Red Toroide

1. 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 mostrar en su salida estándar el valor obtenido. La complejidad del algoritmo no superará O (raíz cuadrada(n)) Con n número de elementos de la red.

2. Planteamiento de la solución:

Para la solución la rutina que he seguido ha sido que el proceso 0 se encarga de distribuir los datos y de mostrar que valor es el más pequeño de la red y cada uno de los demás procesos envía su número a uno de sus vecinos y seguidamente espera a que le llegue el valor del vecino opuesto. Una vez que el proceso tiene ambos valores los compara y se queda con el número más pequeño.

Este procedimiento se realiza tantas veces como nodos tenga el toroide en un lado L – 1. En primer lugar, se hace para las columnas y después para las filas.

3. Diseño del programa:

En el programa lo primero que hago es comprobar si los procesos que se lanzan desde el comando de ejecución son suficientes para realizar los cálculos, en caso de que no sea así el programa acaba su ejecución instantáneamente. También cuento los números que hay en el fichero “datos.dat” y si no hay suficientes valores, el programa no se ejecuta.

Si lanzamos los procesos suficientes para realizar los cálculos, en primer lugar, el proceso 0 (manejador) accede al fichero “datos.dat” y lee su contenido hasta tener los valores que necesita. Acto seguido asigna cada valor a un proceso y se queda a la recepción de que el proceso 1 le devuelva el valor del número más pequeño de la red.

Cuando un proceso recibe su dato, obtiene sus vecinos (Norte, Sur, Este y Oeste) para poder realizar las comparaciones. Una vez que obtiene sus vecinos comienza a ejecutar el algoritmo para obtener el menor número tanto de su fila como de su columna.

Una vez han realizado este algoritmo, todos los procesos que lo han completado acaban su ejecución, salvo el proceso con rank 1 que antes de eso, es el encargado de comunicar al proceso 0 cuál es el número más pequeño de la red.

Además, si se lanzan más procesos de los que se necesitan, estos procesos no realizan estos cálculos y finalizan sin realizar ninguna acción.

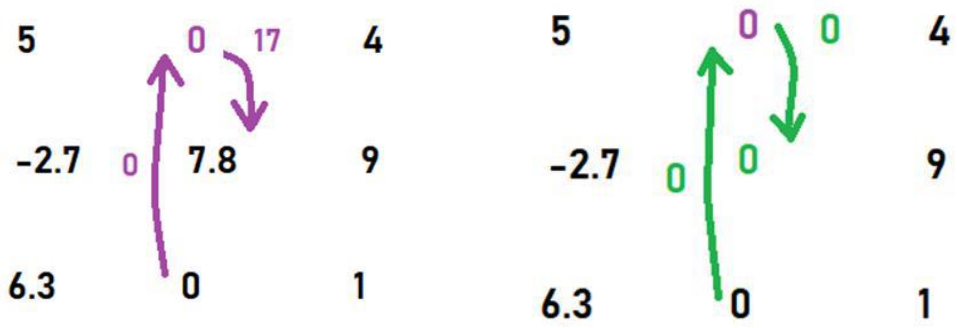
4. Explicación del flujo de datos:

Por ejemplo, supongamos que tenemos una red toroide como la de la imagen.

5	17	4
-2.7	7.8	9
6.3	0	1

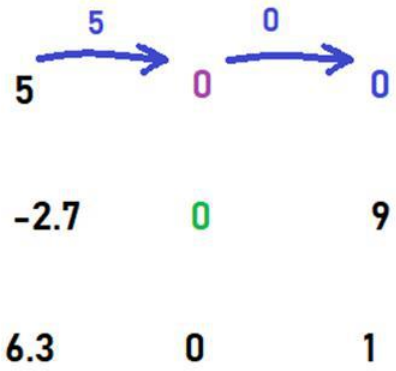
En primer lugar, un proceso (en este caso vamos a usar el proceso con valor 17 como ejemplo) envía su valor a su vecino Sur y espera a recibir el valor de su vecino Norte.

Para el envío se utiliza la instrucción `MPI_Bsend()` para disponer de búferes para evitar un problema de interbloqueos. Para la recepción de valores se usa la instrucción `MPI_Recv()`.



Una vez recibe el valor de su vecino Sur, compara ambos valores y se queda con el valor más pequeño.

Una vez que realiza este procedimiento L – 1 veces, el proceso realiza la misma operación de envío y espera de un valor, pero ahora envía su valor a su vecino Este y espera el valor de su vecino Oeste.



5. Fuentes del programa:

El código del programa es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define MIN(a, b) (((a) < (b)) ? (a) : (b))

#define MANEJADOR 0
#define L 3
#define NUM_NODOS L*L

#define NORTE 0
#define SUR 1
#define ESTE 2
#define OESTE 3


float numeros[NUM_NODOS];
FILE *fp;

int size, rank;
MPI_Status status;

//Cuenta la cantidad de números que hay en el documento 'datos.dat'
int cantidadNumeros(){
   int n = 0;
   float num;
   fp = fopen("datos.dat", "r");
   while((fscanf(fp, "%f,", &num)) != EOF){
      n++;
   }
   return n;
}
//Obtiene los números del documento
void obtenerNumeros(){
   int i=0;
   float num;
   fp = fopen("datos.dat", "r");

   while((fscanf(fp, "%f,", &num)) != EOF){
      if(i<NUM_NODOS){
```

```
            numeros[i] = num;
            i++;
         }else
            break;
         }
      fclose(fp);
}

//Envia los valores a cada proceso
void enviarDatos(){
   int i;
   for(i=0;i<NUM_NODOS;i++){
      MPI_Bsend(&numeros[i], 1, MPI_FLOAT, i+1, i, MPI_COMM_WORLD);
   }
}

//Obtiene los vecinos de cada nodo
void vecinosToroide(int vecinos[]){
   int nodo = rank;
   int fila = (nodo-1)/L;
   int columna = (nodo-1)%L;

   //Calculamos Sur en caso de que nos toque la fila 0
   if (fila == 0){
      vecinos[SUR] = nodo +((L-1)*L);
   }else{
      vecinos[SUR] = nodo-L;
   }

   //Calculamos Norte en caso que nos toque la fila más arriba
   if (fila == L-1){
      vecinos[NORTE] = nodo-(fila*L);
   }else{
      vecinos[NORTE] = nodo+L;
   }
   //Calculamos Oeste en caso de que la columna sea 0
   if (columna == 0){
      vecinos[OESTE] = nodo+(L-1);
   }else{
      vecinos[OESTE] = nodo-1;
   }
   //Calculamos el Este en caso de que sea la columna de más a la
derecha
   if (columna == L-1) {
      vecinos[ESTE] = nodo-(L-1);
   }else{
      vecinos[ESTE] = nodo+1;
   }
}

//Calcula el menor número de toda la red
float calcularMenor(float mi_numero, int vecinos[]){
   int i;
   float su_numero;

   //Envio vertical
      for(i=1;i<L;i++){
         MPI_Bsend(&mi_numero, 1, MPI_FLOAT, vecinos[SUR], i,
MPI_COMM_WORLD);
         MPI_Recv(&su_numero, 1, MPI_FLOAT, vecinos[NORTE],
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

```
            mi_numero = MIN(mi_numero, su_numero);
        }
    //Envio horizontal
    for(i=1;i<L;i++){
        MPI_Bsend(&mi_numero, 1, MPI_FLOAT, vecinos[ESTE], i,
MPI_COMM_WORLD);
        MPI_Recv(&su_numero, 1, MPI_FLOAT, vecinos[OESTE], MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
        mi_numero = MIN(mi_numero, su_numero);
    }
        return mi_numero;
}

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

    float mi_numero;
    int vecinos[4];
    int continua=0;
    int numeros_documento = 0;

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

    numeros_documento = cantidadNumeros();

    if(size<(NUM_NODOS)+1){ //Comprueba que haya los suficientes
procesos para ejecutar
        continua=1;
    }else if(numeros_documento < NUM_NODOS){ //Comprueba que haya los
suficientes valores en 'datos.dat' para ejecutar
        continua=2;
    }

    if(rank==MANEJADOR){
        if(continua==1){ //Comprueba que haya los suficientes procesos
para ejecutar
            fprintf(stderr, "ERROR, no se han lanzado los suficientes
procesos, necesito al menos %d\n", (NUM_NODOS)+1);
        }
        else if (continua==2){ //Comprueba que haya los suficientes
valores en 'datos.dat' para ejecutar
            fprintf(stderr, "ERROR, en el documento 'datos.dat' no hay
suficientes valores, debe haber al menos %d\n", NUM_NODOS);
        }else{
            obtenerNumeros();
            enviarDatos();
            MPI_Recv(&mi_numero, 1, MPI_FLOAT, 1, MPI_ANY_TAG,
MPI_COMM_WORLD,&status);
            printf("El mínimo de la red TOROIDE es %2.2f\n",mi_numero);
        }
    }else{
        if(continua==0 && rank < (NUM_NODOS)+1){ //Comprueba que pueda
ejecutar y que el proceso pertenezca a la red
            MPI_Recv(&mi_numero, 1, MPI_FLOAT, MANEJADOR, MPI_ANY_TAG,
MPI_COMM_WORLD,&status);
            vecinosToroide(&vecinos);
            mi_numero = calcularMenor(mi_numero, &vecinos);
        if(rank==1)
            MPI_Bsend(&mi_numero, 1, MPI_FLOAT, MANEJADOR, 0,
MPI_COMM_WORLD);
```

```
        }
    }
    MPI_Finalize();
    return 0;
}
```

6. Instrucciones de como compilar y ejecutar:

Para ejecutar el programa deberemos utilizar el comando "`mpicc RedToroide.c -o RedToroide`" para hacer uso de las primitivas MPI.

Para ejecutar el programa usaremos el `comando` "`mpirun -n NUM ./RedToroide`" donde *NUM* es el número de procesos que queremos que nuestro programa ejecute.

Para hacer uso del Makefile, basta con escribir "`make compileToroide`" para realizar la compilación y con "`make runToroide`" ejecutamos el programa usando para una red de lado 3 y con 10 procesos para ello.

7. Conclusiones:

La principal conclusión a la hora de realizar un programa con MPI este nos permite realizar varios cálculos en paralelo por lo que nuestro programa tiene una complejidad menor y además ganamos en rendimiento y rapidez.