ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΜ&ΜΥ
Συστήματα Παράλληλης Επεξεργασίας 1^η Άσκηση
Αχ. έτος 2012-2013

Ομάδα 8^η

Μαυρογιάννης Αλέξανδρος   Α.Μ.: 03109677
Λύρας Γρηγόρης   Α.Μ.: 03109687

12 Νοεμβρίου 2012

Πηγαίος κώδικας

Κοινή βιβλιοθήκη

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : common.h
 * Creation Date : 06-11-2012
 * Last Modified : Mon 12 Nov 2012 10:06:15 AM EET
 * Created By : Greg Liras <gregliras@gmail.com>
 * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/

#ifndef DEBUG_FUNC
#define DEBUG_FUNC

#if main_DEBUG
#define debug(fmt,arg...)     fprintf(stdout, "%s: " fmt, __func__ , ##arg)
#else
#define debug(fmt,arg...)     do { } while(0)
#endif /* main_DEBUG */

#endif /* DEBUG_FUNC */

#ifndef COMMON_H
#define COMMON_H

#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

double *allocate_2d(int N, int M);
double *allocate_2d_with_padding(int N, int M, int max_rank);
double *parse_matrix_2d(FILE *fp, int N, int M, double *A);
void fprint_matrix_2d(FILE *fp, int N, int M, double *A);
void print_matrix_2d(int N, int M, double *A);
double timer(void);
void usage(int argc, char **argv);

#ifdef USE_MPI /* USE_MPI */
void propagate_with_send(void *buffer, int count , MPI_Datatype datatype, \
        int root, MPI_Comm comm);
void propagate_with_flooding(void *buffer, int count , MPI_Datatype datatype, \
        int root, MPI_Comm comm);
#endif /* USE_MPI */

#endif /* COMMON_H */
```

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : common.c
 * Creation Date : 06-11-2012
 * Last Modified : Mon 12 Nov 2012 10:06:02 AM EET
 * Created By : Greg Liras <gregliras@gmail.com>
 * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/

#include "common.h"
#include <sys/time.h>

double *allocate_2d(int N, int M)
{
    double *A;
    A = malloc(N * M * sizeof(double));
    return A;
}

double *allocate_2d_with_padding(int N, int M, int max_rank)
{
    double *A;
    A = allocate_2d(N + max_rank, M);
    return A;
}

double *parse_matrix_2d(FILE *fp, int N, int M, double *A)
{
    int i,j;
    double *p;
    p = A;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            if(!fscanf(fp, "%lf", p++)) {
                return NULL;
            }
        }
    }
    return A;
}

void fprint_matrix_2d(FILE *fp, int N, int M, double *A)
{
```

```c
    int i,j;
    double *p;
    p = A;
    for (j = 0; j < M; j++) {
        fprintf(fp, "=");
    }
    fprintf(fp, "\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            fprintf(fp, "%lf\t", *p++);
        }
        fprintf(fp, "\n");
    }
    for (j = 0; j < M; j++) {
        fprintf(fp, "=");
    }
    fprintf(fp, "\n");
}

void print_matrix_2d(int N, int M, double *A)
{
    fprint_matrix_2d(stdout, N, M, A);
}

double timer(void)
{
    static double seconds = 0;
    static int operation = 0;
    struct timeval tv;
    gettimeofday(&tv, NULL);
    if (operation == 0) {
        seconds = tv.tv_sec + (((double) tv.tv_usec)/1e6);
        operation = 1;
        return 0;
    }
    else {
        operation = 0;
        return tv.tv_sec + (((double) tv.tv_usec)/1e6) - seconds;
    }
}

void usage(int argc, char **argv)
{
    if(argc != 3) {
        printf("Usage: %s <matrix file> <output file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
}

#ifdef USE_MPI /* USE_MPI */
void propagate_with_send(void *buffer, int count, MPI_Datatype datatype, \
        int root, MPI_Comm comm)
{
    int rank;
    int i;
    int max_rank;

    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &max_rank);
    if(rank == root) {
        for(i = 0; i < max_rank; i++) {
            if(i == rank) {
                continue;
            }
            else {
                debug("%d\n", i);
                MPI_Send(buffer, count, datatype, i, root, comm);
            }
        }
    }
    else {
        MPI_Status status;
        MPI_Recv(buffer, count, datatype, root, root, comm, &status);
    }
}

void propagate_with_flooding(void *buffer, int count , MPI_Datatype datatype, \
        int root, MPI_Comm comm)
{
    int rank;
    int max_rank;
    int cur;

    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &max_rank);

    if(root != 0) {
        if(rank == root) {
            MPI_Send(buffer, count, datatype, 0, root, comm);
```

```
132            }
133            if(rank == 0) {
134                MPI_Status status;
135                MPI_Recv(buffer, count, datatype, root, root, comm, &status);
136            }
137        }
138
139        if(rank != 0) {
140            MPI_Status status;
141            MPI_Recv(buffer, count, datatype, (rank-1)/2, root, comm, &status);
142        }
143        cur = 2*rank+1;
144        if(cur < max_rank) {
145            MPI_Send(buffer, count, datatype, cur, root, comm);
146        }
147        if(++cur < max_rank) {
148            MPI_Send(buffer, count, datatype, cur, root, comm);
149        }
150    }
151    #endif /* USE_MPI */
```
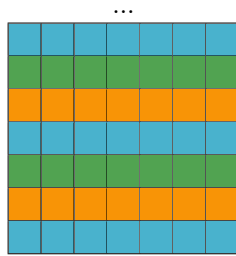
Ζητούμενο 1 Σειριακό Πρόγραμμα

```
1    /* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
2     * File Name : main.c
3     * Creation Date : 30-10-2012
4     * Last Modified : Thu 08 Nov 2012 09:50:55 AM EET
5     * Created By : Greg Liras <gregliras@gmail.com>
6     * Created By : Alex Maurogiannis <nalfemp@gmail.com>
7     _.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/
8
9    #include <stdio.h>
10   #include <stdlib.h>
11
12   #include "common.h"
13
14   int main(int argc, char **argv)
15   {
16       int i,j,k;
17       int N;
18       double *A;
19       double l;
20       double sec;
21
22       FILE *fp = NULL;
23       usage(argc, argv);
24       /*
25        * Allocate me!
26        */
27       fp = fopen(argv[1], "r");
28       if(fp) {
29           if(!fscanf(fp, "%d\n", &N)) {
30               exit(EXIT_FAILURE);
31           }
32       }
33
34       if((A = allocate_2d(N, N)) == NULL) {
35           exit(EXIT_FAILURE);
36       }
37       if(parse_matrix_2d(fp, N, N, A) == NULL) {
38           exit(EXIT_FAILURE);
39       }
40
41
42       sec = timer();
43       for (k = 0; k < N - 1; k++)
44       {
45           for (i = k + 1; i < N; i++)
46           {
47               l = A[i * N + k] / A[k * N + k];
48               for (j = k; j < N; j++)
49               {
50                   A[i * N + j] = A[i * N + j] - l * A[k * N + j];
51               }
52           }
53       }
54       sec = timer();
55       printf("Calc Time: %lf\n", sec);
56
57       fp = fopen(argv[2], "w");
58       fprint_matrix_2d(fp, N, N, A);
59       fclose(fp);
60       free(A);
61
62       return 0;
63   }
```

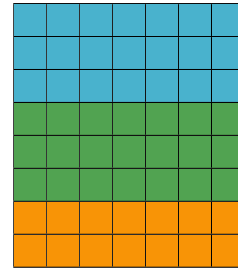## Ζητούμενο 2   Παραλληλισμός Αλγορίθμου

Ο παραλληλισμός του αλγορίθμου εντοπίζεται στο γεγονός ότι υπάρχει ανεξαρτησία του υπολογισμού κατά γραμμές για δεδομένο k. Καθ' όλη την εκτέλεση του προγράμματος κρατάμε σταθερό τον τρόπο διαμοιρασμού των γραμμών του πίνακα, μοιράζοντας σε κάθε επανάληψη την k^{th} γραμμή.

Circular

Continuous

Ζητούμενο 3   Μοντέλο κοινού χώρου διευθύνσεων (OpenMP)

Ζητούμενο 4 Μοντέλο ανταλλαγής μηνυμάτων (MPI)

Ζητούμενο 4.1 Point to Point

Η υλοποίηση **point-to-point** με χρήση flooding για λογαριθμικό propagation και κυκλική κατανομή των γραμμών (*Circular*).

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : main.c
 * Creation Date : 30-10-2012
 * Last Modified : Mon 12 Nov 2012 01:25:21 PM EET
 * Created By : Greg Liras <gregliras@gmail.com>
 * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 _.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

#include "common.h"

#define BLOCK_ROWS 1


void process_rows(int k, int rank, int N, int max_rank, double *A)
{
    /*      performs the calculations for a given set of rows.
     *      In this hybrid version each thread is assigned blocks of
     *      continuous rows in a cyclic manner.
     */
    int i, j, w;
    double l;
    /* For every cyclic repetition of a block */
    for (i = (rank + ((BLOCK_ROWS * max_rank) * (k / (BLOCK_ROWS * max_rank)))); \
            i < N ; i+=(max_rank * BLOCK_ROWS)) {
        if (i > k) {
            /* Calculate each continuous row in the block*/
            for (w = i; w < (i + BLOCK_ROWS) && w < (N * N); w++){
                l = A[(w * N) + k] / A[(k * N) + k];
                for (j = k; j < N; j++) {
                    A[(w * N) + j] = A[(w * N) + j] - l * A[(k* N) + j];
                }
            }
        }
    }
}

int main(int argc, char **argv)
{
    int k;
    int N;
    int rank;
    int max_rank;
    int last_rank;
    double *A = NULL;
    double sec = 0;

    int ret = 0;
    FILE *fp = NULL;
    usage(argc, argv);

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &max_rank);

    if (rank == 0) {
        debug("rank: %d opens file: %s\n", rank, argv[1]);
        fp = fopen(argv[1], "r");
        if(fp) {
            if(!fscanf(fp, "%d\n", &N)) {
                MPI_Abort(MPI_COMM_WORLD, 1);
            }
        }
        else {
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
    propagate_with_flooding(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Everyone allocates the whole table */
    debug("Max rank = %d\n", max_rank);
    if((A = allocate_2d(N, N)) == NULL) {
```

```
82            MPI_Abort(MPI_COMM_WORLD, 1);
83        }
84        /* Root Parses file */
85        if (rank == 0) {
86            if(parse_matrix_2d(fp, N, N, A) == NULL) {
87                MPI_Abort(MPI_COMM_WORLD, 1);
88            }
89            fclose(fp);
90            fp = NULL;
91        }
92        /* And distributes the table */
93        MPI_Barrier(MPI_COMM_WORLD);
94        propagate_with_flooding(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
95
96        last_rank = (N - 1) % max_rank;
97
98        if(rank == 0) {
99            sec = timer();
100        }
101
102        for (k = 0; k < N - 1; k++) {
103            /* The owner of the row for this k broadcasts it*/
104            MPI_Barrier(MPI_COMM_WORLD);
105            propagate_with_flooding(&A[k * N], N, MPI_DOUBLE, \
106                    ((k % (max_rank * BLOCK_ROWS)) / BLOCK_ROWS), MPI_COMM_WORLD);
107            process_rows(k, rank, N, max_rank, A);
108        }
109
110        MPI_Barrier(MPI_COMM_WORLD);
111        if (rank == 0) {
112            sec = timer();
113            printf("Calc Time: %lf\n", sec);
114        }
115        ret = MPI_Finalize();
116
117        if(ret == 0) {
118            debug("%d FINALIZED!!! with code: %d\n", rank, ret);
119        }
120        else {
121            debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
122        }
123
124        /* Last process has table */
125        if (rank == last_rank) {
126            //print_matrix_2d(N, N, A);
127            fp = fopen(argv[2], "w");
128            fprint_matrix_2d(fp, N, N, A);
129            fclose(fp);
130        }
131        free(A);
132
133        return 0;
134 }
```

Η υλοποίηση **point-to-point** με χρήση flooding για λογαριθμικό propagation και συνεχή κατανομή των γραμμών (*Continuous*).

```
1  /* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
2   * File Name : main.c
3   * Creation Date : 30-10-2012
4   * Last Modified : Mon 12 Nov 2012 01:25:05 PM EET
5   * Created By : Greg Liras <gregliras@gmail.com>
6   * Created By : Alex Maurogiannis <nalfemp@gmail.com>
7   _.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/
8
9  #include <mpi.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <signal.h>
13 #include <signal.h>
14 #include <unistd.h>
15 #include <string.h>
16
17 #include "common.h"
18
19
20 int get_bcaster(int *ccounts, int bcaster)
21 {
22     if (ccounts[bcaster]-- > 0 ){
23         return bcaster;
24     } else {
25         return bcaster+1;
26     }
27 }
28
29 void get_displs(int *counts, int max_rank, int *displs)
30 {
31     int j;
```

```
32          displs[0] = 0;
33          for (j = 1; j < max_rank ; j++) {
34              displs[j] = displs[j - 1] + counts[j - 1];
35          }
36      }
37
38      int max(int a, int b)
39      {
40          return a > b ? a : b;
41      }
42
43      void process_rows(int k, int rank, int N, int max_rank, int block_rows, int *displs, double *A)
44      {
45          /*      performs the calculations for a given set of rows.
46           *      In this hybrid version each thread is assigned blocks of
47           *      continuous rows in a cyclic manner.
48           */
49          int j, w;
50          double l;
51          int start = max(displs[rank], k+1);
52          for (w = start; w < (start + block_rows) && w < N; w++){
53              l = A[(w * N) + k] / A[(k * N) + k];
54              for (j = k; j < N; j++) {
55                  A[(w * N) + j] = A[(w * N) + j] - l * A[(k * N) + j];
56              }
57          }
58      }
59
60      /*  distributes the rows in a continuous fashion */
61      void distribute_rows(int max_rank, int N, int *counts)
62      {
63              int j, k;
64              int rows = N;
65
66              /* Initialize counts */
67              for (j = 0; j < max_rank ; j++) {
68                  counts[j] = (rows / max_rank);
69              }
70
71              /* Distribute the indivisible leftover */
72              if (rows / max_rank != 0) {
73                  j = rows % max_rank;
74                  for (k = 0; k < max_rank && j > 0; k++, j--) {
75                          counts[k] += 1;
76                  }
77              } else {
78                  for (k = 0; k < max_rank; k++){
79                      counts[k] = 1;
80                  }
81              }
82
83      }
84
85
86      int main(int argc, char **argv)
87      {
88          int k;
89          int N;
90          int rank;
91          int max_rank;
92          int block_rows;
93          int *counts;
94          int *displs;
95          int *ccounts;
96          int ret = 0;
97          int bcaster = 0;
98          double sec;
99          double *A = NULL;
100         FILE *fp = NULL;
101
102
103         usage(argc, argv);
104
105         MPI_Init(&argc, &argv);
106         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
107         MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
108
109         if (rank == 0) {
110             debug("rank: %d opens file: %s\n", rank, argv[1]);
111             fp = fopen(argv[1], "r");
112             if(fp) {
113                 if(!fscanf(fp, "%d\n", &N)) {
114                     MPI_Abort(MPI_COMM_WORLD, 1);
115                 }
116             }
117             else {
118                 MPI_Abort(MPI_COMM_WORLD, 1);
119             }
120
```

```
121          }
122
123      MPI_Barrier(MPI_COMM_WORLD);
124      propagate_with_flooding(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
125
126      counts = malloc(max_rank * sizeof(int));
127      displs = malloc(max_rank * sizeof(int));
128      ccounts = malloc(max_rank * sizeof(int));
129
130      distribute_rows(max_rank, N, counts);
131      get_displs(counts, max_rank, displs);
132      memcpy(ccounts, counts, max_rank * sizeof(int));
133  #if main_DEBUG
134          printf("CCounts is :\n");
135          for (j = 0; j < max_rank ; j++) {
136              printf("%d\n", ccounts[j]);
137          }
138  #endif
139
140      /* Everybody Allocates the whole table */
141      if((A = allocate_2d(N, N)) == NULL) {
142          MPI_Abort(MPI_COMM_WORLD, 1);
143      }
144      if (rank == 0) {
145          if(parse_matrix_2d(fp, N, N, A) == NULL) {
146              MPI_Abort(MPI_COMM_WORLD, 1);
147          }
148          fclose(fp);
149          fp = NULL;
150      }
151
152      MPI_Barrier(MPI_COMM_WORLD);
153      propagate_with_flooding(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
154
155      /* Start Timing */
156      if(rank == 0) {
157          sec = timer();
158      }
159
160
161      for (k = 0; k < N - 1; k++) {
162          block_rows = counts[rank];
163          bcaster = get_bcaster(ccounts, bcaster);
164
165          debug(" broadcaster is %d\n", bcaster);
166          MPI_Barrier(MPI_COMM_WORLD);
167          propagate_with_flooding(&A[k * N], N, MPI_DOUBLE, bcaster, MPI_COMM_WORLD);
168
169          process_rows(k, rank, N, max_rank, block_rows, displs, A);
170      }
171
172      MPI_Barrier(MPI_COMM_WORLD);
173      if(rank == 0) {
174          sec = timer();
175          printf("Calc Time: %lf\n", sec);
176      }
177      ret = MPI_Finalize();
178
179      if(ret == 0) {
180          debug("%d FINALIZED!!! with code: %d\n", rank, ret);
181      }
182      else {
183          debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
184      }
185
186      if(rank == (max_rank - 1)) {
187          fp = fopen(argv[2], "w");
188          fprint_matrix_2d(fp, N, N, A);
189          fclose(fp);
190      }
191      free(A);
192      free(counts);
193      free(ccounts);
194
195      return 0;
196  }
```

Ζητούμενο 4.2   Collective

Η **collective** υλοποίηση για κυκλική κατανομή των γραμμών (*Circular*).

```
1   /* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
2    * File Name : main.c
3    * Creation Date : 30-10-2012
4    * Last Modified : Mon 12 Nov 2012 01:25:30 PM EET
5    * Created By : Greg Liras <gregliras@gmail.com>
6    * Created By : Alex Maurogiannis <nalfemp@gmail.com>
7    _.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/
```

```
8
9    #include <mpi.h>
10   #include <stdio.h>
11   #include <stdlib.h>
12   #include <signal.h>
13   #include <signal.h>
14   #include <unistd.h>
15   #include <string.h>
16
17   #include "common.h"
18
19   #define BLOCK_ROWS 1
20
21
22   void process_rows(int k, int rank, int N, int max_rank, double *A)
23   {
24       /*      performs the calculations for a given set of rows.
25        *      In this hybrid version each thread is assigned blocks of
26        *      continuous rows in a cyclic manner.
27        */
28       int i, j, w;
29       double l;
30       /* For every cyclic repetition of a block */
31       for (i = (rank + ((BLOCK_ROWS * max_rank) * (k / (BLOCK_ROWS * max_rank)))); i < N ; i+=(max_rank * BLOCK_ROWS)) {
32           if (i > k) {
33               /* Calculate each continuous row in the block*/
34               for (w = i; w < (i + BLOCK_ROWS) && w < (N * N); w++){
35                   l = A[(w * N) + k] / A[(k * N) + k];
36                   for (j = k; j < N; j++) {
37                       A[(w * N) + j] = A[(w * N) + j] - l * A[(k* N) + j];
38                   }
39               }
40           }
41       }
42   }
43
44   int main(int argc, char **argv)
45   {
46       int k;
47       int N;
48       int rank;
49       int max_rank;
50       int last_rank;
51       double *A = NULL;
52       double sec = 0;
53
54       int ret = 0;
55       FILE *fp = NULL;
56       usage(argc, argv);
57
58       MPI_Init(&argc, &argv);
59       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
60       MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
61
62       if (rank == 0) {
63           debug("rank: %d opens file: %s\n", rank, argv[1]);
64           fp = fopen(argv[1], "r");
65           if(fp) {
66               if(!fscanf(fp, "%d\n", &N)) {
67                   MPI_Abort(MPI_COMM_WORLD, 1);
68               }
69           }
70           else {
71               MPI_Abort(MPI_COMM_WORLD, 1);
72           }
73
74       }
75
76       MPI_Barrier(MPI_COMM_WORLD);
77       MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
78
79       /* Everyone allocates the whole table */
80       debug("Max rank = %d\n", max_rank);
81       if((A = allocate_2d(N, N)) == NULL) {
82           MPI_Abort(MPI_COMM_WORLD, 1);
83       }
84       /* Root Parses file */
85       if (rank == 0) {
86           if(parse_matrix_2d(fp, N, N, A) == NULL) {
87               MPI_Abort(MPI_COMM_WORLD, 1);
88           }
89           fclose(fp);
90           fp = NULL;
91       }
92       /* And distributes the table */
93       MPI_Barrier(MPI_COMM_WORLD);
94       MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
95
96       last_rank = (N - 1) % max_rank;
```

```
 97
 98        if(rank == 0) {
 99            sec = timer();
100        }
101
102        for (k = 0; k < N - 1; k++) {
103            /* The owner of the row for this k broadcasts it*/
104            MPI_Barrier(MPI_COMM_WORLD);
105            MPI_Bcast(&A[k * N], N, MPI_DOUBLE, ((k % (max_rank * BLOCK_ROWS)) / BLOCK_ROWS), MPI_COMM_WORLD);
106
107            process_rows(k, rank, N, max_rank, A);
108        }
109
110        MPI_Barrier(MPI_COMM_WORLD);
111        if (rank == 0) {
112            sec = timer();
113            printf("Calc Time: %lf\n", sec);
114        }
115        ret = MPI_Finalize();
116
117        if(ret == 0) {
118            debug("%d FINALIZED!!! with code: %d\n", rank, ret);
119        }
120        else {
121            debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
122        }
123
124        /* Last process has table */
125        if (rank == last_rank) {
126            //print_matrix_2d(N, N, A);
127            fp = fopen(argv[2], "w");
128            fprint_matrix_2d(fp, N, N, A);
129            fclose(fp);
130        }
131        free(A);
132
133        return 0;
134    }
```

Η **collective** υλοποίηση για συνεχή κατανομή των γραμμών (*Continuous*).

```
 1   /* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 2    * File Name : main.c
 3    * Creation Date : 30-10-2012
 4    * Last Modified : Mon 12 Nov 2012 01:25:55 PM EET
 5    * Created By : Greg Liras <gregliras@gmail.com>
 6    * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 7    _.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.*/
 8
 9   #include <mpi.h>
10   #include <stdio.h>
11   #include <stdlib.h>
12   #include <signal.h>
13   #include <signal.h>
14   #include <unistd.h>
15   #include <string.h>
16
17   #include "common.h"
18
19
20   int get_bcaster(int *ccounts, int bcaster)
21   {
22       if (ccounts[bcaster]-- > 0 ){
23           return bcaster;
24       }
25       else {
26           return bcaster+1;
27       }
28   }
29
30   void get_displs(int *counts, int max_rank, int *displs)
31   {
32       int j;
33       displs[0] = 0;
34       for (j = 1; j < max_rank ; j++) {
35           displs[j] = displs[j - 1] + counts[j - 1];
36       }
37   }
38
39   int max(int a, int b)
40   {
41       return a > b ? a : b;
42   }
43
44   void process_rows(int k, int rank, int N, int max_rank, int block_rows, int *displs, double *A)
45   {
46       /*     performs the calculations for a given set of rows.
47        *     In this hybrid version each thread is assigned blocks of
48        *     continuous rows in a cyclic manner.
```

10

```
49          */
50      int j, w;
51      double l;
52      int start = max(displs[rank], k+1);
53      for (w = start; w < (start + block_rows) && w < N; w++){
54          l = A[(w * N) + k] / A[(k * N) + k];
55          for (j = k; j < N; j++) {
56              A[(w * N) + j] = A[(w * N) + j] - l * A[(k * N) + j];
57          }
58      }
59  }
60
61  /*  distributes the rows in a continuous fashion */
62  void distribute_rows(int max_rank, int N, int *counts)
63  {
64      int j, k;
65      int rows = N;
66
67      /* Initialize counts */
68      for (j = 0; j < max_rank ; j++) {
69          counts[j] = (rows / max_rank);
70      }
71
72      /* Distribute the indivisible leftover */
73      if (rows / max_rank != 0) {
74          j = rows % max_rank;
75          for (k = 0; k < max_rank && j > 0; k++, j--) {
76              counts[k] += 1;
77          }
78      }
79      else {
80          for (k = 0; k < max_rank; k++) {
81              counts[k] = 1;
82          }
83      }
84
85  }
86
87
88  int main(int argc, char **argv)
89  {
90      int k;
91      int N;
92      int rank;
93      int max_rank;
94      int block_rows;
95      int *counts;
96      int *displs;
97      int *ccounts;
98      int ret = 0;
99      int bcaster = 0;
100     double sec;
101     double *A = NULL;
102     FILE *fp = NULL;
103
104
105     usage(argc, argv);
106
107     MPI_Init(&argc, &argv);
108     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
109     MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
110
111     if (rank == 0) {
112         debug("rank: %d opens file: %s\n", rank, argv[1]);
113         fp = fopen(argv[1], "r");
114         if(fp) {
115             if(!fscanf(fp, "%d\n", &N)) {
116                 MPI_Abort(MPI_COMM_WORLD, 1);
117             }
118         }
119         else {
120             MPI_Abort(MPI_COMM_WORLD, 1);
121         }
122
123     }
124
125     MPI_Barrier(MPI_COMM_WORLD);
126     MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
127
128     counts = malloc(max_rank * sizeof(int));
129     displs = malloc(max_rank * sizeof(int));
130     ccounts = malloc(max_rank * sizeof(int));
131
132     distribute_rows(max_rank, N, counts);
133     get_displs(counts, max_rank, displs);
134     memcpy(ccounts, counts, max_rank * sizeof(int));
135
136 #if main_DEBUG
137     printf("CCounts is :\n");
```

```
138        for (j = 0; j < max_rank ; j++) {
139            printf("%d\n", ccounts[j]);
140        }
141    #endif
142
143        /* Everybody Allocates the whole table */
144        if((A = allocate_2d(N, N)) == NULL) {
145            MPI_Abort(MPI_COMM_WORLD, 1);
146        }
147        if (rank == 0) {
148            if(parse_matrix_2d(fp, N, N, A) == NULL) {
149                MPI_Abort(MPI_COMM_WORLD, 1);
150            }
151            fclose(fp);
152            fp = NULL;
153        }
154
155        MPI_Barrier(MPI_COMM_WORLD);
156        MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
157
158        /* Start Timing */
159        if(rank == 0) {
160            sec = timer();
161        }
162
163
164        for (k = 0; k < N - 1; k++) {
165            block_rows = counts[rank];
166            bcaster = get_bcaster(ccounts, bcaster);
167
168            MPI_Barrier(MPI_COMM_WORLD);
169            debug(" broadcaster is %d\n", bcaster);
170            MPI_Barrier(MPI_COMM_WORLD);
171            MPI_Bcast(&A[k * N], N, MPI_DOUBLE, bcaster, MPI_COMM_WORLD);
172
173            process_rows(k, rank, N, max_rank, block_rows, displs, A);
174        }
175
176        MPI_Barrier(MPI_COMM_WORLD);
177        if(rank == 0) {
178            sec = timer();
179            printf("Calc Time: %lf\n", sec);
180        }
181        ret = MPI_Finalize();
182
183        if(ret == 0) {
184            debug("%d FINALIZED!!! with code: %d\n", rank, ret);
185        }
186        else {
187            debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
188        }
189
190        if(rank == (max_rank - 1)) {
191            fp = fopen(argv[2], "w");
192            fprint_matrix_2d(fp, N, N, A);
193            fclose(fp);
194        }
195        free(A);
196        free(counts);
197        free(ccounts);
198
199        return 0;
200    }
```