Escuela superior de informática

Diseño de infraestructuras de Red

Practica 1

Cristian Trapero Mora

10 de mayo de 2016

Índice

1. Red toroide

1.1. Enunciado del problema

Dado un archivo con nombre **datos.dat**, cuyo contenido es una lista de valores separados por comas, nuestro programa realizará lo siguiente:

- El proceso de **rank 0 distribuirá** a cada uno de los nodos de un toroide de lado L, los L*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 terminarán.

- 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 $\mathbf{O(\sqrt{N})}$. Con N número de elementos de la red.

1.2. Planteamiento de la solución

Antes de plantear una solución al problema de la red toroide, primero debemos de conocer como es su estructura y funcionamiento.

Un **toroide** [1] es una red de interconexión de nodos 1 en la que cada nodo tiene conexión directa con los nodos en las posiciones **Norte**, **Sur**, **Este** y **Oeste** al mismo. Esto deriva en la creación de redes simétricas, permitiendo que el diámetro de la red se reduzca a la mitad.

El tamaño de una red toroide viene definido por el lado de la misma, de tal forma que el numero de nodos total viene definido como: **N = L*L**

En nuestro caso en concreto la **numeración** de los nodos empieza desde 0, por tanto si tenemos una red toroide de lado L=4, tendremos un total de 16 nodos, numerados desde el 0 hasta el 15.

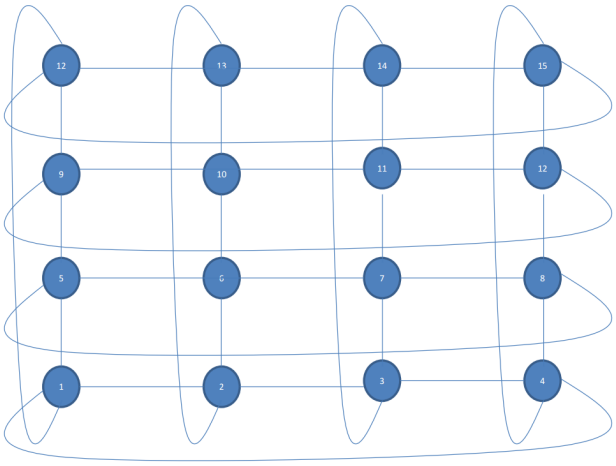

Figura 1: Red toroide

Una vez que hemos definido la estructura de una red toroide, plantearemos la solución al problema principal. En el caso que nos ocupa, podemos distinguir **dos roles** o procesos con funciones específicas en base al rank[1] :

- **Rank 0**:

  Es el proceso encargado de **procesar los números** contenidos en el fichero datos.dat y **enviarlos** a cada proceso.

  Es el proceso encargado de **controlar el numero de procesos** lanzados en base a la dimensión establecida.

  Es el proceso encargado de **controlar la relación** entre el numero de procesos lanzados y el numero de datos contenido en el fichero.

  Es el proceso encargado de **controlar la ejecución** de los demás procesos y de **mostrar el resultado** final tras los cálculos.

- **Todos los rank**:

  Se encargaran de **recibir el dato** enviado por el rank 0.

  Se encargaran de **obtener los vecinos** adyacentes a él mismo.

  Se encargaran de **procesar y enviar** el numero mínimo entre sus vecinos.

[1]Variable que permite identificar al proceso en ejecución.

Tras examinar las distintas funciones que han de tener cada proceso en base a su rank, debemos de detallar las situaciones en las cuales se debe **detener la ejecución** de los procesos:

- Se debe de detener la ejecución de los procesos cuando el **numero de procesos** lanzados sea **distinto** al cuadrado del lado (**L*L**).
- Se debe de detener la ejecución de los procesos cuando el **numero de datos** contenido en el fichero **difiere** del cuadrado del lado (**L*L**).

1.3. Diseño del programa

Tras analizar los distintos problemas que se nos presentan dado el enunciado, plantearemos el diseño del programa cubriendo todos los puntos anteriormente detallados. Primeramente nos centraremos en las **tareas del rank 0**, puesto que es el proceso que mayor peso tiene en la ejecución del programa:

- **Numero de procesos**: Para comprobar que el numero de procesos lanzados se corresponde con la dimensión del toroide haremos uso de la función ***MPI_Comm_size(MPI_COMM_WORLD, &size)*** la cual nos permite obtener el numero de procesos lanzados, para posteriormente compararla con el tamaño de la red toroide para un lado L, la cual viene dada por **L*L**. En el caso que no se corresponda el numero de procesos lanzados con el tamaño de la red, enviaremos una flag, mediante la instrucción ***MPI_Bcast()*** *[2]* a los demás procesos, para que paren su ejecución.
- **Lectura del fichero**: Para procesar los datos del fichero definiremos una función denominada ***leerFichero()***, que nos retornará el **numero de datos** leídos, y el **vector de números** que almacenará los datos contenidos en el fichero. El fichero se estructura como una cadena de números tanto positivos como negativos, separados por coma, por lo tanto para tratar dichos números, haremos uso de la función ***strtok()*** [3] que nos permite obtener un puntero en el fichero, en base al **delimitador coma** (**,**).
- **Numero de datos**: Una vez que hemos leído el fichero y hemos obtenido el vector con los números que debemos de distribuir entre los nodos, comprobaremos que la **cantidad de números leídos** se corresponde

con el tamaño de la red toroide (**L*L**), de tal manera que si esta difiere, enviemos el flag de la forma mencionada anteriormente, para que los procesos paren su ejecución.

- **Envío de datos**: Una vez que hemos comprobado que el numero de datos se corresponde con el tamaño de la red, **enviaremos a cada nodo** un numero del vector mediante la instrucción ***MPI_Send()***, de manera que todos los nodos obtengan un numero del fichero, incluido el rank 0.

Una vez que hemos definido las tareas correspondientes al rank 0, nos centraremos en las que deben de realizar todos los procesos tras las comprobaciones del rank 0. Como hemos comentado antes, la ejecución de los procesos son precedidas por la instruccion ***MPI_Broadcast()*** la cual permite realizar una llamada colectiva, que detiene la ejecución de todos los procesos hasta que todos han ejecutado dicha instrucción. Dicha llamada nos permite pasar como parámetro un **flag** que nos permita definir si queremos seguir con la **ejecución (1)** o **no (0)**.

- **Recepción del dato**: La primera instrucción que nos encontramos es ***MPI_Recv()*** la cual nos permite realizar una espera activa hasta que recibamos el dato proveniente desde el rank 0.

- **Obtención de los vecinos**: Una vez que hemos recibido el dato, obtendremos los vecinos adyacentes en las posiciones Norte, Sur, Este y Oeste a mi posición. Para ello definiremos una función denominada ***obtenerVecinos()***, que realiza dichos cálculos en base a mi rank y al tamaño de la red.

- **Obtener el mínimo**: Una vez que tenemos el numero con el que trabajar y los vecinos con los que interactuaremos, llegamos al punto mas crítico del programa, en el cual debemos de obtener el menor número distribuido entre los nodos de la red. Para ello definiremos una función denominada ***obtenerMinimo()***, la cual se estructura de la siguiente forma: los nodos interactúan de manera directa con los vecinos que tienen al **Este y al Oeste**, de tal manera que se envíen entre ellos el numero que almacenan para poder compararlo con el que ya tienen, si este es menor que el que almacenan actualmente, lo cambian por el, de manera que obtendremos el menor numero en cada **fila de la red**,

pero no en el total de la red; por tanto, necesitamos una interacción con los vecinos del **Norte y del Sur**, de tal forma que enviemos el menor numero de cada fila con la fila superior e inferior, de manera que obtengamos de nuevo el menor numero por **columna**, obteniendo así el menor numero de toda la red, distribuido en todos los nodos de la red. Para ello simplemente emplearemos dos bucles ***for()*** y las llamadas ***MPI_Send()*** y ***MPI_Recv()*** para el envío y la recepción de los datos.

- **Mostrar resultado**: Una vez que hemos obtenido el menor numero de la red, el rank 0 será el encargado de imprimir por pantalla el valor obtenido por la función anteriormente mencionada.

1.4. Fuentes del programa

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

//#define L 3
#define DATOSDAT "datos.dat"
#define MAX_FILE 1024
#define MAX_ITEMS 1024

int leerFichero(double *numeros);
void obtenerVecinos(int rank, int *vecinoSuperior, int *
    vecinoInferior, int *vecinoIzquierdo, int *vecinoDerecho);
double obtenerMinimo(int rank, double bufferNumero, int
    vecinoSuperior, int vecinoInferior, int vecinoIzquierdo, int
    vecinoDerecho);

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

        int rank, size;
        MPI_Status status;
        double bufferNumero;
        double numeroMinimo;
        int vecinoSuperior, vecinoInferior, vecinoIzquierdo,
            vecinoDerecho;
        int seguir=1;
```

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

if (rank==0){

        if(L*L!=size){

                printf("Error: Se deben lanzar %d
                    procesos para un toroide de lado %d \
                    n", L*L, L);
                seguir=0;
                MPI_Bcast(&seguir, 1, MPI_INT, 0,
                    MPI_COMM_WORLD);

        }else{

                double *numeros=malloc(MAX_ITEMS *
                    sizeof(double));
                int cantidadNumeros=leerFichero(numeros)
                    ;

                if(L*L!=cantidadNumeros){
                        printf("Error: Cantidad de
                            numeros contenidos en el
                            fichero incorrecto. Se
                            necesitan %d datos, y se
                            tienen %d.\n", L*L,
                            cantidadNumeros);
                        seguir=0;
                        MPI_Bcast(&seguir, 1, MPI_INT,
                            0, MPI_COMM_WORLD);

                }else{

                        MPI_Bcast(&seguir, 1, MPI_INT,
                            0, MPI_COMM_WORLD);
                        int j;

                        for (j = 0; j < cantidadNumeros;
                            ++j){
                                bufferNumero=numeros[j];
                                MPI_Send(&bufferNumero,
                                    1, MPI_DOUBLE, j, 0,
```

```
                                                MPI_COMM_WORLD);
                                }

                                free(numeros);
                        }


                }

        }

        MPI_Bcast(&seguir, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if(seguir!=0){

                MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                obtenerVecinos(rank, &vecinoSuperior, &
                    vecinoInferior, &vecinoIzquierdo, &
                    vecinoDerecho);
                numeroMinimo=obtenerMinimo(rank, bufferNumero,
                    vecinoSuperior, vecinoInferior,
                    vecinoIzquierdo, vecinoDerecho);

                if(rank==0){
                        printf("Soy el rank %d. El menor numero
                            de la red es: %.3lf\n", rank,
                            numeroMinimo);
                }

        }

        MPI_Finalize();
        return 0;
}

int leerFichero(double *numeros){

        char *listaNumeros=malloc(MAX_FILE * sizeof(char));
        int cantidadNumeros=0;
        char *numeroActual;

        FILE *fichero=fopen(DATOSDAT, "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);
        }

        free(listaNumeros);
        return cantidadNumeros;
}

void obtenerVecinos(int rank, int *vecinoSuperior, int *
    vecinoInferior, int *vecinoIzquierdo, int *vecinoDerecho){
        int fila, columna;


        fila=rank/L;
        columna=rank%L;

        if(fila==0){
                *vecinoInferior = ((L-1)*L)+columna;
        }else{
                *vecinoInferior = ((fila-1)*L)+columna;
        }

        if(fila==L-1){
                *vecinoSuperior = columna;
        }else{
                *vecinoSuperior = ((fila+1)*L)+columna;
        }

        if(columna==0){
                *vecinoIzquierdo = (fila*L)+(L-1);
        }else{
                *vecinoIzquierdo = (fila*L)+(columna-1);
        }
```

```
        if(columna==L-1){
                *vecinoDerecho = (fila*L);
        }else{
                *vecinoDerecho = (fila*L)+(columna+1);
        }

}

double obtenerMinimo(int rank, double bufferNumero, int
    vecinoSuperior, int vecinoInferior, int vecinoIzquierdo, int
    vecinoDerecho){

        int i;
        double minimo=DBL_MAX;
        MPI_Status status;

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

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

                MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoDerecho,
                    i, MPI_COMM_WORLD);
                MPI_Recv(&bufferNumero, 1, MPI_DOUBLE,
                    vecinoIzquierdo, i, MPI_COMM_WORLD, &status);
                printf("Soy el rank %d y he recibido el dato %f
                    \n",rank, bufferNumero );

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


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

                MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoSuperior,
                     i, MPI_COMM_WORLD);
                MPI_Recv(&bufferNumero, 1, MPI_DOUBLE,
                    vecinoInferior, i, MPI_COMM_WORLD, &status);
                printf("Soy el rank %d y he recibido el dato %f
                    \n",rank, bufferNumero );

                if(bufferNumero<minimo){
```

```
                        minimo=bufferNumero;
                    }
                }

                return minimo;
        }
```

2. Red hipercubo

2.1. Enunciado del problema

Dado un archivo con nombre **datos.dat**, cuyo contenido es una lista de valores separados por comas, nuestro programa realizará lo siguiente:

- El proceso de **rank 0 distribuirá** a cada uno de los nodos de un 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, 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 $\mathbf{O}(\log_2 N)$ Con N número de elementos de la red.

2.2. Planteamiento de la solución

Antes de plantear una solución al problema de la red hipercubo, primero debemos de conocer como es su estructura y funcionamiento.

Un **hipercubo** [1] es una **malla n-dimensional** 2 en la que se han suprimido los nodos interiores. Un n-cubo puede formarse interconectando los nodos equivalentes de (n-1)-cubos. El grado de los nodos de un n-cubo es n.

El tamaño de una red hipercubo viene definido por el grado de los nodos, de tal forma que el numero de nodos total viene definido como: 2^N

En nuestro caso en concreto la **numeración** de los nodos empieza desde 0, por tanto si tenemos una red hipercubo de grado 3, tendremos un total de 8 nodos, numerados desde el 0 hasta el 7.

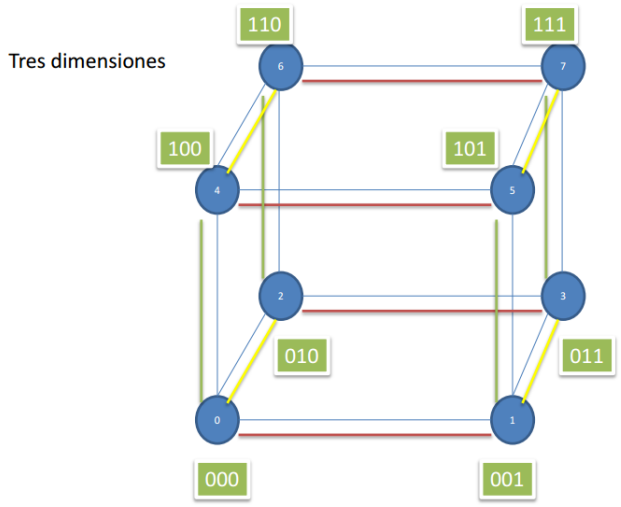

Figura 2: Red hipercubo

Una vez que hemos definido la estructura de una red hipercubo, plantearemos la solución al problema principal. En el caso que nos ocupa, podemos distinguir **dos roles** o procesos con funciones específicas en base al rank:

- **Rank 0**:

  Es el proceso encargado de **procesar los números** contenidos en el fichero datos.dat y **enviarlos** a cada proceso.

  Es el proceso encargado de **controlar el numero de procesos** lanzados en base a la dimensión establecida.

  Es el proceso encargado de **controlar la relación** entre el numero de procesos lanzados y el numero de datos contenido en el fichero.

  Es el proceso encargado de **controlar la ejecución** de los demás procesos y de **mostrar el resultado** final tras los cálculos.

- **Todos los rank**:

  Se encargaran de **recibir el dato** enviado por el rank 0.

  Se encargaran de **obtener los vecinos** adyacentes a él mismo.

  Se encargaran de **procesar y enviar** el numero máximo entre sus vecinos.

Tras examinar las distintas funciones que han de tener cada proceso en base a su rank, debemos de detallar las situaciones en las cuales se debe **detener la ejecución** de los procesos:

- Se debe de detener la ejecución de los procesos cuando el **numero de procesos** lanzados sea **distinto** al numero total de nodos de la red (2^N).
- Se debe de detener la ejecución de los procesos cuando el **numero de datos** contenido en el fichero **difiere** del total de nodos de la red (2^N).

2.3. Diseño del programa

Tras analizar los distintos problemas que se nos presentan dado el enunciado, plantearemos el diseño del programa cubriendo todos los puntos anteriormente detallados. Primeramente nos centraremos en las **tareas del rank 0**, puesto que es el proceso que mayor peso tiene en la ejecución del programa:

- **Numero de procesos**: Para comprobar que el numero de procesos lanzados se corresponde con la dimensión del hipercubo haremos uso de la función ***MPI_Comm_size(MPI_COMM_WORLD, &size)*** la cual nos permite obtener el numero de procesos lanzados, para posteriormente compararla con el grado del hipercubo, la cual viene dada por 2^N. En el caso que no se corresponda el numero de procesos lanzados con el tamaño de la red, enviaremos una flag, mediante la instrucción ***MPI_Bcast()*** a los demás procesos, para que paren su ejecución.
- **Lectura del fichero**: Para procesar los datos del fichero definiremos una función denominada ***leerFichero()***, que nos retornará el **numero de datos** leídos, y el **vector de números** que almacenará los datos contenidos en el fichero. El fichero se estructura como una cadena de números tanto positivos como negativos, separados por coma, por lo tanto para tratar dichos números, haremos uso de la función ***strtok()*** que nos permite obtener un puntero en el fichero, en base al **delimitador coma** (,).
- **Numero de datos**: Una vez que hemos leído el fichero y hemos obtenido el vector con los números que debemos de distribuir entre los

nodos, comprobaremos que la **cantidad de números leídos** se corresponde con el tamaño de la red hipercubo (2^N), de tal manera que si esta difiere, enviemos el flag de la forma mencionada anteriormente, para que los procesos paren su ejecución.

- **Envío de datos**: Una vez que hemos comprobado que el numero de datos se corresponde con el tamaño de la red, **enviaremos a cada nodo** un numero del vector mediante la instrucción ***MPI_Send()***, de manera que todos los nodos obtengan un numero del fichero, incluido el rank 0.

Una vez que hemos definido las tareas correspondientes al rank 0, nos centraremos en las que deben de realizar todos los procesos tras las comprobaciones del rank 0. Como hemos comentado antes, la ejecución de los procesos son precedidas por la instruccion ***MPI_Broadcast()*** la cual permite realizar una llamada colectiva, que detiene la ejecución de todos los procesos hasta que todos han ejecutado dicha instrucción. Dicha llamada nos permite pasar como parámetro un **flag** que nos permita definir si queremos seguir con la **ejecución (1)** o **no (0)**.

- **Recepción del dato**: La primera instrucción que nos encontramos es ***MPI_Recv()*** la cual nos permite realizar una espera activa hasta que recibamos el dato proveniente desde el rank 0.

- **Obtención de los vecinos**: Una vez que hemos recibido el dato, obtendremos los vecinos adyacentes mediante la **operación XOR** entre mi rank y la potencia 2 elevada al grado del hipercubo. Todos los vecinos adyacentes se almacenarán en un vector de enteros para cada rank.

- **Obtener el máximo**: Una vez que tenemos el numero con el que trabajar y los vecinos con los que interactuaremos, llegamos al punto mas crítico del programa, en el cual debemos de obtener el mayor número distribuido entre los nodos de la red. Para ello definiremos una función denominada ***obtenerMaximo()***, la cual funciona de la siguiente forma: Para todos los vecinos de mi rank, envio mi dato a ellos, y lo recibo de ellos, y compruebo que el numero que me han pasado es mayor al actual de tal forma que tras **L iteraciones** todos los nodos tienen el mayor numero de la red.

- **Mostrar resultado**: Una vez que hemos obtenido el mayor numero de la red, el rank 0 será el encargado de imprimir por pantalla el valor obtenido por la función anteriormente mencionada.

2.4. Fuentes del programa

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


#define DATOSDAT "datos.dat"
#define MAX_FILE 1024
#define MAX_ITEMS 1024


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


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

        int rank, size;
        MPI_Status status;
        double bufferNumero;
        double numeroMaximo;

        int numeroProcesosALanzar=(int) round(pow(2,L));
        int vecinos[L];
        int seguir=1;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (rank==0){

                if(numeroProcesosALanzar!=size){
```

```
                printf("Error: Se deben lanzar %d
                    procesos para un hipercubo de
                    dimension %d \n",
                    numeroProcesosALanzar, L);
                seguir=0;
                MPI_Bcast(&seguir, 1, MPI_INT, 0,
                    MPI_COMM_WORLD);

        }else{

                double *numeros=malloc(MAX_ITEMS *
                    sizeof(double));
                int cantidadNumeros=leerFichero(numeros)
                    ;

                if(pow(2,L)!=cantidadNumeros){
                        printf("Error: Cantidad de
                            numeros contenidos en el
                            fichero incorrecto. Se
                            necesitan %d datos, y se
                            tienen %d.\n",
                            numeroProcesosALanzar,
                            cantidadNumeros);
                        seguir=0;
                        MPI_Bcast(&seguir, 1, MPI_INT,
                            0, MPI_COMM_WORLD);

                }else{

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

                        free(numeros);
                }


        }
```

```
        }

        MPI_Bcast(&seguir, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if(seguir!=0){

                MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, 0,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                obtenerVecinos(rank, vecinos);
                numeroMaximo=obtenerMaximo(rank, bufferNumero,
                    vecinos);

                if(rank==0){
                        printf("Soy el rank %d. El mayor numero
                            de la red es: %.3lf\n", rank,
                            numeroMaximo);
                }

        }

        MPI_Finalize();
        return 0;
}

int leerFichero(double *numeros){

        char *listaNumeros=malloc(MAX_FILE * sizeof(char));
        int cantidadNumeros=0;
        char *numeroActual;

        FILE *fichero=fopen(DATOSDAT, "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;
}

void obtenerVecinos(int rank, int *vecinosAdyacentes){
        int i;
        int rankAuxiliar;

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

                rankAuxiliar=(rank^((int)pow(2,i)));
                vecinosAdyacentes[i]=rankAuxiliar;
        }

}

double obtenerMaximo(int rank, double bufferNumero, int *vecinos
    ){

        int i;
        double maximo=DBL_MIN;
        MPI_Status status;

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

                if(bufferNumero>maximo){
                        maximo=bufferNumero;
                }

                MPI_Send(&maximo, 1, MPI_DOUBLE, vecinos[i], i,
                    MPI_COMM_WORLD);
                MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, vecinos[i
                    ], i, MPI_COMM_WORLD, &status);

                if(bufferNumero>maximo){
                        maximo=bufferNumero;
                }
        }

        return maximo;

}
```

3. Flujo de datos MPI

La solución al problema la hemos planteado empleando simplemente tres primitivas de MPI, que son: **MPI_Broadcast**, **MPI_Send** y **MPI_Recv**. Dichas primitivas tienen **naturaleza síncrona**, de manera que realizan una espera activa hasta la recepción de algún tipo de señal. **MPI_Broadcast** a su vez se trata de una **primitiva colectiva**, de manera que todos los procesos lanzados se ven afectados por ella. En nuestro caso en concreto hemos hecho uso de la misma porque nos aporta una solución muy simple a la cancelación de los procesos en el caso de que el rank 0 encuentre alguna condición de parada, usando una variable entera cuyo valor puede ser 0 o 1. Dicha primitiva simplemente se limita a **detener la ejecución** de las instrucciones que le preceden hasta que todos los procesos hayan ejecutado la primitiva.

La primitiva tiene la siguiente forma:

```
MPI_Bcast(&seguir, 1, MPI_INT, 0, MPI_COMM_WORLD)
```

En la cual definimos la **variable** que vamos a enviar, el **numero de datos** que enviamos, el **tipo de variable** que enviamos y el **intercomunicador** al que afecta.

Por otro lado tenemos las funciones **MPI_Send** y **MPI_Recv**, las cuales permiten el envio de datos de forma síncrona. Tienen la siguiente estructura:

```
MPI_Send(&minimo, 1, MPI_DOUBLE, vecinoDerecho, i,
    MPI_COMM_WORLD);

MPI_Recv(&bufferNumero, 1, MPI_DOUBLE, vecinoIzquierdo, i,
    MPI_COMM_WORLD, &status);
```

En ambas primitivas definimos la **variable** que vamos a enviar, el **numero de datos** que enviamos, el **tipo de variable** que enviamos, el **destinatario** de envío, un **tag** o comentario y el **intercomunicador** al que afecta, y en la primitiva de recepción además incluimos una variable de estado.

4. Instrucciones de compilación y ejecución

Para la compilación y ejecución de los programas hemos automatizado el proceso mediante un archivo Makefile, el cual se rige por una serie de

reglas. Para facilitar la ejemplificación en la ejecución de los programas, he creado un **programa en C** denominado ***generarDatos.c***, que nos permite generar una cantidad N de **números aleatorios** en el fichero datos.dat, para posteriormente utilizarlo en la ejecución de los programas.

Para compilar los programas simplemente debemos de ejecutar las siguientes reglas en el directorio donde se encuentran los archivos fuente y el Makefile:

1. Compilación de **toroide**:

```
$ make compilarToroide
```

2. Compilación de **hipercubo**:

```
$ make compilarHipercubo
```

3. Generar **datos aleatorios**:

```
$ make generarDatos
```

En cambio si queremos ejecutar los programas debemos de ejecutar las siguientes instrucciones:

1. Ejecutar **toroide**:

```
$ make ejecutarToroide
```

2. Ejecutar **hipercubo**:

```
$ make ejecutarHipercubo
```

Si queremos automatizar mas aun todo el proceso y desplegarlo con una sola instruccion, debemos de ejecutar las siguientes instrucciones:

1. Desplegar **toroide**:

```
$ make desplegarToroide
```

2. Desplegar **hipercubo**:

```
$ make desplegarHipercubo
```

5. Conclusiones

Tras la realización de la practica las impresiones que he tenido son realmente positivas, porque la inclusión de **tres simples primitivas de MPI** han permitido desarrollar un programa distribuido que se puede escalar sin ningún esfuerzo. Con respecto a la complejidad del programa simplemente comentar que tuve algunas complicaciones hasta descubrir la función strtok utilizada para leer el fichero de datos, y pensar en como obtener el rank de los vecinos en la red Hipercubo. Por todo lo demás, me ha parecido interesante la utilización de la **interfaz de paso de mensajes** MPI, puesto que es realmente flexible y compatible con un gran numero de lenguajes, por lo que puede ser provechoso para el desarrollo de programas distribuidos de manera muy sencilla.

6. Bibliografía

Referencias

[1] Arquitecturas de red

[2] mpitutorial.com: **MPI_Broadcast**

[3] cppreference.com: **Strtok**