Matrix Multiplication with Multi-threading

Adit Danewa, 2019B5A71378H
Anirudh Singh, 2019B5A70948H
Bharath Variar, 2019B5A70930H
Kanika Gandhi, 2019B5A71080H
Shlok Mongia, 2019B2A71527H
Shyam N V, 2020A7PS2081H

Submission for Project 2
CS F372: Operating Systems
Department of Computer Science and Information Systems
BITS Pilani, Hyderabad Campus
November - December 2022

Contents

1 Introduction

Matrix multiplication is a requirement in many practical applications today. Optimising the speed of multiplication of arbitrarily large matrices is crucial for solving many real-life problems, such as those of AC networks in electronics and also weather forecasting.
It can be observed that the reading of rows and columns and the multiplication of said rows and columns can be done in parallel.
In this project, we aim to study the performance of a round-robin scheduler with different time quanta.

2 Reading matrix with multi-threading

Depending on the dimension of the given matrix, each thread is assigned a particular number of rows and columns it has to read. The workload of reading the rows is divided equally among the threads available.

2.1 Handling Shared Memory

We have allocated 8,000,000 bytes for each matrix and 4,000 bytes each for the read status of rows and columns, hence allowing the process to seamlessly read and multiply two 1000 $\times$ 1000 matrices of type long long int.

3 Multiplication with multi-threading

The following approach is taken in order to avoid race conditions:

- In the multi-threading function, each row and its corresponding column are put in a spin-lock.
- Before $P_1()$ starts its execution, the read status of all the rows and columns is initialized to 0 (indicating that none of the rows or columns has been read).
- So until both the rows and columns are fully read, the multi-threading function is held in a spin-lock, and the moment the read status of both the desired row and columns are set to 1, the multiplication takes place.
- For multiplication, each element of the resultant matrix is assigned an individual thread. If the resultant matrix has a thread equal to the number of elements, then each element gets one thread for its multiplication.

4 Scheduling

The Round Robin Scheduler alternates (in a fixed quantum of time 1ms or 2ms) between processes P_1 and P_2. Two different system calls are used to achieve this are:

- **kill()**:
  - Used to send a CONTINUE signal to the process (for example, the scheduler tells $P_1()$ to resume its executions).
  - Used to send a STOP signal to the process (for example, the scheduler tells $P_1()$ to pause its execution after a fixed quantum of time).
  - Identifies the process and sends the appropriate signal based on the `pid` parameter passed as the first argument.
- **usleep()**:
  - Used to suspend (put to sleep) the round-robin scheduler of the amount of this passed as the parameter (in μs). So after the kill() system call resumes the execution of process P_1, the Round Robin Scheduler is suspended for the specified quantum of time. Allowing process $P_1()$ to execute for the specified quanta.

Before scheduling a particular process, the scheduler checks if the process is completed by checking its corresponding flag value. If both the processes are done with their execution, i.e. the scheduler stops its execution (flag variables of both P_1 and P_2 are set to one).

5 Results

5.1 Reading

From Figure 1 we can observe that:
As the number of threads increases, we can observe a greater variation in reading performance, indicating that parallelization does not improve reading times; this can be explained by the fact that the overhead of creating threads and maintaining multiple file pointers and performing the required context switches is greater than the time saved by multi-threading.

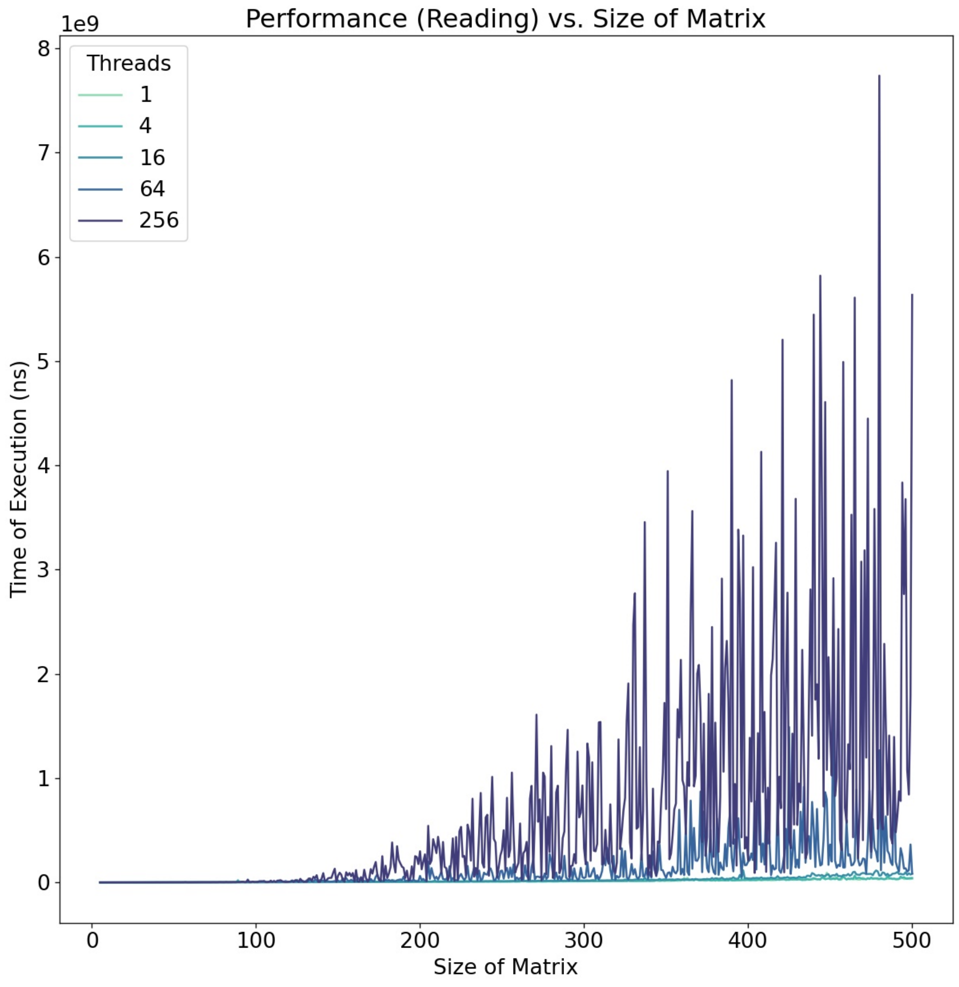

Figure 1: Performance for reading matrices

5.2 Multiplication

From figure 2 we can conclude that:

- A single thread performs better for small matrix sizes, up to 200 × 200 matrices, implying that the overhead of creating new threads is higher than the performance boost due to parallelization at these sizes.

- However, at larger sizes ($> (600 \times 600)$) the processes with the higher number of threads perform significantly better, agreeing with our hypothesis.
- It should be noted that the process with 1024 threads performs poorly compared to those with lesser threads; for matrices larger than (700×700), the 1024 threads start to show better performance.

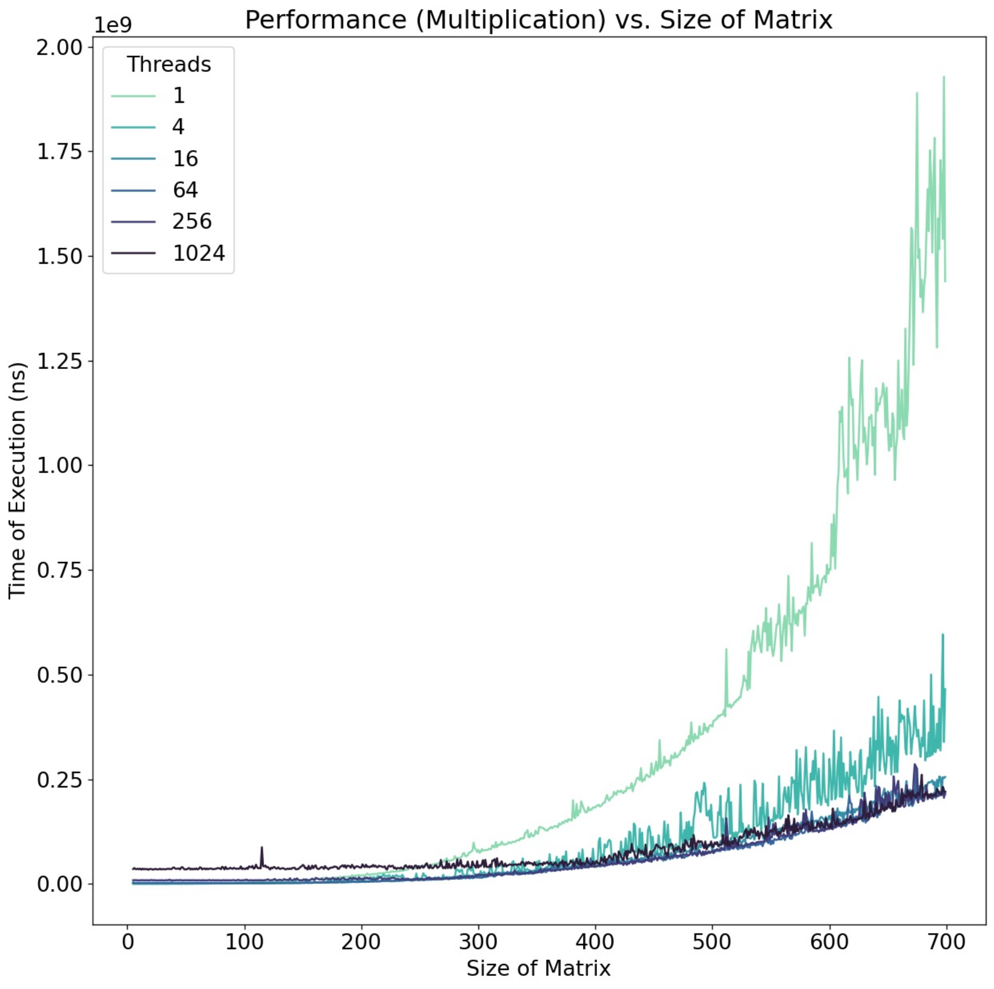

Figure 2: Performance of multiplication

5.3 Turnaround Time

From figures 3 and 4 it can be observed that:

- As matrix size increases, turnaround times for both scheduling schemes (with `q` = 1ms and 2ms) increase following the same pattern. This is due to the increased number of context switches and computations that have to be done for multiplication.
- Note: The seemingly random peaks in the graph are due to the processor of our personal computers carrying out other tasks in the background.

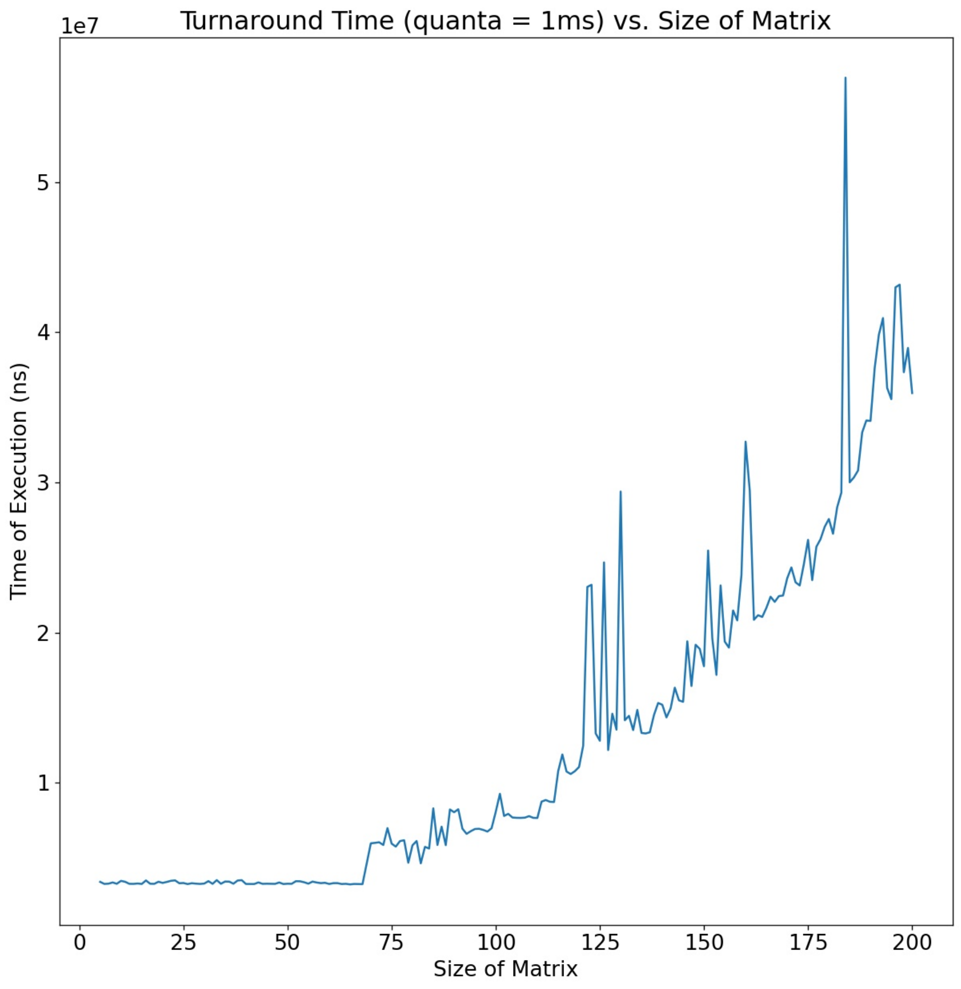

Figure 3: Turnaround time with quanta = 1ms

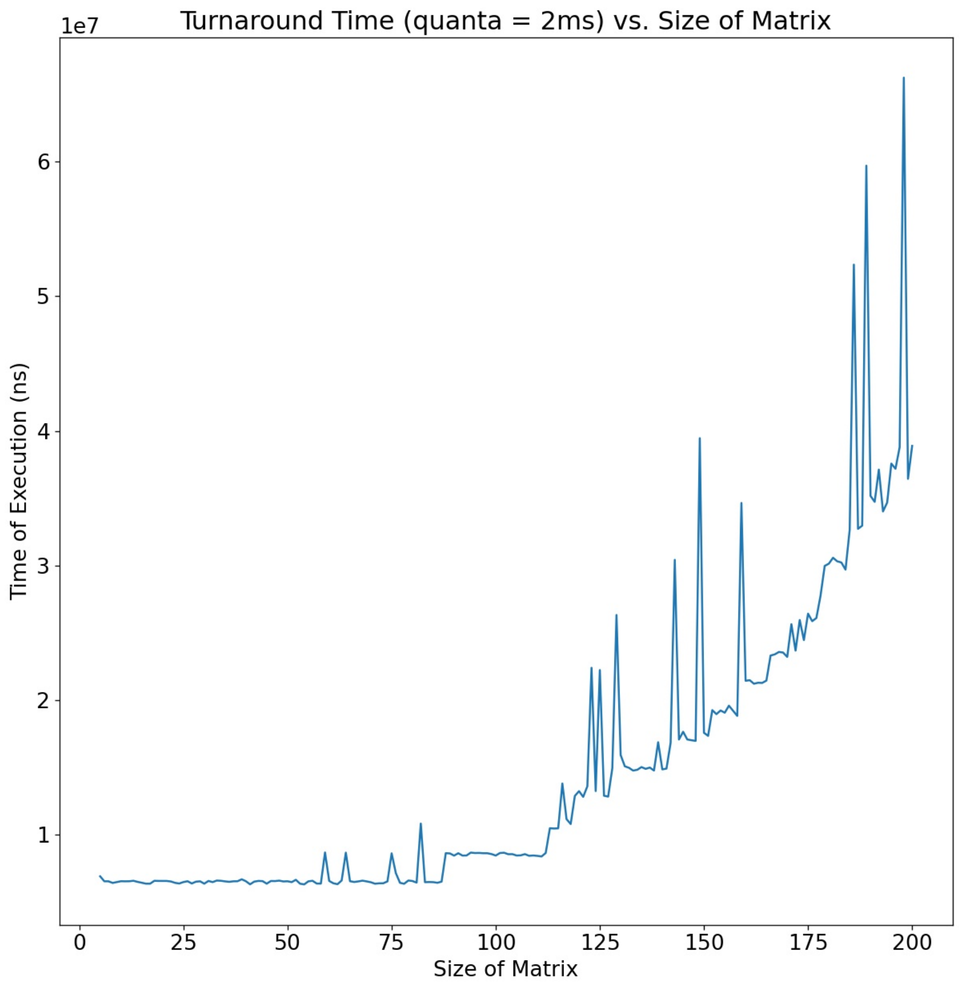

Figure 4: Turnaround time with time quanta = 2ms

5.4 Waiting Time

From figures 5 and 6 we can observe that:

- With increasing size of matrix, the waiting period for each process increases with periodic jumps, depending on the number of time quanta that it has to wait.

- For `q` $= 2$ms, the jumps are larger than that for `q` $= 1$ms since the each waiting time is twice as long.

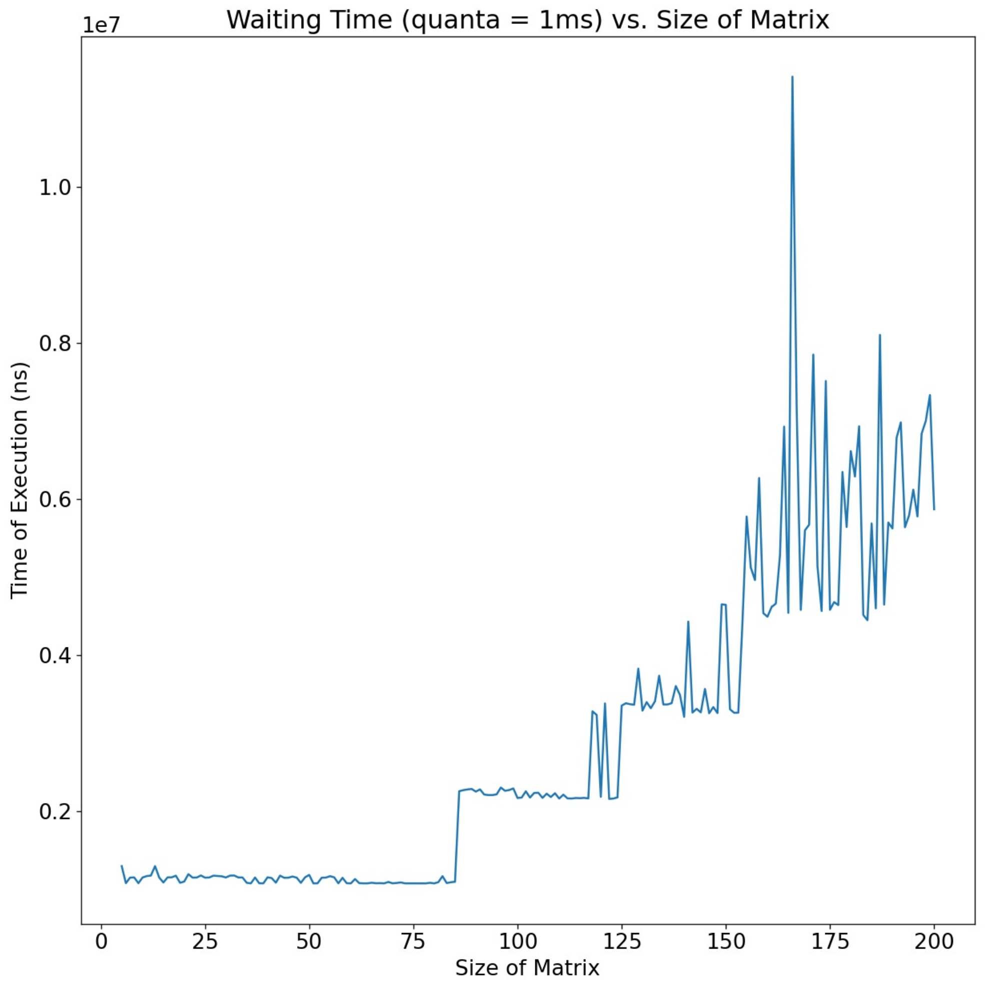

Figure 5: Waiting time with quanta = 1ms

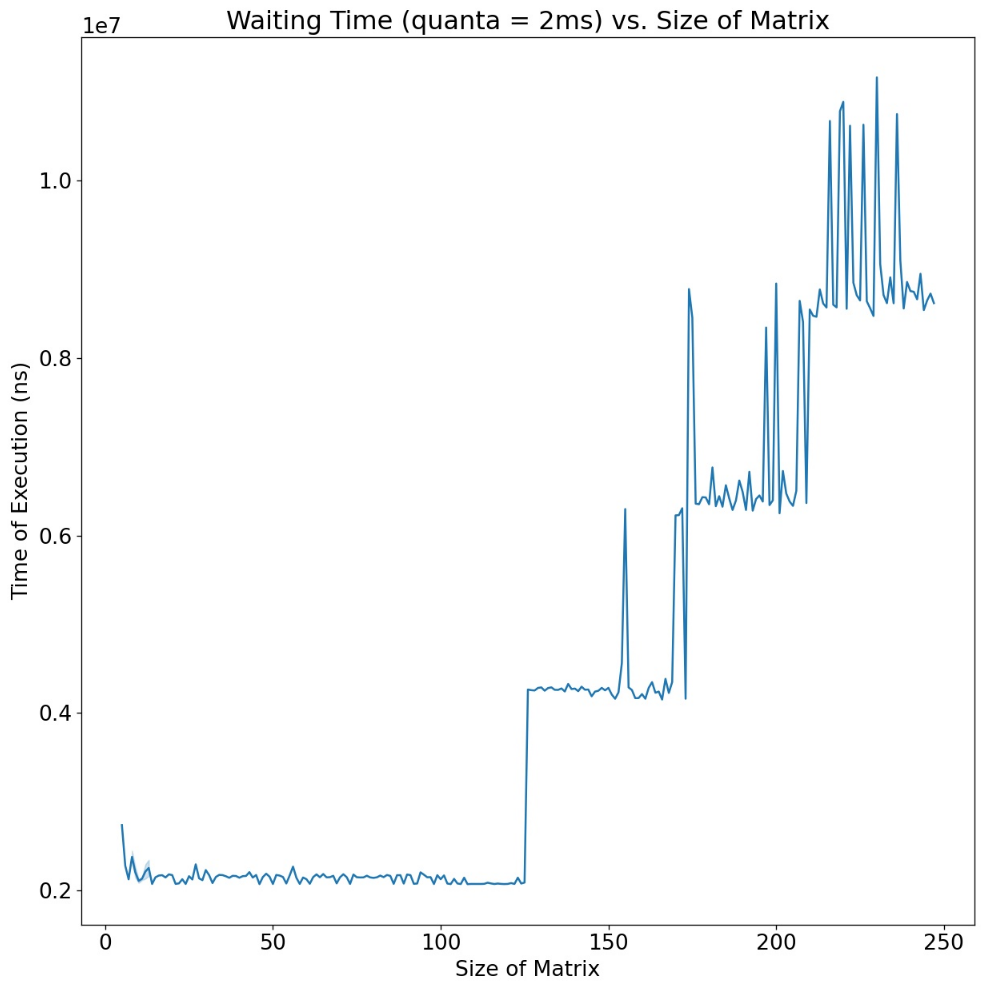

Figure 6: Waiting time with quanta = 2ms

References

[1] All the code in this report has been programmed by us, and can be found in this GitHub Repository.