Universidad de Castilla-La Mancha
Escuela Superior de Informática

Diseño de Infraestructura de Red

Grado en Ingeniería Informática
3º - Ingeniería de Computadores
2019 – 2020

Práctica 1: Red Toroide

Autor:
Mario Pérez Sánchez-Montañez

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($\sqrt{n}$) Con n número de elementos de la red.

2. Planteamiento de la solución

La topología de red toroidal es un tipo de topología de red de cuadrícula. La topología de red de cuadrícula es una topología en la que cada nodo de la red está conectado con dos nodos vecinos a lo largo de una o más dimensiones (ver Figura 1).

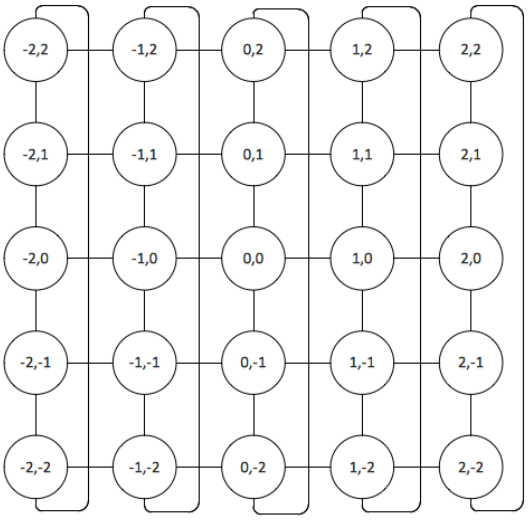

Figura 1: Ejemplo topología toroidal [6]

La solución planteada se basa en crear LxL procesos (siendo L el lado del toroide). En este caso planteo una solución de lado 4, por lo tanto, 16 procesos. En el fichero ***toroidal.c***

se especifica una constante para el lado. Por ello, si se quiere cambiar el lado del toroide se debe modificar esta constante y los nodos indicados en el ***Makefile***.

El el proceso con rank 0 se encarga de leer el fichero ***datos.dat*** y de enviar el respectivo número a cada nodo, incluido sí mismo. Si este proceso se realiza con éxito los procesos comenzarán a enviar sus número a sus vecinos en busca del mínimo número de la red, en caso contrario la variable ***end*** se activará todos los procesos finalizarán.

2.1. Cálculo de vecinos

La parte fundamental de la solución es el cálculo de los respectivos vecinos, ya que esto especificará la topología de la red. Como se ha mencionado anteriormente, cada nodo tiene cuatro vecinos (Norte, Sur, Este u Oeste). Los vecinos Norte y Sur se calcularán gracias a la fila del nodo y los vecinos Este y Oeste a la columna del nodo. La fila en la que se encuentra el nodo se consigue gracias a la división de la posición del nodo (indicada por el *rank*) en la red entre el lado ***L***. La columna se calcula de la misma forma, pero en este caso realizando el módulo. Estos cálculos se pueden realizar gracias a que la red se representa por una matriz cuadrada de ***LxL***.

2.1.1. Vecinos Norte y Sur

Los vecinos que dependen de las filas son los siguientes:

- **Primera fila (superior):** en esta fila el Norte lo encontramos en la última en fila y nuestra misma columna del toroide, debido a su topología (ver Figura 1). El Sur se encuentra la fila inmediatamente inferior de nuestra columna.
- **Filas centrales:** en estas filas el Norte es el nodo que tenemos en la fila anterior de nuestra misma columna y de forma análoga con el Sur.
- **Última fila (inferior):** está fila es también peculiar, al igual que la primera, ya que el Sur lo tenemos en la primera fila y nuestra misma columna. El Norte lo obtenemos de la fila inmediatamente superior.

Fila	Norte	Sur
0	$(L-1)*L+rank$	$rank+L$
Última	$rank-L$	$rank\,\%L$
Resto	$rank-L$	$rank+L$

Cuadro 1: Fórmulas empleadas para los vecinos Este y Oeste

2.1.2. Vecinos Este y Oeste

Los vecinos que dependen de las columnas son los siguientes:

- **Primera columna (izquierda):** en esta columna el Este lo encontramos en la columna de la derecha y nuestra misma fila. El Oeste lo encontramos en la última columna de nuestra misma fila, debido a la topología (ver Figura 1)
- **Columnas centrales:** en estas columnas el Este se encuentra en la columna de la izquierda y el Oeste en la derecha, siempre de nuestra misma fila.
- **Última columna (derecha):** está columna es también peculiar, al igual que la primera, ya que el Este lo obtenemos de la primera columna y el Oeste a la izquierda, de nuestra misma fila.

Fila	Este	Oeste
0	$rank + 1$	$(L * (fila + 1)) - 1$
Última	$fila * L$	$rank - 1$
Resto	$rank + 1$	$rank - 1$

Cuadro 2: Fórmulas empleadas para los vecinos Este y Oeste

3. Diseño de la solución

3.1. Estructura de la función principal

En esta función se inicializan las variables principales de nuestro programa y se realizan las llamadas al resto de funciones, según la acción que se deba realizar. En cuanto a *MPI* se realizan las siguientes acciones:

- Inicializar la estructura de comunicación de MPI entre los procesos [4].
- Determinar el *rank* (identificador) del proceso que lo llama dentro del comunicador seleccionado [2].
- Determinar el tamaño del comunicador seleccionado, es decir, el número de procesos que están actualmente asociados a este [3].

```
1 int main(int argc, char *argv[]){
2         double *data = malloc(N_NODES * sizeof(double));
3         int length;
```

```
4           int rank, size;
5           double num;
6
7           MPI_Init(&argc, &argv);
8           MPI_Comm_rank(MPI_COMM_WORLD,&rank);
9           MPI_Comm_size(MPI_COMM_WORLD,&size);
10
11          if (rank == 0){
12                  get_data(data, &length);
13
14                  check_data(length, LENGTH_MSG);
15
16                  if (!end) check_data(size, SIZE_MSG);
17
18                  if (!end) send_data(data);
19          }
20
21          /* Get confirmation to continue from first node */
22          MPI_Bcast(&end,1,MPI_INT,0,MPI_COMM_WORLD);
23
24          if(!end){
25                  /* Wait the number */
26                  MPI_Recv(&num, 1, MPI_DOUBLE, 0, MPI_ANY_TAG,
                       MPI_COMM_WORLD, NULL);
27                  calculate_min(rank, num);
28          }
29
30          MPI_Finalize();
31
32          return EXIT_SUCCESS;
33  }
```

A continuación, se van a describir las distintas acciones que se llevan a cabo en el código anterior con las respectivas funciones utilizadas.

3.2. Lectura del fichero

La lectura del fichero para obtener los distintos números se realiza con se realiza con la siguiente función:

```
1   void get_data(double *data, int *length){
2           /* For load data from datos.dat */
3
4           FILE *file;
5           char *aux = malloc(1024 * sizeof(char));
6           char *n;
```

```
7
8           if ((file = fopen(FILE_NAME, "r")) == NULL){
9                   fprintf(stderr, "Error opening file\n");
10                  end = TRUE;
11          }else{
12                  *length = 0;
13
14                  fscanf(file, "%s", aux);
15                  fclose(file);
16
17                  data[(*length)++] = atof(strtok(aux,","));
18
19                  while((n = strtok(NULL, ",")) != NULL)
20                  data[(*length)++] = atof(n);
21          }
22
23          free(aux);
24
25  }
```

Los parámetros de la función son un array de *double* para almacenar los números leídos del fichero y una variable entera que representa la cantidad de números obtenidos. Ambos parámetros se introducen por referencia, ya que las modificaciones las necesitamos posteriormente.

3.3. Tratamiento de errores

Una vez concluida la carga de los datos, se realizarán dos comprobaciones para iniciar los cálculos.

- En primer lugar, se comprueba el tamaño obtenido de MPI (variable ***size***)
- En segundo lugar, si la comprobación anterior ha resultado exitosa se comprueba el número de elementos obtenidos, almacenado en la variable ***length***.

Ambas comprobaciones se realizan con la siguiente función, ya que las dos valores a comprobar se comparan con el número de nodos totales (***LxL***).

```
1   void check_data(int var, char *type){
2           /* For check length or size */
3
4           if (var != N_NODES){
5                   fprintf(stderr, "Error in data %s\n", type);
6                   end = TRUE;
```

```
7           }
8   }
```

3.4. Envío de datos

Las comprobaciones activarán la variable global ***end*** si hay algún valor incorrecto. Si los valores comprobados son correctos se procede a enviar los datos, utilizando la siguiente función.

```
1   void send_data(double *data){
2           /* Send numbers to all the nodes */
3
4           double buff_num;
5           int i;
6
7           for(i=0; i < N_NODES; i++){
8                   buff_num = data[i];
9                   MPI_Send(&buff_num, 1, MPI_DOUBLE, i, 0,
                        MPI_COMM_WORLD);
10                  printf("%.2f sended to %d node\n",buff_num, i);
11          }
12
13          free(data);
14  }
```

En lo relativo a MPI, se hace uso de la función de envío como se puede observar en la línea 9 del bloque de código anterior.

3.5. Recepción de datos

Antes de realizar la recepción de los datos se realiza un envío de la variable de comprobación (***end***) a todos los procesos utilizando la función de *broadcast* de MPI [1]. La razón es que los diferentes procesos no deben recibir sus datos si las comprobaciones no han sido exitosas. Para la recepción del número que le corresponde a cada nodo se ha utilizado la primitiva básica de recepción de MPI [5].

3.6. Cálculo del mínimo número de la red

El objetivo del programa es que todos los nodos de la red se queden con el número mínimo de todo el conjunto de nodos. Para ello, los nodos se deben enviar los distintos

números de unos a otros hasta encontrar el mínimo. Tras esto, el primer nodo (con *rank* 0) imprimirá el número que tiene actualmente, es decir, el mínimo de toda la red.

3.6.1. Implementación de los vecinos

Los distintos nodos de la red enviarán y recibirán números de sus vecinos respectivamente. Es por ello, que lo más importante para que el cálculo del mínimo de la red se realice correctamente es calcular los cuatro vecinos que tiene cada nodo de forma correcta.

El siguiente bloque de código representa la implementación del cálculo de los vecinos para un determinado nodo de la red. La forma de obtener los vecinos se ha explicado de manera teórica en la sección 2.1.

```
void toroidal_neighbors(int rank, int neighbors[]){
        /* Calculate the neighbors */

        int row = rank / L;
        int column = rank % L;

        switch (row){
                /*
                1 2
                | |
                3 4
                */
                case 0: /* Lower row */
                        neighbors[SOUTH] = rank + L;
                        neighbors[NORTH] = (L-1) * L + rank;
                        break;
                case L-1: /* Uppper row */
                        neighbors[SOUTH] = rank % L;
                        neighbors[NORTH] = rank - L;
                        break;
                default: /* Central row */
                        neighbors[SOUTH] = rank + L;
                        neighbors[NORTH] = rank - L;
                break;
        }

        switch(column){
                /*
                1 -- 2
                3 -- 4
                */
                case 0: /* Left column */
                        neighbors[WEST] = (L * (row+1))-1;
                        neighbors[EAST] = rank + 1;
```

```
35                          break;
36                  case L-1: /* Right column */
37                          neighbors[WEST] = rank -1;
38                          neighbors[EAST] = row * L;
39                          break;
40                  default: /* Central column */
41                          neighbors[WEST] = rank -1;
42                          neighbors[EAST] = rank + 1;
43                          break;
44          }
45  }
```

El cálculo se divide en dos fases:

- **Filas:** los vecinos Norte y Sur, implementados en el primer *switch*. Las fórmulas empleadas son las del Cuadro 1
- **Columnas:** los vecinos Este y Oeste, implementados en el segundo *switch* del código anterior. Las fórmulas utilizadas se pueden ver en el Cuadro 2.

La función tiene como parámetro un array, donde se almacenarán los *ranks* de los cuatro vecinos del nodo que haya ejecutado la función. Como los vectores en C se pasan por referencia, los valores de este vector los podremos utilizar después de ejecutar está función.

3.6.2. Envío y recepción de los números

Una vez calculados los vecinos de cada nodo, los procesos comenzarán a enviar su número a sus vecinos y recibir otros valores para comparar el que se recibe con el que se tiene actualmente.

En un principio la implementación se dividió en dos bucles, uno para las filas y otro para las columnas, como se muestra en el siguiente bloque de código:

```
1  void calculate_min(int rank, double num){
2          int neighbors[L];
3          int neighbors[N_NEIGHBORS];
4          double his_num;
5          int i;
6
7          toroidal_neighbors(rank, neighbors);
8
9          for(i=1; i < L; i++){
10                 MPI_Send(&num, 1, MPI_DOUBLE, neighbors[SOUTH], 1,
                       MPI_COMM_WORLD);
```

```
11                      MPI_Recv(&his_num,1,MPI_DOUBLE,neighbors[NORTH],1,
                            MPI_COMM_WORLD,NULL);
12                      num = MIN(num,his_num);
13              }
14
15              for(i=1; i < L; i++){
16                      MPI_Send(&num, 1, MPI_DOUBLE, neighbors[EAST], 1,
                            MPI_COMM_WORLD);
17                      MPI_Recv(&his_num,1,MPI_DOUBLE,neighbors[WEST],1,
                            MPI_COMM_WORLD,NULL);
18                      num = MIN(num,his_num);
19              }
20
21              if(rank == 0){
22                      printf("\nThe minimum number is: %.2f\n",num);
23              }
24
25      }
```

De esta forma, los nodos enviarán y recibirán los números de sus vecinos de filas superiores e inferiores en la misma columna. Con lo cual, en cada nodo quedará finalmente el menor número de cada columna, porque los números se envían de arriba a abajo, como se puede ver en el ejemplo de la Figura 2. Cabe destacar que en la imagen se ha obviado una iteración, ya que al ser un toroide de lado tres se realizarían 2 iteraciones en el bucle.

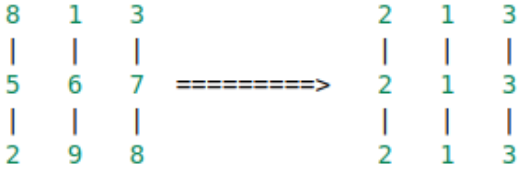

Figura 2: Ejemplo de envío entre nodos Norte y Sur

Con el segundo bucle se conseguiría el mínimo de cada fila, pero esta vez intercambiando números con los vecinos Este y Oeste de cada nodo, que dependen de las columnas. Al finalizar el bucle, en todos los nodos tendríamos el número 1.

Tras estudiar que ambos bucles son análogos, se realizó una implementación en un solo bucle. Realizando el siguiente flujo de eventos:

- Intercambio de números entre vecinos Norte y Sur
- Comparación entre el número actual y el recibido, quedando el mínimo entre ambos.
- Intercambio de números entre vecinos Este y Oeste
- Comparación entre el número actual y el recibido, quedando el mínimo entre ambos.

El resultado es el mismo que en la implementación anterior, pero nos ahorramos ejecutar un bucle. La implementación final se puede observar en el siguiente bloque de código:

```
1  void calculate_min(int rank, double num){
2          /* Calculate the minimum number of all the nodes */
3
4          int neighbors[N_NEIGHBORS];
5          double his_num;
6          int i;
7
8          toroidal_neighbors(rank, neighbors);
9
10         for(i=1; i < L; i++){
11                 /* Rows */
12                 MPI_Send(&num, 1, MPI_DOUBLE, neighbors[SOUTH], 1,
                       MPI_COMM_WORLD);
13                 MPI_Recv(&his_num,1,MPI_DOUBLE,neighbors[NORTH],1,
                       MPI_COMM_WORLD,NULL);
14                 num = MIN(num, his_num);
15
16                 /* Columns */
17                 MPI_Send(&num, 1, MPI_DOUBLE, neighbors[EAST], 1,
                       MPI_COMM_WORLD);
18                 MPI_Recv(&his_num,1,MPI_DOUBLE,neighbors[WEST],1,
                       MPI_COMM_WORLD,NULL);
19                 num = MIN(num, his_num);
20
21         }
22
23         if(rank == 0) printf("\nThe minimum number is: %.2f\n",num)
               ;
24
25 }
```

Referencias

[1] `https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Bcast`. Broadcast mpi.

[2] `https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Comm_rank`. Determinar *rank* mpi.

[3] `https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Comm_size`. Determinar tamaño del comunicador mpi.

[4] `https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Init`. Inicialización mpi.

[5] `https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Recv`. Recepción mpi.

[6] `https://www.conceptdraw.com/How-To-Guide/toroidal-network-topology`. Topología toroidal.