



UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

DISEÑO DE INFRAESTRUCTURAS DE RED

GRADO EN INGENIERÍA INFORMÁTICA
2019 - 2020

Práctica 1: Red Hipercubo

Autor:
David Camuñas Sánchez

Fecha:
18 de marzo de 2020

Índice

1. Enunciado	2
2. Introducción	2
3. Planteamiento de la solución	3
3.1. Tipos de nodos	3
3.2. Algoritmos principales	4
3.2.1. Obtención de los vecinos	4
3.2.2. Cálculo del número máximo	5
4. Diseño de la solución	7
4.1. Función principal del programa (<i>main</i>)	7
4.2. Lectura del fichero (<i>load-data</i>)	8
4.3. Comprobación de la cantidad de números (<i>check-size</i>)	9
4.4. Envío de su número a cada nodo (<i>add-numbers</i>)	10
4.5. Obtención de los vecinos (<i>get-neighbors</i>)	11
4.6. Cálculo del número máximo (<i>calculate-min</i>)	12
4.6.1. Mejora del algoritmo	14

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 (2^D), los 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. Introducción

El objetivo de esta práctica es la creación de uno de los tipos de redes de comunicación existente, esta es la **Red Hipercubo**.

Cuya peculiaridad es que son redes cuadradas, donde cada nodo o *rank*, tiene D vecinos (uno por cada dimensión).

- **Por ejemplo:** *Un hipercubo de 2^3* En este caso, el tamaño del hipercubo es 8, esto significa, que está compuesto por 8 *nodos o ranks*. Donde cada *rank* tiene 3 *vecinos*, uno por cada dimensión ($D = 1, D = 2, D = 3$).

A continuación, se mostrará un ejemplo donde se puede observar varios tipos de dimensiones en un hipercubo.

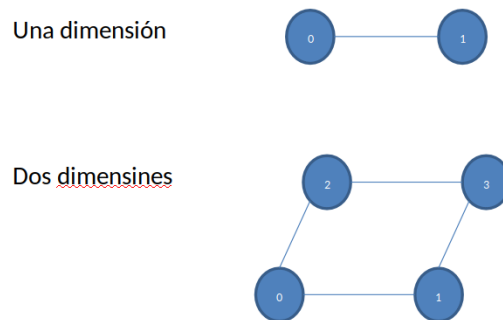


Figura 1: Ejemplo dimensiones de un hipercubo

3. Planteamiento de la solución

Para solucionar el problema, se tiene en cuenta que una *Red Hipercubo* tiene un tamaño de 2^D . Donde el *valor de D* está determinado por: $\log_2(D)$

En este caso el valor de **D** es **3**, por lo tanto, el tamaño (*size*) será **8**, esto quiere decir que este hipercubo estaría formado por *16 procesos* (denominados en el código como *ranks*).

El número de dimensiones **D** del hipercubo se encuentra definido en la *función main()*, como una constante:

```
const int D = log2(size)
```

Si se quiere realizar la simulación con un valor de lado distinto, se deberá cambiar el valor del número total de procesos a crear, pasado como argumento al comando de ejecución (*mpirun*) en la línea de ordenes. Este valor se puede encontrar en el *Makefile* del proyecto.

Otra constante importante, es la que determina el tamaño del buffer lectura del fichero (en este caso *datos.dat*), debido a que si se quiere leer una gran cantidad de números, y así crear una gran cantidad de nodos (*ranks*), se debe de modificar su valor a uno mayor.

```
define MAX-SIZE 1024
```

3.1. Tipos de nodos

En este problema encontramos dos tipos de nodos: el *rank 0* o *nodo 0* y los demás *nodos*.

- **Rank 0:** Este nodo corresponde al primer proceso creado. Encargado de la lectura del fichero *datos.dat*, el cual contiene los números que más tarde asignará el mismo a los demás nodos que forman la red hipercubo.

A la hora de realizar la asignación de los números a los respectivos nodos restantes (incluyéndose el mismo), debe de comprobar que la cantidad de números obtenidos del fichero es igual al tamaño del hipercubo **size** (n^0 de nodos que lo forman) si esta comprobación es exitosa, continuará la ejecución normal del programa.

En caso contrario, a mi elección bien sea por que el tamaño del hipercubo o la cantidad de los números sea menor o mayor. El *rank 0* abortará la ejecución del programa. Tanto si la comprobación es correcta como si no, este difundirá el resultado a los demás nodos. Para ello se ha utilizado la función *Bcast()* de la librería de **MPI**.

Una vez asignados los respectivos números a cada nodo, tras calcular el número máximo de la red hipercubo, *rank 0* mostrará dicho número por pantalla, y el programa

finalizará.

- **Los demás nodos:** Estos tipos de nodos recibirán del *rank 0* la decisión de continuar o no. Si continua la ejecución normal se les asignará un número real, el cual tras obtener cada uno sus respectivos vecinos, se llevará a cabo el algoritmo para obtener *el menor número de la red hipercubo*.

3.2. Algoritmos principales

Los algoritmos principales de esta práctica son: **la obtención de los vecinos** (de cada nodo) y **cálculo del número máximo** de la red hipercubo.

3.2.1. Obtención de los vecinos

Este algoritmo es fundamental en el programa, debido a que gracias a el se ve de una manera más intuitiva la estructura de la red hipercubo, donde cada nodo o rank conoce a sus respectivos vecinos. Cada nodo tiene un total de D *vecinos*, almacenador en el vector de tipo entero denominado *neighbors* (vecinos).

En este caso, los vecinos de se han obtenido mediante el cambio de los bits que forman el *rank* del nodo actual, en este caso según el rank que sea se han cambiado los bits dependiendo de la dimensión a la que pertenezca el vecino.

Este proceso se ha realizado con la ayuda de la operación lógica *XOR*, entre el *rank* actual más la dirección generada por cada dimensión. En la sección correspondiente al algoritmo se explicara de forma más práctica el proceso que sigue.

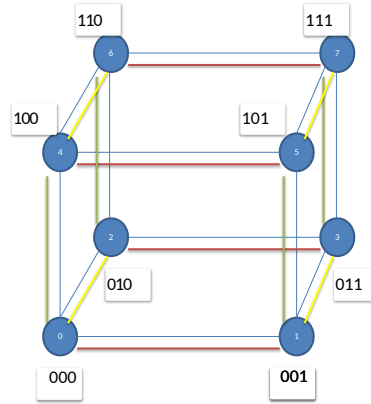


Figura 2: Representación red hipercubo 2^3 con los respectivos ranks.

3.2.2. Cálculo del número máximo

El segundo algoritmo principal de este programa, es el de calcular el número máximo de toda la red hipercubo, para ello cada nodo ira comprobando su número y el de sus vecinos y se quedara con el mayor de ellos, esta comprobación la realizará enviando su numero a cada vecino y recogiendo el de cada vecino para comprobarlo con el suyo.

Al realizar este proceso cada nodo, a nivel global se realizará para todos los nodos de la red, entonces de esta forma al tener cada nodo vecinos que a la vez ellos mismos son vecinos de otros nodos, todos los nodos se quedarán finalmente con el número mayor de toda la red hipercubo.

La secuencia de acciones de este algoritmo es la siguiente:

1. Intercambio de números entre vecino de cada dimensión.
2. Comparación entre el número actual y el recibido, quedandose el nodo actual con el número máximo.

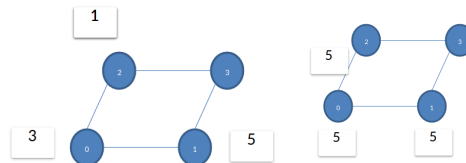


Figura 3: Representación cálculo del valor máximo de un rank y sus vecinos

4. Diseño de la solución

A continuación, se explicarán las funciones que forman parte del programa, el cual da una solución al enunciado de esta práctica.

4.1. Función principal del programa (*main*)

Esta es la función principal del programa, en la cual como se puede observar lo primero que se lleva a cabo es la definición de las variables a utilizar, como: *rank*, *numbers-n* (cantidad de números), *size* (tamaño del hipercubo), *data* (vector donde se almacenarán los números obtenidos del fichero), etc.

Tras esto se declararán las primitivas de **MPI** como:

- `textitMPI-Init()` para inicializar la estructura de comunicación de MPI entre los procesos.
- `MPI-Comm-Size()` para obtener el tamaño de la comunicacion (número de proceso a ejecutar en este caso, ranks), etc.
- `MPI-Comm-rank()` Para establecer el identificador de cada proceso *rank*.

También esta función se lleva a cabo las llamadas a las demás tipos de funciones que componen el programa

```
1  int main(int argc, char *argv[])
2  {
3
4      /*Variables*/
5      int rank, size, numbers_n, finish;
6      long double number, min_number;
7      long double *data = malloc(DATA_SIZE);
8      int *neighbors = malloc(NEIGHBORS_SIZE);
9      MPI_Status status;
10
11     /* Initialize MPI program */
12     MPI_Init(&argc, &argv);
13     MPI_Comm_size(MPI_COMM_WORLD, &size);
14     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16     if (rank == FIRST_RANK)
17     {
18         /*Get quantity of numbers*/
19         numbers_n = load_data(data);
```

```

20         finish = check_size(size, numbers_n);
21
22         if (finish != TRUE)
23         {
24             add_numbers(data, size);
25         }
26     }
27
28     MPI_Bcast(&finish, 1, MPI_INT, 0, MPI_COMM_WORLD);
29
30     if (finish != TRUE)
31     {
32         MPI_Recv(&number, 1, MPI_LONG_DOUBLE, 0, MPI_ANY_TAG,
33                 MPI_COMM_WORLD, NULL);
34         printf("[X] RANK[%d] --> %.2Lf\n", rank, number);
35
36         get_neighbors(rank, neighbors);
37
38         min_number = calculate_min(rank, number, neighbors);
39
40         print_min_number(rank, min_number);
41     }
42
43     /* Finalize MPI program */
44     MPI_Finalize();
45
46     return EXIT_SUCCESS;
47 }

```

4.2. Lectura del fichero (*load-data*)

La primera función a llamar dentro del *main*, es la *load-data()*, encargada de la lectura del fichero, esta función añade los números que almacena el fichero al vector de tipo *long double* denominado *data*. Esta función obtiene los números del fichero, separando cada uno utilizando como separador la *coma*.

Para la apertura del fichero se ha utilizado la función *open-file()*, cuyo principal objetivo es devolver el descriptor del archivo *datos.dat*. A esta función se le pasa como parámetro *DATA-PATH* que es la ruta del archivo a leer y *READ-MOD* que indica el modo de apertura del archivo, en este caso el modo de lectura.

```

1  /* Open file */
2  FILE *open_file(const char *path, const char *mode)

```



```

3 {
4     FILE *file;
5     if ((file = fopen(path, mode)) == NULL)
6     {
7         fprintf(stderr, "[X] RANK[0]: Error opening file.\n");
8         exit(EXIT_FAILURE);
9     }
10    return file;
11 }

```

Una vez cargados los datos en el vector *data*, la función devolverá la variable *i*, la cual almacena el valor de la cantidad de números que contiene el fichero.

```

1  /* Load data (numbers) from datos.dat */
2  int load_data(long double *data)
3  {
4      FILE *file = open_file(DATA_PATH, READ_MOD);
5      char line[MAX_SIZE];
6      char *token;
7      int i;
8
9      fgets(line, sizeof(line), file);
10     fclose(file);
11
12     data[0] = atof(strtok(line, SEPARATOR));
13
14     for (i = 1; (token = strtok(NULL, SEPARATOR)) != NULL; i++)
15     {
16         data[i] = atof(token);
17     }
18
19     return i;
20 }

```

4.3. Comprobación de la cantidad de números (*check-size*)

Esta función se encarga de comprobar si la cantidad de números (*numbers-n*) es igual al tamaño (*size*) del hipercubo. Estos son los argumentos que se les pasa a la función. La función devolverá True o False si es igual o no estas variables, si la devolución es False, entonces finalizará la ejecución normal del programa. Y el proceso *rank 0*, mandará una difusión a los demás procesos, para que así aborten su ejecución. Esto se realiza con la primitiva *MPI-Bcast()*.

```

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

```

```
1  /* Check toroid's size (number of nodes) */
2  int check_size(const int size, const int numbers_n)
3  {
4      int finish = FALSE;
5      if (size != numbers_n)
6      {
7          fprintf(stderr, "[X] RANK[0]: Error, quantity of numbers (%
8              d) is diferent at toroid's size (%d | L = %d)\n",
9              numbers_n, size, L);
10         finish = TRUE;
11     }
12     return finish;
13 }
```

4.4. Envío de su número a cada nodo (*add-numbers*)

Esta función tiene el objetivo de añadir o enviar a cada nodo (*rank*) su número, esta tarea es realizada por *rank 0*.

Tanto para el envío como para la recepción del nodo se han utilizado las primitivas básicas de MPI (*MPI-Send* y *MPI-Recv*). Esta función recoge como parámetros el vector *data* y la variable entera *size* (tamaño hipercubo).

```
1  /* Add number to nodes (ranks) */
2  void add_numbers(long double *data, const int size)
3  {
4      int i;
5      long double number;
6
7      for (i = 0; i < size; i++)
8      {
9          number = data[i];
10         MPI_Send(&number, 1, MPI_LONG_DOUBLE, i, SEND_TAG,
11             MPI_COMM_WORLD);
12     }
13     free(data);
14 }
```

Por último, se libera el espacio de memoria reservado para el vector de números *data*.

4.5. Obtención de los vecinos (*get-neighbors*)

Esta función se encarga de que cada proceso, obtenga el rank de cada vecino, es decir, de sus vecinos *Norte*, *Sur*, *Este* y *Oeste*. Para ello esta función recogerá como parámetros el rank del proceso que la ejecute, y su lista de vecinos vacía en este caso, representada mediante el vector de enteros llamado *neighbors*.

Esta función es muy importante para la función que calcula el máximo número de la red hipercubo *calculate-min()*, debido que aquí cada nodo o rank conoce quienes son sus vecinos, con los cuales realizará dicha comprobación intercambiando sus números.

El objetivo y funcionamiento de este algoritmo se ha explicado de una forma más teórica en el apartado 2, en concreto, la sección 2.2.1.

```
1 void get_neighbors(const int rank, int *neighbors)
2 {
3     int row = rank / L;
4     int column = rank % L;
5
6
7     switch (row)
8     {
9     case 0:
10         neighbors[NORTH] = L * (L - 1) + rank;
11         neighbors[SOUTH] = rank + L;
12         break;
13
14     case L - 1:
15         neighbors[NORTH] = rank - L;
16         neighbors[SOUTH] = rank % L;
17         break;
18
19     default:
20         neighbors[NORTH] = rank - L;
21         neighbors[SOUTH] = rank + L;
22         break;
23     }
24
25     switch (column)
26     {
27     case 0:
28         neighbors[WEST] = rank + (L - 1);
29         neighbors[EAST] = rank + 1;
30         break;
31
32     case L - 1:
33         neighbors[WEST] = rank - 1;
34         neighbors[EAST] = rank - (L - 1);
```

```

35         break;
36
37     default:
38         neighbors[WEST] = rank - 1;
39         neighbors[EAST] = rank + 1;
40         break;
41     }
42 }

```

Como se puede observar el proceso de obtención se debe en dos partes, la obtención de los nodos pertenecientes a las filas y los pertenecientes a lo que serían las columnas.

- **Filas:** por medio de las filas se obtienen los vecinos *Norte* y *Sur*, esto se realiza con la utilización de las fórmulas de la Tabla ??.
- **Columnas:** por medio de las columnas se obtienen los vecinos *Este* y *Oeste*, esto se realiza con la utilización de las fórmulas de la Tabla ??.

La implementación en ambas partes se ha realizado con un *switch*, en el cual dependiendo de la posición del *rank* se obtendrá de una forma u otra los vecinos.

4.6. Cálculo del número máximo (*calculate-min*)

El objetivo principal de esta función es la del cálculo del número máximo de la red hipercubo completa, en el cual todos los elementos de la red deben de contener finalmente el número máximo.

Para ello se explicará a continuación de una forma más práctica el procedimiento de este algoritmo, dado que en la sección 2.2.2 ya ha sido explicado de una forma más teórica.

Por tanto, una vez obtenidos los vecinos de cada nodo, se lleva a cabo el cálculo del máximo. La primera versión de este algoritmo estaba compuesta por dos bucles de tipo *for*, en los cuales primero, se calculaba los números de los vecinos correspondientes a las filas *Norte* y *Sur*, enviando a *Sur* el número que contiene el nodo actual y recibiendo del *Norte* su número. Y más tarde el mismo procedimiento con las columnas *Este* y *Oeste*. Con calcular me refiero a quedarse el nodo actual finalmente con el menor de cada número. Para que esto sea aplicable a los elementos de toda la fila o de toda la columna se harían 1 a L-1 iteraciones en cada bucle.

Por lo que el procedimiento a seguir sería el siguiente:

- **Primer bucle *for*** (cálculo de las filas):
 1. El *rank* actual envía su número (*my-number*) al vecino del *Sur*.

2. El *rank* actual recibe el número de su vecino del *Norte* (*his-number*).
 3. Se realiza el cálculo del número máximo entre estos dos número (se coge el menor de ellos).
- **Segundo bucle *for*** (cálculo de las columnas):
1. El *rank* actual envía su número (*my-number*) al vecino del *Este*.
 2. El *rank* actual recibe el número de su vecino del *Oeste* (*his-number*).
 3. Se realiza el cálculo del número máximo entre estos dos número (se coge el menor de ellos).

```

1  /*Calculate the minium value*/
2  long double calculate_min(const int rank, long double my_number,
3  int *neighbors)
4  {
5      int i;
6      long double his_number;
7
8      for (i = 1; i < L; i++)
9      {
10         /* Calculate rows */
11         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[SOUTH],
12             SEND_TAG, MPI_COMM_WORLD);
13         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[NORTH],
14             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
15         my_number = (his_number > my_number ? my_number :
16             his_number);
17     }
18
19     for (i = 1; i < L; i++){
20         /* Calculate columns */
21         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[EAST],
22             SEND_TAG, MPI_COMM_WORLD);
23         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[WEST],
24             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
25         my_number = (his_number > my_number ? my_number :
26             his_number);
27     }
28
29     free(neighbors);
30
31     return my_number;
32 }

```

Como se puede observar, una vez realizado ambos bucles, todos los elementos de la red tendrán almacenado el número máximo.

4.6.1. Mejora del algoritmo

Una vez desarrollada la primera versión del algoritmo de calcular el máximo, se observó, que realmente se realizaría lo mismo dejando simplemente un bucle *for*, y además esto podría mejorar el rendimiento del programa.

El procedimiento de esta versión mejorada, es mas peculiar, debido a que no se calculan de forma secuencial, todos los vecinos de las filas (*Norte-Sur*) y después los de las columnas *Este-Oeste*, si no que cada nodo en una misma interacción calculará ambas opciones. El procedimiento a seguir sera el siguiente:

En un único bucle *for*:

1. El *rank* actual envía su número (*my-number*) al vecino del *Sur*.
2. El *rank* actual recibe el número de su vecino del *Norte* (*his-number*).
3. Se realiza el cálculo del número máximo entre estos dos número, obtenidos de las filas (se coge el menor de ellos).
4. El *rank* actual envía su número (*my-number*) al vecino del *Este*.
5. El *rank* actual recibe el número de su vecino del *Oeste* (*his-number*).
6. Se realiza el cálculo del número máximo entre estos dos número, obtenidos de las columnas (se coge el menor de ellos).

```

1  /*Calculate the minium value*/
2  long double calculate_min(const int rank, long double my_number,
3  int *neighbors)
4  {
5      int i;
6      long double his_number;
7
8      for (i = 1; i < L; i++)
9      {
10         /* Calculate rows */
11         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[SOUTH],
12             SEND_TAG, MPI_COMM_WORLD);
13         MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[NORTH],
14             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
15         my_number = (his_number > my_number ? my_number :
16             his_number);
17
18         /* Calculate columns */
19         MPI_Send(&my_number, 1, MPI_LONG_DOUBLE, neighbors[EAST],
20             SEND_TAG, MPI_COMM_WORLD);

```

```

16     MPI_Recv(&his_number, 1, MPI_LONG_DOUBLE, neighbors[WEST],
17             MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
18     my_number = (his_number > my_number ? my_number :
19                 his_number);
20 }
21
22 free(neighbors);
23
24 return my_number;
25 }

```

Por último en ambas opciones, se libera el espacio de memoria reservado para el vector de los vecinos *neighbors*.

Un ejemplo de representación del estado de la matriz, en un inicio y al final, tras haber calculado el máximo, sería el siguiente:

5	--	4	--	7		0	--	0	--	0
6	--	2	--	0	----->	0	--	0	--	0
8	--	3	--	1		0	--	0	--	0

Figura 4: Representación red hipercubo al inicio y al final