18-10-2020

PRACTICA 1

Diseño de Infraestructura de Red

César Braojos Corroto

Contenido

1.TOROIDE

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(raíz cuadrada(n)), con n número de elementos de la red.

1.1 Planteamiento de la solución

Para nuestra solución tenemos que tener en cuenta una serie de características:

- La red toroide tiene N elementos donde N= Lado X Lado
- Cada elemento siempre tiene 4 vecinos (Norte, Sur ,Este y Oeste)
- Mi solución basa la cuenta de nodos en el numero 0 , es decir en una red de L=4 y 16 elementos se numerara del 0 al 15.

Diferenciaremos la función respeto al del proceso de la red , el rank 0 tiene una función distinta al resto

RANK 0:

- Cuenta los números que contiene el fichero “datos” y lo envía a los demás procesos
- Controla el numero de procesos que se van a lanzar según el tamaño de la red y comprueba que sea el mismo numero que valores tiene el fichero “datos”
- SI hay algún error en alguna de las condiciones anteriores para la ejecución
- Enviar el dato que le corresponde a los demás procesos
- Muestra el resultado final , es decir , el menor valor de la red .

RESTO DE RANKS:

- Recibe el numero que le envía el rank 0
- Obtiene los números de sus vecinos y así calcula y envía el mínimo valor entres sus vecinos.

POSIBLES ERRORES

- Los procesos que se van a lanzar no corresponden con el numero de nodos de nuestra red.
- El número de valores que contiene el fichero es distinto al numero de nodos que tenemos.

1.2. Diseño del programa

Anteriormente se explico que el programa constaba de 2 funcionalidades dependiendo del rank, por lo tanto lo explicaremos en dos fases .

RANK 0:

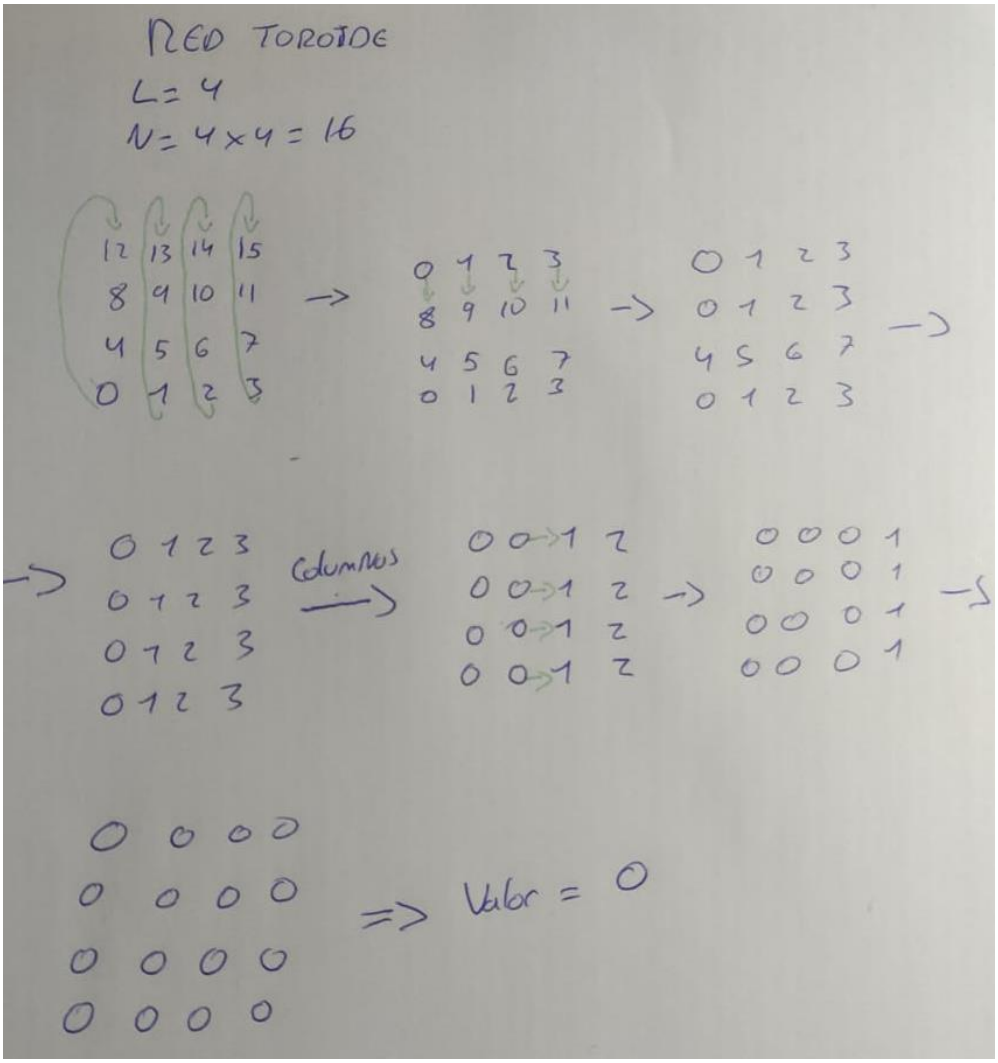
- Obtenemos los datos del fichero, para ello utilizamos la librería **leerfichero.h** la cual nos devuelve la cantidad de números que contiene el fichero.
  Utilizo la función **strtok()** utilizando como delimitador las comas.
- Utilizo la función de MPI **MP_Comm_size (MPI_COMM_WORLD, &size) ,** para obtener el numero de procesos lanzados y poder luego compararlos con el numero de datos o el tamaño de la red .
  Si el numero de procesos no es igual al tamaño de la red , o los procesos que lanzamos no son los mismos que números tenemos en nuestro fichero , el Rank 0 manda al resto de procesos que paren la ejecución mediante la operación **MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD),** donde **error** define si se quiere seguir con la ejecución (0) o no (1).
- Una vez realizadas todas las comprobaciones , el rank 0 procede a enviar a cada uno de los demás nodos el numero que le corresponde (incluyéndose a si mismo), para ello utiliza **MPI_ISEND().**
- Finalmente el rank0 se encarga de mostrar el número mínimo de la red.

RESTO DE RANK:

- Recibimos el numero que nos envio el rank 0 mediante **MPI_Recv()**
- Conocemos a nuestro vecinos mediante la funcione **conocerVecinos().** Esta función divide la tarea en dos partes , en la primera conocemos a los vecinos del norte y del sur , y en la segunda a los vecinos de Este y Oeste.
- Finalment , calculamos el numero mínimo de nuestra red , para ello utilizamos la función **minimo().** Utilizamos **MPI_Send() y MPI_Recv(),** para comunicarnos entre vecinos enviando y recibiendo el numero menor.

1.3. Explicación de flujo de datos MPI

Como vemos en la siguiente imagen , para obtener el numero menor de la red primero miramos las filas, mandamos nuestro numero al vecino del Sur y recibimos el del norte , y comparamos. Después realizamos el mismo proceso por filas , es decir enviamos al Este y recibimos del Oeste.



1.4. Fuentes

TOROIDE.C

```c
/*Red toroide*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include "mpi.h"
#include "leerFichero.h"

#define L 4 //Lado de la red
#define MAX_TAM 1000
#define MAX 1000

void conocerVecinos(int rank, int *norte, int *sur, int *este, int *oeste);
double minimo(int rank, double bufferRank, int norte, int sur, int este, int oeste);

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

    int rank, size;
    double bufferRank;        //buffer que almacena los rank
    int error = 0;            //0  error 1 correcto.
    int numero_Nodos = L * L; //número de procesos
    int norte, sur, este, oeste;

    MPI_Status status;
    MPI_Request request;

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

    //Si soy el rank 0
    if (rank == 0)
    {
        //Se comprueba el numero de procesos
        if (numero_Nodos != size)
        {
            fprintf(stderr, "ERROR. Para un toroide de lado %d se deben lanzar %d procesos\n", L, numero_Nodos);

            //Se para la ejecucion
            error = 1;
            MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);
        }
        else
        {


            double *numleidos;
            numleidos = malloc(MAX_TAM * sizeof(double));

            //Leemos el fichero para obtener los valores
            int valoresFichero = leerFichero(numleidos);

            //Se comprueba que la cantidad de numeros en el fichero es correcta
            if (numero_Nodos != valoresFichero)
            {
                fprintf(stderr, "ERROR. No hay el número correcto de valores en el fichero. Se necesitan %d valores\n", numero_Nodos);

                //Se para la ejecucion
                error = 1;
                MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);
            }
            else
            {
                //Se continua con la ejecucion en el resto de nodos
                MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);

                //El root reparte los valores del fichero para cada uno de los demas nodos
                for (int j = 0; j < valoresFichero; ++j)
                {
                    bufferRank = numleidos[j];
                    MPI_Isend(&bufferRank, 1, MPI_DOUBLE, j, 0, MPI_COMM_WORLD, &request);
                    MPI_Wait(&request, &status);
                }
            }
        }
    }

    //Esperamos que el root nos indique si podemos seguir con la ejecucion.En este MPI_Bcast , bloqueamos la ejecucion
    MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (error == 0)
    {
        MPI_Recv(&bufferRank, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        conocerVecinos(rank, &norte, &sur, &este, &oeste);
        double numeroMinimo = minimo(rank, bufferRank, norte, sur, este, oeste);

        //El rank0 imprime el número menor de la red
        if (rank == 0)
        {
            printf("[RANK %d] El valor mínimo de la red es: %.2f\n", rank, numeroMinimo);
```

```c
    MPI_Finalize();

    return EXIT_SUCCESS;
}
// Método para saber cuales son los vecinos de la red toroide
void conocerVecinos(int rank, int *norte, int *sur, int *este, int *oeste)
{
    int fila = rank / L;
    int columna = rank % L;

    //Para obtener los vecinos sur y norte
    switch (fila)
    {
    case 0: //0
        *sur = (L * (L - 1)) + columna;
        *norte = (L * (fila + 1)) + columna;
        break;

    case L - 1: //3
        *sur = (L * (fila - 1)) + columna;
        *norte = columna;
        break;

    default: //1-2
        *sur = (L * (fila - 1)) + columna;
        *norte = (L * (fila + 1)) + columna;
        break;
    }

    //Para obtener a los vecinos este y oeste
    switch (columna)
    {
    case 0: //0
        *este = (L * fila) + 1;
        *oeste = (L * fila) + (L - 1);
        break;

    case L - 1: //3
        *este = L * fila;
        *oeste = (L * fila) + (columna - 1);
        break;

    default: //1-2
        *este = (L * fila) + (columna + 1);
        *oeste = (L * fila) + (columna - 1);
        break;
    }
```

```c
//Método para obtener el menor número
double minimo(int rank, double bufferRank, int norte, int sur, int este, int oeste)
{
    int i;
    double min;
    MPI_Status status;
    MPI_Request request;

    //Primero obtenemos los minimos de Norte a Sur (por filas)
    for (i = 0; i < L; i++)
    {
        if (bufferRank < min)
        {
            min = bufferRank;
        }
        MPI_Isend(&min, 1, MPI_DOUBLE, sur, i, MPI_COMM_WORLD, &request); //envio mi numero al vecino sur
        MPI_Wait(&request, &status);
        MPI_Recv(&bufferRank, 1, MPI_DOUBLE, norte, i, MPI_COMM_WORLD, &status); //recibo el número de mi vecino norte

        if (bufferRank < min)
        {
            min = bufferRank;
        }
    }

    //Ahora obtengo los valores de mis vecinos de Este a Oeste (por columnas)
    for (i = 0; i < L; i++)
    {
        MPI_Isend(&min, 1, MPI_DOUBLE, este, i, MPI_COMM_WORLD, &request); //envio mi numero al vecino este
        MPI_Wait(&request, &status);
        MPI_Recv(&bufferRank, 1, MPI_DOUBLE, oeste, i, MPI_COMM_WORLD, &status); //recibo el numero de mi vecino oeste

        //obtengo el menor numero
        if (bufferRank < min)
        {
            min = bufferRank;
        }
    }

    return min;
}
```

LIBRERÍA LEERFICHERO.H

```c
/*Lectura del fichero datos.dat*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DATOS "datos.dat"
#define MAX 1000

int leerFichero(double *numeros);

int leerFichero(double *numeros){
    char *listaNumeros=malloc(MAX * sizeof(char));
    int cantidadNumeros=0;
    char *numeroActual;

    FILE *fichero=fopen(DATOS, "r");
    if(!fichero){
        fprintf(stderr,"ERROR: no se pudo abrir el fichero\n.");
        return 0;
    }
    fscanf(fichero, "%s", listaNumeros);
    fclose(fichero);
    numeros[cantidadNumeros++]=atof(strtok(listaNumeros,","));
    while( (numeroActual = strtok(NULL, "," )) != NULL ){
        numeros[cantidadNumeros++]=atof(numeroActual);
    }

    return cantidadNumeros;
}
```

2.HIPERCUBO

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 hipercubo de dimensión D, los 2^D 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, este 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 mayor 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(logaritmo_base_2(n)) con n número de elementos de la red.

2.1 Planteamiento de la solución

Para nuestra solución tenemos que tener en cuenta una serie de características:

- La red hipercubo esta formada por una dimensión D y los elementos son N=2D.
- Cada uno de los elementos de nuestra red tiene D (V_1, V_2, V_3 … V_n)
- Mi solución basa la cuenta de nodos en el numero 0 , es decir en una red de D=3 y 8 elementos se numerara del 0 al 7.

Diferenciaremos la función respeto al del proceso de la red , el rank 0 tiene una función distinta al resto

RANK 0:

- Cuenta los números que contiene el fichero “datos” y lo envía a los demás procesos

- Controla el numero de procesos que se van a lanzar según el tamaño de la red y comprueba que sea el mismo numero que valores tiene el fichero “datos”
- SI hay algún error en alguna de las condiciones anteriores para la ejecución
- Enviar el dato que le corresponde a los demás procesos
- Muestra el resultado final , es decir , el mayorr valor de la red .

RESTO DE RANKS:

- Recibe el numero que le envía el rank 0
- Obtiene los números de sus vecinos y así calcula y envía el máximo valor entres sus vecinos.

POSIBLES ERRORES

- Los procesos que se van a lanzar no corresponden con el numero de nodos de nuestra red.
- El número de valores que contiene el fichero es distinto al número de nodos que tenemos.

2.2. Diseño del programa

Anteriormente se explico que el programa constaba de 2 funcionalidades dependiendo del rank, por lo tanto lo explicaremos en dos fases .

RANK 0:

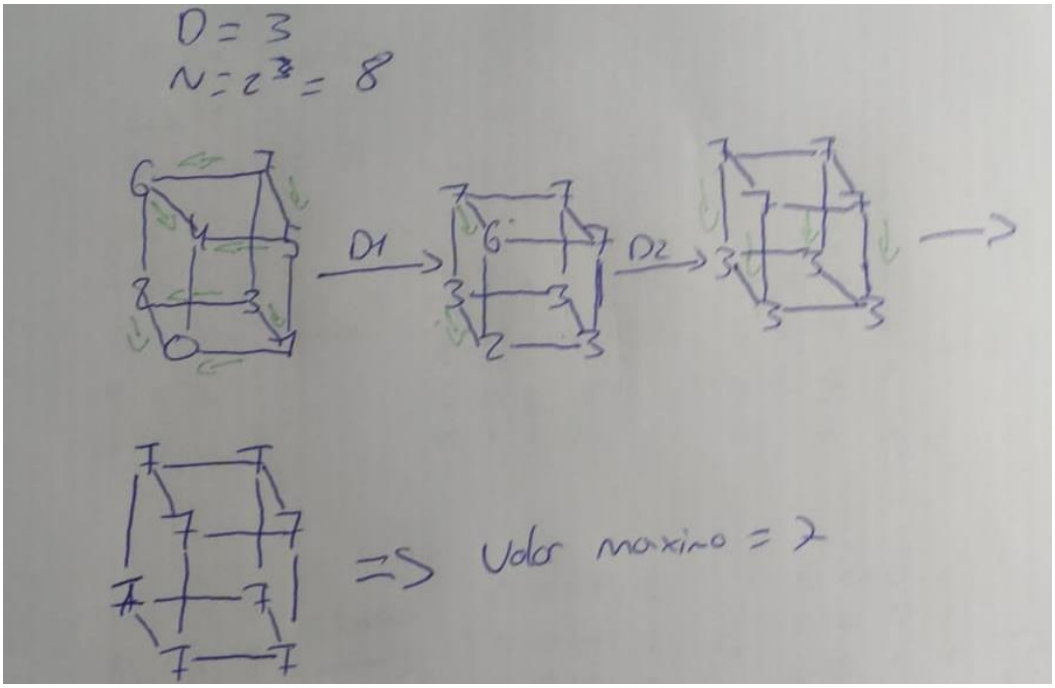
- Obtenemos los datos del fichero, para ello utilizamos la librería **leerfichero.h** la cual nos devuelve la cantidad de números que contiene el fichero.
  Utilizo la función **strtok()** utilizando como delimitador las comas.
- Utilizo la función de MPI **MP_Comm_size (MPI_COMM_WORLD, &size) ,** para obtener el numero de procesos lanzados y poder luego compararlos con el numero de datos o el tamaño de la red .
  Si el numero de procesos no es igual al tamaño de la red , o los procesos que lanzamos no son los mismos que números tenemos en nuestro fichero , el Rank 0 manda al resto de procesos que paren la ejecución mediante la operación **MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD),** donde **error** define si se quiere seguir con la ejecución (0) o no (1).
- Una vez realizadas todas las comprobaciones , el rank 0 procede a enviar a cada uno de los demás nodos el numero que le corresponde (incluyéndose a si mismo), para ello utiliza **MPI_ISEND().**
- Finalmente el rank0 se encarga de mostrar el número maximo de la red.

RESTO DE RANK:

- Recibimos el número que nos envió el rank 0 mediante **MPI_Recv()**
- Conocemos a nuestro vecinos mediante la funcione **ConocerVecinos().** Esta función obtiene los valores de sus D vecinos
- Finalmente , calculamos el número mínimo de nuestra red , para ello utilizamos la función **Maximo().** Utilizamos **MPI_Send() y MPI_Recv(),** para comunicarnos entre vecinos enviando y recibiendo el número mayor.
-

2.3. Explicación de flujo de datos MPI

Como vemos en la siguiente imagen , para obtener el numero máximo de la red , observamos los vecinos de la primera dimensión (los horizontales) y si es necesario intercambiamos los valores , de la misma manera vamos trabajando con las siguientes dimensiones.



2.4. Fuentes

HIPERCUBO.C

```c
/*Red hipercubo*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include <math.h>
#include "mpi.h"
#include "leerFichero.h"
#define D 4 //Dimensión de la red
#define MAX_TAM 1000
#define MAX 1000

void conocerVecinos(int rank, int* vecinos);
double maximo(int rank, double bufferRank, int* vecinos);

int main(int argc, char *argv[]){
    int rank, size;
    int numero_Nodos = pow(2,D); //indica el numero de procesos que vamos a tener dependiendo de la dimension dada
    double bufferRank;
    int error = 0; //0 si error la ejecucion y 1 si no.
    int vecinos[D];
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //Si soy el rank 0
    if(rank == 0){

        if(numero_Nodos != size){

            fprintf(stderr, "ERROR. Para un hipercubo de dimensión %d se deben lanzar %d procesos\n", D, numero_Nodos);
            //error
            error = 1;
            MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);

        }else{

            double *numleidos;
            numleidos = malloc(MAX_TAM * sizeof(double));
            int valFichero = leerFichero(numleidos);

            if(numero_Nodos != valFichero){

                fprintf(stderr, "ERROR. No hay el número correcto de valores en el fichero. Hacen falta %d valores\n", numero_Nodos);
                //error
                error = 1;
```

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

        }else{

            MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);
            //El root reparte los valores
            for(int j = 0; j < valFichero; ++j){
                bufferRank = numleidos[j];
                MPI_Isend(&bufferRank, 1, MPI_DOUBLE, j, 0, MPI_COMM_WORLD,&request);
                MPI_Wait(&request, &status);
            }
        }
    }
    //Esperamos al root
    MPI_Bcast(&error, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if(error == 0){

        MPI_Recv(&bufferRank, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        conocerVecinos(rank, vecinos);
        double numMax = maximo(rank, bufferRank, vecinos);

        if(rank == 0){
            printf("[RANK %d] El valor máximo de la red es: %.2f\n", rank, numMax);
        }
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
// Método obtener los vecinos de la red hipercubo
void conocerVecinos(int rank, int* vecinos){
    int rank_aux;

    for(int i = 0; i < D; i++){
        rank_aux = 1 << i; //movemos el bit i posiciones
        vecinos[i] = rank ^ rank_aux; //hacemos un XOR con cada bit para obtener el vecino
    }
}
```

```
// Método obtener los vecinos de la red hipercubo
void conocerVecinos(int rank, int* vecinos){
    int rank_aux;

    for(int i = 0; i < D; i++){
        rank_aux = 1 << i; //movemos el bit i posiciones
        vecinos[i] = rank ^ rank_aux; //hacemos un XOR con cada bit para obtener el vecino
    }
}

//Método para obtener el número máximo
double maximo(int rank, double bufferRank, int* vecinos){
    int i;
    double Numero_maximo;
    MPI_Status status;
    MPI_Request request;

    for(i = 0; i < D; i++){
        if(bufferRank > Numero_maximo){
            //comparo mi máximo con el valor actual del buffer
            Numero_maximo = bufferRank;
        }

        MPI_Isend(&Numero_maximo, 1, MPI_DOUBLE, vecinos[i], i, MPI_COMM_WORLD,&request);
        MPI_Wait(&request, &status);
        MPI_Recv(&bufferRank, 1, MPI_DOUBLE, vecinos[i], i, MPI_COMM_WORLD, &status);

        if(bufferRank > Numero_maximo){
            //comparo mi máximo con el máximo del vecino
            Numero_maximo = bufferRank;
        }
    }
    return Numero_maximo;
}
```

LIBRERÍA LEERFICHERO.H

```c
/*Lectura del fichero datos.dat*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DATOS "datos.dat"
#define MAX 1000

int leerFichero(double *numeros);

int leerFichero(double *numeros){
    char *listaNumeros=malloc(MAX * sizeof(char));
    int cantidadNumeros=0;
    char *numeroActual;

    FILE *fichero=fopen(DATOS, "r");
    if(!fichero){
        fprintf(stderr,"ERROR: no se pudo abrir el fichero\n.");
        return 0;
    }
    fscanf(fichero, "%s", listaNumeros);
    fclose(fichero);
    numeros[cantidadNumeros++]=atof(strtok(listaNumeros,","));
    while( (numeroActual = strtok(NULL, "," )) != NULL ){
        numeros[cantidadNumeros++]=atof(numeroActual);
    }

    return cantidadNumeros;
}
```

3.Ejecucion

Para la compilación y la ejecución se deben escribir el siguiente comando:

$ make

Si se quiere ver el ejemplo que está en el Makefile se escribirían los siguientes comandos:

$ make testToroide

$ make testHipercubo