MPI PRACTICA 1

RED TOROIDE:

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 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, 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 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 $O(\sqrt{n})$ Con n número de elementos de la red.

2. Planteamiento de la solución.

Para solucionar el problema se crearán LxL procesos, por ejemplo, en un toroide de lado 4, se crearán 16 procesos. El número de lado del toroide se define como:

```
#define L  4
```
(Dentro del fichero RedToroide.c)

Para trabajar con toroides de otro lado se tendrá que cambiar esta línea en el 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 nuestro programa:

- **Rank 0**: El proceso con Rank 0 se encargara 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 la instrucción de continuar o no con la ejecución. Si la ejecución continua recibirán un número real para empezar con el algoritmo para calcular el menor numero.

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 LxL o no se han creado LxL procesos, el Rank 0 mandara un mensaje a los demás procesos para que no continúen su ejecución y todos finalicen. Si la lectura es correcta mandará un mensaje a los demás procesos indicando que pueden continuar su ejecución.

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

El algoritmo consiste en que cada nodo envíe el vecino sur su número y reciba de él su número, cada nodo se quedara con el mínimo entre su número y el que ha recibido. Cuando termine de hacerlo en vertical (SUR-NORTE) lo hará en horizontal (ESTE-OESTE).

```
Desde 1 a L-1:
  enviar(SUR,MiNumero)
  recibir(NORTE,suNumero)
  MiNumero = min(MiNumero,SuNumero)
Desde 1 a L-1:
   enviar(ESTE,MiNumero)
   recibir(OESTE,suNumero
   MiNumero = min(MiNumero,SuNumero)
```

3. Diseño de la solución.

- **Lectura de fichero: (ejecutado por rank0)**

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

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

Se creara el array donde vamos a guardar los datos leidos del fichero “datos.dat” y llamamos a la función obtenerDatos().

```
void obtenerDatos(double* datos,int *continuar,int *cantidadNumeros){
    char *linea;
    char *token;
    FILE *file;
    linea = malloc(MAX_ITEMS*sizeof(char));
    /* Si no se puede leer el fichero mostramos el error y no continuará */
    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){
            token = strtok(NULL,",");
        }
    }
    fclose(file);
    free(linea);
}
```

De esta forma leeremos los valores del fichero y los guardaremos en el array 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 o no un error en la lectura.

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

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

```
/* Si el número de datos no es igual al número de nodos no podrá ejecutarse */
        if(cantidadNumeros!=(L*L)){
            fprintf(stderr,"Error con el número de datos\n");
            continuar=FALSE;
        }
    /* Si la cantidad de nodos no es la misma que L*L no podrá ejecutarse*/
        if(size!=(L*L)){
            fprintf(stderr,"Error con el número de nodos\n");
            continuar=FALSE;
        }
```

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

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

- **Envío y recepción de datos.**

Si no hay ningún error el proceso 0 mandara a todos los demás procesos (incluyéndose) un numero:

```
if(continuar){
     for(i=0;i<(L*L);i++){
        buffNumero = datos[i];
         MPI_Bsend(&buffNumero,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
       }
   /* Liberamos el array de datos puesto que no lo vamos a usar ya*/
   free(datos);
 }
```

Los demás procesos lo recibirán de la siguiente forma:

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

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

Esta función está diseñada es para conocer la identidad de los vecinos con los que tenemos conexión pasando como argumentos nuestro Rank.

```
#define NORTE 0
#define SUR   1
#define ESTE  2
#define OESTE 3

int* vecinosToroide(int node){
    int fil,col;
    static int vecinos[4];

    fil = node / L;
    col = node % L;

    /* Primero calculamos los vecinos NORTE y SUR*/

    switch(fil){
        /* Caso de fila inferior*/
        case 0:
            vecinos[NORTE] = node + L;
            vecinos[SUR] = (L-1)*L + node;
            break;
        /* Caso de fila superior */
        case L-1:
            vecinos[NORTE] = node % L;
            vecinos[SUR] = node - L;
            break;
        /* Caso de fila central */
        default:
            vecinos[NORTE] = node + L;
            vecinos[SUR] = node - L;
            break;
    }


   /* Calculamos los vecinos ESTE y OESTE */

    switch(col){
        /* Caso de columna izquierda*/
        case 0:
            vecinos[OESTE] = (L * (fil+1))-1;
            vecinos[ESTE] = node + 1;
            break;
        /* Caso de columna derecha*/
        case L-1:
            vecinos[OESTE] = node -1;
            vecinos[ESTE] = fil * L;
            break;
        /* Caso de columna central */
        default:
            vecinos[OESTE] = node -1;
            vecinos[ESTE] = node + 1;
            break;
    }

    return vecinos;
}
```

Esta función devolverá un array con 4 posiciones, cada posición representará un vecino.

- **Realización del algoritmo:**

Desde 1 a L-1:
  enviar(SUR,MiNumero)
  recibir(NORTE,suNumero)
  MiNumero = min(MiNumero,SuNumero)
Desde 1 a L-1:
  enviar(ESTE,MiNumero)
  recibir(OESTE,suNumero
  MiNumero = min(MiNumero,SuNumero)

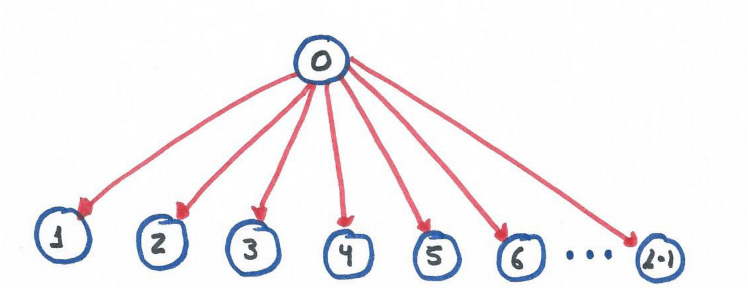
```
void calcularMinimo(int rank, double numero){
    int i;
    int *vecinos;
    MPI_Status status;
    vecinos = vecinosToroide(rank);
    double suNumero;

    for(i=1;i<L;i++){
        MPI_Bsend(&numero,1,MPI_DOUBLE,vecinos[SUR],1,MPI_COMM_WORLD);
        MPI_Recv(&suNumero,1,MPI_DOUBLE,vecinos[NORTE],1,
              MPI_COMM_WORLD,&status);
        numero = minimo(numero,suNumero);
    }

    for(i=1;i<L;i++){
        MPI_Send(&numero,1,MPI_DOUBLE,vecinos[ESTE],1,MPI_COMM_WORLD);
        MPI_Recv(&suNumero,1,MPI_DOUBLE,vecinos[OESTE],1,
             MPI_COMM_WORLD,&status);
        numero = minimo(numero,suNumero);
    }



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

}
```

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);
```

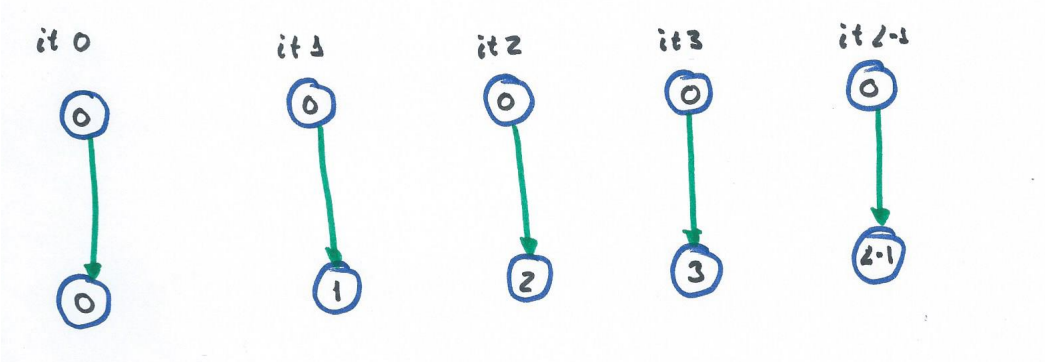


Como es una operación colectiva todos los procesos deben ejecutar el bcast antes de continuar.

2) Envío de datos

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

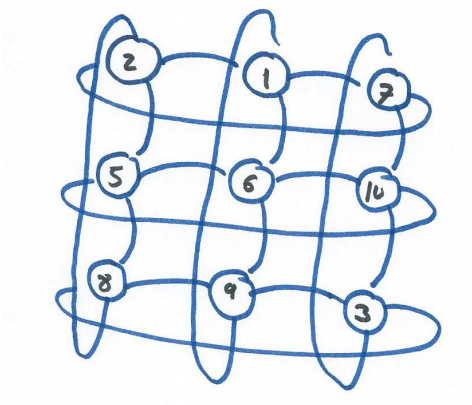
Send(Rank 0) → `MPI_Send(&buffNumero,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);`
Recv: `MPI_Recv(&buffNumero,1,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);`



3) Algoritmo

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

Datos = [8,9,3,5,6,10,2,1,7]



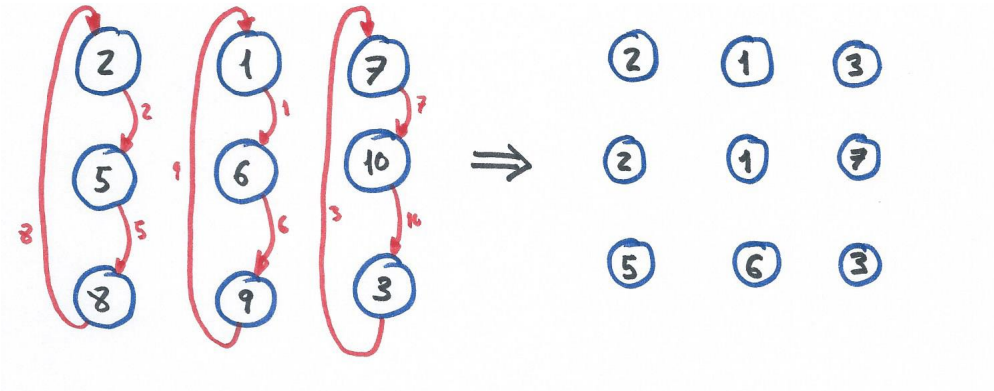
El algoritmo consta de 2 bucles.

El primer bucle:

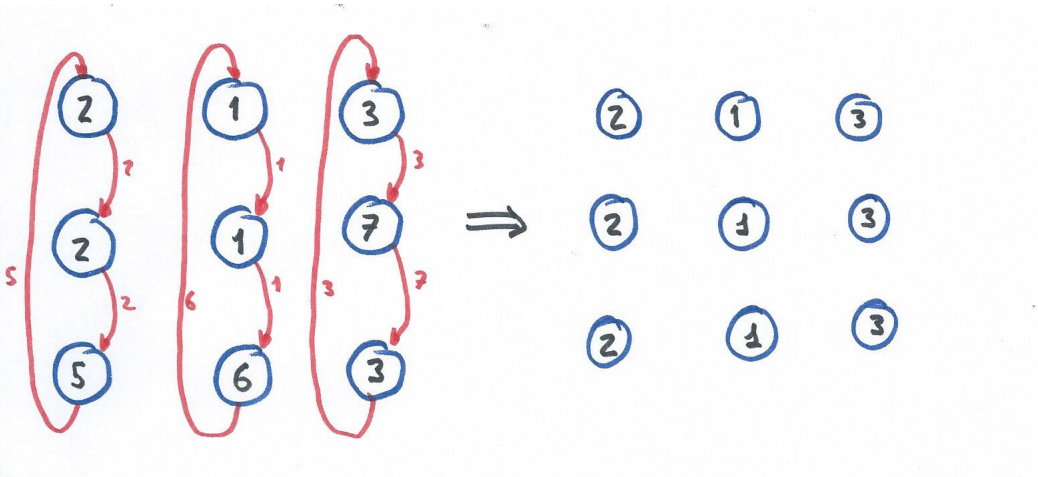
```
for(i=1;i<L;i++){
        MPI_Bsend(&numero,1,MPI_DOUBLE,vecinos[SUR],1,MPI_COMM_WORLD);
        MPI_Recv(&suNumero,1,MPI_DOUBLE,vecinos[NORTE],1,
              MPI_COMM_WORLD,&status);
        numero = minimo(numero,suNumero);
    }
```

Todos envían al vecino sur su dato y reciben del vecino norte y a continuación su dato pasara a ser el mínimoxxx entre el dato enviado y el recibido

Primera iteración de nuestro ejemplo:



Segunda intracion de nuestro ejemplo:

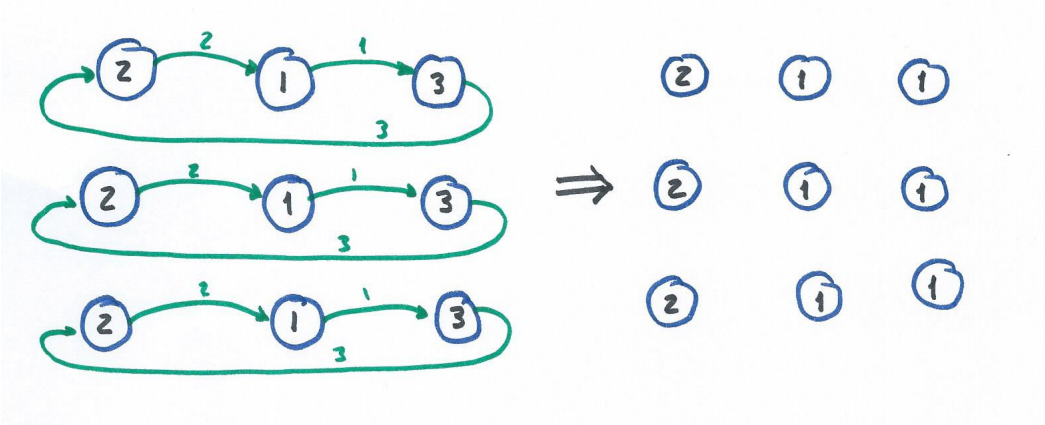


Segundo bucle:

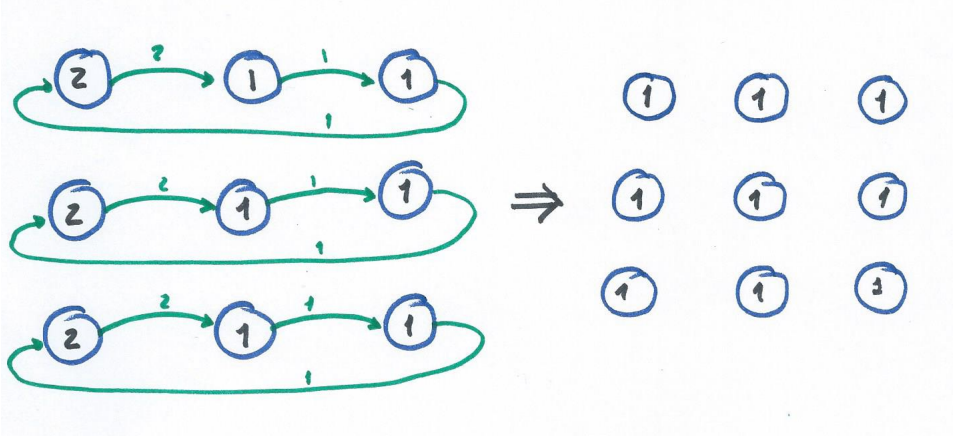
```
for(i=1;i<L;i++){
        MPI_Send(&numero,1,MPI_DOUBLE,vecinos[ESTE],1,MPI_COMM_WORLD);
        MPI_Recv(&suNumero,1,MPI_DOUBLE,vecinos[OESTE],1,
             MPI_COMM_WORLD,&status);
        numero = minimo(numero,suNumero);
    }
```

Todos envían al vecino este su dato y reciben del vecino oeste y a continuación su dato pasara a ser el mínimo entre el dato enviado y el recibido

Primera iteración de nuestro ejemplo:



Segunda iteración de nuestro ejemplo:



Y tras la ejecución de este segundo bucle ya tenemos el elemento menor con una complejidad O($\sqrt{n}$)