Assignment 2 - Report

Plots and Analysis for P1

1.

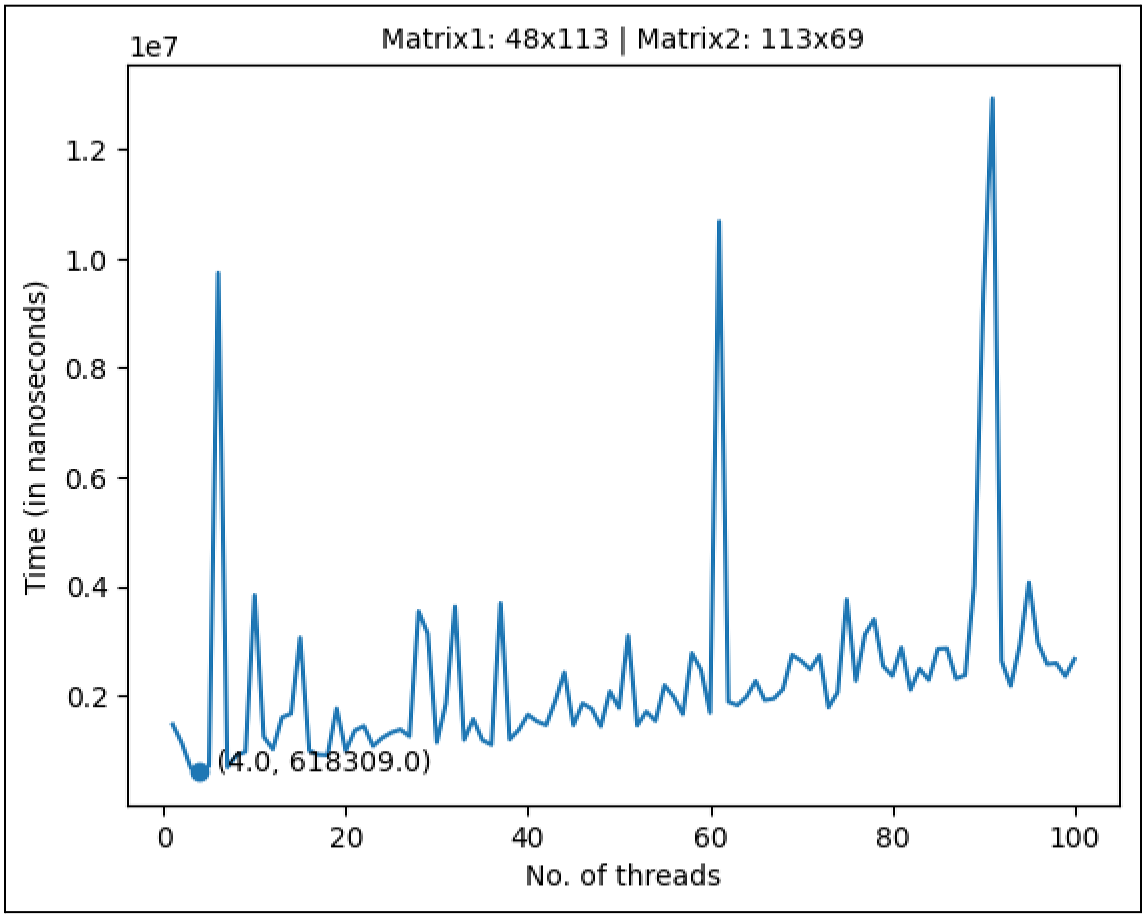


Figure 1.1

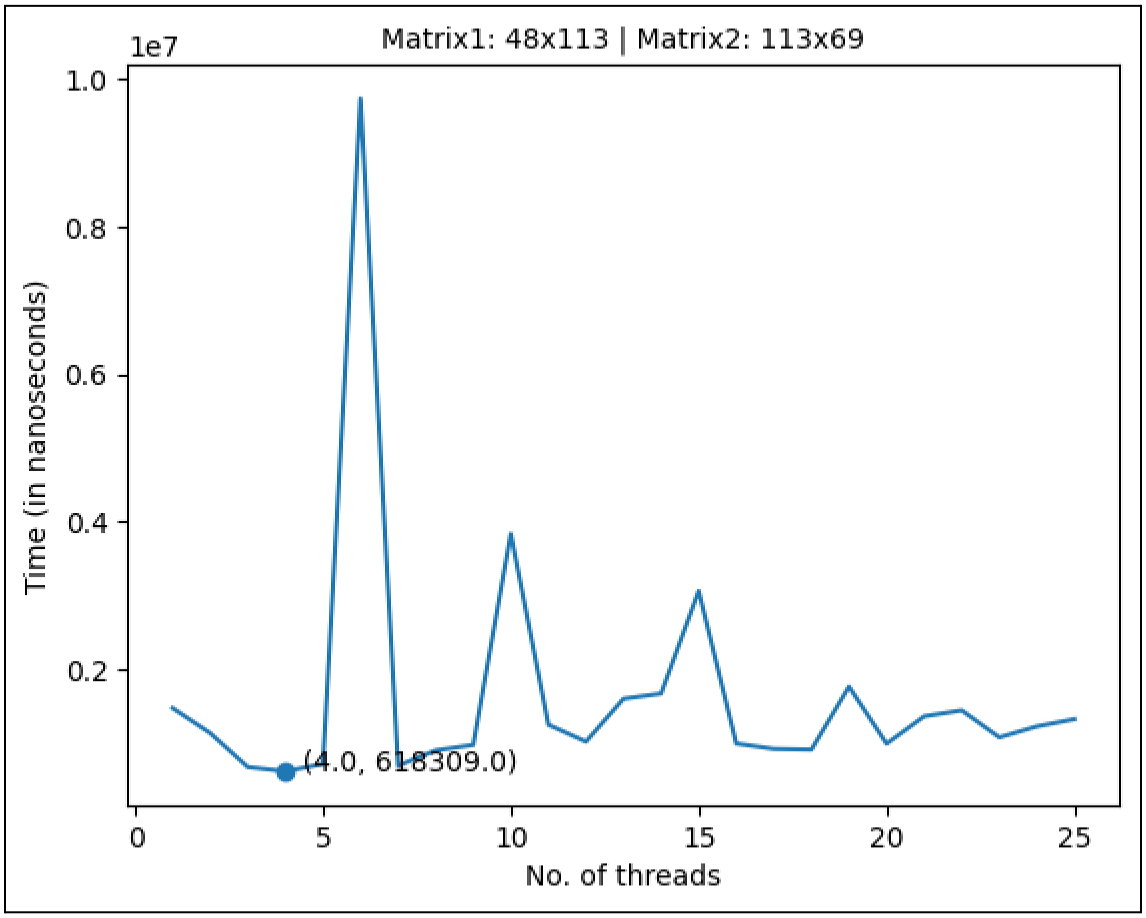


Figure 1.2

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **48 x 113** and **113 x 69** is achieved for **4** threads. The reason that for **4** threads the program runs faster compared to a lesser number of threads, is that now the program **can read more input** (values stored in the matrix) **per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can read more input per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

2.

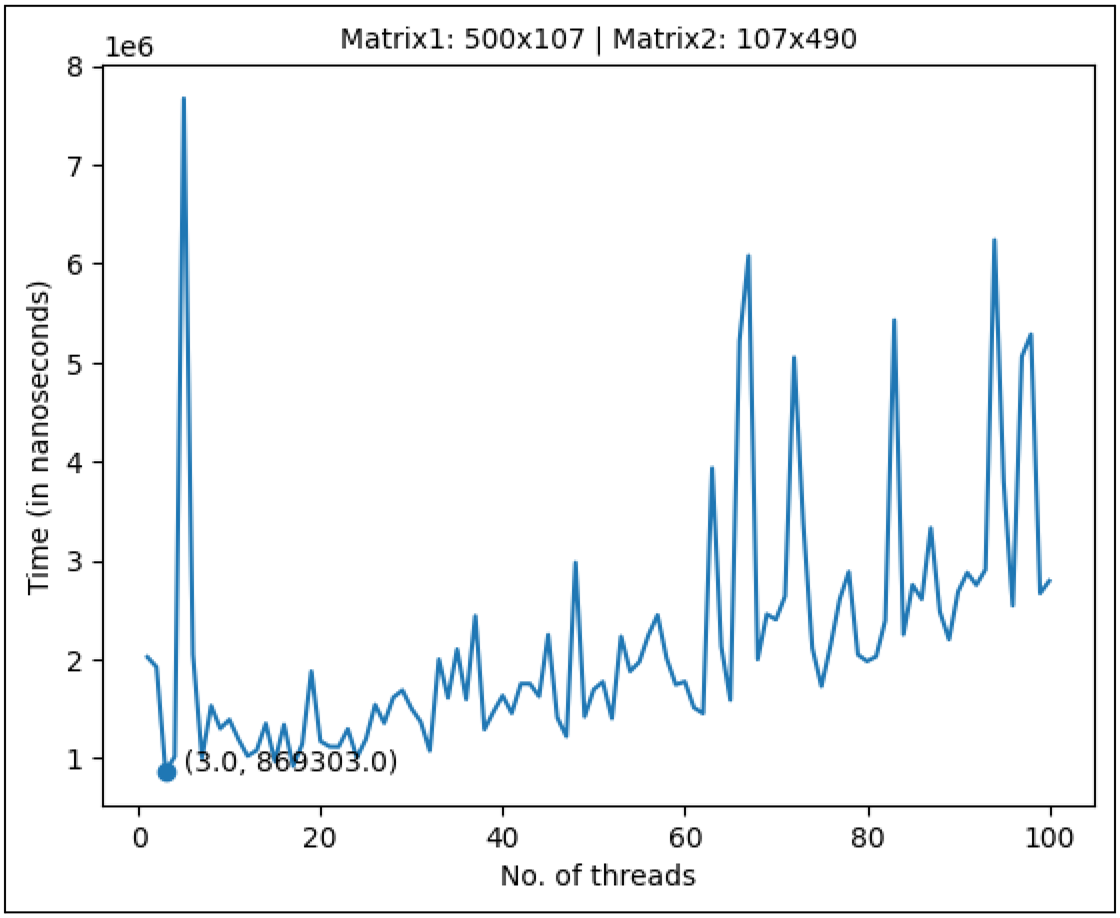

Matrix1: 500x107 | Matrix2: 107x490
1e6
8
7
6
5
4
3
2
1
Time (in nanoseconds)
(3.0, 869303.0)
0
20
40
60
80
100
No. of threads

Figure 1.3

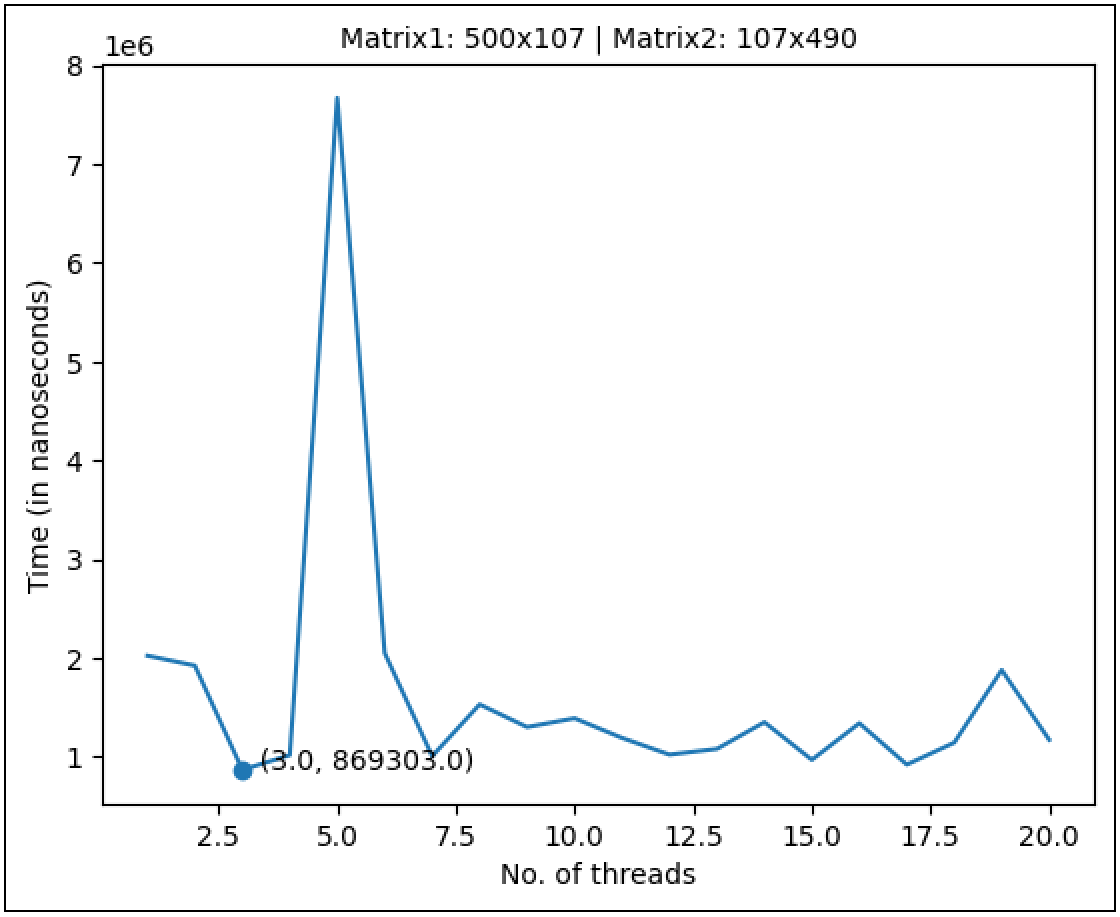


Figure 1.4

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **500 x 107** and **107 x 490** is achieved for **3** threads. The reason that for **3** threads the program runs faster compared to a lesser number of threads, is that now the program **can read more input** (values stored in the matrix) **per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can read more input per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

Analysis

On observing the above 2 cases, it can be seen that if the number of rows for the first matrix and the number of columns for the second matrix are **varied**, while the number of columns for the first matrix and the number of rows for the second are **fixed,** then there is very little change in the number of threads that achieve the minimum execution time. The reason for this may be that upon fixing the number of columns for the first matrix and the number of rows for the second, the **work to be done** by a particlar thread **would not change** as that thread would still have to read the same number of elements. This means that a practicular thread would take the **same amount of time** to finish its job, meaning that more threads would not increase the performance.

3.

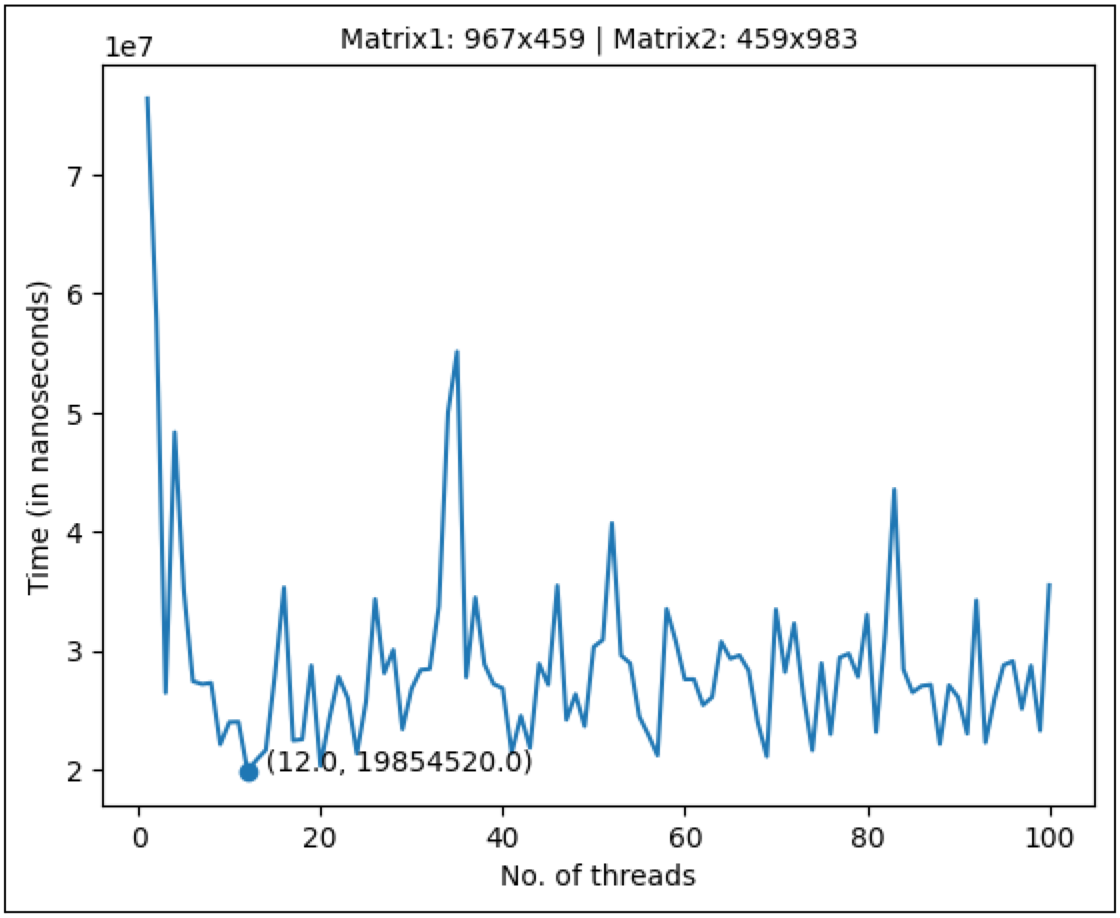


Figure 1.5

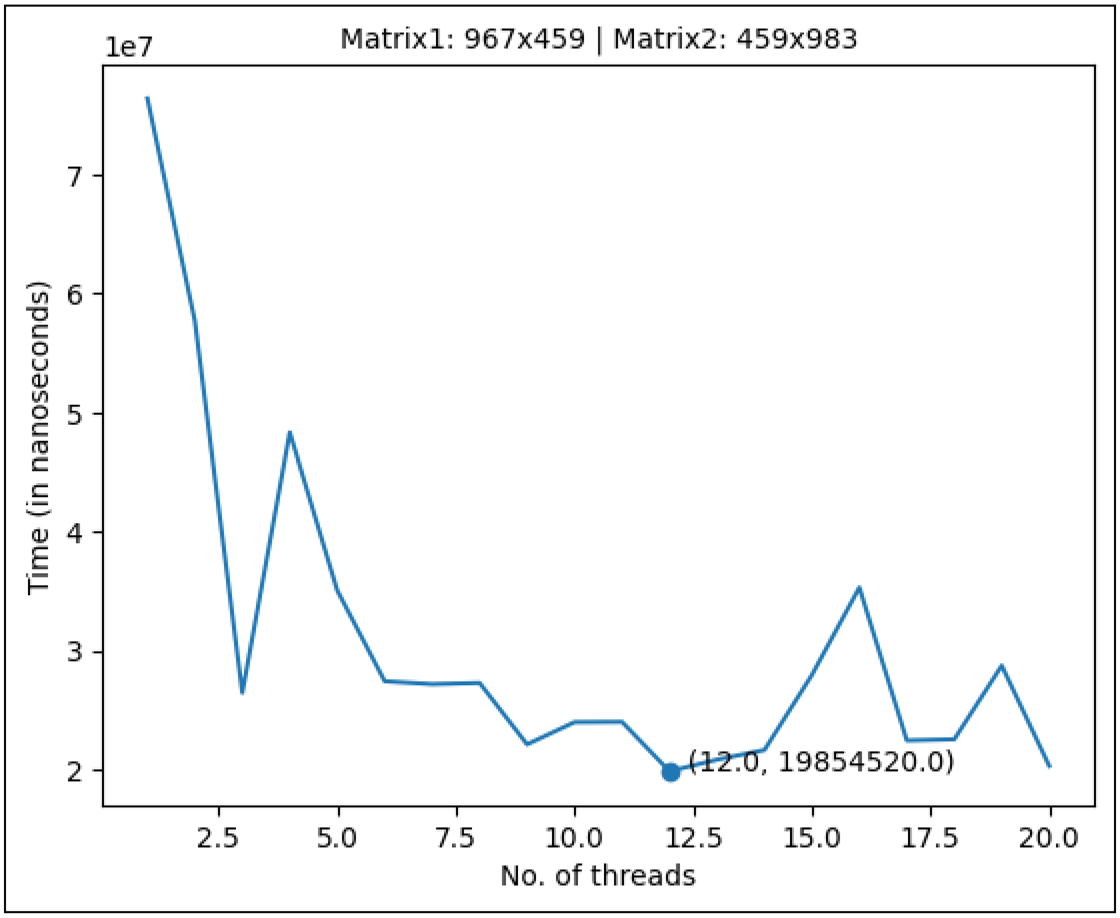


Figure 1.6

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **967 x 459** and **459 x 983** is achieved for **12** threads. The reason that for **12** threads the program runs faster compared to a lesser number of threads, is that now the program **can read more input** (values stored in the matrix) **per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can read more input per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

4.

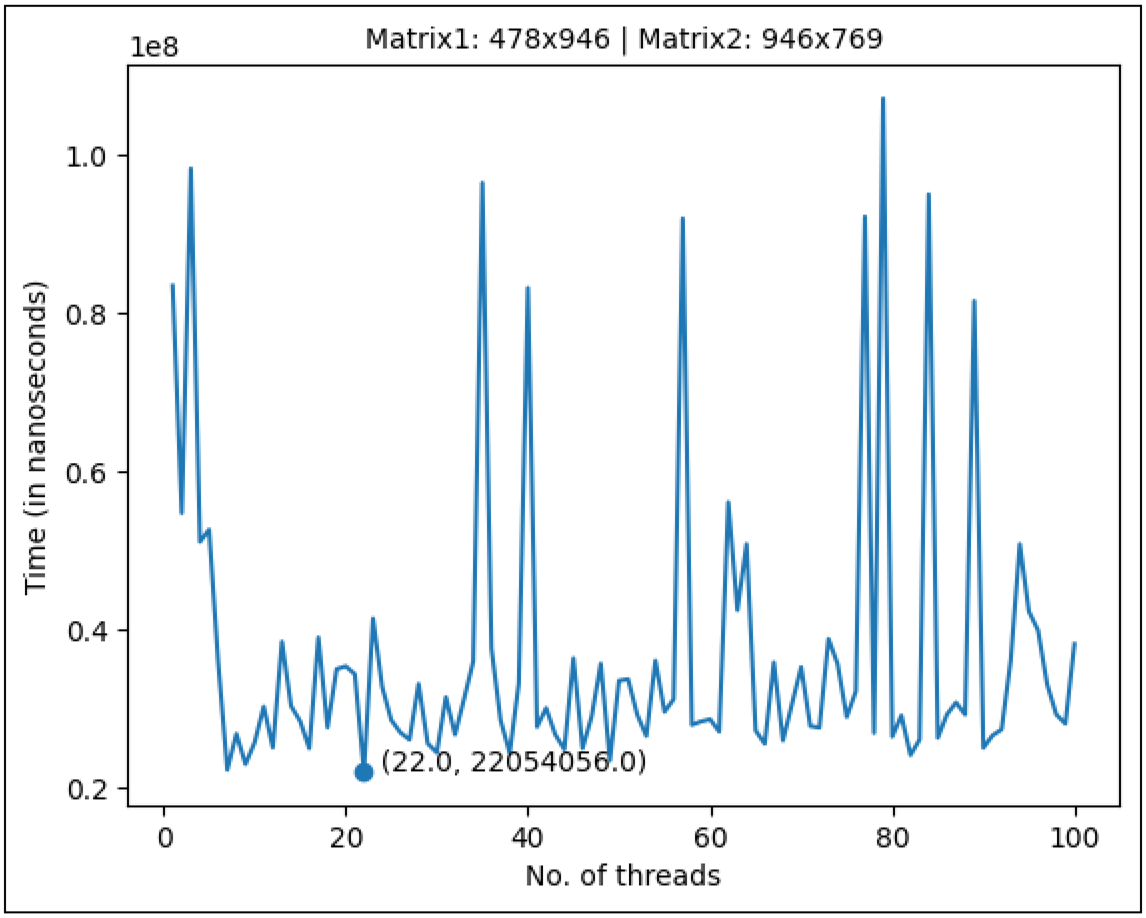

Matrix1: 478x946 | Matrix2: 946x769
1e8
1.0
0.8
0.6
0.4
0.2
Time (in nanoseconds)
0
20
40
60
80
100
No. of threads
(22.0, 22054056.0)

Figure 1.7

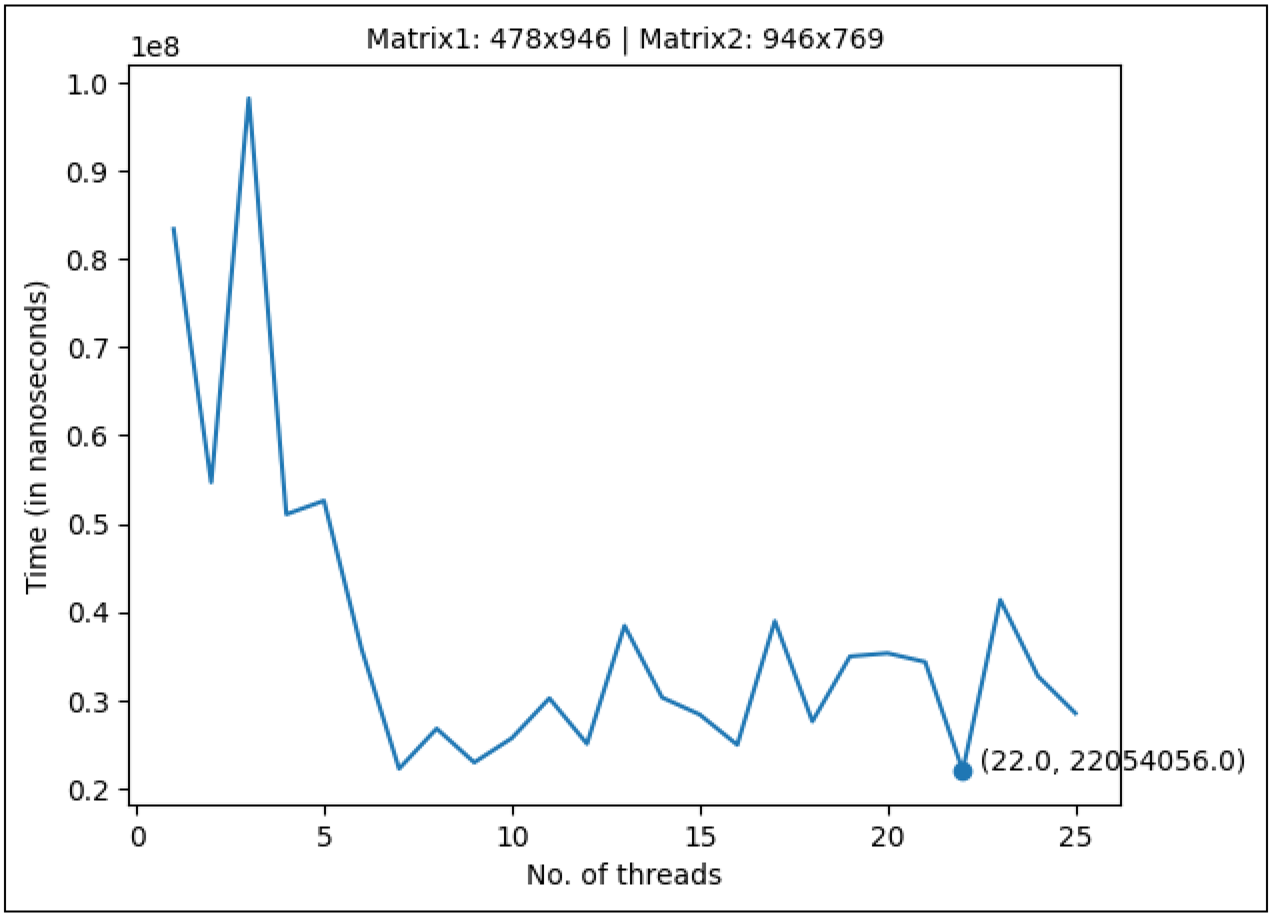


Figure 1.8

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **478 x 946** and **946 x 478** is achieved for **22** threads. The reason that for **22** threads the program runs faster compared to a lesser number of threads, is that now the program **can read more input** (values stored in the matrix) **per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can read more input per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

Analysis

On observing the above 2 cases, it can be seen that if the number of columns for the first matrix and the number of rows for the second are **varied,** then there is an observable change in the number of threads that achieve the minimum execution time. The reason for this may be that the work done by a particlar thread would depend on the number of elements (which depends on the number of columns) that the particular thread has to read. As that thread reads **more number of elements**, it would take **more time** to finish its job. This means that as we increase this column size, using **more number of threads** would give us a **better performance time**.

Plots and Analysis for P2

1.

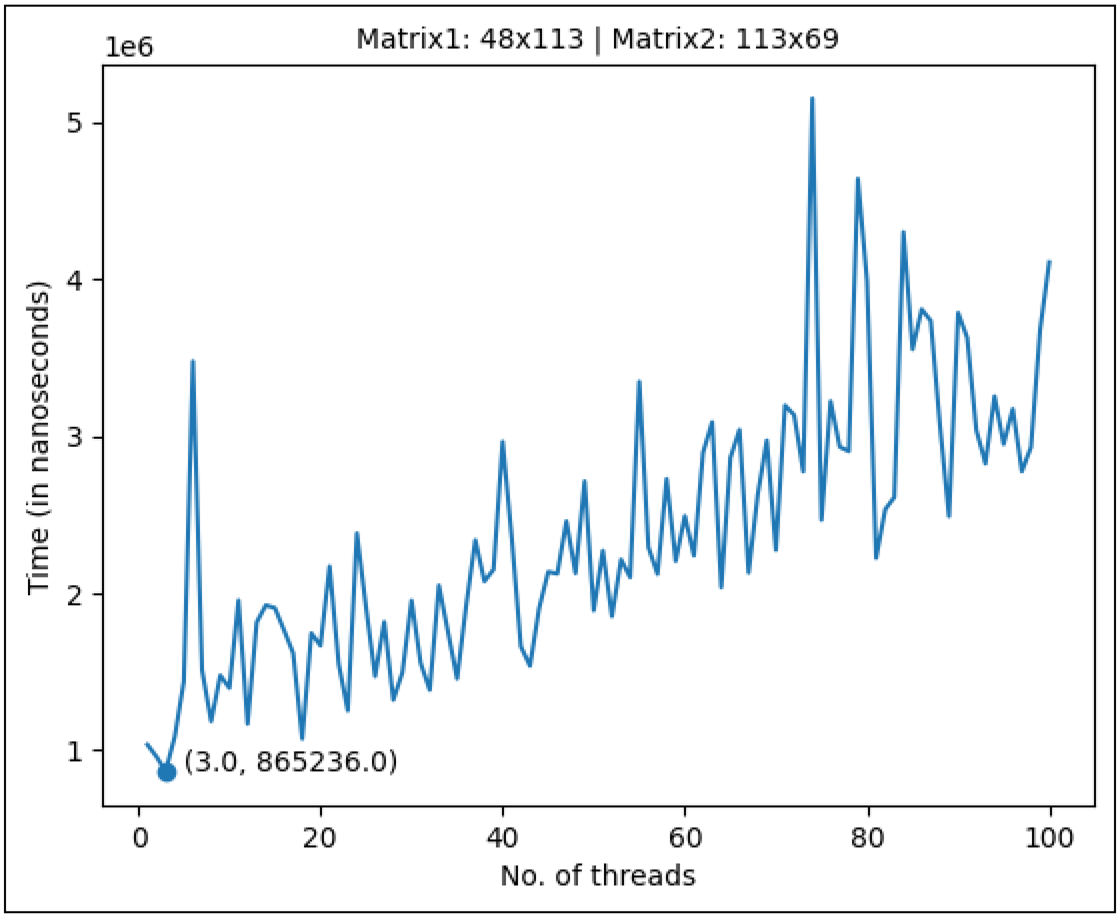


Figure 2.1

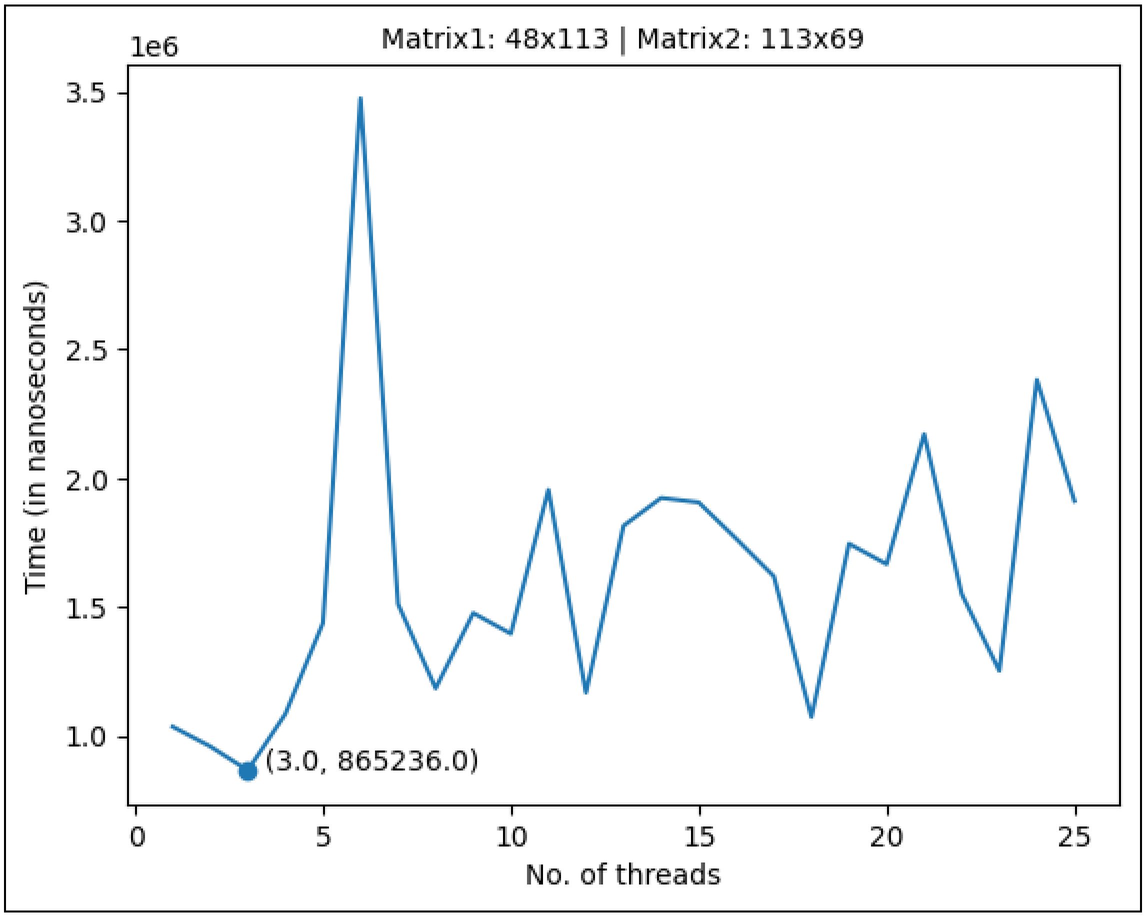


Figure 2.2

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **48 x 113** and **113 x 69** is achieved for **3** threads. The reason that for **3** threads the program runs faster compared to a lesser number of threads, is that now the program can perform **more operations per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can perform more operations per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

2.

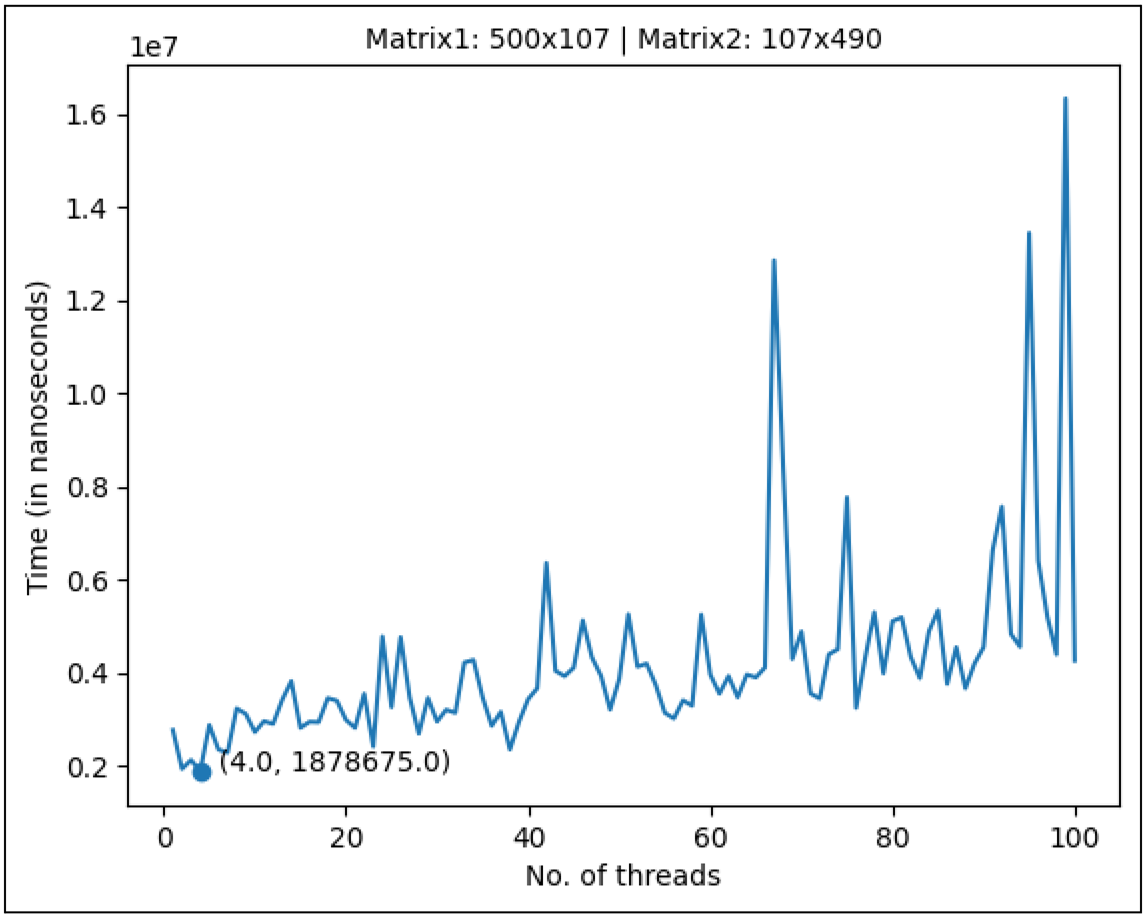

Matrix1: 500x107 | Matrix2: 107x490
1e7
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
Time (in nanoseconds)
(4.0, 1878675.0)
0
20
40
60
80
100
No. of threads

Figure 2.3

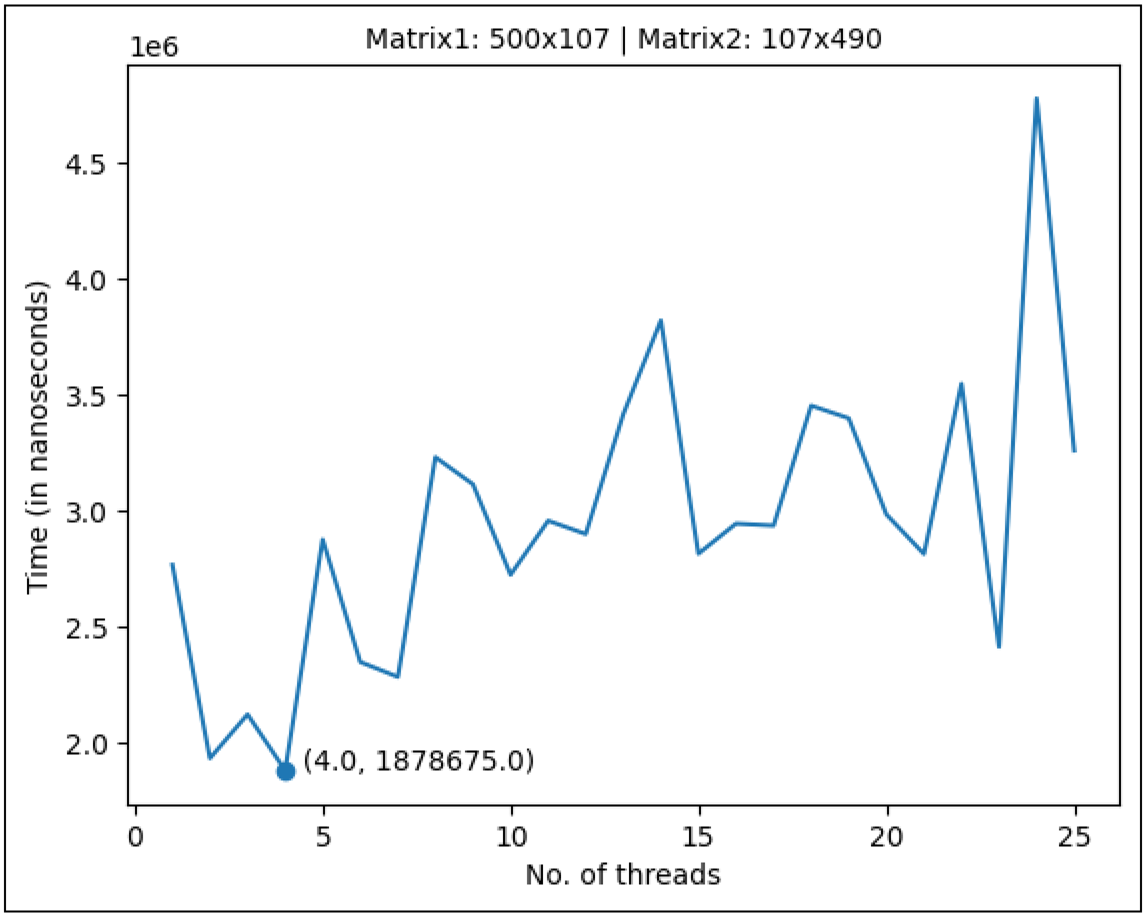


Figure 2.4

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **500 x 107** and **107 x 490** is achieved for **4** threads. The reason that for **4** threads the program runs faster compared to a lesser number of threads, is that now the program can perform **more operations per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can perform more operations per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

3.

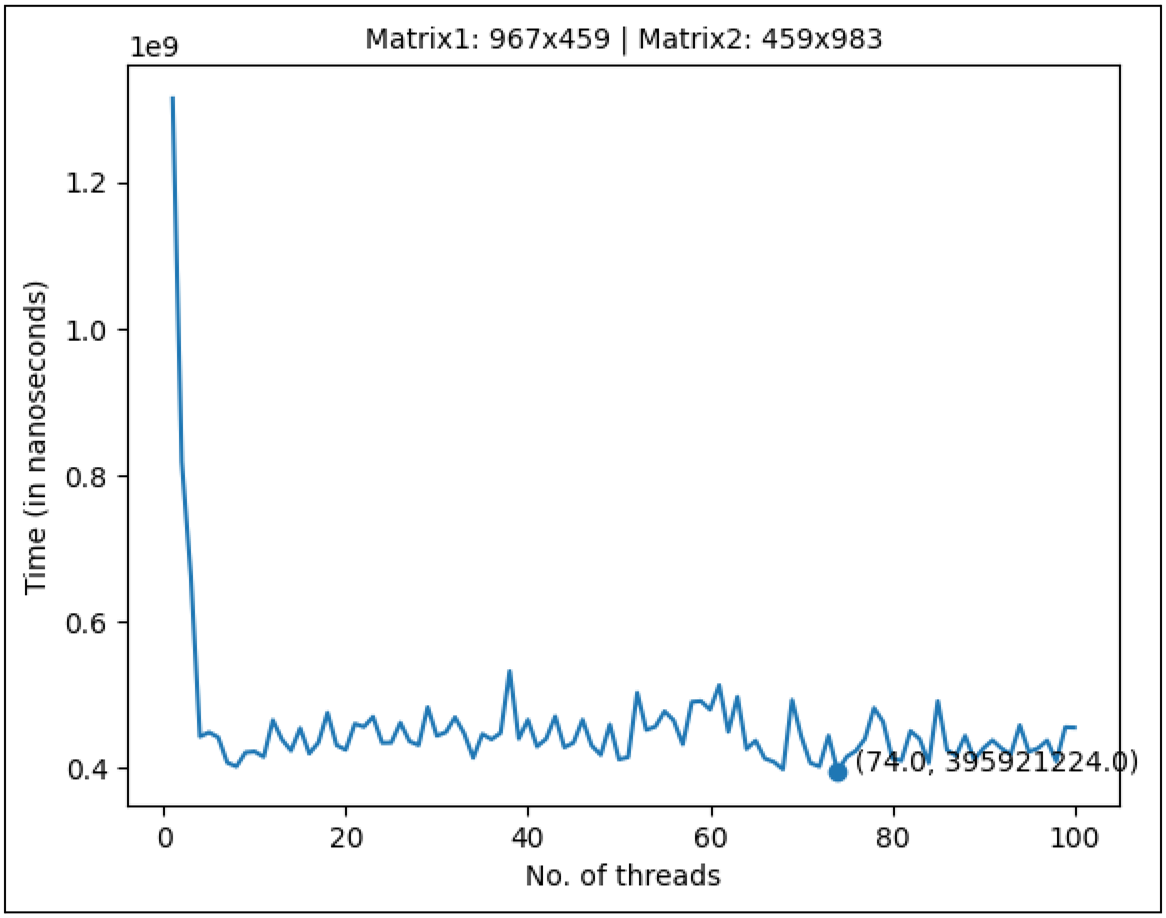

Matrix1: 967x459 | Matrix2: 459x983
1e9
1.2
1.0
0.8
0.6
0.4
Time (in nanoseconds)
0
20
40
60
80
100
No. of threads
(74.0, 395921224.0)

Figure 2.5

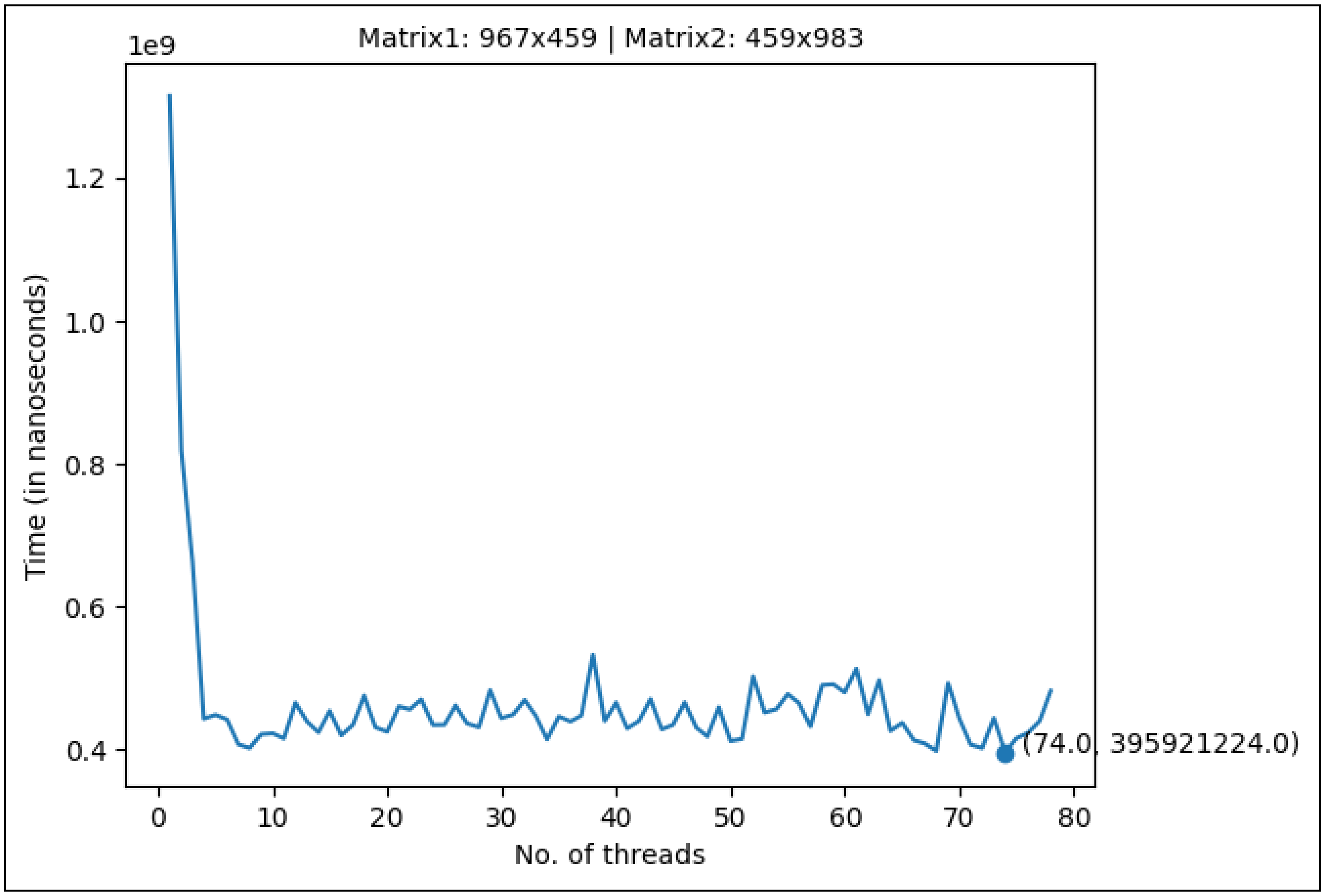


Figure 2.6

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **967 x 459** and **459 x 983** is achieved for **74** threads. The reason that for **74** threads the program runs faster compared to a lesser number of threads, is that now the program can perform **more operations per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can perform more operations per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

4.

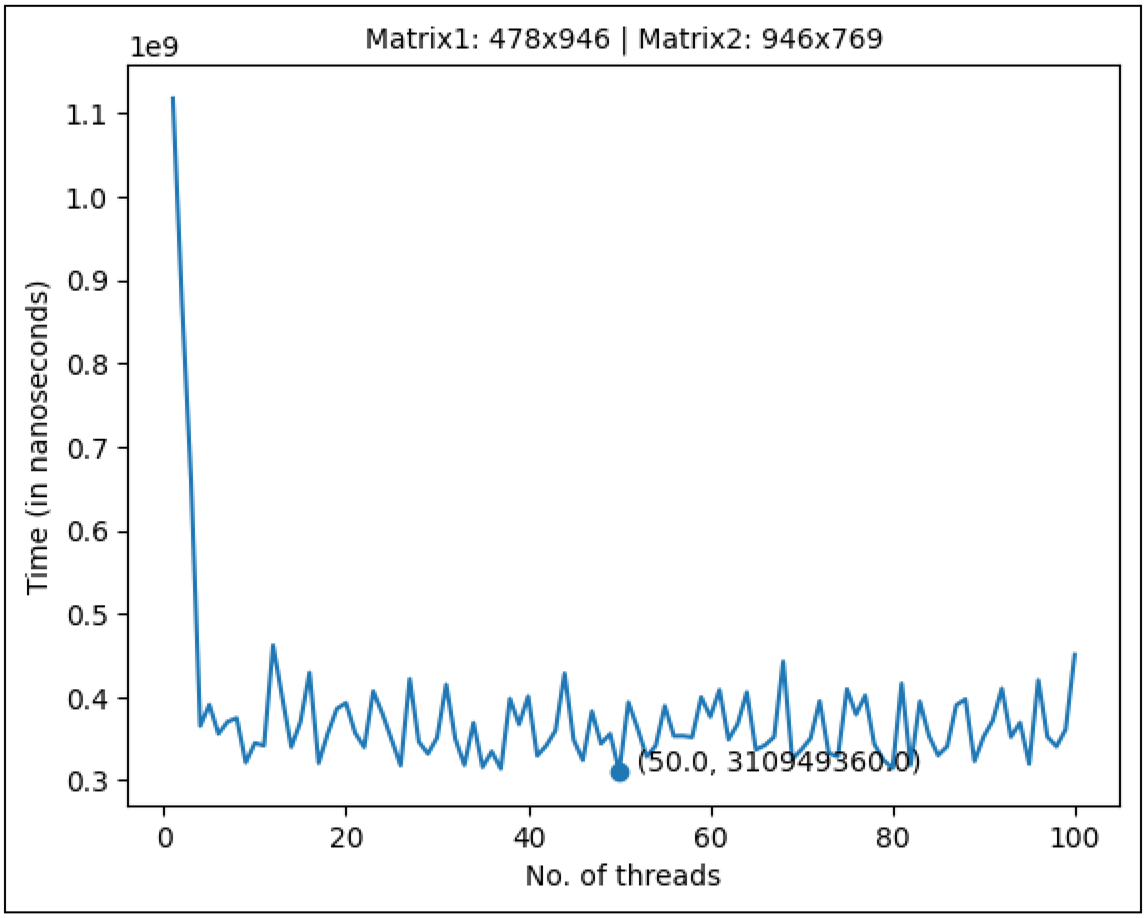

Matrix1: 478x946 | Matrix2: 946x769
1e9
Time (in nanoseconds)
No. of threads
(50.0, 310949360.0)
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
20
40
60
80
100

Figure 2.7

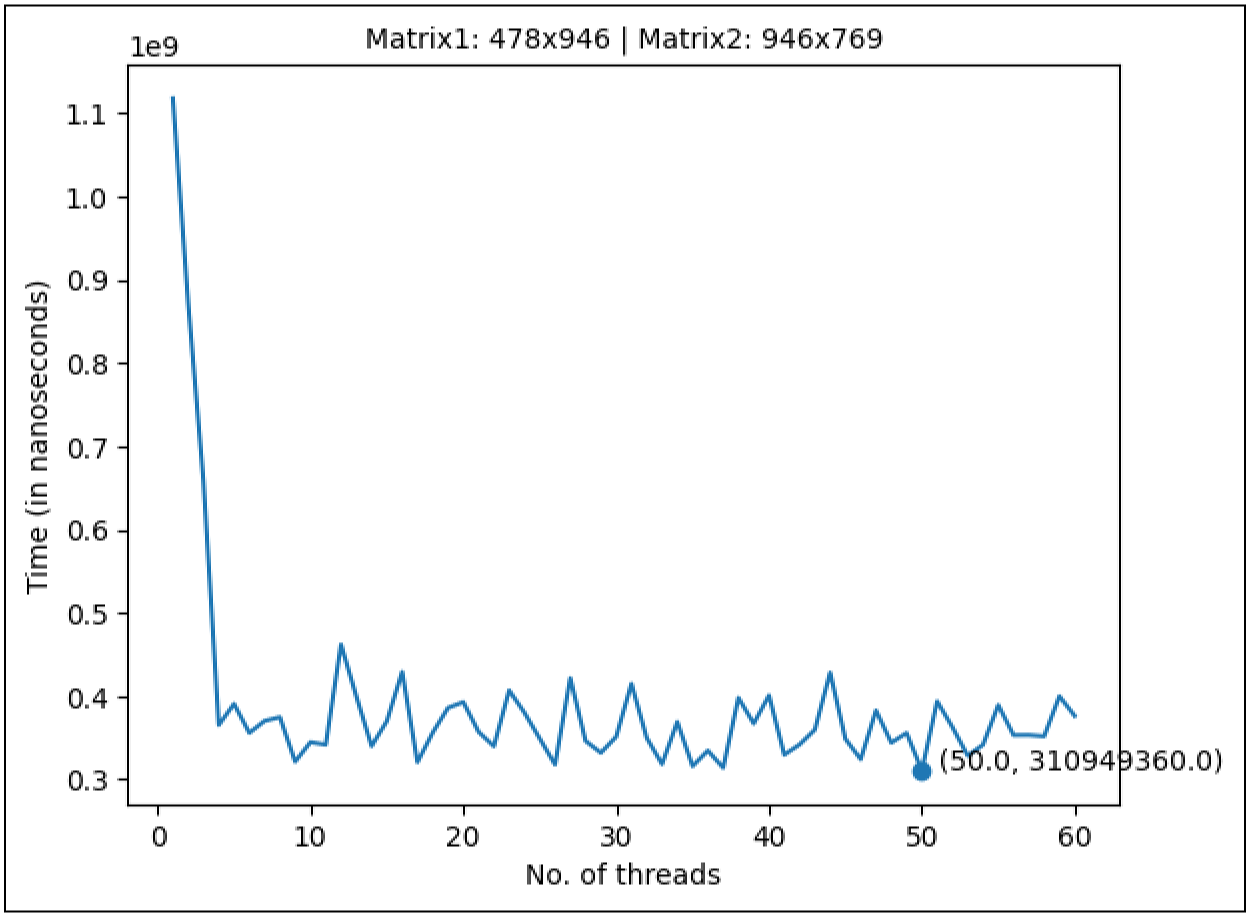


Figure 2.8

From the above plots, it can be seen that the minimum time of execution for matrices of sizes **967 x 459** and **459 x 983** is achieved for **50** threads. The reason that for **50** threads the program runs faster compared to a lesser number of threads, is that now the program can perform **more operations per unit of time**. As we start to use more threads, the execution time starts to increase. The reason for this is that for a large number of threads, even though we can perform more operations per unit of time, there is the additional **overhead of creating these threads**. This additional overhead results in an **overall increase** in the execution time of the program.

Analysis

On observing the above 2 cases, it can be seen that on **decreasing the values** of the number of rows for the first matrix and the number of columns for the second matrix, there is an **observable decrease** in the number of threads required to achieve the minimum execution time. This is because the total **work done** by a particular thread depends on the values of the number of rows for the first matrix and the number of columns for the second matrix. This means that a practicular thread would take the lesser amount of time to finish its job, meaning that lesser number of threads will perform better.

Plots and Analysis for S

1. Round Robin with Time Quantum = 2ms

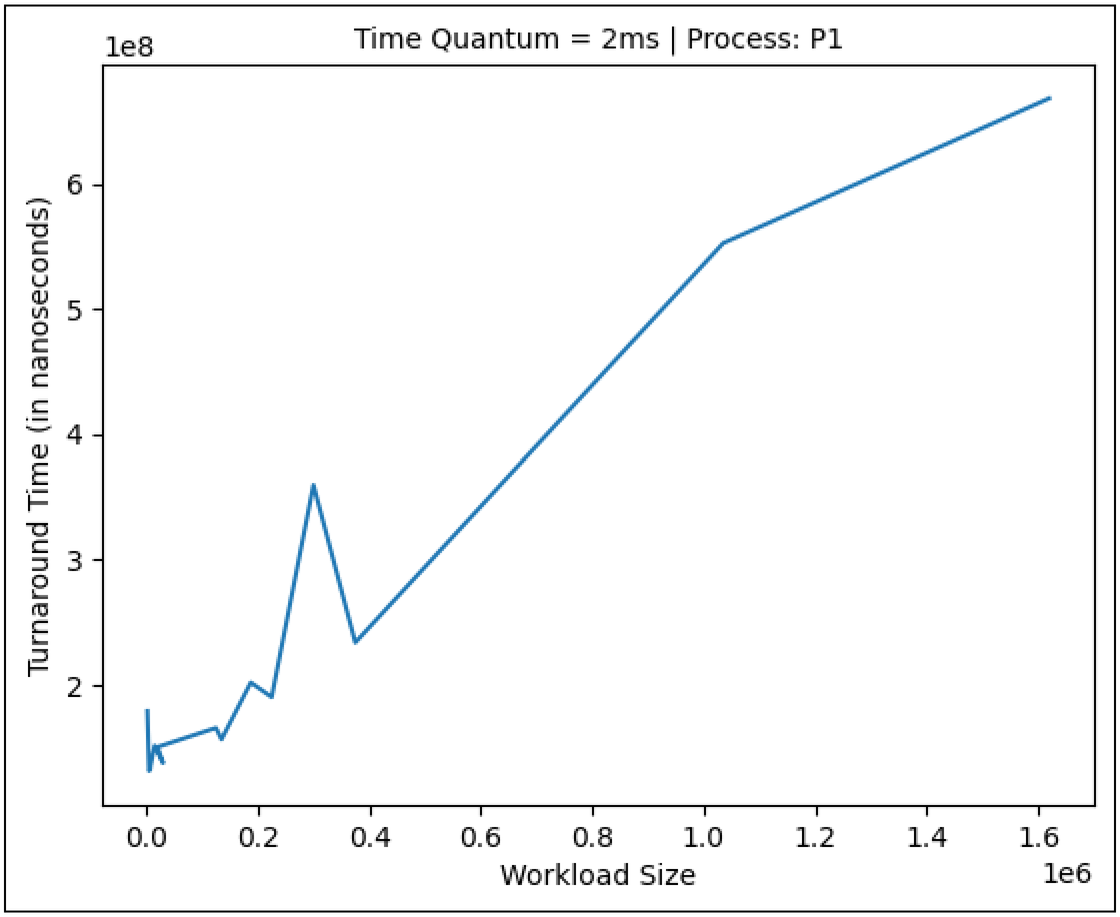


Figure 3.1

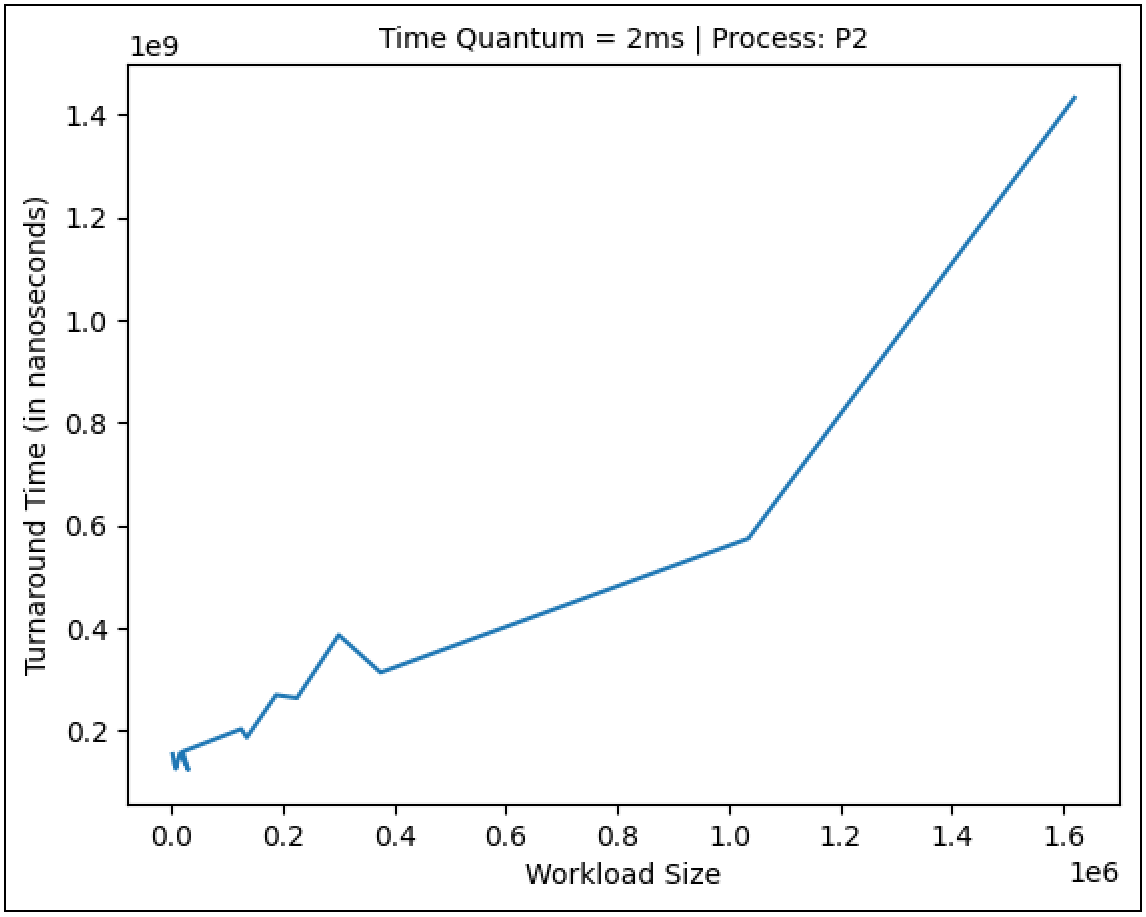


Figure 3.2

On observing the above plots, as the workload size increases, the turnaround time and the waiting time for process P1 increases due to the fact that more work has to be done, and also that the total number of quanta increase.

Comparing turnaround time and waiting time of P1, we get that they are similar with just a shift on the y-axis. The reason for this is that the process P1 is computationally less intensive and the only shift in the timing graph is due to context switch and other overheads.

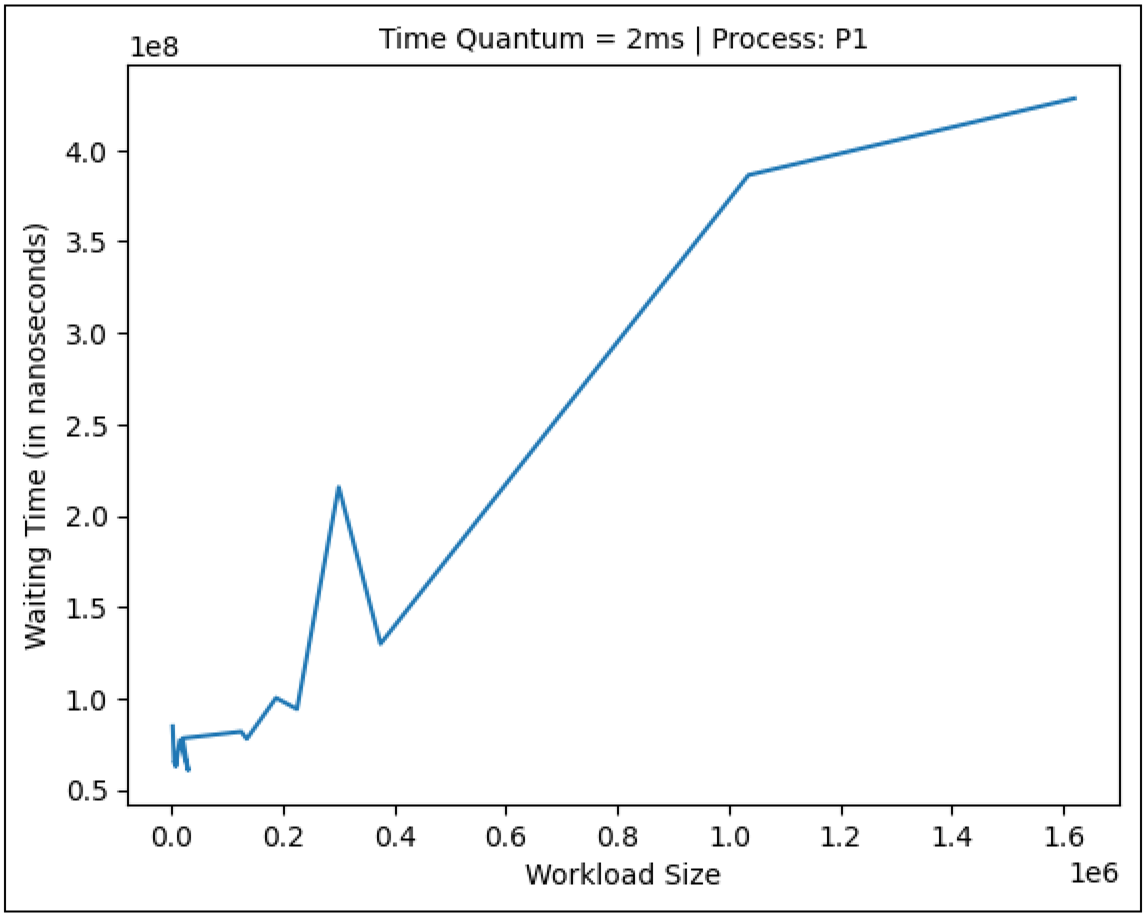


Figure 3.3

On observing the above plots, as the workload size increases, the turnaround time and the waiting time for process P2 increases due to the fact that more work has to be done, and also that the total number of quanta increase.

Comparing turnaround time and waiting time of P2, we see that P2 finishes after P1, which means that the waiting time of P2 is no longer dependent of the execution time of P1 and purely depends on the context switch time.

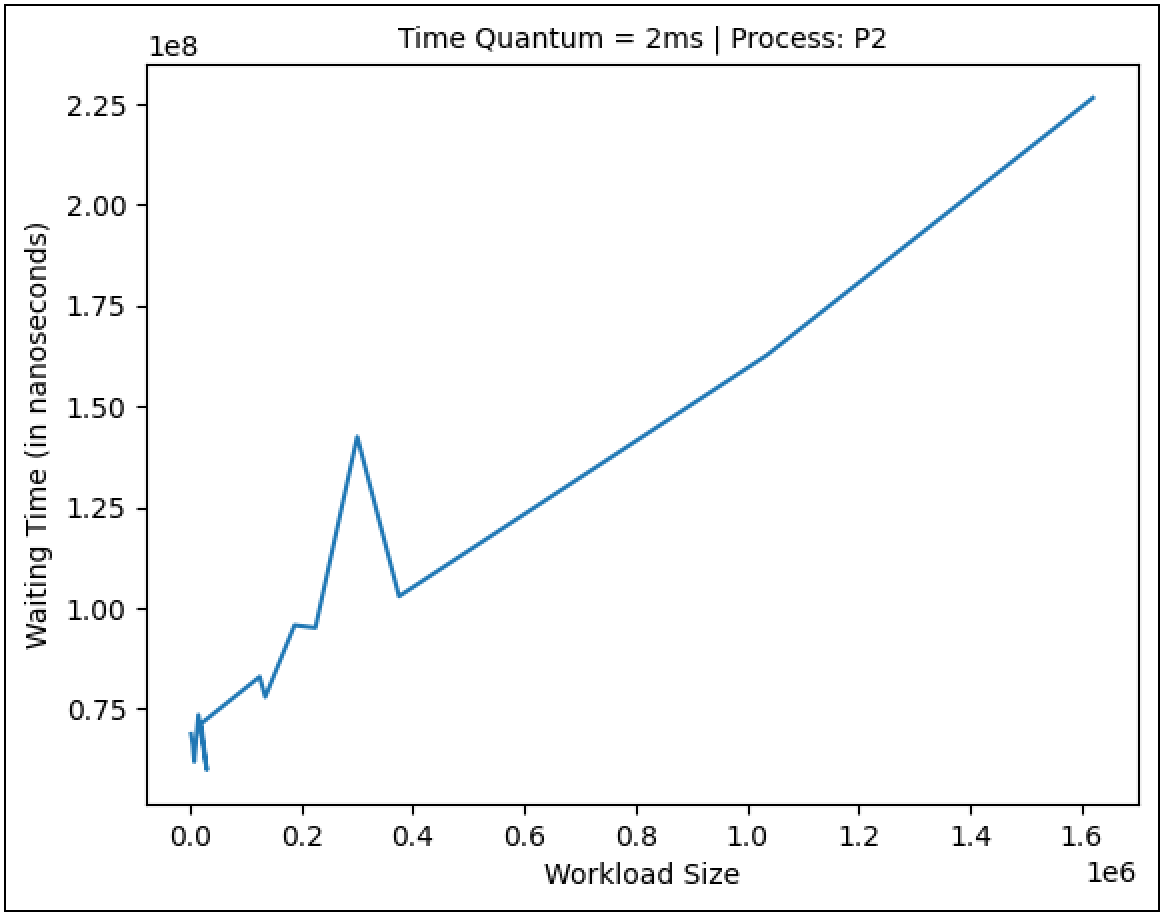
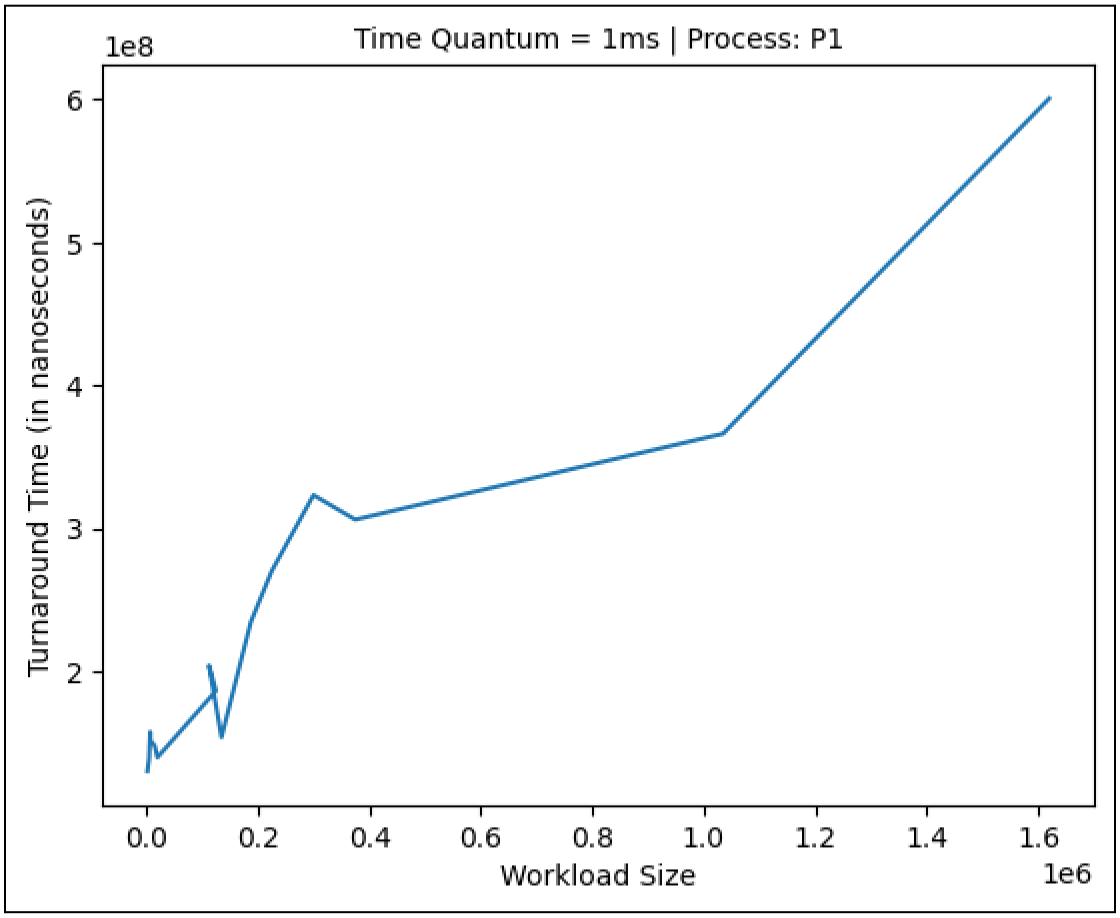

Time Quantum = 2ms | Process: P2
1e8
2.25
2.00
1.75
1.50
1.25
1.00
0.75
Waiting Time (in nanoseconds)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
Workload Size
1e6

Figure 3.4

2. Round Robin with Time Quantum = 1ms



<u>Figure 3.5</u>

On observing the above plots, as the workload size increases, the turnaround time and the waiting time for process P1 increases due to the fact that more work has to be done, and also that the total number of quanta increase.

Comparing turnaround time and waiting time of P1, we get that they are similar with just a shift on the y-axis. The reason for this is that the process P1 is computationally less intensive and the only shift in the timing graph is due to context switch and other overheads.

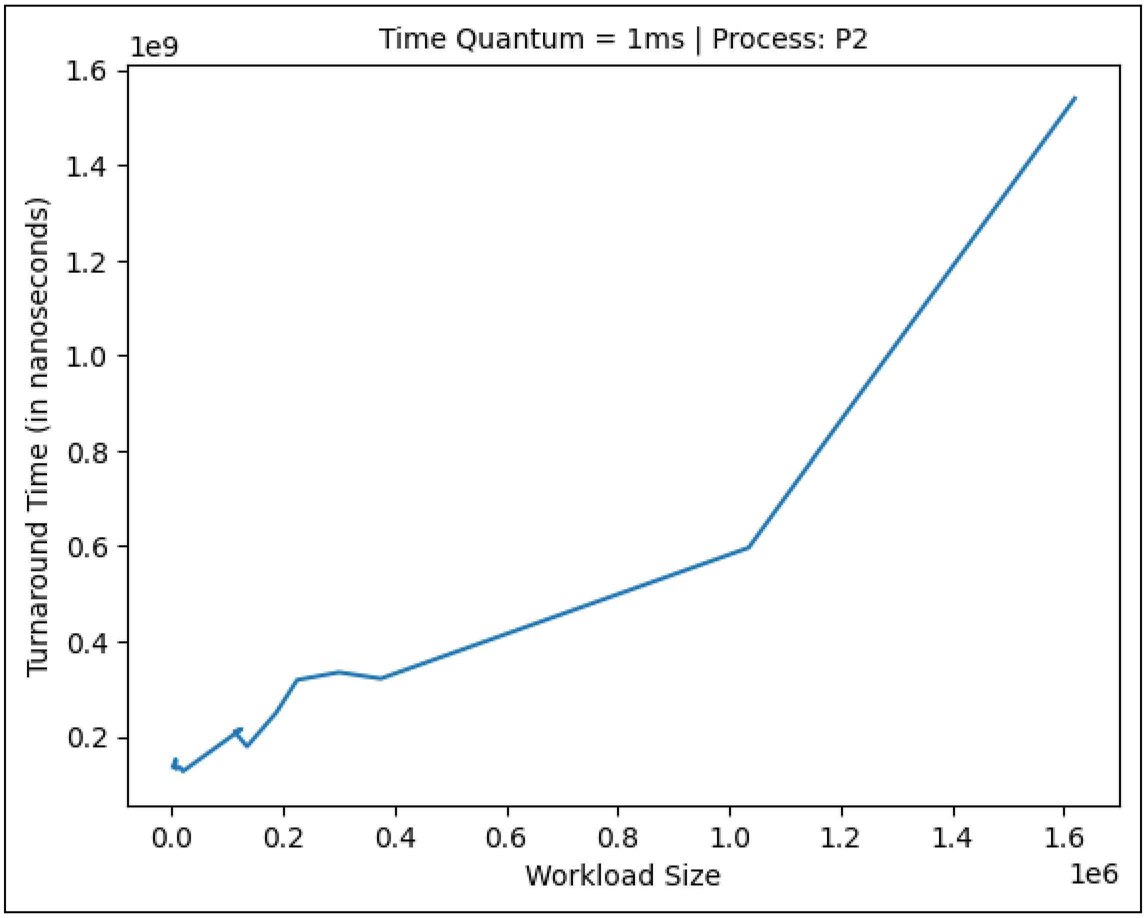


Figure 3.6

On observing the above plots, as the workload size increases, the turnaround time and the waiting time for process P2 increases due to the fact that more work has to be done, and also that the total number of quanta increase.

Comparing turnaround time and waiting time of P2, we see that P2 finishes after P1, which means that the waiting time of P2 is no longer dependent of the execution time of P1 and purely depends on the context switch time.

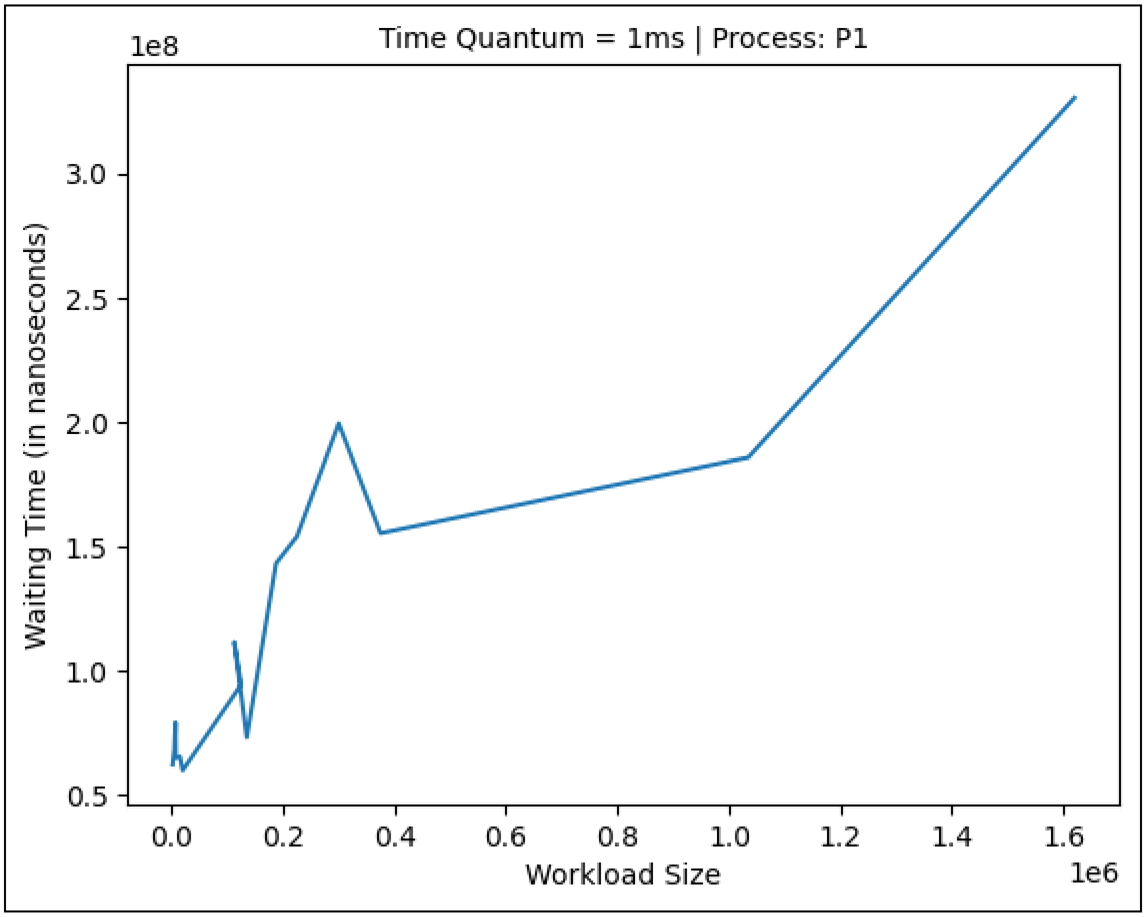

Time Quantum = 1ms | Process: P1
1e8
3.0
2.5
2.0
1.5
1.0
0.5
Waiting Time (in nanoseconds)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
Workload Size
1e6

Figure 3.7

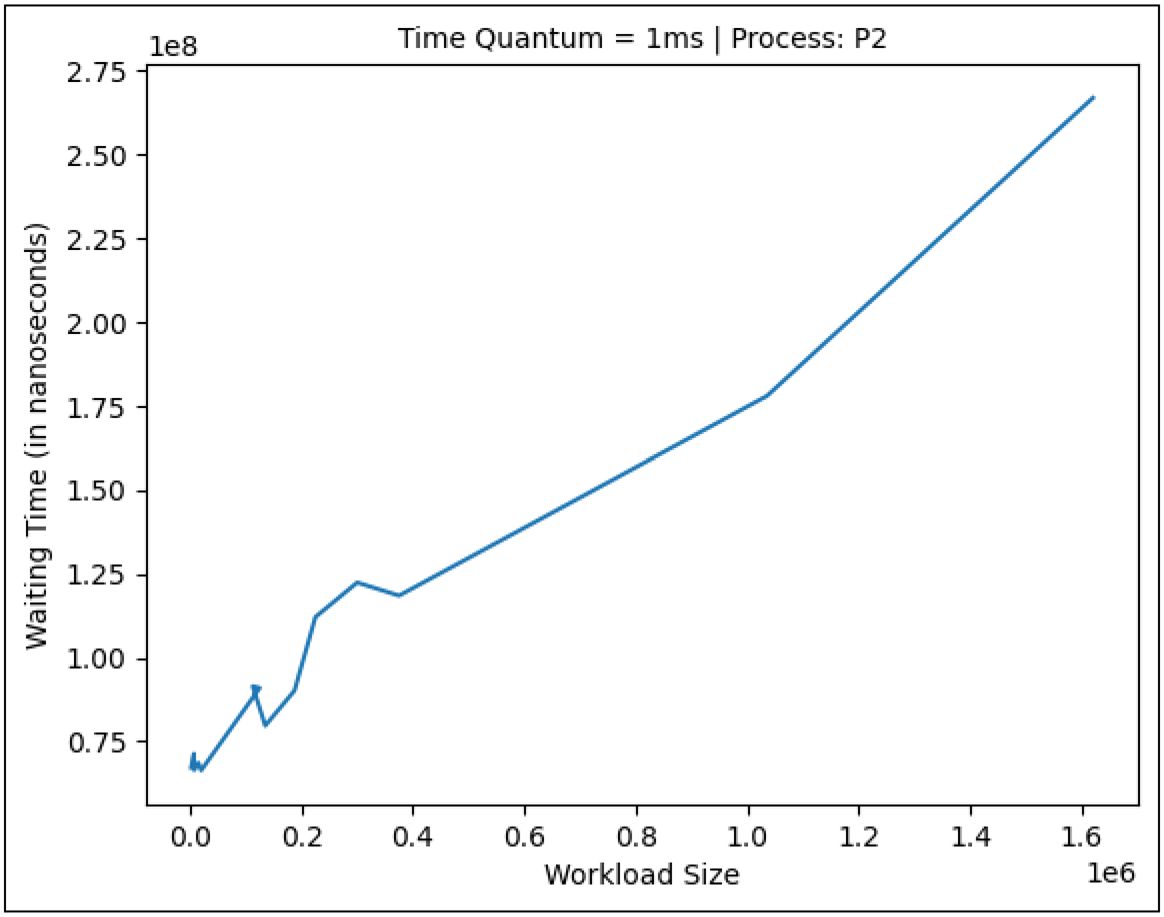


Figure 3.8

If we take time quantum to be 2 msec instead, each process gets 2 msec to perform their execution. This results in overall decrease in turnaround time and waiting time.with slight exceptions. For larger workloads the above mentioned difference is more pronounced.

Design Choices:

For P1:

Pre-Processing:

During pre-processing, we are storing the transpose of the second matrix (input2) in the text file, which makes it easier to retrieve data later. We are also storing the offset of each row of input 1 and transpose of input 2 file in an array to ease parallel reading of different rows from the file.

IPC Mechanism:

For the IPC mechanism, we are using Shared Memory, which is relatively more convenient to use than the message queue in our program, as access to it requires only a pointer which can be moved freely. Also, if we use message queue, Program P2 will wait till the message queue has all the values before doing anything else, which is not the case with Shared Memory. There can also be a case where P1 comes to a stall as no more values can be inserted into the message queue.

Multithreading:

To assure that the program is utilizing multi-threading, we are spawning `NO_OF_THREADS` threads. Each thread then switches between picking a file to read (input1 or input2) and this continues till all the rows in both the files are read. Every thread reads a new row in the respective files they are spawned. To synchronize these threads, we have used a mutex lock for the selection of the file, i.e., only one thread has access to the switch statement at a time, avoiding redundancy in the changes in the variables used.

For P2:

IPC Mechanism:

In this program, we receive messages from Shared Memory. Here, to make sure that the process P1 creates the Shared Memory before it is accessed in process P2, we put the shmget method in a while loop. This is required because when the processes are scheduled using Scheduler S, it might be the case that P2 asks for the Shared Memory before P1 gets the chance to create the memory.

Multithreading:

Here, we have used multithreading as each thread looks at a single cell of the resultant matrix and sees if it is already processed or not. If not, the thread selects that cell and finds the respective row and column to be multiplied and added from the Shared Memory. To ensure synchronization, again we have used mutex to give only one thread at a time to select the next cell to process.

The result of the matrix multiplication is then stored in out.txt.

For S:

The job of the scheduler is to take turns assigning jobs to processes P1 and P2. This is achieved by the combination of fork and exec. The parent utilizes system signals such as SIGCONT and SIGSTOP to control the operation of the two processes.

For the first fork, we pass an exec command to run P1 in the child while forking again in the parent. In the child of the second fork, we pass the exec command to run P2. In the parent of the inner fork is where all the job of switching between processes happens.

We have used Shared Memory again here to define two flags, each representing the completion of one of the processes. While both flags do not return, we continue switching between processes, using SIGSTOP to pause a process and SIGCONT to continue it. Each process gets to execute for the time quantum specified before it is switched, and it continues till both of them finish their execution.