ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

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

Ομάδα 8η

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

11 Νοεμβρίου 2012

Πηγαίος κώδικας

Κοινή βιβλιοθήκη

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
* File Name : common.h
* Creation Date : 06-11-2012
* Last Modified : Sun 11 Nov 2012 06:12:55 PM EET
* Created By : Greg Liras <gregliras@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 : Sun 11 Nov 2012 06:40:49 PM EET
* Created By : Greg Liras <gregliras@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)
{
    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);
        }
        if(rank == 0) {
            MPI_Status status;
            MPI_Recv(buffer, count, datatype, root, root, comm, &status);
        }
    }


    if(rank != 0) {
        MPI_Status status;
        MPI_Recv(buffer, count, datatype, (rank-1)/2, root, comm, &status);
    }
    cur = 2*rank+1;
    if(cur < max_rank) {
        MPI_Send(buffer, count, datatype, cur, root, comm);
    }
    if(++cur < max_rank) {
        MPI_Send(buffer, count, datatype, cur, root, comm);
    }
}
#endif /* USE_MPI */
```

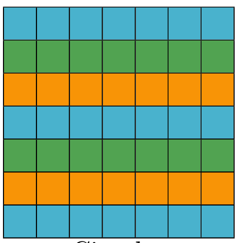
Ζητούμενο 1 Σειριακό Πρόγραμμα

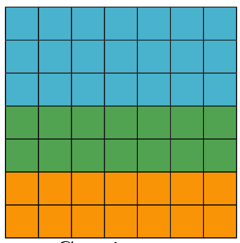
```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
* File Name : main.c
* Creation Date : 30-10-2012
* Last Modified : Thu 08 Nov 2012 09:50:55 AM EET
* Created By : Greg Liras <gregliras@gmail.com>
* Created By : Alex Maurogiannis <nalfemp@gmail.com>
_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.*/

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

#include "common.h"

int main(int argc, char **argv)
{
    int i,j,k;
    int N;
    double *A;
    double l;
    double sec;

    FILE *fp = NULL;
```

```c
    usage(argc, argv);
    /*
     * Allocate me!
     */
    fp = fopen(argv[1], "r");
    if(fp) {
        if(!fscanf(fp, "%d\n", &N)) {
            exit(EXIT_FAILURE);
        }
    }

    if((A = allocate_2d(N, N)) == NULL) {
        exit(EXIT_FAILURE);
    }
    if(parse_matrix_2d(fp, N, N, A) == NULL) {
        exit(EXIT_FAILURE);
    }


    sec = timer();
    for (k = 0; k < N - 1; k++)
    {
        for (i = k + 1; i < N; i++)
        {
            l = A[i * N + k] / A[k * N + k];
            for (j = k; j < N; j++)
            {
                A[i * N + j] = A[i * N + j] - l * A[k * N + j];
            }
        }
    }
    sec = timer();
    printf("Calc Time: %lf\n", sec);

    fp = fopen(argv[2], "w");
    fprint_matrix_2d(fp, N, N, A);
    fclose(fp);
    free(A);

    return 0;
}
```

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

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : main.c
 * Creation Date : 30-10-2012
 * Last Modified : Sun 11 Nov 2012 09:36:48 PM EET
 * Created By : Greg Liras <gregliras@gmail.com>
 * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 _-._-._-._-._-._-._-._-._-._-._-._-._-._-._-._-._-.*/

```

```c
#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_with_padding(N, N, max_rank)) == NULL) {
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    /* Root Parses file */
```

```
    if (rank == 0) {
        if(parse_matrix_2d(fp, N, N, A) == NULL) {
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        fclose(fp);
        fp = NULL;
    }
    /* And distributes the table */
    MPI_Barrier(MPI_COMM_WORLD);
    propagate_with_flooding(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    last_rank = (N - 1) % max_rank;

    if(rank == 0) {
        sec = timer();
    }

    for (k = 0; k < N - 1; k++) {
        /* The owner of the row for this k broadcasts it*/
        MPI_Barrier(MPI_COMM_WORLD);
        propagate_with_flooding(&A[k * N], N, MPI_DOUBLE, ((k % (max_rank * BLOCK_ROWS)) / BLOCK_ROWS), MPI_COMM_WORLD);

        process_rows(k, rank, N, max_rank, A);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        sec = timer();
        printf("Calc Time: %lf\n", sec);
    }
    ret = MPI_Finalize();

    if(ret == 0) {
        debug("%d FINALIZED!!! with code: %d\n", rank, ret);
    }
    else {
        debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
    }

    /* Last process has table */
    if (rank == last_rank) {
        //print_matrix_2d(N, N, A);
        fp = fopen(argv[2], "w");
        fprint_matrix_2d(fp, N, N, A);
        fclose(fp);
    }
    free(A);

    return 0;
}
```

```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : main.c
 * Creation Date : 30-10-2012
 * Last Modified : Sun 11 Nov 2012 09:03:29 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"


int get_bcaster(int *ccounts, int bcaster) {
    if (ccounts[bcaster]-- > 0 ){
        return bcaster;
    } else {
        return bcaster+1;
    }
```

```
}

void get_displs(int *counts, int max_rank, int *displs) {
    int j;
    displs[0] = 0;
    for (j = 1; j < max_rank ; j++) {
        displs[j] = displs[j - 1] + counts[j - 1];
    }
}

int max(int a, int b) {
    return a > b ? a : b;
}

void process_rows(int k, int rank, int N, int max_rank, int block_rows, int *displs, 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 j, w;
    double l;
    int start = max(displs[rank], k+1);
    for (w = start; w < (start + block_rows) && w < 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];
        }
    }
}

/*  distributes the rows in a continuous fashion */
void distribute_rows(int max_rank, int N, int *counts) {
        int j, k;
        int rows = N;

        /* Initialize counts */
        for (j = 0; j < max_rank ; j++) {
            counts[j] = (rows / max_rank);
        }

        /* Distribute the indivisible leftover */
        if (rows / max_rank != 0) {
            j = rows % max_rank;
            for (k = 0; k < max_rank && j > 0; k++, j--) {
                    counts[k] += 1;
            }
        } else {
            for (k = 0; k < max_rank; k++){
                counts[k] = 1;
            }
        }

}


int main(int argc, char **argv)
{
    int i, j, k;
    int N;
    int rank;
    int max_rank;
    int block_rows;
    int *counts;
    int *displs;
    int *ccounts;
    int ret = 0;
    int bcaster = 0;
    double l;
    double sec;
    double *A = NULL;
    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);

    counts = malloc(max_rank * sizeof(int));
    displs = malloc(max_rank * sizeof(int));
    ccounts = malloc(max_rank * sizeof(int));

    distribute_rows(max_rank, N, counts);
    get_displs(counts, max_rank, displs);
    memcpy(ccounts, counts, max_rank * sizeof(int));
#if main_DEBUG
        printf("CCounts is :\n");
        for (j = 0; j < max_rank ; j++) {
            printf("%d\n", ccounts[j]);
        }
#endif

    /* Everybody Allocates the whole table */
    if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    if (rank == 0) {
        if(parse_matrix_2d(fp, N, N, A) == NULL) {
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        fclose(fp);
        fp = NULL;
    }

    MPI_Barrier(MPI_COMM_WORLD);
    propagate_with_flooding(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* Start Timing */
    if(rank == 0) {
        sec = timer();
    }


    for (k = 0; k < N - 1; k++) {
        block_rows = counts[rank];
        bcaster = get_bcaster(ccounts, bcaster);

        debug(" broadcaster is %d\n", bcaster);
        MPI_Barrier(MPI_COMM_WORLD);
        propagate_with_flooding(&A[k * N], N, MPI_DOUBLE, bcaster, MPI_COMM_WORLD);

        process_rows(k, rank, N, max_rank, block_rows, displs, A);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    if(rank == 0) {
        sec = timer();
        printf("Calc Time: %lf\n", sec);
    }
    ret = MPI_Finalize();

    if(ret == 0) {
```

```
        debug("%d FINALIZED!!! with code: %d\n", rank, ret);
    }
    else {
        debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
    }

    if(rank == (max_rank - 1)) {
        fp = fopen(argv[2], "w");
        fprint_matrix_2d(fp, N, N, A);
        fclose(fp);
    }
    free(A);
    free(counts);
    free(ccounts);

    return 0;
}
```

Ζητούμενο 4.2 Collective

```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : main.c
 * Creation Date : 30-10-2012
 * Last Modified : Sun 11 Nov 2012 09:36:48 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);
```

```
56
57      MPI_Init(&argc, &argv);
58      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
59      MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
60
61      if (rank == 0) {
62          debug("rank: %d opens file: %s\n", rank, argv[1]);
63          fp = fopen(argv[1], "r");
64          if(fp) {
65              if(!fscanf(fp, "%d\n", &N)) {
66                  MPI_Abort(MPI_COMM_WORLD, 1);
67              }
68          }
69          else {
70              MPI_Abort(MPI_COMM_WORLD, 1);
71          }
72
73      }
74
75      MPI_Barrier(MPI_COMM_WORLD);
76      MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
77
78      /* Everyone allocates the whole table */
79      debug("Max rank = %d\n", max_rank);
80      if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
81          MPI_Abort(MPI_COMM_WORLD, 1);
82      }
83      /* Root Parses file */
84      if (rank == 0) {
85          if(parse_matrix_2d(fp, N, N, A) == NULL) {
86              MPI_Abort(MPI_COMM_WORLD, 1);
87          }
88          fclose(fp);
89          fp = NULL;
90      }
91      /* And distributes the table */
92      MPI_Barrier(MPI_COMM_WORLD);
93      MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
94
95      last_rank = (N - 1) % max_rank;
96
97      if(rank == 0) {
98          sec = timer();
99      }
100
101     for (k = 0; k < N - 1; k++) {
102         /* The owner of the row for this k broadcasts it*/
103         MPI_Barrier(MPI_COMM_WORLD);
104         MPI_Bcast(&A[k * N], N, MPI_DOUBLE, ((k % (max_rank * BLOCK_ROWS)) / BLOCK_ROWS), MPI_COMM_WORLD);
105
106         process_rows(k, rank, N, max_rank, A);
107     }
108
109     MPI_Barrier(MPI_COMM_WORLD);
110     if (rank == 0) {
111         sec = timer();
112         printf("Calc Time: %lf\n", sec);
113     }
114     ret = MPI_Finalize();
115
116     if(ret == 0) {
117         debug("%d FINALIZED!!! with code: %d\n", rank, ret);
118     }
119     else {
120         debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
121     }
122
123     /* Last process has table */
124     if (rank == last_rank) {
125         //print_matrix_2d(N, N, A);
126         fp = fopen(argv[2], "w");
127         fprint_matrix_2d(fp, N, N, A);
128         fclose(fp);
129     }
130     free(A);
```

```
131
132     return 0;
133 }
```

```
1  /* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
2   * File Name : main.c
3   * Creation Date : 30-10-2012
4   * Last Modified : Sun 11 Nov 2012 09:03:29 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     if (ccounts[bcaster]-- > 0 ){
22         return bcaster;
23     } else {
24         return bcaster+1;
25     }
26 }
27
28 void get_displs(int *counts, int max_rank, int *displs) {
29     int j;
30     displs[0] = 0;
31     for (j = 1; j < max_rank ; j++) {
32         displs[j] = displs[j - 1] + counts[j - 1];
33     }
34 }
35
36 int max(int a, int b) {
37     return a > b ? a : b;
38 }
39
40 void process_rows(int k, int rank, int N, int max_rank, int block_rows, int *displs, double *A){
41     /*      performs the calculations for a given set of rows.
42      *      In this hybrid version each thread is assigned blocks of
43      *      continuous rows in a cyclic manner.
44      */
45     int j, w;
46     double l;
47     int start = max(displs[rank], k+1);
48     for (w = start; w < (start + block_rows) && w < N; w++){
49         l = A[(w * N) + k] / A[(k * N) + k];
50         for (j = k; j < N; j++) {
51             A[(w * N) + j] = A[(w * N) + j] - l * A[(k * N) + j];
52         }
53     }
54 }
55
56 /*  distributes the rows in a continuous fashion */
57 void distribute_rows(int max_rank, int N, int *counts) {
58         int j, k;
59         int rows = N;
60
61         /* Initialize counts */
62         for (j = 0; j < max_rank ; j++) {
63             counts[j] = (rows / max_rank);
64         }
65
66         /* Distribute the indivisible leftover */
67         if (rows / max_rank != 0) {
68             j = rows % max_rank;
69             for (k = 0; k < max_rank && j > 0; k++, j--) {
70                     counts[k] += 1;
71             }
72         } else {
```

```
            for (k = 0; k < max_rank; k++){
                counts[k] = 1;
            }
        }

}


int main(int argc, char **argv)
{
    int i, j, k;
    int N;
    int rank;
    int max_rank;
    int block_rows;
    int *counts;
    int *displs;
    int *ccounts;
    int ret = 0;
    int bcaster = 0;
    double l;
    double sec;
    double *A = NULL;
    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);
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    counts = malloc(max_rank * sizeof(int));
    displs = malloc(max_rank * sizeof(int));
    ccounts = malloc(max_rank * sizeof(int));

    distribute_rows(max_rank, N, counts);
    get_displs(counts, max_rank, displs);
    memcpy(ccounts, counts, max_rank * sizeof(int));
#if main_DEBUG
        printf("CCounts is :\n");
        for (j = 0; j < max_rank ; j++) {
            printf("%d\n", ccounts[j]);
        }
#endif

    /* Everybody Allocates the whole table */
    if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    if (rank == 0) {
        if(parse_matrix_2d(fp, N, N, A) == NULL) {
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        fclose(fp);
        fp = NULL;
    }

```

```
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* Start Timing */
    if(rank == 0) {
        sec = timer();
    }


    for (k = 0; k < N - 1; k++) {
        block_rows = counts[rank];
        bcaster = get_bcaster(ccounts, bcaster);

        MPI_Barrier(MPI_COMM_WORLD);
        debug(" broadcaster is %d\n", bcaster);
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&A[k * N], N, MPI_DOUBLE, bcaster, MPI_COMM_WORLD);

        process_rows(k, rank, N, max_rank, block_rows, displs, A);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    if(rank == 0) {
        sec = timer();
        printf("Calc Time: %lf\n", sec);
    }
    ret = MPI_Finalize();

    if(ret == 0) {
        debug("%d FINALIZED!!! with code: %d\n", rank, ret);
    }
    else {
        debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
    }

    if(rank == (max_rank - 1)) {
        fp = fopen(argv[2], "w");
        fprint_matrix_2d(fp, N, N, A);
        fclose(fp);
    }
    free(A);
    free(counts);
    free(ccounts);

    return 0;
}
```