# TP n° 4 – éléments de réponse

```c
/*
 * ENSICAEN
 * 6 Boulevard Maréchal Juin
 * F-14050 Caen Cedex
 *
 * Unix System Programming
 * Chapter "Threads"
 *
 * 2016 Alain Lebret (alain.lebret@ensicaen.fr)
 */
#ifndef MATRIX_H
#define MATRIX_H

#define SIZE 2000
#define NB_THREADS 4
#define PRINT 0

/** A structure to manipulate matrix. */
typedef struct matrix {
    unsigned int rows;    /*!< Number of rows. */
    unsigned int columns; /*!< Number of columns. */
    double **matrix;      /*!< Data. */
} matrix_t;

typedef struct args_t {
    pthread_t t_id;
    int row_per_thread;
    int remainder;
    unsigned int i_start;
    unsigned int i_stop;
    matrix_t *m1;
    matrix_t *m2;
    matrix_t *ans;
} args_t;

/* Multi thread functions */
void multi_thread(matrix_t *m1, matrix_t *m2);
void *product_matrix_thread(void *args);
```

```c
/* Mono thread functions */
void mono_thread(matrix_t *m1, matrix_t *m2);
matrix_t *product_matrix(matrix_t *m1, matrix_t *m2);
double product_case(int r, int c, matrix_t *m1, matrix_t *m2);

/* Utils matrix functions */
void print_matrix(matrix_t *m);
matrix_t *init_matrix(unsigned int rows, unsigned int columns, int fill);
void fill_matrix(matrix_t *m);
double **alloc_matrix(unsigned int rows, unsigned int columns);
void free_matrix(double **matrix);

#endif /* MATRIX_H */
```

```c
/*
 * ENSICAEN
 * 6 Boulevard Maréchal Juin
 * F-14050 Caen Cedex
 *
 * Unix System Programming
 * Chapter "Threads"
 *
 * 2016 Alain Lebret (alain.lebret@ensicaen.fr)
 */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include "matrix.h"

/**
 * @file matrix_product.c
 *
 * Functions to create, to destroy and to multiplply matrix using threads or
 * not.
 */

/* Multi thread functions */
void multi_thread(matrix_t *m1, matrix_t *m2)
```

```
{
    int remainder;
    int each_thread;
    int i;
    clock_t c_before;
    clock_t c_after;
    time_t t_before;
    time_t t_after;
    pthread_t *t;
    args_t *args;
    matrix_t *answer;

    t = NULL;
    args = NULL;
    answer = NULL;

    t = (pthread_t *) malloc(sizeof(pthread_t) * NB_THREADS);
    args = (args_t *) malloc(sizeof(args_t) * NB_THREADS);
    answer = init_matrix(SIZE, SIZE, 0);

    remainder = SIZE % NB_THREADS;
    each_thread = (int) SIZE / NB_THREADS;

    /* Argument initialization */
    for (i = 0; i < NB_THREADS; ++i) {

        args[i].t_id = i;
        args[i].row_per_thread = each_thread;

        if (i == NB_THREADS - 1) {
            args[i].remainder = remainder;
        } else {
            args[i].remainder = 0;
        }

        args[i].m1 = m1;
        args[i].m2 = m2;

        args[i].ans = answer;

        args[i].i_start = i * each_thread;
        args[i].i_stop = args[i].i_start + each_thread + remainder;
        printf("Thread: %d, row_per_thread: %d, remainder: %d, i_start %d, i_stop
↪   %d\n",
                args[i].t_id, args[i].row_per_thread, args[i].remainder,
                args[i].i_start, args[i].i_stop - 1);
    }

    /* Threaded creation */
    c_before = clock();
    t_before = time(NULL);
    for (i = 0; i < NB_THREADS; ++i) {
        pthread_create(&t[i], NULL, product_matrix_thread, &args[i]);
    }

    /* Wait for threads to end */
    for (i = 0; i < NB_THREADS; ++i) {
        pthread_join(t[i], NULL);
    }
    t_after = time(NULL);
    c_after = clock();

    if (PRINT) {
        printf("\nProduct :\n");
        print_matrix(answer);
    }

    printf("\nClock_t -> %5.3f ticks (%f seconds)\n",
             (float) (c_after - c_before),
             (double) (c_after - c_before) / CLOCKS_PER_SEC);
    printf("Time_t  -> %5.3f seconds\n", difftime(t_after, t_before));

    free(t);
    free(args);
    free_matrix(answer->matrix);
    free(answer);
}

void *product_matrix_thread(void *args)
{
    unsigned int i;
    unsigned int j;
    args_t *arguments;
```

```c
    arguments = (args_t *) args;

    for (i = arguments->i_start; i < arguments->i_stop; ++i) {
        for (j = 0; j < arguments->ans->rows; ++j) {
            arguments->ans->matrix[j][i] = product_case(j, i,
               arguments->m1, arguments->m2);
        }
    }

    return NULL;
}

/* Mono thread functions */
void mono_thread(matrix_t *m1, matrix_t *m2)
{
    matrix_t *m3;
    clock_t c_before;
    clock_t c_after;
    time_t t_before;
    time_t t_after;

    c_before = clock();
    t_before = time(NULL);
    m3 = product_matrix(m1, m2);
    t_after = time(NULL);
    c_after = clock();

    if (PRINT) {
        printf("\nProduct:\n");
        print_matrix(m3);
    }

    printf("\nClock_t -> %5.3f ticks (%f seconds)\n",
             (float) (c_after - c_before),
             (double) (c_after - c_before) / CLOCKS_PER_SEC);
    printf("Time_t  -> %5.3f seconds\n", difftime(t_after, t_before));

    free_matrix(m3->matrix);
    free(m3);
}
```

```c
matrix_t *product_matrix(matrix_t *m1, matrix_t *m2)
{
    unsigned int i;
    unsigned int j;
    matrix_t *result;

    if (m1->columns != m2->rows) {
        printf("Dimensions Error\n");
        return NULL;
    }

    result = init_matrix(m1->rows, m2->columns, 0);

    for (i = 0; i < result->columns; ++i) {
        for (j = 0; j < result->rows; ++j) {
            result->matrix[j][i] = product_case(j, i, m1, m2);
        }
    }

    return result;
}

double product_case(int r, int c, matrix_t *m1, matrix_t *m2)
{
    unsigned int i;
    double tmp;

    tmp = 0;

    for (i = 0; i < m1->columns; ++i) {
        tmp += m1->matrix[r][i] * m2->matrix[i][c];
    }

    return tmp;
}

/* Utils matrix functions */

/**
 * Displays the content of the given matrix.
 * @param m A pointer on the matrix.
 */
```

```c
void print_matrix(matrix_t *m)
{
    unsigned int i;
    unsigned int j;

    for (i = 0; i < m->rows; ++i) {
        for (j = 0; j < m->columns; ++j) {
            printf("%04.2f ", m->matrix[i][j]);
        }
        printf("\n");
    }
}

/**
 * @brief Creates and initializes a matrix structure.
 *
 * @param rows The number of rows of the matrix.
 * @param columns The number of columns of the matrix.
 * @param fill The matrix is filled with random values if fill = 1, 0 elsewhere.
 *
 * @return A pointer on the initialized matrix.
 */
matrix_t *init_matrix(unsigned int rows, unsigned int columns, int fill)
{
    matrix_t *m;

    m = (matrix_t *) malloc(sizeof(matrix_t));
    m->rows = rows;
    m->columns = columns;
    m->matrix = alloc_matrix(rows, columns);

    if (fill) {
        fill_matrix(m);
    }

    return m;
}

/**
 * @brief Fills a matrix with a random value between 10 and 24.
 *
 * @param m The matrix to fill in.
 */
void fill_matrix(matrix_t *m)
{
    unsigned int i;
    unsigned int j;

    for (i = 0; i < m->rows; ++i) {
        for (j = 0; j < m->columns; ++j) {
            m->matrix[i][j] = (double) (rand() % 15) + 10;
        }
    }
}

/**
 * @brief Allocates a matrix and returns it.
 *
 * @param rows The number of rows of the matrix.
 * @param columns The number of columns of the matrix.
 */
double **alloc_matrix(unsigned int rows, unsigned int columns)
{
    double **matrix;
    unsigned int row;

    matrix = (double **) calloc(rows, sizeof(double *));
    if (!matrix) {
        return NULL;
    }

    matrix[0] = (double *) calloc(rows * columns, sizeof(double));
    if (!matrix[0]) {
        free(matrix);
        return NULL;
    }

    for (row = 1; row < rows; ++row) {
        matrix[row] = matrix[row - 1] + columns;
    }

    return matrix;
}
```

```c
/**
 * @brief Frees memory fot the given matrix.
 */
void free_matrix(double **matrix)
{
    free(matrix[0]);
    free(matrix);
}
```