Universidad de Castilla-La Mancha

Escuela Superior de Informática

Diseño de Infraestructuras de Red

Grado en Ingeniería Informática
2019 - 2020

Práctica 1: Red Toroidal

Autor:
David Camuñas Sánchez

Fecha:
18 de marzo de 2020

Índice

1. Enunciado

Dado un archivo con nombre datos.dat, cuyo contenido es una lista de valores separados por comas, nuestro programa realizará lo siguiente: El proceso de rank 0 destribuirá a cada uno de los nodos de un toroide de lado L,los L x L numeros reales que estarán contenidos en el archivo datos.dat. En caso de que no se hayan lanzado suficientes elementos de proceso para los datos del programa, éste emitirá un error y todos los procesos finalizarán. En caso de que todos los procesos han recibido su correspondiente elemento,comenzará el proceso normal del programa. Se pide calcular el elemento menor de toda la red, el elemento de proceso con rank 0 mostrará en su salida estándar el valor obtenido. La complejidad del algoritmo no superará O(n) Con n número de elementos de la red.

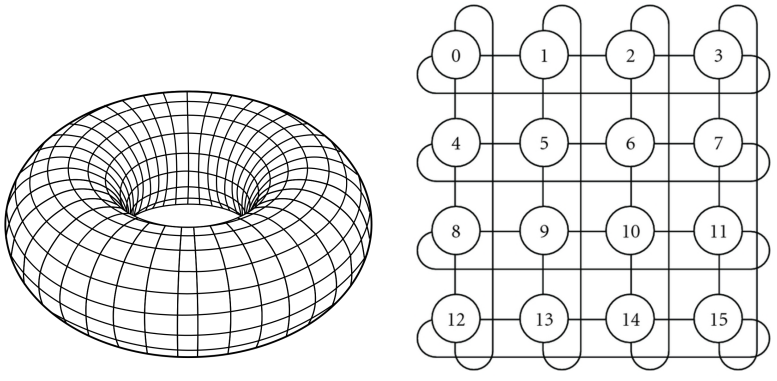
2. Introducción

El objetivo de esta práctica es la creación de uno de los tipos de redes de comunicación existente, esta es la ***Red Toroidal*** o de *Anillo*.

A la hora de dibujar este tipo de red para entender su funcionamiento, se puede mostrar como una matriz.

Cuya peculiaridad es que son redes cuadradas, de *lado* ***L***, donde cada fila o columna de un extremo, conecta con su fila o columna inversa, es decir, los elementos que formarían la última fila de la matriz se conectarían (serían *"vecinos"*) con los elementos de la primera fila de la matriz y asi sucesivamente.

A continuación, se mostrará un ejemplo donde se puede observar la estructura de este tipo de red.



(a) Red toroidal en 3D (b) Topología de una red toroidal

Figura 1: Imágenes red toroidal

3. Planteamiento de la solución

Para solucionar el problema, se tiene en cuenta que una *red Toroidal* tiene un valor de ***lado L***. Donde el *número de procesos* **N** esta determinado por: L^2

En este caso el valor de **L** es **3**, por lo tanto, el valor de **N** será **9**, esto quiere decir que este toroide estaŕa formado por 9 procesos (denominados en el código como *ranks*).

El número de lado **L** del toroide se encuentra definido en */include/definitions*, como:

define L 3

Si se quiere realizar la simulación con un valor de lado distinto, se deberá cambiar el valor de esta constante. Además del número total de procesos a crear, pasado como argumento al comando de ejecución (*mpirun*) en la línea de ordenes. Este valor se puede encontrar en el *Makefile* del proyecto.

Otra constante importante, es la que determina el tamaño del buffer lectura del fichero (en este caso *datos.dat*), debido a que si se quiere leer una gran cantidad de números, y asi crear una gran cantidad de nodos (ranks), se debe de modificar su valor a uno mayor.

define MAX-SIZE 1024

3.1. Tipos de nodos

En este problema encontramos dos tipos de nodos: el ***rank 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 asignara 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 (incluyendose 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, continuara 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* cero 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 libreria de **MPI**.

  Una vez asignados los respectivos números a cada nodo, tras calcular el número mínimo de la red toroidal, *rank 0* mostrara 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 continua la ejecución normal se les asignará un número real, el cual trás 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 el se ve de una manera más intutiva 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 **LxL**. 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 **L** entre el **rank** del nodo actual (**L/rank**). Y para obtener el valor de la columna se hace al igual que la fila, pero en este caso con el módulo (**L %rank**).

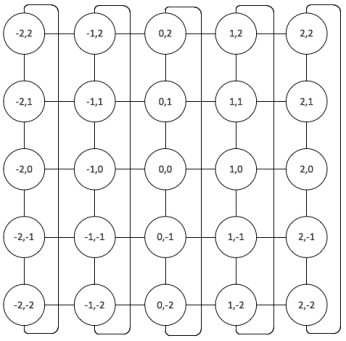


Figura 2: Representación red toroidal como una matriz.

- ***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**
Primera (0)	L * (L - 1) + rank	rank + L
Centrales (0 - [L-1])	rank - L	rank + L
Última (L - 1)	rank - L	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)	rank + (L - 1)	rank + 1
Centrales (0 - [L-1])	rank - 1	rank + 1
Última (L - 1)	rank - (L - 1)	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 cálcular 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 comnprobación 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.

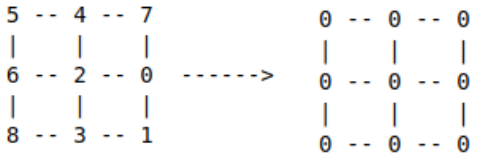


Figura 3: Reprersentación obtrención del valor mínimo red toroidal

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 definicion 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 declarán las primitivas de ***MPI*** como:

- textitMPI-Init() para inicializar la estructura de comunicación de MPI entre los procesos.
- *MPI-Comm-Size*() para obtener el tamaño de la comunicacion (número de proceso a ejecutar en este caso, ranks), etc.
- *MPI-Comm-rank() 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,
                MPI_COMM_WORLD, NULL);
34          printf("[X] RANK[%d] --> %.2Lf\n", rank, number);
35
36          get_neighbors(rank, neighbors);
37
38          min_number = calculate_min(rank, number, neighbors);
39
40          print_min_number(rank, min_number);
41      }
42
43      /* Finalize MPI program */
44      MPI_Finalize();
45
46      return EXIT_SUCCESS;
47  }
```

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 obitiene los números del fichero, seprarando 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)
```

```
{
    FILE *file;
    if ((file = fopen(path, mode)) == NULL)
    {
        fprintf(stderr, "[X] RANK[0]: Error opening file.\n");
        exit(EXIT_FAILURE);
    }
    return file;
}
```

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.

```
/* Load data (numbers) from datos.dat */
int load_data(long double *data)
{
    FILE *file = open_file(DATA_PATH, READ_MOD);
    char line[MAX_SIZE];
    char *token;
    int i;

    fgets(line, sizeof(line), file);
    fclose(file);

    data[0] = atof(strtok(line, SEPARATOR));

    for (i = 1; (token = strtok(NULL, SEPARATOR)) != NULL; i++)
    {
        data[i] = atof(token);
    }

    return i;
}
```

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*) 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 asi aborten su ejecución. Esto se realiza con la primitiva *MPI-Bcast()* [**?**].

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

```
/* Check toroid's size (number of nodes) */
int check_size(const int size, const int numbers_n)
{
    int finish = FALSE;
    if (size != numbers_n)
    {
        fprintf(stderr, "[X] RANK[0]: Error, quantity of numbers (%
           d) is diferent at toroid's size (%d | L = %d)\n",
                numbers_n, size, L);
        finish = TRUE;
    }

    return finish;
}
```

4.4. Envio de su número a cada nodo (*add-numbers*)

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

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

```
/* Add number to nodes (ranks) */
void add_numbers(long double *data, const int size)
{
    int i;
    long double number;

    for (i = 0; i < size; i++)
    {
        number = data[i];
        MPI_Send(&number, 1, MPI_LONG_DOUBLE, i, SEND_TAG,
           MPI_COMM_WORLD);
    }

    free(data);
}
```

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 cálcula 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 comprobaqción intgercambiando 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 debide 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 ambás partes se ha realizado con un *switch*, en el cual dependiendo de la posición del *rank* se obtrendrá 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 continuaicón 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 teoríca.

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 cálculaba los números de los vecinos correspondientes a las filas *Norte y Sur*, enviando a *Sur* el número que contiene el nodo actual y recibiendolo del *Norte* su número. Y más tarde el mismo procedimiento con las columnas *Este y Oeste*. Con cálcular 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 iteracciones 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,
       int *neighbors)
3  {
4      int i;
5      long double his_number;
6
7      for (i = 1; i < L; i++)
8      {
9          /* Calculate rows */
10         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[SOUTH],
               SEND_TAG, MPI_COMM_WORLD);
11         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[NORTH],
               MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
12         my_number = (his_number > my_number ? my_number :
               his_number);
13     }
14
15     for (i = 1; i < L; i++){
16         /* Calculate columns */
17         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[EAST],
               SEND_TAG, MPI_COMM_WORLD);
18         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[WEST],
               MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
19         my_number = (his_number > my_number ? my_number :
               his_number);
20     }
21
22     free(neighbors);
23
24     return my_number;
25 }
```

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 cálcular el mínimo, se observo, que realmente se realizaría lo mismo dejando simplemente un bucle *for*, y además esto podría mejorar el rendmiento 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,
       int *neighbors)
3  {
4      int i;
5      long double his_number;
6
7      for (i = 1; i < L; i++)
8      {
9          /* Calculate rows */
10         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[SOUTH],
               SEND_TAG, MPI_COMM_WORLD);
11         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[NORTH],
               MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
12         my_number = (his_number > my_number ? my_number :
               his_number);
13
14         /* Calculate columns */
15         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[EAST],
               SEND_TAG, MPI_COMM_WORLD);
```

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

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

Un ejemplo de representación del estado de la matriz, en un inicio y al final, tras haber calculado el mínimo, seria el siguiente:

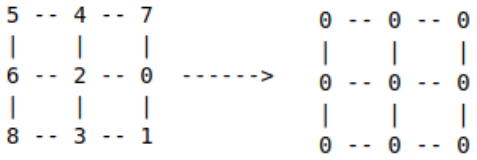


Figura 4: Representación red toroidal al inicio y al final