Práctica 2: Sistema distribuido renderizado de gráficos

Francisco Jesús Díaz Pellejero

Diseño de Infraestructura de Redes

6-5-2022

Enunciado del problema

Se nos pide elaborar un sistema distribuido para procesar imágenes mediante creación de procesos y directivas MPI. Primero se lanzará un primer proceso que creará otros nuevos. Los nuevos procesos leerán la imagen y se la enviarán al primero para que la muestre por pantalla.

Planteamiento de la solución

Será necesario el uso de directivas mpi de recepción por parte del proceso que accede a gráficos y directivos de envío por parte de los procesos que se encargan de leer del archivo.

Cada proceso que accede a disco leerá una sección del archivo, para poder conseguir el mayor nivel de paralelismo y concurrencia posible.

Diseño del programa

Proceso con acceso a gráficos

```
if ((commPadre == MPI_COMM_NULL) && (rank == 0)){
      initX();
      /* Codigo del maestro */
      MPI_Comm_spawn("exec/pract2", MPI_ARGV_NULL, N_HIJOS, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &commPadre, NULL);

      //MPI_Request request; /*For non-blocking call*/
      double begin = MPI_Wtime();

      for(int i = 0; i <  TOTAL_FILAS * TOTAL_COLUMNAS; i++){
            //NON BLOCKING CALL
            /*MPI_Irecv(punto, 5, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, commPadre, &request);
            if(MPI_Wait(&request, MPI_STATUS_IGNORE) != MPI_SUCCESS){
                  printf("RANK 0: Some error ocurred on the send or receive operation\n");
            }*/

            //BLOCKING CALL
            MPI_Recv(punto, 5, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, commPadre, &status);

            printf("Puntos recibidos %d --> x = %d, y = %d, r = %d, g = %d, b = %d\n", (i+1),
                        punto[0], punto[1], punto[2], punto[3], punto[4]);

            dibujaPunto(punto[0], punto[1], punto[2], punto[3], punto[4]);
      }
      printf("Imagen completada. Tiempo de computo: %.3f\n", MPI_Wtime()-begin);
      sleep(2);
}
```

Tras ejecutar las directivas de MPI_init, MPI_Comm_size, etc. Se pasará a una sección de código en la que se comprueba que estamos en un proceso de rango 0 y que además la variable *commPadre* no haya sido inicializada. Dicho proceso pasará a crear un grupo con el resto de los procesos (número definido por la constante *N_hijos*) mediante la directiva *MPI_Comm_spawn* en el que se pasará un intercomunicador *comm_Padre*. Dicho comunicador será usado por los nuevos procesos para el envío de datos

Tras esto, se ejecutará un bucle un número *n* de veces, siendo *n* el número de puntos del archivo o puntos a recibir para mostrar por pantalla.

En cada iteración del bucle se ejecutará la directiva *MPI_RECV* y se recibirá el punto con el código RGB y las coordenadas dentro de la imagen que se almacenará en el buffer *punto,* que es un array de 5 posiciones.

Dicho punto se pasará por parámetro al método *dibujaPunto* para mostrarlo por pantalla.

Tras esto se imprimirá el tiempo de ejecución del programa gracias al uso de directivas *MPI_Wtime* antes y después del bucle prinicpal.

También se dejará un tiempo de gracia para poder admirar la imagen del gato.

Proceso con acceso a disco

```
/* Codigo de todos los trabajadores */
/* El archivo sobre el que debemos trabajar es foto.dat */
int n_filas = TOTAL_FILAS/N_HIJOS;
int bloque = n_filas * TOTAL_COLUMNAS * 3 * sizeof(unsigned char);
int disp = rank * bloque;
int fila_inicial = n_filas * rank;
int fila_final =(rank == (N_HIJOS - 1)) ? TOTAL_FILAS : fila_inicial + n_filas;

printf("HIjo con rango %d creado, disp: %d, n_filas: %d, fila_final: %d\n", rank, disp, n_filas, fila_final);
MPI_File manejador;
MPI_File_open(MPI_COMM_WORLD, FILENAME, MPI_MODE_RDONLY, MPI_INFO_NULL, &manejador);
MPI_File_set_view(manejador, disp, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, "native", MPI_INFO_NULL);

unsigned char color[3];
//MPI_Request request; /*For non-blocking call*/
```

Tras la llamada a *MPI_Comm_Spawn* se ejecutarán los procesos que acceden a disco y se llamará también a primitivas como *MPI_rank* para averiguar su id. Además, se llamará al método *MPI_Comm_get_parent* para poder acceder al intercomunicador para poder realizar el envío de puntos al proceso que accede a gráficos.

Dentro de cada proceso realizará una división del número de filas a procesar. También se deducirá el tamaño del bloque en bytes a procesar. Dicho tamaño será equivalente al número de filas por proceso, entre el total de columnas por la tripleta de 3 datos de tamaño *unsigned char*. Dicha tripleta representa el color RGB.

La fila inicial de será el número de filas por el rango y la final la fila inicial más el número de filas a procesar, a excepción del último proceso, que puede ser que tenga más o menos filas a procesar, siendo su última fila la última del fichero.

Tras esto se llamará a la función *MPI_File_open* para que cada proceso pueda abrir el archivo, y después se llamará al método *MPI_File_set_view* para acceder únicamente a la parte del fichero que corresponde a cada proceso.

Dicha parte será definida por el número de bytes de desplazamiento desde el principio del fichero. Esto es equivalente al tamaño del bloque por el rango del proceso.

Tras esto se tendrá un archivo de lectura y una reserva de una parte del fichero para cada proceso. Por lo tanto, se podrá realizar la lectura de forma paralela.

```
unsigned char color[3];
//MPI_Request request; /*For non-blocking call*/
for(int i = fila_inicial; i < fila_final; i++){
      for(int j = 0; j < TOTAL_COLUMNAS; j++){
            punto[0] = j;
            punto[1] = i;
            MPI_File_read(manejador, color, 3, MPI_UNSIGNED_CHAR, &status);
            punto[2] = (int)color[0];
            punto[3] = (int)color[1];
            punto[4] = (int)color[2];

            //NON BLOCKING CALL
            /*MPI_Isend(punto, 5, MPI_INT, 0, 1, commPadre, &request);

            if(MPI_Wait(&request, MPI_STATUS_IGNORE) != MPI_SUCCESS){
            printf("RANK %d: Some error ocurred on the send or receive operation\n", rank);
            }*/

            /*BLOCKING CALL*/
            MPI_Send(punto, 5, MPI_INT, 0, 1, commPadre);

      }
}

MPI_File_close(&manejador);
```

Se necesitarán 2 bucles anidados en los que se recorran las filas y las columnas (definidas por las variables *i* y *j*). Se realizará una llamada a *MPI_File_read* y por cada llamada se aumentará el offset de la porción de fichero en 3, dicho valor siendo el tamaño de dato el punto (representado por la combinación rgb) que se quiere mostrar por pantalla. Dicho punto (o triada rgb) se almacenará en el buffer *color*.

Tras haber actualizado las coordenadas del punto en el buffer correspondiente, se actualizarán también la triada de colores. Este buffer de 5 enteros será el que habrá que enviar al proceso que accede a gráficos.

Para enviar los datos se utiliza la primitiva *MPI_Send*, enviando el buffer y el intercomunicador *commPadre*.

En la práctica se puede apreciar que se ha proporcionado directivas de *MPI_Isend* y *MPI_Irecv* para el envío de datos no bloqueante. Sin embargo, al no necesitarse computo extra entre cada llamada, la diferencia de tiempo de ejecución entre ambos tipos de llamadas es insignificante.

Compilación y Ejecución

Para crear los directorios *obj* y *exec* ejecutamos *make dirs:*

```
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls
foto.dat  include  Makefile  mpi_config  src
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ make dirs
mkdir -p obj/ exec/
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls
exec  foto.dat  include  Makefile  mpi_config  obj  src
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$
```

Tras la creación de directorios, se podrá compilar el programa mediante *make compile*.

```
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ make compile
mpicc -Iinclude/ -c -Wall src/pract2.c -o obj/pract2.o
mpicc -o exec/pract2 obj/pract2.o -lpthread -lrt -lm -lX11
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls exec/
pract2
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$
```

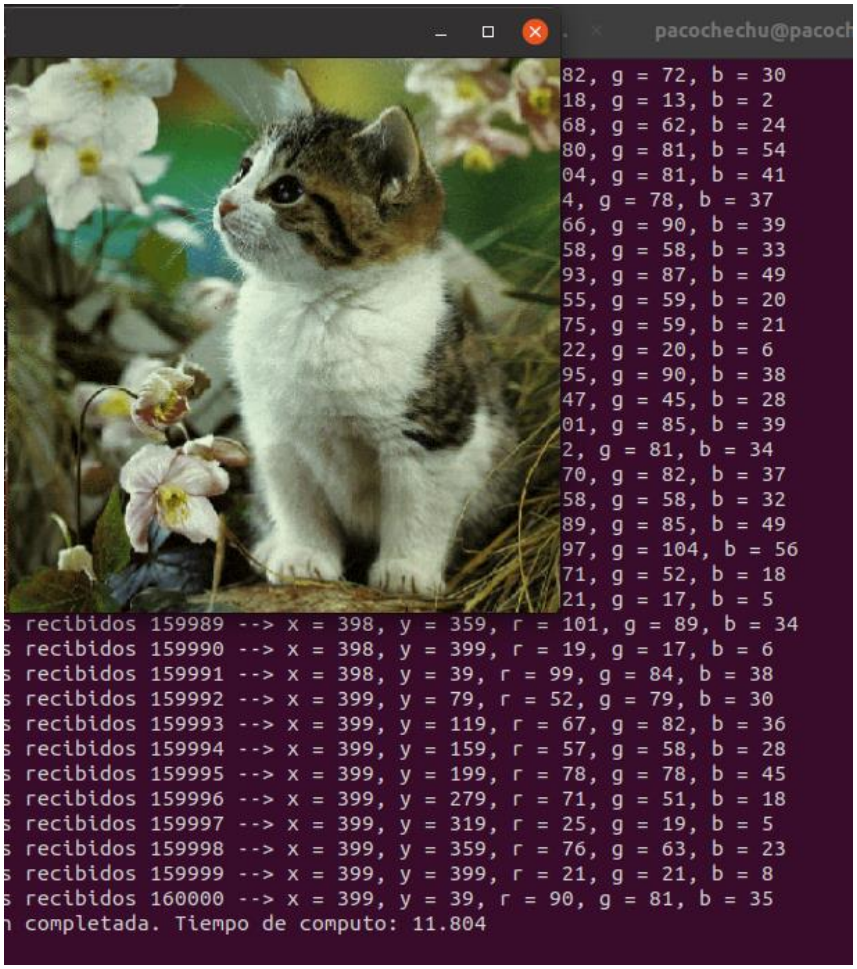
Si se quiere crear los directorios y compilar a la vez ejecutamos *make all:*

```
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ make all
mkdir -p obj/ exec/
mpicc -Iinclude/ -c -Wall src/pract2.c -o obj/pract2.o
mpicc -o exec/pract2 obj/pract2.o -lpthread -lrt -lm -lX11
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls
exec  foto.dat  include  Makefile  mpi_config  obj  src
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls exec/
pract2
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$
```

Si se quieren eliminar los directorios *obj* y *exec* junto con todo su contenido ejecutamos *make clean*:

```
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls
exec  foto.dat  include  Makefile  mpi_config  obj  src
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ make clean
rm -rf *~ core obj/ exec/ include/*~ src/*~
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$ ls
foto.dat  include  Makefile  mpi_config  src
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/lab/lab2$
```

Una vez creados los directorios y compilada la práctica, se podrá pasar a ejecutar la misma mediante *make test:*



Conclusión

Gracias a la compilación distribuida y al uso de directivas de MPI. Se ha conseguido un procesamiento de imágenes distribuida más eficiente frente a una programación monolítica que no emplea paralelismo ni ningún tipo de concurrencia.