Práctica 1: Red toroide y Red hipercubo

Francisco Jesús Día Pellejero

Diseño de Infraestrucura de red

18-04-22

Red Toroide

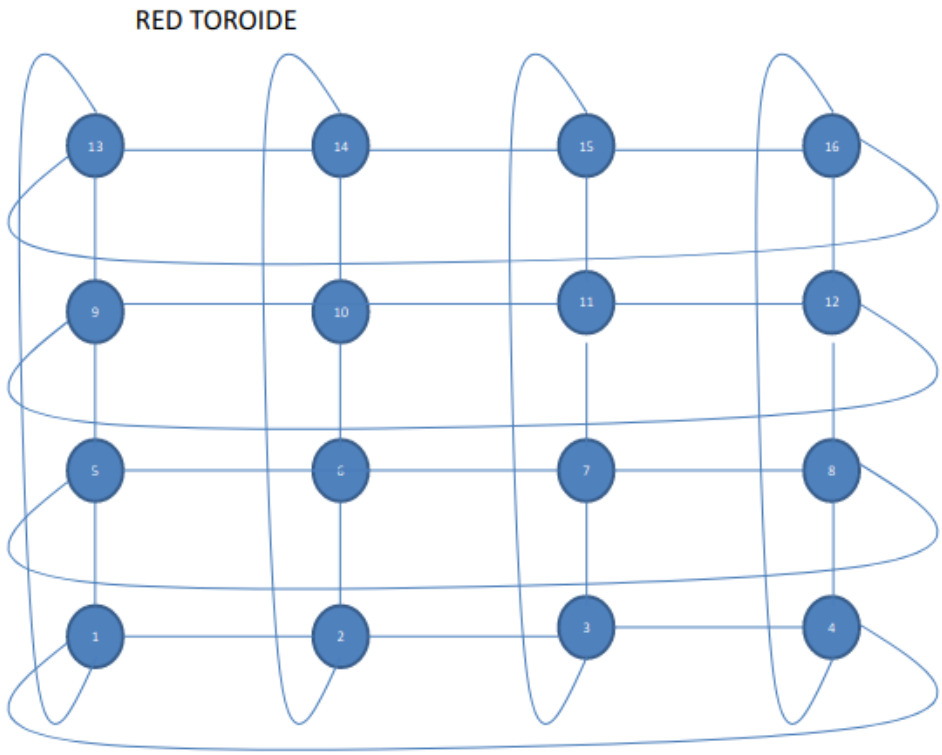
Enunciado

Se nos pide distribuir una lista de números a una red toroide y calcular el elemento menor de toda la red sin alcanzar una complejidad mayor que O(n^(1/2)), siendo n el número de elementos de toda la red.

Planteamiento de la solución

Será necesario el uso de computación distribuida y una topología de red toroide que nos permita la propagación del elemento menor a todos los nodos mediante el uso de MPI.

Un ejemplo de una red toroide de lado 4:



Primero comenzaremos con la propagación entre filas y después entre columnas tras haber distribuido todos los números entre todos los nodos. Cada nodo enviará su número al nodo que tiene al este y recibirá un nodo del nodo oeste, comparará este número y repetirá este proceso hasta L veces, siendo L el lado del toroide.

La idea es similar con la propagación entre columnas. Se enviará el número del nodo a su nodo que esté al sur y se recibirá un número por parte del nodo norte y se repetirá este proceso L veces.

Si se realiza estos pasos al final todos los nodos tendrán el número menor deseado.

Diseño del programa

Dentro del programa *toroide.c* tendremos nuestro diseño de programa:

```c
int main(int argc, char **argv){

    int rank, size, n_numbers, num_available = 0;
    double buf[1];

    /*Start MPI Application*/
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    distribute_numbers(rank, size, &n_numbers, &num_available, buf);
    compute_smallest(rank, n_numbers, buf, num_available);
    if(rank == 0) printf("\n\n************************AFTER COMPUTING SMALLEST NUMBER*****
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Primero se inicializará la aplicación y se distribuirán los números del fichero mediante el método *distribute_numbers()*. A este método se le pasará el rango del proceso MPI lanzado, el número de procesos lanzados, la cantidad de números que hay en el fichero, una variable que indica si un proceso contiene su número y un buffer que contendrá dicho número asociado a cada nodo.

Tras esto se llamará al método *compute*_smallest*()* para calcular el número menor y tras ello el proces de rango 0 imprimirá dicho número.

```c
void compute_smallest(int rank, int n_numbers, double *min_number, int num_available){
    print_message("*************PROPAGATION BETWEEN ROWS***************", rank);
    if (num_available) send_by_rows((rank+1), sqrt(n_numbers), min_number);
    print_message("*************PROPAGATION BETWEEN COLUMNS***************", rank);
    if(num_available) send_by_columns((rank+1), sqrt(n_numbers), min_number);
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Dentro del método *compute_smallest()* cada proceso llamará a los métodos *send_by_rows* y *send_by columns* si tienen un número asociado *(num_available)*. A estos métodos se les pasará el rango + 1 (ya que al encontrarnos con una red toroide es más fácil nombrar a cada nodo a partir del 1) el lado del toroide, que será igual a la raíz cuadrada del número total de números reales que hay en el fichero, y un puntero al buffer en el que se escribirá el número menor para cada proceso.

El método print_message permite imprimir un mensaje por parte del proceso de rango 0 y que este no se solape con otros mensajes gracias al uso de barreras.

```c
void print_message(char *string, int rank){
    MPI_Barrier(MPI_COMM_WORLD);
    if(rank == 0) printf("\n\n%s\n", string);
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Dentro de la propagación por filas se mandará el número al nodo este y se recibirá del nodo oeste. Sin embargo, primero hay que comprobar que el nodo en el que nos encontremos esté en la cara izquierda del toroide o que esté en la cara derecha. Por ejemplo, el nodo 1 tendrá que recibir del nodo 4 y el nodo 8 recibirá un dato del nodo 5.

Se realiza el envío y el recibo y se comprueba que el dato sea menor que el que ya se tenía, repitiendo este proceso L veces.

```
void send_by_rows(int rank, int side_length, double *min_number){
    double buf[1];
    int source = rank - 1, dest = rank + 1;
    MPI_Status status;

    if((rank % side_length) == 1 || rank == 1) source = rank + side_length - 1;

    if((rank % side_length) == 0) dest = rank - side_length + 1;

    printf("Rank: %d, Source: %d, Dest: %d\n",rank, source, dest);

    for(int i = 1; i <= side_length; i++){
        MPI_Send(min_number, 1, MPI_DOUBLE, (dest-1), i, MPI_COMM_WORLD);

        MPI_Recv(buf, 1, MPI_DOUBLE, (source-1), MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        if(*buf < *min_number) *min_number = *buf;
    }
}
```

Esta idea es similar para la propagación por columnas, primero se comprueba que nos encontremos en la cara superior o inferior del toroide y se repite las 2 operaciones L veces. Por ejemplo, el nodo 1 tendrá que enviar su dato al nodo 12

```
void send_by_columns(int rank, int side_length, double *min_number){
    double buf[1];
    int source = rank + side_length, dest = rank - side_length;
    MPI_Status status;

    if(rank > (side_length * (side_length - 1))) source = rank - (side_length * (side_length - 1));

    if(rank <= side_length) dest = rank + (side_length * (side_length - 1));

    printf("Rank: %d, Source: %d, Dest: %d\n", rank, source, dest);

    for(int i = 1; i <= side_length; i++){
        MPI_Send(min_number, 1, MPI_DOUBLE, (dest-1), i, MPI_COMM_WORLD);

        MPI_Recv(buf, 1, MPI_DOUBLE, (source-1), MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        if(*buf < *min_number) *min_number = *buf;
    }
}
```

Flujo de datos

Ahora pasamos a ver las directivas MPI utilizadas:

Dentro del método *distribute_numbers()* el proceso de rango 0 abrirá el fichero, guardará en un array todos los números del fichero y contará cuantos hay y comprobará que se han lanzado procesos suficientes. Además, su buffer lo igualará al primer elemento del array de números y la variable *num_available* pasará a ser 1.

```c
void distribute_numbers(int rank, int size, int *n_numbers, int *num_available, double buf[]){
    double *num_array;
    MPI_Status status;
    if(rank == 0){
        FILE *fp = NULL;
        num_array = malloc(MAX_ARRAY * sizeof(int));

        if((fp = fopen(FILENAME, "r")) == NULL){
            fprintf(stderr, "Error opening file: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }

        for(*n_numbers = 0; fscanf(fp, "%lf,", &num_array[*n_numbers]) == 1; (*n_numbers)++){};
        fclose(fp);

        if(size < *n_numbers){
            fprintf(stderr, "Not enough processes launched\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
            exit(EXIT_FAILURE);
        }
        buf[0] = num_array[0];
        *num_available = 1;
    }
```

Tras esto se realizará la operación colectiva *MPI_Bcast* cuyo proceso raíz tiene rango 0 para distribuir la cantidad de números que hay en el fichero al resto de procesos lanzados, esto será util para descartar procesos innecesarios en caso de que se hayan lanzado procesos de más.

Tras esto el método de rango 0 empezará a enviar los números y el resto de procesos los recibirán. La comprobación (rank < n_numbers) servirá para comprobar si el rango del proceso es menor que la cantidad de números en el fichero. Si es mayor o igual, ese proceso es innecesario y se habrán lanzado más procesos que números hay. En caso de recibir un número, la variable *num_available* se pondrá a 1 y tendrá un número disponible. Esto será útil para realizar la propagación por filas y columnas para que un proceso innecesario sin un número asociado no realice ninguna acción.

```c
MPI_Bcast(n_numbers, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0){
    send_numbers(num_array, *n_numbers);
    free(num_array);
}
else if (rank < *n_numbers){
    MPI_Recv(buf, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    *num_available = 1;
}

if(*num_available) printf("I'm process %d out of %d. My number is: %.2f\n", (rank+1), size, buf[0]);
MPI_Barrier(MPI_COMM_WORLD);
```

Para enviar los números el proceso 0 recorre el array de números y llama a la primitiva *MPI_Send()*.

```c
void send_numbers(double *num_array, int n_numbers){
    for(int i = 1; i < n_numbers; i++){
        MPI_Send((num_array + i), 1, MPI_DOUBLE, i, i, MPI_COMM_WORLD);
    }
}
```

Tras acabar con la distribución de números se empleará *MPI_Barrier()* para esperar a todos los procesos para no pasar a la ejecución del algoritmo antes de tiempo.

El resto de primitivas de MPI se han explicado con anterioridad a la hora de explicar el algoritmo a aplicar en la red toroide.

Compilación y Ejecución

Inicialmente se dispondrá un proyecto con la siguiente estructura:

```
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/MPI_p1$ ls
include  Makefile  mpi_config  numbers.dat  README.md  src
```

El directorio *src* contiene los archivos de código fuente *toroidal.c*, *app_init.c* y *hypercube.c*. El archivo *app_init.c* contiene métodos usados por los otros 2 archivos fuentes como *distribute_numbers* o *print_message*.

El directorio *include* contendrá el archivo *app_init.h* que tendrá declarados constantes y métodos usados por *toroidal.c* y *hypercube.c*.

El archivo *mpi_config* que contiene el número máximo de procesos que se permite lanzar, el archivo *numbers.dat* y un *README*.

Por último, se dispone de un *Makefile* para la construcción y compilación del proyecto.

Para compilar todo el proyecto se introduce *make all* por teclado.

```
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/MPI_p1$ make all
mkdir -p obj/ exec/ debug/
mpicc -Iinclude/ -c -Wall src/toroidal.c -o obj/toroidal.o
mpicc -Iinclude/ -c -Wall src/app_init.c -o obj/app_init.o
mpicc -o exec/toroidal obj/toroidal.o obj/app_init.o -lpthread -lrt -lm
mpicc -Iinclude/ -c -Wall src/hypercube.c -o obj/hypercube.o
mpicc -o exec/hypercube obj/hypercube.o obj/app_init.o -lpthread -lrt -lm
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/MPI_p1$ ls
debug  exec  include  Makefile  mpi_config  numbers.dat  obj  README.md  src
pacochechu@pacochechu-HP-Laptop-14-cf1xxx:~/UclmInformatica/3º/2º_cuatri/InfraRed/MPI_p1$
```

Se crearán los directorios *obj*, *exec* y *debug* con archivos objeto, de ejecución y de deupración respectivamente.

Tenemos 2 test, uno para la red toroide y otro para el hipercubo.

Para ejecutar el test del hipercubo ejecutamos *make test1*.

Conclusiones

Gracias al uso de la computación distribuidas y de MPI se ha conseguido obtener un programa de complejidad O(n^(1/2)) para un problema que resuelto sin ninguna forma de

paralelismo puede alcanzar incluso una complejidad cuadrática dependiendo del alogirtmo que se utilice. Por lo tanto, el uso de la computación distribuida para ciertas topologías de red será útile para la resolución de algoritmos complejos cuyo trabajo se pueda repartir entre varios núcleos de ejecución.

Red Toroide

Enunciado

Se nos pide distribuir una lista de números a una red hipercubo y calcular el elemento mayor de toda la red sin alcanzar una complejidad mayor que O(logaritmo_base_2(n)), siendo n el número de elementos de toda la red.

Planteamiento de la solución

Será necesario el uso de computación distribuida y una topología de red hipercubo que nos permita la propagación del elemento menor a todos los nodos mediante el uso de MPI.

Si nombramos los nodos empezando por el 0 y en binario, un nodo tendrá que enviar y recibir números de sus nodos adyacentes (por ejemplo, el nodo 00 al nodo 01 y 10, hay una diferencia de un bit) e ir comparando para encontrar el mayor. Por lo tanto, este proceso de envío y recibo se realizará D veces, siendo D las dimensiones del hipercubo.

Diseño del programa

La distribución de números en la red hipercubo será igual que en la red toroide, y la estructura del programa será la misma. De hecho, el *distruibute_numbers* es el mismo para ambas redes. Por lo tanto, hay que compilar los archivos objeto del toroide y el hipercubo con el fichero *app_init* que es el que contiene este método, entre otros.

La única diferencia será el método que emplea para calcular el número mayor.

Se calculará mediante la operación XOR (definida como directiva de preprocesado) el nodo sobre el que se tiene que enviar y recibir, esta operación se realiza entre el rango del proceso y la potencia de 2^i, siendo i la iteración en la que nos encontremos, tras enviar y recibir el número mediante primitvas MPI, se comprueba qué número es mayor para asignarlo a la variable *max_number*. Con esto conseguimos una complejidad de O(D).

Flujo de datos

El flujo de datos es similar al de la red toroide, excepto a la hora de computar el número mayor.

Compilación y Ejecución

Para compilar todo: *make all*.

Para ejecutar la red hipercubo: *make test2*.

Conclusiones

Para la red hipercubo conseguimos reducir la complejidad del programa, ya que, por ejemplo, para una red de 4 dimensiones, con 16 nodos podremos conseguir una complejidad de O(D = 4). Esto hace que nuestro programa sea muy escalable para un gran número de nodos.