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 de figuras

3.1. Gráfica de tiempo para P = 4 y N = 64, 128, 256, 512 y 1024 11
3.2. Gráfica de ganancia para P = 4 y N = 64, 128, 256, 512 y 1024 11

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

2.1.1. Descripción

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.

2.1.2. Pseudocódigo

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

2.1.3. 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).

2.1.4. Código

```
1  #include <iostream>
2  #include <fstream>
3  #include <string.h>
4  #include "Graph.h"
5  #include "mpi.h"
6
7  //#define PRINT_ALL
8
9  using namespace std;
10
11 int main (int argc, char *argv[])
12 {
13   /**
14     * Paso 1: Iniciar MPI y obtener tamano e id para cada proceso
15     */
16   int rank, size, tama;
17
18   MPI_Init(&argc, &argv); // Inicializamos la comunicacion de los procesos
19   MPI_Comm_size(MPI_COMM_WORLD, &size); // Numero total de procesos
20   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Valor de nuestro identificador
```

```
21
22    /**
23      * Paso 2: Comprobar entradas
24      */
25    if (argc != 2) { // Debe haber dos argumentos
26      if (rank == 0) { // El proceso 0 imprime el error
27        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 }
```

2.2. Versión bidimensional

2.2.1. Descripción

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

El reparto, a diferencia del enfoque anterior, no es inmediato. En la versión unidimensional realizábamos un *scatter* directamente porque se repartían celdas consecutivas en memoria. En este caso, al distribuir submatrices cuadradas, **deberemos definir un tipo de dato MPI**.

Se realizarán $\sqrt{P}$ particiones en cada dimensión, obteniendo P submatrices de tamaño $N/\sqrt{P} \times N/\sqrt{P}$ cada una. Se repartirán a los procesos de izquierda a derecha y arriba

abajo. Si, por ejemplo, tuviésemos 9 procesos, la de arriba a la izquierda le correspondería al P_0, la de arriba al centro al P_1, la de arriba a la derecha al P_2, la del centro a la izquierda al P_3, la del centro al P_4, la del centro a la derecha al P_5, la de abajo a la izquierda al P_6, la de abajo al centro al P_7 y la de abajo a la derecha al P_8.

Si para el reparto y recolección del resultado se complica la cosa, para el cálculo de éste también. Y es que antes, con la repartición unidimensional, tan solo se necesitaba la fila k, pero **ahora se necesitan valores en las dos dimensiones para calcular el resultado**: hace falta una subfila k y una subcolumna k de los procesos que están colocados en la misma columna y fila respectivamente.

Para llevar a cabo estas comunicaciones hará falta definir **dos comunicadores** y asignarlos a cada proceso para que, en el comunicador horizontal se comuniquen aquellos que se encuentran en la misma fila y en el vertical los que están en la misma columna.

2.2.2. Pseudocódigo

```
1 M[i][j] = A
2 for k = 0 to N-1
3   broadcast(filK)
4   broadcast(colK)
5   for i = 0 to N/sqrt(P)-1
6     for j = 0 to N/sqrt(P)-1
7       M[i][j] = min(M[i][j], ColK[i] + FilK[j])
```

2.2.3. Problemas y soluciones

La complicación de la implementación de esta versión, como ya se comentaron, son el trabajar un **tipo de dato MPI personalizado** y con **comunicadores para filas y columnas**.

Para trabajar con un tipo de dato personalizado, hay que definirlo y empaquetarlo para poder hacer tanto el *scatter* como el *gather*.

Se define con `MPI_Tipe_vector` y se le pasa el número de grupos de bloque (en nuestro caso, las filas de la submatriz), el número de elementos de cada bloque (columnas de la submatriz), la separación entre un bloque y otro (tamaño del problema), el tipo de dato (entero) y el nombre del tipo de dato MPI que se usará. Una vez definido, se realiza un `MPI_Type_commit`.

A continuación hay que **empaquetarlos en un búfer de envío**, para eso se realizarán tantas iteraciones en un bucle como número de procesos haya. En cada una de estas iteraciones se tiene que empaquetar con `MPI_Pack` a la que se le pasarán como parámentros el puntero al inicio del búfer de entrada (para ello se debe calcular el desplazamiento desde el inicio de la matriz para cada proceso), el número de datos de entrada (en nuestro caso

1), el tipo de dato (que será el que definimos anteriormente), el puntero al búfer de salida, el tamaño de éste, la posición (que inicializamos a 0 y el propio `MPI_Pack` se encarga de modificar) y el comunicador. De entre todas estas variables, la única que habría que calcular es el desplazamiento desde el inicio de la matriz para apuntar al inicio de cada submatriz y sería:

$$desplazamiento = col_P \times tama_{bloque} + fil_P \times tama_{bloque}^2 \times \sqrt{P}$$

donde:

$$col_P = i/\sqrt{P}$$

$$fil_P = i\,\%\sqrt{P}$$

$$tama_{bloque} = N/\sqrt{P}$$

Una vez empaquetado se puede realizar el *scatter* con los siguientes parámetros: El búfer de entrada sería el búfer de envío que usamos como búfer de salida en el `MPI_Pack`, el tamaño el necesario para almacenar los enteros de la submatriz, el tipo de dato `MPI_PACKED`, el búfer de recepción el puntero a la submatriz local, el tamaño el de la submatriz, el tipo de dato entero, el proceso raíz el 0 (que ha sido el que ha realizado todas las operaciones anteriores de definición y empaquetado del tipo de dato) y el comunicador el global.

2.2.4. Código

```
1  #include <iostream>
2  #include <fstream>
3  #include <string.h>
4  #include <math.h>
5  #include "Graph.h"
6  #include "mpi.h"
7
8  //#define PRINT_ALL
9
10 using namespace std;
11
12 int main (int argc, char *argv[])
13 {
14   /**
15     * Paso 1: Iniciar MPI y obtener tamano e id para cada proceso
16     */
17   int rank, size, tama;
18
19   MPI_Init(&argc, &argv); // Inicializamos la comunicacion de los procesos
20   MPI_Comm_size(MPI_COMM_WORLD, &size); // Numero total de procesos
21   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Valor de nuestro identificador
22
23   /**
24     * Paso 2: Comprobar entradas
25     */
26   if (argc != 2) { // Debe haber dos argumentos
27     if (rank == 0) { // El proceso 0 imprime el error
28       cerr << "Sintaxis: " << argv[0] << " <archivo de grafo>" << endl;
```

```
29      }
30      MPI_Finalize();
31      return -1;
32    }
33
34    /**
35      * Paso 3: Crear grafo y obtener numero de vertices
36      */
37    Graph G;
38    int nverts;
39
40    if (rank == 0) { // Solo lo hace un proceso
41      G.lee(argv[1]);
42      #ifdef PRINT_ALL
43        cout << "El grafo de entrada es:" << endl;
44        G.imprime();
45      #endif
46      nverts = G.vertices;
47    }
48
49    /**
50      * Paso 4: Hacer broadcast del numero de vertices a todos los procesos
51      */
52    int raizP, tamaBloque;
53    MPI_Bcast(&nverts, 1, MPI_INT, 0, MPI_COMM_WORLD);
54
55    raizP = sqrt(size);
56    tamaBloque = nverts / raizP;
57
58    /**
59      * Paso 5: Crear comunicadores
60      */
61    int colorHorizontal, colorVertical, rankHorizontal, rankVertical;
62    MPI_Comm commHorizontal, commVertical;
63
64    colorHorizontal = rank / raizP;
65    colorVertical = rank % raizP;
66
67    MPI_Comm_split(MPI_COMM_WORLD, colorHorizontal, rank, &commHorizontal);
68    MPI_Comm_split(MPI_COMM_WORLD, colorVertical, rank, &commVertical);
69
70    MPI_Comm_rank(commHorizontal, &rankHorizontal);
71    MPI_Comm_rank(commVertical, &rankVertical);
72
73    /**
74      * Paso 6: Empaquetar
75      */
76    MPI_Datatype MPI_BLOQUE;
77    int buffEnvio[nverts][nverts]; // Buffer para almacenar los datos empaquetados
78    int filaP, columnaP, comienzo;
79
80    if (rank == 0) {
81      // Se define el tipo de bloque cuadrado
82      MPI_Type_vector(tamaBloque, tamaBloque, nverts, MPI_INT, &MPI_BLOQUE);
83      MPI_Type_commit(&MPI_BLOQUE); // Se crea el nuevo tipo
84      for (int i = 0, posicion = 0; i < size; i++) {
85        // Calculo de la posicion de comienzo de cada submatriz
86        filaP = i / raizP;
87        columnaP = i % raizP;
88        comienzo = columnaP * tamaBloque + filaP * tamaBloque * tamaBloque * raizP;
89        MPI_Pack(G.ptrMatriz() + comienzo, 1, MPI_BLOQUE, buffEnvio,
90                 sizeof(int) * nverts * nverts, &posicion, MPI_COMM_WORLD);
```

```
91      }
92      MPI_Type_free(&MPI_BLOQUE); // Se libera el tipo bloque
93    }
94
95    /**
96      * Paso 7: Distribuir la matriz entre los procesos
97      */
98    int M[tamaBloque][tamaBloque], FilK[tamaBloque], ColK[tamaBloque];
99
100   MPI_Scatter(buffEnvio, sizeof(int) * tamaBloque * tamaBloque,
101               MPI_PACKED, M, tamaBloque * tamaBloque, MPI_INT, 0,
102               MPI_COMM_WORLD);
103
104   /**
105     * Paso 8: Bucle principal del algoritmo
106     */
107   int i, j, k, a, vikj, iGlobal, jGlobal, iIniLocal, iFinLocal,
108         jIniLocal, jFinLocal, kEntreTama, kModuloTama;
109
110   iIniLocal = colorHorizontal * tamaBloque; // Fila ini del proceso (global)
111   iFinLocal = (colorHorizontal + 1) * tamaBloque; // Fila fin del proceso (global)
112   jIniLocal = colorVertical * tamaBloque; // Columna ini del proceso (global)
113   jFinLocal = (colorVertical + 1) * tamaBloque; // Columna fin del proceso (global)
114
115   double t = MPI_Wtime();
116
117   for (k = 0; k < nverts; k++) {
118     kEntreTama = k / tamaBloque;
119     kModuloTama = k % tamaBloque;
120     if (k >= iIniLocal && k < iFinLocal) { // La fila K pertenece al proceso
121       // Copia la fila en el vector FilK
122       copy(M[kModuloTama], M[kModuloTama] + tamaBloque, FilK);
123     }
124     if (k >= jIniLocal && k < jFinLocal) { // La columna K pertenece al proceso
125       for (a = 0; a < tamaBloque; a++) {
126         // Copia la columna en el vector ColK
127         ColK[a] = M[a][kModuloTama];
128       }
129     }
130     MPI_Bcast(FilK, tamaBloque, MPI_INT, kEntreTama, commVertical);
131     MPI_Bcast(ColK, tamaBloque, MPI_INT, kEntreTama, commHorizontal);
132     for (i = 0; i < tamaBloque; i++) { // Recorrer las filas (valores locales)
133       iGlobal = iIniLocal + i; // Convertir la fila a global
134       for (j = 0; j < tamaBloque; j++) {  // Recorrer las columnas (val. locales)
135         jGlobal = jIniLocal + j;
136         // No iterar sobre diagonal
137         if (iGlobal != jGlobal && iGlobal != k && jGlobal != k) {
138           vikj = ColK[i] + FilK[j];
139           vikj = min(vikj, M[i][j]);
140           M[i][j] = vikj;
141         }
142       }
143     }
144   }
145
146   t = MPI_Wtime() - t;
147
148   /**
149     * Paso 9: Recoger resultados en la matriz
150     */
151   MPI_Gather(M, tamaBloque * tamaBloque, MPI_INT, buffEnvio,
152               sizeof(int) * tamaBloque * tamaBloque, MPI_PACKED, 0,
```

```
153                     MPI_COMM_WORLD);
154
155     /**
156       * Paso 10: Desempaquetar
157       */
158     if (rank == 0) {
159       // Se define el tipo de bloque cuadrado
160       MPI_Type_vector(tamaBloque, tamaBloque, nverts, MPI_INT, &MPI_BLOQUE);
161       MPI_Type_commit(&MPI_BLOQUE); // Se crea el nuevo tipo
162       for (int i = 0, posicion = 0; i < size; i++) {
163         // Calculo de la posicion de comienzo de cada submatriz
164         filaP = i / raizP;
165         columnaP = i % raizP;
166         comienzo = columnaP * tamaBloque + filaP * tamaBloque * tamaBloque * raizP;
167         MPI_Unpack(buffEnvio, sizeof(int) * nverts * nverts, &posicion,
168                    G.ptrMatriz() + comienzo, 1, MPI_BLOQUE, MPI_COMM_WORLD);
169       }
170       MPI_Type_free(&MPI_BLOQUE); // Se libera el tipo bloque
171     }
172
173     /**
174       * Paso 11: Finalizar e imprimir resultados
175       */
176     MPI_Finalize();
177
178     if (rank == 0) { // Solo lo hace un proceso
179       #ifdef PRINT_ALL
180         cout << endl << "Solucion:" << endl;
181         G.imprime();
182         cout << "Tiempo gastado = " << t << endl << endl;
183       #else
184         cout << t << endl;
185       #endif
186     }
187 }
```

3. Resultados

N	T_{sec}	T_{1D}	T_{2D}	S_{1D}	S_{2D}
64	0,000396482	0,000868711	0,0008886901	0,4564023343	0,4461414615
128	0,003013494	0,004452289	0,004570981	0,6768415078	0,6592663588
256	0,0234622	0,02117407	0,023623352	1,1080628335	0,9931782755
512	0,2003635	0,10693829	0,11696271	1,8736366553	1,7130545282
1024	1,403187	0,4415758	0,4668597	3,1776809327	3,0055860465

Tabla 3.1: Tiempos y ganancia para P = 4 y N = 64, 128, 256, 512 y 1024

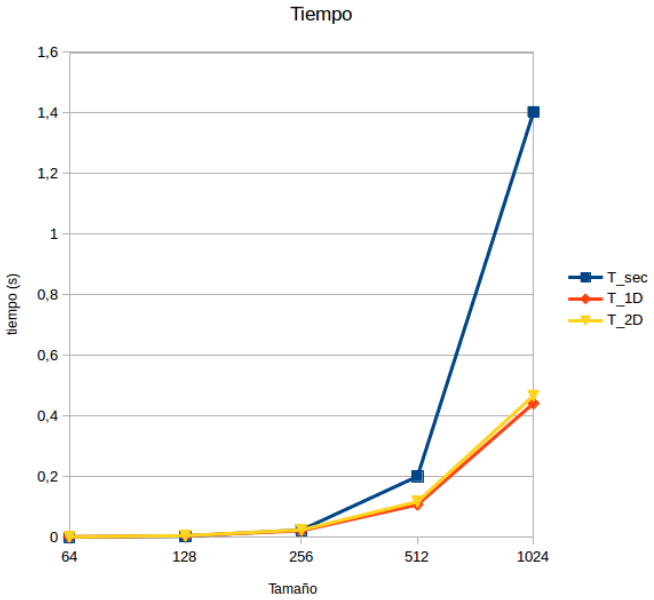

Figura 3.1: Gráfica de tiempo para P = 4 y N = 64, 128, 256, 512 y 1024

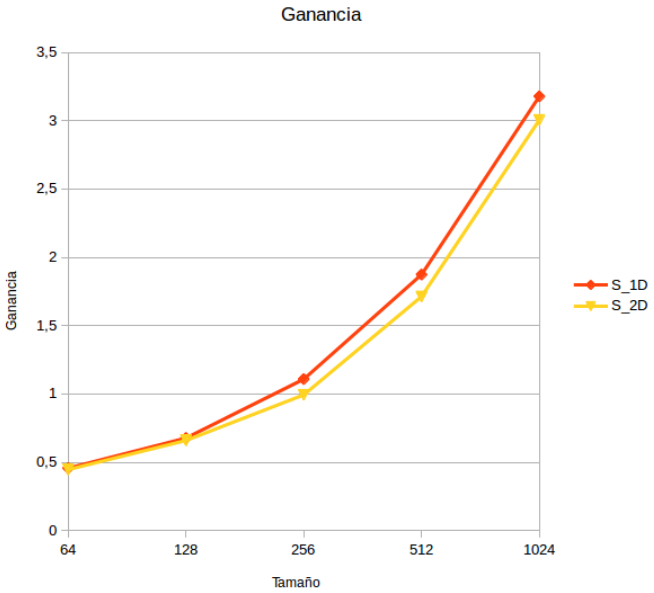

Figura 3.2: Gráfica de ganancia para P = 4 y N = 64, 128, 256, 512 y 1024