Alberto Vázquez Martínez

Documentación práctica 1 – Diseño de Infraestructura de red

Red toroide

Enunciado del problema

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 distribuirá a cada uno de los nodos de un toroide de lado L, los L x L números 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.

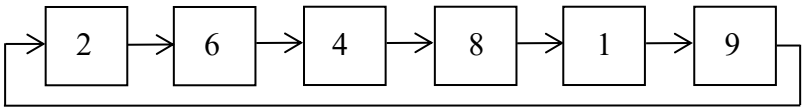
Se pide calcular el elemento menor de toda la red, el elemento de proceso con *rank* 0 mostrará en su salida estándar el valor obtenido. La complejidad del algoritmo no superará $O(\sqrt{n})$ Con n número de elementos de la red.

Planteamiento y diseño de la solución

Lo primero que hay que tener en cuenta para formar la red toroide es si el número de procesos que formarán la red toroide es correcto. Para ello calculamos la raíz cuadrada del número de procesos, si es una raíz exacta, entonces, se podrá formar una red toroide, si no, entonces el programa finalizará y el proceso 0 avisará al resto de procesos para que no continúen con su ejecución.

A continuación, leeremos el archivo *datos.dat* que contiene los números reales. El proceso 0 se encargará de realizar un envío al resto de procesos del número asignado. Los demás procesos recibirán dicho número, incluido el proceso 0, ya que, en esta solución consideraré que el proceso 0 pertenecerá a la red toroide. El envío será asíncrono (MPI_Isend) y la recepción será bloqueante (MPI_Recv) para así asegurarnos de que dato llega a los procesos, ya que, es necesario dicho dato para su correcta ejecución.

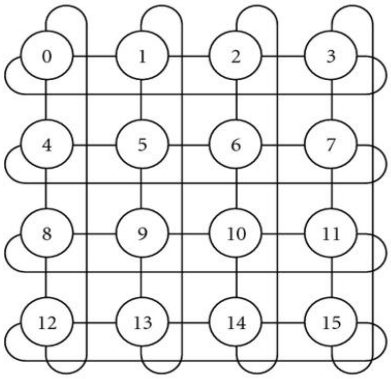
Cada proceso tendrá 4 procesos vecinos, el vecino Norte, el vecino Sur, el vecino Este y el vecino Oeste. Para asignar dichos procesos, se dividirán la red en 2 partes, las filas y las columnas.
Las columnas representarán los procesos Norte y Sur y las filas representarán los procesos Este y Oeste. Tanto para calcular las filas y las columnas se utilizará la misma estrategia, una lista circular utilizando la función del módulo.



En el caso de las filas, se generará la lista correspondiente al proceso, dependiendo del número del proceso y del tamaño del lado del toroide. Por ejemplo: si tenemos proceso con *rank* 3 y nuestro toroide es de lado 4, entonces su lista de filas será: 0,1,2,3,4. En el caso de las columnas no es

necesario generar la lista, ya que podemos calcular directamente los vecinos Norte y Sur con la función del módulo, y el tamaño de la red (número de procesos totales).

Una vez tenemos los vecinos de cada proceso, tendremos que realizar el envío, la recepción y la comparación de cada número. Para ello se utiliza un bucle *for* que irá de 0 hasta la raíz cuadrada de n, siendo n el número de procesos totales. Además, como en esta solución, dividimos la red en filas y columnas, esto estará dentro de un bucle que se repetirá dos veces, una para las filas y otra para las columnas.



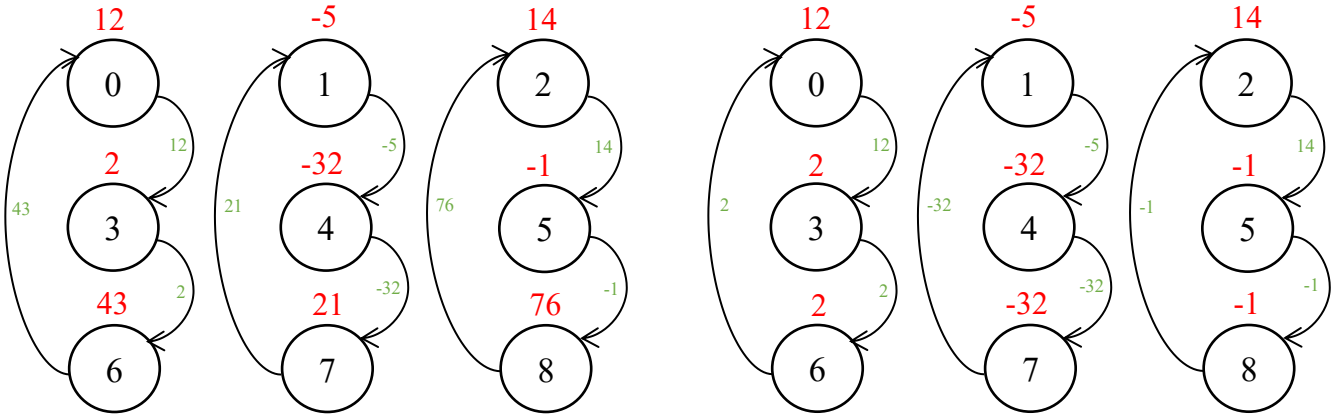
Finalmente, todos los procesos obtendrán el menor número de la lista de números y el proceso 0 lo imprimirá por la salida estándar.

Explicación de flujo de datos

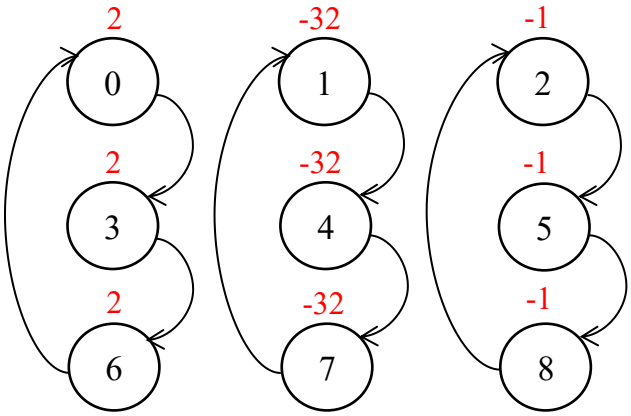
Para ello el proceso enviará a su proceso vecino Sur, su número, y recibirá, del proceso vecino Norte, el número de ese proceso. Este proceso se repetirá dos veces tanto para filas como columnas. Tanto para las filas como para las columnas el proceso se repetirá tantas veces como sea el número de lado del toroide (raíz cuadrada de N, siendo N el número de procesos).

Ejemplo de ejecución con una red toroide de 3x3 (9 procesos)

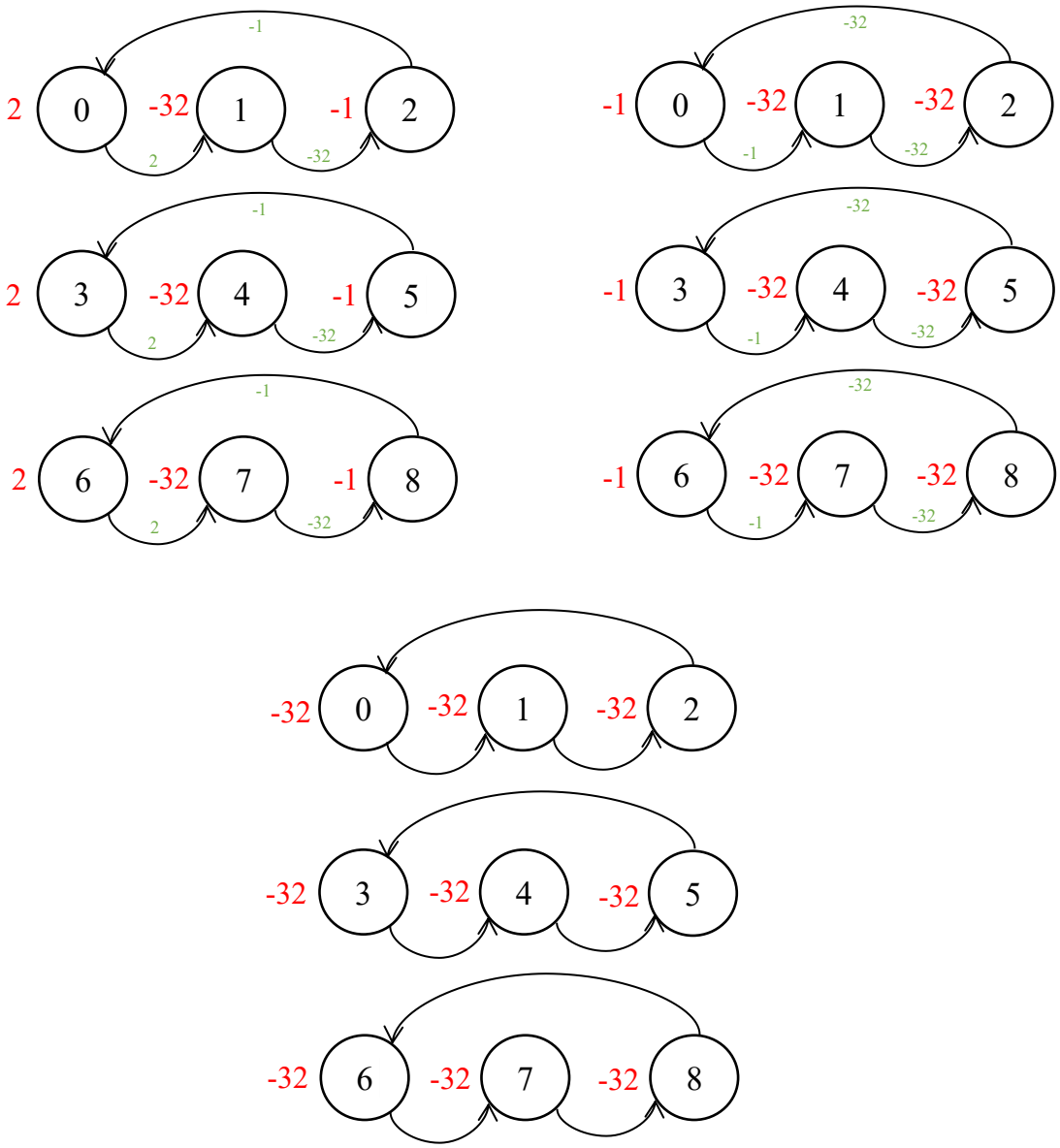
Primero se calculará el valor mínimo de las columnas:



En este momento ya tendremos los valores mínimos de cada columna en cada uno de los procesos.



Procedemos de la misma manera para las filas, pero ahora ya tenemos los valores mínimos de las columnas.



Para el envío y recepción de los datos, he optado por un envío asíncrono (*MPI_Isend y MPI_Irecv*), para así no bloquear a la aplicación con su ejecución, aunque si es necesario un método de sincronización para esperar al dato a recibir, y para ello utilizo *MPI_Test*.

Fuentes del programa

Toroide.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
```

```
#include <definitions.h>
#include "/usr/lib/x86_64-linux-gnu/openmpi/include/mpi.h"

#define NUM_NEIGHBOURS 4

float compare_numbers(float number1, float number2);
void assign_neighbours(int *north_process, int *south_process, int *west_process,
                                                        int *east_process, int l, int rank, int size);
int mod(int a, int b);
float search_min_number(int neighbours[], int size, float send_number);

/* Main function */
int main(int argc, char **argv) {

        int rank, size, truncated, toroidal = false;
        float buf, final_min_number;

        MPI_Status status;
        MPI_Request request;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (rank == 0) {
                double value = sqrt(size);
                truncated = (int) value;

                /*Check if un toroidal network can be possible */
                if (truncated == value) {
                        toroidal = true;
                        send_network_topology_confirmation(toroidal, size, request);

                        FILE* file = open_file("datos.dat","r");
                        read_assign_values(file, size);

                } else {
                        send_network_topology_confirmation(toroidal, size, request);
                        fprintf(stderr, "Error, can't create toroidal network: %d processes\n", size);
                }

        } else {
                MPI_Recv(&toroidal, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        }

        if (toroidal == true) {

                int north_process, south_process, west_process, east_process;

                /* Receive the number of the list */
```

```
		MPI_Recv(&buf, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

		assign_neighbours(&north_process, &south_process, &west_process,
&east_process, sqrt(size), rank, size);

		//printf("Proccess %d: %d %d %d %d\n", rank, north_process, south_process,
east_process, west_process);

		int neighbours[NUM_NEIGHBOURS] = {north_process, south_process,
west_process, east_process};

		final_min_number = search_min_number(neighbours, size, buf);
	}

	if (rank == 0) {
		printf("\n[Process %d] The mininum number is: %.2f\n\n", rank, final_min_number);
	}

	MPI_Finalize();
	return 0;
}

/* Search a min number of column/row */
float search_min_number(int neighbours[], int size, float send_number) {

	MPI_Request request1, request2;
	MPI_Status status;
	int cnt = 0;
	float recv_number;

	for (int j = 0; j < 2; j++) {
		for (int i = 0; i <= sqrt(size); i++) {
			MPI_Irecv(&recv_number, 1, MPI_FLOAT, neighbours[cnt+1],
MPI_ANY_TAG, MPI_COMM_WORLD, &request1);
			MPI_Isend(&send_number, 1, MPI_FLOAT, neighbours[cnt], i,
MPI_COMM_WORLD, &request2);
			MPI_Wait(&request1, &status);

			send_number = compare_numbers(send_number, recv_number);
			recv_number = send_number;
		}
		cnt+=2;
	}

	return recv_number;
}

/* Function to compare which is the smaller number */
float compare_numbers(float number1, float number2) {
```

```
        if (number1 < number2) {
                return number1;
        } else {
                return number2;
        }
}

/* Calculate the module of the divison */
int mod(int a, int b) {
   int r = a % b;
   return r < 0 ? r + b : r;
}

/* Assign neighours of rows to processes */
void assign_neighbours(int *north_process, int *south_process, int *west_process,
                                                        int *east_process, int l, int rank, int size) {
        int row, number_position, j = 0;
        int row_numbers[l];
        row = rank/l;

        for (int i = row*l; i<=((row*l)+l-1); i++) {
                if (rank == i) {
                        number_position = j;
                }
                row_numbers[j++] = i;
        }

        *north_process = mod(rank-l, size);
        *south_process = mod(rank+l, size);
        *west_process = row_numbers[mod(number_position-1, l)];
        *east_process = row_numbers[mod(number_position+1, l)];
}
```

Definitions.h

```
#include "/usr/lib/x86_64-linux-gnu/openmpi/include/mpi.h"
#define BUFFER 2014
#define true 1
#define false 0

/* Open a file with a given permissions and returns a descriptor */
FILE* open_file(char filename[], char *permissions) {
        FILE *file;
        file = fopen(filename, permissions);

        if (file == NULL) {
                fprintf(stderr, "Error opening file '%s': %s\n", filename, strerror(errno));
                exit(EXIT_FAILURE);
        }
        return file;
}
```

```
/* Sends to process != 0 if the topology is correct */
void send_network_topology_confirmation(int network_topology, int size, MPI_Request request) {
   for (int i = 1; i < size; i++) {
      MPI_Isend(&network_topology, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
   }
}

/* Read and assign numbers to child processes*/
void read_assign_values(FILE *file, int size) {
        char buffer[BUFFER];
        float buf;

        fgets(buffer, BUFFER, file);
        char * token = strtok(buffer, ",");

        while ((token != NULL) && (size > 0)) {
                size--;
                buf = atof(token);
                MPI_Bsend(&buf, 1, MPI_FLOAT, size, 0, MPI_COMM_WORLD);
                token = strtok(NULL, ",");
        }
        fclose(file);
}
```

Instrucciones para compilar y ejecutar

La estructura de archivos y directorios es la siguiente:

- Src: código fuente en C.
- Include: archivos cabecera con funciones utilizadas en ambos programas.
- Exec: archivos ejecutables generados en la compilación.
- Datos.dat: Lista de números reales.
- Makefile: utilizado para compilar y ejecutar el código.

Para compilar el programa basta con ejecutar el comando **make**. Se compilarán todos los archivos C del directorio *src*.

Para ejecutar el programa: **make run_toroide**

Para limpiar el directorio *exec* de ejecutables utilizar el comando **make clean**.

Hay que tener en cuenta que el archivo *datos.dat* contiene los números justos para una red máxima de 32 procesos. Por tanto, si se intenta crear una red superior se deberían añadir más números a dicho archivo, de lo contrario, el programa dejará de funcionar.