PRACTICA 1 MPI:
RED TOROIDE E HIPERCUBO

Sergio Jiménez del Coso

Contenido

1. Red Toroidal.

1.1. Enunciado del ejercicio.

El enunciado consiste en realizar una red toroidal que calcule el valor mínimo de toda la red a través de la lectura de un fichero (datos.dat). Los procesos creados serán el cuadrado del lado del toroide (L^2). Todos los nodos tendrán que ir comparando cada uno de sus valores con los valores de sus vecinos (Norte, Sur, Este y Oeste) y así obtener el valor mínimo de toda la red. La distribución de los datos del fichero se realizará a partir del nodo 0 (root), y en caso de fallo, éste finalizará los procesos de la red.

La complejidad del algoritmo no superará O(raíz_cuadrada(n)), con n número de elementos de la red.

1.2. Planteamiento de la solución.

Para realizar el planteamiento he tenido en cuenta una serie de pasos:

- Para crear los elementos de la red toroidal, definir el valor del lado de la red. Con esto sacamos el número de nodos que es L^2. En el ejemplo que tengo en mi código, tendremos 16 nodos, es decir que el lado del toroide será de 4. Aun así, se puede realizar con cualquier número de nodos.
- Cada nodo tendrá un array de enteros que contendrá los 4 nodos vecinos: Norte, Sur, Este y Oeste. Estos vecinos serán calculados teniendo en cuenta el lado de la red toroidal para establecer un eje de coordenadas y poder obtener el identificador de cada nodo (rank).

Para realizar la tarea del nodo 0, hemos tenido en cuenta los siguientes puntos:

- El nodo 0 tendrá que comprobar que el número de nodos definidos sea igual al número de nodos a partir del lado del toroide. En caso de fallo, éste mandará a cada uno de los nodos, incluyéndose así mismo, un mensaje que indicará que todos los nodos finalizarán su tarea y no podrán continuar con su ejecución.
- Del mismo modo, se comprobará la cantidad de datos que el nodo 0 ha leído del fichero. Si este se ha realizado correctamente, el nodo 0 enviará un valor del fichero a cada uno de los nodos de la red y un mensaje para continuar con la ejecución del programa.

Para realizar las tareas de cualquier nodo:

- Cada nodo recibirá el valor del fichero del nodo 0, y recibirá el valor de la opción si el nodo podrá continuar o no con la ejecución del programa.
- Cada uno de los nodos, mandarán su valor mínimo y obtendrán el valor de sus vecinos. Finalmente obtenemos el valor mínimo con la comparación de los valores de cada uno de sus vecinos.

1.3. Diseño del programa.

Para este apartado vamos a diseñar el pseudocódigo del programa:

```
Para la tarea del nodo 0{
```

Si size es distinto L^2 entonces:
opción→finalizar
MPI_Bcast(opción,comunicador)
else:
leerfichero(datos.dat)
Si numero_leidos distinto size
opción→finalizar:
MPI_Bcast(opción,comunicador)
else:
for each nodo hasta size:
MPI_Send(numero,nodo)
opción→continuar
MPI_Bcast(opción,comunicador)

```
}
Para cualquier nodo:
```

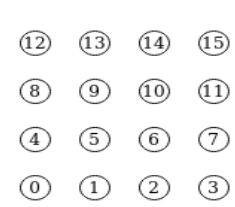
Si opción igual continuar
MPI_Recv(numero,nodo 0)
obtener_vecinos(nodo,vecinos[])
minimo→numero
for each nodo hasta L:
MPI_Isend(minimo,vecino_NORTE)
MPI_Recv(buffer,vecino_SUR)
Si (buffer<minimo) entonces:
minimo→buffer
MPI_Wait(request_isend,status_recv) **Sincronización**

for each nodo hasta L:
MPI_Isend(minimo,vecino_ESTE)
MPI_Recv(buffer,vecino_OESTE)
Si (buffer<minimo) entonces:
minimo→buffer
MPI_Wait(request_isend,status_recv) **Sincronización**

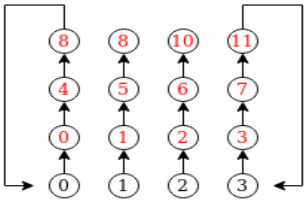
Si rank igual a 0 entonces:
print(minimo)

1.4. Flujo de datos de la red.

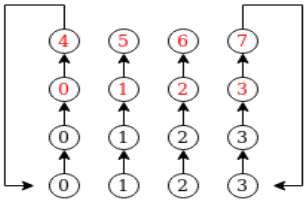
En este apartado describiremos como funciona nuestro programa con un esquema:



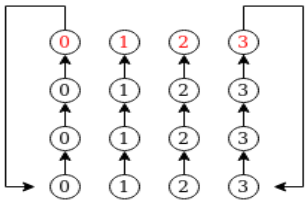
Nodos con datos recibidos



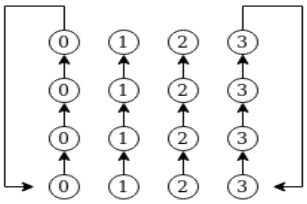
Envío de NORTE y recibir del SUR Iteración 1



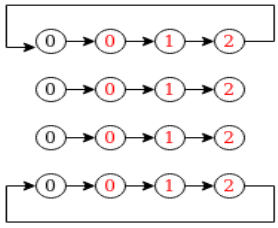
Envío de NORTE y recibir del SUR Iteración 2



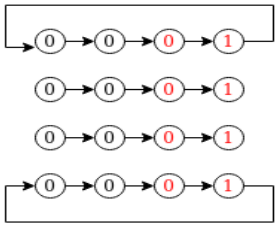
Envío de NORTE y recibir del SUR Iteración 2



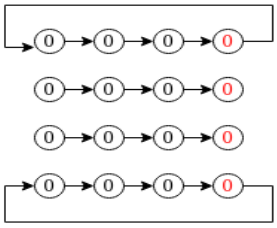
Envío de NORTE y recibir del SUR Iteración 3



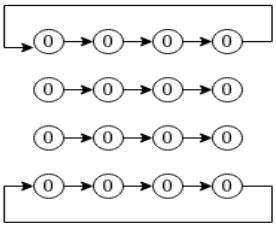
Envío de ESTE y recibir del OESTE Iteración 1



Envío de ESTE y recibir del OESTE Iteración 2



Envío de ESTE y recibir del OESTE Iteración 3



Envío de ESTE y recibir del OESTE Iteración 4

El esquema se divide en dos partes: en la primera parte podemos ver cómo los nodos envían sus valores mínimos a los vecinos NORTE (MPI_Isend)y reciben sus valores del vecino SUR (MPI_Recv). Una vez que hayan recibido el valor de sus vecinos, cada uno de los nodos compara cual es el valor mínimo, si es el del nodo o el valor que ha recibido del nodo SUR. Si hay algún cambio se procederá con la sincronización de dicho valor.

En la segunda parte podemos ver como cada uno de los nodos tienen sus valores actualizados del primer bucle que actualizaron las filas del toroide. Ahora el sentido cambiaría, los nodos enviarán sus valores al vecino del ESTE (MPI_Isend)y recibe del vecino del OESTE (MPI_Recv). El modelo es el mismo que el anterior. Se comparará el valor del nodo con el valor recibido del vecino. Como podemos observar, finalmente todos los nodos tendrán el mismo valor, ya que calculamos el valor mínimo en todos ellos. El nodo 0 dirá cuál es el valor mínimo de la red.

Ambas partes se realizarán de forma no bloqueante, es decir, no se permite reutilizar el buffer hasta que la operación haya terminado. Con esto podemos añadir que la comunicación se realizará inmediatamente sin esperar ningún bloqueo. Para saber que la operación haya finalizado y para la sincronización de datos, utilizamos la función MPI_Wait (), permitiendo asegurar que la operación se haya realizado correctamente.

1.5. Fuentes del programa.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include "mpi.h"

#define archivo "datos.dat"
#define MAX_BUFFER 1000
#define L 4



int leer_fichero(float numeros[]);
void conocer_vecinos(int rank, int vecinos[]);
float minimo(int rank,int vecinos[],float buffer_nodo);

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

    int rank, size; //Definimos las variables rank y size
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int num_leidos,opcion;
    float numeros[L*L];
    //int provisional[L*L] ={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    float buffer_nodo,min_nodo;
    int vecinos[4]; // 0->Norte,1->Sur,2->Este,3->Oeste
    MPI_Status status;
    MPI_Request request;


    if(rank ==0){
        if(size!=L*L){
            opcion = 0; // El rank 0 va finalizar todos los nodos mandando un mensaje a todos los que en el comunicador, MultiCast
            fprintf(stderr,"Error, el número de nodos (%d) no coincide con las dimensiones del Toroide (%d)\n",size,L*L);
            MPI_Bcast(&opcion, 1, MPI_INT,0, MPI_COMM_WORLD);

        }else{

            num_leidos = leer_fichero(numeros); //Leemos los numeros del archivo datos.dat
```

```
            num_leidos = leer_fichero(numeros); //Leemos los numeros del archivo datos.dat

            if(size!=num_leidos){ // El rank 0 va finalizar todos los nodos mandando un mensaje a todos los que en el comunicador, MultiCast
                opcion=0;
                fprintf(stderr,"Error, el número de nodos (%d) no coincide con la cantidad de números leídos (%d)\n",size,num_leidos);
                MPI_Bcast(&opcion,1, MPI_INT,0, MPI_COMM_WORLD);

            }else{

                for (int i=0;i< num_leidos;i++){
                    buffer_nodo = numeros[i];
                    MPI_Send(&buffer_nodo,1,MPI_FLOAT,i,0,MPI_COMM_WORLD); // El rank 0 envía a todos los nodos,
                    //incluyéndose así mismo él mismo, los valores del fichero
                }
                opcion = 1; // El rank 0 evía a todos los procesos un mensaje de continuación, a través de un mensaje MultiCast del comunicador
                MPI_Bcast(&opcion,1,MPI_INT,0, MPI_COMM_WORLD);
            }
        }
    }

    MPI_Bcast(&opcion, 1, MPI_INT, 0, MPI_COMM_WORLD); // Cada nodo recibe el aviso del rank 0, para indicar
    //de que se ha realizado correctamente el envío de datos al fichero

    if(opcion==1){

        MPI_Recv(&buffer_nodo,1,MPI_FLOAT,0,0,MPI_COMM_WORLD,&status);
        conocer_vecinos(rank,vecinos); //Obetenemos los vecinos de cada nodo
        min_nodo = minimo(rank,vecinos,buffer_nodo); // Obtenemos el mínimo de cada nodo

        if (rank==0){

            printf("[RANK %d] El valor minimo es %3f\n",rank,min_nodo); // El rank 0 imprimirá el valor mínimo de toda la red
        }
    }
    MPI_Finalize();
    return 0;
}

```

```
int leer_fichero(float lista_numeros[]){
    FILE* fichero;
    char *buffer = malloc(MAX_BUFFER*sizeof(char));
    int num_numeros=0;
    char *token;
    fichero = fopen(archivo,"r");

    if (fichero == NULL){
        fprintf(stderr,"Error en la lectura del archivo %s\n",archivo);
        exit(EXIT_FAILURE);
    }
    while (fscanf(fichero,"%s",buffer)!=EOF)
    {
    }
    fclose(fichero);

    token = strtok(buffer,",");
    while(token != NULL) {

      lista_numeros[num_numeros] = atof(token);
      token = strtok(NULL,",");
      num_numeros++;
    }
    return num_numeros;

}

void conocer_vecinos(int rank,int vecinos[]){
    int fila = rank/L;
    int columna = rank%L;


    switch (fila) // Saber los vecinos NORTE->0 y SUR->1
    {
    case 0:
        vecinos[0] = rank+L;
        vecinos[1] = (L*(L-1))+rank;
        break;

    case L-1:
        vecinos[0] = columna;
        vecinos[1] = rank-L;
        break;
```

```
125     default:
126         vecinos[0] = rank+L;
127         vecinos[1] = rank-L;
128         break;
129     }
130 
131     switch (columna) //Saber los vecinos ESTE -> 2 y OESTE -> 3
132     {
133     case 0:
134         vecinos[2] = rank+1;
135         vecinos[3] = rank+(L-1);
136         break;
137 
138     case L-1:
139         vecinos[2] = rank-(L-1);
140         vecinos[3] = rank-1;
141         break;
142     default:
143         vecinos[2] = rank+1;
144         vecinos[3] = rank-1;
145         break;
146     }
147 
148 }
149 float minimo(int rank,int vecinos[],float buffer_nodo){
150 
151     MPI_Status status;
152     MPI_Request request;
153     float min_rank;
154     min_rank = buffer_nodo; // Guardamos el valor recibido del rank 0, en el valor mínimo de cada nodo
155         for (int j=0;j<L;j++){
156 
157 
158             MPI_Isend(&min_rank,1,MPI_FLOAT,vecinos[0],10,MPI_COMM_WORLD,&request); //Mandamos a los nodos del NORTE el valor del buffer
159 
160             MPI_Recv(&buffer_nodo,1,MPI_FLOAT,vecinos[1],10,MPI_COMM_WORLD,&status); // Recibimos de los nodos del SUR el valor
161             //del buffer que ellos hayan mandado
162 
163             MPI_Wait(&request,&status); //Garantiza de que el valor haya sido copiado al buffer de
164             //envío y que se haya recibido el valor minimo correctamente y así poder reutilizar el buffer de envío
165             if(buffer_nodo<min_rank){   //Comparamos el valor recibido del SUR (buffer) con el valor mínimo del nodo
166                 min_rank = buffer_nodo;
167             }
```

```
168 
169         }
170 
171         for (int j=0;j<L;j++){
172 
173 
174             MPI_Isend(&min_rank,1,MPI_FLOAT,vecinos[2],32,MPI_COMM_WORLD,&request); //Mandamos a los nodos del ESTE el valor del buffer
175 
176             MPI_Recv(&buffer_nodo,1,MPI_FLOAT,vecinos[3],32,MPI_COMM_WORLD,&status); // Recibimos de los nodos del OESTE el
177             //valor del buffer que ellos hayan mandado
178 
179             MPI_Wait(&request,&status); //Garantiza de que el valor haya sido copiado al buffer
180             //de envío y que se haya recibido el valor minimo correctamente y así poder reutilizar el buffer de envío
181             if(buffer_nodo<min_rank){ //Comparamos el valor recibido del OESTE (buffer) con el valor mínimo del nodo
182                 min_rank = buffer_nodo;
183             }
184 
185         }
186         return min_rank;
```

2. Red Hipercubo

2.1. Enunciado del ejercicio.

El enunciado consiste realizar una red hipercubo que calcule el valor máximo de toda la red a través de la lectura de un fichero (datos.dat). Los procesos creados serán el resultado de dos elevado al número de conexiones que tendrá los nodos o dimensiones que tendrá el hipercubo (2^D). Todos los nodos tendrán que ir comparando cada uno de sus valores con los valores de sus vecinos y así obtener el valor máximo de toda la red. La distribución de los datos del fichero se realizará a partir del nodo 0 (root), y si en caso de fallo éste finalizará los procesos de la red.

La complejidad del algoritmo no superará O(logaritmo_base_2(n)) con n número de elementos de la red.

2.2. Planteamiento de la solución.

Para realizar el planteamiento he tenido en cuenta una serie de pasos:

- Para obtener los elementos de la red hipercubo, definir el valor de dimensiones que tendrá la red, o el número de conexiones que tendrá cada nodo. Con esto sacamos el número de nodos de la red que es 2^D. En el ejemplo que tengo en mi código, tendremos 16 nodos, es decir que el lado de la red será de 4. Aun así, se puede realizar con cualquier número de nodos.
- Cada nodo tendrá un array de enteros que contendrá los D nodos vecinos. Estos vecinos serán calculados teniendo en cuenta el valor del nodo principal. Cuando se quiera calcular el valor de sus vecinos tendremos que cambiar un bit de referencia, es decir hacer un desplazamiento y añadir los bits originales.

Para realizar la tarea del nodo 0, hemos tenido en cuenta los siguientes puntos:

- El nodo 0 tendrá que comprobar que el número de nodos que queremos utilizar sea igual al número de nodos de la red. En caso de fallo, éste mandará a cada uno de los nodos, incluyéndose así mismo, un mensaje que indicará que todos los nodos finalizarán su tarea y no podrán continuar con su ejecución.
- Del mismo modo, se comprobará la cantidad de datos que el nodo 0 ha leído del fichero. Si este se ha realizado correctamente, el nodo 0 enviará un valor del fichero a cada uno de los nodos de la red y un mensaje para continuar con la ejecución del programa.

Para realizar las tareas de cualquier nodo:

- Cada nodo recibirá el valor del fichero del nodo 0, y recibirá el valor de la opción si el nodo podrá continuar o no con la ejecución del programa.
- Cada uno de los nodos, obtendrá el valor de sus vecinos y finalmente obtenemos el valor máximo con la comparación de los valores de cada uno de sus vecinos.

2.3. Diseño del programa.

Para este apartado vamos a diseñar el pseudocódigo del programa:

```
Para la tarea del nodo 0{
        Si size es distinto L² entonces:
                opción→finalizar
                MPI_Bcast(opción,comunicador)
        else:
                leerfichero(datos.dat)
                Si numero_leidos distinto size
                        opción→finalizar:
                        MPI_Bcast(opción,comunicador)
                else:
                        for each nodo hasta size:
                                MPI_Send(numero,nodo)
                        opción→continuar
                        MPI_Bcast(opción,comunicador)
}
```

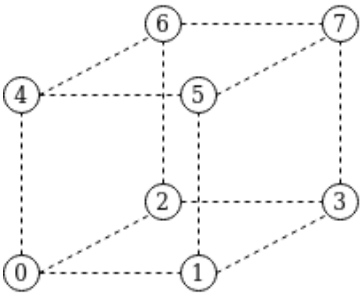
Para cualquier nodo:

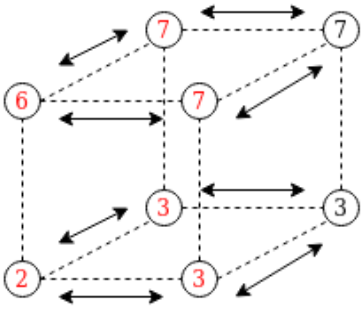
```
Si opción igual continuar
MPI_Recv(numero,nodo 0)
        obtener_vecinos(nodo,vecinos[])
maximo→numero
for each nodo hasta D:
        MPI_Isend(maximo,vecino_NORTE)
        MPI_Recv(buffer,vecino_SUR)
        Si (buffer>maximo) entonces:
                maximo→buffer
        MPI_Wait(request_isend,status_recv) *Sincronización*


Si rank igual a 0 entonces:
        print(maximo)
```

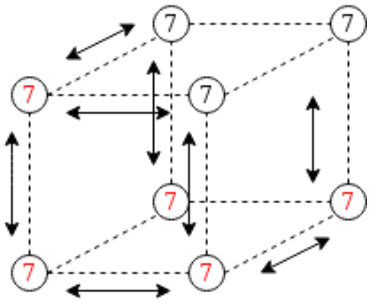
2.4. Flujo de datos de la red.

En este apartado describiremos como funciona nuestro programa con un esquema:





Envío a los vecinos.
Iteración 1



Envío a los vecinos.
Iteración 2 y 3

Como podemos observar, la simulación es mucho más complicada que en la red toroidal ya que deberíamos tener en cuenta la ejecución individual de cada uno de los nodos. Pero la mecánica es siempre la misma, cada uno de los nodos mandará su valor máximo a sus nodos vecinos (MPI_Isend). Éstos lo reciben (MPI_Recv)y hacen la comparación con el valor que ellos tienen guardados, y en el caso de que sea mayor el dato recibido, éste procede a cambiarlo (sincronización y cambio). Cada uno de estos nodos realizará el bucle tantas veces como vecinos tengan. Como podemos observar, ocurre lo mismo que en la red toroidal, todos los nodos finalmente tendrán el máximo valor de la red. Una observación que podemos apreciar es que aquí no debemos tener en cuenta las dimensiones de la red hipercubo, es decir, todas las dimensiones son iguales dependiendo de dónde miremos la perspectiva del hipercubo.

Esta implementación se realizará de forma no bloqueante, es decir, no se permite reutilizar el buffer hasta que la operación haya terminado. Con esto podemos añadir que la comunicación se realizará inmediatamente sin esperar ningún bloqueo. Para saber que la operación haya finalizado y para la sincronización, utilizamos la función MPI_Wait (), permitiendo asegurar que la operación se haya realizado correctamente.

2.5. Fuentes del programa.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include "mpi.h"

#define archivo "datos.dat"
#define MAX_BUFFER 1000
#define D 4



int leer_fichero(float numeros[]);
void conocer_vecinos(int rank, int vecinos[]);
float maximo(int rank,int vecinos[],float buffer_nodo);

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

    int rank, size; //Definimos las variables rank y size
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int num_leidos,opcion,dimension;
    dimension = pow(2,D); // Numero de nodos del hipercubo
    float numeros[dimension]; //Definimos el array de números reales que contendrá el fichero
    float buffer, max_rank;
    int vecinos[D]; //Array de vecinos de cada uno de los nodos
    MPI_Status status;
    MPI_Request request;


    if(rank ==0){
        if(size!=dimension){
            opcion = 0; // El rank 0 va finalizar todos los nodos mandando un mensaje a todos los que en el comunicador, MultiCast
            fprintf(stderr,"Error, el número de nodos (%d) no coincide con las dimensiones del Toroide (%d)\n",size,dimension);
            MPI_Bcast(&opcion, 1, MPI_INT,0, MPI_COMM_WORLD);

        }else{
```

```c
            if(size!=num_leidos){ //El rank 0 va finalizar todos los nodos mandando un mensaje a todos los que en el comunicador, MultiCast
                opcion=0;
                fprintf(stderr,"Error, el número de nodos (%d) no coincide con la cantidad de números leídos (%d)\n",size,num_leidos);
                MPI_Bcast(&opcion,1, MPI_INT,0, MPI_COMM_WORLD);

            }else{

                for (int i=0;i<num_leidos;i++){
                    buffer = numeros[i];
                    MPI_Send(&buffer,1,MPI_FLOAT,i,0,MPI_COMM_WORLD); // El rank 0 envía a todos los nodos,
                    //incluyéndose así mismo él mismo, los valores del fichero


                }
                opcion = 1; // El rank 0 evía a todos los procesos un mensaje de continuación, a través de un mensaje MultiCast del comunicador
                MPI_Bcast(&opcion,1,MPI_INT,0, MPI_COMM_WORLD);
            }
        }
    }
    //Si la opción es 1,continuamos con la ejecución. Si es 0, se anula la ejecución de los nodos
    MPI_Bcast(&opcion, 1, MPI_INT, 0, MPI_COMM_WORLD); // Cada nodo recibe el aviso del rank 0,
    //para indicar de que se ha realizado correctamente el envío de datos al fichero

    if(opcion==1){

        MPI_Recv(&buffer,1,MPI_FLOAT,0,0,MPI_COMM_WORLD,&status); //El nodo recibe el valor enviado por el nodo 0
        conocer_vecinos(rank,vecinos); //Obtenemos los vecinos de cada nodo
        max_rank = maximo(rank,vecinos,buffer); // Obtenemos el máximo de toda la red

        if (rank==0){

            printf("[RANK %d] El valor máximo es %3f\n",rank,max_rank); // El rank 0 imprimirá el valor máximo de toda la red
        }

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

```c
int leer_fichero(float lista_numeros[]){
    FILE* fichero;
    char *buffer = malloc(MAX_BUFFER*sizeof(char));
    int num_numeros=0;
    char *token;
    fichero = fopen(archivo,"r");

    if (fichero == NULL){
        fprintf(stderr,"Error en la lectura del archivo %s\n",archivo);
        exit(EXIT_FAILURE);
    }
    while (fscanf(fichero,"%s",buffer)!=EOF)
    {

    }
    fclose(fichero);

    token = strtok(buffer,",");
    while(token != NULL) {

      lista_numeros[num_numeros] = atof(token);
      token = strtok(NULL,",");
      num_numeros++;
    }
    return num_numeros;

}

void conocer_vecinos(int rank,int vecinos[]){

    for(int i=0;i<D;i++){
        vecinos[i] = rank ^ (1 << i);  //Con el desplazamiento obtenemos el el vecino correspondiente de cada nodo

    }
}
```

```c
float maximo (int rank, int vecinos[],float buffer_nodo){
    float max_rank;
    MPI_Status status;
    MPI_Request request;
    max_rank = buffer_nodo; // Guardamos el valor recibido del rank 0, en el valor máximo de cada nodo
    for (int i=0;i<D;i++){ // Ralizamos un for para poder mandar el valor máximo a cda uno de los vecinos

        MPI_Isend(&max_rank,1,MPI_FLOAT,vecinos[i],D,MPI_COMM_WORLD,&request); //Mandamos a los nodos vecinos el valor del buffer

        MPI_Recv(&buffer_nodo,1,MPI_FLOAT,vecinos[i],D,MPI_COMM_WORLD,&status); // Recibimos de los vecinos el valor del buffer que ellos hayan mandado

        MPI_Wait(&request,&status);
        if(buffer_nodo>max_rank){ //Comparamos el valor recibido del vecino con el del buffer del nodo
            max_rank = buffer_nodo;
        }

    }
        return max_rank;

}
```

3. Instrucciones para ejecutar los programas

Para la compilación de los programas se ejecuta el comando:

$make

Para la ejecución del toroide:

$make run-toriode

Para la ejecución del hipercubo

$make run-hipercubo

4. Conclusiones

Para concluir con esta práctica, pienso que la parte más difícil es ponerse en situación y ver el problema que queremos resolver. A partir de este paso, podemos tener una ligera idea de cómo funciona cada una de la sintaxis de MPI, y sus funciones. Al principio es muy difícil de ver, de cómo funcionan las funciones de MPI, ya que me hacía una idea a la implementación de sockets. Otra de las partes que me ha resultado difícil es la organización del código ya que se trata de varios programas en múltiples nodos. No obstante, he aprendido bastante con esta práctica acerca de que cómo podemos organizar las tareas en varios nodos y así poder obtener un mayor rendimiento de cara al planteamiento de cualquier problema.