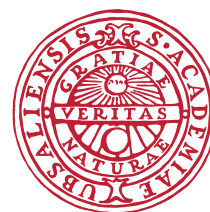


Yohann Teston  
yohann.teston@free.fr  
Matthieu Maury  
mayeu.tik@gmail.com

881003-P792

860928-P210



UPPSALA  
UNIVERSITET

*Department of Information Technology*

# Programming of parallel computers

## Lab 2 - Pthreads

May 7, 2010

# Contents

<b>1</b>	<b>Introduction to Pthreads</b>	<b>3</b>
1.1	Mutex variables . . . . .	3
1.2	Condition variables . . . . .	3
<b>2</b>	<b>Examples</b>	<b>5</b>
2.1	Pi . . . . .	5
2.2	PDE solver . . . . .	5
<b>3</b>	<b>Matrices multiplication</b>	<b>9</b>
<b>A</b>	$\pi$	<b>I</b>
<b>B</b>	<b>Matrices multiplication</b>	<b>III</b>
B.1	Non-optimized code . . . . .	III
B.2	Optimized code . . . . .	V

# 1. Introduction to Pthreads

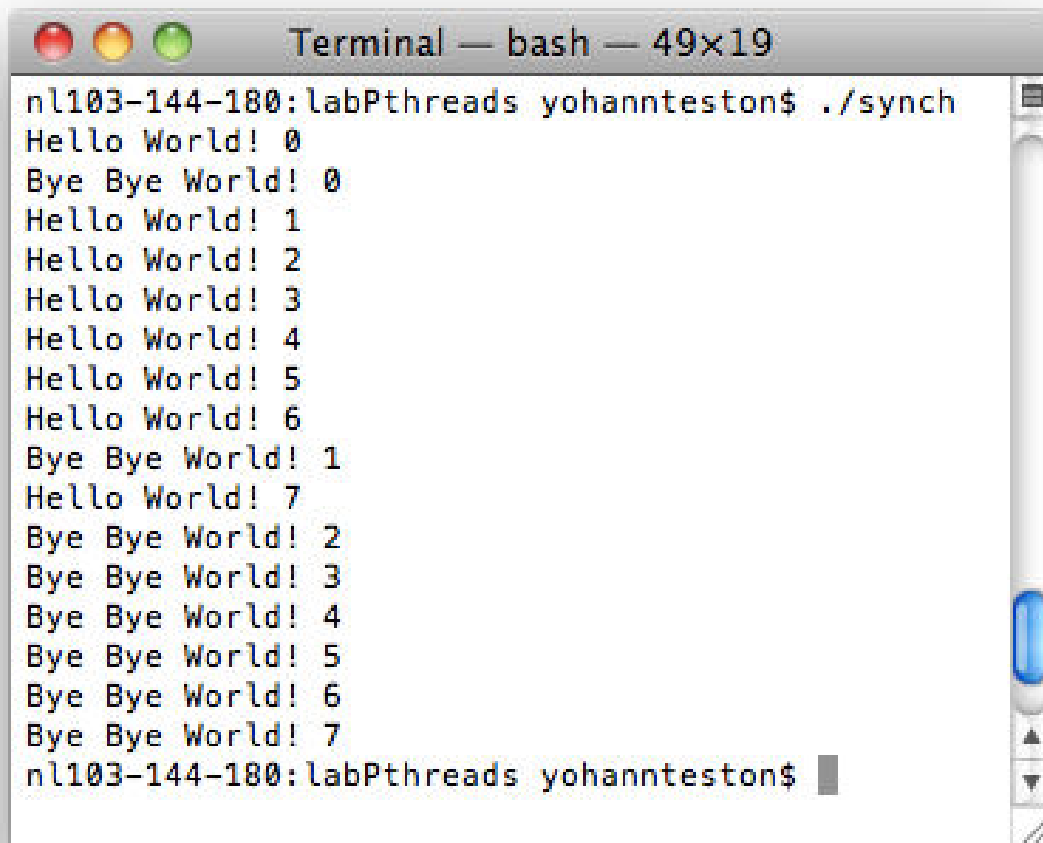
## 1.1 Mutex variables

If a mutex is not used to protect the access to the shared variable *sum*, the result will not be reliable. Indeed, race conditions will appear and modifications will then be lost. A race condition appear when several threads access the same memory space without protection. For example, a thread could read *sum* in a register, modify it and be preempted just before writing it back. When this thread will be allocated the CPU again, it will then erase all modifications that could have happened meanwhile by storing its local value into the shared variable.

Mutexes are then indispensable when it comes to multithreading.

## 1.2 Condition variables

If the barrier is removed in the first program, there will be no synchronization between the threads. They will not wait for everyone to be done with the first printing and selfishly continue. The output will then look like that:

A terminal window titled "Terminal — bash — 49x19" showing the execution of a program. The prompt is "nl103-144-180:labPthreads yohannteston\$". The program output consists of alternating "Hello World!" and "Bye Bye World!" messages, each followed by a thread ID. The sequence is: "Hello World! 0", "Bye Bye World! 0", "Hello World! 1", "Hello World! 2", "Hello World! 3", "Hello World! 4", "Hello World! 5", "Hello World! 6", "Bye Bye World! 1", "Hello World! 7", "Bye Bye World! 2", "Bye Bye World! 3", "Bye Bye World! 4", "Bye Bye World! 5", "Bye Bye World! 6", "Bye Bye World! 7". The prompt "nl103-144-180:labPthreads yohannteston\$" is visible at the bottom.

```
nl103-144-180:labPthreads yohannteston$ ./synch
Hello World! 0
Bye Bye World! 0
Hello World! 1
Hello World! 2
Hello World! 3
Hello World! 4
Hello World! 5
Hello World! 6
Bye Bye World! 1
Hello World! 7
Bye Bye World! 2
Bye Bye World! 3
Bye Bye World! 4
Bye Bye World! 5
Bye Bye World! 6
Bye Bye World! 7
nl103-144-180:labPthreads yohannteston$
```

In *spinwait.c*, the problem arises because the program does not load *state*'s value in a register for each test but do that once before the loop. So, when the last thread modifies *state*, its new value is not read by the other processes. Therefore, they keep spinning indefinitely. To avoid this, we have to make sure the compiler will generate a memory access each time *state* is accessed. We can do that using the keyword *volatile*. We can then try again the program and see that everything works fine.

## 2. Examples

### 2.1 Pi

The C code for this program is available in the appendix A.

### 2.2 PDE solver

Here are the speedup graphs for both of the synchronization methods.

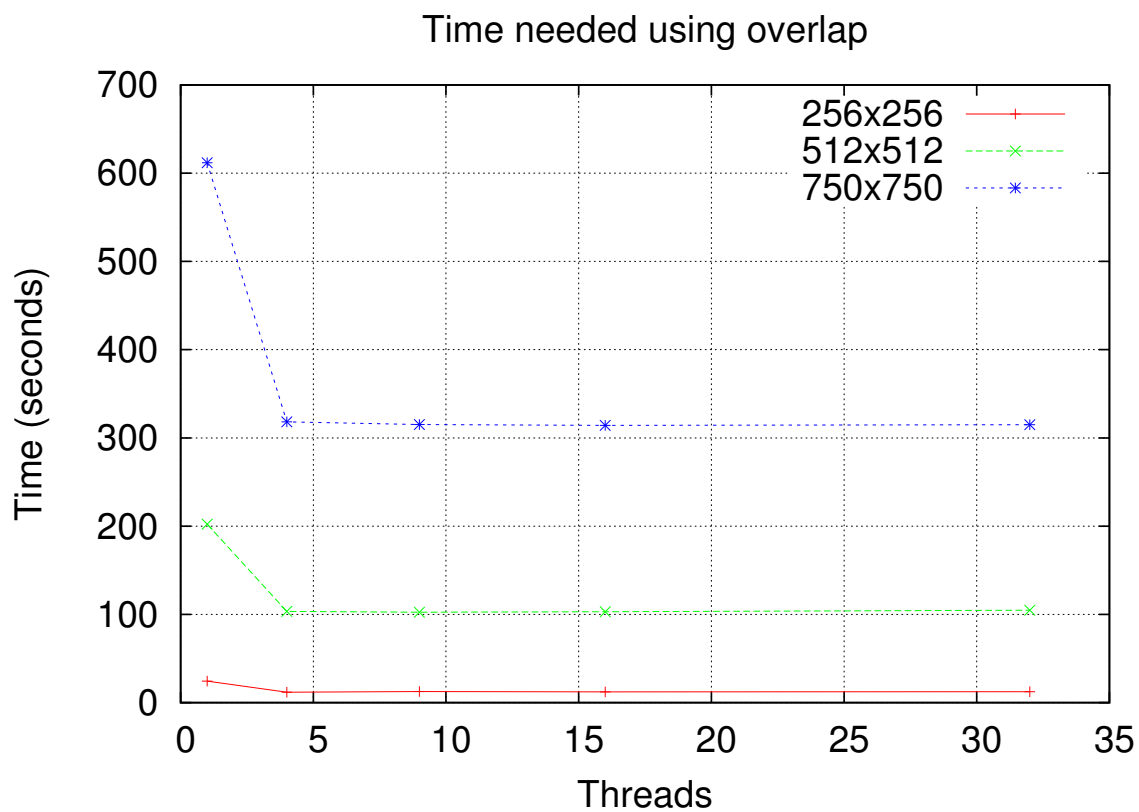


Figure 2.1: Time needed using overlap

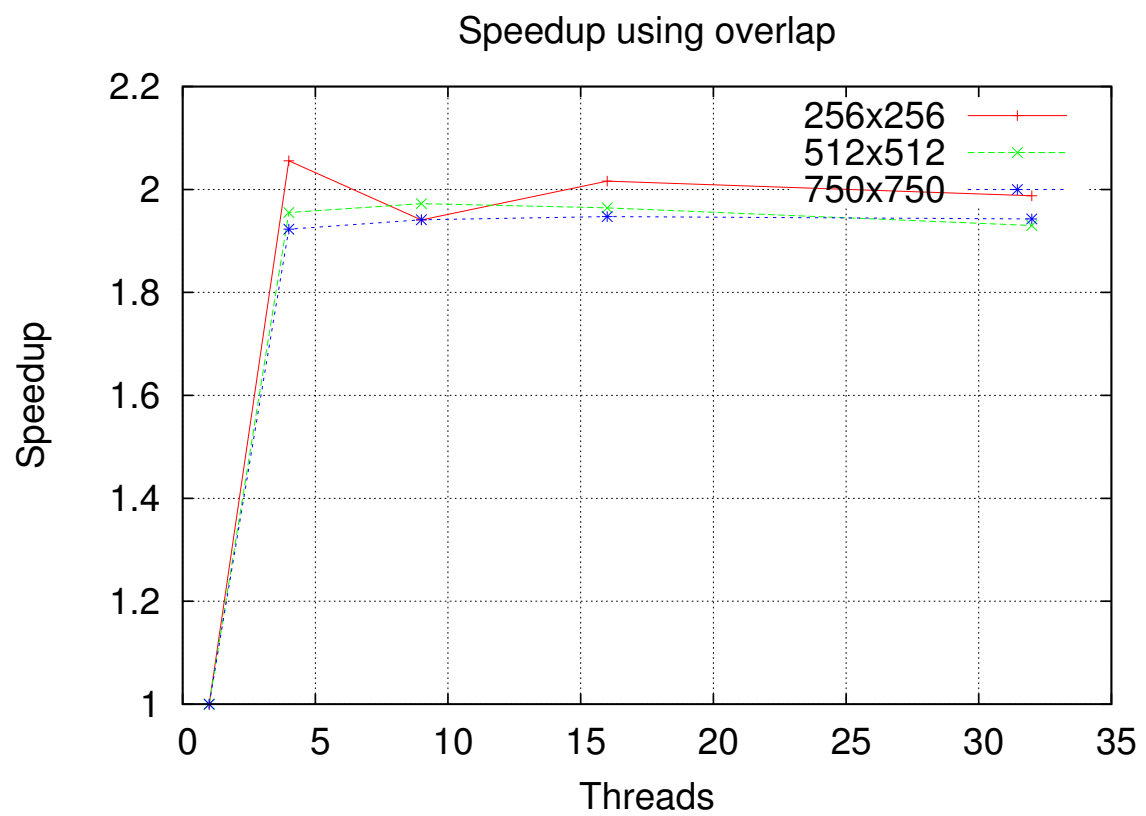


Figure 2.2: Speedup using overlap

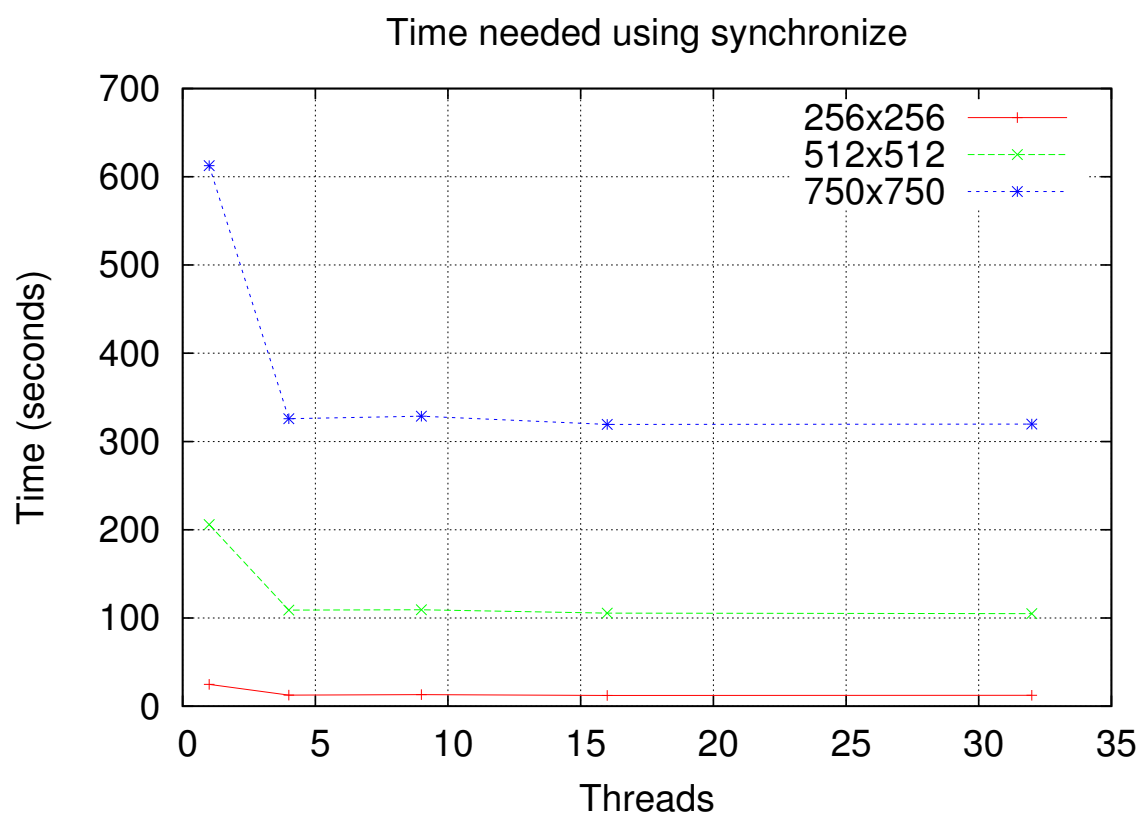


Figure 2.3: Time needed using synchronize

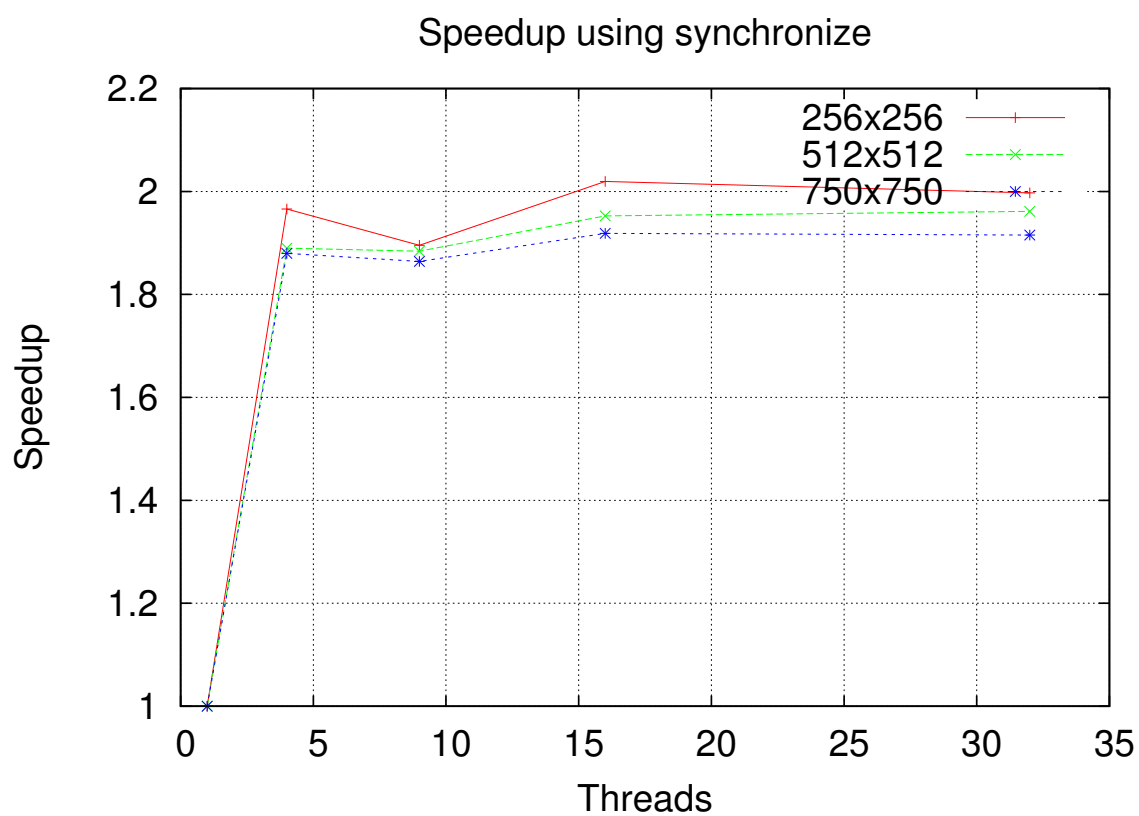


Figure 2.4: Speedup using synchronize



### 3. Matrices multiplication

The code of the non-optimized version of this program can be found in the appendix B.1 and that of the optimized version in the appendix B.2. The optimization done in the version is simply an inversion of the two inner loops used to perform the matrices multiplication. It optimizes the use of the cache and this is enough to speed the program up considerably.

The following picture presents the results.

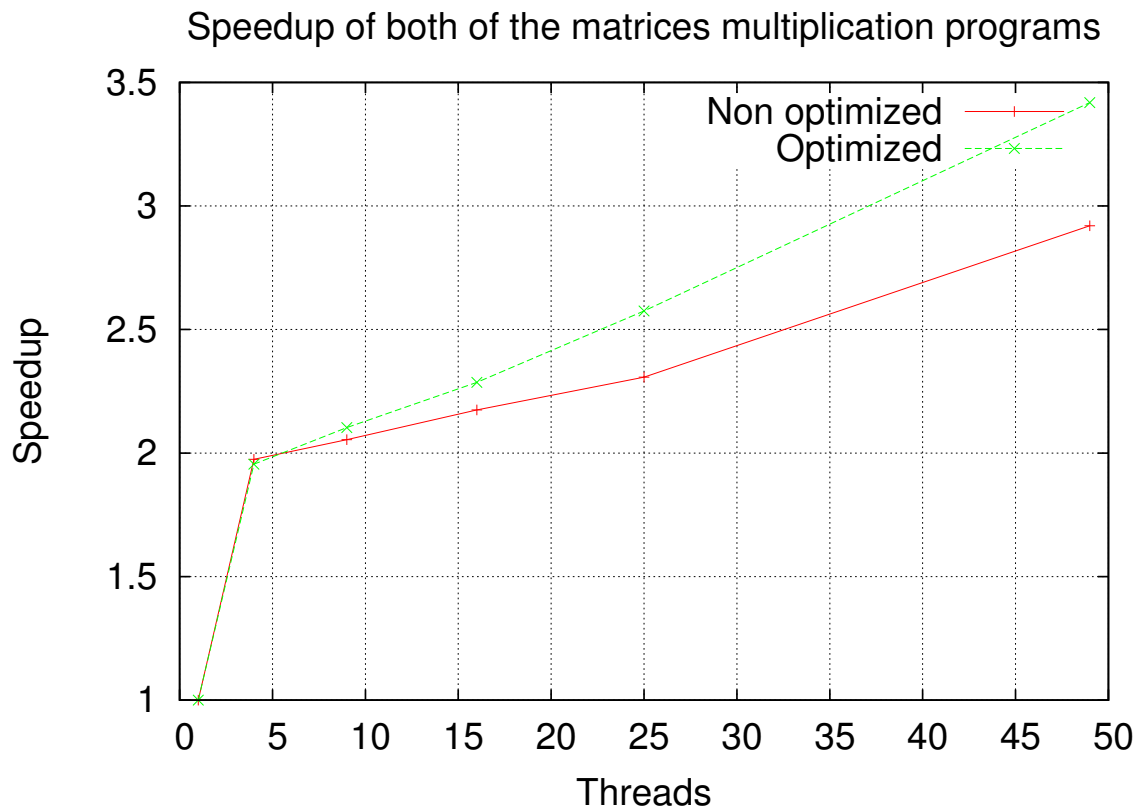


Figure 3.1: Speedup of both of the matrices multiplication programs

As we can see on the diagram, performances are better with the cache optimisation of the second version. This shows the importance of knowing how data is stored in the computer and how to optimize the access to it.

## A. $\pi$

```
/* *****  
 * This program calculates pi using C  
 *  
 ***** */  
#include <stdio.h>  
#include <pthread.h>  
#define NB_THREADS 20  
  
const int intervals = 100000000L ;  
int slice ;  
double dx ;  
double glob_sum ;  
pthread_mutex_t lock ;  
  
void* pi(void* arg){  
    int i ;  
    int end ;  
    int tid = (int) arg ;  
    double x, sum ;  
  
    sum = 0 ;  
  
    if (tid == NB_THREADS-1)  
        //we make sure the last step stops exactly at intervals  
        end = intervals ;  
    else  
        end = tid*slice+slice ;  
    /* computing pi */  
    for (i = tid*slice+1; i <= end; i++) {  
        x = dx*(i - 0.5) ;  
        sum += dx*4.0/(1.0 + x*x) ;  
    }  
    //a mutex is needed to protect the global variable  
    pthread_mutex_lock(&lock) ;  
    glob_sum += sum ;  
    pthread_mutex_unlock(&lock) ;  
}
```

```

    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

    int status, i;
    pthread_t threads[NB_THREADS];

    pthread_mutex_init(&lock, NULL);

    slice = intervals/NB_THREADS;
    dx = 1.0/intervals;
    glob_sum = 0.0;

    for (i = 0; i < NB_THREADS; i++) {
        pthread_create(&threads[i], NULL, pi, (void*)i);
    }
    for (i = 0; i < NB_THREADS; i++) {
        pthread_join(threads[i], (void*)&status);
    }

    pthread_mutex_destroy(&lock);
    printf("PI is approx. ~%.16f\n", glob_sum);
    pthread_exit(NULL);
    return 0;
}

```

## B. Matrices multiplication

### B.1 Non-optimized code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <pthread.h>

typedef struct{
    int i;
    int j;
}coords;

double **A,**B,**C;
int n, block_size;

void* matmul(void* arg){
    int i,j,k;
    //gets our coords on the "grid"
    coords c = *((coords*)arg);
    // Multiply C=A*B
    for(i=c.i*block_size;i<c.i * block_size + block_size;i++)
        for(j=c.i*block_size;j<c.j * block_size + block_size;j++)
            for(k=0;k<n;k++)
                C[i][j]+=A[i][k]*B[k][j];
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    int i,j,k,time,t,st;
    pthread_t* threads;
    coords* threads_coords;

    n = atoi(argv[1]);
    t = atoi(argv[2]);

    st = sqrt(t);
    if(t != st*st)
```

```

    exit(1);

    if(!(st % n))
        exit(1);
    block_size = n/st;
    // Allocate threads id
    threads = (pthread_t*)malloc(t*sizeof(pthread_t));
    threads_coords = (coords*)malloc(t*sizeof(coords));
    //Allocate and fill matrices
    A = (double **)malloc(n*sizeof(double *));
    B = (double **)malloc(n*sizeof(double *));
    C = (double **)malloc(n*sizeof(double *));
    for(i=0;i<n;i++){
        A[i] = (double *)malloc(n*sizeof(double));
        B[i] = (double *)malloc(n*sizeof(double));
        C[i] = (double *)malloc(n*sizeof(double));
    }

    for (i = 0; i<n; i++)
        for (j=0;j<n;j++){
            A[i][j] = rand() % 5 + 1;
            B[i][j] = rand() % 5 + 1;
            C[i][j] = 0.0;
        }

    time=timer();
    i = 0;
    j = 0;
    /*calculates the position of each thread on a virtual grid
    and passes them to the concerned thread */
    for (k = 0; k < t; k++){
        if(t && !(t % block_size)){
            i++;
            j = 0;
        }
        threads_coords[k].i = i;
        threads_coords[k].j = j;
        pthread_create(&threads[k], NULL, matmul,(void*)&threads_coords[k]);
    }

    for(k = 0; k<t; k++)
        pthread_join(threads[k], (void*)&i);
    time=timer()-time;
    printf("Elapsed_time:_%f_\n",time/1000000.0);
    free(threads);
    free(threads_coords);

```

```

    pthread_exit(NULL);
    return 0;

}

```

## B.2 Optimized code

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <pthread.h>

typedef struct{
    int i;
    int j;
}coords;

double **A,**B,**C;
int n, block_size;

void* matmul(void* arg){
    int i,j,k;
    coords c = *((coords*)arg);
    // Multiply C=A*B
    for(i=c.i*block_size;i<c.i * block_size + block_size;i++)
        //optimization: j and k loops have been swapped
        for (k=0;k<n;k++)
            for (j=c.i*block_size;j<c.j * block_size + block_size;j++)
                C[i][j]+=A[i][k]*B[k][j];
    pthread_exit(NULL);
}

void print(double **e){
    int i,j,k;
    for(i=0;i<n;i++){
        for(j=0;j<n;j++)
            printf("_%f_",e[i][j]);
        printf("\n");
    }
}

int main(int argc, char *argv[]) {
    int i,j,k,time,t,st;
    pthread_t* threads;
    coords* threads_coords;

```

```

n = atoi(argv[1]);
t = atoi(argv[2]);

st = sqrt(t);
if(t != st*st)
    exit(1);

if(!(st % n))
    exit(1);
block_size = n/st;
// Allocate threads id
threads = (pthread_t*)malloc(t*sizeof(pthread_t));
threads_coords = (coords*)malloc(t*sizeof(coords));
//Allocate and fill matrices
A = (double **)malloc(n*sizeof(double *));
B = (double **)malloc(n*sizeof(double *));
C = (double **)malloc(n*sizeof(double *));
for(i=0;i<n;i++){
    A[i] = (double *)malloc(n*sizeof(double));
    B[i] = (double *)malloc(n*sizeof(double));
    C[i] = (double *)malloc(n*sizeof(double));
}

for (i = 0; i<n; i++)
    for (j=0;j<n;j++){
        A[i][j] = rand() % 5 + 1;
        B[i][j] = rand() % 5 + 1;
        C[i][j] = 0.0;
    }

time=timer();
i = 0;
j = 0;
/* calculates the position of each thread on a virtual grid
   and passes them to the concerned thread */
for (k = 0; k < t; k++){
    if(t && !(t % block_size)){
        i++;
        j = 0;
    }
    threads_coords[k].i = i;
    threads_coords[k].j = j;
    pthread_create(&threads[k], NULL, matmul, (void*)&threads_coords[k]);
}

```

```
    for (k = 0; k < t; k++)
        pthread_join(threads[k], (void*)&i);
    time = timer() - time;
    printf("Elapsed_time: %f\n", time / 1000000.0);
    free(threads);
    free(threads_coords);
    pthread_exit(NULL);
    return 0;
}
```