BIRZEIT UNIVERSITY

Faculty of Engineering & Technology Electrical & Computer Engineering Department

OPERATING SYSTEMS

ENCS3390

Process and Thread Management

Prepared by: QOSSAY RIDA 1211553

Instructor: Dr. Bashar Tahayna

Section: Four

Date: 2023/11/20

Table of Contents

Table of Figures

Table of Tables

Introduction:

Matrix multiplication is a fundamental operation in linear algebra and has applications in various domains, such as computer graphics, scientific computing, and machine learning. This Task explores different approaches to optimize matrix multiplication in C, ranging from a naive serial implementation to parallel implementations using processes and threads.

The primary objective is to compare the performance of different approaches to matrix multiplication, including a naive serial solution, a process-based solution, and various thread-based solutions. The focus is on achieving optimal execution time while maintaining correct results.

Implemented Approaches:

1. Naive Approach

The naive approach represents a conventional method of matrix multiplication, using nested loops. It serves as a baseline for performance comparison.

2. Process-Based Approach

The process-based approach involves dividing the matrix multiplication task among multiple processes. Each process is responsible for a subset of the rows, and communication between processes is achieved using inter-process communication (IPC) via pipes.

3. Thread-Based Approaches

3.1 Join Threads

This approach uses multiple joinable threads to parallelize the matrix multiplication task. Each thread is assigned a subset of rows to compute, and results are combined upon thread completion.

3.2 Mix of Join and Detached Threads

This approach combines joinable and detached threads to exploit the benefits of both. The detached threads focus on completing their tasks independently, while the joinable threads ensure synchronization before completing the entire computation.

3.3 Detached Threads

This approach employs detached threads, allowing them to run independently without the need for explicit synchronization. The results are checked for correctness after completion.

The code is appended in Appendix A.

Experimental Setup:

The trials were executed on a system equipped with an 11th Gen Intel® Core™ i7-1165G7 processor boasting 8 cores. Notably, authentic hardware was utilized, eschewing the use of a simulated environment, and the primary operating system installed was Linux. The matrix dimensions were defined as 100x100, and, for illustrative purposes, the configuration involved 4 processes and threads. It is important to highlight that subsequent comparisons will delve into variations in the number of processes and threads.

Results and Analysis

I will run the code five times and will include the sample run in Appendix B. I will then calculate the average time for each method to ensure greater accuracy.

1. **Naive Approach**

The naive approach provides a baseline for execution time, serving as a reference point for performance comparison.

The data presented in Table 1 from Appendix B highlights the time taken by this method. Upon running the code four times, the computed average time required by the method is 0.0036725, resulting in a throughput of 272 times per second.

Figure 1 visually illustrates the consistency and stability of the method's execution time across each run.

Run No.	Time (Seconds)
1	0.003716
2	0.0036
3	0.003351
4	0.004023

Average=	0.0036725
Throughput=	272.2940776

Table 1: Data presented from naive approach

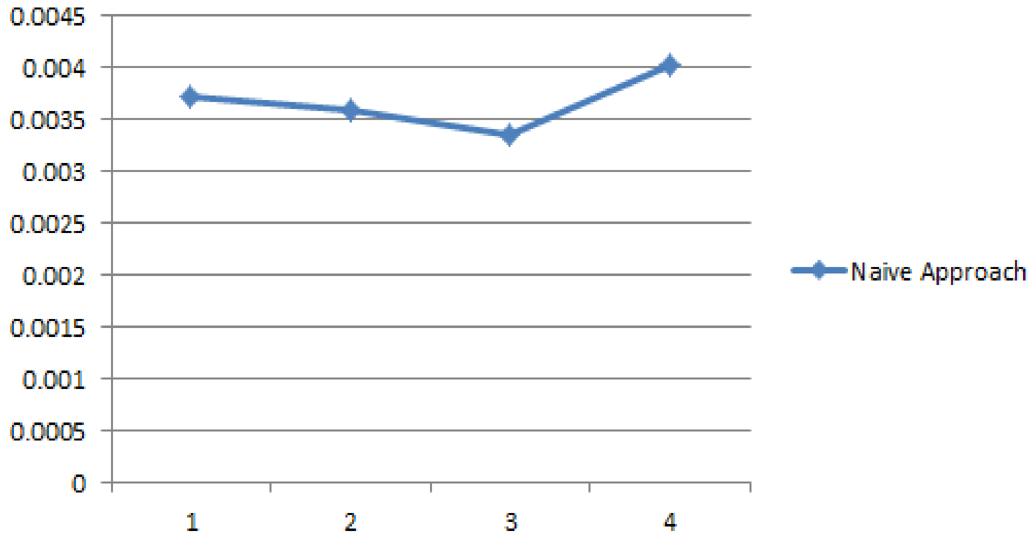

Figure 1: Stability of naive approach.

2. Process-Based Approach

The process-based approach shows improvements over the naive approach, with the use of multiple processes. However, the overhead of inter-process communication impacts overall performance.

The data presented in Table 2 from Appendix B highlights the time taken by this method. Upon running the code four times, the computed average time required by the method is 0.001755, resulting in a throughput of 570 times per second.

Figure 2 visually illustrates the consistency and stability of the method's execution time across each run.

Run No.	Time (Seconds)
1	0.002048
2	0.002059
3	0.001288
4	0.001625

Average=	0.001755
Throughput=	569.8005698

Table 2: Data presented from process-based.

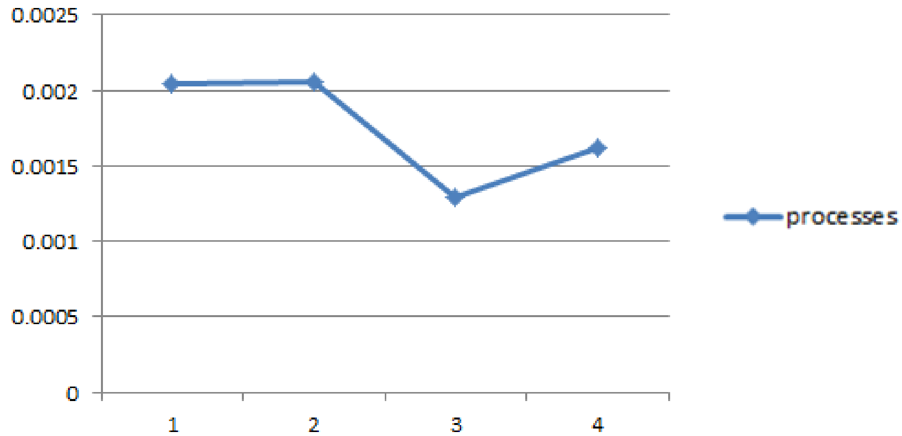

Figure 2: Stability of process-based.

3. Thread-Based Approaches

3.1 Join Threads

The use of joinable threads provides a speedup compared to the naive approach. Joinable threads refer to threads that can be explicitly waited for or 'joined' by the main program, allowing it to synchronize its execution with the completion of the threads. This ensures that the main program doesn't proceed before the threads it has spawned have finished their tasks.

The data presented in Table 3 from Appendix B highlights the time taken by this method. Upon running the code four times, the computed average time required by the method is 0.0014965, resulting in a throughput of 668 times per second.

Figure 3 visually illustrates the consistency and stability of the method's execution time across each run.

Run No.	Time (Seconds)
1	0.001875
2	0.001871
3	0.001097
4	0.001143

Average=	0.0014965
Throughput=	668.2258603

Table 3: Data presented from join thread.

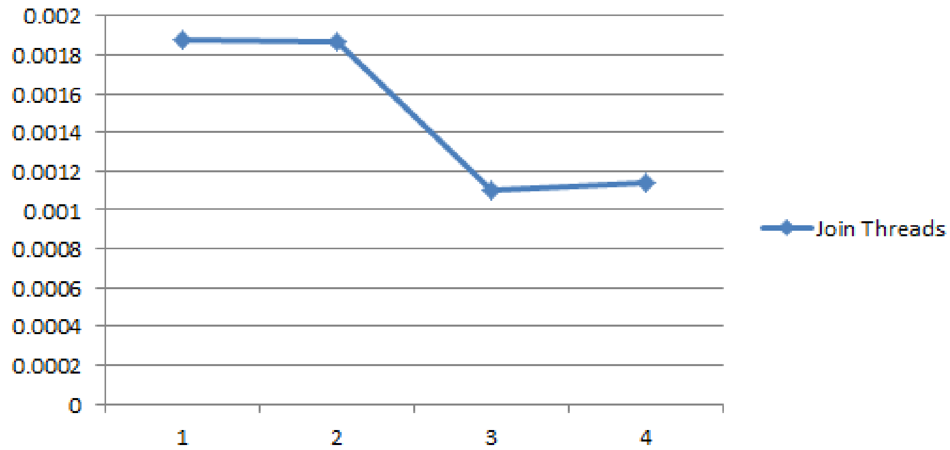

Figure 3: Stability of join threads.

3.2 Mix of Join and Detached Threads

Combining joinable and detached threads aims to balance the advantages of synchronization and independent execution. Joinable threads allow the main program to wait for the thread to complete its execution as we have shown. On the other hand, detached threads operate independently of the main program, releasing system resources upon termination without the need for an explicit join. This autonomy enhances efficiency, as the main program can continue its execution without waiting for the detached thread to complete.

The data presented in Table 4 from Appendix B highlights the time taken by this method. Upon running the code four times, the computed average time required by the method is 0.00154325 for worst case, resulting in a throughput of 648 times per second.

Figure 4 visually illustrates the consistency and stability of the method's execution time across each run.

Run No.	Thread 0	Thread 1	Thread 2	Thread 3	Worst case (Seconds)
1	0.001741	0.001373	0.00179	0.001508	0.00179
2	0.01163	0.000919	0.0 01741	0.001731	0.001731
3	0.000923	0.000939	0.000923	0.001001	0.001001
4	0.001644	0.000919	0.000923	0.001651	0.001651

Average=	0.00154325
Throughput=	647.983152

Table 4: Data presented from mix threads.

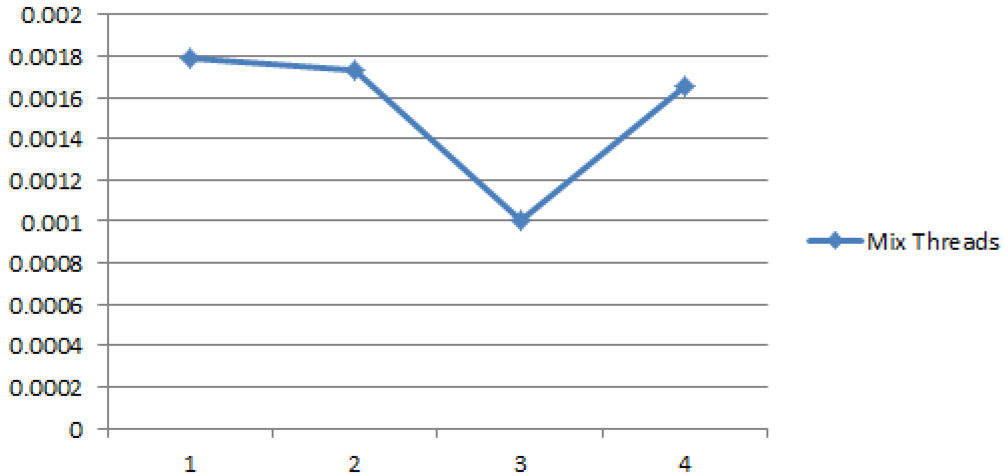

Figure 4: Stability of mix threads.

3.3 Detached Threads

The detached threads approach, which shows potential for further optimization, involves creating threads that operate independently without requiring explicit synchronization with the main thread. In this approach, once a detached thread is created, it continues to execute independently of the main thread, allowing for concurrent and parallel processing, Which means that the time he will need from the main thread is only the time of creation, so his time was calculated separately and his result was verified separately because the main thread does not wait for him.

The data presented in Table 5 from Appendix B highlights the time taken by this method. Upon running the code four times, the computed average time required is 0.001614 for worst case and the computed average time for creation is 0.000052, resulting in a throughput of 648 times per second.

Figure 5 visually illustrates the consistency and stability of the method's execution time across each run.

Run No.	Thread 0	Thread 1	Thread 2	Thread 3	Creation	Worst case (Seconds)
1	0.001739	0.001755	0.001764	0.001757	0.000077	0.001764
2	0.000985	0.001739	0.001043	0.00177	0.000048	0.00177
3	0.00093	0.001162	0.000948	0.001031	0.000027	0.001162
4	0.001762	0.000973	0.001012	0.0001758	0.000056	0.001762

Creation avg =	0.000052
Average=	0.0016145
Throughput=	619.386807

Table 5: Data presented from detached thread.

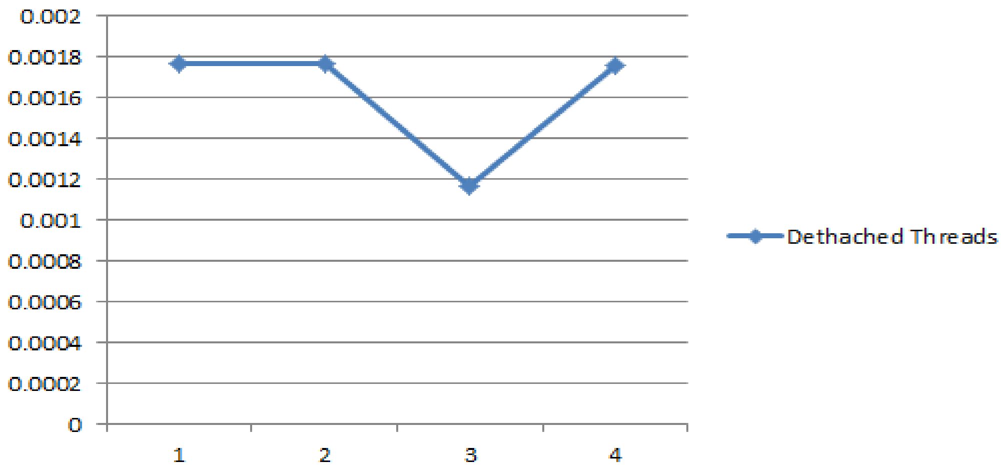

Figure 5: Stability of detached threads.

Compare results

Analyze the results of the different approaches and determine which method is the most efficient.

- Naive Approach:
  Average Time: 0.0036725 seconds
  Throughput: 272 times per second

- Process-Based Approach:
  Average Time: 0.001755 seconds
  Throughput: 570 times per second

- Thread-Based Approach - Join Threads:
  Average Time: 0.0014965 seconds
  Throughput: 668 times per second

- Thread-Based Approach - Mix of Join and Detached Threads:
  Average Time: 0.00154325 seconds (worst case)
  Throughput: 648 times per second

- Thread-Based Approach - Detached Threads:
  Average Time for Execution: 0.001614 seconds (worst case)
  Average Time for Creation: 0.000052 seconds
  Throughput: 648 times per second

The naive approach has the slowest performance, and all other approaches show significant improvements, such as the process-based approach has a better throughput than the naive approach but is outperformed by all thread-based approaches, Among the thread-based approaches, "Join Threads" has the best throughput, followed closely by the "Mix of Join and Detached Threads", While detached threads have a quick creation time, their overall performance is not significantly better than the joinable threads. However, if the creation time is of utmost importance and the slightly lower throughput is acceptable, detached threads might be a viable option.

In summary, Processes provide parallelism by running independently but may incur some overhead due to IPC, threads share the same memory space, reducing communication overhead and allowing for efficient parallelism and detached Threads further reduce creation time but may sacrifice some level of synchronization.

Performance: Multiprocessing vs. Multithreading

Table 6 presents data from Appendix C, detailing the time taken by processes and threads as their numbers increase. In Figure 6, we explored the execution times for different numbers of processes and threads on an 8-core device. The results show that as the number of processes and threads increases, so does the execution time decrease if the number remains less than four. If the number exceeds four the 8-core architecture becomes crucial, and surpassing this core count leads to context switching, causing increased overhead, due to heightened competition for limited resources like CPU time, cache, and memory bandwidth. Performance degradation is influenced by overhead from thread creation and management, along with synchronization challenges in increased parallelism. Striking a balance between the number of processes/threads and available resources is essential for optimal performance. These findings emphasize the importance of careful resource allocation and configuration tuning for parallel computing on multi-core systems.

No.	processes	join threads	mix threads	detached threads
2	0.00213	0.00178	0.001662	0.00176
3	0.00161	0.00139	0.001259	0.001251
4	0.001254	0.001009	0.000925	0.0011
5	0.001804	0.00135	0.001399	0.001
6	0.001621	0.001412	0.001174	0.0012

Table 6: Data presented from Appendix C.

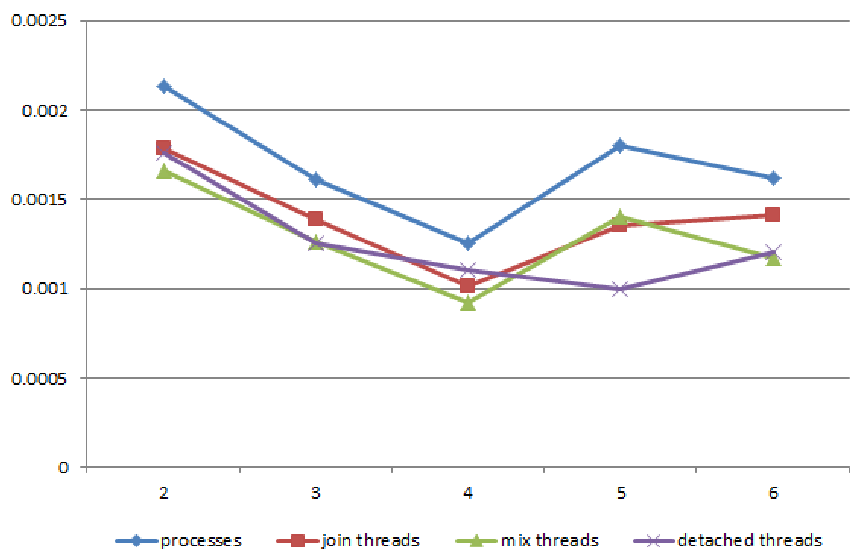

Figure 6: Chart Multiprocessing vs. Multithreading.

Appendix A

Including libraries, function prototypes, and defining matrices:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>
#define MATRIX_SIZE 100
#define VALUE_SIZE 40
#define NUM_PROCESSES 4
#define NUM_THREADS 4

// Function pointer type for functions that take no arguments and return void
typedef void (*FunctionPointer)();

// Matrices for input, output, and results of different multiplication approaches
int matrixA[MATRIX_SIZE][MATRIX_SIZE];
int matrixB[MATRIX_SIZE][MATRIX_SIZE];
int resultByNaiveApproach[MATRIX_SIZE][MATRIX_SIZE];
int resultByProcesses[MATRIX_SIZE][MATRIX_SIZE];
int resultByJoinThreads[MATRIX_SIZE][MATRIX_SIZE];
int resultByMixThreads[MATRIX_SIZE][MATRIX_SIZE];
int resultByDetachedThreads[MATRIX_SIZE][MATRIX_SIZE];

// Function to initialize a matrix with random values based on a string
void initializeMatrix(int matrix[MATRIX_SIZE][MATRIX_SIZE], char string[VALUE_SIZE]);

// Function to check if two matrices are equal
void areResultsEqual(int M1[MATRIX_SIZE][MATRIX_SIZE], int
M2[MATRIX_SIZE][MATRIX_SIZE]);

// Function to print a matrix
void printMatrix(int matrix[MATRIX_SIZE][MATRIX_SIZE]);

// Function to measure the execution time of a given function
double measureExecutionTime(FunctionPointer func);

// Function to perform matrix multiplication using the naive approach
void multiplyMatrices();

// Function to perform matrix multiplication using processes
void parallelMMWithProcesses();

// Function to perform matrix multiplication using threads with join
void parallelMMWithJoinThreads();
void* joinThreadMM(void* arg);

// Function to perform matrix multiplication using mixed threads (combination of join
and detached)
void parallelMMWithMixThreads();
void* mixThreadMM(void* arg);

// Function to perform matrix multiplication using detached threads
void parallelMMWithDetachedThreads();
void* detachedThreadMM(void* arg);
```

The main function to manage and print the final results:

```
int main() {
    // Input numbers as strings
    char myNumber[] = "1211553";
    char birthYear[] = "2003";

    // Convert strings to long long integers
    long long num1 = atoi(myNumber);
    long long num2 = atoi(birthYear);
    long long result = num1 * num2;

    // Convert the result back to a string
    char resultStr[VALUE_SIZE];
    sprintf(resultStr, "%lld", result);

    // Initialize matrices with the input numbers and the result
    initializeMatrix(matrixA, myNumber);
    initializeMatrix(matrixB, resultStr);

    printf("QOSSAY RIDI %s\n", myNumber);

    // Measure and display the execution time of the naive matrix multiplication
    double naiveTime = measureExecutionTime(multiplyMatrices);
    printf("Naive solution:\n\tNaive approach: %f seconds\n\n", naiveTime);

    // Measure and display the execution time of matrix multiplication with processes
    double processTime = measureExecutionTime(parallelMMWithProcesses);
    printf("Processes solution:\n\tProcess-based approach with %d processes: %f
seconds\n", NUM_PROCESSES, processTime);

    // Check if results from naive and process approaches are equal
    areResultsEqual(resultByNaiveApproach, resultByProcesses);

    // Measure and display the execution time of matrix multiplication with join
threads
    double joinThreadTime = measureExecutionTime(parallelMMWithJoinThreads);
    printf("Threads solution:\n\t1- Thread-based approach with %d join threads: %f
seconds\n", NUM_THREADS, joinThreadTime);

    // Check if results from naive and thread (join) approaches are equal
    areResultsEqual(resultByNaiveApproach, resultByJoinThreads);

    // Perform matrix multiplication with mixed (join and detached) threads
    parallelMMWithMixThreads();

    // Measure and display the execution time of matrix multiplication with detached
threads
    double detachedThreadTime = measureExecutionTime(parallelMMWithDetachedThreads);
    printf("\n\t3- Thread-based approach with %d detached threads: such as time of
creation = %f seconds\n",NUM_THREADS,detachedThreadTime);

    // Exit the main thread ; this allow to wait all thread not finished
    pthread_exit(0);
}
```

Some functions that act as equipment:

```c
// Function to initialize a matrix with repeating digits from a string
void initializeMatrix(int matrix[MATRIX_SIZE][MATRIX_SIZE], char string[VALUE_SIZE]) {
    int size = 0;

    // Calculate the size of the string
    while (string[size] != '\0')
        size++;

    int index = 0;

    // Fill the matrix with repeating digits from the string
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            matrix[i][j] = string[index] - '0';
            index = (index + 1) % size;
        }
    }
}

// Function to measure the execution time of a given function
double measureExecutionTime(FunctionPointer func) {
    struct timespec start_time, end_time;

    // Record the start time
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // Execute the specified function
    func();

    // Record the end time
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    // Calculate and return the elapsed time in seconds
    return (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
}

// Function to check if two matrices are equal
void areResultsEqual(int M1[MATRIX_SIZE][MATRIX_SIZE], int M2[MATRIX_SIZE][MATRIX_SIZE]) {
    int flag = 1;

    // Compare each element of the matrices
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++)
            if (M1[i][j] != M2[i][j])
                flag = 0;

    // Print the result of the equality check
    if (flag == 1)
        printf("\t\tThe result is identical to that of the naive approach\n\n");
    else
        printf("\t\tThe result isn't identical to that of the naive approach\n\n");
}

// Function to print a matrix
void printMatrix(int matrix[MATRIX_SIZE][MATRIX_SIZE]) {
    // Print each element of the matrix
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

Naive approach:

```
// ******************************************************************************************
// ******************************************************************************************
//              A naive approach, matrix multiplication in a conventional way
//
// ******************************************************************************************
// ******************************************************************************************

void multiplyMatrices() {
    // Iterate through each element of the result matrix
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            resultByNaiveApproach[i][j] = 0;
            // Perform the multiplication and accumulate the result
            for (int k = 0; k < MATRIX_SIZE; k++)
                resultByNaiveApproach[i][j] += matrixA[i][k] * matrixB[k][j];

        }
    }
}
```

Processes approach:

```c
// ********************************************************************************************
// ********************************************************************************************
//                  Processes approach, using several processes to multiply matrices
// ********************************************************************************************
// ********************************************************************************************

void parallelMMWithProcesses() {
    pid_t child_pid = 0;
    int status;
    int pipes[NUM_PROCESSES][2];

    // Create pipes for communication between parent and child processes
    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (pipe(pipes[i]) == -1) {
            printf("Pipe failed");
            return;
        }

        // Fork a child process
        child_pid = fork();

        if (child_pid == 0) {
            // Code executed by child processes

            // Calculate the range of rows to be processed by each child
            int start_row = i * (MATRIX_SIZE / NUM_PROCESSES);
            int end_row = (i + 1) * (MATRIX_SIZE / NUM_PROCESSES);

            // Adjust the end_row for the last process
            if (i == NUM_PROCESSES - 1)
                end_row = MATRIX_SIZE;

            // Perform matrix multiplication for the assigned rows
            for (int j = start_row; j < end_row; j++) {
                for (int k = 0; k < MATRIX_SIZE; k++) {
                    resultByProcesses[j][k] = 0;
                    for (int l = 0; l < MATRIX_SIZE; l++)
                        resultByProcesses[j][k] += matrixA[j][l] * matrixB[l][k];
                }
            }

            // Write the result to the pipe and exit the child process
            write(pipes[i][1], resultByProcesses + start_row, (end_row - start_row) * MATRIX_SIZE * sizeof(int));
            close(pipes[i][0]); // Close the reading end of the pipe
            close(pipes[i][1]); // Close the writing end of the pipe
            exit(0);
        }
    }

    // Code executed by the parent proess

    // Wait for all child processes to finish
    for (int i = 0; i < NUM_PROCESSES; i++) {
        wait(&status);

        // Calculate the range of rows to be read from each child
        int start_row = i * (MATRIX_SIZE / NUM_PROCESSES);
        int end_row = (i + 1) * (MATRIX_SIZE / NUM_PROCESSES);

        // Adjust the end row for the last process
        if (i == NUM_PROCESSES - 1)
            end_row = MATRIX_SIZE;

        // Read the result from the pipe and close the pipe
        read(pipes[i][0], resultByProcesses + start_row, (end_row - start_row) * MATRIX_SIZE * sizeof(int));
        close(pipes[i][0]); // Close the reading end of the pipe
        close(pipes[i][1]); // Close the writing end of the pipe
    }
}
```

Join threads approach:

```
// *********************************************************************************************
// *********************************************************************************************
//                 Join threads approach, using several join threads to multiply matrices
// *********************************************************************************************
// *********************************************************************************************

void parallelMMWithJoinThreads() {
    // Array to store thread identifiers and an array for thread IDs
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, joinThreadMM, &thread_ids[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* joinThreadMM(void* arg) {
    // Extract the thread ID from the argument
    int thread_id = *((int*)arg);

    // Calculate the range of rows to be processed by this thread
    int start_row = thread_id * (MATRIX_SIZE / NUM_THREADS);
    int end_row = (thread_id + 1) * (MATRIX_SIZE / NUM_THREADS);

    // Adjust the end_row for the last thread
    if (thread_id == NUM_THREADS - 1)
        end_row = MATRIX_SIZE;

    // Perform matrix multiplication for the assigned rows
    for (int j = start_row; j < end_row; j++) {
        for (int k = 0; k < MATRIX_SIZE; k++) {
            resultByJoinThreads[j][k] = 0;
            for (int l = 0; l < MATRIX_SIZE; l++)
                resultByJoinThreads[j][k] += matrixA[j][l] * matrixB[l][k];
        }
    }

    return NULL;
}
```

Mix of join & detached threads approach:

```c
// ***********************************************************************************************
// ***********************************************************************************************
//       Mix of join & detached threads approach, using several threads to multiply matrices
// ***********************************************************************************************
// ***********************************************************************************************

void parallelMMWithMixThreads() {
    printf("\t2- Thread-based approach with mix join & detached threads\n");
    pthread_t threads[NUM_THREADS];         // Array to store thread identifiers for detached threads

    // Allocate memory for thread IDs for detached threads
    int* detachedThread_ids = (int*)malloc(NUM_THREADS / 2 * sizeof(int));

    for (int i = 0; i < NUM_THREADS / 2; i++) {
        detachedThread_ids[i] = i;
        pthread_create(&threads[i], NULL, mixThreadMM, &detachedThread_ids[i]);
        pthread_detach(threads[i]);    // Create and detach detached threads
    }

    int joinThread_ids[NUM_THREADS / 2];    // Array to store thread IDs for join threads
    for (int i = NUM_THREADS / 2; i < NUM_THREADS; i++) {
        joinThread_ids[i - NUM_THREADS / 2] = i;
        pthread_create(&threads[i], NULL, mixThreadMM, &joinThread_ids[i - NUM_THREADS / 2]);
    }

    // Wait for join threads to finish
    for (int i = NUM_THREADS / 2; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}

void* mixThreadMM(void* arg) {

    int thread_id = *((int*)arg); // Extract the thread ID from the argument

    struct timespec start_time, end_time;   // Record the start time
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // Calculate the range of rows to be processed by this thread
    int start_row = thread_id * (MATRIX_SIZE / NUM_THREADS);
    int end_row = (thread_id + 1) * (MATRIX_SIZE / NUM_THREADS);
    if (thread_id == NUM_THREADS - 1)   // Adjust the end_row for the last thread
        end_row = MATRIX_SIZE;

    // Perform matrix multiplication for the assigned rows
    for (int j = start_row; j < end_row; j++) {
        for (int k = 0; k < MATRIX_SIZE; k++) {
            resultByMixThreads[j][k] = 0;
            for (int l = 0; l < MATRIX_SIZE; l++)
                resultByMixThreads[j][k] += matrixA[j][l] * matrixB[l][k];
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time);  // Record the end time
    double time = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_nsec - start_time.tv_nsec) / 1e9;

    // Check if the result matches the naive approach
    int flag = 1;
    for (int i = start_row; i < end_row; i++)
        for (int j = 0; j < MATRIX_SIZE; j++)
            if (resultByMixThreads[i][j] != resultByNaiveApproach[i][j])
                flag = 0;

    // Print information about the thread's work completion
    if (flag && thread_id < NUM_THREADS / 2) {
        printf("\t\tFrom detached thread No. %d, ", thread_id);
        printf("The required work was completed successfully within %f seconds\n", time);
    } else if (flag) {
        printf("\t\tFrom join thread No. %d, ", thread_id);
        printf("The required work was completed successfully within %f seconds\n", time);
    } else
        printf("\t\tFrom thread No. %d, There is an error that has occurred\n", thread_id);

    return NULL;
}
```

Detached threads approach:

```c
// *********************************************************************************************
// *********************************************************************************************
//              Detached threads approach, using several detached threads to multiply matrices
// *********************************************************************************************
// *********************************************************************************************

void parallelMMWithDetachedThreads() {
    // Array to store thread identifiers and allocate memory for thread IDs
    pthread_t threads[NUM_THREADS];
    int* thread_ids = (int*)malloc(NUM_THREADS * sizeof(int));

    // Create and detach detached threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, detachedThreadMM, &thread_ids[i]);
        pthread_detach(threads[i]);
    }
}

void* detachedThreadMM(void* arg) {
    // Extract the thread ID from the argument
    int thread_id = *((int*)arg);

    // Record the start time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // Calculate the range of rows to be processed by this thread
    int start_row = thread_id * (MATRIX_SIZE / NUM_THREADS);
    int end_row = (thread_id + 1) * (MATRIX_SIZE / NUM_THREADS);

    // Adjust the end_row for the last thread
    if (thread_id == NUM_THREADS - 1)
        end_row = MATRIX_SIZE;

    // Perform matrix multiplication for the assigned rows
    for (int i = start_row; i < end_row; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            resultByDetachedThreads[i][j] = 0;
            for (int k = 0; k < MATRIX_SIZE; k++)
                resultByDetachedThreads[i][j] += matrixA[i][k] * matrixB[k][j];
        }
    }

    // Record the end time
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    // Calculate the elapsed time in seconds
    double time = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_nsec - start_time.tv_nsec) / 1e9;

    // Check if the result matches the naive approach
    int flag = 1;
    for (int i = start_row; i < end_row; i++)
        for (int j = 0; j < MATRIX_SIZE; j++)
            if (resultByDetachedThreads[i][j] != resultByNaiveApproach[i][j])
                flag = 0;

    // Print information about the thread's work completion
    if (flag) {
        printf("\t\tFrom detached thread No. %d, ", thread_id);
        printf("The required work was completed successfully within %f seconds\n", time);
    } else {
        printf("\t\tFrom detached thread No. %d, There is an error that has occurred\n", thread_id);
    }

    return NULL;
}
```

Appendix B

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003716 seconds

Processes solution:
    Process-based approach with 4 processes: 0.002048 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 4 join threads: 0.001875 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From detached thread No. 1, The required work was completed successfully within 0.001373 seconds
        From join thread No. 3, The required work was completed successfully within 0.001508 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001741 seconds
        From join thread No. 2, The required work was completed successfully within 0.001790 seconds

    3- Thread-based approach with 4 detached threads: such as time of creation = 0.000077 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001739 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001755 seconds
        From detached thread No. 3, The required work was completed successfully within 0.001757 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001764 seconds
```

Figure 7: Sample run #1.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003600 seconds

Processes solution:
    Process-based approach with 4 processes: 0.002059 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 4 join threads: 0.001871 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From join thread No. 2, The required work was completed successfully within 0.000919 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001163 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001731 seconds
        From join thread No. 3, The required work was completed successfully within 0.001741 seconds

    3- Thread-based approach with 4 detached threads: such as time of creation = 0.000048 seconds
        From detached thread No. 0, The required work was completed successfully within 0.000985 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001043 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001739 seconds
        From detached thread No. 3, The required work was completed successfully within 0.001770 seconds
```

Figure 8: Sample run #2.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003351 seconds

Processes solution:
    Process-based approach with 4 processes: 0.001288 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 4 join threads: 0.001029 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From join thread No. 2, The required work was completed successfully within 0.000923 seconds
        From detached thread No. 1, The required work was completed successfully within 0.000939 seconds
        From detached thread No. 0, The required work was completed successfully within 0.000923 seconds
        From join thread No. 3, The required work was completed successfully within 0.001001 seconds

    3- Thread-based approach with 4 detached threads: such as time of creation = 0.000027 seconds
        From detached thread No. 0, The required work was completed successfully within 0.000930 seconds
        From detached thread No. 2, The required work was completed successfully within 0.000948 seconds
        From detached thread No. 3, The required work was completed successfully within 0.001031 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001162 seconds
```

Figure 9: Sample run #3.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.004023 seconds

Processes solution:
    Process-based approach with 4 processes: 0.001625 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 4 join threads: 0.001143 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From detached thread No. 1, The required work was completed successfully within 0.000919 seconds
        From join thread No. 2, The required work was completed successfully within 0.000923 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001644 seconds
        From join thread No. 3, The required work was completed successfully within 0.001651 seconds

    3- Thread-based approach with 4 detached threads: such as time of creation = 0.000056 seconds
        From detached thread No. 1, The required work was completed successfully within 0.000973 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001012 seconds
        From detached thread No. 3, The required work was completed successfully within 0.001758 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001762 seconds
```

Figure 10: Sample run #4.

Appendix C

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003371 seconds

Processes solution:
    Process-based approach with 2 processes: 0.002136 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 2 join threads: 0.001782 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From join thread No. 1, The required work was completed successfully within 0.001662 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001652 seconds

    3- Thread-based approach with 2 detached threads: such as time of creation = 0.000027 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001760 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001669 seconds
```

Figure 11: Sample run for 2 processes & threads.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003360 seconds

Processes solution:
    Process-based approach with 3 processes: 0.001610 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 3 join threads: 0.001396 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From join thread No. 2, The required work was completed successfully within 0.001259 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001219 seconds
        From join thread No. 1, The required work was completed successfully within 0.001216 seconds

    3- Thread-based approach with 3 detached threads: such as time of creation = 0.000034 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001225 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001226 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001251 seconds
```

Figure 12: Sample run for 3 processes & threads.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003423 seconds

Processes solution:
    Process-based approach with 4 processes: 0.001254 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 4 join threads: 0.001009 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From detached thread No. 1, The required work was completed successfully within 0.000917 seconds
        From join thread No. 2, The required work was completed successfully within 0.000924 seconds
        From detached thread No. 0, The required work was completed successfully within 0.000915 seconds
        From join thread No. 3, The required work was completed successfully within 0.000925 seconds

    3- Thread-based approach with 4 detached threads: such as time of creation = 0.000044 seconds
        From detached thread No. 1, The required work was completed successfully within 0.000942 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001037 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001744 seconds
        From detached thread No. 3, The required work was completed successfully within 0.001760 seconds
```

Figure 13: Sample run for 4 processes & threads.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003413 seconds

Processes solution:
    Process-based approach with 5 processes: 0.001804 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 5 join threads: 0.001535 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From detached thread No. 1, The required work was completed successfully within 0.000759 seconds
        From join thread No. 3, The required work was completed successfully within 0.000738 seconds
        From join thread No. 2, The required work was completed successfully within 0.000759 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001398 seconds
        From join thread No. 4, The required work was completed successfully within 0.001399 seconds

    3- Thread-based approach with 5 detached threads: such as time of creation = 0.000049 seconds
        From detached thread No. 1, The required work was completed successfully within 0.000741 seconds
        From detached thread No. 3, The required work was completed successfully within 0.000756 seconds
        From detached thread No. 2, The required work was completed successfully within 0.000817 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001346 seconds
        From detached thread No. 4, The required work was completed successfully within 0.001359 seconds
```

Figure 14: Sample run for 5 processes & threads.

```
QOSSAY RIDI 1211553
Naive solution:
    Naive approach: 0.003727 seconds

Processes solution:
    Process-based approach with 6 processes: 0.001621 seconds
        The result is identical to that of the naive approach

Threads solution:
    1- Thread-based approach with 6 join threads: 0.001412 seconds
        The result is identical to that of the naive approach

    2- Thread-based approach with mix join & detached threads
        From join thread No. 5, The required work was completed successfully within 0.000797 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001098 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001108 seconds
        From join thread No. 3, The required work was completed successfully within 0.001125 seconds
        From join thread No. 4, The required work was completed successfully within 0.001133 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001174 seconds

    3- Thread-based approach with 6 detached threads: such as time of creation = 0.000069 seconds
        From detached thread No. 3, The required work was completed successfully within 0.000789 seconds
        From detached thread No. 4, The required work was completed successfully within 0.001054 seconds
        From detached thread No. 0, The required work was completed successfully within 0.001096 seconds
        From detached thread No. 1, The required work was completed successfully within 0.001123 seconds
        From detached thread No. 2, The required work was completed successfully within 0.001157 seconds
        From detached thread No. 5, The required work was completed successfully within 0.001168 seconds
```

Figure 15: Sample run for 6 processes & threads.