2020

TOPOLOGÍAS DE RED

RED TOROIDE Y RED HIPERCUBO

DISEÑO E INFRAESTRUCTURA DE RED

ÁLVARO CERDÁ PULLA

ÍNDICE

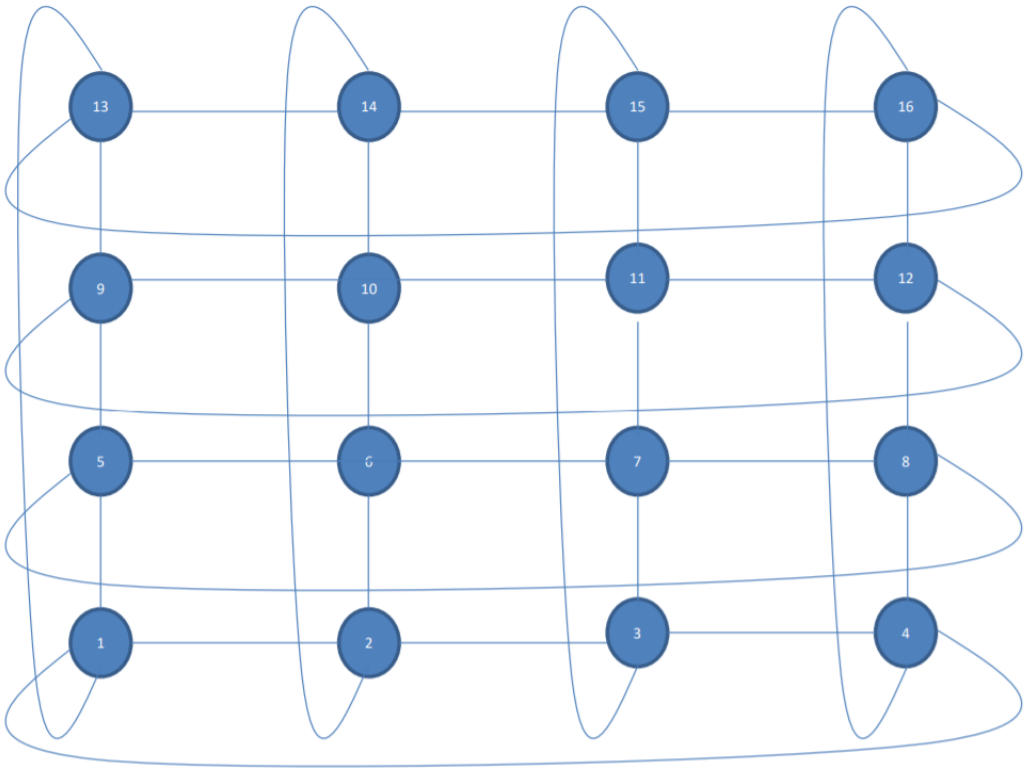
TOROIDE

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 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(raiz_cuadrada(n)) Con n número de elementos de la red.

Planteamiento de la Solución

La red toroide es una red de interconexión entre nodos. El número de nodos que tiene esta red será el lado de la propia red al cuadrado (N = L*L). Todos los elementos tienen 4 vecinos, que consieramos norte, sur, este y oeste. La numeración de los nodos la basamos en 1.



El rank 0, es el que distribuirá a cada uno de los nodos del toroide, por lo tanto, es el primero que debemos controlar.
Lo primero que he tenido en cuenta, han sido las entradas que tendrá mi programa (procesos, lado del toroide) y las excepciones que debo controlar como pide en el enunciado del ejercicio.
A continuación, el rank 0, es el que distribuirá a cada uno de los nodos del toroide, por lo tanto, es el primero que debemos controlar.

Rank 0

- Es el proceso que se encarga de controlar que el número de procesos sea el cuadrado del lado del toroide.
- Es el proceso que lee el fichero y controla que la cantidad de números que haya sea igual al número de procesos que hemos creado previamente. Posteriormente, almacena los números leídos.
- Es el proceso envía los números almacenados a todos los nodos.
- Es el proceso que está encargado de la ejecución del resto y de mostrar la salida final, indicando cual es el número menor.

Resto de Ranks

- Están encargados de recibir los datos enviados por el rank 0.
- Se encargan de obtener los vecinos que están junto a él.
- Son encargados de envíar tanto al este como al norte de sus vecinos.

Diseño del Programa

A la hora de diseñar el programa, he distinguido mediante un condicional el rank 0, del resto de ranks.

- **Rank 0**

Lo primero que hacía si era el rank 0 era contar los datos que tenía el fichero, para saber si había el mismo número de procesos que nodos. Para leer he usado la función strtok().

Posteriormente, controlo el número de procesos que se lanzan mediante otro condicional.

Si alguna de estas excepciones se cumplía, entonces mandaba el mensaje de error junto una señal con la función MPI_Bcast para alertar al resto de ranks.

Cuando he comprobado todo eso, entonces enviaré el contenido del fichero a todos los ranks mediante MPI_Send().

Cuando termina todo, es el encargado de mostrar por pantalla el número mínimo del fichero.

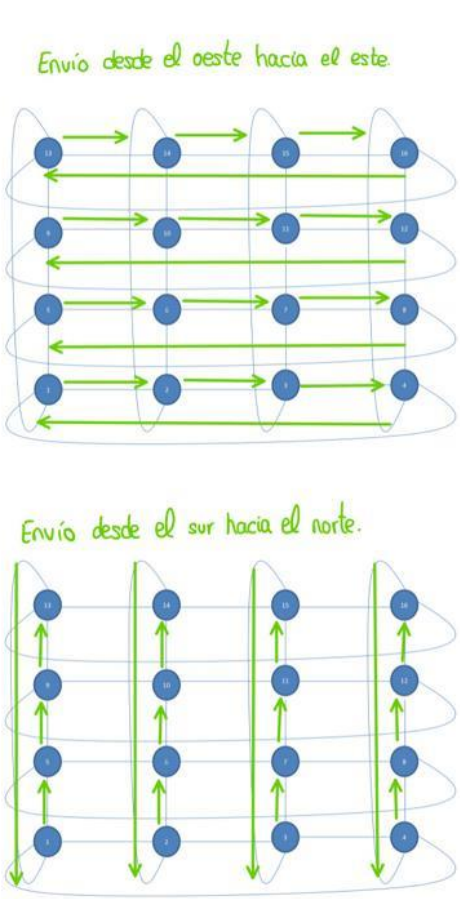
- **Resto de ranks**

Si el estado está OK, es decir, que el rank 0 dice que no hay errores, entonces recibirán los datos del rank 0 que hemos enviado mediante MPI_Recv().

Cuando cada nodo tiene asignado un número, entonces paso a conocer el valor de cada uno de sus vecinos mediante la función obtenerVecinos(). En esta función enviaré los datos desde el oeste al este y desde el sur al norte, controlando siempre los extremos haciendo uso de un condicional que comprueba la fila y la columna.

Para calcular el número mínimo antes de mostrarlo en el rank 0, uso la función enviarVecinos(). En esta función, enviaré mediante Isend(), por lo que haré uso del MPI_Wait(), para asegurar que no se sobreescriba el buffer y perdamos información.

Explicación de flujo de datos en la red de MPI



MPI_Init(&argc, &argv): Inicializo la estructura de comunicación entre los procesos.

MPI_Comm_rank(MPI_COMM_WORLD, &rank): Determina el rango (identificador) del proceso que lo llama dentro del comunicador seleccionado.

MPI_Comm_size(MPI_COMM_WORLD, &size): Determina el tamaño del comunicador seleccionado, es decir, el número de procesos que están actualmente asociados a este.

MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD): Esta función la utilizo para comunicar el estado en el que me encuentro (OK, ERROR) desde el rank 0, al resto de ranks.

MPI_Send(&bufferNumero, 1, MPI_DOUBLE, j, 0, MPI_COMM_WORLD): Envío los datos desde un rank a otro rank.

MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status): Recibo los datos desde un rank (en este caso el rank 0 es que el envía). Lo usaré posteriormente para recibir datos del vecino oeste y a continuación del sur.

MPI_Isend(&minimo, 1, MPI_DOUBLE, vecinos[NORTE], 0, MPI_COMM_WORLD, &request): Envío de forma no bloqueante, copiando en el buffer destinatario el contenido del mensaje. Lo

utilizaré primero para enviar el vecino que está situado al este y posteriormente para el vecino que está al norte.

MPI_Wait(&request, &status): Lo utilizaré de forma que cuando lleguen los datos, entonces podrá continuar, mientras tanto bloqueará la ejecución.

MPI_Finalize(): Termina la ejecución de MPI. Después de esta función, no se podrán enviar mensajes.

Fuentes del programa

He creado una clase funciones.c que contiene todos los métodos que he utilizado, dejando así el main.c limpio.

MAIN

```
35          if (rank == 0)
36          {
37              int cantidadNumeros;
38              double vector[MAX_FILE];
39              cantidadNumeros = leerFichero(vector);
40
41              if(size != lado*lado){
42                  fprintf(stderr, "*****Error*****\nSe esperaban %d procesos para un toroide de lado %d.\nUsted tiene %d procesos\n", lado*lado, lado, size);
43                  estado = ERROR;
44                  MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
45              }else{
46                  if (cantidadNumeros != size)
47                  {
48                      fprintf(stderr, "*****Error*****\nEl fichero contiene %d valores y se necesitan %d.\n",cantidadNumeros, size);
49                      estado = ERROR;
50                      MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
51                  }else{
52                      MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
53
54                      int j;
55
56                      for(j=0; j<cantidadNumeros; j++){
57                      bufferNumero = vector[j];
58                      MPI_Send(&bufferNumero, 1, MPI_DOUBLE, j, 0, MPI_COMM_WORLD);
59                      }
60                  }
61
62              }
63
64
65
66          }
67          MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
67          MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
68          if(estado == OK){
69              MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
70              obtenerVecinos(vecinos, rank, lado);
71              printf("RANK[%d]. Norte: %d\t Este: %d\t Sur: %d\t, Oeste: %d \n", rank, vecinos[NORTE], vecinos[ESTE], vecinos[SUR], vecinos[OESTE]);
72
73
74              minimo = bufferNumero;
75              minimo = enviarVecinos(lado, minimo, bufferNumero, vecinos);
76
77              if (rank == 0){
78                  printf("RANK[%d].El numero minimo es %f\n",rank, minimo);
79              }
80          }
81
82
83          MPI_Finalize();
84          return EXIT_SUCCESS;
85      }
86
87
88
89
```

FUNCIONES

TopologiaPractica1 > Toroide > funciones.c > enviarVecinos(int, double, double, int *)

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

#define DATOS "datos.dat"
#define MAX_FILE 1024


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

MPI_Request request;
MPI_Status status;


int enviarVecinos(int lado, double minimo, double bufferNumero, int *vecinos){
    int a;
    for(a=0; a<lado; a++){
        MPI_Isend(&minimo, 1, MPI_DOUBLE, vecinos[ESTE], 0, MPI_COMM_WORLD, &request);
        MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, vecinos[OESTE], MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);

        if (bufferNumero < minimo)
        {
            minimo = bufferNumero;
        }
    }


    int b;
    for(b=0; b<lado; b++){
    MPI_Isend(&minimo, 1, MPI_DOUBLE, vecinos[NORTE], 0, MPI_COMM_WORLD, &request);
    MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, vecinos[SUR], MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Wait(&request, &status);
        {
            vecinos[OESTE] = rank - (lado - 1);
        }
        else
        {
            vecinos[OESTE] = rank + 1;
        }
        if (f == lado - 1)
        {
            vecinos[SUR] = rank - (f * lado);
        }
        else
        {
            vecinos[SUR] = rank + lado;
        }

    }


    int leerFichero(double *numeros){

        //Vector auxiliar de char para trabajar con los numeros del fichero
        char *listaNumeros=malloc(MAX_FILE * sizeof(char));

        //Tamanio del vector de numeros
        int cantidadNumeros=0;

        //Caracter auxiliar para trabajar con los numeros del fichero
        char *numeroActual;

        //Abrimos el fichero con permisos de lectura
        FILE *fichero=fopen(DATOS, "r");
        //Abrimos el fichero con permisos de lectura
        FILE *fichero=fopen(DATOS, "r");
        if(!fichero){
            fprintf(stderr,"*****Error*****: no se pudo abrir el fichero\n.");
            return 0;
        }
        //Copiamos los datos del fichero al vector auxiliar de char
        fscanf(fichero, "%s", listaNumeros);

        //Cerramos el fichero
        fclose(fichero);

        //Leemos el primer numero hasta la primera coma. Usamos la funcion strtok. Con atof transformamos el string a double
        numeros[cantidadNumeros++]=atof(strtok(listaNumeros,","));

        //Vamos leyendo hasta que no haya mas numeros delante de las comas
        while( (numeroActual = strtok(NULL, "," )) != NULL ){
            //Metemos en el vector el numero correspondiente
            numeros[cantidadNumeros++]=atof(numeroActual);
        }

        free(listaNumeros);
        return cantidadNumeros;
    }
```

Instrucciones de como compilar y ejecutar

Compilar: make compilarToroide

Ejecutar: make ejecutarToroide

Compilar y ejecutar: make all

Dentro del Makefile estará la cantidad de nodos y el lado del toroide para modificarlo.

HIPERCUBO

Enunciado del proyecto

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 Hipercubo de dimensión D, los 2^D 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 mayor de toda la red, el elemento de proceso con rank 0 mostrará en su salida estándar el valor obtenido.

Planteamiento de la Solución

Para esta red de intercomunicaciún, usaremos una dimensión D, que añadiremos como parámetro. El número de elementos total será el resultado de elevar 2 a la dimensión que hayamos escogido. Cada elemento tiene D vecinos.

El rank 0 distribuirá a cada uno de los nodos de un Hipercubo de dimensión D, los 2^D numeros reales que estarán contenidos en el archivo “datos.dat”.

Lo primero que he tenido en cuenta, han sido las entradas que tendrá mi programa (procesos, dimesión del hipercubo) y las excepciones que debo controlar como pide en el enunciado del ejercicio. A continuación, el rank 0, es el que distribuirá a cada uno de los nodos del hipercubo, por lo tanto, es el primero que debemos controlar.

- **Rank 0**

Lo primero que hacía si era el rank 0 era contar los datos que tenía el fichero, para saber si había el mismo número de procesos que nodos. Para leer he usado la función strtok().

Posteriormente, controlo el número de procesos que se lanzan mediante otro condicional.

Si alguna de estas excepciones se cumplía, entonces mandaba el mensaje de error junto una señal con la función MPI_Bcast para alertar al resto de ranks.

Cuando he comprobado todo eso, entonces enviaré el contenido del fichero a todos los ranks mediante MPI_Send().

Cuando termina todo, es el encargado de mostrar por pantalla el número máximo del fichero.

- **Resto de ranks**

Si el estado está OK, es decir, que el rank 0 dice que no hay errores, entonces recibirán los datos del rank 0 que hemos enviado mediante MPI_Recv().

Cuando cada nodo tiene asignado un número, entonces paso a conocer el valor de cada uno de sus vecinos mediante la función obtenerVecinos().

Para calcular el número mínimo antes de mostrarlo en el rank 0, uso la función enviarVecinos(). En esta función, enviaré mediante Isend(), por lo que haré uso del MPI_Wait(), para asegurar que no se sobreescriba el buffer y perdamos información.

Diseño del Programa

A la hora de diseñar el programa, he distinguido mediante un condicional el rank 0, del resto de ranks.

- **Rank 0**

Lo primero que hacía si era el rank 0 era contar los datos que tenía el fichero, para saber si había el mismo número de procesos que nodos. Para leer he usado la función strtok().

Posteriormente, controlo el número de procesos que se lanzan mediante otro condicional.

Si alguna de estas excepciones se cumplía, entonces mandaba el mensaje de error junto una señal con la función MPI_Bcast para alertar al resto de ranks.

Cuando he comprobado todo eso, entonces enviaré el contenido del fichero a todos los ranks mediante MPI_Send().

Cuando termina todo, es el encargado de mostrar por pantalla el número mínimo del fichero.

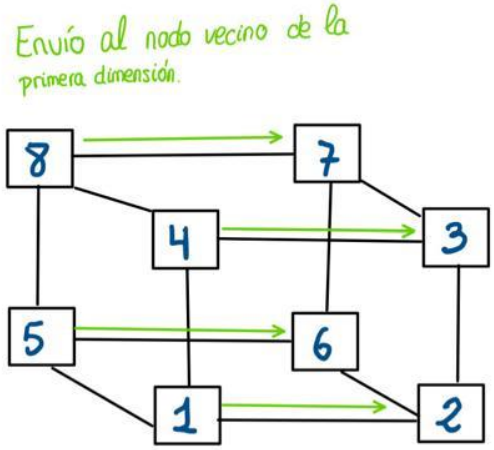
- **Resto de ranks**

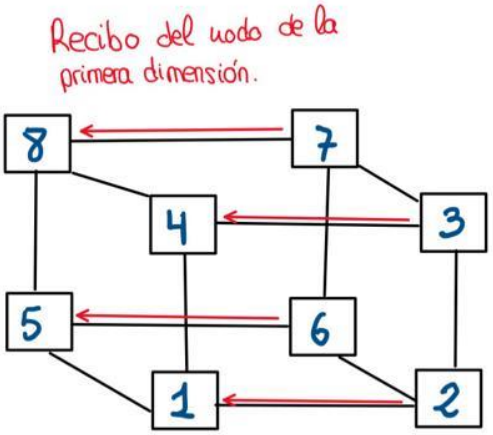
Si el estado está OK, es decir, que el rank 0 dice que no hay errores, entonces recibirán los datos del rank 0 que hemos enviado mediante MPI_Recv().

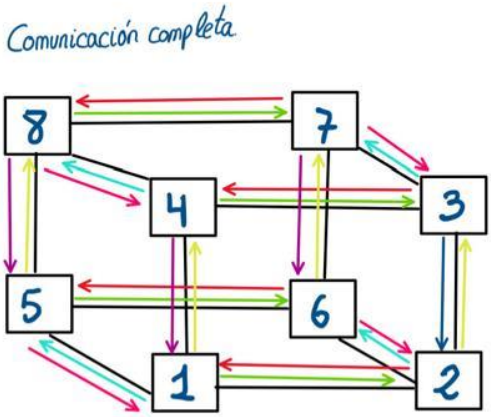
Cuando cada nodo tiene asignado un número, entonces paso a conocer el valor de cada uno de sus vecinos mediante la función obtenerVecinos(). En esta función usaré rank ^ (2,i) para asiganar cada vecino.

Para calcular el número mínimo antes de mostrarlo en el rank 0, uso la función enviarVecinos(). En esta función, enviaré mediante Isend(), por lo que haré uso del MPI_Wait(), para asegurar que no se sobreescriba el buffer y perdamos información.

Explicación de flujo de datos en la red de MPI







MPI_Init(&argc, &argv): Inicializo la estructura de comunicación entre los procesos.

MPI_Comm_rank(MPI_COMM_WORLD, &rank): Determina el rango (identificador) del proceso que lo llama dentro del comunicador seleccionado.

MPI_Comm_size(MPI_COMM_WORLD, &size): Determina el tamaño del comunicador seleccionado, es decir, el número de procesos que están actualmente asociados a este.

MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD): Esta función la utilizo para comunicar el estado en el que me encuentro (OK, ERROR) desde el rank 0, al resto de ranks.

MPI_Send(&bufferNumero, 1, MPI_DOUBLE, j, 0, MPI_COMM_WORLD): Envío los datos desde un rank a otro rank.

MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status): Recibo los datos desde un rank (en este caso el rank 0 es que el envía). Lo usaré posteriormente para recibir datos del vecino oeste y a continuación del sur.

MPI_Isend(&minimo, 1, MPI_DOUBLE, vecinos[NORTE], 0, MPI_COMM_WORLD, &request): Envío de forma no bloqueante, copiando en el buffer destinatario el contenido del mensaje. Lo

utilizaré primero para enviar el vecino que está situado al este y posteriormente para el vecino que está al norte.

MPI_Wait(&request, &status): Lo utilizaré de forma que cuando lleguen los datos, entonces podrá continuar, mientras tanto bloqueará la ejecución.

MPI_Finalize(): Termina la ejecución de MPI. Después de esta función, no se podrán enviar mensajes.

Fuentes del programa

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

#define OK 0
#define ERROR 1

#define MAX_FILE 1024
#define DATOS "datos.dat"

MPI_Status status;
MPI_Request request;

int leerFichero(double *numeros);
void obtenerVecinos(int *vecinos, int rank, int dimension);
void obtenerMaximo(int rank, double bufferNumero, int *vecinos, int dimension, double *maximo);

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

    int rank, size, estado;
    double bufferNumero;
    double numeroMaximo;
    int dimension = atoi(argv[1]);
    int N = (int) round(pow(2,dimension));
    int vecinos[N];

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

    if (rank == 0){

        int cantidadNumeros;
```

```c
    if (rank == 0){

        int cantidadNumeros;
        double vector[MAX_FILE];
        cantidadNumeros = leerFichero(vector);

        if(size != N){
            fprintf(stderr, "*****Error*****\nSe esperaban %d procesos para un toroide de dimension %d.\nUsted tiene %d procesos\n", N, dimension, size);
            estado = ERROR;
            MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
        }else{
            if (cantidadNumeros != size){
                fprintf(stderr, "*****Error*****\nEl fichero contiene %d valores y se necesitan %d.\n",cantidadNumeros, N);
                estado = ERROR;
                MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
            }
            else
            {
                MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);

                    int j;
                    for(j=0; j<cantidadNumeros; j++){
                        bufferNumero = vector[j];
                        MPI_Send(&bufferNumero, 1, MPI_DOUBLE, j, 0, MPI_COMM_WORLD);
                    }
            }

        }

    }

    MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if(estado == OK){
        MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

```
    MPI_Bcast(&estado, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if(estado == OK){
        MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        obtenerVecinos(vecinos, rank, N);
       obtenerMaximo(rank, bufferNumero, vecinos, dimension, &numeroMaximo);


       if(rank == 0){
        printf("RANK[%d]: El numero maximo es %f\n", rank, numeroMaximo);
        }
    }

    MPI_Finalize();
    return EXIT_SUCCESS;
}


void obtenerMaximo(int rank, double bufferNumero, int *vecinos, int dimension, double *maximo){

    int i;
    *maximo = bufferNumero;
    double recibido;

    for(i=0;i<dimension;i++){

        //Si el numero actual es mayor que el maximo
        if(recibido> *maximo){
            *maximo=recibido;

        }
        MPI_Isend(maximo, 1, MPI_DOUBLE, vecinos[i], 0, MPI_COMM_WORLD, &request);
        MPI_Recv(&recibido, 1, MPI_DOUBLE, vecinos[i], MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
        if(recibido>*maximo){
            *maximo=recibido;
        }

        }
}


void obtenerVecinos(int *vecinos, int rank, int dimension){
    int i;
    for(i = 0; i < dimension; i++){
        vecinos[i] = rank ^ (int) pow(2,i);
    }
}


int leerFichero(double *numeros){

        //Vector auxiliar de char para trabajar con los numeros del fichero
        char *listaNumeros=malloc(MAX_FILE * sizeof(char));

        //Tamanio del vector de numeros
        int cantidadNumeros=0;

        //Caracter auxiliar para trabajar con los numeros del fichero
        char *numeroActual;

        //Abrimos el fichero con permisos de lectura
        FILE *fichero=fopen(DATOS, "r");
        if(!fichero){
            fprintf(stderr,"*****Error*****: no se pudo abrir el fichero\n.");
            return 0;
        }
        //Copiamos los datos del fichero al vector auxiliar de char
        fscanf(fichero, "%s", listaNumeros);

        fscanf(fichero, "%s", listaNumeros);

        //Cerramos el fichero
        fclose(fichero);

        //Leemos el primer numero hasta la primera coma. Usamos la funcion strtok. Con atof transformamos el string a double
        numeros[cantidadNumeros++]=atof(strtok(listaNumeros,","));

        //Vamos leyendo hasta que no haya mas numeros delante de las comas
        while( (numeroActual = strtok(NULL, "," )) != NULL ){
            //Metemos en el vector el numero correspondiente
            numeros[cantidadNumeros++]=atof(numeroActual);
        }

        free(listaNumeros);
        return cantidadNumeros;
    }
```

Instrucciones de como compilar y ejecutar

He creado el Makefile de manera que se podra utilizar de la siguiente manera:

Compilar: compilarHipercubo

Ejecutar: ejecutarHipercubo

Compilar y ejecutar: all

Dentro del Makefile estará la cantidad de nodos y la dimensión del hipercubo para modificarlo.

Conclusión

La principal conclusión que he obtenido es que el uso de las topologías de computación distribuida y una interfaz de paso de mensajes puede aumentar en gran medida el rendimiento y la rapidez de una operación siguiendo unos patrones de diseños adecuados.