UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMATICA

DISEÑO DE INFRAESTRUCTURA DE RED

GRADO EN INGENIERÍA INFORMÁTICA

2021-2022

PRÁCTICA 1

Autor:

DIEGO DORADO GALÁN

Fecha:

18-04-2022

Índice de contenido

1. RED TOROIDE

1.1 ENUNCIADO

El primer problema consiste en la representación de una red toroide que permita obtener el valor menor de toda la red. Estos valores serán distribuidos por el rank 0 a todos los nodos de la red, comprobando previamente si existen suficientes valores para satisfacer a todos los componentes.

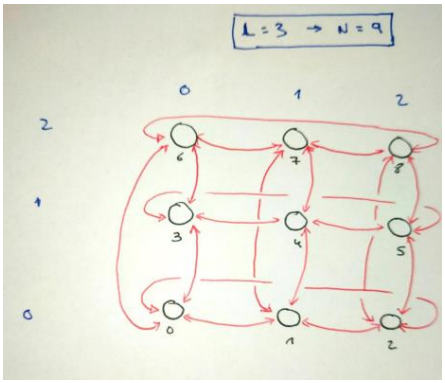
Además, para satisfacer las exigencias del enunciado, la complejidad del algoritmo no superará O(raiz_cuadrada(n)), siendo n el número de elementos de la red (L^2). Al final del programa, el nodo rank 0 mostrará la solución por pantalla.

1.2 PLANTEAMIENTO DE LA SOLUCIÓN

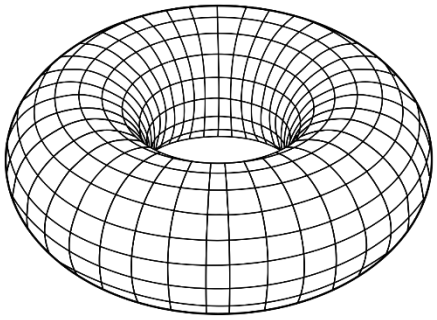
Un toroide (o también conocido como toro) es una red estática de interconexión de nodos. Cada uno de los nodos, está directamente conectado con otros cuatro nodos (o vecinos): Norte, Sur, Este y Oeste.

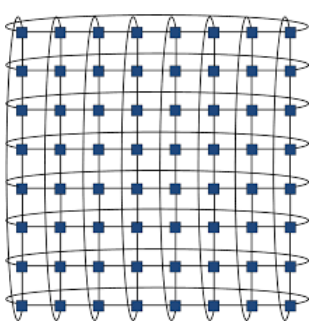
Podemos definir su tamaño a partir del lado de este, siendo $N = L^2$.

La forma en que los nodos quedan numerados es de izquierda a derecha y de abajo a arriba. Podemos apreciarlo gráficamente mediante este boceto (de baja calidad) como quedan representados los nodos y como se interconectan entre ellos:



Algunas imágenes de mayor fidelidad son las siguientes:





Una vez definida su topología y estructura, reflejaré la problemática principal. Nuestro programa diferencia claramente entre dos funcionalidades: rank 0, que se ocupará de gran parte de la lógica del programa; y resto de nodos.

- **Rank 0:**

  Este nodo se encarga de procesar el fichero datos.dat y los datos que contiene este. Es el encargado de enviarlos a cada proceso, y en mi caso también de imprimir la secuencia de envío por pantalla.
  A su vez, controlará la relación entre el número de datos contenidos en el fichero y el número de procesos lanzados para el correcto funcionamiento del programa. Controlará la ejecución del resto de procesos e imprimirá el resultado por pantalla.

- **Resto de nodos:**

  Reciben los números enviados por el rank 0, calculan sus vecinos (Norte, Sur, Este y Oeste). En mi caso, esta nomenclatura varía y uso Norte, Sur, Izquierda y Derecha por simple comodidad y simplicidad.
  Envía el número mínimo a sus vecinos a la vez que recibe el de otros, comparando con el valor almacenado en ese momento.

Se debe cumplir una serie de circunstancias para el correcto funcionamiento del programa, como son que el número de procesos lanzados sea igual al cuadrado del lado de la red toroide. También, se debe garantizar que el número de datos leídos del fichero sea equivalente al cuadrado del lado de la red toroide.

En caso contrario, se ha proporcionado un control de errores y excepciones para evitar la finalización no controlada del programa.

Como inconveniente al desarrollo de la práctica (y pendiente de rediseño), el lado se define como una constante en el código, al igual que el número de procesos a lanzar quedan predefinidos en el Makefile. Por tanto, para una ejecución de una red con distinto lado de 4 (definido en el código) se deberá modificar previamente ambos archivos.

1.3 DISEÑO DEL PROGRAMA

Mediante la función *MPI_COMM_Size* somos capaces de conocer el número de procesos lanzados. Si coincide con respecto al número de nodos de la red (L*L), procedemos a asignar un espacio de memoria al array números que almacenará los datos leídos del fichero mediante la función *leerdatos()*. Una vez almacenados, pasamos a comprobar si existen tantos datos como nodos en el toroide. En caso afirmativo, el rank 0 enviará mediante la función *enviarDatos()* (que contiene la primitiva *MPI_Send())* un double a cada nodo. Tras esto se liberará el espacio ocupado por el array números nombrado anteriormente. En caso negativo, se notificará por pantalla el error y se emitirá un Broadcast (*MPI_Bcast())* al resto de nodos del comunicador (*MPI_COMM_WORLD*) para que detengan su ejecución.

Si no se cumplía la primera de las condiciones (que el tamaño obtenido mediante *MPI_COMM_Size* sea igual al número de nodos de la red) se realizará lo mismo que hemos mencionado anteriormente. Se emitirá un broadcast al resto de procesos.

Esta serie de condiciones lo podemos controlar gracias a la variable *condición*.

Antes de la ejecución del resto de procesos, realizamos un *MPI_Broadcast()*, que como llamada colectiva que es detendrá la ejecución del resto de nodos a modo de sincronización.

Tras ello, mediante *MPI_Recv()* realizamos una espera activa (ya que es una llamada bloqueante) hasta recibir el valor del fichero proveniente del rank 0.

Mediante la función *getVecinos()* obtenemos los vecinos propios de cada nodo y que mostraré posteriormente.

La función *getMinimo()* se basa en el siguiente pseudocódigo para su ejecución:

```
/* Envío a nodo este y recibo de oeste
    for(i=0;i<L;i++){

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

        send(e);
        recv(o);

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

    /* Envio a nodo sur y recibo de norte
    for(i=0;i<L;i++){

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

        send(e);
        recv(o);

        if(num<min){
            min = num;
        }
    }
```

Una vez calculado el mínimo, el rank 0 muestra por pantalla el resultado obtenido.

Mediante MPI_Finalize cerramos el bloque de código MPI.

1.4 CÓDIGO FUENTE DEL PROGRAMA

```
/* ********** CONSTANTES Y VARIABLES GLOBALES*************** */

#define FICHERO "datos.dat"
#define MAX_SIZE 1024
#define L 4
#define n (L*L)
#define TRUE 1
#define FALSE 0

/*************** FUNCIONES AUXILIARES **********************/

int leerdatos(double numeros[]);
void getVecinos(int rank, int vecinos[]);
double getMinimo(int rank, double buffer, int vecinos[]);
void enviarDatos(double *numeros);

/*************** FUNCION PRINCIPAL **************************/


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

    int rank, size;

    MPI_Status status;

    int vecinoNorte, vecinoSur, vecinoIzq, vecinoDch;
    int vecinos[L];

    double min;
    double buffer;

    int leidos;
    double numeros[n];

    int condicion = TRUE;

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

    if(rank == 0){

        if(size == n){

            double *numeros = malloc(size * sizeof(double));

            leidos = leerdatos(numeros);
            if(n == size){

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

                enviarDatos(numeros);

                condicion=TRUE;
                free(numeros);
            }
```

```
            else {

                condicion = FALSE;
                fprintf(stderr, "El número de datos leídos del fichero
(%d) no es suficiente. Se necesitan %d\n", leidos, n);

                MPI_Bcast(&condicion,1,MPI_INT,0,MPI_COMM_WORLD);
            }

        }

        else {

            condicion = FALSE;
            fprintf(stderr,"Tenemos que ejecutar %d procesos para un
toroide de lado %d \n",n,L);

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

        }
    }

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

    if(condicion==TRUE){


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

        getVecinos(rank, vecinos);

        min = getMinimo(rank,buffer, vecinos);

        if(rank == 0)
            printf("El menor número encontrado en la red es
%.5lf\n",min);

    }

    MPI_Finalize();
    return 0;
}


/*********** FUNCIONES AUXILIARES *****************/

void enviarDatos(double *numeros){
    int i;
    double buffer;

    for(i=0;i<n;i++){
        buffer = numeros[i];
        MPI_Send(&buffer,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
        printf("%5.lf enviado al nodo %d \n",buffer,i);
    }
}
```

```
int leerdatos(double numeros[]){

    char *buffer = malloc(MAX_SIZE * sizeof(char));
    errno = 0;
    int contador = 0;
    int size = 0;
    char *aux;


    FILE *fp = fopen(FICHERO,"r");
    if(errno != 0){
        fprintf(stderr, "Se ha detectado un error (%d) al intentar abrir
el fichero\n",errno);
        return 0;
    }

    fscanf(fp,"%s",buffer);

    numeros[size++] = atof(strtok(buffer,","));

    while ((aux = strtok(NULL,",")) != NULL) {
        numeros[size++]=atof(aux);
    }


free(buffer);

fclose(fp);

return size;

}



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

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


    switch(columna){

        case 0:
            vecinos[2] = rank + (L-1);
            vecinos[3] = rank + 1;
            break;
        case L-1:
            vecinos[2] = rank -1;
            vecinos[3] = rank - (L-1);
            break;
        default:
            vecinos[2] = rank-1;
            vecinos[3] = rank+1;
            break;
    }
```

```
    switch(fila){

        case 0:
            vecinos[0] = rank+L;
            vecinos[1] = (L * (L-1))+ rank;
            break;
        case L-1:
            vecinos[0] = columna;
            vecinos[1] = (rank - L);
            break;
        default:
            vecinos[0] = rank +L;
            vecinos[1] = rank - L;
            break;
    }
}

double getMinimo(int rank, double buffer, int vecinos[]){
    int i;

    MPI_Status status;

    double min;
    min = buffer;

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

        MPI_Send(&min,1,MPI_DOUBLE, vecinos[3],i,MPI_COMM_WORLD);

        MPI_Recv(&buffer,1,                                    MPI_DOUBLE,
vecinos[2],i,MPI_COMM_WORLD,&status);

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


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


        MPI_Send(&min,1,MPI_DOUBLE, vecinos[1],i,MPI_COMM_WORLD);

        MPI_Recv(&buffer,1,                                    MPI_DOUBLE,
vecinos[0],i,MPI_COMM_WORLD,&status);

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

    return min;
}
```

2. RED HIPERCUBO

2.1 ENUNCIADO DEL PROBLEMA

El segundo problema consiste en la representación de una red hipercubo que permita obtener el valor mayor de toda la red. Estos valores serán distribuidos por el rank 0 a todos los nodos de la red, comprobando previamente si existen suficientes valores para satisfacer a todos los componentes.

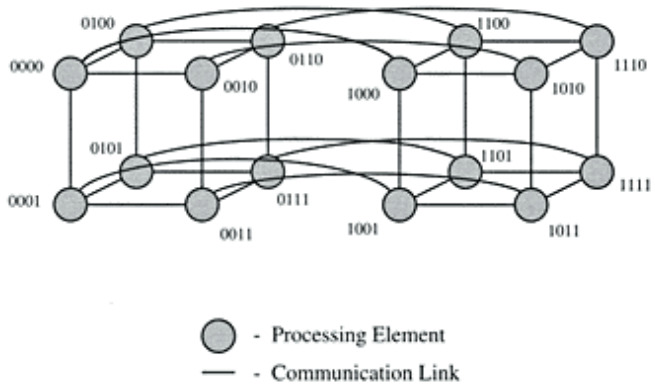
Además, para satisfacer las exigencias del enunciado, la complejidad del algoritmo no superará $O(\log_2 N)$ siendo n el número de elementos de la red (2^n). Al final del programa, el nodo rank 0 mostrará la solución por pantalla.

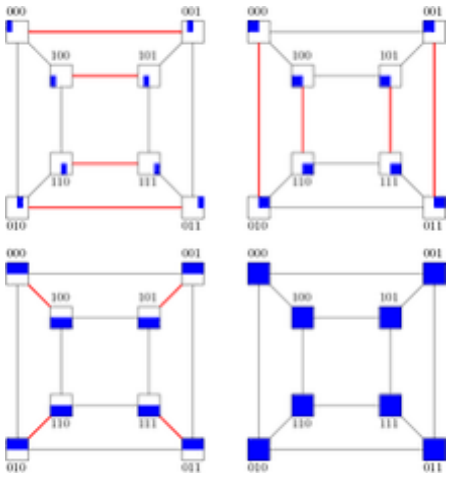
2.2 PLANTEAMIENTO DE LA SOLUCIÓN

Podemos definir un hipercubo es un tipo de topología de red que se utiliza para conectar varios nodos. Es, básicamente, una malla multidimensional en la que quedan suprimidos los nodos interiores.

El tamaño de una red hipercubo queda determinado por el grado de los nodos (2^n). Recordamos que, al igual que en el problema anterior, los nodos comienzan a enumerarse desde 0. En nuestro código, los nodos quedan numerados entre el 0 y el 15, siendo el tamaño de la red de 16 nodos con un grado 4.

Algunos ejemplos de la topología de un hipercubo son las siguientes:





Una vez definida su topología y estructura, reflejaré la problemática principal. Nuestro programa diferencia claramente entre dos funcionalidades: rank 0, que se ocupará de gran parte de la lógica del programa; y resto de nodos.

- **Rank 0:**

  Este nodo se encarga de procesar el fichero datos.dat y los datos que contiene este. Es el encargado de enviarlos a cada proceso, y en mi caso también de imprimir la secuencia de envío por pantalla.

A su vez, controlará la relación entre el número de datos contenidos en el fichero y el número de procesos lanzados para el correcto funcionamiento del programa. Controlará la ejecución del resto de procesos e imprimirá el resultado por pantalla.

- **Resto de nodos:**

  Reciben los números enviados por el rank 0, calculan sus vecinos.
  Envía el número mínimo a sus vecinos a la vez que recibe el de otros, comparando con el valor almacenado en ese momento.

Se debe cumplir una serie de circunstancias para el correcto funcionamiento del programa, como son que el número de procesos lanzados sea igual al número de nodos de la red. También, se debe garantizar que el número de datos leídos del fichero sea equivalente a la dimensión de la red.

En caso contrario, se ha proporcionado un control de errores y excepciones para evitar la finalización no controlada del programa.

Como inconveniente al desarrollo de la práctica (y pendiente de rediseño), el lado se define como una constante en el código, al igual que el número de procesos a lanzar quedan predefinidos en el Makefile. Por tanto, para una ejecución de una red con distinto lado de 4 (definido en el código) se deberá modificar previamente ambos archivos.

2.3 DISEÑO DEL PROGRAMA

En primer lugar, debo mencionar que no he conseguido que el programa funcione de la forma esperada. Creo que el flujo de datos es correcto, pero no consigo encontrar el número máximo de la red, por lo que esta parte quedará pendiente de mejora para la siguiente convocatoria.

Mediante la función *MPI_COMM_Size* somos capaces de conocer el número de procesos lanzados. Si coincide con respecto al número de nodos de la red (pow(2, L), procedemos a asignar un espacio de memoria al array números que almacenará los datos leídos del fichero mediante la función *leerdatos()*. Una vez almacenados, pasamos a comprobar si existen tantos datos como nodos en el hipercubo. En caso afirmativo, el rank 0 enviará mediante la función *enviarDatos()* (que contiene la primitiva *MPI_Send())* un double a cada nodo. Tras esto se liberará el espacio ocupado por el array números nombrado anteriormente. En caso negativo, se notificará por pantalla el error y se emitirá un Broadcast (*MPI_Bcast())* al resto de nodos del comunicador (*MPI_COMM_WORLD*) para que detengan su ejecución.

Si no se cumplía la primera de las condiciones (que el tamaño obtenido mediante *MPI_COMM_Size* sea igual al número de nodos de la red) se realizará lo mismo que hemos mencionado anteriormente. Se emitirá un broadcast al resto de procesos.

Esta serie de condiciones lo podemos controlar gracias a la variable *condición*.

Antes de la ejecución del resto de procesos, realizamos un *MPI_Broadcast()*, que como llamada colectiva que es detendrá la ejecución del resto de nodos a modo de sincronización.

Tras ello, mediante *MPI_Recv()* realizamos una espera activa (ya que es una llamada bloqueante) hasta recibir el valor del fichero proveniente del rank 0.

Mediante la función *getVecinos()* obtenemos los vecinos propios de cada nodo. La topología de un hipercubo consiste en que cada nodo itera ejecutando la función XOR(rank, 2^i) para obtener cada vecino. Por ejemplo, para un hipercubo de 4 dimensiones (como lo es en nuestro código), cada nodo tendrá desde 0 hasta 4-1 vecinos. El código es el siguiente:

```
1 void getVecinos(int rank, int vecinos[]){
2
3     int i;
4     int aux;
5
6     for(i=0;i<L;i++){
7         vecinos[i]= XOR(rank,(int)pow(2,1));
8     }
9 }
```

La función XOR ha sido predefinida en la cabecera del código:

```
#define XOR(a,b) (a^b)
```

La función *getMax()* se basa en el siguiente código para su ejecución:

```
1  double getMax(int rank, double buffer, int vecinos[]){
2
3      int i;
4
5      double max;
6
7      max = buffer;
8
9      for(i=0;i<L;i++){
10
11         if(buffer > max){
12             max = buffer;
13         }
14
15         MPI_Send(&max, 1, MPI_DOUBLE, vecinos[i],i,MPI_COMM_WORLD);
16
17         MPI_Recv(&buffer, 1, MPI_DOUBLE, vecinos[i],i,
18 MPI_COMM_WORLD,NULL);
19
20         if(buffer > max){
21             max = buffer;
22         }
23
24     }
25
26     return max;
27
 }
```

2.4 CÓDIGO FUENTE DEL PROGRAMA

```
/* ********************* BIBLIOTECAS ************************** */

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

/* *********** CONSTANTES Y VARIABLES GLOBALES***************** */

#define FICHERO "datos.dat"

/* Definimos la operación XOR que utilizaremos para encontrar vecinos
*/
#define XOR(a,b) (a^b)

#define MAX_SIZE 1024
#define L 4
#define dim (int) pow(2,L)
#define TRUE 1
#define FALSE 0

/*************** FUNCIONES AUXILIARES ***********************/

int leerdatos(double numeros[]);
void getVecinos(int rank, int *vecinos);
double getMax(int rank, double buffer, int *vecinos);
void enviarDatos(double *numeros);

/*************** FUNCION PRINCIPAL **************************/

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

    int rank, size;

    MPI_Status status;

    int vecinos[L];

    double max;
    double buffer;

    int leidos;
    double numeros[dim];

    int condicion = TRUE;

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

    if(rank == 0){
        if(size == dim) {
```

```
            double *numeros = malloc(size * sizeof(double));
            leidos = leerdatos(numeros);

            if(dim == size){
                MPI_Bcast(&condicion,1,MPI_INT,0,MPI_COMM_WORLD);

                enviarDatos(numeros);

                condicion=TRUE;
                free(numeros);
            }
            else {
                condicion = FALSE;
                fprintf(stderr, "El número de datos leídos del fichero
(%d) no es suficiente. Se necesitan %d\n", leidos, dim);

                MPI_Bcast(&condicion,1,MPI_INT,0,MPI_COMM_WORLD);
            }

        }

        else {
            condicion = FALSE;
            fprintf(stderr,"Tenemos que ejecutar %d procesos para un
hipercubo de lado %d \n",dim,L);


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

        }
    }
    MPI_Bcast(&condicion,1,MPI_INT,0,MPI_COMM_WORLD);

    if(condicion==TRUE){


        MPI_Recv(&buffer,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&sta
        tus);

        getVecinos(rank,vecinos);

        max = getMax(rank,buffer, vecinos);

        if(rank == 0)
            printf("El mayor número encontrado en la red es
%.5lf\n",max);

    }

    MPI_Finalize();
    return 0;
}


/* ********* FUNCIONES AUXILIARES ****************************/
```

```
void enviarDatos(double *numeros){
    int i;
    double buffer;

    for(i=0;i<dim;i++){
        buffer = numeros[i];
        MPI_Send(&buffer,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
        printf("%5.lf enviado al nodo %d \n",buffer,i);
    }
}


int leerdatos(double numeros[]){
    char *buffer = malloc(MAX_SIZE * sizeof(char));

    errno = 0;
    int contador = 0;
    int size = 0;
    char *aux;


    FILE *fp = fopen(FICHERO,"r");
    if(errno != 0){
        fprintf(stderr, "Se ha detectado un error (%d) al intentar
abrir el fichero\n",errno);
        return 0;
    }

    fscanf(fp,"%s",buffer);
    numeros[size++] = atof(strtok(buffer,","));

    while ((aux = strtok(NULL,","))!=NULL) {
        numeros[size++]=atof(aux);
    }

free(buffer);

fclose(fp);

return size;

}



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

    int i;
    int aux;

    for(i=0;i<L;i++){
        vecinos[i]= XOR(rank,(int)pow(2,1));
    }
}

double getMax(int rank, double buffer, int vecinos[]){
```

```
    int i;

    double max;

    max = buffer;

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

        if(buffer > max){
            max = buffer;
        }

        MPI_Send(&max, 1, MPI_DOUBLE, vecinos[i],i,MPI_COMM_WORLD);

        MPI_Recv(&buffer, 1, MPI_DOUBLE, vecinos[i],i,
MPI_COMM_WORLD,NULL);

        if(buffer > max){
            max = buffer;
        }

    }

    return max;

}
```

3. USO DE PRIMITIVAS MPI

La solución para ambos problemas ha sido resuelta mediante las primitivas MPI_Send, MPI_Recv y MPI_Bcast. Esto es posible debido a que son primitivas bloqueantes. En primera instancia me propuse realizarlo mediante el uso de primitivas asíncronas para evitar las esperas activas y bloqueos entre procesos, pero esto suponía un mayor uso de buffers que en caso de una mala lógica de programación y posible congestión podía suponer tener que usar tantos buffers como mensajes que queramos enviar. Además, también requiere que el sistema operativo aloje memoria propia, lo que supone un aumento de uso de memoria para la paralelización de las comunicaciones.

Como anotación y duda, al incluir el tag MPI_ANY_TAG en la función MPI_Send, en la cual el rank 0 propaga los diferentes valores al resto de nodos, me indicaba que era un tag incorrecto (indicaba que un proceso enviaba una señal 4) y no encontré cuál es el posible error. Por ello, he usado el tag 0, tal y como me invitaban los distintos foros donde me he apoyado para nutrirme y realizar la práctica.

4. INSTRUCCIONES PARA COMPILAR Y EJECUTAR

Gracias al uso de un archivo Makefile he automatizado la ejecución y compilación de ambos problemas.

Para facilitar esto, he utilizado en ambos casos un lado/dimensión de 4, ya que para ambos problemas el tamaño de la red será de 16 nodos y facilita el uso del fichero datos.dat. En caso contrario, deberemos compilar y ejecutar el código de forma manual.

4.1 Compilación

Para la compilación de ambos problemas únicamente debemos realizar la función:

$ make compilar

Que ejecutará internamente los siguientes comandos:

$ mpicc toroidal.c -o toroide

$ mpicc hipercubo.c -o hipercubo

4.2 Ejecución del toroide

Para la ejecución del toroide basta con llamar a la función:

$ make ejecutarToroide

Que llama a la función:

$ mpirun --hostfile my-hostfile -n 16 ./toroide

En el hostfile he indicado que la ejecución requiere de al menos 16 procesos.

4.3 Ejecución del hipercubo

Para la ejecución del hipercubo basta con llamar a la función:

$ make ejecutarHipercubo

Que llama a la función:

$ mpirun --hostfile my-hostfile -n 16 ./hipercubo

4.4 Eliminar archivos binarios creados

Si llamamos a la siguiente función, eliminaremos los ejecutables creados previamente:

$ make clean

5. REFLEXIONES

Como pequeña reflexión y conclusión del trabajo, he de mencionar que me ha resultado de gran interés realizar esta práctica ya que me ha impulsado a bucear y adentrarme en distintos foros y páginas web para encontrar información, a la vez que aprender, sobre MPI.

Como punto negativo, la insatisfacción por no haber podido lograr resolver la segunda de las problemáticas. Intentaré que quede resuelta lo antes posible y así entregarlo para la próxima evaluación o convocatoria.

6. BIBLIOGRAFÍA Y PÁGINAS DE INTERÉS

- https:/www.delftstack.com/es/howto/c/strtok-in-c/

- https:/stackoverflow.com/questions/3889992/how-does-strtok-split-the-string-into-tokens-in-c

-https:/docs.microsoft.com/es-es/cpp/c-runtime-library/reference/atof-atof-l-wtof-wtof-l?view=msvc-170

- https://en.wikipedia.org/wiki/Hypercube

- https:/hackernoon.com/how-to-solve-the-hamming-distance-problem-in-c

- A Comprehensive MPI Tutorial Resource · MPI Tutorial