UCLM - Escuela Superior de Informática - Ciudad Real

Diseño e Infraestructura de Redes.
Memoria Práctica 1.

Hecho por:
Benjamín Cádiz de Gracia.

Fecha: Febrero de 2018.

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 obtener nuestra solución vamos a utilizar una *red toroide*. La *red toroide* es una malla que permite comunicar un dato con un vecino suyo. Nuestro planteamiento sera utilizar este tipo de red para mediante, primero por filas y despues por columnas, obtener el mínimo de todos los valores asociados a cada nodo de la red.

1.3. Diseño del programa

El programa está dividido en:

- **Main**. Un main principal, donde su función será la del algoritmo principal de obtener el número mas pequeño.
- **Obtener vecinos**. Una función donde podremos calcular la posición de todos y cada uno de los vecinos. Simplemente será una función para que cada nodo tenga una referencia de donde están todos sus vecinos.
- **Distribuir**. Un método cuya función será leer el fichero donde contiene los datos, y por cada dato se distribuirá con el fin de asignarlo a su nodo correspondiente.
- **Obtener**. Este método se complementa con el anterior, ya que recibirá el dato enviado y además, se asignará el dato.

1.4. Flujo de datos en la red.

1.5. Código fuente.

```
1  //
2  //  red_toroide.c
3  //
4  //
5  //  Created by Benjamin Cadiz de Gracia on 27/2/18.
6  //
7
8  #include <mpi.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 #define DATOS "Datos.dat"
```

```
13  #define L 3
14
15  void obtain_neighbors(int* north, int* south, int* east, int* west, int rank,
16                        int numtasks){
17
18      *south = (rank + L) % numtasks;
19
20      *north = (rank - L) % numtasks;
21      if(*north < 0){
22          *north += numtasks;
23      }
24
25      *east = rank + 1;
26      if(*east % L == 0){
27          *east -= L;
28      }
29
30      *west = (rank - 1) % numtasks;
31      if(*west % L == L-1){
32          *west += L;
33      }else if(*west == -1){
34          *west = L-1;
35      }
36  }
37
38  void distribuir(int numtask, int rank, float* token){
39      int i = 0;
40      int rc;
41      FILE* file;
42      float num[16];
43      MPI_Status status;
44      MPI_Request request;
45      float element=0;
46
47      //Leemos el archivo.
48      if ((file=fopen(DATOS,"r"))==NULL){
49          fprintf(stderr, "Error opening the file\n");
50          exit(EXIT_FAILURE);
51      }else
52          while(!feof (file) & (i < numtask)){
53              fscanf (file,"%g,", &element);
54
55              //Enviamos en el primer elemento.
56              if(i == rank){
57                  *token = element;
58              }else{
59                  rc = MPI_Isend(&element, 1, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &request);
60                  if (rc != MPI_SUCCESS) {
61                      printf("Send error in task %d\n", rank);
62                      MPI_Finalize();
63                      exit(1);
64                  }
65                  MPI_Wait(&request, &status);
66              }
67              i++;
68          }
69          fclose(file);
70  }
71  void obtain_token(float* token, int rank){
72      int rc;
73
74      MPI_Status status;
```

```
75      rc = MPI_Recv(token, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
76      if (rc != MPI_SUCCESS) {
77          printf("Receive error in task %d\n", rank);
78          MPI_Finalize();
79          exit(1);
80      }
81  }
82
83  int main(int argc, char** argv) {
84      float token;
85      FILE* file;
86
87      int rank,size,numtask;
88      int  north,south,east,west;
89
90      // Inicializacion de MPI.
91      MPI_Init(&argc, &argv);
92      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
93      MPI_Comm_size(MPI_COMM_WORLD, &numtask);
94
95      //Obtenemos vecinos.
96      obtain_neighbors(&north,&south,&east,&west,rank,numtask);
97
98      //Distribuimos los elementos del fichero en nuestros nodos.
99      if(rank == 0){
100         distribuir(numtask,rank, &token);
101     }else{
102         obtain_token(&token, rank);
103     }
104
105     //Realizamos el algoritmo.
106     int rc;
107     float new_token;
108     MPI_Status status;
109     for( int i =0;i<L-1;i++){
110         rc = MPI_Send(&token,1, MPI_FLOAT, north,i,MPI_COMM_WORLD);
111         if (rc != MPI_SUCCESS){
112             printf("Receive error in task %d\n", rank);
113             MPI_Finalize();
114             exit(1);
115         }
116         rc = MPI_Recv(&new_token,1,MPI_FLOAT,south,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
117         if (rc != MPI_SUCCESS){
118             printf("Receive error in task %d\n", rank);
119             MPI_Finalize();
120             exit(1);
121         }
122         if(new_token < token){
123             token = new_token;
124         }
125     }
126
127     for( int i =0;i<L-1;i++){
128         rc = MPI_Send(&token,1, MPI_FLOAT, west,i,MPI_COMM_WORLD);
129         if (rc != MPI_SUCCESS){
130             printf("Receive error in task %d\n", rank);
131                 MPI_Finalize();
132             exit(1);
133         }
134         rc = MPI_Recv(&new_token,1,MPI_FLOAT,east,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
135         if (rc != MPI_SUCCESS){
136             printf("Receive error in task %d\n", rank);
```

```
137                 MPI_Finalize();
138                 exit(1);
139             }
140             if(new_token < token)
141                 token = new_token;
142         }
143 
144         if(rank ==0)
145             printf("The minimun value is:  %g \n",token);
146 
147         MPI_Finalize();
148         return EXIT_SUCCESS;
149     }
```

1.6. Compilación y ejecución.

Debemos situarnos en la carpeta donde están los archivos *red_toroide.c*, *red_hipercubo.c* y ***Makefile***. Allí abriremos una terminal y ejecutaremos:

- Primero limpiamos todos los posibles ejecutables que haya en el proyecto.

```
1  make clean
```

- En segundo lugar compilaremos bien *red_toroide.c* o bien todo.

```
1  make red_toroide                //Compila   nicamente   el toroide.
2  make all                                        //Compila todo
```

- En tercer lugar lanzaremos el ejecutable.

```
1  make run_toroide
```

Cabe decir que por defecto nos viene 9 hilos para la ejecución.

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 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(logaritmo_base_2(n)))) Con n número de elementos de la red.

2.2. Planteamiento de la solución.

Para obtener nuestra solución vamos a utilizar una *red toroide*. La *red toroide* es una malla que permite comunicar un dato con un vecino suyo. Nuestro planteamiento sera utilizar este tipo de red para mediante, primero por filas y despues por columnas, obtener el mínimo de todos los valores asociados a cada nodo de la red.

2.3. Diseño del programa

El programa está dividido en:

- **Main**. Un main principal, donde su función será la del algoritmo principal de obtener el número mas pequeño.
- **Obtener vecinos**. Una función donde podremos calcular la posición de todos y cada uno de los vecinos. Simplemente será una función para que cada nodo tenga una referencia de donde están todos sus vecinos.
- **Distribuir**. Un método cuya función será leer el fichero donde contiene los datos, y por cada dato se distribuirá con el fin de asignarlo a su nodo correspondiente.
- **Obtener**. Este método se complementa con el anterior, ya que recibirá el dato enviado y además, se asignará el dato.

2.4. Flujo de datos en la red.

2.5. Código fuente.

```
1  //
2  //  red_hipercubo.c
3  //
4  //
5  //  Created by Benjamin Cadiz de Gracia on 16/3/18.
6  //
7
8  #include <mpi.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <math.h>
12
13 #define DATOS "Datos.dat"
14 #define D 4
15
16 int obtain_neighbor(int rank, int dimension){
17     int neighbor, node;
18     for(node = 0; node < (int)pow(2,D); node++){
19         if((rank ^ node) == (int)pow(2,dimension - 1)){
20             neighbor = node;
21             break;
22         }
23     }
24     return neighbor;
25 }
26 void distribuir(int numtask, int rank, float* token){
27     int i = 0;
28     int rc;
29     FILE* file;
```

```
30      float num[16];
31      MPI_Status status;
32      MPI_Request request;
33      float element=0;
34
35      //Leemos el archivo.
36      if ((file=fopen(DATOS,"r"))==NULL){
37                  fprintf(stderr, "Error opening the file\n");
38          exit(EXIT_FAILURE);
39      }else
40          while(!feof (file) & (i < numtask)){
41              fscanf (file,"%g,", &element);
42
43              //Enviamos en el primer elemento.
44              if(i == rank){
45                  *token = element;
46              }else{
47                  rc = MPI_Isend(&element, 1, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &request);
48                  if (rc != MPI_SUCCESS) {
49                      printf("Send error in task %d\n", rank);
50                      MPI_Finalize();
51                      exit(1);
52                  }
53                  MPI_Wait(&request, &status);
54              }
55              i++;
56          }
57      fclose(file);
58  }
59  void obtain_token(float* token, int rank){
60      int rc;
61
62      MPI_Status status;
63      rc = MPI_Recv(token, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
64      if (rc != MPI_SUCCESS) {
65          printf("Receive error in task %d\n", rank);
66          MPI_Finalize();
67          exit(1);
68      }
69  }
70
71  int main(int argc, char** argv) {
72      float token;
73
74      int rank,size,numtask;
75      FILE* file;
76
77      // Inicializacion de MPI.
78      MPI_Init(&argc, &argv);
79      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
80      MPI_Comm_size(MPI_COMM_WORLD, &numtask);
81
82
83      //Distribuimos los elementos del fichero en nuestros nodos.
84      if(rank == 0){
85          distribuir(numtask,rank, &token);
86      }else{
87          obtain_token(&token, rank);
88      }
89
90      //Realizamos el algoritmo.
91      int rc;
```

```
92      float new_token;
93      MPI_Status status;
94      for( int i =1;i<=D;i++){
95          rc = MPI_Send(&token,1, MPI_FLOAT,obtain_neighbor(rank, i),i,MPI_COMM_WORLD);
96
97          if (rc != MPI_SUCCESS){
98              printf("Receive error in task %d\n", rank);
99              MPI_Finalize();
100             exit(1);
101         }
102         rc = MPI_Recv(&new_token,1,MPI_FLOAT,obtain_neighbor(rank, i),MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
103         if (rc != MPI_SUCCESS){
104             printf("Receive error in task %d\n", rank);
105             MPI_Finalize();
106             exit(1);
107         }
108         if(new_token > token){
109             token = new_token;
110         }
111     }
112
113     if(rank ==0)
114         printf("The maximum value is:  %g \n",token);
115     MPI_Finalize();
116     return EXIT_SUCCESS;
117 }
```

2.6. Compilación y ejecución.

Debemos situarnos en la carpeta donde están los archivos *red_toroide.c*, *red_hipercubo.c* y ***Makefile***. Allí abriremos una terminal y ejecutaremos:

- Primero limpiamos todos los posibles ejecutables que haya en el proyecto.

```
1  make clean
```

- En segundo lugar compilaremos bien *red_hipercubo.c* o bien todo.

```
1  make red_hipercubo      //Compila   nicamente   el toroide.
2  make all                                          //Compila todo
```

- En tercer lugar lanzaremos el ejecutable.

```
1  make run_toroide
```

Cabe decir que por defecto nos viene 16 hilos para la ejecución.

3. Makefile.

```
1  #!/usr/bin/make -f
2  # -*- mode:makefile -*-
3
```

```
4  CC := mpicc
5  DIREXEC := ejecutables/
6  RUN := mpirun
7
8  dirs:
9          mkdir -p $(DIREXEC)
10
11 red_toroide: dirs
12         $(CC) red_toroide.c -o $(DIREXEC)toroide
13
14 red_hipercubo: dirs
15         $(CC) red_hipercubo.c -o $(DIREXEC)hipercubo
16
17 all: red_toroide red_hipercubo
18
19 run_hipercubo: red_hipercubo
20         $(RUN) -n 16 $(DIREXEC)hipercubo
21
22 run_toroide: red_toroide
23         $(RUN) -n 9 $(DIREXEC)toroide
24
25 clean:
26         rm -rf $(DIREXEC)
```