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

12 Νοεμβρίου 2012

Πηγαίος κώδικας

Κοινή βιβλιοθήκη

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : common.h
 * Creation Date : 06-11-2012
 * Last Modified : Mon 12 Nov 2012 09:04:50 PM 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>

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 */
#include <mpi.h>
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)
{
```

```
    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);
```

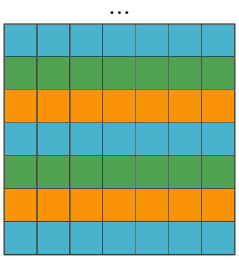
```
        }
        if(rank == 0) {
            MPI_Status status;
            MPI_Recv(buffer, count, datatype, root, root, comm, &status);
        }
    }

    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

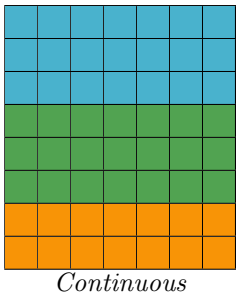
```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
* File Name : main.c
* Creation Date : 30-10-2012
* Last Modified : Thu 08 Nov 2012 09:50:55 AM EET
* Created By : Greg Liras <gregliras@gmail.com>
* Created By : Alex Maurogiannis <nalfemp@gmail.com>
_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-.*/

#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;
    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)

Η υλοποίηση μοντέλου κοινού χώρου διευθύνσεων βασίζεται στην δομή pragma omp for, με χρήση private μεταβλητών divisor και A2 για κάθε thread ώστε να αποφεύγονται όσο γίνεται οι προσβάσεις στην κοινή μνήμη. Επιπλέον, έχουν πραγματοποιηθεί βελτιστοποιήσεις μέσω flags του gcc, όπως φαίνεται στο Makefile.

```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
* File Name : main.c
* Creation Date : 30-10-2012
* Last Modified : Mon 12 Nov 2012 08:54:16 PM EET
* Created By : Greg Liras <gregliras@gmail.com>
* Created By : Alex Maurogiannis <nalfemp@gmail.com>
_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-.*/

#include <stdio.h>
#include <stdlib.h>
#include <omp.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;
    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);
    }


    int chunk = N/omp_get_max_threads();
    double divisor;
    double *A2;
    chunk = 1;

    sec = timer();

    for (k = 0; k < N - 1; k++)
    {
#pragma omp parallel private(divisor)
        {
            divisor = A[k * N + k];
#pragma omp for schedule(static, chunk) private(l,j, A2)
            for (i = k + 1; i < N; i++)
```

```
            {
                A2 = &A[i * N];

                l = A2[k] / divisor;
                for (j = k; j < N; j++)
                {
                    A2[j] = A2[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;
}
```

```
TARGET = main
CC = gcc
CCFLAGS +=
OPTCFLAGS+= -march=native -O3 -fexpensive-optimizations -funroll-loops \
         -fmove-loop-invariants -fprefetch-loop-arrays -ftree-loop-optimize \
         -ftree-vect-loop-version -ftree-vectorize
LDFLAGS += -fopenmp
OPT = 2

ifndef DEBUG
       DEBUG = n
endif

ifeq ($(DEBUG),y)
        CCFLAGS += -D$(TARGET)_DEBUG=1
        CCFLAGS += -g -O0 -Werror -Wall -Wextra -Wuninitialized
        LDFLAGS += -lefence
else
        CCFLAGS += -D$(TARGET)_DEBUG=0
        CCFLAGS += -Werror -Wall
        CCFLAGS += $(OPTCFLAGS)
endif

CCFILES += $(wildcard *.c)
OBJ +=  $(patsubst %.c,%.o,$(CCFILES))
DEPENDS += $(wildcard *.h)


all: $(TARGET).exec

$(TARGET).exec: $(OBJ) $(DEPENDS)
        $(CC) $(LDFLAGS) $(OBJ) -o $(TARGET).exec

%.o:        %.c
        $(CC) -c $(LDFLAGS) $(CCFLAGS) $< -o $@


.PHONY:        clean all indent tags
clean:
        rm -f $(OBJ) $(TARGET)
indent:
        astyle --style=linux $(CCFILES)
tags:
        ctags -R *
```

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) {
```

```
        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;
}
```

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

```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : main.c
 * Creation Date : 30-10-2012
 * Last Modified : Mon 12 Nov 2012 01:34:38 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.
     */
    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 k;
    int N;
    int rank;
    int max_rank;
    int block_rows;
    int *counts;
    int *displs;
    int *ccounts;
    int ret = 0;
    int bcaster = 0;
    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(N, N)) == 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);

    block_rows = counts[rank];

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

    for (k = 0; k < N - 1; k++) {
        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);
    free(displs);

    return 0;
}
```

Ζητούμενο 4.2 Collective

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

```
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : main.c
 * Creation Date : 30-10-2012
 * Last Modified : Mon 12 Nov 2012 01:25:30 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);
    MPI_Bcast(&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) {
        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);
    MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    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.
```

```
     */
    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 k;
    int N;
    int rank;
    int max_rank;
    int block_rows;
    int *counts;
    int *displs;
    int *ccounts;
    int ret = 0;
    int bcaster = 0;
    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(N, N)) == 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;
}
```