Universidad de Castilla-La Mancha

Escuela Superior de Informática

Diseño de Infraestructuras de Red

Grado en Ingeniería Informática
2019 - 2020

Práctica 2: Renderizado Gráfico

Autor:
David Camuñas Sánchez

Fecha:
29 de abril de 2020

Índice

1. Enunciado

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.

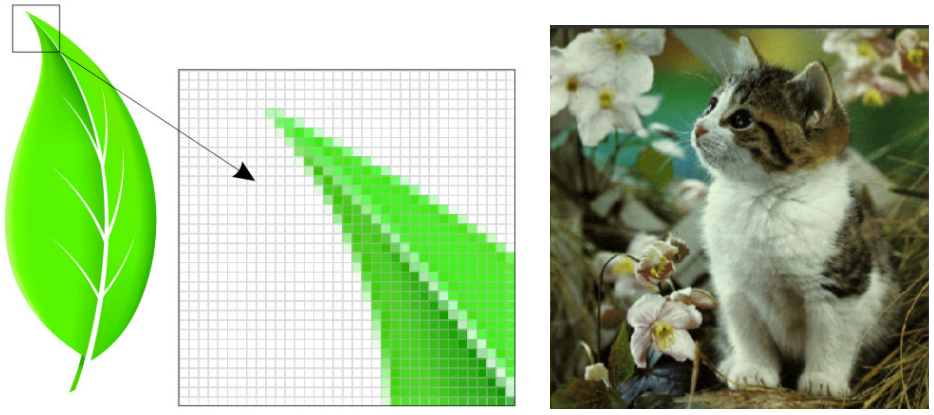
Usaremos la plantilla pract2.c para comenzar a desarrollar la práctica. En ella debemos completar el código que ejecuta el proceso con acceso a la ventana de gráficos (rank 0 inicial) y la de los procesos "trabajadores". Se proporciona el archivo foto.dat. La estructura interna de este archivo es 400 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*().

2. Introducción

El objetivo de esta práctica es la creación de un programa de rendrerizado gráfico, mediante el uso de las *funciones* que nos ofrece ***MPI***.

A la hora de analizar una imagen, en este caso foto.dat, se puede tratar como si fuese una matriz de X (filas) por Y (columnas) dimensiones. En este caso, se trata de una matriz cuadrada de *400 píxeles*, es decir, donde tanto el valor de las **filas** y **columnas** es de **400**. Por tanto el número total de pixeles que forman la matriz, es 400^2 *(*L^2*)* = **160000 píxeles**.
Estos valores estan indicados en el programa, en forma de constantes, en concreto, en el archivo *include/definitions.h*.

En la Figura 1, se puede observar la estructura de una imagen junto con la imagen a renderizar. Donde cada píxel está formado por una tripleta de tres parámetros de tipo "*unsigned char*" correspondiendo al valor R,G y B de cada uno de los *colores primarios*.



(a) Pixeles de una imagen (b) Imagen a renderizar

Figura 1: Representación de píxeles que forman una imagen

3. Planteamiento de la solución

El programa comienza con a creación de un solo proceso, este proceso es el proceso principal, el cual corresponde al proceso cuyo *RANK* es igual a 0. También conocido a lo largo del programa como el proceso ***maestro***.
Una vez lanzado el proceso *maestro*, este creará una ventana gráfica haciendo uso de la librería ***lX11*** (función *initX()*). Además también creará **N** procesos hijos, conocidos a lo largo del programa como ***esclavos o trabajadores*** (definidos con la constante creada eb tiempo de compilación *EMPLOYEES-NUMBER*).
Cabe mencionar, que la implementación que se ha llevado a cabo, a la hora de obtener la constante ***EMPLOYEES-NUMBER***, se hace en tiempo de compilación, es decir, a la hora de compilar el código mediante la herramienta *Makefile*, se pregunta al usuario sobre el numero de trabajadores que quiere generar, este valor se le asignará una vez introducido a esta constante.
Tras lanzar el ***maestro*** los N trabajadores, este se maltendra a la espera en todo momento, para recibir los pixeles de cada trabajador, los cuales debe de dibujar en la imagen. Mientras el maestro se mantiene en dicha espera, a cada ***trabajador***, se le asignara una zona de trabaja, esta zona pertenece a un rango de lineas del fichero, el cual será obtenido de dividir el número de filas del fichero entre los trabajadores generados (el resto de la operación, si es que lo hubiese, será asignado al último trabajador, es decir al *RANK* N-1).

3.1. Tipos de nodos

En este problema encontramos dos tipos de nodos: el ***rank o nodo 0*** y los demás ***nodos***.

- **Maestro (Rank 0)**: Este nodo corresponde al proceso principal del programa. Encargado de crear una ventana gráfica haciendo uso de la librería ***lX11*** (función *initX()*). Tras esto creará los procesos hijos, conocidos como trabajadores, donde el maestro recibirá cada píxel (punto) a pintar de la imagen (almacenada su información en un vector de enteros, en este caso el será el encargado de pintar dicho píxel dentro de la ventana gráfica.
- **Trabajadores (Hijos de Rank 0)**: Nada mas ser creado el proceso, calculará su "zona de trabajo", es decir, la linea de inicio y de final que le corresponda del fichero: *foto.dat*. Una vez asignadas las líneas cada proceso accederá de forma concurrente a su zona del fichero, y comenzará a obtener la información de cada píxel que forme la imagen. Tras darle el modo de filtrado deseado lo enviará al proceso maestro para que lo pinte en la pantalla.

Todo este proceso se describirá mas detalladamente en el siguiente apartado: ***Diseño de la solución***.

3.2. Representación del flujo de eventos

Como se puede observar en la Figura 2, el proceso ***maestro*** (principal), comenzará creando N procesos trabajadores (indicado en **Rojo**). Tras su creación, el maestro se mantendrá a la espera mientras los trabajadores realizan sus tareas en su zona de la imagen (procesamiento de los píxeles que les ha tocado).

Cuando un ***trabajador*** termina de procesar un pixel, junto todas sus comprobacines, y este esta listo para ser pintado, se lo envia al maestro (indicado en **Verde**). Como se puede observar el mensaje estará formado por la información del pixel a dibujar, la cual esta formada por los valores de los **tres colores primarios**, más su **posición** en la imagen. Una vez enviado, será recibido por el ***maestro*** y pintará el pixel en su posición.

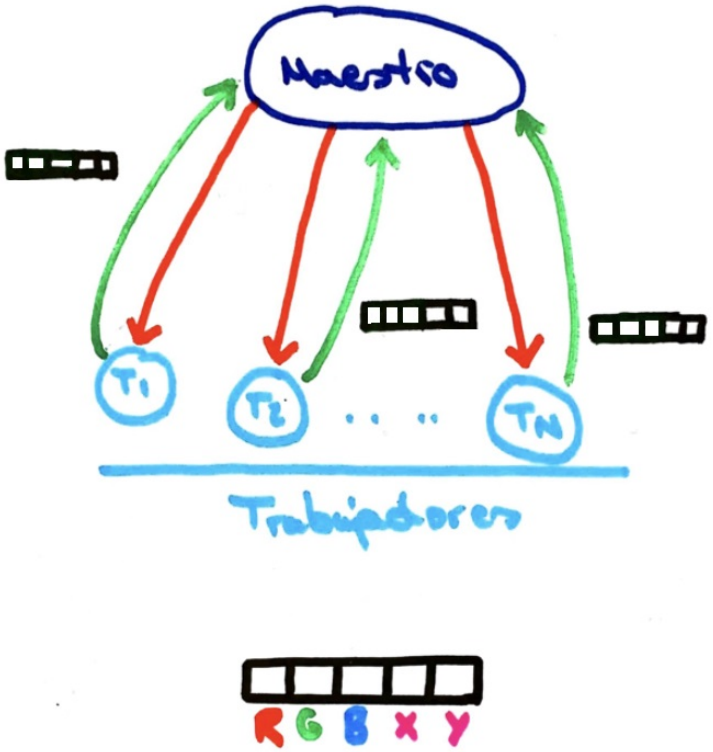


Figura 2: Representación del flujo de eventos.

4. Diseño de la solución

En este apartado se describirá las funciones que componen el programa, el cual, hace posible el funcionamiento del proceso anteriormente descrito.

4.1. Función principal del programa (*main*)

Esta es la función principal del programa, en la cual como se puede observar lo primero que se lleva a cabo es la definición de las variables a utilizar, como: *rank*, *size*, *error-codes* (vector donde se almacenarán si surge algún error a la hora de crear los trabajadores), *commPadre* (comunicador del proceso padre, utilizado a la hora de enviar y recibir información entre los procesos), status (indica el estado de la recepción de datos por parte del proceso padre), etc.

Como se puede observar en esta función principal (*main*), encontramos dos bifurcaciones, las cuales pertenece la primera al flujo de eventos del proceso **maestro** (Rank 0), y la otra bifurcación que pertenece a las acciones que debe de llevar a cabo los **trabajadores** (demás ranks).

Tras esto se declarán las primitivas de ***MPI*** como:

- *MPI-Init*() [1]: Para inicializar la estructura de comunicación de MPI entre los procesos.
- *MPI-Comm-Size*() [1]: Para obtener el tamaño de la comunicacion (número de proceso a ejecutar en este caso, ranks), etc.
- *MPI-Comm-rank*() [1]: Para establecer el identificador de cada proceso *rank*.
- *MPI-Comm-get-parent*() [2]: Se encarga de retornar el comunicador del proceso padre al proceso actual.

También en esta función se llevan a cabo las llamadas a las demás tipos de funciones que componen el programa

```
1 int main(int argc, char *argv[])
2 {
3
4       int rank, size;
5       MPI_Comm commPadre;
6       MPI_Status status;
7       int error_codes[EMPLOYEES_NUMBER];
8       double init = MPI_Wtime();
9
```

```
10          MPI_Init(&argc, &argv);
11          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12          MPI_Comm_size(MPI_COMM_WORLD, &size);
13          MPI_Comm_get_parent(&commPadre);
14
15          if ((commPadre == MPI_COMM_NULL) && (rank == MASTER_RANK))
16          {
17                /* Codigo del maestro */
18                print_title();
19                initX();
20                MPI_Comm_spawn(EXEC_PATH, argv, EMPLOYEES_NUMBER,
                     MPI_INFO_NULL, MASTER_RANK, MPI_COMM_WORLD, &
                     commPadre, error_codes);
21
22                printf("[MAESTRO] :: Dibujando imagen...\n");
23                receive_points(commPadre, &status);
24                print_final_info(init);
25          }
26
27          else
28          {
29                /* Codigo de todos los trabajadores */
30                int lines_per_employee, rest_lines, start_line,
                     end_line;
31                long row_bytes;
32                char mode;
33                MPI_File image;
34
35                calculate_file_lines(&lines_per_employee, &rest_lines,
                     &row_bytes);
36
37                assign_employee_lines(rank, &start_line, &end_line,
                     lines_per_employee, rest_lines);
38
39                image = open_file(rank, row_bytes);
40
41                /* Coger modo de filtro */
42                mode = argv[argc - 1][0];
43
44                parse_image(rank, start_line, end_line, mode, image,
                     commPadre);
45
46                MPI_File_close(&image);
47          }
48
49          MPI_Finalize();
50
51          return EXIT_SUCCESS;
52    }
```

4.2. Recepción de los píxeles (*receive-points*)

El encargado de ejecutar esta función (como bien hemos dicho anteriormente) es el proceso **maestro** (Rank 0). La cual, se encarga de invocar a la primitiva de MPI [2], ***MPI-Recv***() [2], para la recepción por medio de un buffer de tipo entero (point-to-paint), en el cual se almacenará los datos del mensaje enviado por un trabajador, este contiene la información necesaria para pintar ese píxel en la imagen, tras esto se pinta dicho píxel por pantalla con la función ***dibujarPunto()***.

```
1  void receive_pixels(MPI_Comm parent_comm, MPI_Status *status)
2  {
3        int pixel_to_paint[PIXEL_INFO_N];
4        int i;
5
6        for (i = 0; i < IMAGE_SIZE; i++)
7        {
8              MPI_Recv(&pixel_to_paint, PIXEL_INFO_N, MPI_INT,
                   MPI_ANY_SOURCE, TAG, parent_comm, status);
9              dibujaPunto(pixel_to_paint[ROW], pixel_to_paint[COLUMN
                   ], pixel_to_paint[R], pixel_to_paint[G],
                   pixel_to_paint[B]);
10       }
11 }
```

4.3. Calcular líneas del fichero (*calculate-file-lines*)

Esta función llevada a cabo por todo proceso de tipo trabajador (distinto de Rank 0), se encarga de calcular la zona de trabajo que le corresponde a cada trabajador (líneas que le corresponden). Se realiza como se puede observar por la division entre *IMAGE-SIDE* (lado de la imagen), que es el número de filas o líneas que tiene la imagen (400 en este caso), entre el *EMPLOYEES-NUMBER* (constante que representa el número de trabajadores). También calculamos el resto de la operación anterior, y el número de bytes de cada fila (*row-bytes*), esta variable almacenara el número de bytes o elementos por fila.

```
1  void calculate_file_lines(int *lines_per_employee, int *rest_lines,
       long *row_bytes)
2  {
3        *lines_per_employee = IMAGE_SIDE / EMPLOYEES_NUMBER;
4        *rest_lines = IMAGE_SIDE % EMPLOYEES_NUMBER;
5        *row_bytes = *lines_per_employee * IMAGE_SIDE * sizeof(
             unsigned char) * 3;
```

```
}
```

4.4. Asignar zona de trabajo (*assign-work-zone*)

El encargado de ejecutar esta función al igual que la anterior, y las posteriores es todo proceso de tipo trabajador (distinto a Rank 0). Esta función de asignar su zona de trabajo a cada proceso trabajador. Es decir, de asignarle una linea de inicio y otra de fin. Para ello cada **línea de comienzo** (*start-line*) de cada trabajador será: su *rank* x *lines-per-employee* (cantidad de líneas asignada a cada trabajador). Tras esto se calculará la linea de fin (*end-line*), que se obtendrá de calcular la misma operación anterior pero incrementando el rank en 1. En cambio, si el rank actual es el último, en este caso se le asignará también *rest-lines* (resto de las líneas si es que las hubiese).

```
void assign_work_zone(int rank, int *start_line, int *end_line, int
    lines_per_employee, int rest_lines)
{
      *start_line = rank * lines_per_employee;

      rank == (EMPLOYEES_NUMBER - 1) ? (*end_line = (rank + 1) *
         lines_per_employee + rest_lines) : (*end_line = (rank + 1)
          * lines_per_employee);
}
```

4.5. Apertura del fichero (*open-file*)

En esta función llevada a cabo por cada trabajador, se realiza la apertura del fichero de la imagen. Para ello como se puede observar se utiliza la función ***MPI-File-open()*** [2], y posteriormente se utiliza la función ***MPI-File-set-view()*** [2] la cual posiciona a cada rank en su zona de trabajo, y evitar así que interfieran entre ellos.
Finalmente devolvemos el descriptor del archivo, ya que sera necesario más adelante.

```
MPI_File open_file(int rank, long row_bytes)
{
      MPI_File image;
      MPI_File_open(MPI_COMM_WORLD, IMAGE_PATH, MPI_MODE_RDONLY,
         MPI_INFO_NULL, &image);
      MPI_File_set_view(image, rank * row_bytes, MPI_UNSIGNED_CHAR,
          MPI_UNSIGNED_CHAR, NATIVE_MOD, MPI_INFO_NULL);

      return image;
}
```

4.6. Leer un píxel de la imagen (*read-pixel*)

En esta función llevada a cabo por cada trabajador, donde leerá y almacena la información de cada píxel de la imagen (que pertenezca a su zona de trabajo), mediante la función ***MPI-File-read()*** [2]. Esta información se almacenara en un vector de tipo *unsigned char* denominado ***píxel***. Donde se almacenarán los valores de ese pixel correspondientes a cada color primario, es decir, **R, G, B** (Rojo, Verde y Azul). Tras esto, se llamará ma la función aplicar filtro (***apply-filter***()).

```
1  void read_pixel(int rank, int start_line, int end_line, char mode,
       MPI_File image, MPI_Comm parent_comm)
2  {
3         unsigned char pixel[PRIMARY_COLORS_N];
4         int i, j;
5
6         for (i = start_line; i < end_line; i++)
7         {
8               for (j = 0; j < IMAGE_SIDE; j++)
9               {
10                    MPI_File_read(image, pixel, PRIMARY_COLORS_N,
                          MPI_UNSIGNED_CHAR, MPI_STATUS_IGNORE);
11                    put_filter(j, i, pixel, mode, parent_comm);
12              }
13        }
14 }
```

4.7. Comprobación del píxel (*check-pixel*)

En esta función llevada a cabo por cada trabajador, se comprobará si el valor de los colores primarios que forman el píxel, se encuentra dentro de su rango, ya que el color se calcula entre el rango de 0 a 255.

```
1  void check_pixel(int *pixel_to_paint)
2  {
3         int i;
4
5         for (i = R; i <= B; i++)
6         {
7               if (pixel_to_paint[i] < 0){pixel_to_paint[i] = 0;}
8               if (pixel_to_paint[i] > 255){pixel_to_paint[i] = 255;}
9         }
10 }
```

4.8. Aplicar filtro (*apply-filter*)

En esta función llevada a cabo por cada trabajador, una vez que tiene la información del pixel leido (almacenada en ***pixel***), se asignará dicha información, junto con la posición en la que se encentra el pixel en la imagen (Fila, Columna), al vector de tipo entero ***point-to-paint***. Este será utilizado para almacenar la información del píxel durante su configuración, es decir, para aplicarle el filtro que haya deseado el usuario.

Los filtros de los cuales dispone el programa, como se puede observar, son:

- **Por defecto**: Color que tiene el píxel en la imagen original.
- **Sepia.**
- **Blanco y Negro.**
- **Negativo**: color complementario al que se encuentre el píxel.

Una vez, que se ha aplicado el filtro, se debera de comprobar si el filtro ha sido aplicado de forma correcta, mediante la función ***check-pixel***.
Finalmente, se mandará la información correspondiente al pixel (almacenda en ***point-to-paint***)al proceso maestro, mediante la función ***MPI-Send()*** [2], para que así el maestro pueda pintar el píxel por pantalla.

```
1  void apply_filter(int row, int column, unsigned char *pixel, char
       mode, MPI_Comm parent_comm)
2  {
3         int pixel_to_paint[PIXEL_INFO_N];
4
5         pixel_to_paint[ROW] = row;
6         pixel_to_paint[COLUMN] = column;
7
8         switch (mode)
9         {
10        case SEPIA:
11              pixel_to_paint[R] = (int)(pixel[R] * 0.393) + (int)(
                    pixel[G] * 0.769) + (int)(pixel[B] * 0.189);
12              pixel_to_paint[G] = (int)(pixel[R] * 0.349) + (int)(
                    pixel[G] * 0.686) + (int)(pixel[B] * 0.168);
13              pixel_to_paint[B] = (int)(pixel[R] * 0.272) + (int)(
                    pixel[G] * 0.534) + (int)(pixel[B] * 0.131);
14              break;
15
16        case BLACK_WHITE:
17              pixel_to_paint[R] = (int)(pixel[R] * 0.2986) + (int)(
                    pixel[G] * 0.587) + (int)(pixel[B] * 0.114);
```

```
18              pixel_to_paint[G] = (int)(pixel[R] * 0.2986) + (int)(
                    pixel[G] * 0.587) + (int)(pixel[B] * 0.114);
19              pixel_to_paint[B] = (int)(pixel[R] * 0.2986) + (int)(
                    pixel[G] * 0.587) + (int)(pixel[B] * 0.114);
20              break;
21
22        case NEGATIVE:
23              pixel_to_paint[R] = 255 - (int)pixel[R];
24              pixel_to_paint[G] = 255 - (int)pixel[G];
25              pixel_to_paint[B] = 255 - (int)pixel[B];
26              break;
27
28        default:
29              pixel_to_paint[R] = (int)pixel[R];
30              pixel_to_paint[G] = (int)pixel[G];
31              pixel_to_paint[B] = (int)pixel[B];
32              break;
33        }
34
35        check_pixel(pixel_to_paint);
36        MPI_Send(&pixel_to_paint, PIXEL_INFO_N, MPI_INT, MASTER_RANK,
              TAG, parent_comm);
37  }
```

Referencias

[1] *Funciones de MPI - Diseño de Infraestructuras de Redes.* Javier ayllón.

[2] *Open MPI Organization.* `https://www.open-mpi.org`.