



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

DISEÑO DE INFRAESTRUCTURA DE RED

3º INGENIERÍA DE COMPUTADORES

MPI. Practica 1

Autor:
José Ángel Martín Baos

Fecha:
24 de abril de 2017

Índice

1. Red Toroide	2
1.1. Enunciado	2
1.2. Planteamiento de la solución	2
1.3. Diseño y explicación del programa	2
1.4. Fuentes del programa	3
1.5. Instrucciones de compilación y ejecución	6
1.6. Conclusiones	7
2. Red Hipercubo	7
2.1. Enunciado	7
2.2. Planteamiento de la solución	7
2.3. Diseño y explicación del programa	8
2.4. Fuentes del programa	8
2.5. Instrucciones de compilación y ejecución	11
2.6. Conclusiones	11

1. Red Toroide

1.1. 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 $L \times L$ 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.

La complejidad del algoritmo no superará $O(\text{raiz_cuadrada}(n))$ con n número de elementos de la red.

1.2. Planteamiento de la solución

Para resolver este ejercicio vamos a usar una red toroide. Las redes toroides se usan en redes de computadores de alto rendimiento. Es una malla en la cual sus filas y columnas tienen conexiones en anillo y son muy apropiadas en grandes instalaciones.

Su funcionamiento consiste en que cada nodo intercambia datos únicamente con sus vecinos. Aplicando esto a nuestro programa, primero se intercambiaran los datos por columnas, dónde cada nodo mandará su dato a su vecino inferior (al sur) $L-1$ veces (donde L es el lado de la red, o sea, la raíz cuadrada del número de nodos de la red). Después se repetirá este mismo procedimiento por filas.

De esta manera, obtendremos la solución al ejercicio con $2 * \sqrt{N} - 1$ mensajes (siendo N el número total de nodos de la red), por lo tanto logramos cumplir el objetivo de que la complejidad del algoritmo no superase $O(\text{raiz_cuadrada}(N))$.

1.3. Diseño y explicación del programa

Debemos resaltar algunas cuestiones de diseño. La primera tiene que ver con la lectura y distribución de los distintos elementos del archivo. El proceso cuyo *rank* sea el 0, será el que leerá el archivo y distribuirá sus datos mediante la llamada a la función `read_and_distribute_token()`. Con la llamada a está función se leerán tantos elementos como el numero de procesos lanzados. Debe haber un número de elementos en el archivo mayor o igual al de procesos lanzados. En caso de que se lancen menos proceso que elementos hay en el archivo, solo se leerán los primeros. El resto de procesos llamarán a la función `obtain_token()` que obtendrá su elemento asociado. Estos elementos serán

números en punto flotante separados por comas.

Para calcular los vecinos de un nodo (vecinos norte, sur, este y oeste) se ha usado la función `obtain_neighbors`. Esta función mediante unas pequeñas cuentas obtiene el *rank* de los vecinos de un determinado nodo (proceso). Para calcular el nodo norte suma al *rank* del proceso actual el tamaño del lado y realiza el módulo con el número total de procesos. Para el nodo sur el proceso es similar, pero restando el tamaño del lado y teniendo en cuenta que el resultado no sea negativo. Para el nodo este, suma al *rank* del nodo actual 1, teniendo en cuenta que este no sea múltiplo de *L*, en cuyo caso hay que restarle *L*. Para el caso del nodo oeste, el proceso es similar al anterior pero restando 1 y teniendo en cuenta que si es múltiplo de *L* - 1 hay que sumarle *L*.

El algoritmo que ejecuta cada proceso consiste en dos bucles *for*. Ambos iterarán tantas veces como el valor de *L* (el valor del lado). El primero de estos bucles permitirá que los procesos se manden los datos por columnas, como se indicó en el planteamiento de la solución. De tal manera que cada proceso mandará al que tenga en el sur, y recibirá del que tenga en el norte. El segundo hará lo propio en filas, mandando su *token* al proceso en el este y recibiendo el que tenga en el oeste. En cada iteración de estos bucles se comprueba cuál es el menor, y este es conservado como *token*.

Finalmente el proceso con *rank* 0 será el encargado de mostrar el elemento menor de la red, que será el estado actual de su *token*. Este elemento será el mismo en todos los nodos de la red en este momento.

1.4. Fuentes del programa

Listado 1: `red.toroides.c`

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define L 4
5  #define DATOS_FILE "datos.dat"
6
7  void read_and_distribute_token(float* token, int rank, int numtasks){
8      int rc;
9      MPI_Status status;
10     MPI_Request request;
11     FILE* file = fopen(DATOS_FILE , "r");
12     int i = 0;
13     float element = 0;
14     while (!feof (file) && i < numtasks){
15         fscanf (file, "%g,", &element);
16         if(i == rank){
17             *token = element;
18         }else{
```

```
19     rc = MPI_Isend(&element, 1, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &
20         request);
21     if (rc != MPI_SUCCESS) {
22         printf("Send error in task %d\n", rank);
23         MPI_Abort(MPI_COMM_WORLD, rc);
24         exit(1);
25     }
26     MPI_Wait(&request, &status);
27     i++;
28 }
29 fclose (file);
30 }
31
32 void obtain_token(float* token, int rank){
33     int rc;
34     MPI_Status status;
35     rc = MPI_Recv(token, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
36         status);
37     if (rc != MPI_SUCCESS) {
38         printf("Receive error in task %d\n", rank);
39         MPI_Abort(MPI_COMM_WORLD, rc);
40         exit(1);
41     }
42 }
43
44 void obtain_neighbors(int* north, int* south, int* east, int* west, int
45     rank,
46     int numtasks){
47     //North neighbor
48     *north = (rank + L) % numtasks;
49
50     //South neighbor
51     *south = (rank - L) % numtasks;
52     if(*south < 0){
53         *south += numtasks;
54     }
55
56     //East neighbor
57     *east = rank + 1;
58     if(*east % L == 0){
59         *east -= L;
60     }
61
62     //West neighbor
63     *west = (rank - 1) % numtasks;
64     if(*west % L == L-1){
65         *west += L;
66     }else if(*west == -1){
```

```
65     *west = L-1;
66 }
67 }
68
69 int main (int argc, char *argv[]){
70     int numtasks,          // Number of MPI tasks
71         rank,              // Task number of the current process
72         north,             // Rank of the north neighbor
73         south,             // Rank of the south neighbor
74         east,              // Rank of the east neighbor
75         west;              // Rank of the west neighbor
76     float token;          // The number from datos.dat for this process
77
78     MPI_Init(&argc, &argv);
79     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
80     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
81
82     if (numtasks != L*L && rank == 0){
83         printf("Error: %d task are needed.\n", L*L);
84         fflush(stdout);
85         MPI_Abort(MPI_COMM_WORLD, 1);
86         exit(1);
87     }
88
89     obtain_neighbors(&north, &south, &east, &west, rank, numtasks);
90
91     if(rank == 0){
92         read_and_distribute_token(&token, rank, numtasks);
93     }else{
94         obtain_token(&token, rank);
95     }
96
97     // Algorithm
98     int i, rc;
99     float in_token;
100     MPI_Status status;
101     for (i = 0; i < L-1; i++){
102         rc = MPI_Send(&token, 1, MPI_FLOAT, south, i, MPI_COMM_WORLD);
103         if (rc != MPI_SUCCESS) {
104             printf("Send error in task %d\n", rank);
105             MPI_Abort(MPI_COMM_WORLD, rc);
106             exit(1);
107         }
108         rc = MPI_Recv(&in_token, 1, MPI_FLOAT, north, MPI_ANY_TAG,
109                     MPI_COMM_WORLD,
110                     &status);
111         if (rc != MPI_SUCCESS) {
112             printf("Receive error in task %d\n", rank);
113             MPI_Abort(MPI_COMM_WORLD, rc);
```

```
113     exit(1);
114 }
115 if(in_token < token){
116     token = in_token;
117 }
118 }
119
120 for (i = 0; i < L-1; i++){
121     rc = MPI_Send(&token, 1, MPI_FLOAT, east, i, MPI_COMM_WORLD);
122     if (rc != MPI_SUCCESS) {
123         printf("Send error in task %d\n", rank);
124         MPI_Abort(MPI_COMM_WORLD, rc);
125         exit(1);
126     }
127     rc = MPI_Recv(&in_token, 1, MPI_FLOAT, west, MPI_ANY_TAG,
128                  MPI_COMM_WORLD,
129                  &status);
130     if (rc != MPI_SUCCESS) {
131         printf("Receive error in task %d\n", rank);
132         MPI_Abort(MPI_COMM_WORLD, rc);
133         exit(1);
134     }
135     if(in_token < token){
136         token = in_token;
137     }
138 }
139 if(rank == 0){
140     printf("El menor elemento de toda la red es %g.\n", token);
141 }
142
143 MPI_Finalize();
144 exit(0);
145 }
```

1.5. Instrucciones de compilación y ejecución

Para compilar el ejercicio vamos a ejecutar el siguiente comando:

```
$ make red_toroide
```

Para ejecutarlo vamos a usar la siguiente orden:

```
$ mpirun -n 16 exec/red_toroide
```

Como se puede observar se ha usado la opción `-n 16` para crear 16 procesos dado que

la constante L se ha definido a 4. En caso de que esta fuera otra, se deben crear $L * L$ procesos.

También podemos usar la orden siguiente para ejecutar el programa:

```
$ make test_red_toroide
```

1.6. Conclusiones

Las conclusiones que se obtienen de la realización de este ejercicio están relacionadas con las ventajas que conlleva saber en que tipo de arquitectura está corriendo nuestro programa. En este caso, estamos simulando que nuestro programa va a correr en un sistema distribuido en el cual los nodos están conectados usando una red Toroide.

En este caso, hemos aprovechado las peculiaridades de la red, para poder ejecutar el algoritmo con una complejidad inferior a $O(\sqrt{N})$. Además, con este algoritmo no solo mejoramos la complejidad, sino que el flujo de datos por la red está controlado y ordenado. De esta forma estamos evitando colisiones y mejorando de una manera muy notoria el rendimiento de la red.

2. Red Hipercubo

2.1. 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 , los 2^D 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 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 $O(\logaritmo_base_2(n))$ Con n número de elementos de la red.

2.2. Planteamiento de la solución

Para resolver este segundo ejercicio usaremos una red hipercubo. Las redes hipercubo son redes formadas por una malla de D dimensiones en las que se suprimen los nodos interiores. Por ejemplo, una malla de dimensión 1 sólo tiene 2 nodos. El grado de los nodos

de una red hipercubo de D dimensiones es D . Estas redes han sido bastante utilizadas en computadores paralelos, pero han ido evolucionando en otras topologías más escalables.

Su funcionamiento consiste en que cada nodo intercambia mensajes sólo con aquellos nodos con los que está conectado y sólo en la dimensión en la que estemos iterando. Un nodo está conectado con otro si y sólo si su distancia de Hamming es 1, es decir, sólo varía un bit en sus representaciones numéricas en binario. Dependiendo de que bit cambie podemos saber en que dimensión nos estamos comunicando. Por lo tanto, lo que haremos será iterar en todas las dimensiones, y en ellas intercambiaremos mensajes entre los distintos nodos con aquel nodo resultante de mutar el bit correspondiente a esa dimensión.

De esta manera, si disponemos de N nodos en la red, la red será de dimensión $\log_2 N$. Dado que tenemos que iterar en todas las dimensiones, la complejidad del algoritmo será de $\log_2 N$, cumpliendo por tanto con el objetivo establecido en cuanto a complejidad.

2.3. Diseño y explicación del programa

En cuanto al diseño del programa, hemos reutilizado algunas funciones del programa anterior: `read_and_distribute_token()` y `obtain_token()`. Además se ha añadido una nueva función: `obtain_neighbor`, que permite obtener el vecino de cada nodo dada una dimensión. Por ejemplo, el vecino del nodo 2 en la dimensión 1 es el 3 y viceversa.

El funcionamiento de la función `obtain_neighbor` es el siguiente: En un bucle iteramos en todos los posibles nodos, y comparamos la representación binaria del nodo actual con cada uno de estos nodos, hasta obtener uno que solo cambie un bit (distancia hamming de 1). Además, este bit que varía debe corresponder con la dimensión en la cual estamos obteniéndolo. Para ello podemos usar la operación OR entre nuestro nodo y el posible nodo y compararlo con el número $2^{dimension-1}$. Este último número tiene una representación binaria que consiste en un 1 en la posición correspondiente a la dimensión deseada y el resto 0. Aunque este no es el mecanismo más eficiente para lograr el resultado, resulta muy sencillo de implementar.

El algoritmo que ejecuta cada proceso simplemente utiliza un bucle for, en el cual se van recorriendo las diferentes dimensiones de la red. En cada interacción del bucle (una dimensión distinta), se envía y mensaje al nodo con el que está conectado en dicha dimensión y recibe otro mensaje de este. Se compara si el *token* recibido es mayor que el asignado, y de ser así, se sustituye el asignado al proceso por el recibido.

Finalmente el proceso con *rank* 0 será el encargado de mostrar el elemento mayor de la red, que será el estado actual de su *token*. Este elemento será el mismo en todos los nodos de la red en este momento.

2.4. Fuentes del programa

Listado 2: red_hipercubo.c

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #define D 4 // Number of dimensions
6  #define DATOS_FILE "datos.dat"
7
8  void read_and_distribute_token(float* token, int rank, int numtasks){
9      int rc;
10     MPI_Status status;
11     MPI_Request request;
12     FILE* file = fopen(DATOS_FILE , "r");
13     int i = 0;
14     float element = 0;
15     while (!feof (file) && i < numtasks){
16         fscanf (file, "%g,", &element);
17         if(i == rank){
18             *token = element;
19         }else{
20             rc = MPI_Isend(&element, 1, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &
                request);
21             if (rc != MPI_SUCCESS) {
22                 printf("Send error in task %d\n", rank);
23                 MPI_Abort(MPI_COMM_WORLD, rc);
24                 exit(1);
25             }
26             MPI_Wait(&request, &status);
27         }
28         i++;
29     }
30     fclose (file);
31 }
32
33 void obtain_token(float* token, int rank){
34     int rc;
35     MPI_Status status;
36     rc = MPI_Recv(token, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
        status);
37     if (rc != MPI_SUCCESS) {
38         printf("Receive error in task %d\n", rank);
39         MPI_Abort(MPI_COMM_WORLD, rc);
40         exit(1);
41     }
42 }
43
44 int obtain_neighbor(int rank, int dimension){
45     int neighbor, node;
46     for(node = 0; node < (int)pow(2,D); node++){
```

```

47     if((rank ^ node) == (int)pow(2,dimension - 1)){
48         neighbor = node;
49         break;
50     }
51 }
52 return neighbor;
53 }
54
55 int main (int argc, char *argv[]){
56     int numtasks,          // Number of MPI tasks
57     rank;                  // Task number of the current process
58     float token;           // The number from datos.dat for this process
59
60     MPI_Init(&argc, &argv);
61     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
62     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
63
64     if (numtasks != (int)pow(2,D) && rank == 0){
65         printf("Error: %d task are needed.\n", (int)pow(2,D));
66         fflush(stdout);
67         MPI_Abort(MPI_COMM_WORLD, 1);
68         exit(1);
69     }
70
71     if(rank == 0){
72         read_and_distribute_token(&token, rank, numtasks);
73     }else{
74         obtain_token(&token, rank);
75     }
76
77     // Algorithm
78     int i, rc;
79     float in_token;
80     MPI_Status status;
81     for (i = 1; i <= D; i++){
82         rc = MPI_Send(&token, 1, MPI_FLOAT, obtain_neighbor(rank, i), i,
83                     MPI_COMM_WORLD);
84         if (rc != MPI_SUCCESS) {
85             printf("Send error in task %d\n", rank);
86             MPI_Abort(MPI_COMM_WORLD, rc);
87             exit(1);
88         }
89
90         rc = MPI_Recv(&in_token, 1, MPI_FLOAT, obtain_neighbor(rank, i),
91                     MPI_ANY_TAG,
92                     MPI_COMM_WORLD, &status);
93         if (rc != MPI_SUCCESS) {
94             printf("Receive error in task %d\n", rank);
95             MPI_Abort(MPI_COMM_WORLD, rc);

```

```
95     exit(1);
96 }
97
98     if(in_token > token){
99         token = in_token;
100     }
101 }
102
103 if(rank == 0){
104     printf("El mayor elemento de toda la red es %g.\n", token);
105 }
106
107 MPI_Finalize();
108 exit(0);
109 }
```

2.5. Instrucciones de compilación y ejecución

Para compilar el ejercicio vamos a ejecutar el siguiente comando:

```
$ make red_hipercubo
```

Para ejecutarlo vamos a usar la siguiente orden:

```
$ mpirun -n 16 exec/hipercubo
```

Como se puede observar se ha usado la opción `-n 16` para crear 16 procesos dado que la constante D se ha definido a 4. En caso de que esta fuera otra, se deben crear 2^D procesos.

También podemos usar la orden siguiente para ejecutar el programa:

```
$ make test_red_hipercubo
```

2.6. Conclusiones

Las conclusiones obtenidas son iguales que las del ejercicio anterior. Hemos podido observar las ventajas que conlleva programar teniendo en mente las características físicas de la red subyacente. De esta manera y dadas las propiedades de la red hemos logrado una complejidad de $\log_2 N$. Y de igual manera que en el ejercicio anterior, también obtenemos la ventaja de que el flujo de datos por la red está ordenado y controlado.

Este tipo de redes son muy caras, dado que escalan de manera muy costosa. Subir de

una dimensión a otra incrementa de manera exponencial el número de nodos necesarios. Por ello, a veces se implementa cada nodo como un router con varios nodos en él.

Referencias

- [1] Openmpi documentation. <https://www.open-mpi.org/doc/>. [Online; accedido el 31-marzo-2017].
- [2] Redes de interconexión. <https://www.infor.uva.es/~bastida/Arquitecturas%20Avanzadas/Redes.pdf>. [Online; accedido el 8-abril-2017].
- [3] Javier Ayllon. *Apuntes de Diseño de Infraestructura de Red*, 2017.