**Lebanese American University**

**Department of Computer Science and Mathematics**



## CSC447 - Parallel Programming

### Assignment 2

Instructor: Hamdan Abdullatif

## Student Name: Samir Khansa

## Student ID Number: 202001781

#### Paralyzing Matrix Multiplication:

**OpenMP:**

OpenMP is an Application Programming Interface (API) and it allows our computer to use multiple threads at the same time allowing our algorithm to be executed in parallel while using multiple threads. It is commonly used in the programming languages C, C++ and Fortran. Moreover, OpenMP is an easy technique in allowing the programmers to execute their code using multiple threads.

In parallelizing our matrix multiplication code while using OpenMP, I used the OpenMP pragma omp that commands the threads to execute in parallel the specific chunk of code. In other words, it tells the compiler to let 10 threads (specified by the programmer), where every thread would be executing a part of the code in parallel. Furthermore, in the sequential code of the matrix multiplication algorithm, there are three nested loops that are iterating over the rows and columns of the matrix. For parallelizing the three nested loops I have added the pragma omp before the first loop as shown in the pseudo code below. Moreover, the private section would be informing the threads that each thread should have its own copy of the variables. In other words, i ,j and k should not be common among different threads they need to have their own private variable of i, j, and k. Furthermore, the shared variable that are the matrix\_a, matrix\_b, and result\_matrix will be shared among all the threads. In other words, all the threads will be able to adjust information in those matrix in parallel. In conclusion, when the code starts running, the OpenMP would create threads according to the number that the programmer assigned, where each thread would execute a copy of the loop. Moreover, the threads are assigned iterations of the outer loop based on the schedule clause. The schedule clause would allow us to control how the iteration of this loop are distributed among the threads. Furthermore, OpenMP uses static scheduling. In other words, each threads would assigned be assigned a specific block of iterations for execution. In other words, I will be dividing the total size of the iterations with the number of threads then I will assign each thread a specific chunk of the iterations so that they would be running in parallel.

**#progma omp parallel for private(i,j,k) shared (matrix\_a, matrix\_b, result\_matrix);**

**for(i=0;i<1024;i++){**

**for(j=0;j<1024;j++){**

**for(k=0;k<1024;k++){**

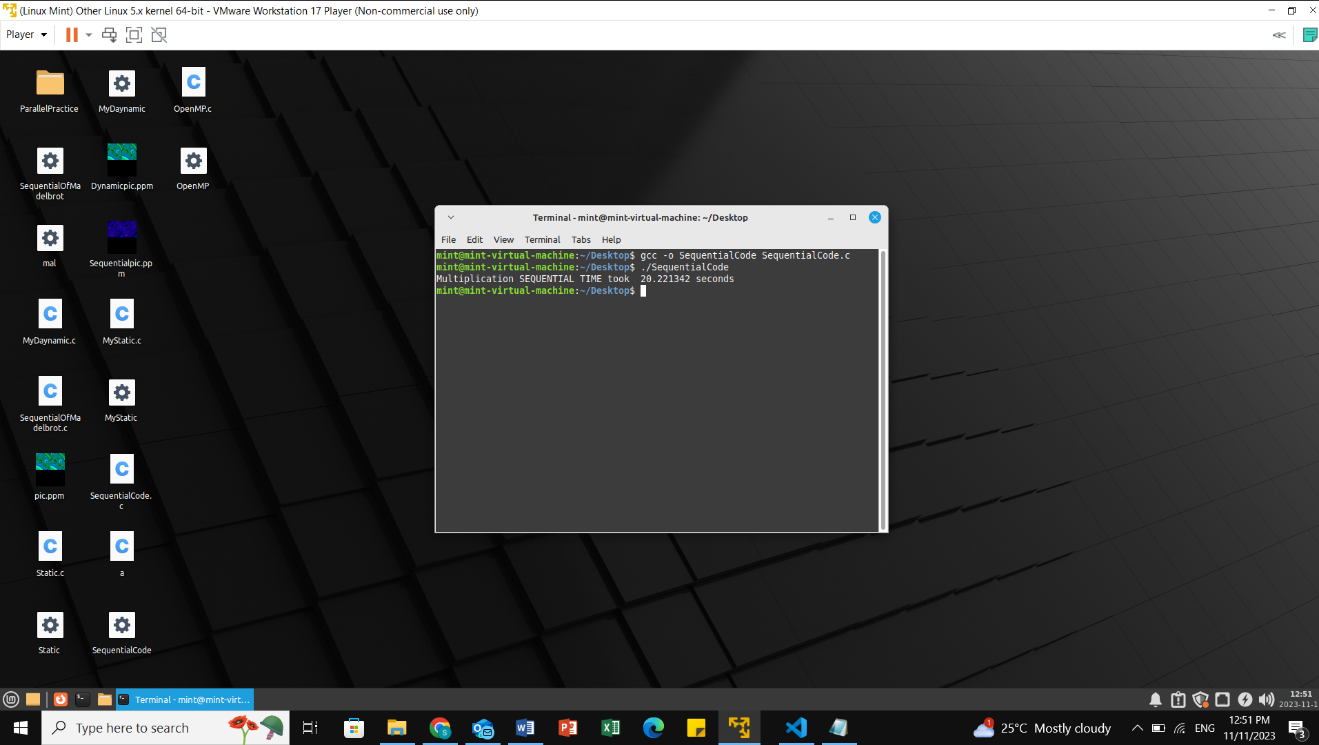
**result\_matrix[i][j]+= matrix\_a[i][k] \* matrix\_b[k][j];**

**}**

**}**

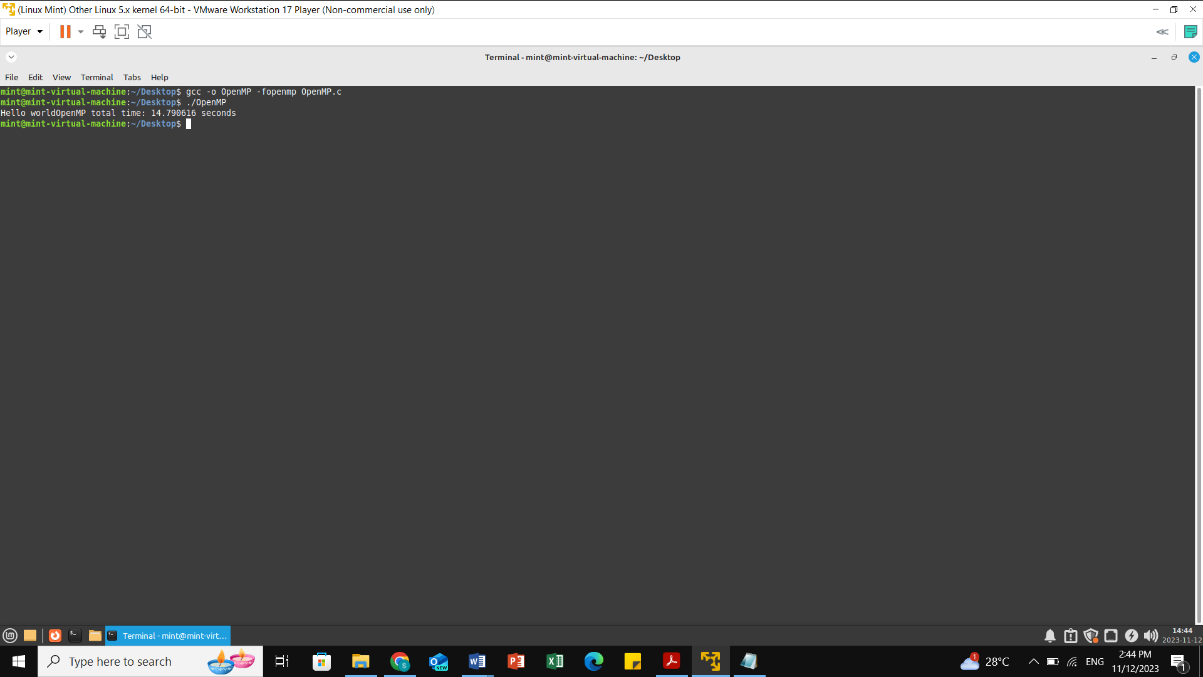
**}**

Now to calculate the Speedup factor, efficiency and Scalability we need to first identify what is the execution time of the sequential code and the parallelized code.



The sequential Code total execution time is 20.221342. The matrix in my sequential code is the size of 1024x1024 that’s why the code is taking a significant amount of time.

The Parallelized code total time would be 14.790616

****

The speedup ratio measures how quickly one process executes compared to several processes. It stands for the performance in using one processor from many processors. Calculating the speedup factor is as follows:

The SpeedUp factor can be calculated by dividing the sequential time with the parallel time.

##### SpeedUp factor=Sequential time/ parallel time= 20.221342/14.790616= 1.367173754

Efficiency refers to how well a system uses its available processors. It is described as the speedup divided by the total number of processors being utilized.

**Efficiency=SpeedUp Factor/Number Of Threads=**1.367173754**/8\*100= 17.0896%**

Scalability measures how well a program can take advantage of increased resources to solve large or more complex problems without a significant decrease in performance or efficiency.

**Scalability = (Sequential time / (Number of threads\* Parallel time))= 20.221342/8\***14.790616**=0.1708961792**

**PThreads:**

The Sequential code was also paralyzed using Pthread library that helps in providing a way in creating multiple threads for the code to be executed. Moreover, creating multiple threads is beneficial to dividing the algorithm into multiple smaller chunks and each thread would take a chunk and work on it in parallel. Furthermore, our matrix multiplication algorithm was divided into multiple chunks of data and divided across the threads to work on it in parallel. We defined the number of threads using the variable num\_threads. The matrix 1024\*1024 is divided equally across the threads. Furthermore, two loops with the range of the number of threads after the second loop we will create a thread and will enter the slave function matrix\_multiply\_block which is responsible of multiplying the matrix with a specific block size. The slave function will take an argument called arg that is the id number of the thread. Furthermore, is calculates the its portion of the matrix independently from other. In other words, we divide the overall number of rows in the matrix by the total number of threads and assigns each thread its row to calculate. Finally for every row in the thread it calculates it and adds the result to the result matrix. When all the treads had finished their calculations, the main would wait for all the threads to finish using the function pthread\_join, and finally it would print the time taken to calculate the result matrix. This Is a psudo code below

Every thread is independent from the computation of the other threads, hence their will be no two threads doing the same part in the matrix.

**for each block row i of Matrix\_A**

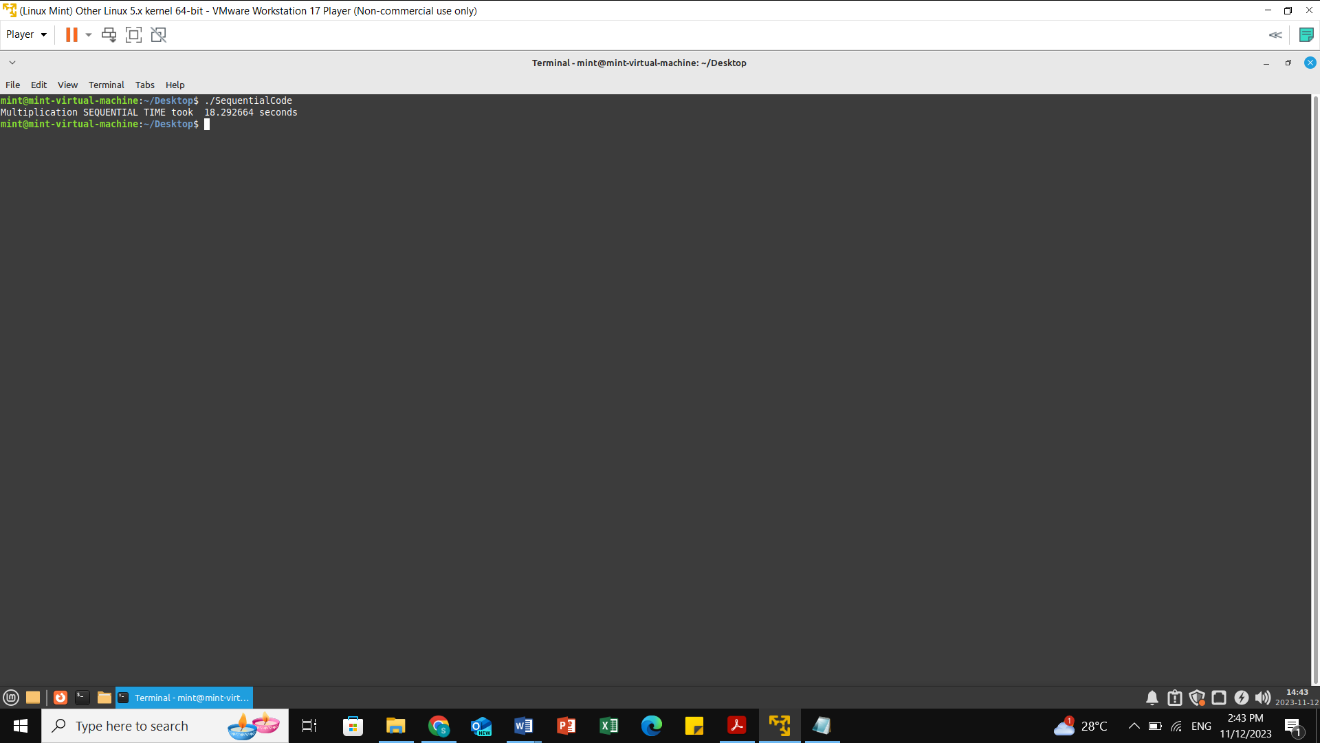
**for each block column j of Matrix\_B**

**create a new thread then we will compute the product of blocks i,k and k,j then store it in the result matrix**

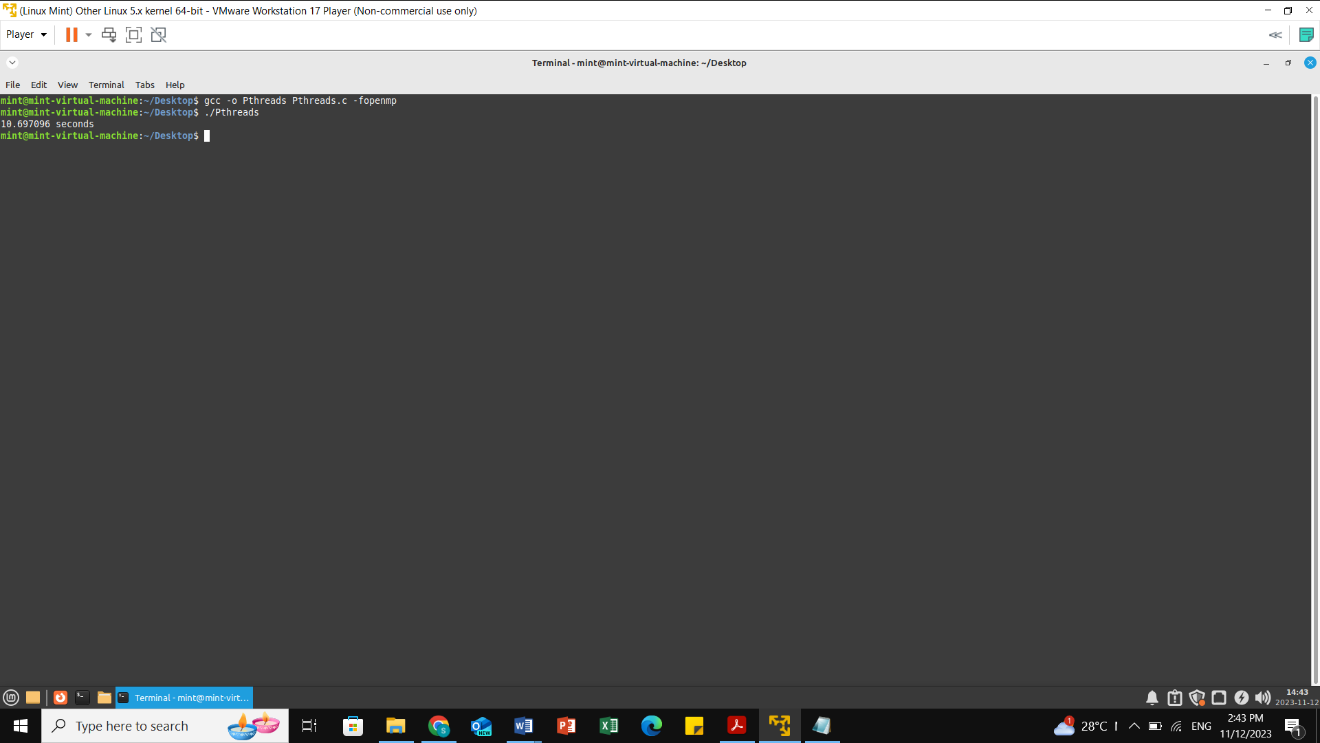
**Finally we will wait for all the threads to finish.**

In conclusion this algorithm is using a specific strategy in paralyzing algorithm by dividing the code evenly across multiple threads. In the main() the algorithm is creating a list of threads and each thread is assigned a specified range of rows to execute. The identifier is assigned the range of rows and columns assigned to each thread. The function matrix\_multiply\_block is called with the id of the thread to calculate its part of the matrix. Finally after all the threads have finished executing the total time is printed in seconds.

The below time shows the sequential time taken to compute a 1024x1024 matrix.



The below time shows the total time taken to execute the Pthread code of 1024x1024 matrix, which took a total time of 10.697096 seconds.



The speedup ratio measures how quickly one process executes compared to several processes. It stands for the performance in using one processor from many processors. Calculating the speedup factor is as follows:

The SpeedUp factor can be calculated by dividing the sequential time with the parallel time.

##### SpeedUp factor=Sequential time/ parallel time= 20.221342/10.697096= 1.890358093

Efficiency refers to how well a system uses its available processors. It is described as the speedup divided by the total number of processors being utilized.

**Efficiency=SpeedUp Factor/Number Of Threads=1.890358093/8\*100= 23.62947617%**

Scalability measures how well a program can take advantage of increased resources to solve large or more complex problems without a significant decrease in performance or efficiency.

**Scalability = (Sequential time / (Number of threads\* Parallel time))=20.221342/8\***10.697096**=0.2362947617**

**3. GithHub Link**

**4. Comparison of the two implementations (Pthread and OpenMP)**

Both implementation techniques Pthread and OpenMP are very similar in paralyzing the C code for several reasons. The first reason is that both OpenMP and Pthread manage multiple threads for execution in parallel. In other words, it gives the programmer an advantage in executing a code in parallel using multi threads that would increase the time taken to execute the code. Similarly, another similarity is that they both allow for data to be shared between threads. Finally both OpenMP and Pthreads are usable across different systems.

Moreover, multiple differences exists between OpenMP and Pthread where OpenMP is simpler than Pthread in using in penalization techniques using threads. In other words, while using OpenMP, the programmer is able to use multiple commands and built in libraries in the programming language C to specify what parts should be paralyzed and which parts should not be paralyzed. Those libraries are processed with the compiler which would allow for the threads to execute in parallel. In conclusion, this helps every programmer in paralyzing their code which they do not need to write low-level code or would have to manage the threads.

However, Pthreads has an advantage with providing control and is able to change more easily than OpenMP. In other words, while using Pthread the programmer is creating and managing the threads. However, the programmer should also take into consideration the synchronization access to shared memory, where for example two threads cannot increment a variable at the same time, to solve this issue he would have to work on mutex or other synchronization primitives. Hence, writing code using Pthreads is more challenging for the programmer since there could be multiple issues that he is facing including the race condition and other synchronization issues. However, the programmer also has more control through the penalization technique that is being through the algorithm.

Moreover, between OpenMP and Pthreads, the better performance is according to the requirements of the algorithm. Furthermore, OpenMP is easier to implement that Pthread yet according to runtime Pthreads was faster in executing a matrix of 1024x1024 size than using OpenMP. However, there is a possibility that if the matrix was smaller OpenMP would be faster that Pthread. Finally, Pthreads may be more suitable for a complex algorithms while OpenMP is more suitable to less advanced algorythms or for programmers who are interested in parallizing their code in a simpler way.

In conclusion, Pthreads and OpenMP are very important tools that are used in the programming language C by programmers in paralyzing their code to having a faster execution time. In chosing the best choice depends on the multiple circumstances, the complexity of the algorithm and how much the programmer wants control over the parallelization technique.

**Conclusion:**

In Conclusion, our results are evidence that the penalization of matrix multiplication can improve the overall performance in a significant manner. The Pthread implementation had better performance than OpenMP implementation due to it providing more control to the programmer. However, both implementation are limited to the matrix size and the number of available CPU. In conclusion, in choosing the better penalization technique is based on the requirements of the algorithm and the hardware resources available. Furthermore, Pthreads and OpenMP have their specific strength and weaknesses, where choosing the parallelization technique with the right algorithm would lead to the best performance.­