PRÁCTICA OPTATIVA

DISEÑO DE INFRAESTRUCTURA DE RED

Gráficos distribuidos

Filtro detección de bordes

PABLO TOMÁS TOLEDANO GONZÁLEZ

pablotomas.toledano@alu.uclm.es

Contenido

1. Enunciado del problema

Utilizaremos las primitivas pertinentes MPI2 como acceso paralelo a disco y gestión de procesos dinámico al igual que en la práctica 2 con la diferencia que aquí se aplicara un filtro del tipo *Edge Detection*:

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 pixeles 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

- El nodo inicial se encargará de iniciar la ventana gráfica, posteriormente creará los procesos especificados y se mantendrá a la espera de recibir todos los pixeles de la imagen para dibujarlos en la ventana iniciada anteriormente.
- El resto de los procesos iniciados por el original se encargarán de leer el archivo foto.dat, en particular exclusivamente los pixeles de la región que se le asigne según su rank, aplicarán primero un filtro gaussiano de desenfocado, posteriormente realizara las convoluciones de la detección de bordes de Canny y lo enviara al padre para su posterior dibujo en pantalla.

3. Diseño de programa

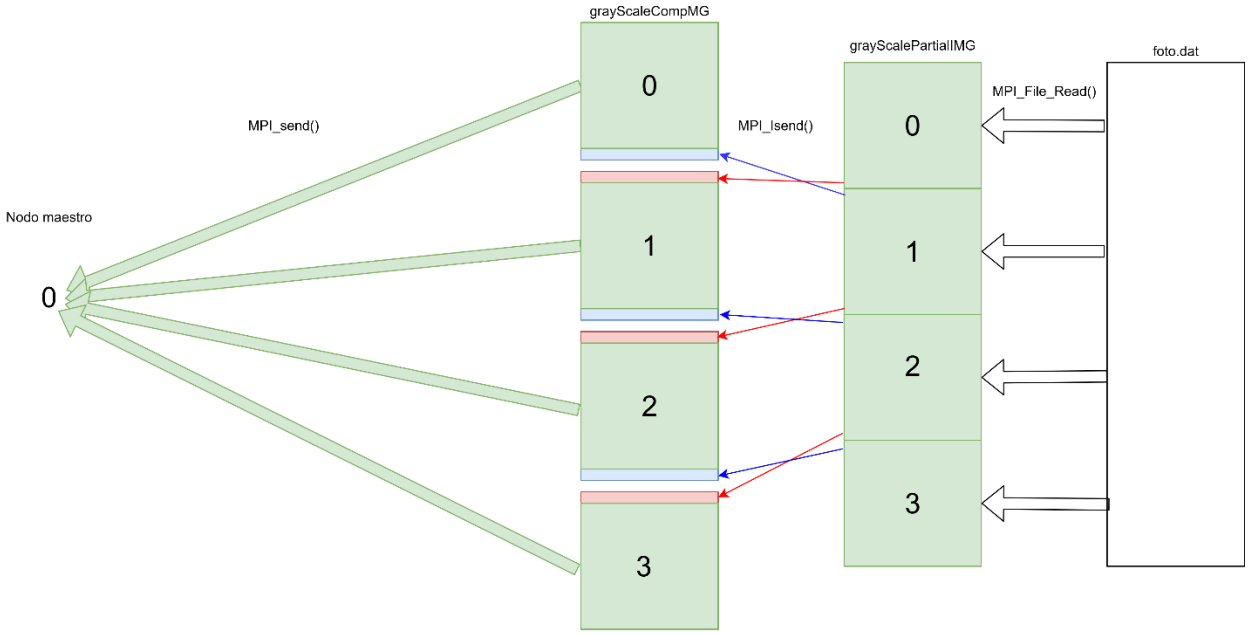
En primer lugar, solo existe un proceso "padre" el cual se encargará de crear la ventana donde luego se dibujarán los distintos pixeles. Después de crear la ventana el proceso padre creara los procesos que se encargaran de leer los pixeles y procesarlos. Las funciones usadas en el programa son las siguientes:

- **edgeDetection**: función ejecutada por los procesos encargados de leer la imagen y procesarla. En ella cada proceso abre el archivo foto.dat con MPI_File_open, se posiciona en su lugar del archivo correspondiente a su rank con MPI_File_set_view, lee la franja de la imagen necesaria para aplicar primero el filtro de desenfocado y posteriormente el de detección de bordes.
- **getPhotoPixels:** función ejecutada por el proceso padre en la cual se encarga de recibir todos lo

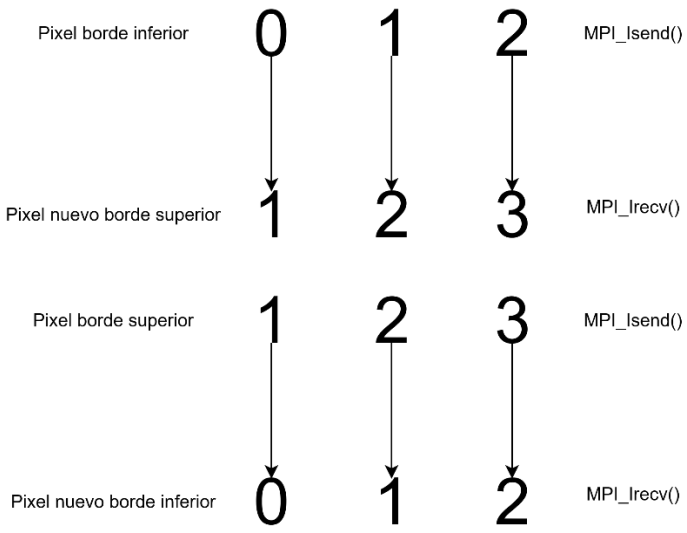
pixeles de la imagen y de dibujarlos en la ventana creada al inicio del programa.

- **getReadRows:** devuelve las filas que serán leídas por los distintos nodos.
- **getStarRow:** devuelve la fila por la que el nodo empezara a enviar.
- **Convolution:** realiza la convolución de una matriz in con un kernel y guarda el resultado en la matriz out.
- **Gaussian_filter:** crea un kernel de desenfocado dependiendo de la sigma establecida y aplica la convolución entre ese kernel y la matriz in.
- **sendResult:** envia al proceso padre el resultado de aplicar el filtro.
- **readFile:** guarda en una matriz el valor de los pixeles de la fotografía.
- **transferFilterPixels:** función en la que se envían y reciben los pixeles necesarios de otros procesos para aplicar el filtro.

4. Flujo de datos

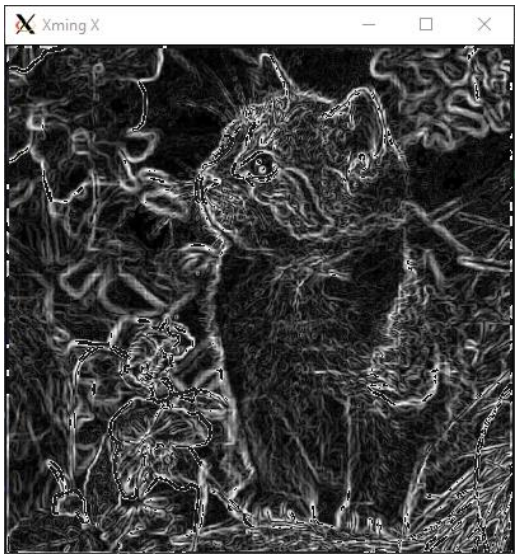


Cada uno de los procesos creados por el "maestro" leen de forma paralela su parte correspondiente del archivo foto.dat. Como para la aplicación de los distintos filtros es necesarios disponer de unas filas de pixeles extra, cada proceso enviara de manera no bloqueante su fila superior y/o inferior para que la guarde su proceso vecino correspondiente. Ejemplo con 4 procesos:
El proceso 0 necesita el borde superior del proceso 1 para sus cálculos del borde inferior, para ello el proceso 1 envía pixel por pixel todo su borde con Isend() y el proceso 0 los recibe con Irecv(). El ejemplo de flujo de datos sería:

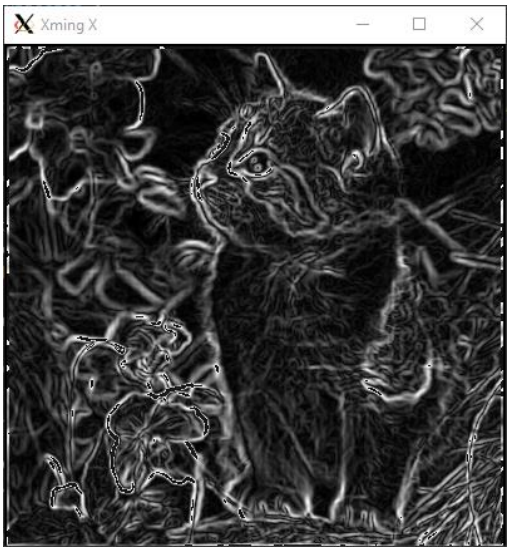


5. Resultados de la ejecución

El valor del filtro varía según el valor que le demos a la sigma del filtro de desenfocado gaussiano. Esto es así debido a que el filtro aplicado de detección de bordes es extremadamente sensible.



Sigma 0.5



Sigma 1

Sigma 2

6. Fuentes del programa

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

#define NIL (0)
#define PHOTO "foto.dat"
#define NPROCESS 4
#define PHOTOCOLUMNS 400
#define PHOTOROWS 400
#define MAX_BRIGHTNESS 255
//Para cambiar el threshold del edge detection cambiar SIGMA, probado con
0.5,1,2,3
#define SIGMA 1

/*Variables Globales */

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

```
/*Funciones auxiliares */

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,
                                    PHOTOCOLUMNS, PHOTOROWS, 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);

}

void getPhotoPixels(MPI_Comm commPadre){
	int bufferPixelData[3];
	MPI_Status status;
	int photosize = PHOTOCOLUMNS*PHOTOROWS;
	for (size_t i = 0; i < photosize; i++)
	{

MPI_Recv(&bufferPixelData,3,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,commPadre,&sta
tus);

dibujaPunto(bufferPixelData[0],bufferPixelData[1],bufferPixelData[2],bufferPi
xelData[2],bufferPixelData[2]);

	}


}
```

```
int getStartRow(int rank){
	int rowsperworker = PHOTOROWS/NPROCESS;
	int startrow = rank * rowsperworker;

	return startrow;
}

int getEndRow(int rank, int startrow){
	int rowsperworker = PHOTOROWS/NPROCESS;
	int endrow = startrow + rowsperworker ;
	if(rank == NPROCESS-1)
		endrow = PHOTOROWS;

	return endrow;
}



int getReadRows(int rank){
	int readrows = PHOTOROWS/NPROCESS;
	if(rank == 0 ){
		readrows += 16;
	}else{
		readrows += 48;
	}
	return readrows;
}

static void convolution(const int *in,
		int       *out,
		const float   *kernel,
		const int      nx,
		const int      ny,
		const int      kn,
		const int     normalize){
	const int khalf = kn / 2;
	float min = 0.5;
	float max = 254.5;
	float pixel = 0.0;
	size_t c = 0;
	int m, n, i, j;

	assert(kn % 2 == 1);
	assert(nx > kn && ny > kn);

	for (m = khalf; m < nx - khalf; m++) {
	for (n = khalf; n < ny - khalf; n++) {
		pixel = c = 0;

		for (j = -khalf; j <= khalf; j++)
		for (i = -khalf; i <= khalf; i++)
		pixel += in[(n - j) * nx + m - i] * kernel[c++];

		if (normalize == 1)
		pixel = MAX_BRIGHTNESS * (pixel - min) / (max - min);
```

```
            out[n * nx + m] = (int) pixel;
      }
      }
}

void gaussian_filter(const int *in,
                int       *out,
                const int      nx,
                const int      ny,
                const float    sigma){
      const int n = 2 * (int) (2 * sigma) + 3;
      const float mean = (float) floor(n / 2.0);
      float kernel[n * n];
      int i, j;
      size_t c = 0;

      for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
            kernel[c++] = exp(-0.5 * (pow((i - mean) / sigma, 2.0) + pow((j -
mean) / sigma, 2.0))) / (2 * M_PI * sigma * sigma);
      }

      convolution(in, out, kernel, nx, ny, n, 1);
}

void edgeDetection(int rank,MPI_Comm commPadre){
      MPI_File photo;
      MPI_Status status;
      int bufferPixelData[3];
      int red,green,blue;

      int startrow = getStartRow(rank);
      int endrow = getEndRow(rank,startrow);
      int readrows = getReadRows(rank);


      unsigned char *photoData =
malloc(3*readrows*PHOTOCOLUMNS*sizeof(unsigned char)); //falta multiplicar
los pixeles que voy a necesitar
      int *grayScaleIMG = calloc(readrows*PHOTOCOLUMNS, sizeof(int));
      int *blurIMG = calloc(readrows*PHOTOCOLUMNS, sizeof(int));
      int *edgeImgY = calloc(readrows*PHOTOCOLUMNS, sizeof(int));
      int *edgeImgX = calloc(readrows*PHOTOCOLUMNS, sizeof(int));
      MPI_Offset point =  ( ( ( rank* (PHOTOROWS/NPROCESS)-32 ))
*sizeof(unsigned char)*3*PHOTOCOLUMNS );
      if(rank == 0){
           point= 0 ;
      }


MPI_File_open(MPI_COMM_WORLD,PHOTO,MPI_MODE_RDONLY,MPI_INFO_NULL,&photo);
      MPI_File_set_view(photo, point, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR,
"native", MPI_INFO_NULL);

MPI_File_read(photo,photoData,3*readrows*PHOTOCOLUMNS,MPI_UNSIGNED_CHAR,&stat
us);
      printf("Soy: %d voy a leer, mi offset: %lld\n",rank,point);
```

```
	for (int i = 0; i < readrows; i++)
	{
		for (int j = 0; j < PHOTOCOLUMNS; j++)
		{
			//pixel RGB data
			red = (int)*photoData;
			photoData = photoData + sizeof(unsigned char);
			green = (int)*photoData;
			photoData = photoData + sizeof(unsigned char);
			blue = (int)*photoData;
			photoData = photoData + sizeof(unsigned char);
			grayScaleIMG[i*PHOTOCOLUMNS+j] = red*.2+ green*.7+ blue*.1;

		}
	}
	MPI_File_close(&photo);
	printf("Tengo imagen %d\n",rank);
	const float Gx[] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
	const float Gy[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

	gaussian_filter(grayScaleIMG,blurIMG,PHOTOCOLUMNS,readrows,SIGMA);
	convolution(blurIMG, edgeImgX, Gx, PHOTOCOLUMNS, readrows, 3, 0);
	convolution(blurIMG, edgeImgY, Gy, PHOTOCOLUMNS, readrows, 3, 0);
	printf("Aplique el filtro %d\n",rank);
	int x = 0;
	int y = startrow;
	int startread = 0;
	if(rank != 0 ){
		startread = 32;
	}


	for (int i = startread; i < ((PHOTOROWS/NPROCESS)+startread); i++)
	{
		for (int j = 0; j < PHOTOCOLUMNS; j++)
		{
			bufferPixelData[0] = x;
			bufferPixelData[1] = y;
			float temp = 0;
			temp =
sqrt(edgeImgX[i*PHOTOCOLUMNS+j]*edgeImgX[i*PHOTOCOLUMNS+j] +
edgeImgY[i*PHOTOCOLUMNS+j]*edgeImgY[i*PHOTOCOLUMNS+j]);
			bufferPixelData[2] = temp;

			//printf("Envio el: %d %d\n",y,x);
			MPI_Send(&bufferPixelData,3,MPI_INT,0,1,commPadre);
			x = x+1;
		}
		x=0;
		y = y+1;
	}



}
```

```
/* Programa principal */

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

  int rank,size;
  MPI_Comm commPadre;
  int tag;
  MPI_Status status;
  int buf[5];
  int errCodes[NPROCESS];


  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();
      sleep(1);
        /* Codigo del maestro */

MPI_Comm_spawn("./pract2",MPI_ARGV_NULL,NPROCESS,MPI_INFO_NULL,0,MPI_COMM_WOR
LD,&commPadre,errCodes);
        /*En algun momento dibujamos puntos en la ventana algo como
          */
      getPhotoPixels(commPadre);
      sleep(10000);

  }


  else {
    /* Codigo de todos los trabajadores */
    /* El archivo sobre el que debemos trabajar es foto.dat */
      edgeDetection(rank,commPadre);

  }

  MPI_Finalize();

}
```

7. Versiones alternativas

En el desarrollo de la práctica se han realizado varias versiones con distinto planteamiento, aunque finalmente se ha optado por la que se ha mostrado y explicado anteriormente, puede ser de su interés disponer de ellas. Las versiones están incluidas en la carpeta *AltVerWIP* y son:

- **Lecturasolapada:** en esta versión la lectura de los datos necesarios para el calculo del filtro se

realizaba directamente por los procesos mediante un “solapamiento” en la lectura. Es decir, los procesos no hablan entre ellos para transferirse la información necesaria para el calculo de los pixeles de los bordes.

- **PxPreadHDD**: en esta versión solo se almacena en memoria principal la información necesaria para el cálculo de un único píxel (matriz de 5x5) a la vez por cada uno de los procesos. Esto se consigue moviendo la posición de la que se lee con MPI_File_read_at() . Se desarrollo esta versión a modo de prueba y con el objetivo de ver que era posible conseguir un prototipo funcionando en el poco tiempo disponible que se tenía. En todo caso esta versión no es recomendable ya que tenemos un cuello de botella muy grande en la memoria secundaria.

8. Conclusiones

La realización de esta práctica he aprendido y sufrido la importancia de comprender el funcionamiento de los punteros en C. Además, he profundizado los conocimientos del funcionamiento de las funciones de I/O de OpenMPI y del funcionamiento de los offset. También he comprendido lo que es una convolución de matrices y la importancia que esta operación tiene para la mayoría de los filtros de imágenes. Por último, he afianzado lo aprendido en la practica 2 anterior donde se resaltaba la importancia de dividir bien los distintos datos sobre los que trabajaría un nodo.