Red Hipercubo

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 O (logaritmo base 2(n)) Con n número de elementos de la red.

2. Planteamiento de la solución:

Para la solución la rutina que he seguido ha sido que cada proceso envía su número a uno de sus vecinos y seguidamente espera a que le llegue el valor de ese mismo vecino. Una vez que el proceso tiene ambos valores los compara y se queda con el número más grande.

Este proceso se realiza tantas veces como dimensiones tenga el Hipercubo del problema.

3. Diseño del programa:

En el programa lo primero que hago es comprobar si los procesos que se lanzan desde el comando de ejecución son suficientes para realizar los cálculos, en caso de que no sea así el programa acaba su ejecución instantáneamente. También cuento los números que hay en el fichero “datos.dat” y si no hay suficientes valores, el programa no se ejecuta.

Si lanzamos los procesos suficientes para realizar los cálculos, en primer lugar, el proceso 0 (manejador) accede al fichero “datos.dat” y lee su contenido hasta tener los valores que necesita. Acto seguido él se queda con el primer valor y los demás los asigna a cada proceso.

Cuando cada proceso tiene su dato, obtiene sus vecinos (que son tantos como dimensiones tenga el hipercubo) para poder realizar las comparaciones. Una vez que

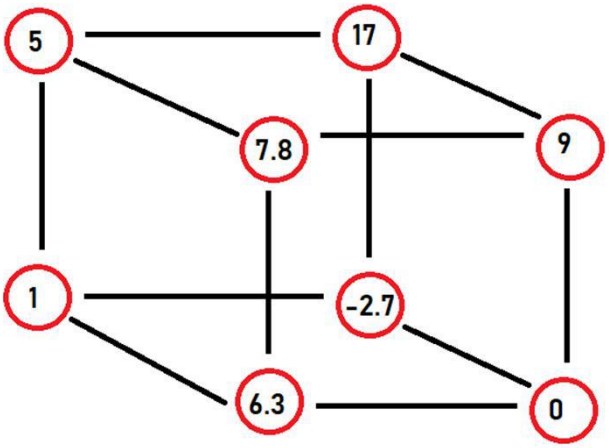
obtiene sus vecinos comienza a ejecutar el algoritmo para obtener el mayor número entre el suyo y el de todos sus vecinos.

Una vez han realizado este algoritmo, todos los procesos que lo han completado acaban su ejecución, salvo el proceso 0 que es el que se encarga de mostrar que valor es el más grande de la red antes de finalizar.

Además, si se lanzan más procesos de los que se necesitan, estos procesos no realizan estos cálculos y finalizan sin realizar ninguna acción.

4. Explicación del flujo de datos:

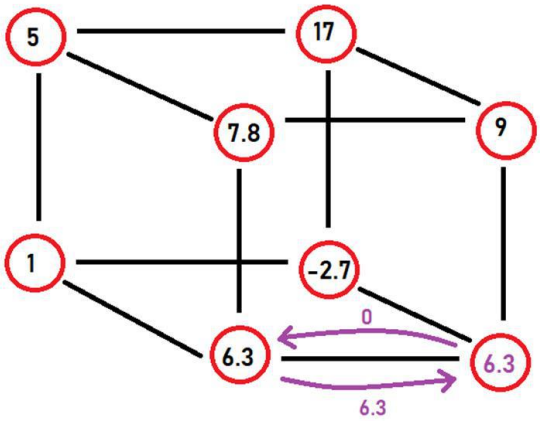
Por ejemplo, supongamos que tenemos una red hipercubo como la de la imagen de 3 dimensiones.



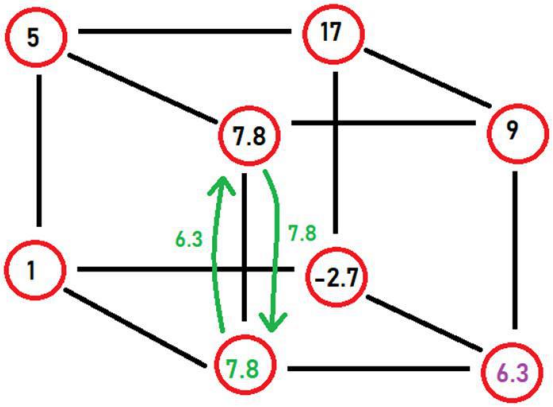
En primer lugar, un proceso (en este caso vamos a usar el proceso con valor 6.3 como ejemplo) envía su valor a su vecino en la primera dimensión (la dimensión en el eje x).

Una vez recibe el valor de su vecino de la dimensión 1, compara ambos valores y se queda con el valor más grande.

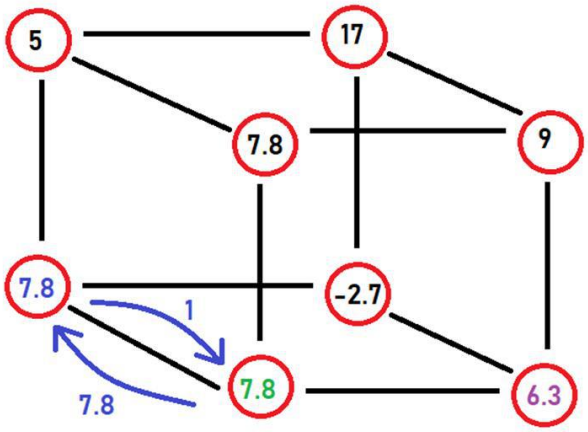
Para el envío se utiliza la instrucción `MPI_Bsend()` para disponer de búferes para evitar un problema de interbloqueos. Para la recepción de valores se usa la instrucción `MPI_Recv()`.



Después de realizar las operaciones con la dimensión 1, se hace lo mismo con la dimensión 2 (dimensión en el eje y).



Finalmente se realiza la misma operación con la dimensión 3 (dimensión en el eje z).



5. Fuentes del programa:

El código del programa es el siguiente:

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

#define MAX(a, b) (((a) > (b)) ? (a) : (b))

#define D 3
#define NUM_NODOS (int)pow(2,D)

FILE *fp;

int size, rank;
MPI_Status status;

//Cuenta la cantidad de números que hay en el documento 'datos.dat'
int cantidadNumeros(){
   int n = 0;
   float num;
   fp = fopen("datos.dat", "r");
   while((fscanf(fp, "%f,", &num)) != EOF){
        n++;
   }
   return n;
}

//Obtiene los números del documento
void obtenerNumeros(float numeros[]){
   int i=0;
   float num;
   fp = fopen("datos.dat", "r");

   while((fscanf(fp, "%f,", &num)) != EOF){
      if(i<NUM_NODOS){
         numeros[i] = num;
            i++;
      }else
         break;
   }
      fclose(fp);
}
//Envia los valores a cada proceso
void enviarDatos(float numeros[]){
   int i;
   for(i=0;i<NUM_NODOS-1;i++){
      MPI_Bsend(&numeros[i+1], 1, MPI_FLOAT, i+1, i, MPI_COMM_WORLD);
   }
}
//Obtiene los vecinos de cada nodo
void vecinosHipercubo(int vecinos[]){
   int i;
   for(i=0;i<D;i++){
      vecinos[i] = rank ^ (int)pow(2,i);
   }
}

//Calcula el mayor número de toda la red
float calcularMayor(float mi_numero, int vecinos[]){
   int i;
   float su_numero;
```

```
   //Envio por dimensiones
      for(i=1;i<=D;i++){
         MPI_Bsend(&mi_numero, 1, MPI_FLOAT, vecinos[i-1], 0,
MPI_COMM_WORLD);
         MPI_Recv(&su_numero, 1, MPI_FLOAT, vecinos[i-1], 0,
MPI_COMM_WORLD, &status);
            mi_numero = MAX(mi_numero, su_numero);
      }
      return mi_numero;
}
int main(int argc, char *argv[])
{
   float numeros[NUM_NODOS];
   float mi_numero;
   int vecinos[D];
   int continua = 0;
   int numeros_documento = 0;

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

   numeros_documento = cantidadNumeros();

   if(size<NUM_NODOS){ //Comprueba que haya los suficientes procesos
para ejecutar
      continua=1;
   }else if(numeros_documento < NUM_NODOS){ //Comprueba que haya los
suficientes valores en 'datos.dat' para ejecutar
      continua=2;
   }

   if(rank==0){
      if(continua==1){ //Comprueba que haya los suficientes procesos
para ejecutar
         fprintf(stderr, "ERROR, no se han lanzado los suficientes
procesos, necesito al menos %d\n", NUM_NODOS);
      }
      else if (continua==2){ //Comprueba que haya los suficientes
valores en 'datos.dat' para ejecutar
         fprintf(stderr, "ERROR, en el documento 'datos.dat' no hay
suficientes valores, debe haber al menos %d\n", NUM_NODOS);
      } else {
          obtenerNumeros(&numeros);
          enviarDatos(&numeros);

          mi_numero=numeros[0]; // Se queda con el primer dato porque
es necesario que intervenga en la red Hipercubo
          vecinosHipercubo(&vecinos);
          mi_numero = calcularMayor(mi_numero, &vecinos);
          printf("El máximo de la red HIPERCUBO es
%2.2f\n",mi_numero);
      }
}
      else {
         if(continua==0 && rank < NUM_NODOS){ //Comprueba que pueda
ejecutar y que el proceso pertenezca a la red
            MPI_Recv(&mi_numero, 1, MPI_FLOAT, 0, MPI_ANY_TAG,
MPI_COMM_WORLD,&status);
            vecinosHipercubo(&vecinos);
            mi_numero = calcularMayor(mi_numero, &vecinos);
```

```
            }
          }
        MPI_Finalize();
        return 0;
}
```

6. Instrucciones de como compilar y ejecutar:

Para ejecutar el programa deberemos utilizar el comando "`mpicc RedHipercubo.c -o RedHipercubo -lm`" para hacer uso de las primitivas MPI.

Para ejecutar el programa usaremos el comando "`mpirun -n NUM ./RedHipercubo`" donde *NUM* es el número de procesos que queremos que nuestro programa ejecute.

Para hacer uso del Makefile, basta con escribir "`make compileHipercubo`" para realizar la compilación y con "`make runHipercubo`" ejecutamos el programa usando para una red 3 dimensiones y con 8 procesos para ello.

7. Conclusiones:

La principal conclusión a la hora de realizar un programa con MPI este nos permite realizar varios cálculos en paralelo por lo que nuestro programa tiene una complejidad menor y además ganamos en rendimiento y rapidez.