5-4-2019

MPI PRACTICA 1

Red Toroide y Red Hipercubo

ANDRÉS GUTIÉRREZ CEPEDA

Índice de contenidos:

Red Toroide:

Enunciado del Problema:

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

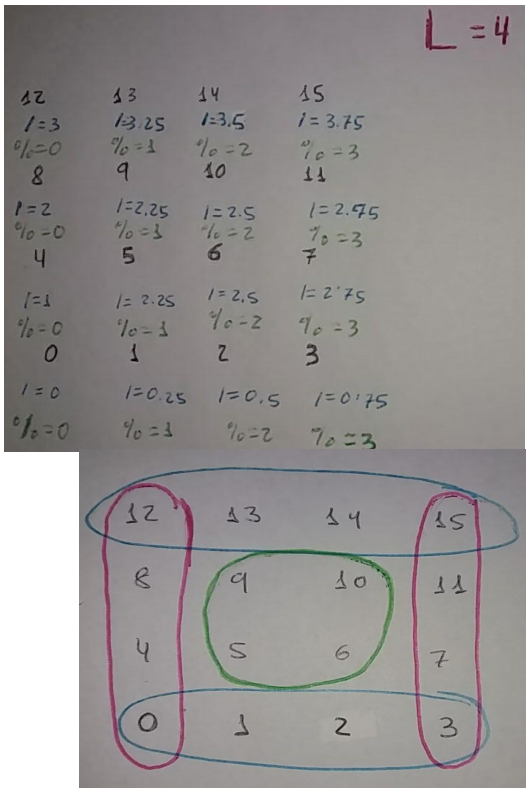
El proceso de rank 0 distribuirá a cada uno de los nodos de un toroide de lado L, los L x L números reales que estarán contenidos en el archivo datos.dat. En caso de que no se hayan lanzado suficientes elementos de proceso para los datos del programa, este emitirá un error y todos los procesos finalizaran.

En caso de que todos los procesos han recibido su correspondiente elemento, comenzará el proceso normal del programa. Se pide calcular el elemento menor de toda la red, el elemento de proceso con rank 0 mostrará en su salida estándar el valor obtenido.

La complejidad del algoritmo no superará O(raiz_cuadrada(n)) Con n número de elementos de la red.

Planteamiento de la solución:

El problema radica principalmente en la complejidad que tiene que tener el algoritmo de búsqueda del numero menor, para conseguir esta complejidad se hará uso de las funcionalidades de MPI.
La primera tarea para realizar será la de identificar los vecinos que tiene un nodo cualquiera de la red toroide dentro de la misma, para lo cual he planteado la siguiente solución:



De esta manera mediante la división del identificador del nodo por el lado podemos identificar la fila y con el modulo del identificador del nodo por el lado obtenemos la columna. De esta manera básica podemos identificar y acceder a todos los nodos de la matriz a través de uno dado, esto se vera de manera mas concreta en la explicación del código. Como podemos observar en la imagen inferior, para conocer los vecinos este y oeste de las zonas rodeadas en rosa tendremos que hacer un trato especial, dado que sus vecinos no se encuentran directamente al lado de estos si no en el otro extremo, al igual que para los vecinos norte y sur de las zonas azules, esto lo veremos mas adelante en el código.

Para conseguir la complejidad necesaria se realizarán envíos y recepciones de los nodos vecinos para encontrar el numero menor de esta manera:

For(i=0;i<L;i++){
Enviar a norte (mi numero);
Recibir de sur (su numero);
Mi numero=minimo(mi numero, su numero);
}
For(i=0;i<L;i++){
Enviar a este (mi numero);
Recibir de Oeste (su numero);
Mi numero=minimo(mi numero, su numero);
}

De esta manera todos los nodos tienen el nodo de menor valor, que será imprimido por el nodo 0.

Diseño de programa:

```
8   #define MAX_DATOS 1024
9   #define NORTE 0
10  #define SUR 1
11  #define ESTE 2
12  #define OESTE 3
13  #define L 4
14
15  int* vecinosToroide(int nodo);
16
17  int contadorDatos();
```

Definimos norte, sur, este y oeste como los identificadores del vector de vecinos y definimos el tamaño del lado del toroide para que sea mas sencillo trabajar con el programa como se nos recomendó en clase. Definimos dos funciones, la que comprobara los vecinos del toroide y la que contara los datos para comprobar si son suficientes para crear la red toroide.

```
19  int main(int argc, char *argv[]){
20      int i=0;
21
22      int* vecinoToroide;
23      double numero = 0;
24      double numeroNuevo = 0;
25      double valores[L*L];
26      char* char_fichero;
27      double valor;
28      char linea[80];
29      char *token;
30      const char separador[2] = ",";
31      FILE *archivo;
32      int datos;
33      int continuar=0;
34
35      MPI_Init(&argc,&argv);
36      MPI_Status status;
37      int size,rank;
38      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
39      MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Declaramos un array de vecinos, el numero que tendrá cada nodo, el que recibirá cada nodo y los valores que se leerán del fichero, se declara un char que se leerá desde el fichero, el valor que será leído y la línea que se va a leer desde el fichero datos.dat, el token que se creara al partir la línea por el separador ‘,’, el archivo que se leerá, los datos que contiene el fichero y un int continuar que indicara si todos los datos son correctos y se puede continuar con la ejecución del programa. Se iniciará el programa MPI y se inicializaran el Rank de cada nodo y el tamaño de nodos que hay.

```
42      if(rank==0){
43          datos=contadorDatos();
44
45          if(datos>size){
46              continuar=1;
47              printf("El numero de nodos a ejecutar es menor que el numero de datos\r\n");
48              MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
49          }else{
50              if(L*L==size){
51                  continuar=0;
52                  printf("El numero de nodos a ejecutar correcto\r\n");
53                  MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
54              }else{
55                  continuar=1;
56                  printf("El numero de nodos ejecutados es distinto a los nodos requeridos\r\n");
57                  MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
58              }
59
60          }
```

Si el Rank del proceso es 0 se encargará de lanzar el conteo de datos. Si estos datos son mayores que el numero de nodos lanzado por línea de comandos el programa se detendrá, se enviara a todos los procesos un valor por Bcast que indica que no deben continuar con el programa. En caso de que procesos suficientes para los datos se comprobara que esos datos son suficientes para rellenar toda la red toroide, si son suficientes se continuara la ejecución normal del código mediante un mensaje Bcast al igual que en el caso anterior, en caso de no ser suficientes se detendrá de la misma manera.

```
62      if(continuar==0){
63
64      if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
65          printf("Error al abrir el fichero\n");
66          MPI_Finalize();
67          exit(EXIT_FAILURE);
68      }
69
70      char_fichero=fgets(linea,MAX_DATOS*sizeof(char),archivo);
71
72      token = strtok(linea, separador);
73
74      while( token != NULL ) {
75          valores[i++]=atof(token);
76          token = strtok(NULL, separador);
77      }
78
79      fclose(archivo);
80
81      for(i=1;i<(L*L);i++){
82          numero=valores[i];
83          MPI_Bsend(&numero,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
84      }
85      numero=valores[0];
86      vecinoToroide=vecinosToroide(rank);
87      for (i=0;i<L;i++){
88          MPI_Bsend(&numero,1,MPI_DOUBLE,vecinoToroide[NORTE],0,MPI_COMM_WORLD);
89          MPI_Recv(&numeroNuevo,1,MPI_DOUBLE,vecinoToroide[SUR],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
90          if(numeroNuevo<numero){
91              numero=numeroNuevo;
92          }
93      }
94      for (i=0;i<L;i++){
95          MPI_Bsend(&numero,1,MPI_DOUBLE,vecinoToroide[ESTE],0,MPI_COMM_WORLD);
96          MPI_Recv(&numeroNuevo,1,MPI_DOUBLE,vecinoToroide[OESTE],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
97          if(numeroNuevo<numero){
98              numero=numeroNuevo;
99          }
100     }
101     printf("Numero minimo %lf\n",numero);
102     }
```

En caso de que se deba continuar con la ejecución del programa, el nodo cero se encargara de leer los datos del fichero, peticionarlos de manera individual y enviarlos mediante MPI_Bsend al resto de nodos del programa. Una vez ha enviado todos los números, este se quedara con uno también y descubrirá cuales son sus vecinos, cuando sabe sus vecinos comenzara enviando su numero con un Bsend a su vecino norte y recibirá el de su vecino sur con un MPI_Recv, si el numero recibido es menor que el numero que el tenia lo reemplazara por este, quedándose así con el menor de los dos, repetirá este proceso tantas veces como lado tenga el Toroide, una vez la columna tenga en numero menor, realizara el mismo proceso para los vecinos este y oeste, quedándose así en la red toroide el menor valor de los datos. El proceso de rango 0 imprimirá este valor obtenido.

```
103     }else{
104
105         MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
106         if(continuar==0){
107
108         MPI_Recv(&numero,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
109         vecinoToroide=vecinosToroide(rank);
110         for (i=0;i<L;i++){
111             MPI_Bsend(&numero,1,MPI_DOUBLE,vecinoToroide[NORTE],0,MPI_COMM_WORLD);
112             MPI_Recv(&numeroNuevo,1,MPI_DOUBLE,vecinoToroide[SUR],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
113             if(numeroNuevo<numero){
114                 numero=numeroNuevo;
115             }
116         }
117         for (i=0;i<L;i++){
118             MPI_Bsend(&numero,1,MPI_DOUBLE,vecinoToroide[ESTE],0,MPI_COMM_WORLD);
119             MPI_Recv(&numeroNuevo,1,MPI_DOUBLE,vecinoToroide[OESTE],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
120             if(numeroNuevo<numero){
121                 numero=numeroNuevo;
122             }
123         }
124         }
125     }
126
127     MPI_Finalize();
128
129     return EXIT_SUCCESS;
130 }
```

```
132  int* vecinosToroide(int nodo){
133
134      static int vecinos [L];
135
136      int modulo = nodo%L;
137      int division  = nodo/L;
138
139      switch(modulo){
140          case (L-1):
141              vecinos[ESTE]=nodo-1;
142              vecinos[OESTE]=nodo-(L-1);
143          break;
144          case 0:
145              vecinos[ESTE]=nodo+(L-1);
146              vecinos[OESTE]=nodo+1;
147          break;
148          default:
149              vecinos[ESTE]=nodo-1;
150              vecinos[OESTE]=nodo+1;
151          break;
152      }
153
154      switch(division){
155          case (L-1):
156              vecinos[SUR]=nodo-L;
157              vecinos[NORTE]=nodo-(L*(L-1));
158          break;
159          case 0:
160              vecinos[SUR]=nodo+(L*(L-1));
161              vecinos[NORTE]=nodo+L;
162          break;
163          default:
164              vecinos[SUR]=nodo-L;
165              vecinos[NORTE]=nodo+L;
166          break;
167      }
168
169      return vecinos;
170  }
```

Cuando se trata del resto de nodos harán un trabajo muy similar al del nodo 0, estos recibirán el numero indicatorio de si deben continuar o no con la ejecución del programa, si tienen que seguir con la ejecución distribuirán su numero entre los vecinos y se quedaran con el menor.

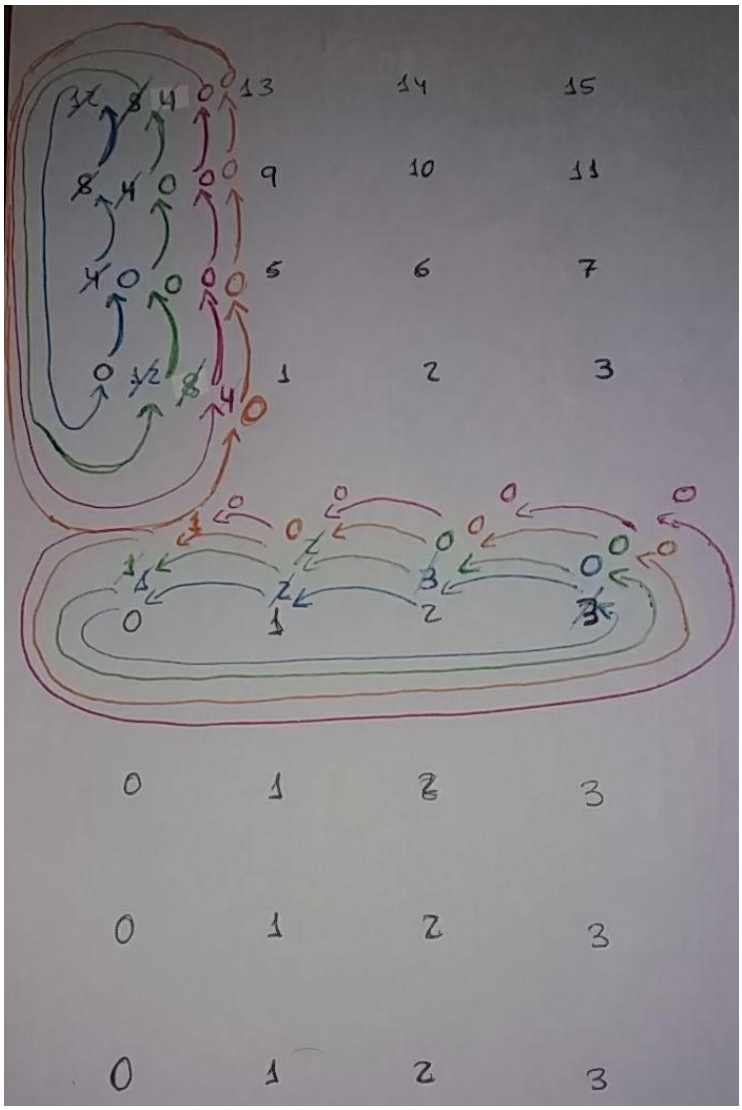
Para conocer los vecinos de un nodo dado se hará como antes se menciono mediante el modulo y la fracción. Si el modulo es 3 nos encontraremos en la cuarta columna, en la cual tenemos que hacer un trato especial para encontrar el vecino oeste, para encontrar este vecino será el Rank del nodo menos el tamaño del lado menos uno, para encontrar el vecino este será de manera normal, restándole 1 al Rank del nodo. Cuando el modulo da cero nos encontramos en la primera columna y para ello realizaremos las mismas operaciones, pero sumando en lugar de restando, en el resto de los casos realizaremos un tratamiento especial. Para encontrar los vecinos norte y sur lo realizaremos mediante el uso de la división, para ello en caso de que la división de 3 nos encontraremos en la fila superior, que necesitara un tratamiento especial de su vecino norte, que será el nodo restado a la multiplicación del lado multiplicado por el lado menos uno, en caso de que la división de cero, nos encontramos en la fila inferior y será el mismo procedimiento, esta vez para sur pero sumando, en el resto de casos se realizaran las operaciones por defecto.

```
172  int contadorDatos(){
173      int datos=0;
174
175      char* char_fichero;
176      char linea[80];
177      char *token;
178      const char separador[2] = ",";
179      FILE *archivo;
180
181      if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
182              printf("Error al abrir el fichero\n");
183              exit(EXIT_FAILURE);
184          }
185
186          char_fichero=fgets(linea,MAX_DATOS*sizeof(char),archivo);
187
188          token = strtok(linea, separador);
189
190          while( token != NULL ) {
191              datos++;
192              token = strtok(NULL, separador);
193          }
194
195          fclose(archivo);
196
197          return datos;
198  }
```

Para comprobar la cantidad de datos leídos desde el fichero de datos.dat, realizamos una función de lectura que cuente la cantidad de númeroos recogidos de ese fichero.

Explicación del flujo de datos:

El flujo de datos comienza desde el nodo 0, el cual envía un mensaje Bcast a todos los demás nodos, indicándoles si estos deben o no deben seguir con la ejecución del programa, en el caso de que deban seguir con la ejecución del código, este nodo 0 leerá todos los números indicados en el fichero datos.dat y los distribuirá de manera individual entre cada uno de los nodos de la red toroide, el resto de nodos recibirán este mensaje y tomaran el numero que les ha sido enviado. Una vez tienen su numero comenzara con el flujo de datos de la búsqueda del valor mínimo.



El flujo de datos se puede representar de esta manera, los nodos envían sus números a los vecinos superiores y reciben de los números de los nodos inferiores, si estos son menores lo remplazan por su numero que fue entregado por el nodo 0. Esto lo repiten un total de L veces y luego lo repiten para sus vecinos laterales de la misma manera, para tener así todos los valores mínimos como se puede mostrar en la imagen. El color azul representa el primer envió y recepción, el verde el segundo, el rosa el tercero y el naranja el cuarto, en el caso de ejemplo de una red toroide de lado 4.

Instrucciones de como compilar y ejecutar:

Para realizar la compilación se hará mediante el makefile, mediante el uso del comando *make all*, una vez este compilado podremos hacer dos pruebas básicas, colocando 16 datos en el archivo datos.dat podremos ejecutar el comando *testToroide.*

Conclusiones:

Como conclusiones podemos destacar el alto grado de eficiencia, paralelización y control que podemos tener sobre la red gracias a MPI. Dado que sin este la eficiencia se reduciría en L veces.

Red Hipercubo:

Enunciado del Problema:

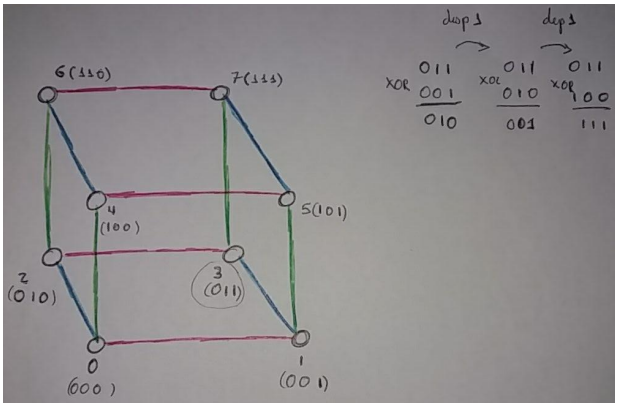
Dado un archivo con nombre datos.dat, cuyo contenido es una lista de valores separados por comas, nuestro programa realizará lo siguiente:

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, esté 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.

Planteamiento de la solución:



Para trabajar con la red hipercubo trabajaremos con los nodos a nivel de bit, para identificar los nodos lo haremos a nivel de bit, dado que un nodo solo se diferenciará con sus vecinos en un único bit, de esta manera mediante la función xor podremos obtener los vecinos de un nodo dado, esto se explicará mas detalladamente en el código del programa.

Para conseguir la complejidad necesaria se realizarán envíos y recepciones de los nodos vecinos para encontrar el numero menor de esta manera:

```
For(i=0;i<D;i++){
        Enviar (mi numero, vecino dimensión iesima);
        Recibir (su numero, vecino dimensión iesima);
        Mi numero=máximo (mi numero, su numero);
}
```

De esta manera todos los nodos tienen el nodo de mayor valor, que será imprimido por el nodo 0.

Diseño de programa:

```c
#define MAX_DATOS 1024
#define D 3

int* vecinosHipercubo(int nodo);

int contadorDatos();

```

Definimos el tamaño del lado del hipercubo para que sea mas sencillo trabajar con el programa como se nos recomendó en clase. Definimos dos funciones, la que comprobara los vecinos del hipercubo y la que contara los datos para comprobar si son suficientes para crear la red hipercubo.

```c
int main(int argc, char *argv[]){
    int i=0;
    int* vecinoHipercubo;
    double numero = 0;
    double numeroNuevo = 0;
    int tamanoVector=pow(2,D);
    double valores[tamanoVector];
    char* char_fichero;
    double valor;
    char linea[80];
    char *token;
    const char separador[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 numero de cada nodo, el numero recibido del vecino, el vector de valores, el valor que se leerá de fichero, el carácter y línea que se leerá desde el fichero, el token que contendrá los valores de la línea separados por ‘,’, el fichero, y el rank y size de los nodos.

```c
if(rank==0){
    datos=contadorDatos();

    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(pow(2,D)==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\n");
        exit(EXIT_FAILURE);
    }

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

    token = strtok(linea, separador);

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

    fclose(archivo);

    for(i=1;i<tamanoVector;i++){
        numero=valores[i];
        MPI_Bsend(&numero,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
    }
    numero=valores[0];

    vecinoHipercubo=vecinosHipercubo(rank);
    for (i=0;i<D;i++){
        MPI_Bsend(&numero,1,MPI_DOUBLE,vecinoHipercubo[i],0,MPI_COMM_WORLD);
        MPI_Recv(&numeroNuevo,1,MPI_DOUBLE,vecinoHipercubo[i],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        if(numeroNuevo>numero){
            numero=numeroNuevo;
        }
    }
    printf("Numero maximo %lf\n",numero);

    }
```

Al igual que en el caso del toroide, el proceso de Rank 0 se encargará de comprobar el tamaño y de decir al resto de nodos si deben continuar o no.

Si el proceso 0 tiene que continuar este leerá los datos del fichero datos.dat y los distribuirá entre el resto de los nodos, este proceso a su vez enviara y recibirá los números de sus vecinos y los comparara con los suyos para poder obtener el numero mayor.

```
93      }else{
94          MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
95          if(continuar==0){
96
97          MPI_Recv(&numero,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
98
99          vecinoHipercubo=vecinosHipercubo(rank);
100         for (i=0;i<D;i++){
101             MPI_Bsend(&numero,1,MPI_DOUBLE,vecinoHipercubo[i],0,MPI_COMM_WORLD);
102             MPI_Recv(&numeroNuevo,1,MPI_DOUBLE,vecinoHipercubo[i],MPI_ANY_TAG,MPI_COMM_WORLD,&status);
103             if(numeroNuevo>numero){
104                 numero=numeroNuevo;
105             }
106         }
107
108         }
109     }
110
111     MPI_Finalize();
112
113     return EXIT_SUCCESS;
114 }
```

En el resto de los nodos recibirán su numero y realizaran la tarea de envió y recepción para quedarse con el máximo.

```
116 int* vecinosHipercubo(int nodo){
117     static int vecinos [D];
118     int i=0;
119     int desplazamiento=1;
120
121     for (i=0;i<D;i++){
122         vecinos[i]=nodo^desplazamiento;
123         desplazamiento=desplazamiento<<1;
124     }
125     return vecinos;
126 }
```

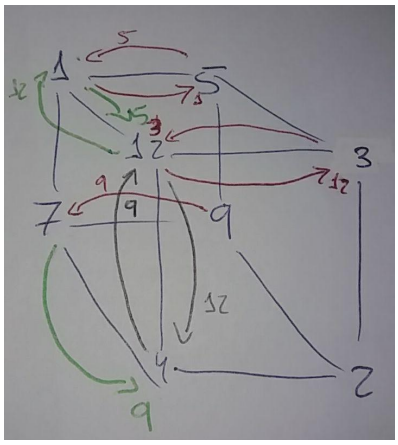
Para obtener los vecinos de un nodo dado se hará como he mencionado con anterioridad, dado un nodo dado, se ira haciendo la función xor con ese nodo y un numero que cambia solo un bit de posición, para hacer que ese numero cambie solo un bit de posición se hará mediante el uso de un desplazamiento, como se ve en la línea 123.

```
128 int contadorDatos(){
129     int datos=0;
130
131     char* char_fichero;
132     char linea[80];
133     char *token;
134     const char separador[2] = ",";
135     FILE *archivo;
136
137     if((archivo=fopen("./dirs/datos.dat","r"))==NULL){
138         printf("Error al abrir el fichero\n");
139         exit(EXIT_FAILURE);
140     }
141
142     char_fichero=fgets(linea,MAX_DATOS*sizeof(char),archivo);
143
144     token = strtok(linea, separador);
145
146     while( token != NULL ) {
147         datos++;
148         token = strtok(NULL, separador);
149     }
150
151     fclose(archivo);
152
153     return datos;
154 }
```

La comprobación del numero de datos introducidos en el fichero se realizará de la misma manera que en el caso del Toroide.

Explicación del flujo de datos:

El flujo de datos inicial de esta red hipercubo es igual que en el caso de la red toroide, el nodo 0 comenzara mandando un mensaje Bcast a todos los nodos indicándoles si estos deben seguir con la ejecución del programa o no, en caso de que estos deban seguir con la ejecución, el nodo 0 leerá el fichero datos.dat y distribuirá mediante el uso de un mensaje Bsend los números a cada uno de los nodos. Una vez los nodos reciban su numero estos comenzaran con el flujo de datos de búsqueda de el valor mayor que se muestra en la imagen.



Cada nodo distribuirá entre sus vecinos un total de D veces su numero actual, recibiendo a su vez el de sus vecinos, estos compararan dicho numero recibido con el suyo y se quedaran con el mayor, de esta manera todos los nodos obtendrán el valor mayor de la red. En este ejemplo que vemos en la imagen podemos observar como los nodos distribuyen sus números entre los vecinos de su dimensión iesima recibiendo el numero de estos a su vez, quedándose de esta manera con el mayor de estos.

Instrucciones de como compilar y ejecutar:

Para realizar la compilación se hará mediante el makefile, mediante el uso del comando *make all*, una vez este compilado podremos hacer dos pruebas básicas, colocando 8 datos en el archivo datos.dat podremos ejecutar el comando *testHipercubo.*

Conclusiones:

Como conclusiones podemos destacar el alto grado de eficiencia, paralización y control que podemos tener sobre la red gracias a MPI. También podemos comprobar que podemos trabajar a nivel de bit para identificar los nodos de una red y sus vecinos.

MPI nos ofrece una gran cantidad de herramientas para la paralelizarían de tareas y la comunicación entre estas, permitiéndonos el envió de mensajes nodo a nodo, envió de mensajes a múltiples nodos, o la recolección de datos de múltiples nodos.