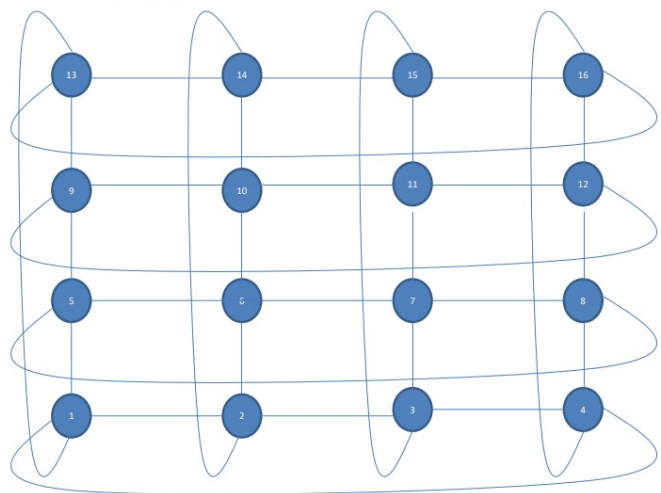


Red Toroide y Red Hipercubo

Práctica 1. Infraestructura de red.



Profesor: Javier Ayllón Pérez – Javier.Ayllon@uclm.es

Trabajo realizado por: Rosa María Sacedón Ortega. - RosaMaria.Sacedon@alu.uclm.es

Índice

- RED TOROIDE
 - Enunciado del problema 3
 - Planteamiento de la solución..... 3
 - Diseño del programa 4
 - Explicación del flujo de datos 8
 - Cómo compilar y ejecutar 8
 - Conclusiones 9
- RED HIPERCUBO 9
 - Enunciado del problema 9
 - Planteamiento de la solución9
 - Diseño del programa 9
 - Explicación del flujo de datos 12
 - Cómo compilar y ejecutar 12
 - Conclusiones 13

1. Red Toroide

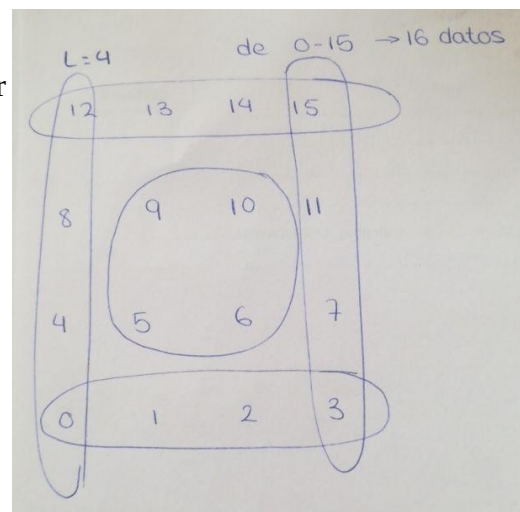
Enunciado

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 $L \times L$ 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 $O(\text{raiz_cuadrada}(n))$ con n igual al número de elementos de la red.

Planteamiento de la solución

La complejidad de la red toroide está en que al tener el algoritmo de búsqueda del número menor. Para conseguir la complejidad buscada en el enunciado utilizaremos las funcionalidades de MPI.

Lo primero que tenemos que hacer es poder identificar los vecinos que tiene cada nodo en nuestra red toroide. Para poder averiguarlos, debemos de tener en cuenta tanto la división del identificador del nodo por el lado (con el cual obtenemos la fila) como el modulo del identificador del nodo por el lado (con este obtenemos la columna). Con el módulo identificamos los vecinos ESTE y OESTE y, mediante la división, obtenemos los vecinos NORTE y SUR.



Otro tema a tener en cuenta en el desarrollo de la práctica es conseguir la complejidad dada en el enunciado para encontrar el menor de todos los números. Se realizará mediante un método específico que realizará envíos y recepciones continuas y a través del cual se consigue que todos los nodos tengan el nodo de menor valor, y éste será impreso por el nodo 0.

Diseño del programa

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

#define MAX_DATOS 1024

#define NORTE 0
#define SUR 1
#define ESTE 2
#define OESTE 3
#define L 4

void calcularNumMinimo(int rank, double num);
int* vecinosToroide(int nodo);
int contar();

int main(int argc, char *argv[]){
    int i=0;
    int* vecinoToroide;
    int nodoT=7;
    int datos;
    int continuar=0;
    double num = 0;
    double new_num = 0;
    double values[L*L];
    double value;
    char* char_fichero;
    char line[80];
    char *token;
    const char separator[2] = ",";
    FILE *archivo;
```

Lo primero, definimos las librerías que nos van a hacer falta para el desarrollo de nuestro programa, los puntos NORTE, SUR, ESTE y OESTE como los identificadores del vector de vecinos, y por último definimos el tamaño del lado del toroide. Definimos las siguientes funciones:

- `calcularNumMinimo()`: calculará el menor número de todos los datos
- `vecinosToroide()`: comprobará los vecinos del toroide
- `contar()`: contará los datos para comprobar si son suficientes para crear la red toroide

A continuación, se declara una serie de datos como un *array* de los nodos vecinos de cada nodo junto con su numero y el nuevo numero que recibirá, los valores que se van a leer del fichero *datos.dat*, el valor que será leído, la línea que se va a leer desde *datos.dat*, el *token* que se creara al partir la línea con el *separator* (,) y el archivo que se leerá, los datos que contiene y, por último, el

Práctica 1. Infraestructura de red. Red toroide y red hipercubo

entero *continuar* que hace referencia a que todos los datos son correctos y se puede continuar con la ejecución del programa.

```
MPI_Init(&argc,&argv);
MPI_Status status;
int size,rank;

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Se iniciará el programa MPI (*MPI_Init()*) y se inicializan el rank de cada nodo y el tamaño de nodos del que disponemos (*MPI_Comm_rank()* y *MPI_Comm_size()*).

```
if(rank==0){
    datos=comprobarFichero();

    if(datos>size){
        continuar=1;
        printf("El numero de nodos a ejecutar es menor que el numero de datos\r\n");
        MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
    }else{
        if(L*L==size){
            continuar=0;
            printf("El numero de nodos a ejecutar correcto\r\n");
            MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
        }else{
            continuar=1;
            printf("El numero de nodos ejecutados es distinto a los nodos requeridos\r\n");
            MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
        }
    }

    if(continuar==0){

        if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
            printf("Error al abrir el fichero\r\n");
            MPI_Finalize();
            exit(EXIT_FAILURE);
        }

        char_fichero=fgets(line,MAX_DATOS*sizeof(char),archivo);

        token = strtok(line, separator);
```

El código continua con una estructura *if* en la que hacemos que si el *rank* del proceso es 0 entonces se lanzará el contador de datos, si estos datos resultan ser mayores que el número de nodos que se han lanzado, el programa se detendrá, enviará a todos los procesos un valor por la primitiva *Bcast* para que no continúen el programa. En el caso de que se comprobara que los datos son suficientes para rellenar nuestra red toroide, continuará la ejecución normal por un mensaje por la primitiva *Bcast*.

Práctica 1. Infraestructura de red. Red toroide y red hipercubo

```
token = strtok(line, separator);

while( token != NULL ) {
    values[i++] = atof(token);
    token = strtok(NULL, separator);
}

fclose(archivo);

for(i=1; i<(L*L); i++){
    num = values[i];
    MPI_Bsend(&num, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
}
num = values[0];
vecinoToroide = vecinosToroide(rank);
calcularNumMinimo(rank, num);
}
}else{

    MPI_Bcast(&continuar, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if(continuar == 0){

        MPI_Recv(&num, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        vecinoToroide = vecinosToroide(rank);
        calcularNumMinimo(rank, num);
    }
}

MPI_Finalize();

return EXIT_SUCCESS;
}

void calcularNumMinimo(int rank, double num){
    int i;
    int *vecinoToroide;
    MPI_Status status;
    vecinoToroide = vecinosToroide(rank);
    double new_num;

    for (i=0; i<L; i++){
        MPI_Bsend(&num, 1, MPI_DOUBLE, vecinoToroide[NORTE], 0, MPI_COMM_WORLD);
        MPI_Recv(&new_num, 1, MPI_DOUBLE, vecinoToroide[SUR], MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if(new_num < num){
            num = new_num;
        }
    }
    for (i=0; i<L; i++){
        MPI_Bsend(&num, 1, MPI_DOUBLE, vecinoToroide[ESTE], 0, MPI_COMM_WORLD);
        MPI_Recv(&new_num, 1, MPI_DOUBLE, vecinoToroide[OESTE], MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if(new_num < num){
            num = new_num;
        }
    }

    if(rank == 0){
        printf("El número mínimo es: %f\n", num);
    }
}

int* vecinosToroide(int nodo){

    static int vecino [L];

    int modulo = nodo % L;
```

Práctica 1. Infraestructura de red. Red toroide y red hipercubo

Si la ejecución del programa continúa, entonces el nodo 0 es el encargado de leer los datos del fichero y enviarlos mediante el uso de la primitiva MPI_Bsend al resto de los nodos que conforman la red. El nodo 0 también debe quedarse con un número asignado, entonces una vez los ha asignado todos los números al resto de nodos, él se asigna uno y puede empezar a averiguar quiénes son sus vecinos. Una vez que ha conocido a sus vecinos, envía su número con la primitiva MPI_Bsend a su vecino del NORTE, recibiendo su vecino del SUR con el uso de la primitiva MPI_Recv. Si el número que recibe es menor que el número que él tenía, entonces lo reemplazará para quedarse con el menor de los dos números. Se realiza el mismo proceso con los vecinos de ESTE y OESTE, consiguiendo así quedarse el proceso 0 con el número de menor valor, que será el que imprimirá.

```
int* vecinosToroide(int nodo){
    static int vecino [L];

    int modulo = nodo%L;
    int division = nodo/L;

    /*Localizar vecino de ESTE y OESTE*/
    if(modulo == L-1) {
        vecino[ESTE]=nodo-1;
        vecino[OESTE]=nodo-(L-1);
    } else if(modulo == 0) {
        vecino[ESTE]=nodo+(L-1);
        vecino[OESTE]=nodo+1;
    } else {
        vecino[ESTE]=nodo-1;
        vecino[OESTE]=nodo+1;
    }

    /*Localizar vecino de NORTE y SUR*/
    if(division == L-1) {
        vecino[SUR]=nodo-L;
        vecino[NORTE]=nodo-(L*(L-1));
    } else if(division == 0) {
        vecino[SUR]=nodo+(L*(L-1));
        vecino[NORTE]=nodo+L;
    } else {
        vecino[SUR]=nodo-L;
        vecino[NORTE]=nodo+L;
    }

    return vecino;
}
```

En la función vecinosToroide() se trata el caso del resto de nodos que no son el de rango 0. Éstos reciben un número de si deben continuar con la ejecución del programa, siendo en caso afirmativo cuando distribuirán su número entre los vecinos y se quedarán con el menor de ellos. Para poder conocer los vecinos de un nodo, se hará utilizando el módulo en el caso de los de ESTE y OESTE y la división en caso de NORTE y SUR. En el caso del módulo se utiliza el rank del nodo menos el tamaño del lado menos 1 ya que estaríamos en la cuarta columna, pero también puede darse el caso de que el módulo da 0, entonces se realizan las mismas operaciones pero sumando 1 en lugar de restándoselo debido a que nos encontramos en la primera columna. Al utilizar la división, si el resultado da 3 nos encontramos en la primera fila y tratamiento de dicha fila será el nodo restado a la multiplicación del lado multiplicado por el lado menos uno. Si la división diese 0, nos encontramos en la última fila y se realiza el mismo procedimiento que para la primera cambiando únicamente la resta por una suma.

```
int contar(){
    int datos=0;

    char* char_fichero;
    char line[80];
    char *token;
    const char separator[2] = ",";
    FILE *archivo;

    if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
        printf("Error al abrir el fichero\n");
        exit(EXIT_FAILURE);
    }

    char_fichero=fgets(line,MAX_DATOS*sizeof(char),archivo);

    token = strtok(line, separator);

    while( token != NULL ) {
        datos++;
        token = strtok(NULL, separator);
    }

    fclose(archivo);

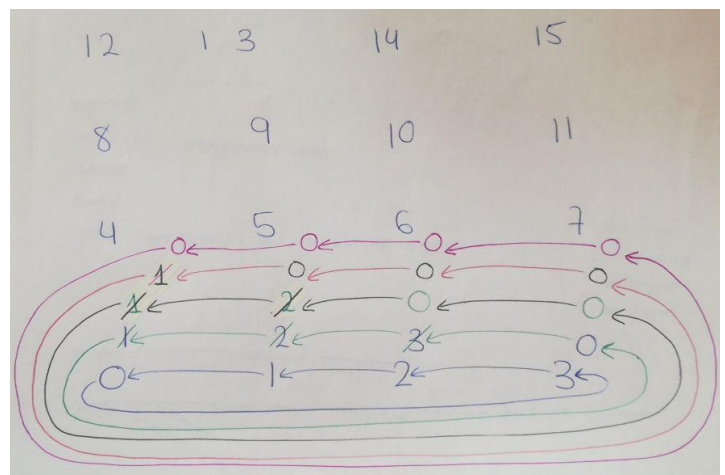
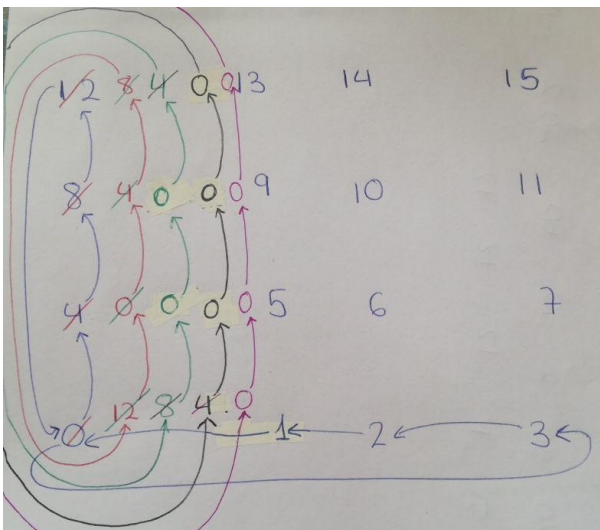
    return datos;
}
```

Finalmente, en la función contar() se comprueba la cantidad de datos leídos del fichero datos.dat.

Explicación del flujo de datos

El flujo de datos comienza en el nodo 0 enviando un mensaje Bcast a todos los demás nodos, indicándoles si estos deben o no seguir con la ejecución del programa. Si deben de seguir ejecutándose, el nodo 0 lee todos los números del fichero y los distribuye de manera que a cada nodo se le asigne un número, incluyéndose a él mismo. A continuación, comienza el flujo de datos para poder encontrar el menos de todos los números del fichero.

El flujo de datos lo he represando tal y como se nos explicó en clase para que sea una manera más sencilla de entender. Los nodos envían el número que se les ha asignado a cada uno de sus vecinos superiores (NORTE) y reciben el número que tienen asignado sus vecinos inferiores (SUR). Si el número que reciben es menor al que ellos tenían, se quedan con el que les ha enviado su vecino. De la misma manera sucede con sus vecinos laterales (ESTE/OESTE). Una vez finalizado este envío y recepción de los números que tenían asignado, saldrá como resultado el número de menor valor que tuviese el fichero.



Cómo compilar y ejecutar

La compilación del programa se realizará mediante el *makefile* que se encuentra en la carpeta del proyecto. En el archivo *datos.dat* introduciremos 16 números aleatorios para poder comprobar que la práctica funciona correctamente.

- Compilar: usaremos el comando *make all*
- Ejecutar: usaremos el comando *maketestToroide*

Práctica 1. Infraestructura de red. Red toroide y red hipercubo

```
rosa@rosa-YOGA:~/Escritorio/ing_informatica/INGENIERIA DE COMPUTADORES/infraestructura/Practica 1$ make all
mpicc src//RedToroide.c -o obj//RedToroide.o
mpicc src//RedToroide.c -o exec//RedToroide
mpicc src//RedHiper cubo.c -o obj//RedHiper cubo.o
mpicc src//RedHiper cubo.c -o exec//RedHiper cubo
rosa@rosa-YOGA:~/Escritorio/ing_informatica/INGENIERIA DE COMPUTADORES/infraestructura/Practica 1$ make testToroide
mpirun -n 16 ./exec/RedToroide
El numero de nodos a ejecutar correcto
Numero minimo -100.100000
rosa@rosa-YOGA:~/Escritorio/ing_informatica/INGENIERIA DE COMPUTADORES/infraestructura/Practica 1$
```

Conclusiones

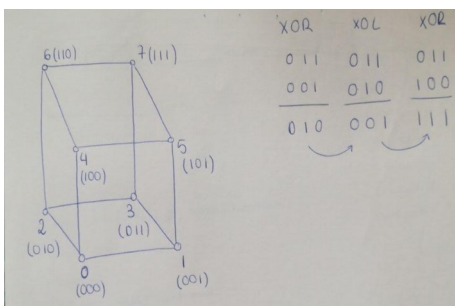
Con la realización de dicha práctica he podido comprobar el funcionamiento el control que podemos tener sobre la red gracias a MPI. El alto grado de eficiencia de MPI es muy interesante ya que sin este la eficiencia se reduciría en L veces.

2. Red Hiper cubo

Enunciado

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 Hiper cubo 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, 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. La complejidad del algoritmo no superará $O(\logaritmo_base_2(n))$ con n igual al número de elementos de la red.

Planteamiento de la solución



En el hipercubo se trabaja con los nodos a nivel de bit, para poder identificarlos ya que un nodo solo se diferencia de sus vecinos en un bit. Así, gracias a la función XOR, podemos conseguir los vecinos de un nodo dado.

Para poder conseguir la complejidad necesaria se realizan una serie de envíos y recepciones de los nodos vecinos para encontrar el número mayor del fichero. Esto está implementado en la función `calcularNumMaximo()`. Así todos los nodos se quedan con el mayor valor de los números que han obtenido y lo imprime el nodo de rango 0.

Diseño del programa

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

#define MAX_DATOS 1024
#define D 3

int* vecinosHipercubo(int nodo);
double maximo(double a, double b);
int contador();
void calcularNumMaximo(int rank, double num);
```

Se define el tamaño del lado del hipercubo y, a su vez, definimos las funciones que vamos a utilizar:

- vecinosHipercubo(): comprobará los vecinos del hipercubo
- contador(): comprobará si son suficientes los datos para crear la red hipercubo
- calcularNumMaximo(): calculará el mayor número

de todos los datos del fichero.

- Maximo():

```
void calcularNumMaximo(int rank, double num);

int main(int argc, char *argv){
    int i=0;
    int nodoHiper=0;
    int* vecinoHipercubo;
    double num = 0;
    double new_num = 0;
    int tamVect=pow(2,D);
    double values[tamVect];
    char* char_fichero;
    double value;
    char line[80];
    char *token;
    const char separator[2] = ",";
    FILE *archivo;
    int indice_char=0;
    MPI_Status status;
    int size,rank;
    int datos;
    int continuar=0;

    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Declaramos un vector de vecinos de un nodo del hipercubo, el num de cada nodo, el new_num que será el número recibido de cada vecino, el vector de valores, el valor, el carácter y la línea que se leerá desde el fichero, el token que contiene los valores de la línea separados por el separator (,), el fichero y tanto el rank como el size.

Práctica 1. Infraestructura de red. Red toroide y red hipercubo

```
if(rank==0){
    datos=contador();

    if(datos>size){
        continuar=1;
        printf("El numero de nodos a ejecutar es menor que el numero de datos\n");
        MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
    }else{
        if(pow(2,D)==size){
            continuar=0;
            printf("El numero de nodos a ejecutar correcto\n");
            MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
        }else{
            continuar=1;
            printf("El numero de nodos ejecutados es distinto a los nodos requeridos\n");
            MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
        }
    }

    if(continuar==0){

        if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
            printf("Error al abrir el fichero\n");
            exit(EXIT_FAILURE);
        }

        char_fichero=fgets(line,MAX_DATOS*sizeof(char),archivo);

        token = strtok(line, separator);

        while( token != NULL ) {
            values[i++]=atof(token);
            token = strtok(NULL, separator);
        }
    }
}
```

El proceso rank 0 es el encargado de comprobar el tamaño y de decir a los demás nodos que conforman la red hipercubo si deben o no continuar con su ejecución normal del programa. Si continúa su ejecución, el proceso 0 tiene que leer los datos del fichero datos.dat y distribuirlos entre los demás nodos. A su vez, este proceso envía y recibe los números de sus vecinos y los compara con los suyos para poder quedarse con el número que mayor valor tenga.

```
fclose(archivo);

for(i=1;i<tamVect;i++){
    num=values[i];
    MPI_Bsend(&num,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
}
num=values[0];

vecinoHiper cubo=vecinosHiper cubo(rank);
calcularNumMaximo(rank, num);
}
}else{
    MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
    if(continuar==0){

        MPI_Recv(&num,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);

        vecinoHiper cubo=vecinosHiper cubo(rank);
        calcularNumMaximo(rank, num);
    }
}

MPI_Finalize();

return EXIT_SUCCESS;
}
```

En el resto de los nodos, se recibirá el número que les asigne el proceso rank 0 y realizarán la misma tarea de enviar y recibir para poder quedarse también con el número de mayor valor. Una

Práctica 1. Infraestructura de red. Red toroide y red hipercubo

vez que todos los nodos se han quedado con el número de mayor valor, será el proceso 0 el que imprima dicho número.

```
void calcularNumMaximo(int rank, double num){
    int i;
    int *vecinoHiper cubo;
    MPI_Status status;
    vecinoHiper cubo = vecinosHiper cubo(rank);
    double new_num;

    for (i=0;i<D;i++){
        MPI_Bsend(&num,1,MPI_DOUBLE,vecinoHiper cubo[i],0,MPI_COMM_WORLD);
        MPI_Recv(&new_num,1,MPI_DOUBLE,vecinoHiper cubo[i],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        num=maximo(new_num,num);
    }

    if(rank==0){
        printf("El numero maximo es: %.2f\n",num);
    }
}

double maximo(double a, double b){
    if(a>b){
        return a;
    }
    else{
        return b;
    }
}
```

Esta función es la encargada de poder calcular el número de mayor valor utilizando las primitivas MPI_Bsend para el envío y MPI_Recv para la recepción. La función máximo que se observa al final es la encargada de poder comparar dos números para poder devolver el de mayor valor.

```
int* vecinosHiper cubo(int nodo){
    static int vecinos [D];
    int i=0;
    int desplazamiento=1;

    for (i=0;i<D;i++){
        vecinos[i]=nodo^desplazamiento;
        desplazamiento=desplazamiento<<1;
    }
    return vecinos;
}

int contador(){
    int datos=0;
    char* char_fichero;
    char line[80];
    char *token;
    const char separator[2] = ",";
    FILE *archivo;

    if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
        printf("Error al abrir el fichero\n");
        exit(EXIT_FAILURE);
    }
    char_fichero=fgets(line,MAX_DATOS*sizeof(char),archivo);
    token = strtok(line, separator);

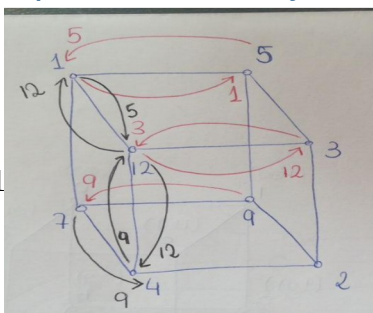
    while( token != NULL ) {
        datos++;
        token = strtok(NULL, separator);
    }
    fclose(archivo);

    return datos;
}
```

Para obtener los nodos vecinos de cada nodo que forman nuestra red hipercubo, se va a ejecutar la función XOR con el nodo seleccionado podrá hacer que vaya variando un bit de posición gracias a la línea: *desplazamiento=desplazamiento<<1;*

Finalmente, para poder comprobar que el número de datos introducidos en el fichero datos.dat es el correcto, se realiza el mismo método que el implementado en la red toroide ya que es un contador de números normal.

Explicación del flujo de datos



El flujo de datos inicial es igual que en el caso de la red toroide. El nodo 0 comienza mandando mensajes a todos los nodos indicando si

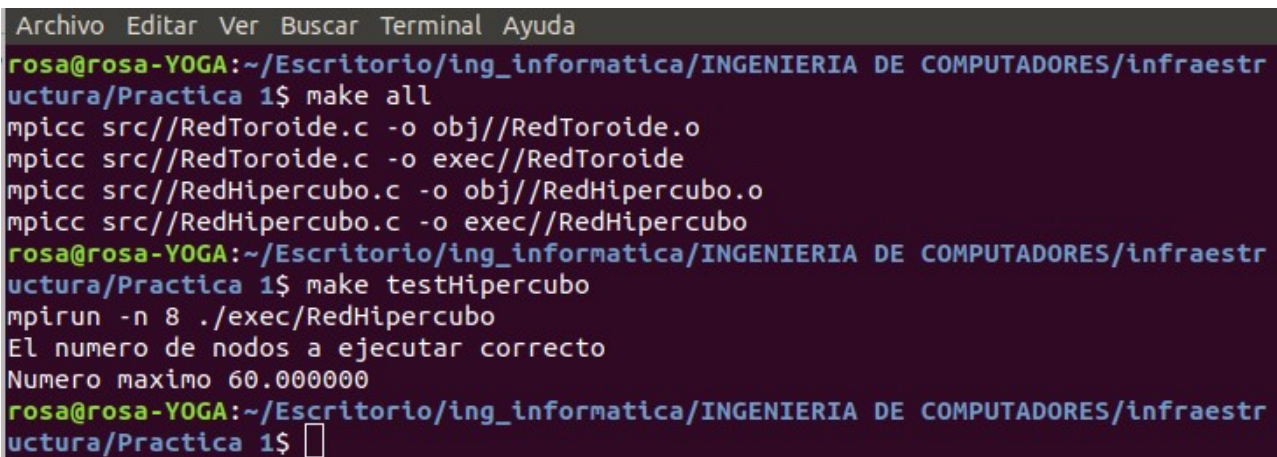
Práctica 1. Infraestructura de red. Red toroide y red hipercubo

deben seguir la ejecución normal del programa o deben de detenerse. Si continúan su ejecución normal, el nodo 0 lee el fichero datos.dat y distribuye los números a cada uno de los nodos. Una vez los nodos reciben su respectivo número, comienzan con la búsqueda del mayor valor. Cada nodo distribuye entre sus vecinos un total de D veces su número actual, recibiendo, a su vez, el de sus vecinos. Comparan el número que han recibido de sus vecinos con el que ellos tenían y se quedan con el de mayor valor de los dos. Así, toda la red terminará quedándose con el número de mayor valor que exista en el fichero datos.dat.

Cómo compilar y ejecutar

La compilación del programa se realizará mediante el *makefile* que se encuentra en la carpeta del proyecto. En el archivo datos.dat introduciremos 8 números aleatorios para poder comprobar que la práctica funciona correctamente.

- Compilar: usaremos el comando *make all*
- Ejecutar: usaremos el comando *maketestHiper cubo*



```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
rosa@rosa-YOGA:~/Escritorio/ing_informatica/INGENIERIA DE COMPUTADORES/infraestr
uctura/Practica 1$ make all
mpicc src//RedToroide.c -o obj//RedToroide.o
mpicc src//RedToroide.c -o exec//RedToroide
mpicc src//RedHiper cubo.c -o obj//RedHiper cubo.o
mpicc src//RedHiper cubo.c -o exec//RedHiper cubo
rosa@rosa-YOGA:~/Escritorio/ing_informatica/INGENIERIA DE COMPUTADORES/infraestr
uctura/Practica 1$ make testHiper cubo
mpirun -n 8 ./exec/RedHiper cubo
El numero de nodos a ejecutar correcto
Numero maximo 60.000000
rosa@rosa-YOGA:~/Escritorio/ing_informatica/INGENIERIA DE COMPUTADORES/infraestr
uctura/Practica 1$
```

Conclusiones

Con la realización de dicha práctica he podido comprobar el funcionamiento el control que podemos tener sobre la red gracias a MPI. En este caso, cabe destacar que podemos trabajar a nivel de bit para poder conocer tanto los nodos como los vecinos de los nodos dentro de nuestra red hipercubo.