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

8 Ιανουαρίου 2013

Πηγαίος κώδικας

Κοινή βιβλιοθήκη

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : common.h
 * Creation Date : 06-11-2012
 * Last Modified : Wed 12 Dec 2012 08:37:45 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


/* Operation Mode */
enum OPMODE { CONTINUOUS, CYCLIC, OPMODE_SIZE };
typedef enum OPMODE OPMODE;

#define MIN(a,b) ((a) < (b)) ? (a) : (b)
#define MAX(a,b)  ((a) > (b)) ? (a) : (b)

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

struct time_struct {
    struct timeval latest_timestamp;
    struct timeval current_duration;
};

typedef struct time_struct time_struct;

void time_struct_init(time_struct *ts);
void time_struct_set_timestamp(time_struct *ts);
void time_struct_add_timestamp(time_struct *ts);
double get_seconds(time_struct *ts);

struct Matrix {
    int N;
    double *A;
};

typedef struct Matrix Matrix;

Matrix *get_matrix(char *filename, int max_rank, OPMODE operation);
double **appoint_2D(double *A, int N, int M);
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);
void *  get_propagation(int argc, char **argv);

void upper_triangularize(int N, double **Ap2D);

#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);
void gather_to_root_cyclic(double **Ap2D, int max_rank, int rank, int root, double **A2D, int N, int M);
void get_counts(int max_rank, int N, int *counts);
void get_displs(int *counts, int max_rank, int *displs);
#endif /* USE_MPI */

#endif /* COMMON_H */
```

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
 * File Name : common.c
 * Creation Date : 06-11-2012
 * Last Modified : Wed 12 Dec 2012 08:51:50 PM EET
 * Created By : Greg Liras <gregliras@gmail.com>
 * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 _._._._._._._._._._._._._._._._._._._._._.*/

#include "common.h"
```

```c
#include <sys/time.h>
#include <string.h>




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

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

static double *parse_matrix_2d_cyclic(FILE *fp, unsigned int N, unsigned int M, double *A, int max_rank)
{
    int i,j;
    double *p;
    int workload = N / max_rank + 1;
    int remainder = N % max_rank;
    double **A2D = appoint_2D(A, N + max_rank, M);

    for(i = 0; i < workload - 1; i++) {
        for(j = 0; j < max_rank; j++) {
            p = A2D[j*workload + i];

            if(fread(p, sizeof(double), M, fp) != M) {
                return NULL;
            }
        }
    }



    /* this loop reads any remaining data from the file */
    for(i = 1; i <= remainder; i++) {
        p = A2D[i*workload] - M;
        if(fread(p, sizeof(double), M, fp) != M) {
            return NULL;
        }
    }

    /* this loop memsets the final line of the bottom parts */
    for(i = max_rank - remainder + 1; i < max_rank; i++) {
        p = A2D[i*workload] - M;
        memset(p, 0, M*sizeof(double));
    }


    free(A2D);
    return A;
}

static double *parse_matrix_2d(FILE *fp, int N, int M, double *A, int max_rank, OPMODE operation)
{
    switch(operation) {
        case CONTINUOUS:
            return parse_matrix_2d_cyclic(fp, N, M, A, 1);
        case CYCLIC:
            return parse_matrix_2d_cyclic(fp, N, M, A, max_rank);
        default:
            return NULL;
    }
}

/* Turns a 2D matrix to upper triangular */
void upper_triangularize(int N, double **Ap2D)

{
    int i,j;
    for (i=1; i < N; i++) {
        for (j=0; j < i; j++) {
            Ap2D[i][j] = 0;
        }
    }
}
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);
}


/* Initialize ts to zero */
void time_struct_init(time_struct *ts)
{
    ts->latest_timestamp.tv_sec = 0;
    ts->latest_timestamp.tv_usec = 0;
    ts->current_duration.tv_sec = 0;
    ts->current_duration.tv_usec = 0;
}

/* Set ts timestamp to current time */
void time_struct_set_timestamp(time_struct *ts)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    ts->latest_timestamp.tv_sec = tv.tv_sec;
    ts->latest_timestamp.tv_usec = tv.tv_usec;
}

/* Set ts timestamp to current time and add the diff to current_duration */
void time_struct_add_timestamp(time_struct *ts)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);

    ts->current_duration.tv_sec += tv.tv_sec - ts->latest_timestamp.tv_sec;
    ts->current_duration.tv_usec += tv.tv_usec - ts->latest_timestamp.tv_usec;

    ts->latest_timestamp.tv_sec = tv.tv_sec;
    ts->latest_timestamp.tv_usec = tv.tv_usec;
}

double get_seconds(time_struct *ts)
{
    return ts->current_duration.tv_sec + (((double) ts->current_duration.tv_usec)/1e6);
}



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)
{
#ifdef USE_MPI /* USE_MPI */
    if(argc > 4 || argc < 3) {
        printf("Usage: %s <matrix file> <output file> [propagation mode: default=0 (ptp)]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
#else
    if(argc != 3) {
        printf("Usage: %s <matrix file> <output file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
#endif
}

Matrix *get_matrix(char *filename, int max_rank, OPMODE operation)
```

```c
{
    FILE *fp;
    double *A;
    int N;
    Matrix *mat;

    if(NULL == (mat = malloc(sizeof(struct Matrix)))) {
        debug("Could not allocate empty Matrix\n");
        exit(EXIT_FAILURE);
    }
    fp = fopen(filename, "rb");
    if(fp) {
        if(fread(&N, sizeof(int), 1, fp) != 1) {
            debug("Could not read N from file\n");
            exit(EXIT_FAILURE);
        }
    }
    if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
        debug("Could not allocate enough contiguous memory\n");
        exit(EXIT_FAILURE);
    }
    if(parse_matrix_2d(fp, N, N, A, max_rank, operation) == NULL) {
        debug("Could not parse matrix\n");
        exit(EXIT_FAILURE);
    }
    fclose(fp);
    mat->N = N;
    mat->A = A;

    return mat;
}

double **appoint_2D(double *A, int N, int M)
{
    int i;
    double **A2D = (double **) malloc(N*sizeof(double *));
    /* sanity check */
    if(NULL == A2D) {
        return NULL;
    }
    for(i = 0; i < N; i++) {
        A2D[i] = &A[i*M];
    }
    return A2D;
}

#ifdef USE_MPI /* USE_MPI */

/* get operation mode from the third argument.
 * 1 for continuous, 0 for ptp */
void * get_propagation(int argc, char **argv)
{
    if (argc > 3) {
        if (argv[3][0] == '1') {
            return &MPI_Bcast;
        }
    }
    return &propagate_with_flooding;
}

void propagate_with_send(void *buffer, int count, MPI_Datatype datatype, \
        int root, MPI_Comm comm)
{
    int rank;
    int i;
    int max_rank;
    MPI_Status status;

    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_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);
    MPI_Status status;

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

/* Returns the displacements table in rows */
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];
    }
}

/*  distributes the rows in a continuous fashion */
void get_counts(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;
        }
    }
}


/* Gather everything to root */
void gather_to_root_cyclic(double **Ap2D, int max_rank, int rank, int root, double **A2D, int N, int M)
{
    int i;
    int bcaster;
    int current_row;
    MPI_Status status;
    for(i = 0; i < N; i++) {
        bcaster = i % max_rank;
        current_row = i / max_rank;
        MPI_Barrier(MPI_COMM_WORLD);
        if(rank == bcaster) {
            if(bcaster == root) {
                memcpy(A2D[i], Ap2D[current_row], M*sizeof(double));
            }
            else {
                MPI_Send(Ap2D[current_row], M, MPI_DOUBLE, 0, i, MPI_COMM_WORLD);
            }
        }
        else if (rank == root) {
            MPI_Recv(A2D[i], M, MPI_DOUBLE, bcaster, i, MPI_COMM_WORLD, &status);
        }
    }
```

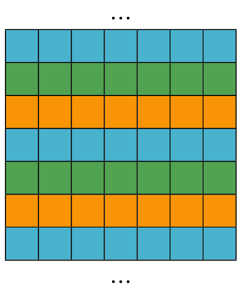
```c
}


#endif /* USE_MPI */
```

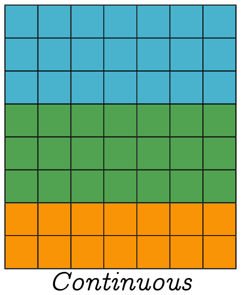
Ζητούμενο 1   Σειριακό Πρόγραμμα

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
* File Name : main.c
* Creation Date : 30-10-2012
* Last Modified : Thu 29 Nov 2012 03:19:28 PM 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!
     */
    Matrix *mat = get_matrix(argv[1],0, CONTINUOUS);
    N = mat->N;
    A = mat->A;
    fp = fopen(argv[1], "rb");


    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.

```c
/* -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
* File Name : main.c
* Creation Date : 30-10-2012
* Last Modified : Thu 20 Dec 2012 01:27:37 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 **A2D;
    int flag = 1;

    double l;
    double sec;

    FILE *fp = NULL;
    usage(argc, argv);
    /*
     * Allocate me!
     */

    Matrix *mat = get_matrix(argv[1],0, CONTINUOUS);
    N = mat->N;
    A = mat->A;
    A2D = appoint_2D(A, N, N);



    int chunk = N/omp_get_max_threads();
    double *Ai;
    double *Ak;
    chunk = 1;

    sec = timer();

    for (k = 0; k < N - 1; k++)
    {
#pragma omp parallel private(Ak)
        {
            if(flag)
            {
                printf("%d %d\n", omp_get_num_threads(), omp_get_max_threads());
                flag=0;
            }
            Ak = A2D[k];
#pragma omp for schedule(static, chunk) private(l,j, Ai)
            for (i = k + 1; i < N; i++)
            {
                Ai = A2D[i];

                l = Ai[k] / Ak[k];
                for (j = k; j < N; j++)
                {
                    Ai[j] = Ai[j] - l * Ak[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 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*).

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

Ζητούμενο 4.2 Collective

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

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