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:
8 de abril de 2017

Índice

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 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.

La complejidad del algoritmo no superará O(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 raiz 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(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.

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).

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 iteracción 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_toroide.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(int* 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   int element = 0;
14   while (!feof (file) && i < numtasks){
15     fscanf (file, "%d", &element);
16     if(i == rank){
17       *token = element;
18     }else{
19       rc = MPI_Isend(&element, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &
           request);
20       if (rc != MPI_SUCCESS) {
21          printf("Send error in task %d\n", rank);
22          MPI_Abort(MPI_COMM_WORLD, rc);
23          exit(1);
24       }
25     }
26     i++;
```

```
27    }
28    fclose (file);
29  }
30
31  void obtain_token(int* token, int rank){
32    int rc;
33    MPI_Status status;
34    rc = MPI_Recv(token, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
          status);
35    if (rc != MPI_SUCCESS) {
36       printf("Receive error in task %d\n", rank);
37       MPI_Abort(MPI_COMM_WORLD, rc);
38       exit(1);
39    }
40  }
41
42  void obtain_neighbors(int* north, int* south, int* east, int* west, int
        rank,
43    int numtasks){
44    //North neighbor
45    *north = (rank + L)  % numtasks;
46
47    //South neighbor
48    *south = (rank - L)  % numtasks;
49    if(*south < 0){
50      *south += numtasks;
51    }
52
53    //East neighbor
54    *east = rank + 1;
55    if(*east  % L == 0){
56      *east -= L;
57    }
58
59    //West neighbor
60    *west = (rank - 1)  % numtasks;
61    if(*west  % L == L-1){
62      *west += L;
63    }else if(*west == -1){
64      *west = L-1;
65    }
66  }
67
68  int main (int argc, char *argv[]){
69    int numtasks,       // Number of MPI tasks
70        rank,           // Task number of the current process
71        north,          // Rank of the north neighbor
72        south,          // Rank of the south neighbor
73        east,           // Rank of the east neighbor
```

```
74          west,           // Rank of the west neighbor
75          token;          // The number from datos.dat for this process
76
77      MPI_Init(&argc, &argv);
78      MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
79      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
80
81      if (numtasks != L*L && rank == 0){
82        printf("Error: %d task are needed.", L*L);
83        MPI_Abort(MPI_COMM_WORLD, 1);
84        exit(1);
85      }
86
87      obtain_neighbors(&north, &south, &east, &west, rank, numtasks);
88
89      if(rank == 0){
90        read_and_distribute_token(&token, rank, numtasks);
91      }else{
92        obtain_token(&token, rank);
93      }
94
95      // Algorithm
96      int i, in_token, rc;
97      MPI_Status status;
98      for (i = 0; i < L-1; i++){
99        rc = MPI_Send(&token, 1, MPI_INT, south, i, MPI_COMM_WORLD);
100       if (rc != MPI_SUCCESS) {
101          printf("Send error in task %d\n", rank);
102          MPI_Abort(MPI_COMM_WORLD, rc);
103          exit(1);
104       }
105       rc = MPI_Recv(&in_token, 1, MPI_INT, north, MPI_ANY_TAG,
              MPI_COMM_WORLD,
106                     &status);
107       if (rc != MPI_SUCCESS) {
108          printf("Receive error in task %d\n", rank);
109          MPI_Abort(MPI_COMM_WORLD, rc);
110          exit(1);
111       }
112       if(in_token < token){
113         token = in_token;
114       }
115     }
116
117     for (i = 0; i < L-1; i++){
118       rc = MPI_Send(&token, 1, MPI_INT, east, i, MPI_COMM_WORLD);
119       if (rc != MPI_SUCCESS) {
120          printf("Send error in task %d\n", rank);
121          MPI_Abort(MPI_COMM_WORLD, rc);
```

```
122         exit(1);
123       }
124       rc = MPI_Recv(&in_token, 1, MPI_INT, west, MPI_ANY_TAG,
              MPI_COMM_WORLD,
125                     &status);
126       if (rc != MPI_SUCCESS) {
127         printf("Receive error in task %d\n", rank);
128         MPI_Abort(MPI_COMM_WORLD, rc);
129         exit(1);
130       }
131       if(in_token < token){
132         token = in_token;
133       }
134     }
135
136     if(rank == 0){
137       printf("El menor elemento de toda la red es %d.\n", token);
138     }
139
140     MPI_Finalize();
141     exit(0);
142   }
```

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 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.

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 destribuirá 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

2.3. Diseño y explicación del programa

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(int* 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    int element = 0;
15    while (!feof (file) && i < numtasks){
16      fscanf (file, "%d", &element);
17      if(i == rank){
18        *token = element;
19      }else{
20        rc = MPI_Isend(&element, 1, MPI_INT, 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      }
27      i++;
28    }
29    fclose (file);
30  }
31
32  void obtain_token(int* token, int rank){
33    int rc;
34    MPI_Status status;
35    rc = MPI_Recv(token, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
          status);
36    if (rc != MPI_SUCCESS) {
37       printf("Receive error in task %d\n", rank);
38       MPI_Abort(MPI_COMM_WORLD, rc);
39       exit(1);
40    }
41  }
42
43  int obtain_neighbor(int rank, int dimension){
44    int neighbor, node;
45    for(node = 0; node < (int)pow(2,D); node++){
46      if((rank ^ node) == (int)pow(2,dimension - 1)){
47        neighbor = node;
48        break;
49      }
50    }
51    return neighbor;
52  }
53
54  int main (int argc, char *argv[]){
55    int numtasks,       // Number of MPI tasks
56        rank,           // Task number of the current process
57        token;          // The number from datos.dat for this process
58
59    MPI_Init(&argc, &argv);
```

```
60    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
61    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
62
63    if (numtasks != (int)pow(2,D) && rank == 0){
64      printf("Error: %d task are needed.\n", (int)pow(2,D));
65      fflush(stdout);
66      MPI_Abort(MPI_COMM_WORLD, 1);
67      exit(1);
68    }
69
70    if(rank == 0){
71      read_and_distribute_token(&token, rank, numtasks);
72    }else{
73      obtain_token(&token, rank);
74    }
75
76    // Algorithm
77    int i, in_token, rc;
78    MPI_Status status;
79    for (i = 1; i <= D; i++){
80      rc = MPI_Send(&token, 1, MPI_INT, obtain_neighbor(rank, i), i,
81                    MPI_COMM_WORLD);
82      if (rc != MPI_SUCCESS) {
83         printf("Send error in task %d\n", rank);
84         MPI_Abort(MPI_COMM_WORLD, rc);
85         exit(1);
86      }
87
88      rc = MPI_Recv(&in_token, 1, MPI_INT, obtain_neighbor(rank, i),
           MPI_ANY_TAG,
89                    MPI_COMM_WORLD, &status);
90      if (rc != MPI_SUCCESS) {
91         printf("Receive error in task %d\n", rank);
92         MPI_Abort(MPI_COMM_WORLD, rc);
93         exit(1);
94      }
95
96      if(in_token > token){
97        token = in_token;
98      }
99    }
100
101   if(rank == 0){
102     printf("El mayor elemento de toda la red es %d.\n", token);
103   }
104
105   MPI_Finalize();
106   exit(0);
107 }
```

2.5. Instrucciones de compilación y ejecución

2.6. Conclusiones

Referencias

[1] Openmpi documentation. `https://www.open-mpi.org/doc/`. [Online; accedido el 31-marzo-2017].

[2] Javier Ayllon. *Apuntes de Diseño de Infraestructura de Red*, 2017.