VILLAR ASENSIO JAVIER PRÁCTICA 1

TOROIDE

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 destribuirá a cada uno de los nodos de un toroide de lado L,los L x L numeros 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.

PLANTEAMIENTO

Para solucionar el problema tenemos que tener claros los conceptos básicos de un toroide; este va a estar formado por filas y columnas ya que es una matriz cuadrada de nodos, en este, cada nodo tiene que averiguar los valores que tengan los nodos que corresponden a su fila y columna. Esto se va a conseguir iterando en su fila y columna y enviando y recibiendo el número.

DISEÑO

Lo primero de todo es la definición de unos valores que van a ser fijos en el programa:

- La dimensión que está puesta de 8 porque el programa está diseñado para 64 nodos, para cambiar esto solo habría que cambiar este número en el programa.
- Los nodos del toroide que es el cuadrado de la dimensión.
- Las cuatro direcciones (arriba, abajo, derecha e izquierda) con las que se va a comunicar cada nodo.
- La fórmula lógica que se utiliza para calcular el mínimo.
- El nombre del archivo donde tenemos los datos (“datos.dat”)

```
#define DIMENSION 8
#define NODOS_TOROIDE DIMENSION*DIMENSION

#define ARRIBA 0
#define ABAJO 1
#define DERECHA 2
#define IZQUIERDA 3

#define MIN(x, y) (((x) < (y)) ? (x) : (y))

#define ARCHIVO "datos.dat"
```

Lo primero que necesitamos es leer los datos del fichero y almacenarlos en una variable, en el método “coger_numeros” es donde nos encargamos de esto. Abrimos el descriptor de archivo

comprobando que no dé error, en ese caso se acabaría el programa, y creamos un puntero donde vamos a guardar la lectura, ya teniendo el fichero abierto copiamos el contenido en este puntero para poder irlo separando por comas e ir guardando los números en el puntero que va a usar el programa principal.

```
void coger_numeros(double *datos){
    FILE *arhivo;
    char *copia = malloc(1500 * sizeof(char));
    char *numero;
    int *posicion = 0;

    if ((arhivo = fopen(ARCHIVO, "r")) == 0){
        perror("Error abriendo archivo");
        exit(EXIT_FAILURE);
    }else{
        fscanf(arhivo, "%s", copia);

        datos[(*posicion)++] = atof(strtok(copia,","));

        while((numero = strtok(NULL, ",")) != NULL)
            datos[(*posicion)++] = atof(numero);
    }

    fclose(arhivo);
    free(copia);
}
```

Los números tienen que ser enviados a cada uno de los nodos puesto que con un simple bucle en el método “enviar_numeros” enviamos a cada nodo del comunicador uno de los datos que tenemos guardados.

```
void enviar_numeros(double *datos){
    for(int i=0; i < NODOS_TOROIDE; i++)
        MPI_Send(&datos[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
}
```

Para averiguar los vecinos con los que se va a comunicar cada nodo tenemos el método “coger_vecinos” con el que guardaremos en un array el rango que corresponda a los nodos de las cuatro direcciones con las que se comunica.

Para ello, primero averiguamos en qué fila va a estar cada nodo según su rango, dividiendo su rango por la dimensión, el cociente que da, va a corresponder con la fila en la que se va a colocar en el toroide, pudiendo ser cero. La columna va a ser el resto de esta misma división. De esta manera los colocaremos en una matriz cuadrada. Una vez cada nodo ya sabe su posición ya puede hablar con sus nodos adyacentes, lo único que hay que tener en cuenta es cuando se dé el caso de los nodos que son o bien el fin, o bien el comienzo de una fila o columna cuando se tendrá que comunicar con el del extremo contrario.

```c
void coger_vecinos(int vecinos[], int rango){
    int columna = rango % DIMENSION;
    int fila = rango / DIMENSION;

    switch(columna){
        case 0:
            vecinos[IZQUIERDA] = (DIMENSION * (fila+1))-1;
            vecinos[DERECHA] = rango + 1;
            break;
        case DIMENSION-1:
            vecinos[IZQUIERDA] = rango -1;
            vecinos[DERECHA] = fila * DIMENSION;
            break;
        default:
            vecinos[IZQUIERDA] = rango -1;
            vecinos[DERECHA] = rango + 1;
            break;
    }

    switch (fila){
        case 0:
            vecinos[ABAJO] = rango + DIMENSION;
            vecinos[ARRIBA] = (DIMENSION-1) * DIMENSION + rango;
            break;
        case DIMENSION-1:
            vecinos[ABAJO] = rango % DIMENSION;
            vecinos[ARRIBA] = rango - DIMENSION;
            break;
        default:
            vecinos[ABAJO] = rango + DIMENSION;
            vecinos[ARRIBA] = rango - DIMENSION;
            break;
    }
}
```

El algoritmo que se encarga de la búsqueda del elemento mínimo consiste en un bucle en el que cada iteración va a decirle al vecino que corresponda el valor que guarda en ese momento, y recibir lo que los vecinos que le correspondan les digan que guardan. En este caso el sentido de envío es abajo y a la derecha y la recepción de arriba y de la izquierda. Ya teniendo el número que reciba del vecino lo compara con el suyo para quedarse con el menor. De esta forma el valor más pequeño se propagará a todos los nodos con una cantidad de iteraciones igual a la dimensión del toroide, por lo que se conocerá de una manera muy rápida.

```c
void toroide(int rango, double mi_numero){
    int vecinos[4];
    double numero_recibido;

    coger_vecinos(vecinos, rango);

    for(int i=1; i < DIMENSION; i++){
        MPI_Send(&mi_numero, 1, MPI_DOUBLE, vecinos[ABAJO], 1, MPI_COMM_WORLD);
        MPI_Recv(&numero_recibido,1,MPI_DOUBLE,vecinos[ARRIBA],1,MPI_COMM_WORLD,NULL);
        mi_numero = MIN(mi_numero, numero_recibido);

        MPI_Send(&mi_numero, 1, MPI_DOUBLE, vecinos[DERECHA], 1, MPI_COMM_WORLD);
        MPI_Recv(&numero_recibido,1,MPI_DOUBLE,vecinos[IZQUIERDA],1,MPI_COMM_WORLD,NULL);
        mi_numero = MIN(mi_numero, numero_recibido);

    }

    if(rango == 0)
        printf("El valor mínimo en toroide es: %.2f\n",mi_numero);
}
```

Una vez ya sabemos el comportamiento de las llamadas entenderemos el comportamiento del programa. Lo primero alojamos en memoria la estructura donde vamos a guardar los datos que he llamado “datos” y declaramos las variables que vamos a usar; el rango y tamaño para el comunicador del MPI y el número que cada nodo va a guardar. Una vez inicializado MPI, el nodo raíz se encargará de la lectura y distribución de los números, como esto solo lo tiene que hacer uno de los nodos he utilizado un Broadcast de MPI para la sincronización de los nodos puesto que los métodos de difusión colectiva son bloqueantes. Ya sincronizados reciben su número y llaman al método con el que se va a realizar la búsqueda. Al terminar todo esto se finaliza MPI y se liberan recursos.

```
int main(int argc, char *argv[]){
    double *datos = malloc(NODOS_TOROIDE * sizeof(double));
    int rango, tam;
    double mi_numero;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rango);
    MPI_Comm_size(MPI_COMM_WORLD,&tam);

    if (rango == 0){
        coger_numeros(datos);
        enviar_numeros(datos);
    }

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

    MPI_Recv(&mi_numero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, NULL);

    toroide(rango, mi_numero);

    MPI_Finalize();
    free(datos);

    return EXIT_SUCCESS;
}
```

HIPERCUBO

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 rango 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 rango 0 mostrará en su salida estándar el valor obtenido.

PLANTEAMIENTO

Para solucionar el problema con un hipercubo tenemos que saber la manera con la que vamos a elegir el vecino con quién nos vamos a comunicar ya que el resto es igual que el caso anterior. Esta problemática se resuelve con la distancia del código Gray, en la que se estipula que la comunicación

se hará entre los nodos que tengan una distancia de 1, siendo el valor de esta distancia la cantidad de bits que difieren entre el número de nodo en binario.

DISEÑO

Como anteriormente, comenzamos con la definición de unos datos que nos van a servir a lo largo del programa y que son invariables:

- La dimensión del hipercubo.
- Los nodos del cubo, que es la potencia de 2 elevado a la dimensión del hipercubo.
- Las fórmulas lógicas necesarias para averiguar el máximo y el nodo con distancia 1 (XOR).
- El archivo donde se encuentran los datos.

```
#define DIMENSION 6
#define NODOS_CUBO (int)pow(2,DIMENSION)

#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define XOR(x, y) (x ^ y)

#define ARCHIVO "datos.dat"
```

La lectura y distribución de los números es igual que en el caso anterior así que no se volverá a explicar.

El algoritmo del hipercubo es muy sencillo, trata de un bucle con la misma cantidad de iteraciones como dimensiones tenga el hipercubo. En este bucle se averigua el vecino con el que se va a comunicar en cada iteración con la operación XOR, este vecino cambiará en cada iteración puesto que se utilizará el bit diferencia en la posición equivalente al número de iteración del bucle, ya conociendo al vecino se envían y reciben los números y se quedan cada uno con el mayor. Al final del bucle el número mayor ya se habrá propagado a todos los nodos y el raíz los mostrará.

```
void hipercubo(int rango, double mi_numero){
    double numero_recibido;
    int vecino;

    for(int i=0; i < DIMENSION; i++){
        vecino = XOR(rango, (int)pow(2,i));

        MPI_Send(&mi_numero, 1, MPI_DOUBLE, vecino, 1, MPI_COMM_WORLD);
        MPI_Recv(&numero_recibido, 1, MPI_DOUBLE, vecino, 1,MPI_COMM_WORLD, NULL);

        mi_numero = MAX(mi_numero, numero_recibido);
    }

    if(rango == 0)
        printf("El mayor número en hipercubo es: %.2f\n",mi_numero);
}
```

La ejecución del programa es igual que en el caso del toroide, el raíz coge y distribuye los números, se sincronizan y cada uno se encarga de hacer la comunicación y comparación de sus números para propagar el mayor.

```c
int main(int argc, char *argv[]){
    double *datos = malloc(NODOS_CUBO * sizeof(double));
    int rango, tam;
    double mi_numero;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rango);
    MPI_Comm_size(MPI_COMM_WORLD,&tam);

    if (rango == 0){
        coger_numeros(datos);
        enviar_numeros(datos);
    }

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

    MPI_Recv(&mi_numero, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, NULL);

    hipercubo(rango, mi_numero);

    MPI_Finalize();
    free(datos);

    return EXIT_SUCCESS;
}
```

REFERENCIAS

He utilizado el foro StackOverflow para ver cómo se podían hacer las operaciones lógicas:

https://stackoverflow.com/questions/3437404/min-and-max-in-c

Y otros lugares para otras dudas:

https://www.programiz.com/c-programming/bitwise-operators

https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Bcast

Más el material de la asignatura dado en clase.

CONCLUSIONES

Con estos sencillos programas está bien irte acostumbrando a trabajar con clusters ya que es un programa con una lógica sencilla que me ha resultado fácil adaptar a un programa MPI, además te das cuenta de que puedes encontrar soluciones sencillas y rápidas a soluciones muy grandes gracias a herramientas muy avanzadas como son este tipo de sistemas. Lo único que puede ser complicado de entender es la forma con la que vas a comunicar a los nodos para que sea de la forma más eficaz y eficiente.