Alberto Vázquez Martínez

Documentación práctica 1 – Diseño de Infraestructura de red

Red hipercubo

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($\log_2 N$) Con n número de elementos de la red.

Planteamiento y diseño de la solución

Lo primero que hay que tener en cuenta para formar la red hipercubo, es si el número de procesos que formarán la red hipercubo es correcto. Para ello calculamos el logaritmo en base 2 del número de procesos, si el resultado es un número entero, entonces, se podrá formar una red hipercubo. De lo contrario, el programa finalizará y el proceso 0 notificará al resto de procesos para que no continúen con la ejecución.

A continuación, leeremos el archivo *datos.dat* que contiene los números reales. El proceso 0 se encargará de realizar un envío al resto de procesos del número asignado. Los demás procesos recibirán dicho número, incluido el proceso 0, ya que, en esta solución consideraré que el proceso 0 pertenecerá a la red hipercubo. El envío será asíncrono (*MPI_Isend*) y la recepción será bloqueante (*MPI_Recv*) para así asegurarnos de que dato llega a los procesos, ya que, es necesario dicho dato para su correcta ejecución.

Cada proceso tendrá D vecinos, siendo D la dimensión de la red hipercubo, calculada como $\log_2 N$ (siendo N el número de procesos). Para calcular los vecinos, se sabe que la distancia de *haming* de dos nodos o procesos vecinos es de 1 por lo que podemos utilizar la operación lógica XOR. Si dos nodos son vecinos, entonces esta operación nos devolverá un número en binario con un único bit a 1. Dicho bit, indicará la dimensión en la que se encuentra la "relación" o enlace entre los nodos.

Ejemplo operación XOR

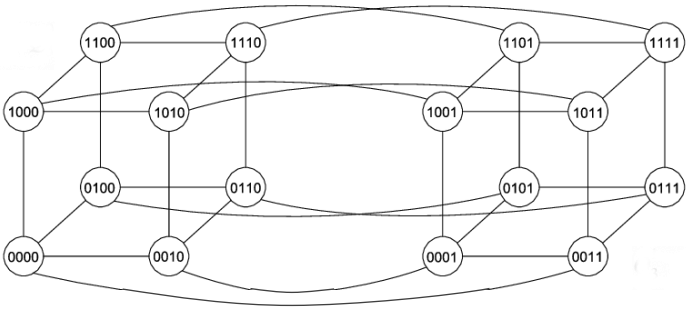
$$\begin{array}{r} 111 \\ \underline{110} \\ 001 \end{array} \quad 001 \rightarrow \textit{Dimensión } 1 \qquad \begin{array}{r} 111 \\ \underline{101} \\ 010 \end{array} \quad 010 \rightarrow \textit{Dimensión } 2 \ldots$$

Por tanto, tendremos una función que calcule los vecinos de un nodo/proceso dependiendo de su *rank* y del número de dimensiones de la red. Para ello utilizaremos varios bucles *for* donde, se compruebe con una operación XOR si solo cambia en un bit el resultado y, donde se asigne la posición correcta en un array de vecinos de acuerdo con su dimensión

En este caso se ha optado por almacenar los vecinos en el orden según su dimensión:

D1	D2	D3	D4	…	D_i

Una vez tenemos los vecinos de cada proceso, tendremos que realizar el envío, la recepción y la comparación de cada número. Para ello utilizamos un bucle *for* que irá desde 0 hasta la dimensión de la red ($\log_2 N$, siendo N el número de procesos). De esta manera, cada proceso enviará a todos sus procesos vecinos su número asignado y recibirá el número de sus vecinos igualmente.



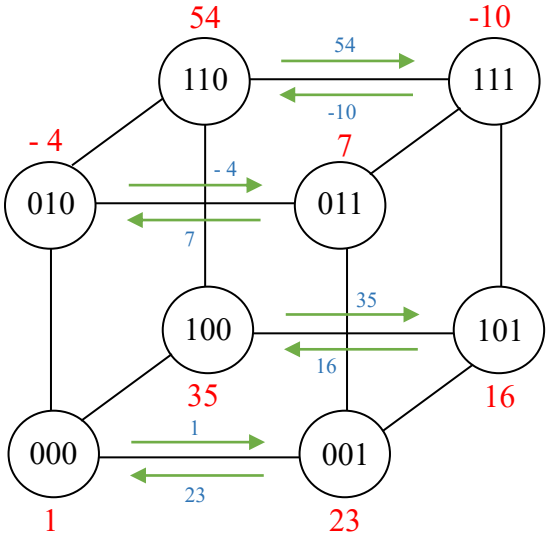
Finalmente, todos los procesos obtendrán el mayor número de la lista de números y el proceso 0 lo imprimirá por la salida estándar.

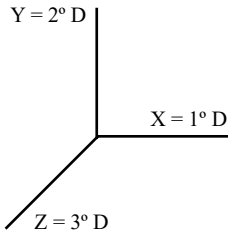
Explicación de flujo de datos

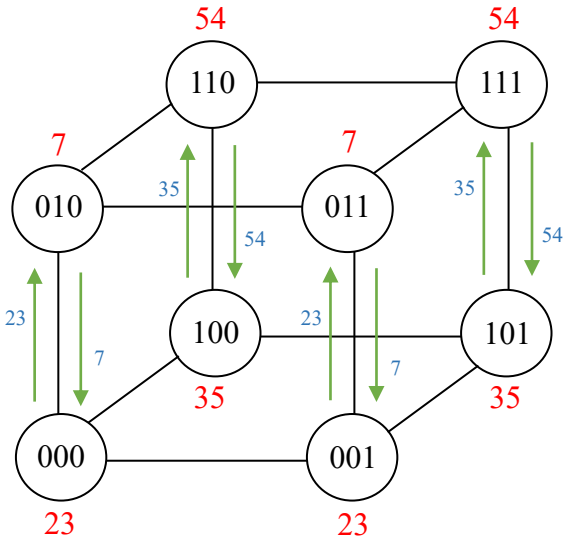
Para calcular cual es el número mayor de una lista de números asignado a los nodos de una red hipercubo, lo que haremos será ir calculando el número mayor de cada dimensión hasta llegar a la última dimensión.

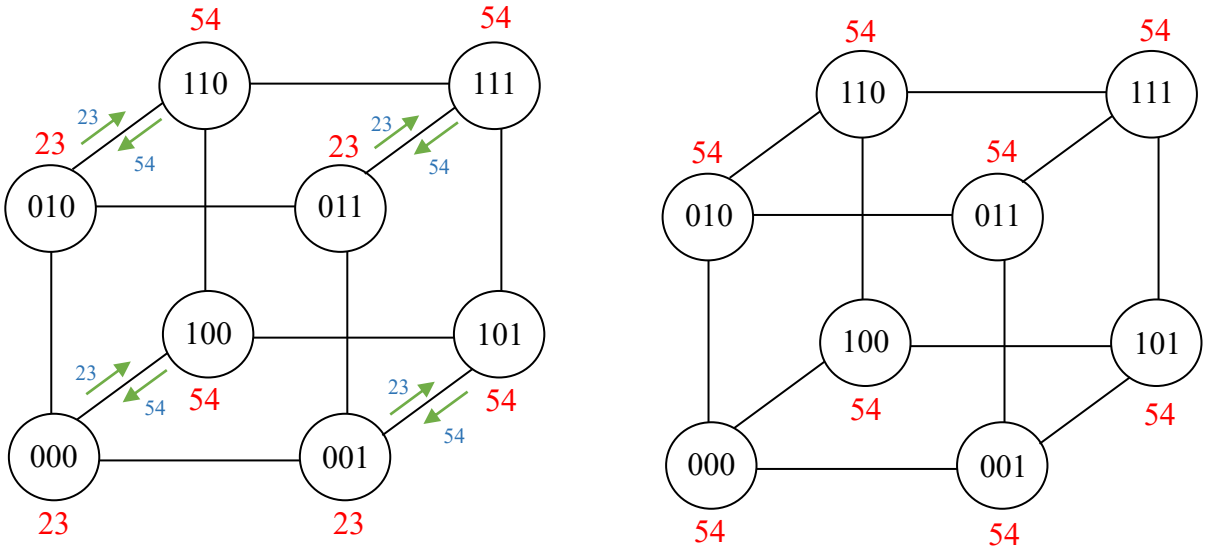
Ejemplo de ejecución con una red hipercubo de dimensión 3 (8 procesos)

Comenzaremos con la dimensión 1, después 2, … y así sucesivamente:









Para el envío y recepción de los datos, he optado por un envío asíncrono (*MPI_Isend* y *MPI_Irecv*), para así no bloquear a la aplicación con su ejecución, aunque si es necesario un método de sincronización para esperar al dato a recibir, y para ello utilizo *MPI_Test*.

Fuentes del programa

Hipercubo.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

#include <definitions.h>
#include "/usr/lib/x86_64-linux-gnu/openmpi/include/mpi.h"

void assign_neighbours(int *neighbours, int size, int rank, int dimension);
float search_max_number(int *neighbours, int size,  int dimension, float send_number, int rank);

/* Main function */
int main(int argc, char **argv) {
   int rank, size, dimension, hypercube = false;
   float buf,  max_number;

        MPI_Request request;
   MPI_Status status;

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

   if (rank == 0) {
      double res = log(size)/log(2);
      dimension = (int) res;
```

```
        if (dimension == res) {
            hypercube = true;
            send_network_topology_confirmation(hypercube, size, request);

            FILE* file = open_file("datos.dat", "r");
            read_assign_values(file, size);

        } else {
            send_network_topology_confirmation(hypercube, size, request);
            fprintf(stderr, "Error, can't create toroidal network: %d processes\n", size);
        }
    } else {
        MPI_Recv(&hypercube, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }

    if (hypercube == true) {
        dimension = log(size)/log(2);
        /* [Neighbour dimension 1, neighbour dimension 2, ... , neighbour dimension D...] */
        int neighbours[dimension];

        MPI_Recv(&buf, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        assign_neighbours(neighbours, size, rank, dimension);
        /*
        printf("Process %d neighbours: ",rank);
        for (int i = 0; i < dimension; i++) {
            printf("%d ",neighbours[i]);
        }
        printf("\n");
        */

        max_number = search_max_number(neighbours, size, dimension, buf, rank);

    }

    if (rank == 0) {
        printf("\n[Process %d] The maximum number is: %.2f\n\n", rank, max_number);
    }

    MPI_Finalize();
}

/* Function to compare which is the bigger number */
float compare_numbers(float number1, float number2) {

        if (number1 > number2) {
                return number1;
        } else {
                return number2;
        }
}
```

```
/* Search the max number sending the number assigned and asking to neighbours for their numbers
*/
float search_max_number(int *neighbours, int size, int dimension, float send_number, int rank) {

   MPI_Request request1;
        MPI_Request request2;
        MPI_Status status;

        float recv_number;

   for (int i = 0; i < dimension; i++) {
      MPI_Irecv(&recv_number, 1, MPI_FLOAT, neighbours[i], MPI_ANY_TAG,
MPI_COMM_WORLD, &request1);
      MPI_Isend(&send_number, 1, MPI_FLOAT, neighbours[i], i, MPI_COMM_WORLD,
&request2);
      MPI_Wait(&request1, &status);

                  send_number = compare_numbers(send_number, recv_number);
                  recv_number = send_number;
   }

   return recv_number;
}

/* Assign neighbouts using XOR operation */
void assign_neighbours(int *neighbours, int size, int rank, int dimension) {
   int cnt = 0;

   for (int i = 0; i < size; i++) {
      if (cnt == dimension) {
         break;
      } else {
         int res = rank^i;
         for (int j = 0; j < dimension; j++) {
            /* Check if only change 1 bit in res variable */
            if (res == (int) pow(2,j)) {
               for (int k = 0; k < dimension; k++) {
                  /* Check the dimension to assign it in the right position into the array */
                  if ((i == rank - (int) pow(2,k)) || (i == rank + (int) pow(2,k)))  {
                     neighbours[k] = i;
                     cnt++;
                     break;
                  }

               }

            }
         }
      }
   }
}
```

Definitions.h

```
#include "/usr/lib/x86_64-linux-gnu/openmpi/include/mpi.h"

#define BUFFER 2014
#define true 1
#define false 0

/* Open a file with a given permissions and returns a descriptor */
FILE* open_file(char filename[], char *permissions) {
        FILE *file;
        file = fopen(filename, permissions);

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

        return file;
}

/* Sends to process != 0 if the topology is correct */
void send_network_topology_confirmation(int network_topology, int size, MPI_Request request) {
   for (int i = 1; i < size; i++) {
      MPI_Isend(&network_topology, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
   }
}

/* Read and assign numbers to child processes*/
void read_assign_values(FILE *file, int size) {
        char buffer[BUFFER];
        float buf;

        fgets(buffer, BUFFER, file);
        char * token = strtok(buffer, ",");

        while ((token != NULL) && (size > 0)) {
                size--;
                buf = atof(token);
                MPI_Bsend(&buf, 1, MPI_FLOAT, size, 0, MPI_COMM_WORLD);
                token = strtok(NULL, ",");
        }
        fclose(file);
}
```

Instrucciones para compilar y ejecutar

La estructura de archivos y directorios es la siguiente:

- Src: código fuente en C.
- Include: archivos cabecera con funciones utilizadas en ambos programas.
- Exec: archivos ejecutables generados en la compilación.
- Datos.dat: Lista de números reales.
- Makefile: utilizado para compilar y ejecutar el código.

Para compilar el programa basta con ejecutar el comando **make**. Se compilarán todos los archivos C del directorio *src*.

Para ejecutar el programa: **make run_hipercubo**

Para limpiar el directorio *exec* de ejecutables utilizar el comando **make clean**.

Hay que tener en cuenta que el archivo *datos.dat* contiene los números justos para una red máxima de 32 procesos. Por tanto, si se intenta crear una red superior se deberían añadir más números a dicho archivo, de lo contrario, el programa dejará de funcionar.