poder sustituir la cadena de tokens correspondientes a cada aparición del fichero foto.dat y el tamaño de la imagen (400).

```
<#define IMAGEN "foto.dat"> <#define TAM 400>
```

En la *ilustración 3* se observan las tres funciones auxiliares que vamos a utilizar: max(int num1, int num2), initX() y dibujaPunto(int x,int y, int r, int g, int b). Cada una de estas funciones tiene un propósito específico que se utiliza a lo largo del programa.

MPI2

```
int max(int num1, int num2) { /*funcion para poder resolver el filtro cianotipia*/
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result; //Ejercicios aprenderaprogramar.com
}

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);
}
```

Ilustración 3

La primera que nos encontramos es `max(int num1, int num2)` sacará el valor máximo de los píxeles. Gracias a esta función, se puede calcular, más adelante, el filtro cianotipia.

Las funciones `initX()` y `dibujaPunto(int x,int y, int r, int g, int b)` han sido proporcionadas por el profesor.

En la *ilustración 4*, se definen las variables que se van a necesitar a lo largo del desarrollo del programa principal: *rank*, *size*, *commPadre*, *intercomm*, *status*, *tag*, *buf[5]*, *errCodes*, *pixel*, *numeroFilas*, *tamBloque*, *inicioLectura* y *finLectura*.

`numeroFilas = TAM / N`

`tamBloque = numeroFilas * TAM * 3 * sizeof(unsigned char)`

`inicioLectura = rank * numeroFilas`

`finLectura = inicioLectura + numeroFilas`

Se asignan las variables para el rank, la variable del size y la variable del comm:

`MPI_Init(&argc, &argv);`

`MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

`MPI_Comm_size(MPI_COMM_WORLD, &size);`

`MPI_Comm_get_parent(&commPadre);`

MPI2

Como se menciona anteriormente, el proceso principal va a crear tantos procesos trabajadores como se definen al inicio del código (`#define N 10`) y para ello se hace uso de la primitiva **MPI_Comm_spawn**.

Después de crear los procesos trabajadores, el proceso principal permanecerá en espera hasta que le envíen los pixeles y a medida que los recibe llama a la función **dibujarpuntos** para pintarlos en pantalla. El `sleep(10)` sirve para que una vez que esté completa la imagen, no se cierre inmediatamente.

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

    int rank,size;
    MPI_Comm commPadre, intercomm;
    MPI_Status status;
    int tag;
    int buf[5];
    int errCodes[N];
    unsigned char pixel[3];
    int numeroFilas = TAM / N;
    int tamBloque = numeroFilas * TAM * 3 * sizeof(unsigned char);
    int inicioLectura = rank * numeroFilas;
    int finLectura = inicioLectura + numeroFilas;

    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 */
        MPI_Comm_spawn("pract2", &argv[1], N, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &intercomm, errCodes);

        /*En algun momento dibujamos puntos en la ventana algo como
        dibujaPunto(x,y,r,g,b); */
        for(int i = 0; i < TAM*TAM; i++){
            MPI_Recv(&buf, 5, MPI_INT, MPI_ANY_SOURCE, 0, intercomm, &status);
            dibujaPunto(buf[0], buf[1], buf[2], buf[3], buf[4]);
        }
        sleep(10);
    }
}
```

Ilustración 4

MPI2

```

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

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

    if(rank == N-1){
        finLectura = TAM - floor(TAM/N)*(N-1); /*calcula líneas del ultimo profeso*/
    } else {
        finLectura = floor(TAM/N); /*reparto líneas entre otros*/
    }

    for(int y = 0; y < finLectura; y++){ /*recorremos el eje y*/
        for(int x = 0; x < TAM; x++){ /*recorremos el eje x*/

            MPI_File_read(imagen, pixel, 3, MPI_UNSIGNED_CHAR, &status);
            buf[0] = x;
            buf[1] = y + (rank * (ceil(TAM/N)));
            switch(*argv[1]){
                case 'B': /*blue*/
                    buf[2] = 0;
                    buf[3] = 0;
                    buf[4] = (int)pixel[2];
                    break;
                case 'G': /*green*/
                    buf[2] = 0;
                    buf[3] = (int)pixel[1];
                    buf[4] = 0;
                    break;
                case 'R': /*red*/
                    buf[2] = (int)pixel[0];
                    buf[3] = 0;
                    buf[4] = 0;
                    break;
            }
        }
    }
}

```

Ilustración 5

Gracias a **MPI_File**, **MPI_File_open** y **MPI_File_set_view** podemos hacer que cada proceso abra una vista de la imagen correspondiente a la primera posición que les toca leer del fichero. (*ilustración 5*)

En la *ilustración 5*, observamos que con los *if-else* se calculan las líneas que se le van a asignar al último proceso y, en el *else*, el reparto de líneas entre los demás procesos. Gracias a la anidación de los bucles *for* que nos encontramos a continuación, se pretende recorrer la imagen tanto en su eje x como en su eje y para poder ir asignando a cada píxel un color. Con **MPI_File_read** leemos el fichero y le pasamos los argumentos necesarios.

A continuación, iniciamos el *switch* con el que vamos a empezar a aplicar los diferentes filtros creados a la foto que, sobre los valores R, G, B, se aplica el filtro y se manda al proceso 0. Estos filtros se identificarán a través de una letra que tendremos que insertar a la hora de la ejecución para saber qué filtro se le aplica. Si no introducimos la letra en la ejecución, nos dará error ya que faltaría un argumento para la correcta ejecución.

Los tres primeros *case* del *switch* corresponden a los colores primarios rojo, verde y azul cuyos identificadores corresponden a sus iniciales anglosajonas B = azul (blue), G = verde (green) y R = rojo (red). Lo principal en estos tres *case* es activar el píxel correspondiente para dar color a la imagen, por ejemplo, para que se active el píxel de color azul, debemos activar el `buf[4]`, para el verde se activa el `buf[3]` y para el rojo, el `buf[2]`.

El *switch case* continua en la *ilustración 6*, en la que se realizan los filtros correspondientes al blanco y negro, sepia y negativo.


```
case 'W': /*black&white*/
    buf[2] = ((int)pixel[0] + (int)pixel[1] + (int)pixel[2]) / 3;
    buf[3] = ((int)pixel[0] + (int)pixel[1] + (int)pixel[2]) / 3;
    buf[4] = ((int)pixel[0] + (int)pixel[1] + (int)pixel[2]) / 3;
    break;
case 'S': /*sepia*/
    buf[2] = ((int)pixel[0] * 0.393) + ((int)pixel[1] * 0.769) + ((int)pixel[2] * 0.189);
    buf[3] = ((int)pixel[0] * 0.349) + ((int)pixel[1] * 0.686) + ((int)pixel[2] * 0.168);
    buf[4] = ((int)pixel[0] * 0.272) + ((int)pixel[1] * 0.534) + ((int)pixel[2] * 0.131);
    break;
case 'N': /*negative*/
    buf[2] = 255-(int)pixel[0];
    buf[3] = 255-(int)pixel[1];
    buf[4] = 255-(int)pixel[2];
    break;
```

Ilustración 6

A la hora de aplicar el filtro blanco y negro lo que hacemos es una media de los 3 colores primarios. En el filtro sepia se le van asignando diferentes valores a los píxeles correspondientes al R, G, B para poder conseguir la tonalidad correspondiente al color sepia. Finalmente, el negativo simplemente trata de restarle 255 a los tres colores primarios para invertir los colores.

A continuación, en la *ilustración 7*, se ha intentado implementar el filtro *cianotipia* a la imagen ya que resulta curioso y bonito a simple vista. Este filtro es una mezcla entre el negativo y el color cian. Y, dependiendo de la superficie en la que se realice, se va a plasmar el color de dicha superficie. Es decir, si la superficie es de color blanco, en la imagen aparecerán tanto el cian como el blanco; para una superficie oscura (generalmente negra), deberán aparecer los colores cian y negro; y si la superficie es roja, deberán de aparecer los colores cian y rojo. Este último efecto de la *cianotipia* me salió por casualidad al intentar sacar el filtro para el color blanco.

Como dato curioso, la opción que más costó conseguir fue la superficie blanca, ya que, si se tocaba demasiado los colores, podrían empezar a coger una tonalidad más grisácea.

```

    case 'C': /*cianotipia sobre papel blanco*/
        buf[2] = 0;
        buf[3] = (255-(int)pixel[0] + 255-(int)pixel[1] + 255-(int)pixel[2]) / 3;
        buf[4] = (255-(int)pixel[0] + 255-(int)pixel[1] + 255-(int)pixel[2]) / 3;

        if(buf[2] <= 150 && buf[3] <= 150 && buf[4] <= 150){
            /*si está por debajo de 150 son colores oscuros, de sombra*/
            int temporal = max(buf[2], buf[3]);
            temporal = max(temporal, buf[4]);
            buf[2] = 255-temporal;
            buf[3] = 255-temporal;
            buf[4] = 255-temporal;
        }
        break;
    case 'E': /*cianotipia sobre papel negro*/
        buf[2] = 0;
        buf[3] = (255-(int)pixel[0] + 255-(int)pixel[1] + 255-(int)pixel[2]) / 3;
        buf[4] = (255-(int)pixel[0] + 255-(int)pixel[1] + 255-(int)pixel[2]) / 3;
        break;
    case 'O': /*cianotipia sobre papel rojo*/
        buf[2] = ((int)pixel[0] + (int)pixel[1] + (int)pixel[2]) / 3;
        buf[3] = (255-(int)pixel[0] + 255-(int)pixel[1] + 255-(int)pixel[2]) / 3;
        buf[4] = (255-(int)pixel[0] + 255-(int)pixel[1] + 255-(int)pixel[2]) / 3;
        break;
    default:
        buf[2] = (int)pixel[0];
        buf[3] = (int)pixel[1];
        buf[4] = (int)pixel[2];
        break;
}
for(int k = 2; k <= 10; k++){ /*control para que no se salgan de los límites de los colores*/
    if(buf[k] > 255){
        buf[k] = 255;
    }else if (buf[k] < 0){
        buf[k] = 0;
    }
}
}

```

Ilustración 7

En el default del *switch*, lo que encontramos es la imagen original. Para poder controlar que los colores no se salgan de unos valores límites, se ha realizado el for que podemos observar al finalizar el *switch-case*.

Para finalizar, en la *ilustración 8*, se muestra con la primitiva **MPI_Bsend** la devolución al proceso padre de los píxeles con sus valores correspondientes al filtro que se ha decidido aplicar, cerramos el fichero de la imagen, finalizamos el programa *MPI* y devolvemos un *EXIT_SUCCESS* para decir que la ejecución se ha realizado con éxito.

```

        MPI_Bsend(&buf, 5, MPI_INT, 0, 0, commPadre);
    }
}
MPI_File_close(&imagen);
}
MPI_Finalize();
return EXIT_SUCCESS;
}

```

Ilustración 8

4. EXPLICACIÓN DEL FLUJO DE DATOS

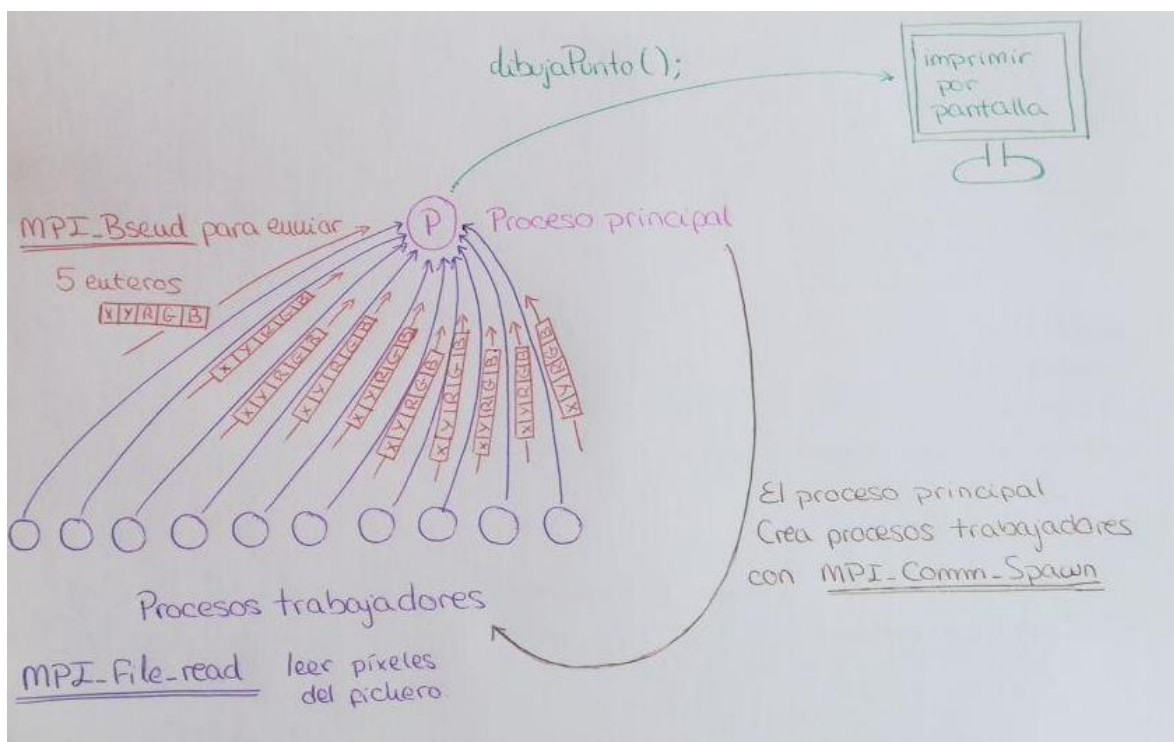


Ilustración 9

Como podemos observar en la ilustración 9, tenemos un proceso principal y 10 procesos trabajadores. El proceso principal es el encargado de crear a través de la primitiva **MPI_Comm_spawn** los 10 procesos trabajadores definidos en el `#define N 10`. Los procesos trabajadores son los encargados de tratar la imagen y, una vez que la tengan dibujada, mandarán un mensaje al proceso principal con 5 enteros que serán el eje X, el eje Y y los colores primarios R, G, B.

5. COMPILACIÓN Y EJECUCIÓN DEL PROGRAMA

Se facilita un archivo *makefile* en el que viene toda la información necesaria para la compilación y la ejecución del programa.

Compilar de manera automática haciendo uso del *makefile*:

```
< make all>
```

Para poder compilar el programa de forma manual usaremos el comando:

```
<mpicc pract2.c -o pract2 -lX11>
```

MPI2

A la hora de la ejecución manual bastará con poner:

```
<mpirun -np 1 ./pract2 <letra_identificatoria_filtro>
```

Las letras con las que se identifican cada filtro son:

FILTRO	LETRA IDENTIFICADORA
Azul	B
Verde	G
Rojo	R
Blanco & Negro	W
Sepia	S
Negativo	N
Cianotipia sobre superficie blanca	C
Cianotipia sobre superficie negra	E
Cianotipia sobre superficie roja	O

Muestra del código correspondiente al archivo *makefile*:

```

Makefile
nodos := 16
run := mpirun
mcc := mpicc
cc := gcc

all:
    $(mcc) pract2.c -o pract2 -lx11

Blue:
    $(run) -np 1 ./pract2 B
Green:
    $(run) -np 1 ./pract2 G
Red:
    $(run) -np 1 ./pract2 R
BlackWhite:
    $(run) -np 1 ./pract2 W
Sepia:
    $(run) -np 1 ./pract2 S
Negative:
    $(run) -np 1 ./pract2 N
Cianotipia sobre blanco:
    $(run) -np 1 ./pract2 C
Cianotipia sobre negro:
    $(run) -np 1 ./pract2 E
Cianotipia sobre rojo:
    $(run) -np 1 ./pract2 O

run:
    $(run) -np 1 ./pract2 A

```

Ilustración 10

MPI2

Ejemplo de la compilación y ejecución del programa con la aplicación del filtro sepia.

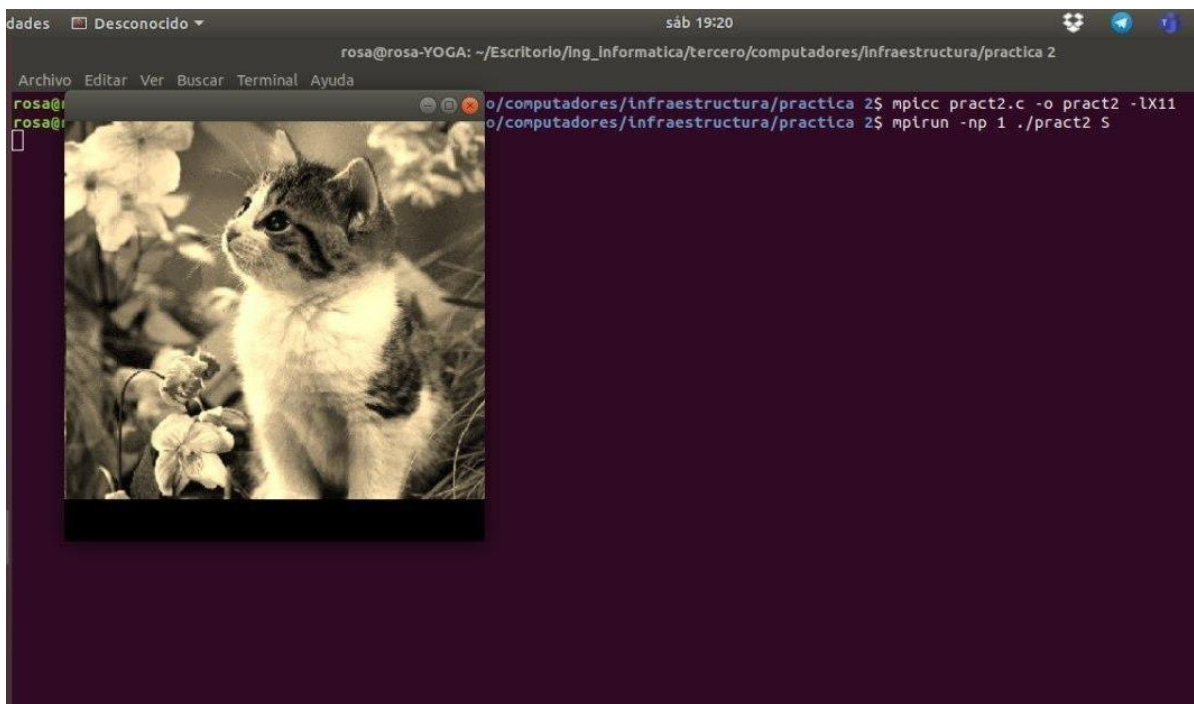


Ilustración 11

6. CONCLUSIONES

MPI facilita el trabajo de acceso de manera compartida a un mismo fichero para distribuir el acceso de los diferentes procesos a las funciones de acceso a disco y a gráficos. A pesar de esta división entre los procesos, la comunicación entre ellos no se pierde.