



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:
18 de marzo de 2020

Índice

1. Enunciado	2
2. Introducción	3
3. Planteamiento de la solución	4
3.1. Tipos de nodos	4
3.2. Algoritmos principales	4
3.2.1. Obtención de los vecinos	5
3.2.2. Cálculo del número mínimo	7
4. Diseño de la solución	8
4.1. Función principal del programa (<i>main</i>)	8
4.2. Lectura del fichero (<i>load-data</i>)	9
4.3. Comprobación de la cantidad de números (<i>check-size</i>)	10
4.4. Envío de su número a cada nodo (<i>send-numbers</i>)	11
4.5. Obtención de los vecinos (<i>get-neighbors</i>)	12
4.6. Cálculo del número mínimo (<i>calculate-min</i>)	13
4.6.1. Mejora del algoritmo	15
4.6.2. Representación del flujo de eventos de envío y recepción	16
Referencias	18

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 renderizado 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 píxeles que forman la matriz, es $400^2 (L^2) = \mathbf{160000}$ **píxeles**.

Estos valores están 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*.

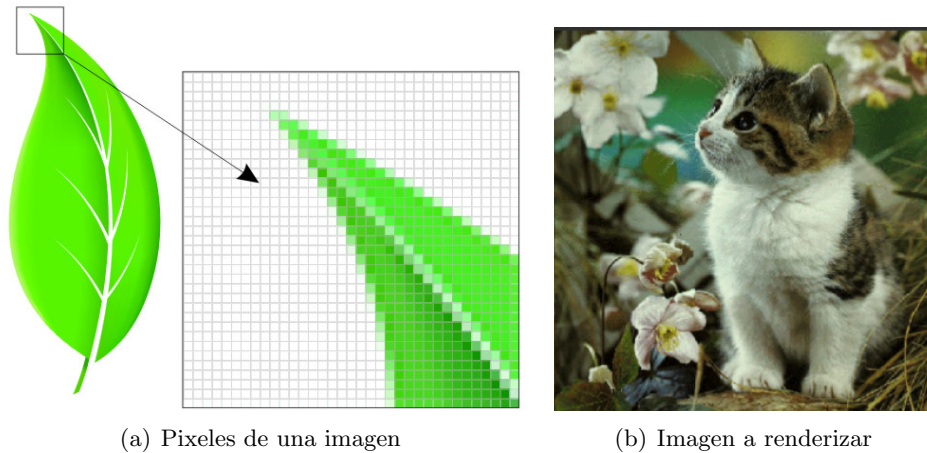


Figura 1: Representación de píxeles que forman una imagen

3. Planteamiento de la solución

El programa comienza con la 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 *IX11* (función *initX()*). Además también creará *N* procesos hijos, conocidos a lo largo del programa como *esclavos o trabajadores* (definidos con la constante creada en tiempo de compilación *EMPLOYEES-NUMBER*).

Tras lanzar el *maestro* los *N* trabajadores, este se mantendrá a la espera en todo momento, para recibir los píxeles de cada trabajador, los cuales debe de dibujar en la imagen.

Mientras el maestro se mantiene en dicha espera, a cada *trabajador*, se le asignará una zona de trabajo, esta zona pertenece a un rango de líneas 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 0* o *nodo 0* y los demás *nodos*.

- **Rank 0:** Este nodo corresponde al primer proceso creado. Encargado de la lectura del fichero *datos.dat*, el cual contiene los números que más tarde asignará el mismo a los demás nodos que forman la red toroidal.

A la hora de realizar la asignación de los números a los respectivos nodos restantes (incluyéndose el mismo), debe de comprobar que la cantidad de números obtenidos del fichero es igual al tamaño del toroide *N* (*nº* de nodos que lo forman) si esta comprobación es exitosa, continuará la ejecución normal del programa.

En caso contrario, a mi elección bien sea por que el tamaño del toroide o la cantidad de los números sea menor o mayor. El *rank 0* abortará la ejecución del programa. Tanto si la comprobación es correcta como si no, este difundirá el resultado a los demás nodos. Para ello se ha utilizado la función *Bcast()* de la librería de **MPI**.

Una vez asignados los respectivos números a cada nodo, tras calcular el número mínimo de la red toroidal, *rank 0* mostrará dicho número por pantalla, y el programa finalizará.

- **Los demás nodos:** Estos tipos de nodos recibirán del *rank 0* la decisión de continuar o no. Si continúa la ejecución normal se les asignará un número real, el cual tras

obtener cada uno sus respectivos vecinos, se llevará a cabo el algoritmo para obtener *el menor número de la red toroidal*.

3.2. Algoritmos principales

Los algoritmos principales de esta práctica son: **la obtención de los vecinos** (de cada nodo) y **cálculo del número mínimo** de la red toroidal.

3.2.1. Obtención de los vecinos

Este algoritmo es fundamental en el programa, debido a que gracias a él se ve de una manera más intuitiva la estructura de la red toroidal, donde cada nodo o rank conoce a sus respectivos vecinos. Cada nodo tiene un total de *cuatro vecinos*, denominados: **Norte, Sur, Este y Oeste**.

En concreto los vecinos Norte y Sur de cada nodo, se han obtenido gracias al uso de las filas de la red toroidal. En cambio los vecinos Este y Oeste, se han obtenido gracias a las columnas.

Para saber en que posición se encuentra cada nodo o rank, dentro de la red toroidal, representada como una matriz de $\mathbf{L} \times \mathbf{L}$. Que esta determinada por el número de fila y de columna (**fila,columna**), por ejemplo: (0,0), (0,1), (1,0), etc. Para determinar el valor de la fila, se ha utilizado la división de \mathbf{L} entre el **rank** [3] del nodo actual (\mathbf{L}/\mathbf{rank}). Y para obtener el valor de la columna se hace al igual que la fila, pero en este caso con el módulo ($\mathbf{L} \% \mathbf{rank}$).

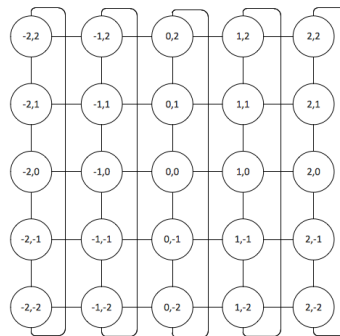


Figura 2: Representación red toroidal como una matriz.

▪ Vecinos Norte y Sur

Se han categorizado la forma de obtener estos vecinos de la siguiente forma:

- **Primera fila (superior):** Esta es la fila con el valor 0 de nuestra matriz, para obtener el vecino del *Norte*, debemos de obtener el rank que pertenece a la última fila, y en la misma columna de nuestra matriz (representación de la red toroidal). En cambio, el vecino *Sur* será el rank o nodo inmediatamente inferior al actual.
- **Última fila (inferior):** Esta es la fila con el valor $L - 1$ de nuestra matriz, al igual que la primera fila es peculiar. Esto se debe a la forma de obtener el vecino del *Sur*, dado que este sera el rank que pertenece a la primera fila y esta situado en la misma columna. En cambio el vecino del *Norte*, sera el rank que este situado justo encima de el, en la misma columna.
- **Filas centrales (superior - inferior):** En este tipo de filas el vecino *Norte* es el vecino inmediatamente superior y el vecino *Sur* es el *rank* inmediatamente inferior.

Las distintas operaciones utilizadas para obtener cada tipo de vecino, segun la posición en la que se encuentre *rank* o nodo, son las siguientes:

Fila	Norte	Sur
<i>Primera (0)</i>	$L * (L - 1) + \text{rank}$	$\text{rank} + L$
<i>Centrales (0 - [L-1])</i>	$\text{rank} - L$	$\text{rank} + L$
<i>Última (L - 1)</i>	$\text{rank} - L$	$\text{rank} \% L$

Cuadro 1: Fórmulas para obtención vecinos *Norte* y *Sur*.

■ Vecinos *Este* y *Oeste*

- **Primera columna (izquierda):** Con esta columna, nos referimos a la columna 0 de la matriz. En este caso, el vecino del *Oeste* lo encontramos en la misma fila, pero en la columna de más a la derecha. En cambio el vecino del *Este* se encuentra inmediatamente a continuación del nodo o rank actual, es decir, en la misma fila pero en la siguiente columna (columna actual + 1).
- **Última columna (derecha):** Esta columna es la columna $L - 1$ de la matriz. En este caso, el vecino del *Este* lo encontramos en la misma fila, pero en la columna de más a la izquierda. En cambio el vecino del *Oeste* se encuentra inmediatamente a continuación del nodo o rank actual, es decir, en la misma fila pero en la columna anterior (columna actual - 1).
- **Columnas centrales (superior - inferior):** En este tipo de columnas el vecino *Este* es el vecino que esta situado a la derechaa del *rank* (nodo) actual y el vecino *Oeste* es el *rank* inmediatamente a la izquierda del actual.

Columna	Oeste	Este
Primera (0)	$\text{rank} + (L - 1)$	$\text{rank} + 1$
Centrales ($0 - [L-1]$)	$\text{rank} - 1$	$\text{rank} + 1$
Última ($L - 1$)	$\text{rank} - (L - 1)$	$\text{rank} \% L$

Cuadro 2: Fórmulas para obtención vecinos *Oeste* y *Este*.

3.2.2. Cálculo del número mínimo

El segundo algoritmo principal de este programa, es el de calcular el número mínimo de toda la red toroidal, para ello cada nodo ira comprobando su número y el de sus vecinos y se quedara con el menor de ellos, esta comprobación la realizará enviando su numero a cada vecino y recogiendo el el de cada vecino para comprobarlo con el suyo.

Al realizar este proceso cada nodo, a nivel global se realizará para todos los nodos de la red, entonces de esta forma al tener cada nodo vecinos que a la vez ellos mismos son vecinos de otros nodos, todos los nodos se quedarán finalmente con el número menor de toda la red toroidal.

La secuencia de acciones de este algoritmo es la siguiente:

1. Intercambio de números entre vecino *Norte* y *Sur* (envio Norte, recibo Sur).
2. Comparación entre el número actual y el recibido, quedandose el nodo actual con el número mínimo.
3. Intercambio de números entre vecino *Este* y *Oeste* (envio Este, recibo Oeste).
4. Comparación entre el número actual y el recibido, quedandose el nodo actual con el número mínimo.

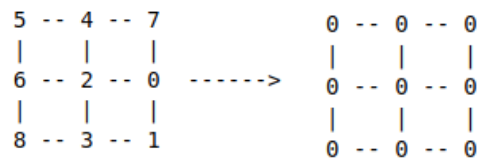


Figura 3: Representación obtención del valor mínimo red toroidal

4. Diseño de la solución

A continuación, se explicarán las funciones que forman parte del programa, el cual da una solución al enunciado de esta práctica.

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*, *numbers-n* (cantidad de números), *size* (tamaño del toroide), *data* (vector donde se almacenarán los números obtenidos del fichero), etc.

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

- *MPI-Init()* [5]: Para inicializar la estructura de comunicación de MPI entre los procesos.
- *MPI-Comm-Size()* [4]: Para obtener el tamaño de la comunicacion (número de proceso a ejecutar en este caso, ranks), etc.
- *MPI-Comm-rank()* [3]: Para establecer el identificador de cada proceso *rank*.

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

```
1  int main(int argc, char *argv[])
2  {
3
4      /*Variables*/
5      int rank, size, numbers_n, finish;
6      long double number, min_number;
7      long double *data = malloc(DATA_SIZE);
8      int *neighbors = malloc(NEIGHBORS_SIZE);
9      MPI_Status status;
10
11     /* Initialize MPI program */
12     MPI_Init(&argc, &argv);
13     MPI_Comm_size(MPI_COMM_WORLD, &size);
14     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16     if (rank == FIRST_RANK)
17     {
18         /*Get quantity of numbers*/
19         numbers_n = load_data(data);
```

```

20
21     finish = check_size(size, numbers_n);
22
23     if (finish != TRUE)
24     {
25         add_numbers(data, size);
26     }
27 }
28
29 MPI_Bcast(&finish, 1, MPI_INT, 0, MPI_COMM_WORLD);
30
31 if (finish != TRUE)
32 {
33     MPI_Recv(&number, 1, MPI_LONG_DOUBLE, 0, MPI_ANY_TAG,
34             MPI_COMM_WORLD, NULL);
35     printf("[X] RANK[%d] --> %.2Lf\n", rank, number);
36
37     get_neighbors(rank, neighbors);
38
39     min_number = calculate_min(rank, number, neighbors);
40
41     print_min_number(rank, min_number);
42 }
43
44 /* Finalize MPI program */
45 MPI_Finalize();
46
47 return EXIT_SUCCESS;
48 }

```

4.2. Lectura del fichero (*load-data*)

La primera función a llamar dentro del *main*, es la *load-data()*, encargada de la lectura del fichero, esta función añade los números que almacena el fichero al vector de tipo *long double* denominado *data*. Esta función obtiene los números del fichero, separando cada uno utilizando como separador la *coma*.

Para la apertura del fichero se ha utilizado la función *open-file()*, cuyo principal objetivo es devolver el descriptor del archivo *datos.dat*. A esta función se le pasa como parámetro *DATA-PATH* que es la ruta del archivo a leer y *READ-MOD* que indica el modo de apertura del archivo, en este caso el modo de lectura.

```

1  /* Open file */
2  FILE *open_file(const char *path, const char *mode)

```

```

3 {
4     FILE *file;
5     if ((file = fopen(path, mode)) == NULL)
6     {
7         fprintf(stderr, "[X] RANK[0]: Error opening file.\n");
8         exit(EXIT_FAILURE);
9     }
10    return file;
11 }

```

Una vez cargados los datos en el vector *data*, la función devolverá la variable *i*, la cual almacena el valor de la cantidad de números que contiene el fichero.

```

1  /* Load data (numbers) from datos.dat */
2  int load_data(long double *data)
3  {
4      FILE *file = open_file(DATA_PATH, READ_MOD);
5      char line[MAX_SIZE];
6      char *token;
7      int i;
8
9      fgets(line, sizeof(line), file);
10     fclose(file);
11
12     data[0] = atof(strtok(line, SEPARATOR));
13
14     for (i = 1; (token = strtok(NULL, SEPARATOR)) != NULL; i++)
15     {
16         data[i] = atof(token);
17     }
18
19     return i;
20 }

```

4.3. Comprobación de la cantidad de números (*check-size*)

Esta función se encarga de comprobar si la cantidad de números (*numbers-n*) es igual al tamaño (*size*) [4] del toroide. Estos son los argumentos que se les pasa a la función. La función devolverá True o False si es igual o no estas variables, si la devolución es False, entonces finalizará la ejecución normal del programa. Y el proceso *rank 0*, mandará una difusión a los demás procesos, para que así aborten su ejecución. Esto se realiza con la primitiva *MPI-Bcast()* [2].

```

1     MPI_Bcast(&finish, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

```
1  /* Check toroid's size (number of nodes) */
2  int check_size(const int size, const int numbers_n)
3  {
4      int finish = FALSE;
5      if (size != numbers_n)
6      {
7          fprintf(stderr, "[X] RANK[0]: Error, quantity of numbers (%d) is different at toroid's size (%d | L = %d)\n",
8                  numbers_n, size, L);
9          finish = TRUE;
10     }
11
12     return finish;
13 }
```

4.4. Envío de su número a cada nodo (*send-numbers*)

Esta función tiene el objetivo de añadir o enviar a cada nodo (*rank*) [3] su número, esta tarea es realizada por *rank 0*.

Tanto para el envío como para la recepción del nodo se han utilizado las primitivas básicas de MPI (*MPI-Send* [7] y *MPI-Recv* [6]). Esta función recoge como parámetros el vector *data* y la variable entera *size* (tamaño toroide).

```
1  void send_numbers(long double *data, const int size)
2  {
3      int i;
4      long double number;
5
6      for (i = 0; i < size; i++)
7      {
8          number = data[i];
9          MPI_Send(&number, 1, MPI_LONG_DOUBLE, i, SEND_TAG,
10                 MPI_COMM_WORLD);
11     }
12     free(data);
13 }
```

Por último, se libera el espacio de memoria reservado para el vector de números *data*.

4.5. Obtención de los vecinos (*get-neighbors*)

Esta función se encarga de que cada proceso, obtenga el rank de cada vecino, es decir, de sus vecinos *Norte*, *Sur*, *Este* y *Oeste*. Para ello esta función recogerá como parámetros el rank del proceso que la ejecute, y su lista de vecinos vacía en este caso, representada mediante el vector de enteros llamado *neighbors*.

Esta función es muy importante para la función que calcula el mínimo número de la red toroidal *calculate-min()*, debido que aquí cada nodo o rank conoce quienes son sus vecinos, con los cuales realizará dicha comprobación intercambiando sus números.

El objetivo y funcionamiento de este algoritmo se ha explicado de una forma más teórica en el apartado 2, en concreto, la sección 2.2.1.

```
1 void get_neighbors(const int rank, int *neighbors)
2 {
3     int row = rank / L;
4     int column = rank % L;
5
6
7     switch (row)
8     {
9     case 0:
10         neighbors[NORTH] = L * (L - 1) + rank;
11         neighbors[SOUTH] = rank + L;
12         break;
13
14     case L - 1:
15         neighbors[NORTH] = rank - L;
16         neighbors[SOUTH] = rank % L;
17         break;
18
19     default:
20         neighbors[NORTH] = rank - L;
21         neighbors[SOUTH] = rank + L;
22         break;
23     }
24
25     switch (column)
26     {
27     case 0:
28         neighbors[WEST] = rank + (L - 1);
29         neighbors[EAST] = rank + 1;
30         break;
31
32     case L - 1:
33         neighbors[WEST] = rank - 1;
34         neighbors[EAST] = rank - (L - 1);
```

```
35         break;
36
37     default:
38         neighbors[WEST] = rank - 1;
39         neighbors[EAST] = rank + 1;
40         break;
41     }
42 }
```

Como se puede observar el proceso de obtención se debe en dos partes, la obtención de los nodos pertenecientes a las filas y los pertenecientes a lo que serían las columnas.

- **Filas:** por medio de las filas se obtienen los vecinos *Norte* y *Sur*, esto se realiza con la utilización de las fórmulas de la Tabla 1.
- **Columnas:** por medio de las columnas se obtienen los vecinos *Este* y *Oeste*, esto se realiza con la utilización de las fórmulas de la Tabla 2.

La implementación en ambas partes se ha realizado con un *switch*, en el cual dependiendo de la posición del *rank* se obtendrá de una forma u otra los vecinos.

4.6. Cálculo del número mínimo (*calculate-min*)

El objetivo principal de esta función es la del cálculo del número mínimo de la red toroidal completa, en el cual todos los elementos de la red deben de contener finalmente el número mínimo.

Para ello se explicará a continuación de una forma más práctica el procedimiento de este algoritmo, dado que en la sección 2.2.2 ya ha sido explicado de una forma más teórica.

Por tanto, una vez obtenidos los vecinos de cada nodo, se lleva a cabo el cálculo del mínimo. La primera versión de este algoritmo estaba compuesta por dos bucles de tipo *for*, en los cuales primero, se calculaba los números de los vecinos correspondientes a las filas *Norte* y *Sur*, enviando a *Sur* el número que contiene el nodo actual y recibiendo del *Norte* su número. Y más tarde el mismo procedimiento con las columnas *Este* y *Oeste*. Con calcular me refiero a quedarse el nodo actual finalmente con el menor de cada número. Para que esto sea aplicable a los elementos de toda la fila o de toda la columna se harían 1 a L-1 iteraciones en cada bucle.

Por lo que el procedimiento a seguir sería el siguiente:

- **Primer bucle *for*** (cálculo de las filas):
 1. El *rank* actual envía su número (*my-number*) al vecino del *Sur*.

2. El *rank* actual recibe el número de su vecino del *Norte* (*his-number*).
 3. Se realiza el cálculo del número mínimo entre estos dos número (se coge el menor de ellos).
- **Segundo bucle *for*** (cálculo de las columnas):
1. El *rank* actual envía su número (*my-number*) al vecino del *Este*.
 2. El *rank* actual recibe el número de su vecino del *Oeste* (*his-number*).
 3. Se realiza el cálculo del número mínimo entre estos dos número (se coge el menor de ellos).

```

1  /*Calculate the minium value*/
2  long double calculate_min(const int rank, long double my_number,
3  int *neighbors)
4  {
5      int i;
6      long double his_number;
7
8      for (i = 1; i < L; i++)
9      {
10         /* Calculate rows */
11         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[SOUTH],
12             SEND_TAG, MPI_COMM_WORLD);
13         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[NORTH],
14             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
15         my_number = (his_number > my_number ? my_number :
16             his_number);
17     }
18
19     for (i = 1; i < L; i++){
20         /* Calculate columns */
21         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[EAST],
22             SEND_TAG, MPI_COMM_WORLD);
23         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[WEST],
24             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
25         my_number = (his_number > my_number ? my_number :
26             his_number);
27     }
28
29     free(neighbors);
30
31     return my_number;
32 }

```

Como se puede observar, una vez realizado ambos bucles, todos los elementos de la red tendrán almacenado el número mínimo.

4.6.1. Mejora del algoritmo

Una vez desarrollada la primera versión del algoritmo de calcular el mínimo, se observó, que realmente se realizaría lo mismo dejando simplemente un bucle *for*, y además esto podría mejorar el rendimiento del programa.

El procedimiento de esta versión mejorada, es mas peculiar, debido a que no se calculan de forma secuencial, todos los vecinos de las filas (*Norte-Sur*) y después los de las columnas *Este-Oeste*, si no que cada nodo en una misma interacción calculará ambas opciones. El procedimiento a seguir sera el siguiente:

En un único bucle *for*:

1. El *rank* actual envía su número (*my-number*) al vecino del *Sur*.
2. El *rank* actual recibe el número de su vecino del *Norte* (*his-number*).
3. Se realiza el cálculo del número mínimo entre estos dos número, obtenidos de las filas (se coge el menor de ellos).
4. El *rank* actual envía su número (*my-number*) al vecino del *Este*.
5. El *rank* actual recibe el número de su vecino del *Oeste* (*his-number*).
6. Se realiza el cálculo del número mínimo entre estos dos número, obtenidos de las columnas (se coge el menor de ellos).

```

1  /*Calculate the minium value*/
2  long double calculate_min(const int rank, long double my_number,
3  int *neighbors)
4  {
5      int i;
6      long double his_number;
7
8      for (i = 1; i < L; i++)
9      {
10         /* Calculate rows */
11         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[SOUTH],
12             SEND_TAG, MPI_COMM_WORLD);
13         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[NORTH],
14             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
15         my_number = (his_number > my_number ? my_number :
16             his_number);
17
18         /* Calculate columns */
19         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[EAST],
20             SEND_TAG, MPI_COMM_WORLD);

```



```

16     MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[WEST],
17             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
18     my_number = (his_number > my_number ? my_number :
19                 his_number);
20 }
21 free(neighbors);
22 return my_number;
23 }

```

Por último en ambas opciones, se libera el espacio de memoria reservado para el vector de los vecinos *neighbors*.

4.6.2. Representación del flujo de eventos de envío y recepción

Un ejemplo de representación del estado de la matriz, en un inicio y al final, este proceso se lleva a cabo en la función *calculate_min()*, tras este proceso el resultado obtenido sería el siguiente:

5	--	4	--	7		0	--	0	--	0
					----->					
6	--	2	--	0		0	--	0	--	0
8	--	3	--	1		0	--	0	--	0

Figura 4: Representación red toroidal al inicio (izquierda) y al final (derecha)

Este proceso se lleva a cabo, como ya se ha explicado anteriormente gracias a un bucle *for* que va de *1* a $(L - 1)$. A continuación, se mostrará de forma representativa (mediante una matriz que simulara la red toroidal), los cambios y pasos que sigue este proceso, desde su estado inicial al su estado final, se pueden ver en la Figura 4. Caba destacar que el valor de la variable ***L*** en este ejemplo es de **3**.

1. Estado inicial

5	--	4	--	7
6	--	2	--	0
8	--	3	--	1

2. **Primera iteración:** Cada **Rank** envía a su vecino *Sur* su número y recibe el número de su vecino del *Norte*. También cada *Rank* enviará su número a su vecino *Este* y recibirá el número del vecino del *Oeste*. Entre medias de ambos procesos se calculará el mínimo de cada cambio (explicado anteriormente en el apartado 4.6.1).

```

5 -- 3 -- 1
|   |   |
0 -- 0 -- 0
|   |   |
0 -- 0 -- 0

```

3. **Segunda iteración:** Se realizarán de nuevo el mismo proceso que en la primera iteración, debido a que así los ranks restantes tomen como número el valor mínimo. Este sería el **estado final del toroide**.

```

0 -- 0 -- 0
|   |   |
0 -- 0 -- 0
|   |   |
0 -- 0 -- 0

```

Referencias

- [1] Javier Ayllón. Topologías de redes - practica 1 - diseño de infraestructuras de redes.
- [2] https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Bcast. Broadcast mpi.
- [3] https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Comm_rank. Determinar *rank* mpi.
- [4] https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Comm_size. Determinar tamaño del comunicador mpi.
- [5] https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Init. Inicialización mpi.
- [6] https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Recv. Recepción mpi.
- [7] https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Send. Envío mpi.