**MPI PRACTICA 1**

**RED HIPERCUBO:**

# 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 $O(log_2 n)$, Con n número de elementos de la red.

# 2. Planteamiento de la solución.

Para solucionar el problema se crearán 2^D procesos, por ejemplo, en un hipercubo de 3 dimensiones, se crearán 8 procesos. En número de dimensiones del hipercubo se define como:

```
#define D  3
```
(Dentro del fuente HiperCubo.c)

Para trabajar con hipercubos de otra dimensión se tendrá que cambiar esta línea del fichero fuente y compilar. Si se desea ejecutar con un gran número de nodos se tendrá que cambiar también `#define MAX_ITEMS 1024`

Podemos distinguir dos roles en el programa:

- **Rank 0**: El proceso con Rank 0 se encargará de leer el fichero "datos.dat". Leerá los datos de fichero y enviara uno a cada proceso (Rank 0 incluido). El proceso de Rank 0 también controlará si se ha producido algún error y si los procesos deben o no continuar con la ejecución.
- **Todos los nodos**: todos los nodos recibirán del Rank 0 si se debe o no continuar con la ejecución. Si la ejecución continua recibirán un número real para empezar con el algoritmo para calcular el número mayor.

El proceso con Rank 0 se encargará de leer el fichero "datos.dat". Los datos que lea de este fichero serán almacenados en un array para luego enviarlos. Tras leer este fichero decidirá si continua o no con la ejecución. Si los números leídos no son 2^D o no se han creado 2^D procesos, el Rank 0 mandara un mensaje a todos los procesos del comunicador global (con la operación Bcast) para que no continúen la ejecución. Si la lectura es correcta mandara un mensaje a todos los procesos (incluyéndose) indicando que pueden continuar con la ejecución.

Tras haber mandado que se ha de continuar con la ejecución el proceso de Rank 0 mandará a todos los procesos (incluyéndose) uno de los números reales leídos en el fichero. Cuando todos reciban su número se comenzará con la ejecución del algoritmo para calcular el máximo entre ellos.

El algoritmo consiste en que cada nodo envié su número a su vecino en la dimensión Di y reciba de esa misma dimensión. Después, su número será el máximo entre el suyo y el que ha recibido. Y mandara su nuevo número en la siguiente dimensión.

```
Desde 1 a L:
        Enviar(Di,MiNumero);
        Recibir(Di,SuNumero);
        MiNumero = max(MiNumero,SuNumero);
```

# 3. Diseño de la solución.

- **Lectura de fichero (ejecutado por Rank 0).**

```
  double *datos;
  int cantidadNumeros=0,continuar=TRUE;

  datos = malloc(MAX_ITEMS*sizeof(double));
  obtenerDatos(datos,&continuar,&cantidadNumeros);
```

Se crea el array donde vamos a guardar los datos leídos del fichero "datos.dat" y llamamos a la función obtenerDatos().

```
void obtenerDatos(double* datos,int *continuar,int *cantidadNumeros){
    char *linea;
    char *token;

    FILE *file;
    /* Creamos la linea que vamos a leer del fichero */
    linea = malloc(MAX_ITEMS*sizeof(char));

    if((file = fopen(FILENAME,"r"))==NULL){
        fprintf(stderr,"Error al abrir el archivo %s\n",FILENAME);
        continuar = FALSE;

    }
    else{
     /*Hacemos uso de la funcion strtok para leer la cadena separada por
comas*/
        fgets(linea,MAX_ITEMS*sizeof(char),file);
        token = strtok(linea,",");

        while(token!=NULL){
            datos[(*cantidadNumeros)++]=atof(token);
            token = strtok(NULL,",");
        }

    }
    fclose(file);
    free(linea);

}
```

De esta forma leeremos los valores del fichero y los guardaremos en el array de datos que es pasado por referencia, al igual que continuar y cantidadNumeros. Tras la ejecución de esta función tendremos los datos, el número de datos leídos y si se ha producido un error en la lectura

- **Tratamiento de errores (Rank 0)**

Tras la lectura del fichero comprobamos que los números leídos no sean igual a 2^D y también comprobamos que el número de nodos lanzados no sea igual a 2^D. Si alguna de estas condiciones se cumple el programa no podrá ejecutarse.

```
/*Si el numero de datos no es igual a 2 elevado al numero de dimensiones*/
     if(cantidadNumeros!=((int)pow(2,D))){
         fprintf(stderr,"Error con el número de datos\n");
         continuar = FALSE;
      }
/* Si la cantidad de nodos no es la misma que 2 elevado al número de dimensiones*/
     if(size!=((int)pow(2,D))){
          fprintf(stderr,"Error con el número de nodos\n");
          continuar = FALSE;
      }
```

Para notificar a los demás procesos si ha de seguir con la ejecución o no, se ejecuta la orden MPI_Bcast().

```
     /* Multidifusión para saber si continuar con la ejecución o no*/
        MPI_Bcast(&continuar,1,MPI_INT,0,MPI_COMM_WORLD);
```

- **Envío y recepción de datos.**

```
/* Si se continua con la ejecución mandamos el dato a cada proceso*/
        if(continuar){
            for(i=0;i<size;i++){
                buffNumero = datos[i];
                MPI_Bsend(&buffNumero,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
            }
            /* Liberamos el array, puesto que no lo vamos a volver a usar*/
            free(datos);
        }
```

Los procesos recibirán de la siguiente forma el dato:

```
if(continuar){
   MPI_Recv(&buffNumero,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
   calcularMaximo(rank,buffNumero);
 }
```

- **Obtención de vecinos (todos los procesos)**

Función usada para conocer los vecinos de un nodo en todas las dimensiones.

```
int* vecinosHipercubo(int node){
    int i;
    int aux;
    static int vecinos[D];

    for(i=0;i<D;i++){
        aux = 1 << (i);
        vecinos[i] = node ^ aux;
    }

    return vecinos;
}
```

Para realizarlo simplemente se realiza un bucle de "D" iteraciones y se hará la operación XOR al rango del nodo que quiere calcular los vecinos con el bit 1 desplazado "i" posiciones.
Este dato es guardado en un array, donde cada posición indica el vecino del nodo en esa posición.

- **Realización del algoritmo:**

Desde 1 a L:
Enviar(Vecino[Di],MiNumero);
Recibir(Vecino[Di],SuNumero);
MiNumero = max(MiNumero,SuNumero);

```
void calcularMaximo(int rank, double numero){
    int i;
    int *vecinos;
    MPI_Status status;
    vecinos = vecinosHipercubo(rank);
    double suNumero;

    for(i=0;i<D;i++){
        MPI_Bsend(&numero,1,MPI_DOUBLE,vecinos[i],1,MPI_COMM_WORLD);
        MPI_Recv(&suNumero,1,MPI_DOUBLE,vecinos[i],1,MPI_COMM_WORLD,&status);
        numero = maximo(numero,suNumero);
    }

    if(rank==0){
        printf("El número máximo es: %.2f\n",numero);
    }
}
```

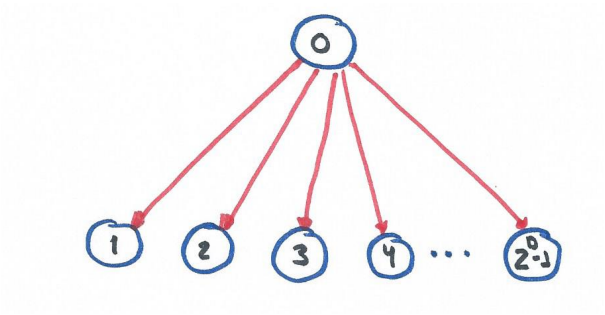
Cada nodo enviara y recibirá une el dato del vecino de su dimensión, después cambiara su número por el máximo entre su número y el número del vecino. Este proceso se hará "D" iteraciones. El algoritmo tendrá una complejidad $O(\log_2(n))$

# 4. Explicación del flujo de datos de la red.

## 1) Envío de Continuar.

Tras la lectura de los datos y comprobación de errores se mandará a los demás si se ha de continuar o no mediante:

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

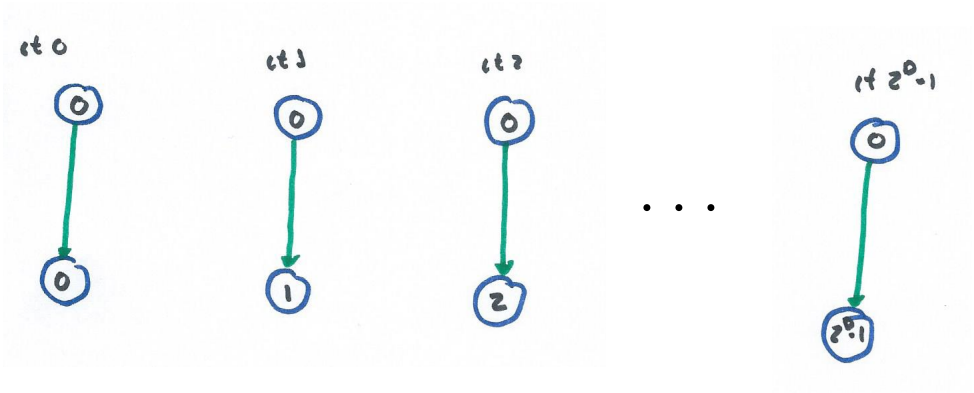



Como es una operación colectiva todos los procesos deben ejecutar el bcast antes de continuar.

## 2) Envío de datos

El envío de datos se realiza mediante un bucle que envía a cada proceso su dato correspondiente.

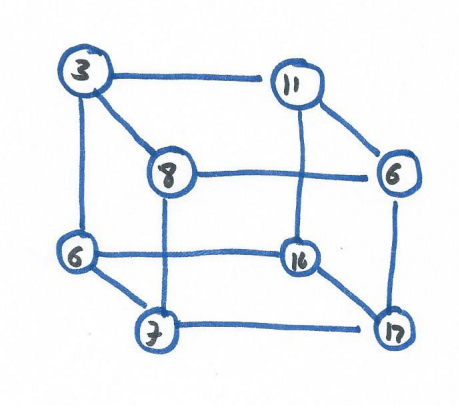
Send(Rank 0) → `MPI_Bsend(&buffNumero,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);`
Recv: `MPI_Recv(&buffNumero,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);`




3) Algoritmo

Para ver el flujo de datos de nuestro programa simularemos la ejecución para un hipercubo de 3 dimensiones con los siguientes datos.

Datos = [7,12,6,10,8,6,3,11]

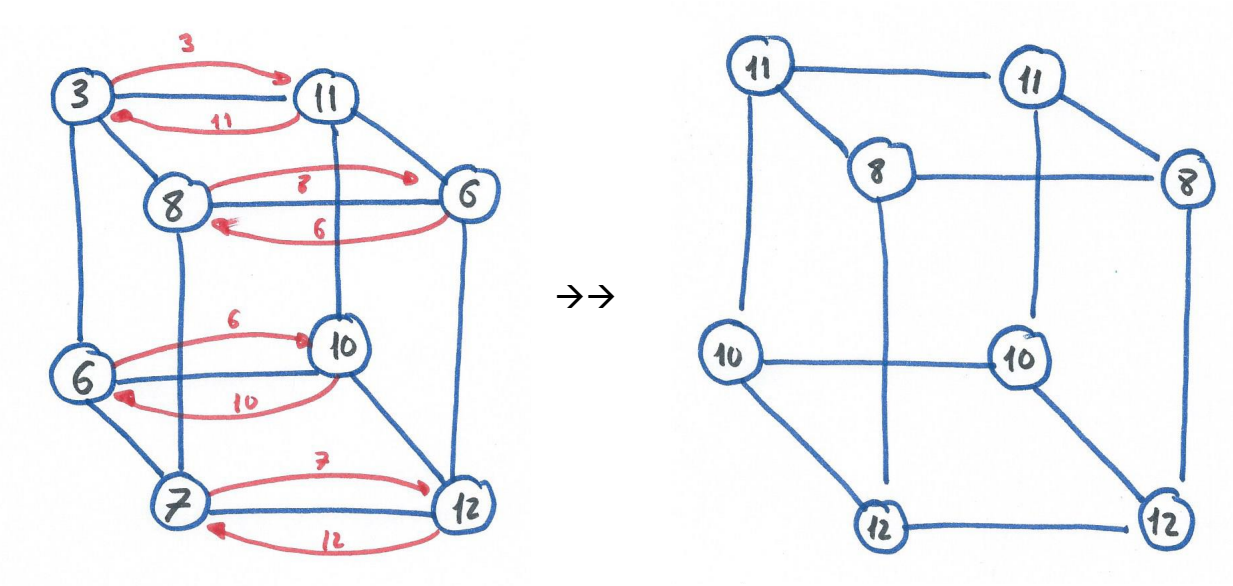


El algoritmo consta de un bucle:

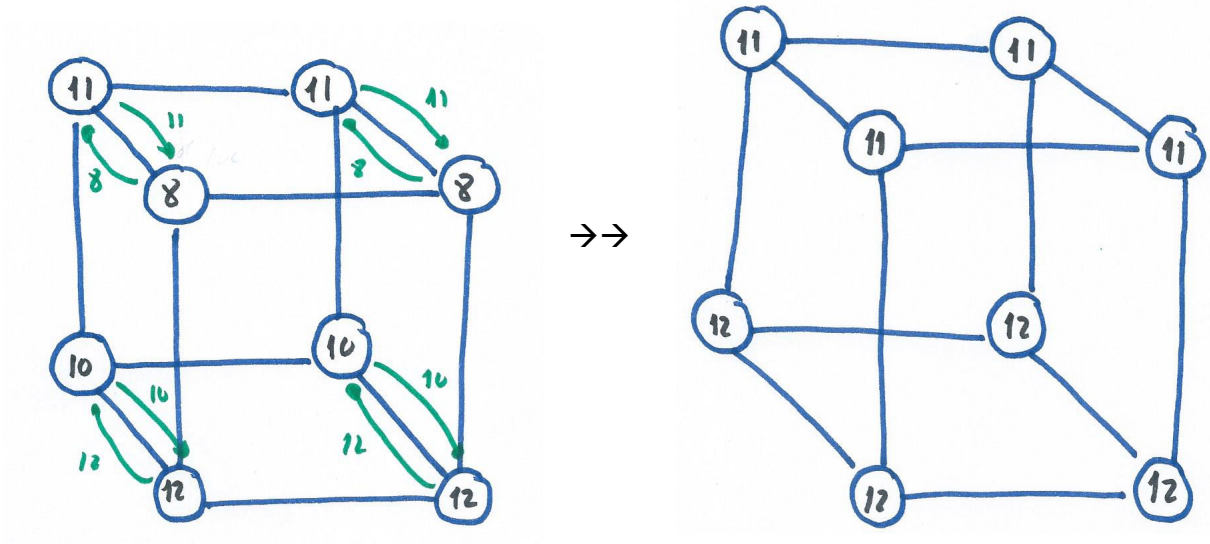
```
for(i=0;i<D;i++){
        MPI_Bsend(&numero,1,MPI_DOUBLE,vecinos[i],1,MPI_COMM_WORLD);
        MPI_Recv(&suNumero,1,MPI_DOUBLE,vecinos[i],1,MPI_COMM_WORLD,&status);
        numero = maximo(numero,suNumero);
    }
```

Todos envían su dato y reciben el dato del vecino que están en la dimensión $D_i$ y a continuación su dato pasa a ser el máximo entre esos dos valores.

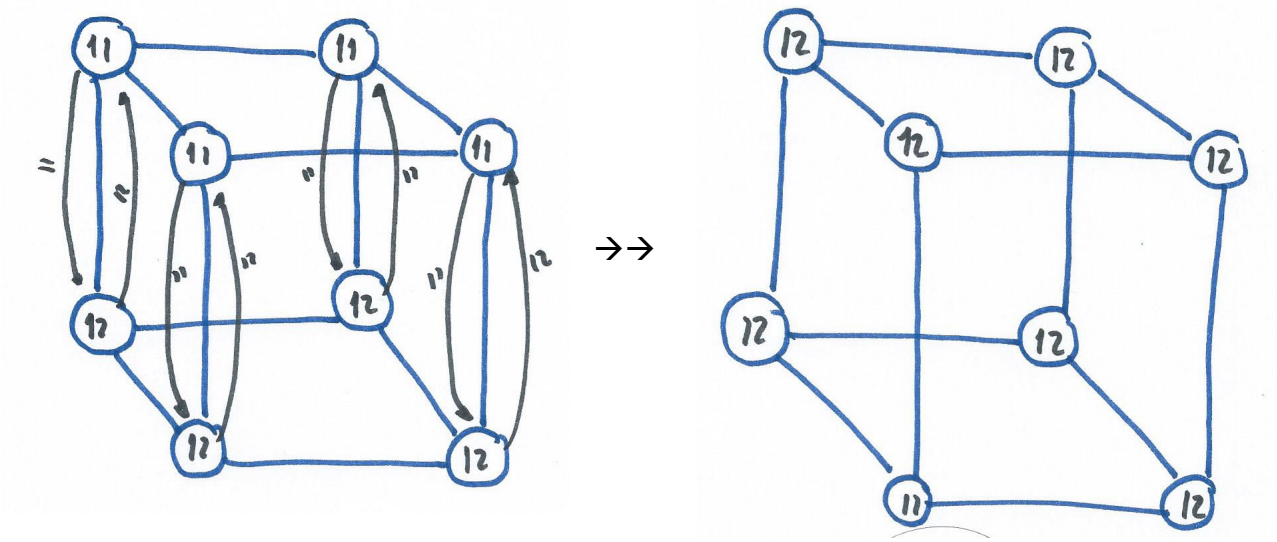
Primera iteración de nuestro ejemplo:



Segunda iteración de nuestro ejemplo:



Tercera iteración de nuestro ejemplo:



Y de esta forma ya tenemos el elemento mayor con una complejidad de $O(\log_2(n))$