



# **PRACTICA 2 MPI: SISTEMA DISTRIBUIDO DE RENDERIZADO DE GRÁFICOS**



Sergio Jiménez del Coso

## Contenido

1.	Sistema Distribuido de Renderizado de Gráficos.....	3
1.1.	Enunciado del ejercicio. ....	3
1.2.	Planteamiento de la solución.....	3
1.3.	Diseño del programa.....	5
1.4.	Flujo de datos de la red.....	6
1.5.	Fuentes del programa. ....	8
1.6.	Instrucción para ejecutar el programa. ....	9
1.7.	Conclusiones. ....	9

## 1. Sistema Distribuido de Renderizado de Gráficos.

### 1.1. Enunciado del ejercicio.

El enunciado de esta práctica consiste en lanzar un proceso, el cual tendrá el acceso exclusivo a la pantalla de gráficos y no al disco. Éste a su vez lanzará un conjunto de N procesos, que será definido en el código y que tendrán acceso al disco y no a la pantalla de gráficos.

Los procesos que han sido lanzados se encargarán de leer de forma paralela los datos de un archivo (foto.dat). Después, enviarán cada uno de los píxeles al proceso que lanzó los procesos trabajadores, con acceso al disco, para que éste se encargue de representar cada uno de los píxeles en la pantalla de gráficos. La estructura de este archivo consta de 400 filas y de 400 columnas de puntos, formados por una tupla de *unsigned char* que corresponde a los valores de los colores rojo, verde y azul (RGB). Estos valores serán impresos por la función *dibujaPunto*.

### 1.2. Planteamiento de la solución.

Para realizar el planteamiento del ejercicio lo hemos dividido en dos partes, la cual se representará a través de la Figura 1.

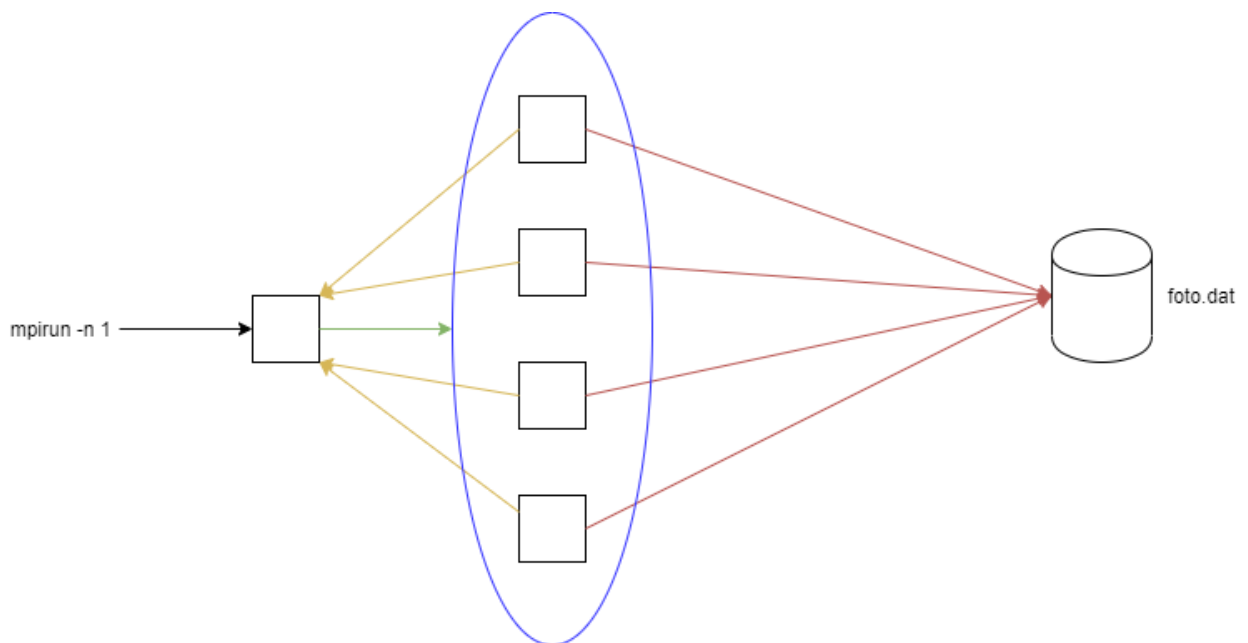


Figura 1. Esquema del planteamiento

Cuando se lanza el proceso (rank 0), éste es el encargado de lanzar los procesos trabajadores para la lectura del archivo en paralelo. Para ello el proceso padre lanzará N procesos que realizarán el manejo del fichero de forma paralela. Para ello, tenemos que asociarle un Communicator (MPI\_COMM\_WORLD) diferente para que los procesos lanzados realicen de forma independiente el acceso al fichero con el objetivo de que el proceso padre solo tenga que recibir los datos del nuevo Communicator e imprimirlos por pantalla. Este intracommunicator contiene el conjunto de procesos que han sido lanzados por el proceso padre y el Communicator que está asociado al padre es CommPadre. Finalmente, el proceso padre recibirá cada uno de los valores de los píxeles que fueron mandados por los procesos lanzados y los imprimirá por pantalla.

En la segunda parte, los procesos que fueron lanzados realizarán la gestión del fichero, primero la apertura de éste, después la división de cada parte del fichero de forma colectiva a través de las primitivas de MPI2. Una vez que cada uno de los procesos hayan asignado el área del fichero, se procederá a su lectura. Cada proceso leerá los pixeles del área asignada. Un píxel está formado por tres valores que indican los tres colores RGB (Red, Green, Blue), el cual se almacenará en un búfer para luego enviarlo al proceso padre, el cual se encargará de imprimir la imagen en pantalla. Hemos incluido un apartado que consiste en la inserción de filtros a la imagen. Cada filtro consiste en modificar los valores del vector *unsigned char* que los procesos trabajadores leen, permitiendo así modificar la tonalidad de cada valor del píxel. En mi caso he incluido estos filtros:

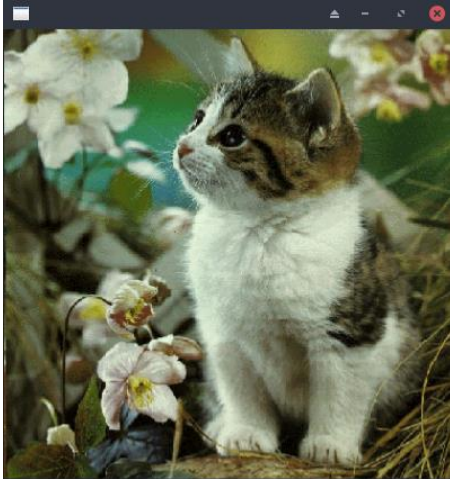


Figura 2. Filtro normal

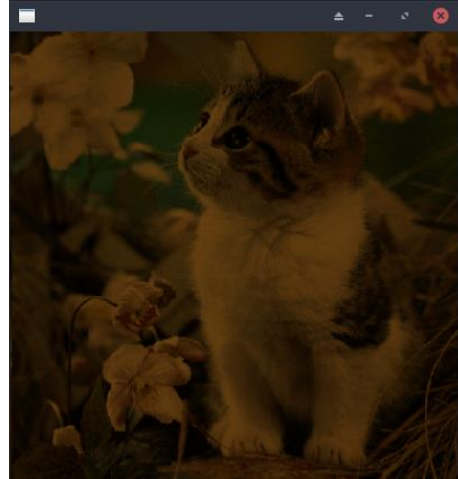


Figura 3. Filtro sepia

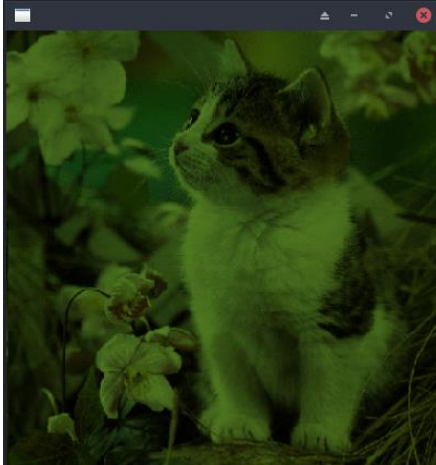


Figura 4. Filtro de Grises

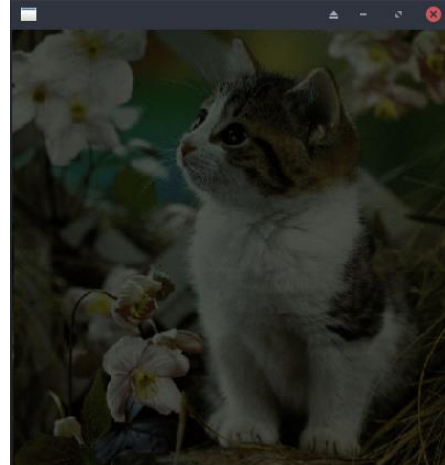


Figura 5. Filtro de bajo contraste

### 1.3. Diseño del programa.

Para este apartado vamos a diseñar el pseudocódigo del programa:

Para la tarea del nodo padre (commPadre){

    IniciarPantalla()

    For 0 hasta 400\*400:

        MPI\_Recv(buff, MPI\_ANY\_SOURCES, commPadre)

        DibujarPunto(buff[0], buff[1], buff[2], buff[3], buff[4])

    Sleep(1) /\*Dormimos el proceso para que se pueda visualizar la imagen\*/

}

Para cualquier nodo en COMM\_WORLD:

fila → longitud\_fichero/n\_nodos

inicio → fila\*rank

final → ((rank+1) \* fila)-1

MPI\_Offset area → fila\*longitud\_fichero\*3\*sizeof(unsigned char)

MPI\_Offset total\_area → area\*rank

MPI\_File\_open("foto.dat", MPI\_MODE\_RDONLY, fd)

MPI\_File\_set\_view(fd, total\_area, MPI\_UNSIGNED\_CHAR)

for inicio nodo hasta final:

    for 0 hasta longitud\_fichero:

        MPI\_File\_read(fd, colores, 3, MPI\_UNSIGNED\_CHAR)

        buff[0] → y

buff [1] → x

buff [2] → colores [0]

buff [3] → colores [1]

buff [4] → colores [2]

MPI\_Bsed(buff, 5, commPadre)

MPI\_File\_close(fd)

#### 1.4. Flujo de datos de la red.

En este apartado describiremos el flujo de datos, para ello hemos realizado un diagrama para que se entienda mucho mejor:

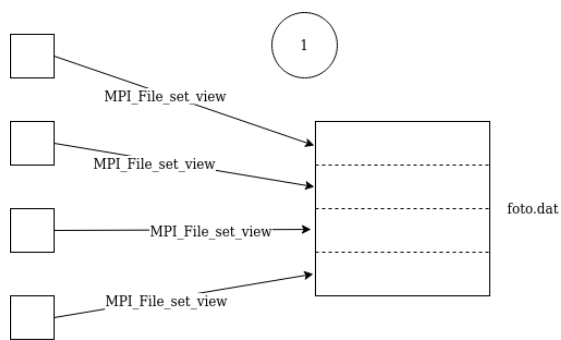


Figura 6. Secuencia 1

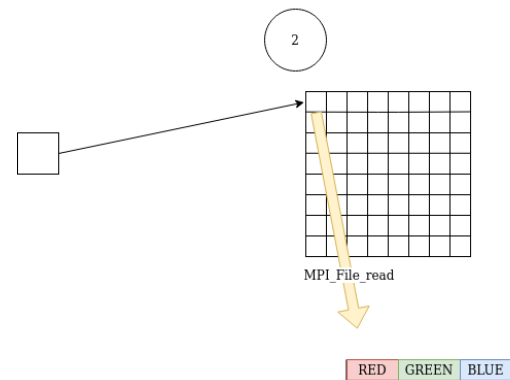


Figura 7. Secuencia 2

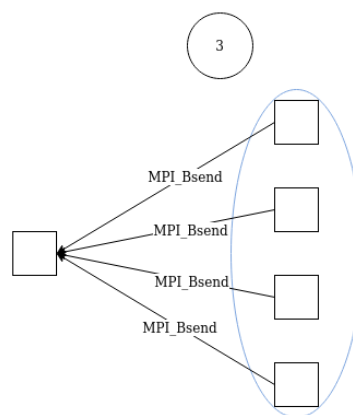


Figura 8. Secuencia 3

En la Figura.1 podemos observar el flujo de datos general de la práctica y para ello lo dividiremos en dos partes:

En la parte del proceso padre, podemos ver como sólo se ejecuta un solo proceso el cual se encargará a través de la primitiva `MPI_Comm_spawn`, que consiste en el lanzamiento de los procesos trabajadores para llevar a cabo la distribución y el manejo de datos del fichero. Finalmente, éste recibe los valores de cada uno de los píxeles, a través de la primitiva `MPI_Recv`, para imprimirlos por pantalla con la ayuda de la librería *X11/Xlib*.

En la Figura.6, 7 y 8, podemos visualizar cómo se comportan cada uno de los procesos que fueron lanzados. En la secuencia 1, los procesos tendrán que conocer el área que van a leer la información del fichero. Una vez que hayamos calculado el área designada de cada proceso, abrirá el fichero con la primitiva `MPI_File_open` ya que se trata de una operación colectiva, con el modo `MPI_MODE_RDONLY` (sólo lectura). Para asignar el área definida a cada proceso, utilizamos la función `MPI_File_set_view`, el cual tendremos que indicar el área total de cada proceso. Como el desplazamiento de cada uno de ellos es el mismo tendremos que multiplicarlo por el rank, esto es porque cada uno de los procesos tendrán su desplazamiento individual. En la secuencia 2, cada rank tendrá que leer cada uno de los píxeles, desde la línea inicial designada hasta la última de ésta. Y como ya hemos mencionado, cada píxel está formado por un array de *unsigned char* de tres valores que son los que irán leyendo cada proceso de forma individual a través de la primitiva `MPI_File_read`, el cual indicaremos que leerá tres valores que corresponde con los tres valores del array de *colores* [3]. Y finalmente, en la secuencia 3, cada rank irá almacenando los valores de la posición x e y del área y los valores de los colores en un búfer que se enviará al proceso padre, a través de la primitiva `MPI_Bsend`, indicando el Communicator que quieren mandar los procesos trabajadores, al CommPadre. Finalmente, el proceso padre recibirá los datos del Communicator que los procesos trabajadores habían enviado el contenido del búfer.

## 1.5. Fuentes del programa.

```

66 int main (int argc, char *argv[]) {
67
68     int rank,size;
69     MPI_Comm commPadre;
70     MPI_Status status;
71     int errcodes[N_NODOS];
72     int buff_punto[5]; /*Indicamos en 0->y, 1->x, 2->R, 3->G, 4->B*/
73
74     MPI_Init(&argc, &argv);
75     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
76     MPI_Comm_size(MPI_COMM_WORLD, &size);
77     MPI_Comm_get_parent( &commPadre );
78     if ((commPadre==MPI_COMM_NULL) && (rank==0)){
79
80         initX();
81         MPI_Comm_spawn("pract2", MPI_ARGV_NULL, N_NODOS, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &commPadre,errcodes); /*Lanzamiento de los nodos
82                                                                                                     trabajadores para el manejo del archivo*/
83
84         for (int i = 0;i<LONGITUD_FICHERO*LONGITUD_FICHERO;i++){
85             MPI_Recv(&buff_punto,5, MPI_INT, MPI_ANY_SOURCE,0,commPadre,&status); /*En este caso el proceso padre recibe los datos
86                                                                                                     del intercomunicador que lo hijos han enviado el mensaje, es decir, ¿a dónde mandan los hijos?*/
87             dibujaPunto(buff_punto[0],buff_punto[1],buff_punto[2],buff_punto[3],buff_punto[4]);
88         }
89     }
90     /*Codigo del maestro */
91
92     /*En algun momento dibujamos puntos en la ventana algo como
93     dibujaPunto(x,y,r,g,b); */
94     sleep(1);/*Esperamos un segundo para que se pueda ver la imagen*/
95
96
97 }else {
98     /*Codigo de todos los trabajadores */
99     /* El archivo sobre el que debemos trabajar es foto.dat */
100     MPI_File fd;
101     MPI_Status status;
102     int opcion = NORMAL; /*Filtro de la imágenes*/
103     int fila_fichero = LONGITUD_FICHERO / N_NODOS; /*Dividimos el tamaño del fichero para distintos ranks*/
104     int inicio_fichero = rank * fila_fichero; /*Fila de inicio que va a leer el proceso*/
105     int final_fichero = ((rank+1) * fila_fichero)-1; /*Fila final que va a leer cada proceso*/
106     if(rank == N_NODOS-1){
107         final_fichero = LONGITUD_FICHERO-1;
108     }
109
110     MPI_Offset area_rank = fila_fichero * LONGITUD_FICHERO * 3 * sizeof(unsigned char); /*Definimos el área que cada uno
111                                                                                                     de los procesos tienen que leer en el fichero el cual siempre es el mismo para cada uno de ellos*/
112     MPI_Offset total_area = area_rank *rank; /*Multiplicamos el desplazamiento por el rank de cada uno de los procesos para obtener el offset total*/
113     unsigned char colores[3]; /*Aquí tenemos el array de los colores*/
114     MPI_File_open(MPI_COMM_WORLD, "foto.dat", MPI_MODE_RDONLY, MPI_INFO_NULL, &fd); /*Abrimos el fichero de solo lectura*/
115
116
117     MPI_File_set_view(fd,total_area,MPI_UNSIGNED_CHAR,MPI_UNSIGNED_CHAR,"native",MPI_INFO_NULL); /*Dividimos la sección de cada uno de los hijos*/
118
119     for(int x = inicio_fichero; x<=final_fichero;x++){
120         for(int y = 0;y<LONGITUD_FICHERO;y++){
121
122             buff_punto[0] = y; /*Cambiaremos los valores para enderezar la imagen*/
123             buff_punto[1] = x;
124
125             MPI_File_read(fd, colores, 3 , MPI_UNSIGNED_CHAR, &status);
126
127
128             switch (opcion)
129             {
130                 case NORMAL: /*Filtro normal*/
131                     buff_punto[2] = (int)colores[0];
132                     buff_punto[3] = (int)colores[1];
133                     buff_punto[4] = (int)colores[2];
134                     break;
135
136                 case SEPIA: /*Filtro de sepia*/
137                     buff_punto[2] = (int)colores[0]*0.439;
138                     buff_punto[3] = (int)colores[1]*0.259;
139                     buff_punto[4] = (int)colores[2]*0.078;
140                     break;
141
142                 case LOW_CONTRAST: /*Filtro de contraste bajo*/
143                     buff_punto[2] = (int)colores[0]*0.333;
144                     buff_punto[3] = (int)colores[1]*0.333;
145                     buff_punto[4] = (int)colores[2]*0.333;
146                     break;
147                 case GREY: /*Filtro de escala de grises*/
148                     buff_punto[2] = (int)colores[0]*0.35;
149                     buff_punto[3] = (int)colores[1]*0.5;
150                     buff_punto[4] = (int)colores[2]*0.15;
151                     break;
152             }
153
154             MPI_Bsend(buff_punto, 5, MPI_INT , 0, 0, commPadre); /*Enviamos los datos recogidos del buffer a través del intercomunicador*/
155         }
156     }
157
158     MPI_File_close(&fd); /*Cerramos el fichero*/
159 }
160 MPI_Finalize();
161

```



### 1.6. Instrucción para ejecutar el programa.

Para la compilación del programa se ejecuta el comando:

**\$make**

Para la ejecución del programa:

**\$make run**

### 1.7. Conclusiones.

Para concluir con esta práctica, me ha gustado poder desarrollar un sistema de renderizado y conocer otras primitivas de MPI, ya que no se trata solo del envío y recibo de mensajes sino también poder acceder a un fichero ya que es una estructura de almacenamiento muy común. Lo que me ha costado ha sido definir el desplazamiento del área ya que no entendía claramente el concepto, pero una vez resuelta la duda, es bastante práctico. Dicho esto, he podido aprender cómo implementar una solución teniendo en cuenta las diferentes funcionalidades que realizan los procesos trabajadores y el proceso padre con el objetivo de obtener un mayor rendimiento y abstracción del mismo.