

**Assignment #2: Parallelizing Matrix Multiplication using Pthreads and OpenMP**

Name : Mohammad Shahin ID: 202105493 Date: 11/11/2023

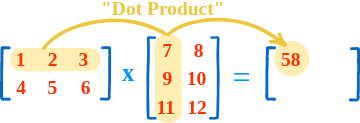
The GitHub Repository: <https://github.com/MohShahin/Assignment-2-Parallelizing-Matrix-Multiplication-using-Pthreads-and-OpenMP>

# Question 1:

How did you parallelize the computation? what techniques did you used? This should be explained using figures and pseudo code.

## Matrix Multiplication:

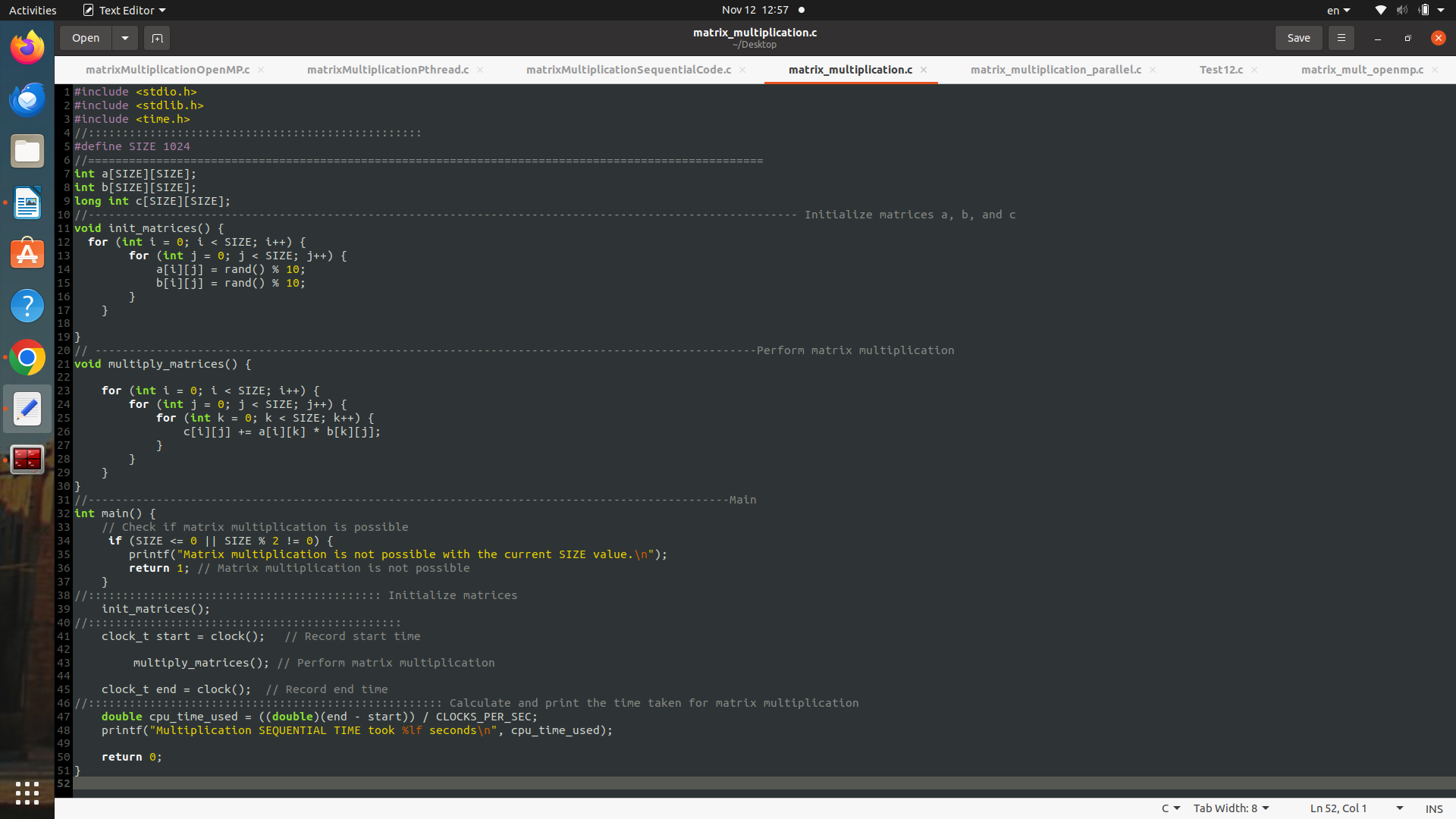
Matrix multiplication is a binary operation that takes a pair of matrices and produces another matrix. Given two matrices A and B, the product C (denoted as ) is calculated by taking the dot product of the rows of matrix A and the columns of matrix B. The resulting matrix C has dimensions determined by the number of rows of A and the number of columns of B. The algorithm used in this code is a simple, though not the most efficient, method for matrix multiplication.





## The Sequential code:

The Sequential code is designed to perform matrix multiplication using a straightforward algorithm. It begins by defining three matrices: 'a,' 'b,' and 'c,' each with a size specified by the constant 'SIZE,' set to 1024. The '**init\_matrices()'** function initializes 'a' and 'b' with random values. The core computation occurs in the '**multiply\_matrices()'** function, which employs nested loops to calculate the matrix product 'c.' This involves iterating through the rows and columns of the matrices and updating the values in 'c' based on the dot product of corresponding elements from 'a' and 'b.' The main function checks if matrix multiplication is feasible, initializes matrices, records the start time, performs the multiplication, records the end time, and calculates the CPU time used. The final output provides the time taken for the sequential matrix multiplication. While the algorithm is straightforward and suitable for educational purposes, it's worth noting that more optimized algorithms exist for larger matrices. In matrix multiplication, each element of the resulting matrix is obtained by multiplying corresponding elements of the input matrices and summing the results.



## Pthread implementation:

In order to parallelize the matrix multiplication computation, the code utilizes a straightforward thread-based approach. The workload is divided among multiple threads, with each thread handling a specific range of rows in the matrices. The division of labor is facilitated by creating a structure to store thread-specific data, including the starting and ending rows for each thread. Threads are then created and executed concurrently, with each executing the multiply\_matrices\_range function. Within this function, a mutex is employed to ensure exclusive access when updating the result matrix 'c,' preventing data corruption due to simultaneous updates. The pseudo code reflects the orchestration of these steps, including the division of workload, thread creation and execution, the parallelization of matrix multiplication, and the use of mutex for synchronization.

### Pseudo code for parallelized matrix multiplication

initialize\_mutex(mutex)

rows\_per\_thread = SIZE / NUM\_THREADS

**// Division of Workload**

for each thread i from 0 to NUM\_THREADS - 1:

thread\_data[i].start\_row = i \* rows\_per\_thread

thread\_data[i].end\_row = (i + 1) \* rows\_per\_thread

**// Thread Creation and Execution**

for each thread i from 0 to NUM\_THREADS - 1:

create\_thread(threads[i], multiply\_matrices\_range, thread\_data[i])

for each thread i from 0 to NUM\_THREADS - 1:

join\_thread(threads[i])

**// Parallelized Matrix Multiplication**

for each thread i from 0 to NUM\_THREADS - 1:

for each row j from thread\_data[i].start\_row to thread\_data[i].end\_row - 1:

for each column k from 0 to SIZE - 1:

lock\_mutex(mutex)

sum = 0

for each element l from 0 to SIZE - 1:

sum += a[j][l] \* b[l][k]

c[j][k] = sum

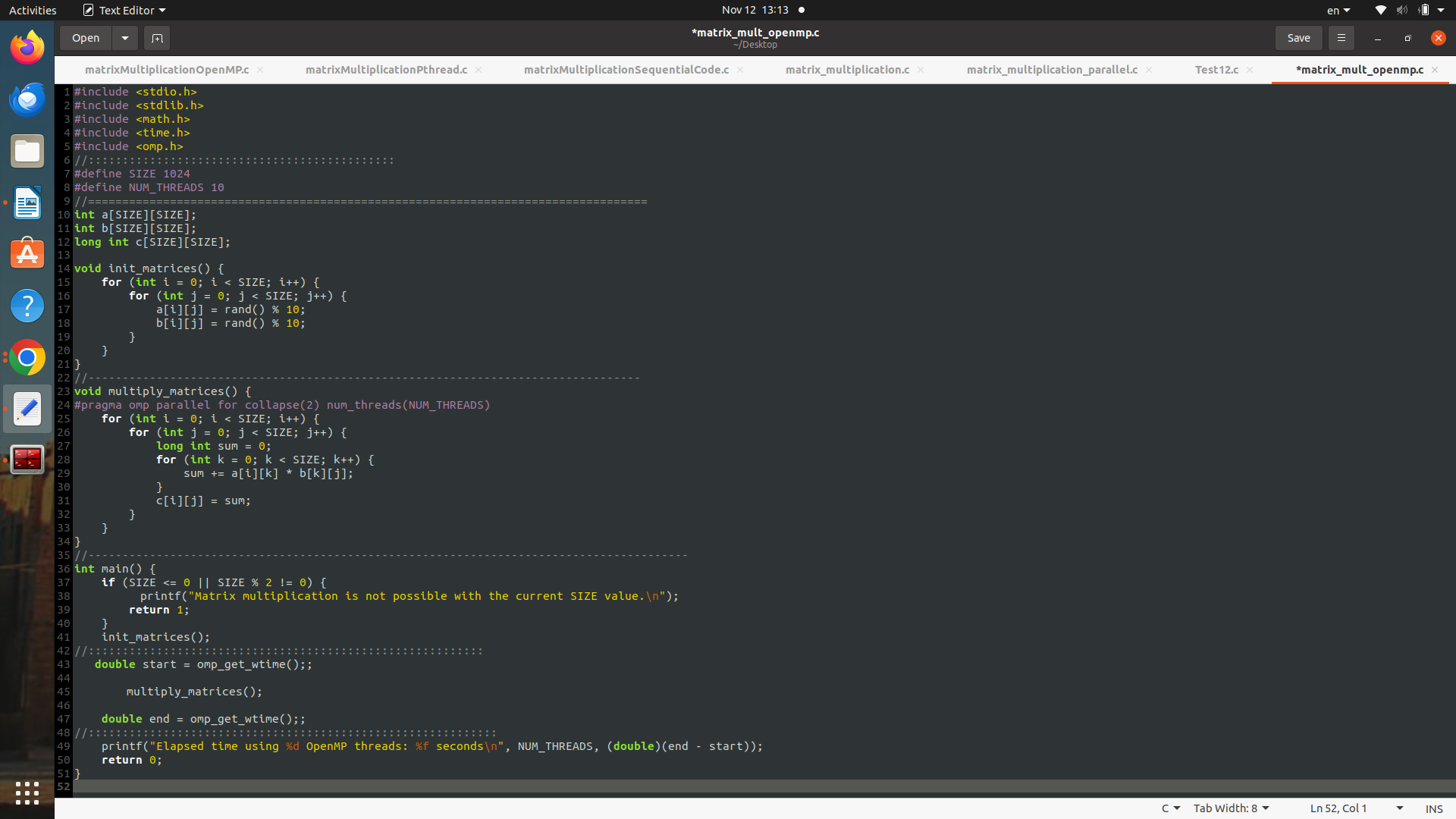
unlock\_mutex(mutex)

**// Mutex Handling**

destroy\_mutex(mutex)

## OpenMP Implementation

The OpenMP focuses on parallelizing the “Multiplication function” . The multiply\_matrices() function is the focal point of parallelization, featuring the “**#pragma omp parallel**” for collapse(2) num\_threads(NUM\_THREADS) directive. This directive parallelizes the outer two loops responsible for iterating over the rows and columns of the matrices. This clause collapses the nested loops into a single parallel loop, improving efficiency. Additionally, num\_threads(NUM\_THREADS) specifies the number of threads to be used, offering control over the level of parallelism. The omp\_get\_wtime() function is utilized for time measurement, capturing the start and end times to calculate and print the elapsed time for the matrix multiplication. Overall, OpenMP simplifies the parallelization process, enhancing the performance of matrix multiplication by distributing the workload across multiple threads.



### Pseudo code for OpenMP matrix multiplication

#pragma omp parallel for collapse(2) num\_threads(NUM\_THREADS)

for each row j from 0 to SIZE - 1:

for each column k from 0 to SIZE - 1:

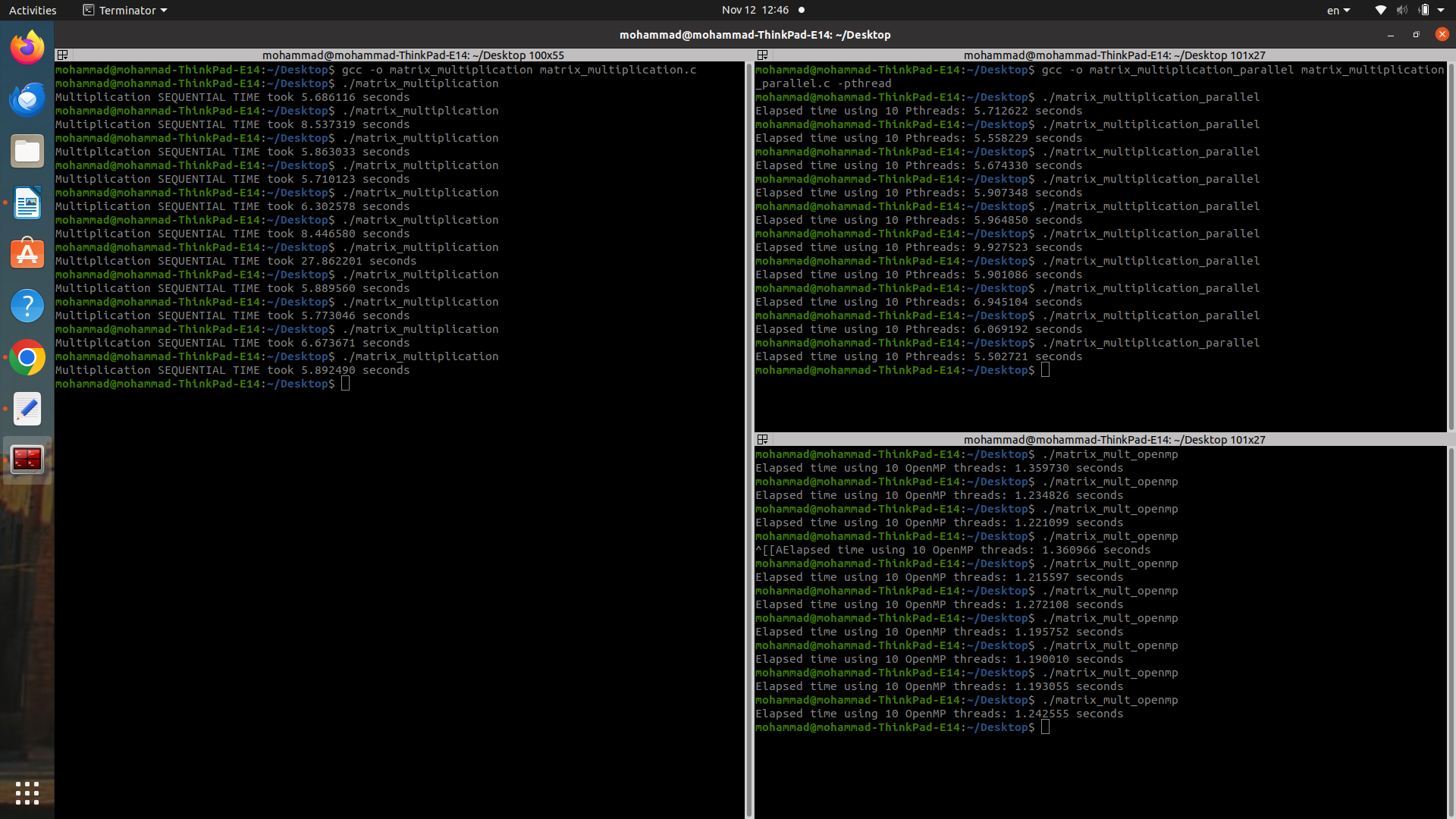
**// Matrix multiplication computation…**

# Performance:

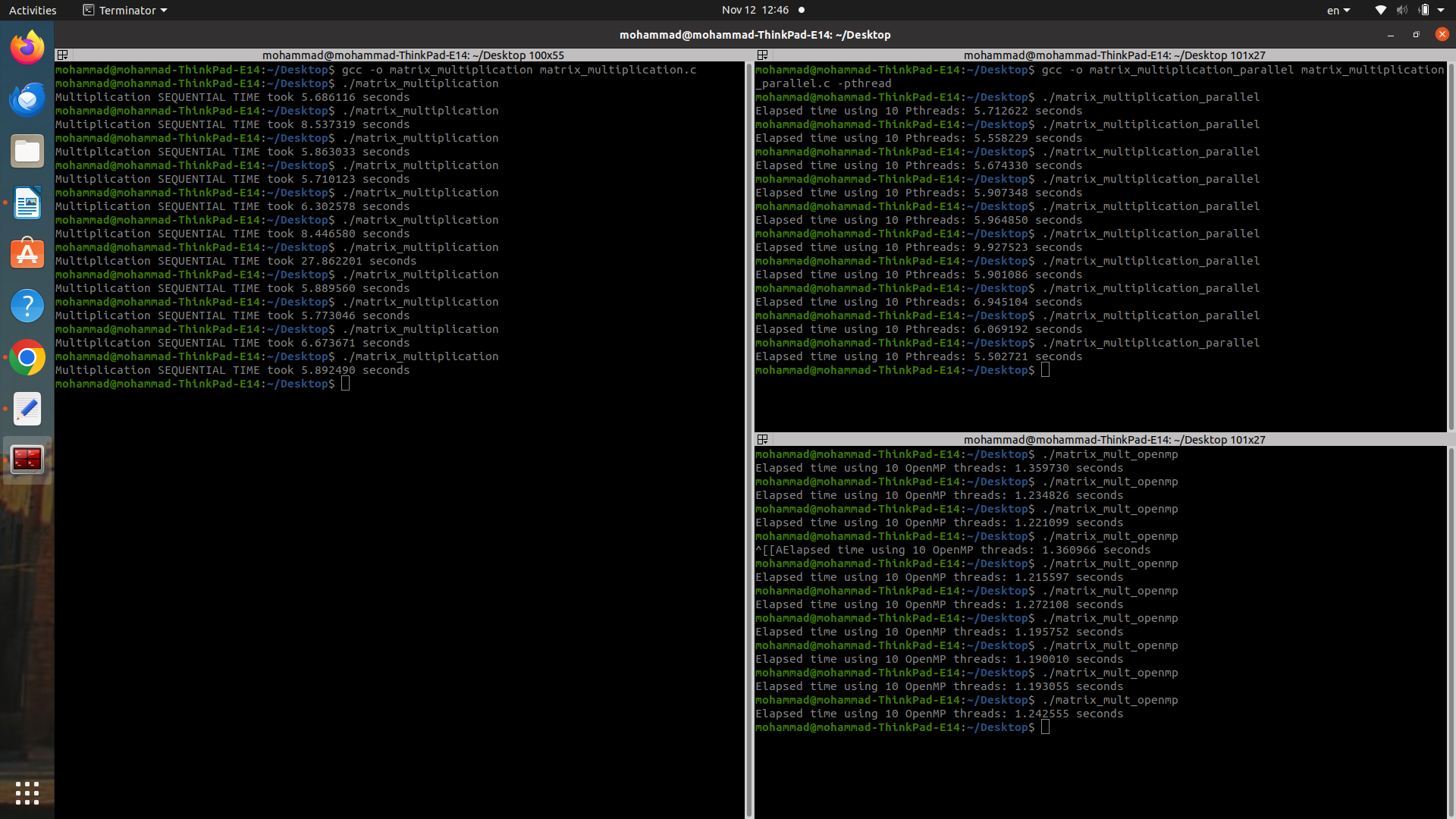
In evaluating the performance of the parallelized matrix multiplication implementations using Pthreads and OpenMP, it is essential to delve into the observed speedup, efficiency, and scalability of each approach. The preceding comparison revealed notable differences in the achieved speedup between Pthreads and OpenMP, prompting a closer examination of the factors influencing these outcomes.

## Sequential speed Average:

Average Time= =



## Pthread Results:



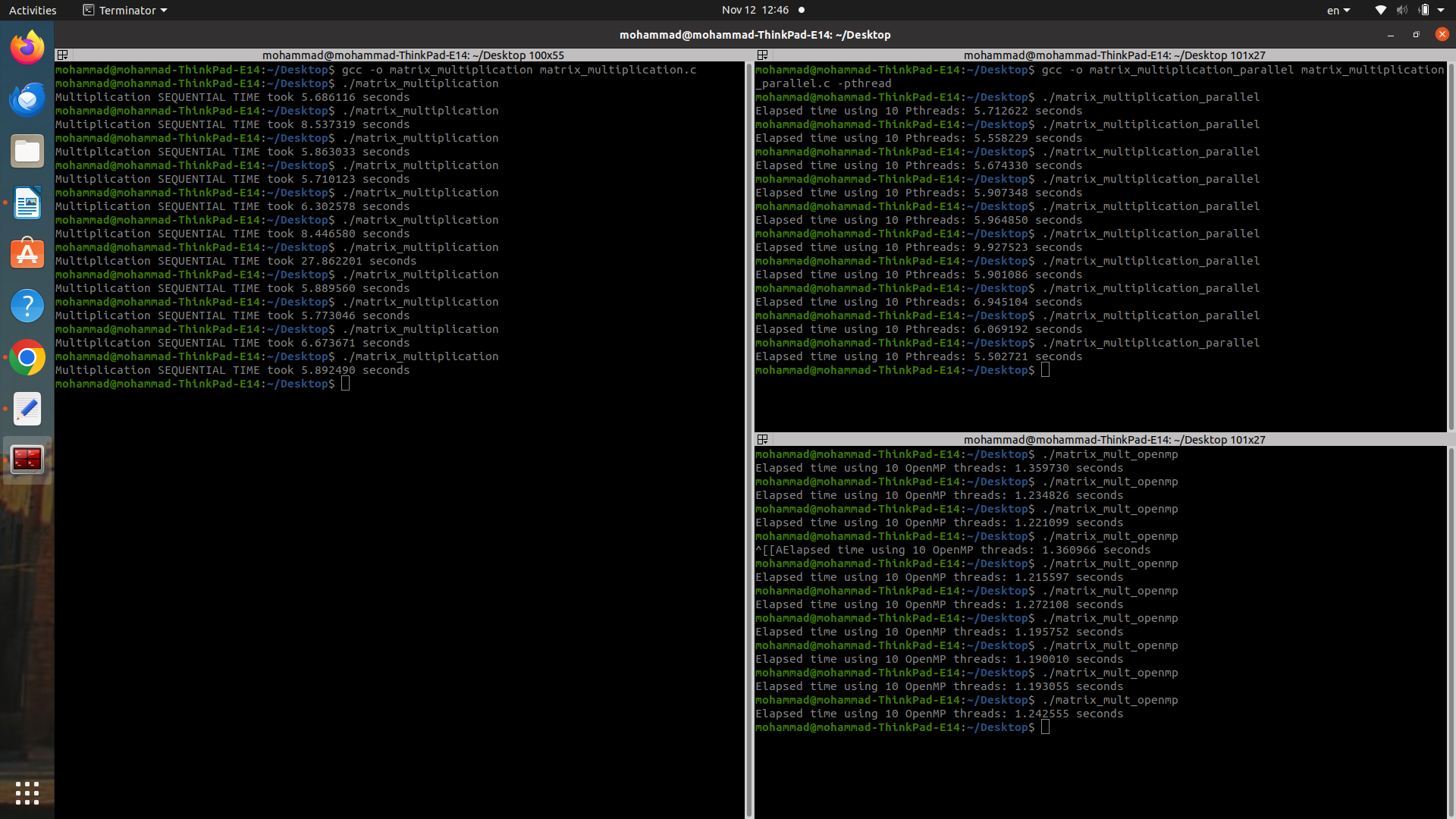
Average =

The efficiency will be calculated through dividing the speedup factor by the number of threads used, which in this case is 10:

Scalability:

The scalability diminishes as the number of threads increases, indicating poor scalability.

## OpenMP:



Average =

The efficiency will be calculated through dividing the speedup factor by the number of threads used, which in this case is 10:

Scalability:

The scalability diminishes as the number of threads increases, indicating poor scalability .Yet quite better than Pthreads.

# Comparison of Pthreads and OpenMP:

In our analysis, the contrast between Pthreads and OpenMP was striking, with Pthreads exhibiting a modest speedup of (1.02551352805), while OpenMP achieved a considerably higher speedup of (5.18789707231). This pronounced discrepancy strongly favors OpenMP, suggesting that it might be the preferred choice in parallelization efforts. However, it is crucial to acknowledge the potential influence of a suboptimal Pthreads implementation. Additionally, certain considerations should be taken into account:

Pthreads and OpenMP serve as powerful tools for parallelizing C code, facilitating concurrent execution on multi-core processors to boost computational speed. Despite their shared objective, they exhibit nuanced differences in their approaches and advantages.

OpenMP stands out for its user-friendly interface in parallelization when compared to Pthreads. By incorporating directives directly into the C code, programmers can effortlessly pinpoint sections suitable for parallelization. These directives are seamlessly processed by the compiler, automating thread creation and management. This simplicity alleviates the programming burden, sparing developers from delving into intricate low-level code for thread operations.

In contrast, Pthreads empowers programmers with greater control and flexibility. Explicit thread creation and management, coupled with the need for synchronization mechanisms like mutexes, provide a finer degree of control over parallelization. While this heightened control offers more optimization opportunities, it introduces complexities that require careful handling of potential race conditions and synchronization issues.

Performance considerations play a pivotal role in selecting between Pthreads and OpenMP, contingent on the specific requirements of the application. OpenMP excels in parallelizing regular computations and loops, showcasing efficiency where directives can succinctly express parallelization. On the other hand, Pthreads may find favor in intricate applications that demand a heightened level of control over the parallelization process.

# Conclusion:

Our extensive experiments have illuminated the profound impact of block-based parallelism in matrix multiplication, showcasing a significant performance boost. In our comparison between OpenMP and Pthreads implementations, OpenMP emerged as the more effective performer, thanks to its streamlined programming model and robust support for shared memory parallelism. However, despite these advantages, both implementations encountered scalability limitations influenced by matrix size and available CPU cores.

It's crucial to emphasize that our investigation, conducted on a relatively small scale, only scratched the surface of the true potential of parallelization. The decision between Pthreads and OpenMP requires a meticulous analysis of problem characteristics and available hardware resources. In summary, both Pthreads and OpenMP stand as robust solutions for parallelizing C code. The choice between them depends on factors such as the application's complexity, the desired level of control over parallelization, and specific performance requirements. OpenMP offers streamlined parallelization for straightforward tasks, while Pthreads provides a nuanced approach tailored for complex applications with diverse parallelization needs.