Programación Paralela (2015-2016)
Lenguajes y Sistemas de Información

Grado en Ingeniería Informática

E. T. S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Práctica 1: Implementación distribuida de un algoritmo paralelo de datos usando MPI

Francisco Javier Bolívar Lupiáñez

28 de marzo de 2016

Índice

Índice de figuras

1. Planteamiento

En esta práctica se llevará a cabo la paralelización del **algoritmo Floyd** para la búsqueda del camino más corto en un grafo.

Se desarrollarán dos versiones:

- **Unidimensional**: Se repartirán las filas de la matriz a los procesos.
- **Bidimensional**: Se repartirán submatrices de la matriz a los procesos.

1.1. Algoritmo de Floyd

El algoritmo de Floyd deriva una matriz en N pasos (tantos como número de nodos), obteniendo en cada paso una matriz intermedia con el camino más corto entre cada par de nodos.

1.1.1. Pseudocódigo

```
1 M[i][j] = A
2 for k = 0 to N-1
3         for i = 0 to N-1
4                 for j = 0 to N-1
5                         M[i][j] = min(M[i][j], M[i][k] + M[k][j])
```

2. Solución

2.1. Versión unidimensional

Para solucionarlo con este enfoque, asumiendo que el tamaño del problema es divisible entre el número de procesos, **cada proceso tendrá una matriz local de tamaño** $N/P \times N$.

El reparto se realizará por bloques. Es decir. Al primer proceso le corresponderán las primeras N/P filas, al segundo las siguientes... Por ejemplo, si el tamaño del problema es 8 y tenemos 4 procesos, al P_0 le corresponderás las filas 0 y 1, al P_1 las 2 y 3, al P_2 las 4 y 5 y al P_3 las 6 y 7.

En el cálculo de cada submatriz resultado, cada proceso necesitará, en el paso k, la fila k y puede tener suerte y ser suya o no y corresponderle a otro proceso. Entonces debería comunicarse con éste para poder realizar el cálculo. Por tanto, en cada iteración del primer bucle k, **el proceso detectará si la fila k le pertenece y si es así, hace un *broadcast* al resto de procesos**.

Por tanto, para solucionar el problema, nos basta con un ***scatter* para repartir la matriz**, un ***broadcast* para difundir cada fila k** y un ***gather* para recolectar la**

matriz resultado y el único problema que nos podríamos encontrar es el traducir de local a global un índice según lo que se necesite.

2.1.1. Pseudocódigo

```
M[i][j] = A
for k = 0 to N–1
        broadcast(K)
        for i = 0 to N/P–1
                for j = 0 to N–1
                        M[i][j] = min(M[i][j], M[i][k] + K[j])
```

2.1.2. Problemas y soluciones

El principal problema que se puede encontrar en esta versión es, como he comentado anteriormente, el **traducir un índice de local en un proceso a global**.

No obstante, si se trabaja con índices locales, es decir, el búcle va desde 0 a N/P, solo se necesita el índice global para comprobar que no se está iterando sobre la diagonal de la matriz (i=j, i=k o j=k).

El otro problema con el que me encontré fue durante el *broadcast* de la fila k. Y es que al escribir la sentencia `MPI_BCast` no cambié el índice del proceso raíz y siempre mandaba la fila k el proceso 0. Esto es fácil de corregir, una vez detectado el fallo, pues el índice del proceso raíz siempre será k / tamaño de bloque (siendo el tamaño de bloque N/P).

```
#include <iostream>
#include <fstream>
#include <string.h>
#include "Graph.h"
#include "mpi.h"

//#define PRINT_ALL

using namespace std;

int main (int argc, char *argv[])
{
  /**
    * Paso 1: Iniciar MPI y obtener tamano e id para cada proceso
    */
  int rank, size, tama;

  MPI_Init(&argc, &argv); // Inicializamos la comunicacion de los procesos
  MPI_Comm_size(MPI_COMM_WORLD, &size); // Numero total de procesos
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Valor de nuestro identificador

  /**
    * Paso 2: Comprobar entradas
    */
  if (argc != 2) { // Debe haber dos argumentos
    if (rank == 0) { // El proceso 0 imprime el error
      cerr << "Sintaxis: " << argv[0] << " <archivo de grafo>" << endl;
```

```
28      }
29      MPI_Finalize();
30      return -1;
31    }
32
33    /**
34      * Paso 3: Crear grafo y obtener numero de vertices
35      */
36    Graph G;
37    int nverts;
38
39    if (rank == 0) { // Solo lo hace un proceso
40      G.lee(argv[1]);
41      #ifdef PRINT_ALL
42        cout << "El grafo de entrada es:" << endl;
43        G.imprime();
44      #endif
45      nverts = G.vertices;
46    }
47
48    /**
49      * Paso 4: Hacer broadcast del numero de vertices a todos los procesos
50      */
51    MPI_Bcast(&nverts, 1, MPI_INT, 0, MPI_COMM_WORLD);
52
53    /**
54      * Paso 5: Reservar espacio para matriz y fila k
55      */
56    int tamaLocal, tamaBloque;
57
58    tamaLocal = nverts * nverts / size;
59    tamaBloque = nverts / size;
60
61    int M[tamaBloque][nverts], K[nverts]; // Matriz local y fila k
62
63    /**
64      * Paso 6: Repartir matriz entre los procesos
65      */
66    MPI_Scatter(G.ptrMatriz(), tamaLocal, MPI_INT, &M[0][0], tamaLocal, MPI_INT, 0,
67                MPI_COMM_WORLD);
68
69
70    /**
71      * Paso 7: Bucle principal del algoritmo
72      */
73    int i, j, k, vikj, iGlobal, iIniLocal, iFinLocal, kEntreTama, kModuloTama;
74
75    iIniLocal = rank * tamaBloque; // Fila inicial del proceso (valor global)
76    iFinLocal = (rank + 1) * tamaBloque; // Fila final del proceso (valor global)
77
78    double t = MPI_Wtime();
79
80    for (k = 0; k < nverts; k++) {
81      kEntreTama = k / tamaBloque;
82      kModuloTama = k % tamaBloque;
83      if (k >= iIniLocal && k < iFinLocal) { // La fila K pertenece al proceso
84        copy(M[kModuloTama], M[kModuloTama] + nverts, K);
85      }
86      MPI_Bcast(K, nverts, MPI_INT, kEntreTama, MPI_COMM_WORLD);
87      for (i = 0; i < tamaBloque; i++) { // Recorrer las filas (valores locales)
88        iGlobal = iIniLocal + i; // Convertir la fila a global
89        for (j = 0; j < nverts; j++) {
```

```
90            if (iGlobal != j && iGlobal != k && j != k) { // No iterar sobre diagonal
91              vikj = M[i][k] + K[j];
92              vikj = min(vikj, M[i][j]);
93              M[i][j] = vikj;
94            }
95          }
96        }
97      }
98
99      t = MPI_Wtime() - t;
100
101     /**
102       * Paso 8: Recoger resultados en la matriz
103       */
104     MPI_Gather(&M[0][0], tamaLocal, MPI_INT, G.ptrMatriz(), tamaLocal, MPI_INT, 0,
105                MPI_COMM_WORLD);
106
107     /**
108       * Paso 9: Finalizar e imprimir resultados
109       */
110     MPI_Finalize();
111
112     if (rank == 0) { // Solo lo hace un proceso
113       #ifdef PRINT_ALL
114         cout << endl << "Solucion:" << endl;
115         G.imprime();
116         cout << "Tiempo gastado = " << t << endl << endl;
117       #else
118         cout << t << endl;
119       #endif
120     }
121 }
```