PRÁCTICA 1. RED TOROIDE Y RED HIPERCUBO

Diseño de Infraestructura de Red

Jose Miguel Cabanillas Mansilla
JoseMiguel.Cabanillas@alu.uclm.es

Contenido

1.Red toroide

1. 1. Enunciado

Dado un archivo con nombre datos.dat, cuyo contenido es una lista de valores (reales) separados por comas, el problema realizará lo siguiente:
El proceso con rank=0 distribuirá a cada uno de los nodos de un toroide de lado L, los LxL 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 el caso de que todos los procesos hayan recibido su correspondiente elemento, comenzará el proceso normal del programa.
Se pide calcular el elemento menor de toda la red, el elemento con rank=0 mostrará en su salida estándar el valor obtenido.
La complejidad del algoritmo no superará O(raíz_cuadrada(n)) Con n número de elementos de la red.

1. 2. Planteamiento de la solución

Básicamente el programa girará en torno al nodo 0 (rank 0), partiendo de esa idea, vamos adjudicandole las funciones que se nos especifíca.
En primer lugar el rank 0 será el encargado de leer los datos que tengamos dentro del fichero datos.dat. Tras su lectura comprobará que se cumplen los requisitos.
Una vez procesados los datos, el rank 0 distribuirá un único dato sobre un nodo, por tanto utilizaremos los comandos MPI_Send y MPI_Recv.
Ahora, podremos calcular quién es vecino de quién, en este caso al ser una red Toroide, deberemos buscar los 4 vecinos que tendrá cada nodo. Tendremos que tener en cuenta las limitaciones que nos plantea la red Toroidal en cuanto al número de nodos.
Ya calculados los vecinos de manera satisfactoria, crearemos un método por medio del cual calcularemos el menor número que se encontraba en el fichero datos.dat. Hay que tener en cuenta que son números reales.
Cada nodo mandará su número a sus nodos vecinos, luego buscaremos el menor global dentro de la red.
Finalmente como se específica en el enunciado, el rank 0 será el encargado de mostrar cual es el número resultante que estábamos buscando.

También tendremos que tener en cuenta los posibles errores y como tratarlos. Se nos darán una serie de situaciones en las cuales tendremos que abortar el programa, ya que no se cumplirá alguno de los requisitos del enunciado o de la propia topología de la red toroidal. Estos son:
- Si el número de datos dentro del fichero datos.dat no es igual a la dimensión de nuestra red toroidal (L x L).
- Si el número de datos es distinto al número de procesos que lanzamos al ejecutar el programa
Controlando esas dos situaciones también controlamos que el número de procesos lanzados tenga que ser igual que la dimensión del toroide.

1. 3. Diseño

Para facilitar la compresión y lectura del código he creado distintos métodos dividiendo así la problemática que se nos plantea.

- Método int leerfichero(float *n,char *d). Este es el método que utilizamos para leer el fichero. El método devuelve el número de datos procesados, utilizado posteriormente para comprobar que no haya fallos en cuanto al número de datos con la dimensión de la red y el número de procesos lanzados. Como parámetros recibe dos punteros, uno float y otro char respectivamente, que son dos variables creadas en el main con un tamaño suficientemente grande para almacenar los datos. Cuando leemos el fichero escribiremos en la dirección de memoria de char *d y luego en float *n almacenaremos los números para tratarlos y distribuirlos por la red.
  Para leer los datos utilizamos las funciones fscanf, almacenando los datos en char*d, y luego con strtok, filtramos las comas, para quedarnos con los números que los almacenaremos como float en *n.
- Método void Vecinos(int rank, int l, int *vNorth, int *vSouth, int *vEast, int *vWest). Es el método que utilizamos para calcular los vecinos de un nodo, para ello, pasamos como parámetros rank del nodo que queremos calcular sus vecinos, el tamaño del lado, puesto que vamos a ir calculando por fila y columna y un puntero a cada vecino. El método no retorna nada, ya que escribe el valor de los vecinos directamente sobre la memoria de los punteros.
- Método calculoMínimo(int rank,float buffer,int lado,int vEast,int vWest,int vNorth,int vSouth). Con este método calculamos el valor mínimo de la red. Introducimos el rank del nodo actual, el buffer utilizado para la comunicación con MPI_Send y MPI_recv, el lado ya que calcularemos en orden de columna y fila y los vecinos del rank actual.
  Básicamente iteramos sobre la fila y comprobamos si el valor es mínimo de entre el rank y sus vecinos y lo vamos almacenando en la variable global, que es la que retornamos cuando hemos acabado de iterar entre los todos los nodos.

Una vez comentados los métodos utilizados nos vamos al main:

En el main utilizamos el rank 0, para realizar todas las comprobaciones:

- En primer lugar

```
if(numDatos!=(L*L)){
    printf("El numero de datos generados no es compatible con la dimension del toroide\n");
    flag=0;
    MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);
}
```

Con esta parte del código comprobamos que el número de datos sea igual a LxL, es decir sea igual que la dimensión cuadrática de nuestro toroide.
Podemos ver que tenemos MPI_Bcast, esto es un comando de mpi por medio del cual vamos a comunicar a todos los demás nodos algo, más tarde veremos que junto a flag es el mecanismo que tenemos para que si sucede un error se le comunique a los demás que finalizarán sin ningún problema.

- Esta parte también se corresponde con la comprobación de errores

```
if(numDatos!=size){
    printf("El numero de procesos %d es distinto al numero de dat
    flag=0;
     MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);
```

Aquí comprobamos que el número de datos leídos del fichero sea igual que el número de procesos lanzados y si eso no ocurre se realiza el procedimiento anterior.
Si ninguno de los dos casos anteriores supone un error, enviamos los números a cada nodo.

```
for(i=0;i<size;i++){
    buffer=numeros[i];
    MPI_Send(&buffer,1,MPI_FLOAT,i,0,MPI_COMM_WORLD);
}
```

Una vez que el rank 0 ha realizado estas acciones, realizamos un MPI_Bcast. Si ha habido un error en las anteriores secciones del código, ahora recibiremos un 0, lo que evitará que entremos en la siguiente parte del código, si por el contrario no ha habido ningún error, se introducirá un 1 y por tanto accederemos aquí

```
MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);

if(flag!=0){
    MPI_Recv(&buffer,1,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&statu
    Vecinos(rank,L,&vNorth,&vSouth,&vEast,&vWest);
    global=calculoMinimo(rank,buffer,L,vEast,vWest,vNorth,vSouth);

    if(rank==0){
        printf("El dato cuyo valor es menor en la red es %3.2f\n",gl
    }
}
```

Aquí podemos observar el MPI_Bcast del que ya hemos dicho su función, si no recibe nada, entraremos en el bloque del if.
Recibiremos los números mandados por el rank0, llamaremos al método Vecinos y luego al método calculoMínimo cuyo valor de retorno guardaremos en la variable global. Luego el rank 0 será el encargado de mostrar tal valor que es el mínimo de la red.

1. 4. Código fuente

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"
#define DATOS "datos.dat"
//#define L 4
#define STAN_SIZE 1024

int leerfichero(float *n,char *d);
void Vecinos(int rank, int sq, int *vNorth, int *vSouth, int *vEast, int *vWest);
float calculoMinimo(int rank,float buffer,int sq,int vEast,int vWest,int vNorth,int vSouth);

int main(int argc, char *argv[])
{
	float *numeros=malloc(STAN_SIZE*sizeof(float));
	char *datos=malloc(STAN_SIZE*sizeof(char));
	int i,flag=1,rank,size,numDatos,vNorth,vSouth,vEast,vWest;
	float global,buffer;
	MPI_Status status;
	MPI_Init(&argc,&argv);
	MPI_Comm_rank(MPI_COMM_WORLD,&rank);
	MPI_Comm_size(MPI_COMM_WORLD,&size);

	if (rank==0){
		numDatos=leerfichero(numeros,datos);
		if(numDatos!=(L*L)){
			printf("No se puede generar una red toroidal con los datos especificados\n");
			flag=0;
			MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);
		}
		if(numDatos!=size){
		  printf("El numero de procesos %d es distinto al numero de datos %d\n",size,numDatos);
		  flag=0;
			MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);

		}
		for(i=0;i<size;i++){
			buffer=numeros[i];
			MPI_Send(&buffer,1,MPI_FLOAT,i,0,MPI_COMM_WORLD);
		}
	}

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

	if(flag!=0){
		MPI_Recv(&buffer,1,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
		Vecinos(rank,L,&vNorth,&vSouth,&vEast,&vWest);
		global=calculoMinimo(rank,buffer,L,vEast,vWest,vNorth,vSouth);

		if(rank==0){
			printf("El dato cuyo valor es menor en la red es %3.2f\n",global);
```

```
                }
    }

        MPI_Finalize();
        free(numeros);
        free(datos);
        return 0;
}

void Vecinos(int rank, int l, int *vNorth, int *vSouth, int *vEast, int *vWest){
        int fila,columna;
        fila=rank/l;
        columna=rank%l;

        //Saco el nodo al SUR del rank actual
        if(fila==0){
                *vSouth=((l-1)*l)+columna;
        }
        else{
                *vSouth=((fila-1)*l)+columna;
        }
        //Sacaremos el nodo al NORTE del rank actual

        if(fila==l-1){
                *vNorth=columna;
        }
        else{
                *vNorth=((fila+1)*l)+columna;
        }
        //Sacaremos el nodo al OESTE(izquierda) del rank actual
        if(columna==0){
                *vWest=(fila*l)+(l-1);
        }else{
                *vWest=(fila*l)+(columna-1);
        }
        //Sacamos el nodo al ESTE(derecha) del rank actual
        if((columna+1)==l){
                *vEast=fila*l;
        }
        else{
                *vEast=(fila*l)+(columna+1);
        }
}


int leerfichero(float *n,char *d){

FILE *fp;  //Descriptor del archivo
char *f; //Puntero que vamos a utilizar para ir recogiendo el valor de strtok
int i=0; //Numero de datos finales que tendremos en Numeros

if((fp=fopen(DATOS,"r"))==NULL){
        fprintf(stderr,"Error al abrir el archivo %s\n",DATOS);
```

```
        return EXIT_FAILURE;
}
//almacenamos los string dentro del puntero datos
fscanf(fp,"%s",d);
fclose(fp);
n[i++] = atof(strtok(d,",")); //Guardamos el primer elemento en numeros
while( (f=strtok(NULL,",")) != NULL){
        n[i++] =atof(f);//Recorremos hasta que sea NULL, es decir el ultimo elemento
}
return i; // a la vez vamos almacenando el numero de datos que tenemos en el datos.dat
}
float calculoMinimo(int rank,float buffer,int lado,int vEast,int vWest,int vNorth,int vSouth){
        int i;
        float global=3000,minimo;
        MPI_Status status;
        //Calculamos en orden de fila
        for(i=0;i<lado;i++){
                minimo=buffer;
                if(buffer<global){
                        global=minimo;
                }
                MPI_Send(&global,1,MPI_FLOAT,vEast,i,MPI_COMM_WORLD);
                MPI_Recv(&buffer,1,MPI_FLOAT,vWest,i,MPI_COMM_WORLD,&status);
                if(buffer<global){
                        global=buffer;
                }
        }
        //Calculamos en orden de columna
        for(i=0;i<lado;i++){
                MPI_Send(&global,1,MPI_FLOAT,vNorth,i,MPI_COMM_WORLD);
                MPI_Recv(&buffer,1,MPI_FLOAT,vSouth,i,MPI_COMM_WORLD,&status);
                if(buffer<global){
                        global=buffer;
                }
        }
return global;
}
```

2. Red hipercubo

2.1. 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 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 O(logaritmo_base_2(n)) Con n número de elementos de la red.

2.2. Planteamiento de la solución

Básicamente el programa girará en torno al nodo 0 (rank 0), partiendo de esa idea, vamos adjudicandole las funciones que se nos especifíca.
En primer lugar, el rank 0 será el encargado de leer los datos que tengamos dentro del fichero datos.dat. Tras su lectura comprobará que se cumplen los requisitos.
Una vez procesados los datos, el rank 0 distribuirá un único dato sobre un nodo, por tanto, utilizaremos los comandos MPI_Send y MPI_Recv.
Ahora, podremos calcular quién es vecino de quién, en este caso al ser una red Hipercubo, deberemos buscar los D (Dependerá de la dimensión) vecinos que tendrá cada nodo. Tendremos que tener en cuenta las limitaciones que nos plantea la red Toroidal en cuanto al número de nodos.
Ya calculados los vecinos de manera satisfactoria, crearemos un método por medio del cual calcularemos el menor número que se encontraba en el fichero datos.dat. Hay que tener en cuenta que son números reales.
Cada nodo mandará su número a sus nodos vecinos, luego buscaremos el menor global dentro de la red.
Finalmente, como se específica en el enunciado, el rank 0 será el encargado de mostrar cual es el número resultante que estábamos buscando.

También tendremos que tener en cuenta los posibles errores y como tratarlos. Se nos darán una serie de situaciones en las cuales tendremos que abortar el programa, ya que no se cumplirá alguno de los requisitos del enunciado o de la propia topología de la red toroidal. Estos son:
- Si el número de datos dentro del fichero datos.dat no es igual a la dimensión de nuestra hipercubo (2 ^ D siendo D la dimensión).
- Si el número de datos es distinto al número de procesos que lanzamos al ejecutar el programa
Controlando esas dos situaciones también controlamos que el número de procesos lanzados tenga que ser igual que la dimensión del hipercubo.

2.3. Diseño

Para facilitar la compresión y lectura del código he creado distintos métodos dividiendo así la problemática que se nos plantea.

- Método int leerfichero(float *n,char *d). Este es el método que utilizamos para leer el fichero. El método devuelve el número de datos procesados, utilizado posteriormente para comprobar que no haya fallos en cuanto al número de datos con la dimensión de la red y el número de procesos lanzados. Como parámetros recibe dos punteros, uno float y otro char respectivamente, que son dos variables creadas en el main con un tamaño suficientemente grande para almacenar los datos. Cuando leemos el fichero escribiremos en la dirección de memoria de char *d y luego en float *n almacenaremos los números para tratarlos y distribuirlos por la red.
  Para leer los datos utilizamos las funciones fscanf, almacenando los datos en char*d, y luego con strtok, filtramos las comas, para quedarnos con los números que los almacenaremos como float en *n.
- Método void Vecinos(int rank,int dimension,int *vecinos). Es el método que utilizamos para calcular los vecinos de un nodo, para ello, pasamos como parámetros rank del nodo que queremos calcular sus vecinos, la dimensión del hipercubo y una matriz de enteros donde iremos almacenando los vecinos de ese rank. El método no retorna nada, ya que escribe el valor de los vecinos directamente sobre la memoria de la matriz de enteros.
- Método calculoMaximo(int rank,float buffer,int dimension, int *vecinos). Con este método calculamos el valor máximo de la red. Introducimos el rank del nodo actual, el buffer utilizado para la comunicación con MPI_Send y MPI_recv, la dimensión y los vecinos del rank actual.

Básicamente iteramos sobre los vecinos y comprobamos si el valor es maximo de entre el rank y sus vecinos y lo vamos almacenando en la variable global, que es la que retornamos cuando hemos acabado de iterar entre los todos los nodos.

Una vez comentados los métodos utilizados nos vamos al main:

En el main utilizamos el rank 0, para realizar todas las comprobaciones:

- En primer lugar

Con esta

```
if(numDatos!=size){
    printf("El numero de procesos %d es distinto al numero de da
    flag=0;
    MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);// Envio el codi
```

parte del código comprobamos que el número de datos sea igual al número de procesos lanzados. Podemos ver que tenemos MPI_Bcast, esto es un comando de mpi por medio del cual vamos a comunicar a todos los demás nodos algo, más tarde veremos que junto a flag es el mecanismo que tenemos para que si sucede un error se le comunique a los demás que finalizarán sin ningún problema.

- Esta parte también se corresponde con la comprobación de errores

```
if(log2(numDatos) != (DIMENSION)){
    printf("El numero de datos generados no es apto para la Dimension que tenemos\n");
    flag=0;
    MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD); // Envio el código de error a todos los nodos
}
```

Aquí comprobamos que el logaritmo en base 2 de datos leídos del fichero sea igual que la dimensión de la red del hipercubo

- Si ninguno de los dos casos anteriores supone un error, enviamos los números a cada nodo.

```
for(i=0;i<size;i++){
    buffer=numeros[i];
    MPI_Send(&buffer,1,MPI_FLOAT,i,0,MPI_COMM_WORLD);
}
```

Una vez que el rank 0 ha realizado estas acciones, realizamos un MPI_Bcast. Si ha habido un error en las anteriores secciones del código, ahora recibiremos un 0, lo que evitará que entremos en la siguiente parte del código, si por el contrario no ha habido ningún error, se introducirá un 1 y por tanto accederemos aquí

```
MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD); // Si se ha producido a
//el valor de flag cambiado, no pudiendo entrar así en la condición

if(flag != 0){
    MPI_Recv(&buffer,1,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&statu
    VecinosHiper(rank,DIMENSION,vecinos);
    global=calculoMaximo(rank,buffer,DIMENSION,vecinos);

    if(rank==0){
        printf("El dato cuyo valor es mayor en la red es %3.2f\n",gl
    }
}
```

Aquí podemos observar el MPI_Bcast del que ya hemos dicho su función, si no recibe nada, entraremos en el bloque del if.
Recibiremos los números mandados por el rank0, llamaremos al método Vecinos y luego al método calculoMaximo cuyo valor de retorno guardaremos en la variable global. Luego el rank 0 será el encargado de mostrar tal valor que es el mínimo de la red.

2.4. Código fuente

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

//#define DIMENSION 4
#define DATOS "datos.dat"
#define STAN_SIZE 1024

float calculoMaximo(int rank,float buffer,int dimension,int *vecinos);
int leerfichero(float *n,char *d);
void VecinosHiper(int rank,int dimension,int *vecinos);

int main(int argc, char *argv[])
{
	float *numeros=malloc(STAN_SIZE*(sizeof(float)));
	char *datos=malloc(STAN_SIZE*sizeof(char));
	int rank,size,i,flag=1;
	int vecinos[DIMENSION];
	float buffer,global;
	MPI_Status status;

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


	if (rank==0){
		int numDatos=leerfichero(numeros,datos);
		if(log2(numDatos) != DIMENSION){
			printf("No se puede generar una red hipercubo con los datos especificados\n");
			flag=0;
			MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD); // Envio el código de error a
todos los nodos
		}
		if(numDatos!=size){
		   printf("El numero de procesos %d es distinto al numero de datos %d\n",size,numDatos);
		   flag=0;
			MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD);// Envio el codigo de error a
todos los nodos

		}
		for(i=0;i<size;i++){
			buffer=numeros[i];
			MPI_Send(&buffer,1,MPI_FLOAT,i,0,MPI_COMM_WORLD);
		}
	}
	MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD); // Si se ha producido algún error recibiría
	//el valor de flag cambiado, no pudiendo entrar así en la condición
```

```
	if(flag != 0){
		MPI_Recv(&buffer,1,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
		VecinosHiper(rank,DIMENSION,vecinos);
		global=calculoMaximo(rank,buffer,DIMENSION,vecinos);

		if(rank==0){
			printf("El dato cuyo valor es mayor en la red es %3.2f\n",global);
		}
	}

	MPI_Finalize();
	free(numeros);
	free(datos);
	return 0;
}


int leerfichero(float *n,char *d){

FILE *fp;  //Descriptor del archivo
char *f; //Puntero que vamos a utilizar para ir recogiendo el valor de strtok
int i=0; //Numero de datos finales que tendremos en Numeros

if((fp=fopen(DATOS,"r"))==NULL){
	fprintf(stderr,"Error al abrir el archivo %s\n",DATOS);
	return EXIT_FAILURE;
}
//almacenamos los string dentro del puntero datos
fscanf(fp,"%s",d);
fclose(fp);
n[i++] = atof(strtok(d,",")); //Guardamos el primer elemento en numeros
while( (f=strtok(NULL,",")) != NULL){
	n[i++] =atof(f);//Recorremos hasta que sea NULL, es decir el ultimo elemento
}
return i; // a la vez vamos almacenando el numero de datos que tenemos en el datos.dat
}

float calculoMaximo(int rank,float buffer,int dimension,int *vecinos){
	int i;
	float global=-3000,mayor;
	MPI_Status status;

	for(i=0;i<dimension;i++){
		mayor=buffer;
		if(buffer>global){
			global=mayor;
		}
		MPI_Send(&global,1,MPI_FLOAT,vecinos[i],i,MPI_COMM_WORLD);
		MPI_Recv(&buffer,1,MPI_FLOAT,vecinos[i],i,MPI_COMM_WORLD,&status);
		if(buffer>global){
			global=buffer;
		}
	}
```

```
return global;
}

void VecinosHiper(int rank,int dimension,int *vecinos){
int i;

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

}
```

3. Explicación del flujo de datos en la red para cada comando MPI

Con MPI_Send(&buffer,1,MPI_FLOAT,i,0,MPI_COMM_WORLD) mandamos un dato almacenado en el buffer, que será un dato FLOAT, y lo mandaremos al destino i,(que dependerá del rank), por eso lo introduciremos dentro de un bucle for. Comunicaremos a los destinos que esten dentro del MPI_COMM_WORLD.
Con MPI_Recv(&buffer,1,MPI_FLOAT,0,MPI_ANY_TA,MPI_COMM_WORLD,&status) recibimos uno dato, que se encuentra en el buffer, que será un dato FLOAT, procedente del grupo MPI_COMM_WORLD, con un status que nos dará información sobre la operación.
Con MPI_Bcast(&flag,1,MPI_INT,0,MPI_COMM_WORLD) utilizaremos dos veces esta operación, una para mandar un 1 o un 0 que será la variable flag a todos los nodos de la red. En otra parte del código tendremos la misma operación que esperará la recepción de ese 0 o 1 permitiéndonos entrar o no en la sección de código que tenemos acceso por medio de la variable flag.

4. Instrucciones de como compilar y ejecutar

Se ha realizado un Makefile para facilitar el proceso de compilado y ejecución de los programas Además se ha realizado un programa en C que genera número reales aleatorios y los guarda en un fichero cuyo nombre es *datos.dat.*
En primer lugar ejecutaremos **make** en nuestro directorio, para borrar en caso de que los haya programas de anteriores ejecuciones. También compilamos el programa GenerarNumerosAleatorios.c

- **make**

Si directamente deseamos compilar y ejecutar un caso de prueba introducimos lo siguiente:

- **make toroideFallo1**. Nos dará un error controlado al ejecutar. Lanzamos 15 procesos para 16 datos
- **make toroideFallo2**. No podemos generar la red toroidal porque el número de datos es distinto a la dimensión de la red (LxL)
- **make toroideBueno1**. Caso de ejecución correcta.
- **make toroideBueno2**. Caso de ejecución correcta.

- **make hipercuboFallo1**. No podemos generar la red de hipercubo porque el número de datos es distinto a la dimensión de la red (2^x)
- **make hipercuboFallo2.** Nos dará un error controlado al ejecutar este programa. Lanzamos 15 procesos para 16 datos
- **make hipercuboBueno1**. Caso de ejecución correcta.
- **make hipercuboBueno2**. Caso de ejecución correcta.

5. Conclusiones

He comprendido y aprendido el funcionamiento básico de MPI. Ha sido una práctica que me ha resultado algo compleja, he tenido que recurrir a otras fuentes a parte de los apuntes para poder realizarla.

A pesar de ello el resultado final es el esperado.

El utilizar MPI ha facilitado la funcionalidad de la práctica, ya que sin este no podríamos haberla resuelto. Si nos fijamos bien, el código entre un enunciado y otro difiere en el algoritmo utilizado para calcular los vecinos, que está relacionado directamente con la topología de la red utilizada, lo demás ha sido todo igual.

MPI por tanto es una especificación de una librería muy potente y versátil, por medio de la cual podremos resolver problemas muchos más complejos que los propuestos, de manera mucho más sencilla que si los resolviésemos sin la misma