Operating Systems - Assignment 2

Group 48

Archis Sahu	2020A7PS1692H
Siddhant Panda	2020A7PS0264H
Yash Ratnani	2020A7PS0039H
Rishi Poddar	2020A7PS1195H
Shubh Badjate	2020A7PS0028H
Ankit Yadav	2020A7PS2046H

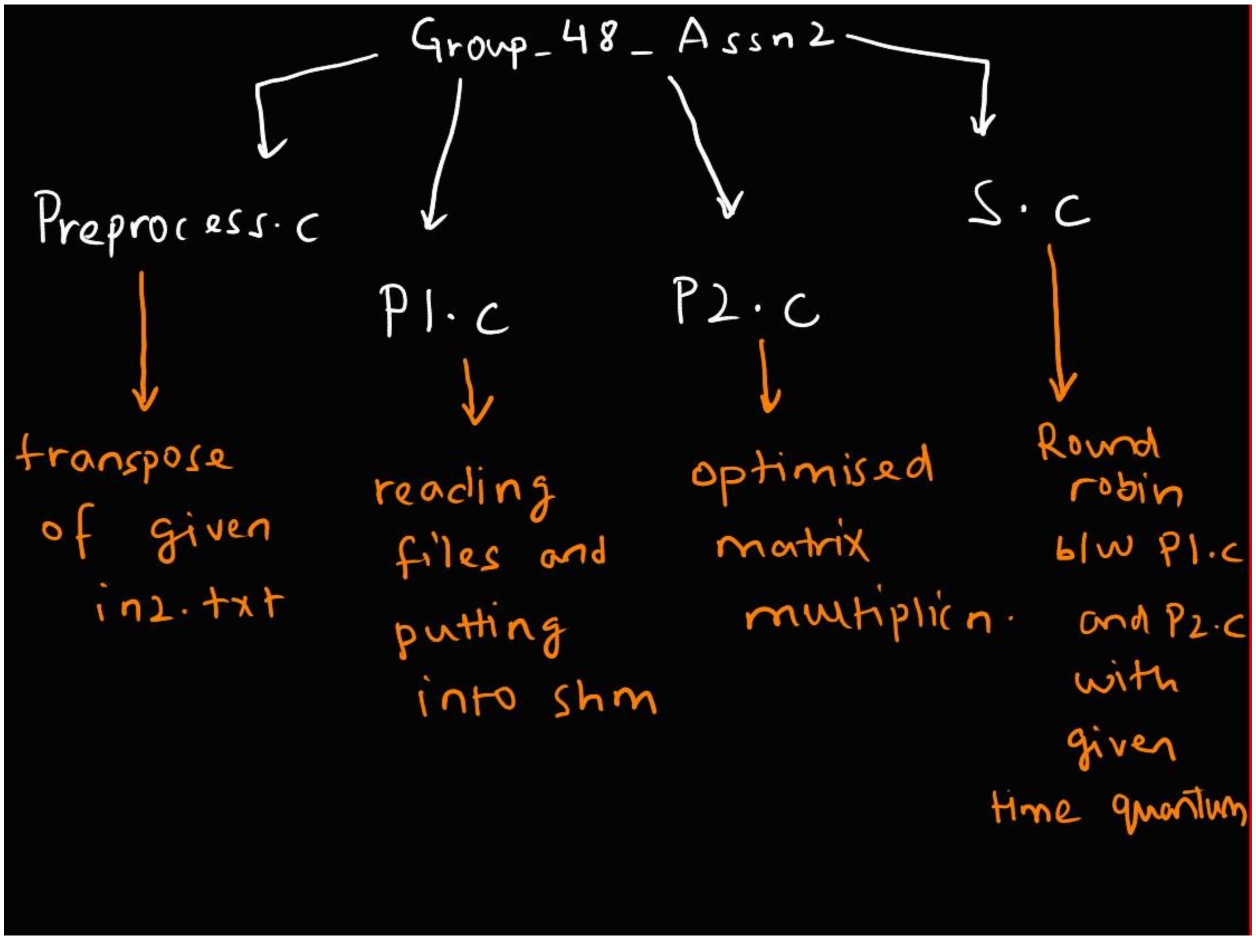
Problem Statement :

The problem asks us to analyze multiplication of arbitrarily large sized ***2-D*** matrices; also to compare and contrast the results with varying parameters, such as number of threads and Matrix dimensions.

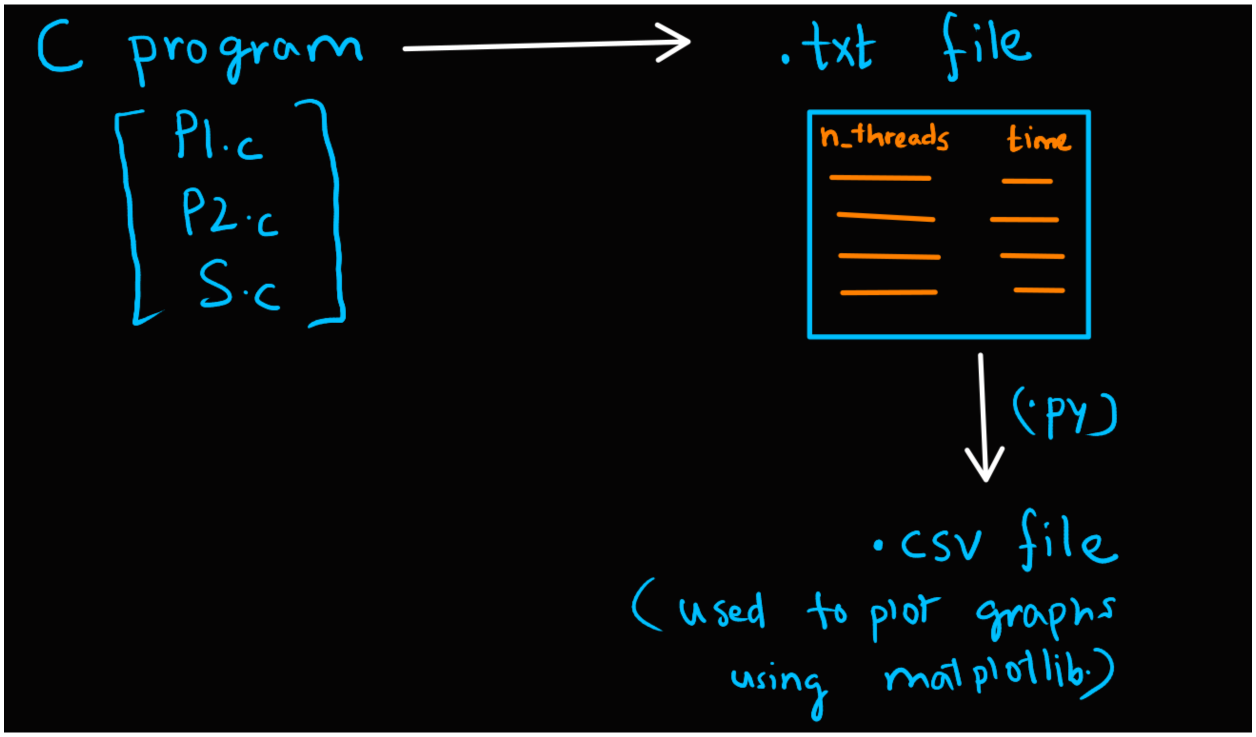
We have used parallelism provided by the Linux process and threads libraries.

Flow of the assignment :

The flow of the entire assignment is explained in the below flow chart :

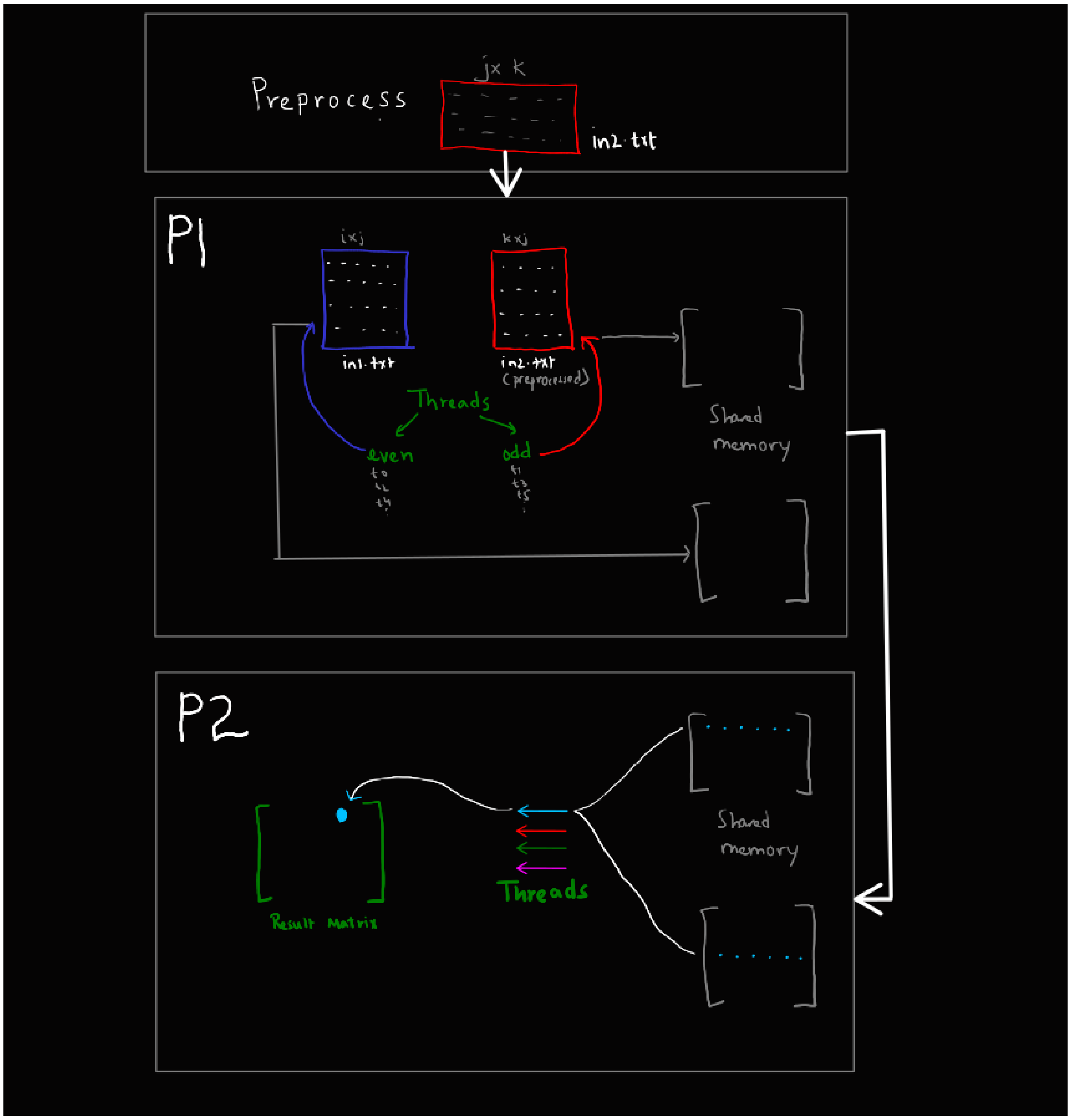

Group_48_Assn2
Preprocess.c
P1.c
P2.c
S.c
transpose of given in2.txt
reading files and putting into shm
optimised matrix multiplicn.
Round robin b/w P1.c and P2.c with given time quantum

The above figure shows the C files used for the assignment, with their respective tasks.
In addition, Python is also used in the shown methods, to generate and analyze the results :



The respective C program gives an output in the form of a .txt file, with the first column as number of threads, and the second column as time taken. Python is then used to convert the .txt file into .csv file, which is used for plotting the graphs using the library matplotlib.

Now, let us analyze each program in detail :



Preprocess :

- The matrices given as input are 2-Dimensional arrays of form ***i x j*** and ***j x k*** respectively. We preprocess the second input matrix and take the transpose, so as to make the process of multiplication easier, proceeding in a row-major fashion.
- In conclusion, ***in2.txt*** after preprocessing is now a 2-Dimensional matrix of form ***k x j***.

P1 :

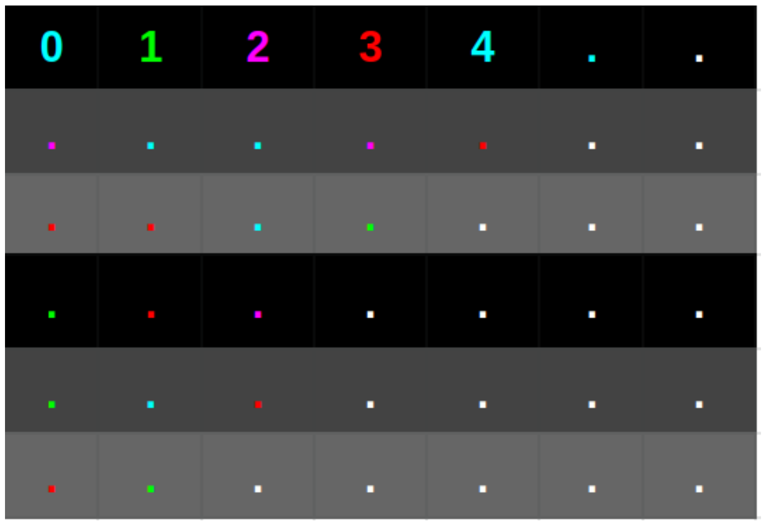
- The objective of this part is to read the given input files efficiently, where the number of threads reading the different parts of the file are variable.
- The idea behind doing this is to divide the threads into two subgroups – even numbered threads, and odd numbered threads.
- The even numbered threads read the first input file ***in1.txt*** and the odd numbered files read the second input file ***in2.txt***. While reading from the files, we've ensured the following implementations :
    - An index variable is maintained, which indicates where exactly a thread that was reading the file stopped reading it, and hence the coming thread knows where to start reading from.
    - A mutex lock is also used on the index variable to avoid race conditions.
    - Though this diminishes the effect of multithreading, it still is an optimisation, because using various threads ensures that the two input files are being read at the same time.
    - A boolean matrix is also created and put in the shared memory which is accessed by P2. This checks if the input has already been read before calculating the result of multiplication.

- The read inputs are stored in shared memory in the form of a 1-dimensional array, which is then further used by the next part (P2).

P2 :

- The task achieved by this part is to access the Matrices stored in the shared memory, and to multiply them optimally.
- The different approaches tried before arriving at the optimum approach are elucidated below :
    - Approach 1 :
        - ➢ A certain number of threads (say ***num_threads*** = 4) enter the shared memory to take the first row of the first matrix, and the threads according

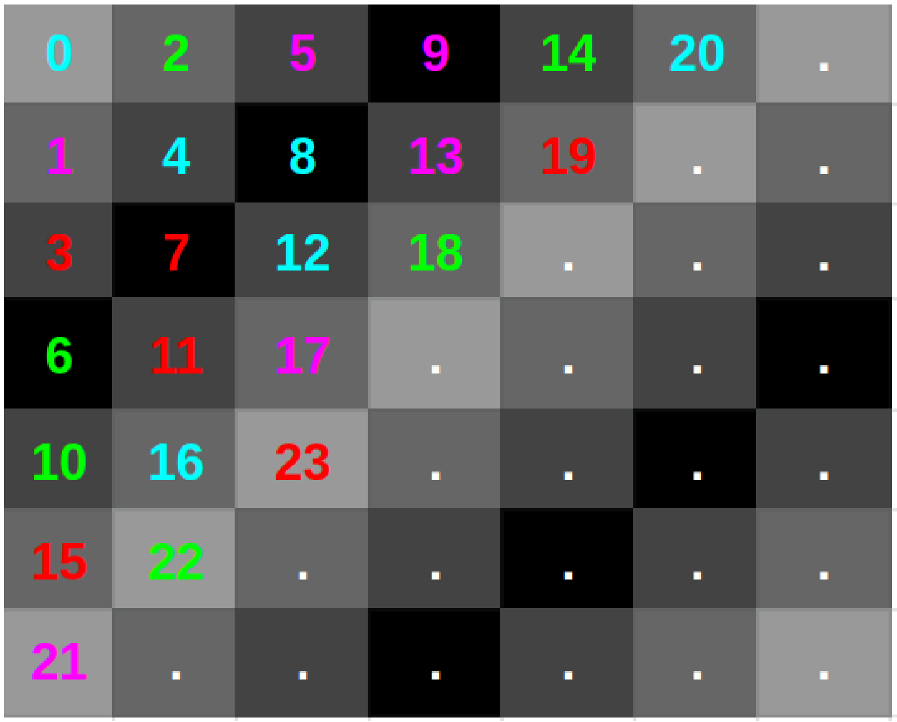
to their indices multiply corresponding rows in the second matrix.



- ➢ For example : Thread 1 multiplies the first row of the first matrix with the first row of the second matrix, and then proceeds to multiply the first row of the first matrix with the 5th (1 + ***num_threads***) row of the second matrix, and this continues. Meanwhile, the second thread multiplies the first row with second, and then first with 6th(2 + ***num_threads***) and so on.
- ➢ Once all rows of the second matrix have been multiplied with the first row, we move on to the second row of the first matrix and repeat the same process.
- ➢ This means that we have to wait for a significant amount of time for the entirety of the second matrix to be read JUST to multiply with the first row of the first matrix. This creates an imbalance.

- ○ Approach 2 - Optimisation on imbalance :
  - ➢ We now try to improve on the imbalance mentioned earlier, by using a certain method to traverse the input files, which is determined by a

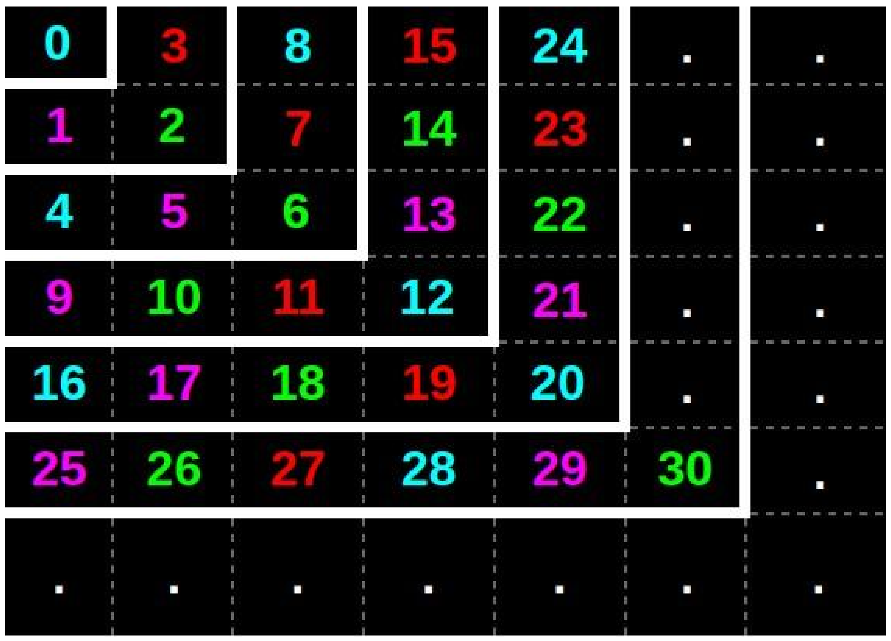
particular order of generation of the result matrix as shown :



0	2	5	9	14	20	.
1	4	8	13	19	.	.
3	7	12	18	.	.	.
6	11	17	.	.	.	.
10	16	23	.	.	.	.
15	22	.	.	.	.	.
21	.	.	.	.	.	.

- ➢ We notice that after calculating the result corresponding to cell marked 0, thread 1 moves to compute cell marked 4, for which we require row 2 of the first matrix and row 2 of the second matrix.
- ➢ But in the previous case, we needed row 1 of the first matrix and row 5 of the second matrix for the next computation.

- ○ Approach 3 - Even better optimisation on imbalance :
  - ➢ The aforementioned problem leads to the final optimisation, in which we read the rows of the matrices based on an order followed by the result matrix as shown :



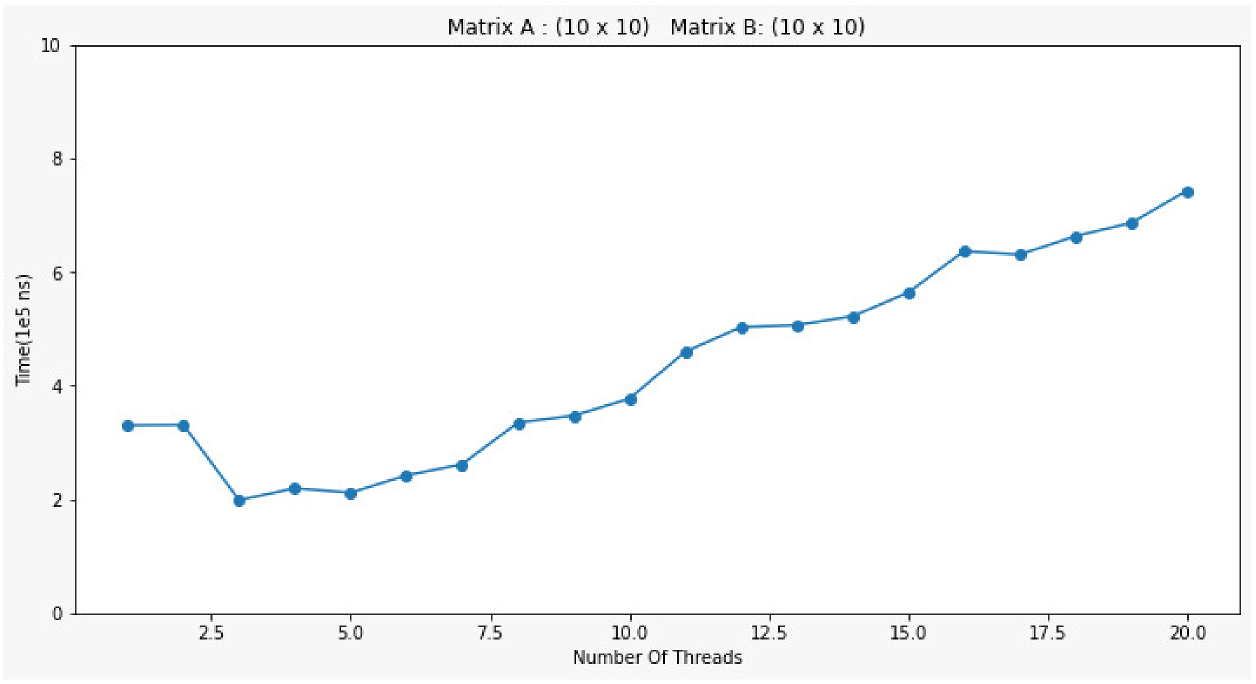
- ➢ We notice that the cells are computed in a ***"ring"*** fashion, which gives us an even efficient approach to compute our results.
- ➢ The intuition behind this lies in the fact that as we proceed ring by ring, and let's say we are at the ***N^th^*** ring, we are now calculating the elements that require 1 to ***N^th^*** rows only of both the files, which are probably read by now.
- ➢ This eliminates the imbalance completely.

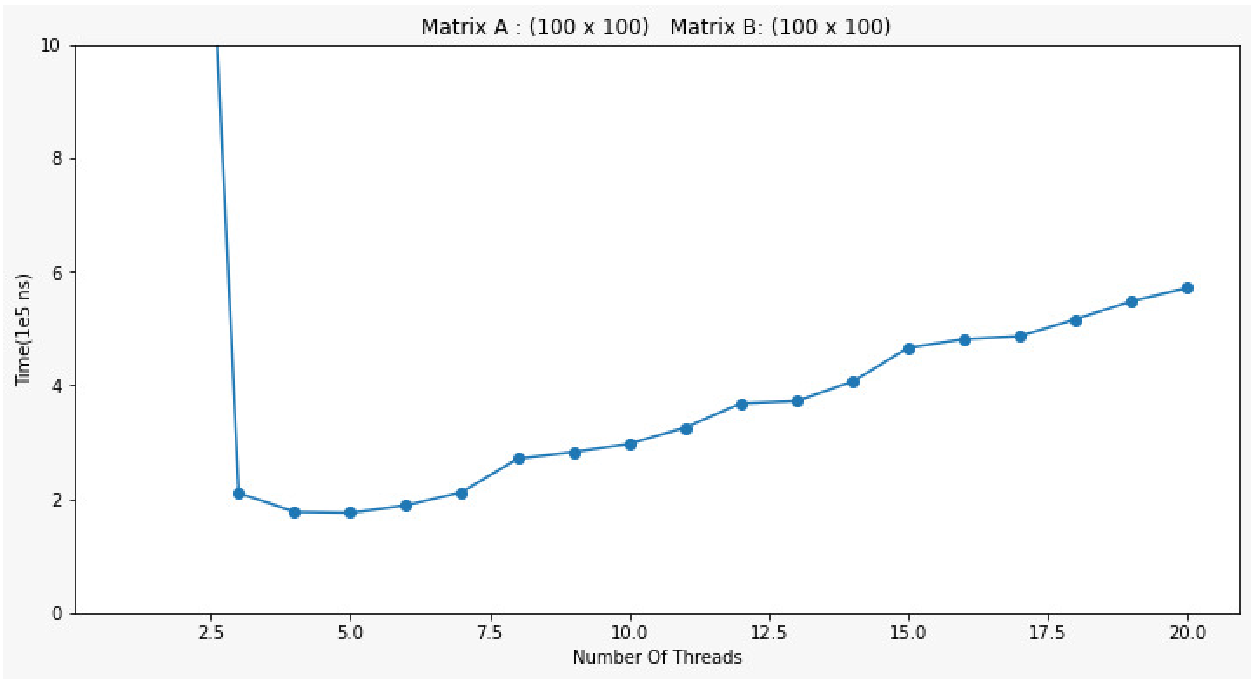
S :

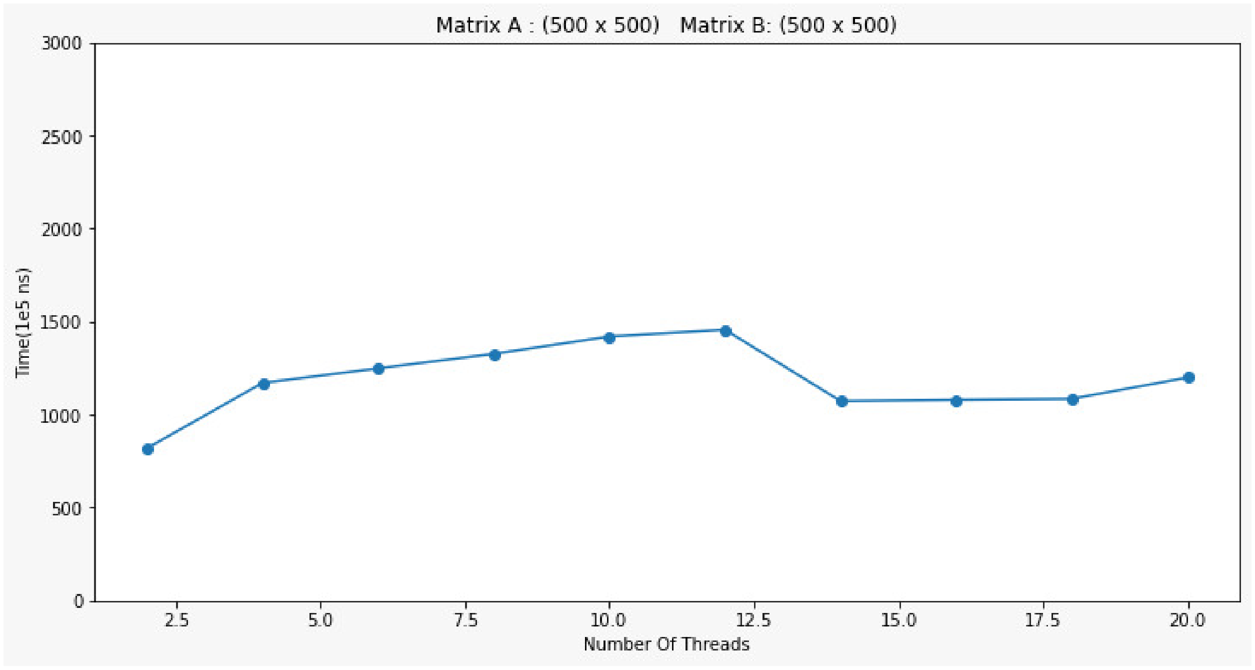
- The objective of this program is to schedule the above two given programs in a Round Robin fashion, simulating a uni-process scheduler with a given time quantum.
- We use kill signals, specifically ***SIGCONT*** and ***SIGPAUSE*** to coordinate between the two processes.
- We also use two shared memory variables which are set when each of the respective processes terminate, and the scheduler exits.

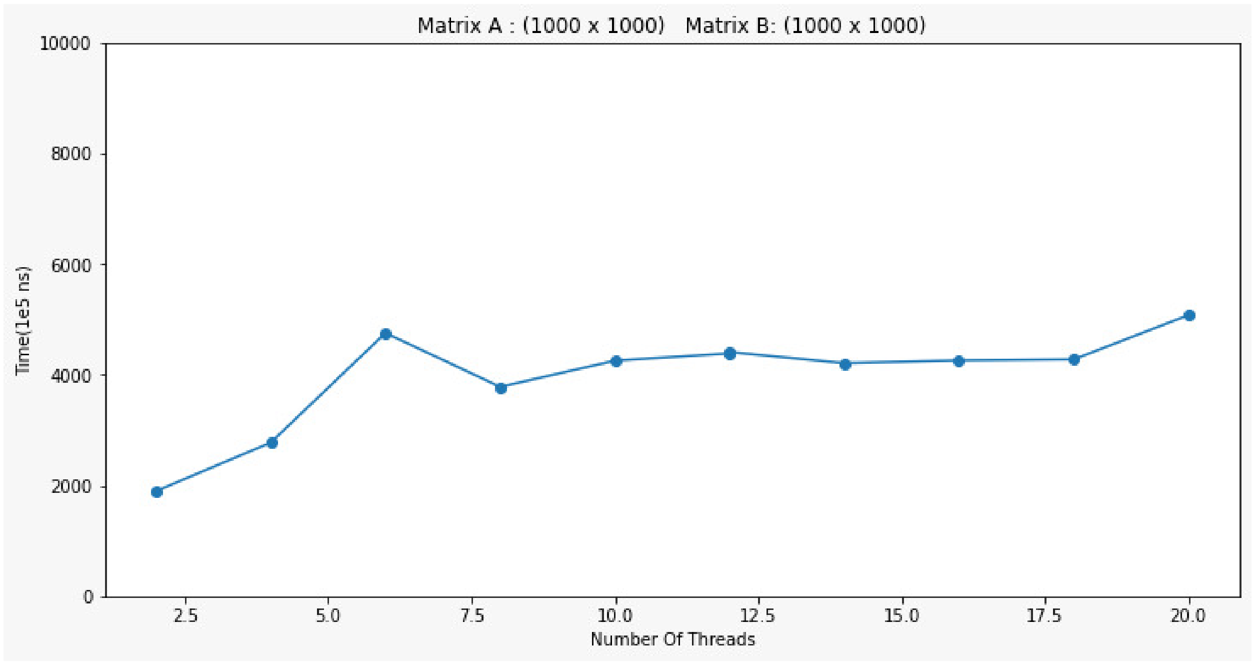
Results :

Results for P1:









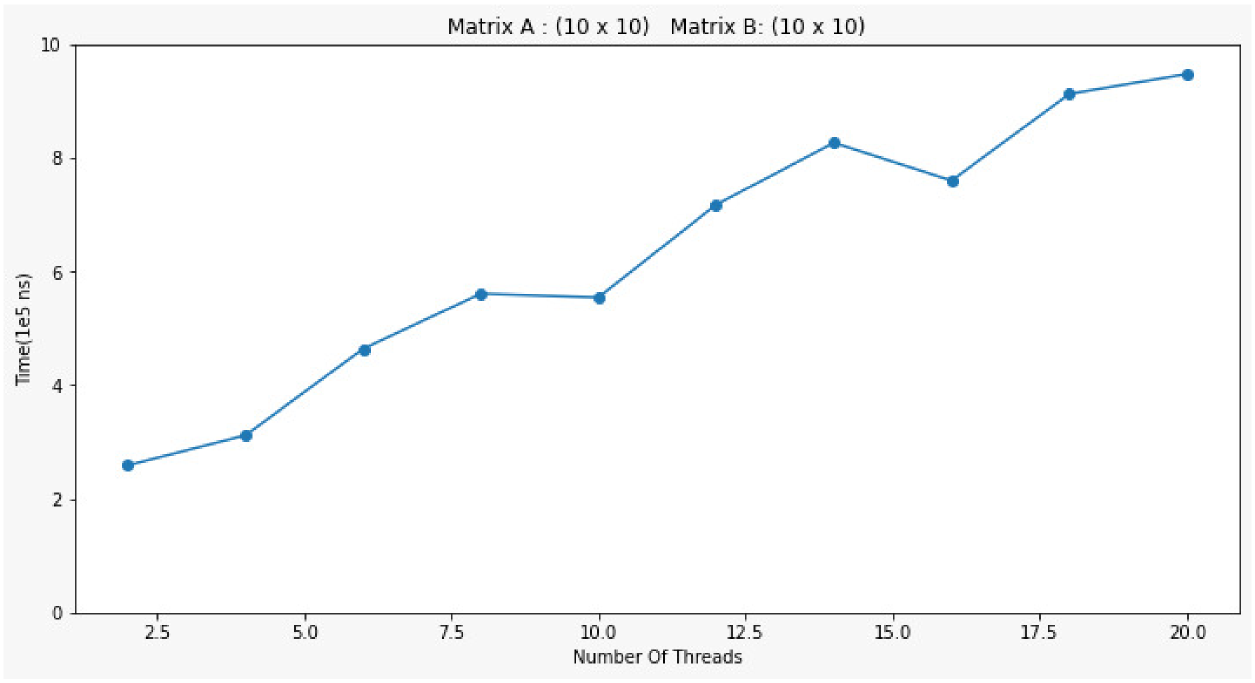
Inferences:

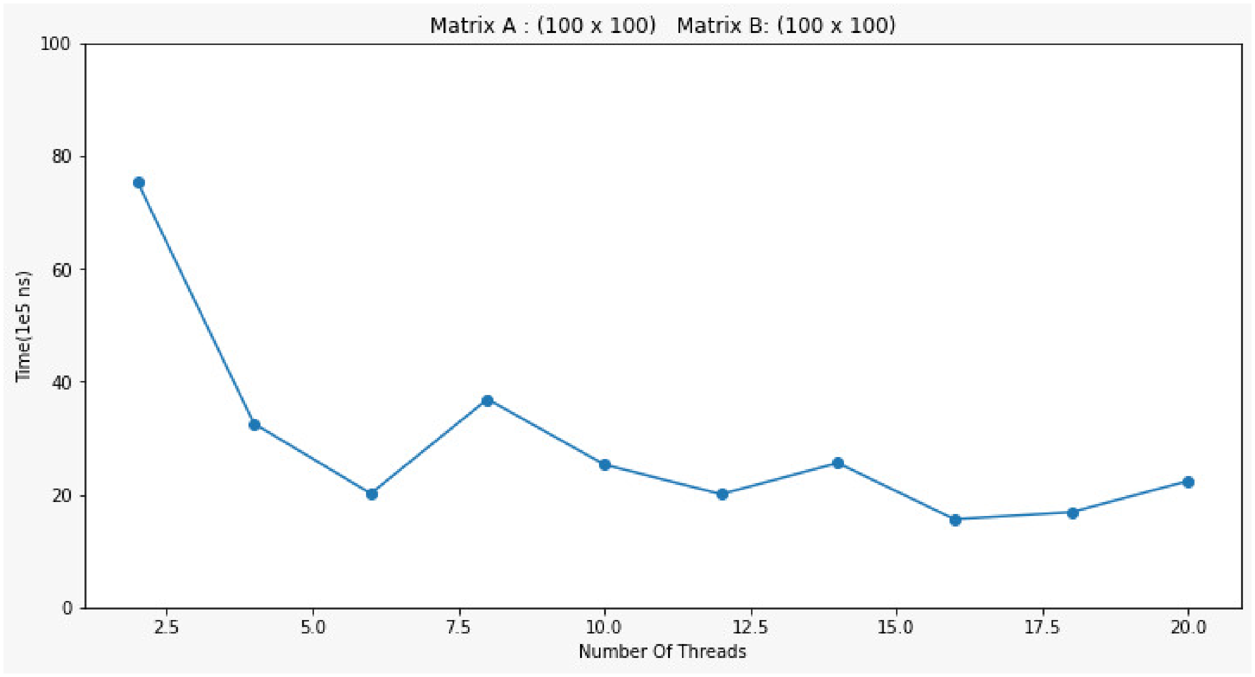
- We notice from the above analysis that for P1, where we have to read the files, the reading time first decreases till a point, and then an increase is seen.
- This observation is due to the following :
    - As multi threading allows for multiple parts of the file to be read by different threads, the time take obviously decreases, but the increase in time is due to the

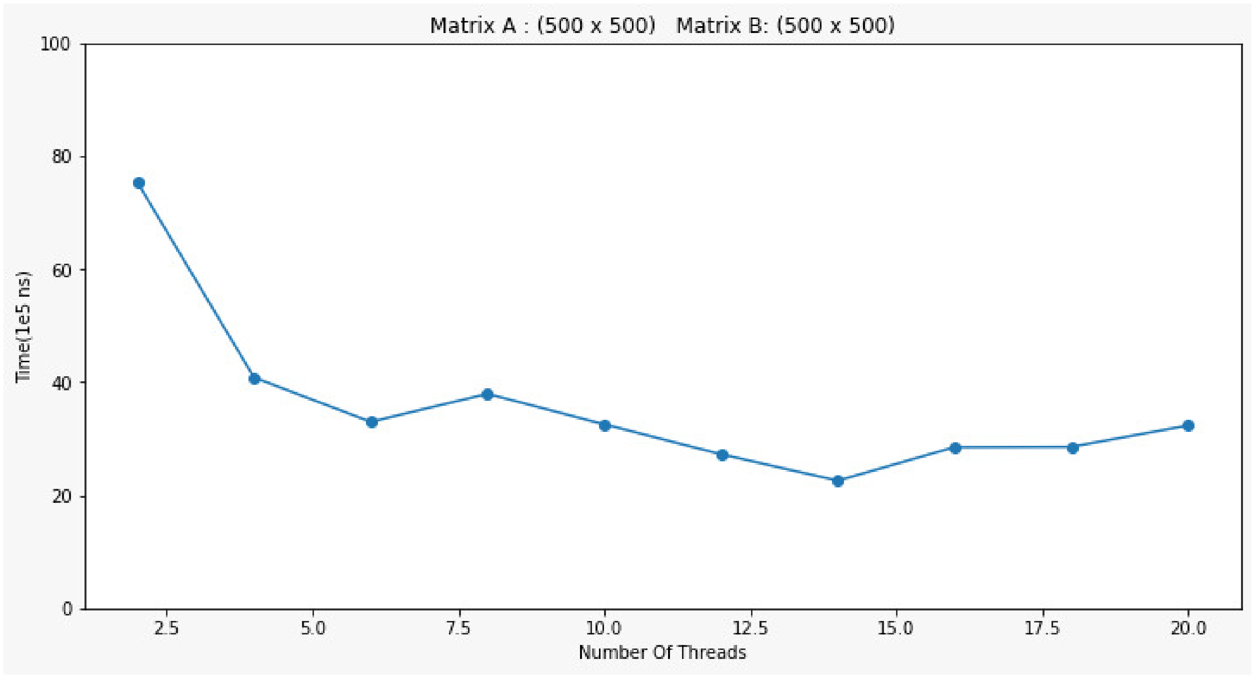
fact that creating and working with threads also comes with its own overheads, and hence after a certain point, the time taken to read increases and continues to increase with increasing number of threads.

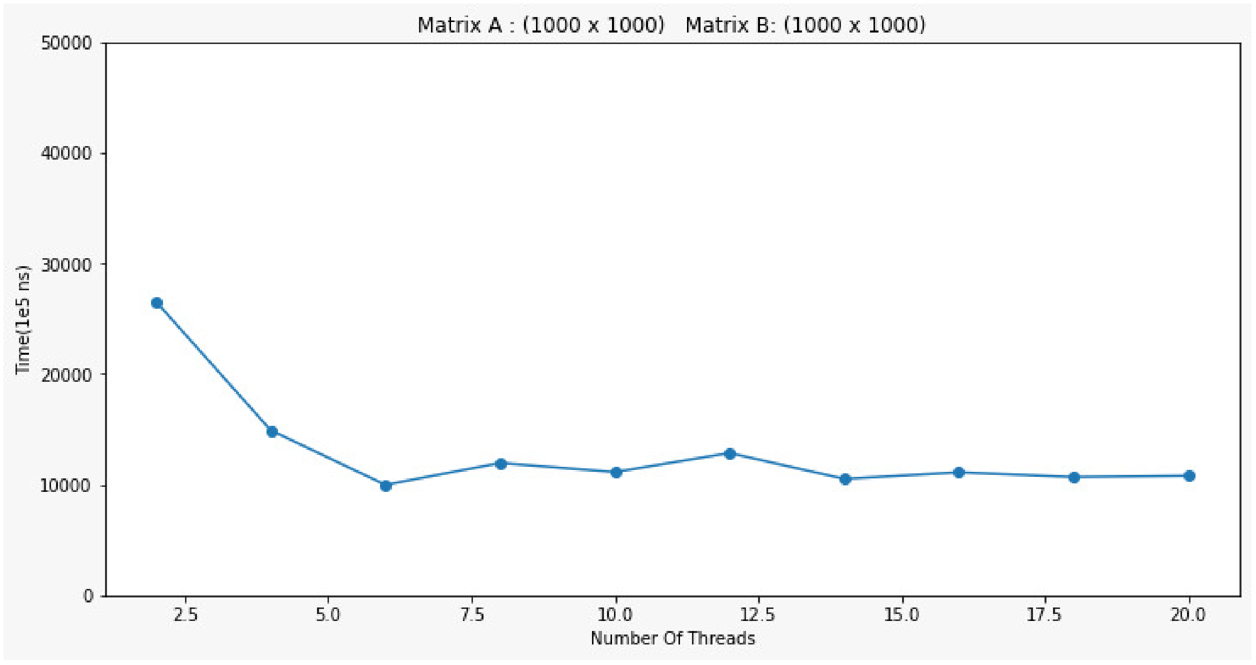
- The optimal amount of threads can be found from the graph, which differs in every case considering the input parameters, and also the specifications of the machine it is run on.
- The random spikes and disturbances observed in the graph may be due to other background processes running on the machine the code is being tested on.

Results for P2:




Matrix A : (100 x 100)   Matrix B: (100 x 100)
Time(1e5 ns)
Number Of Threads
0
20
40
60
80
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Matrix A : (500 x 500)   Matrix B: (500 x 500)
Time(1e5 ns)
Number Of Threads
0
20
40
60
80
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

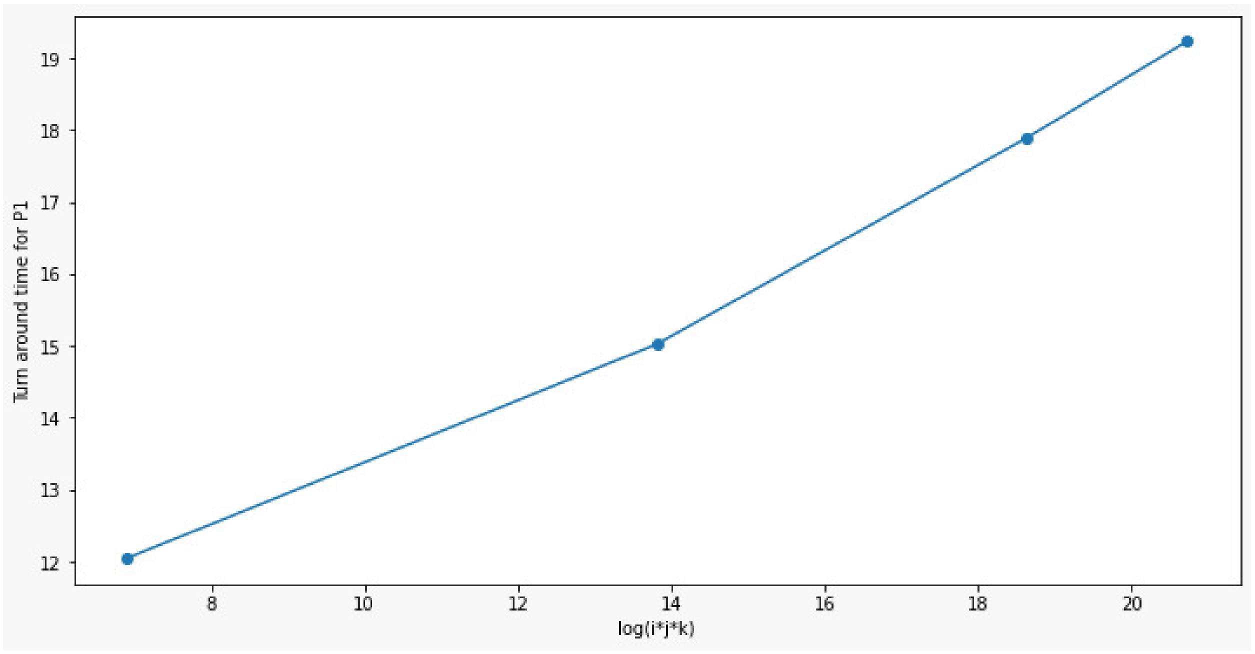


Inferences:

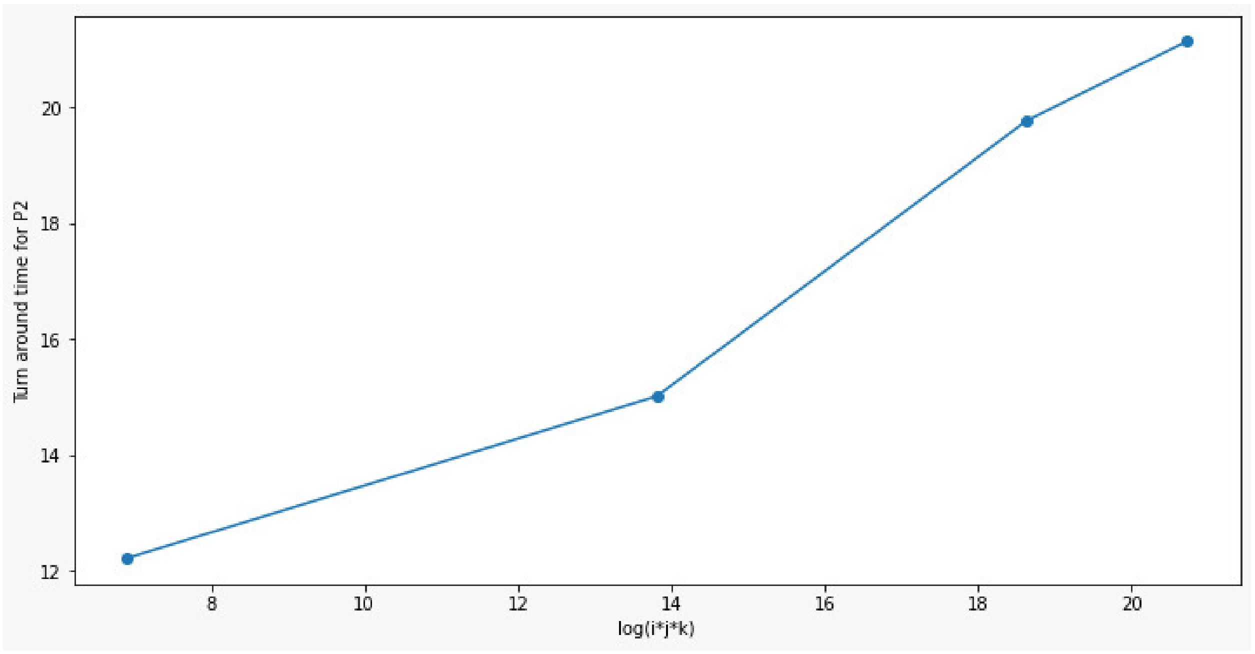
- We notice that for P2, for a smaller size of matrices, increasing the number of threads increases the overhead and hence the time, which makes complete intuitive sense because there are not as many rows to be read, as required with a large number of threads generated.
- On the contrary, we notice that for significantly large sizes of matrices, there is a minima for a certain number of threads, which of course varies according to various factors like the input parameters, and then there is an increase after a certain point which signifies the point where the overhead time dominates. But this difference is not as significant as seen in lower size matrices, owing to the reason that large matrices REQUIRE a large number of threads to perform the operations efficiently.

Results for S:

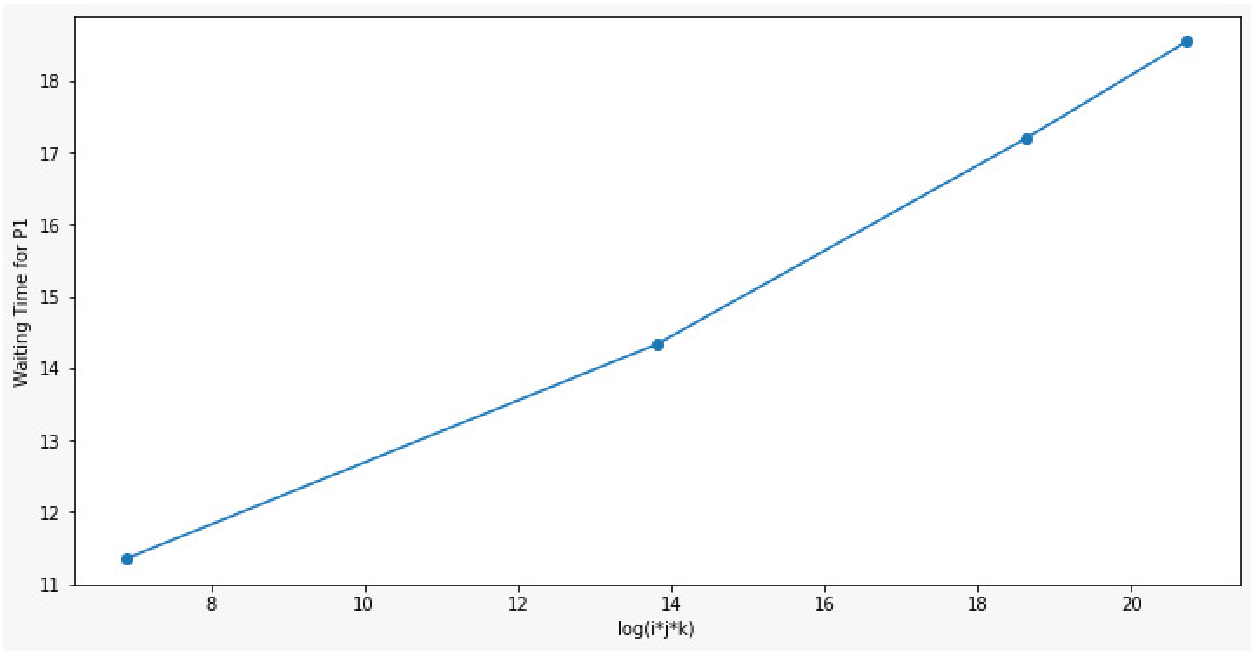
Plot of turnaround time vs workload size for quanta = 1ms for P1



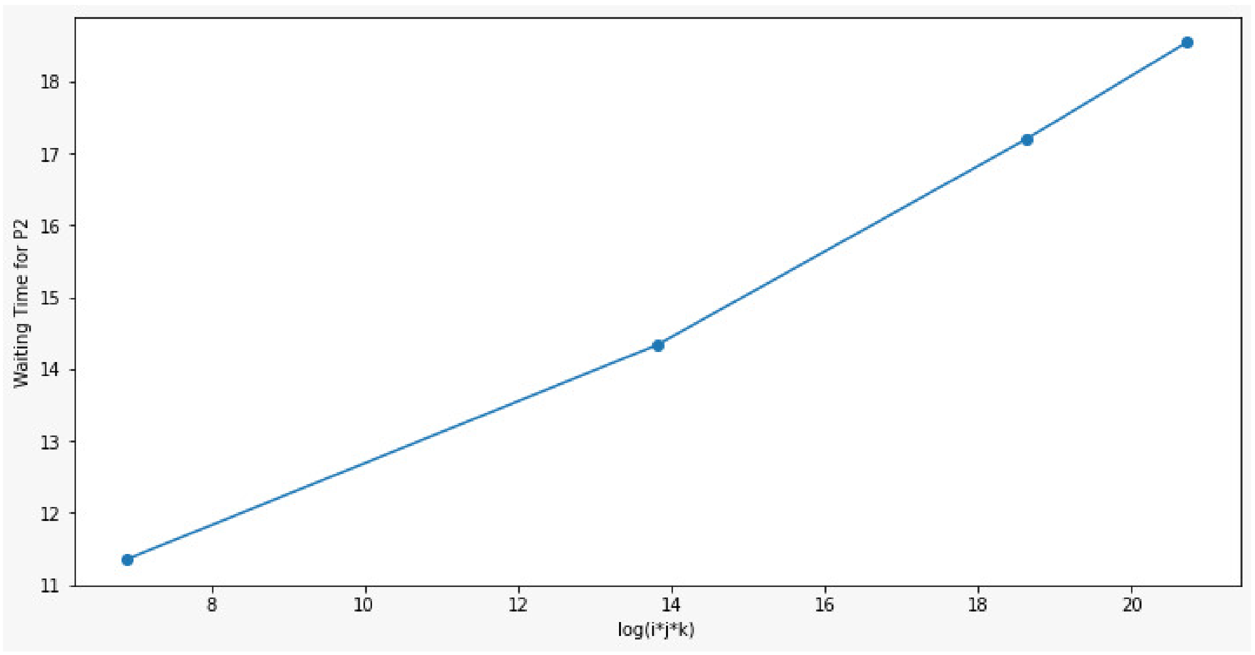
Plot of turnaround time vs workload size for quanta = 1ms for P2



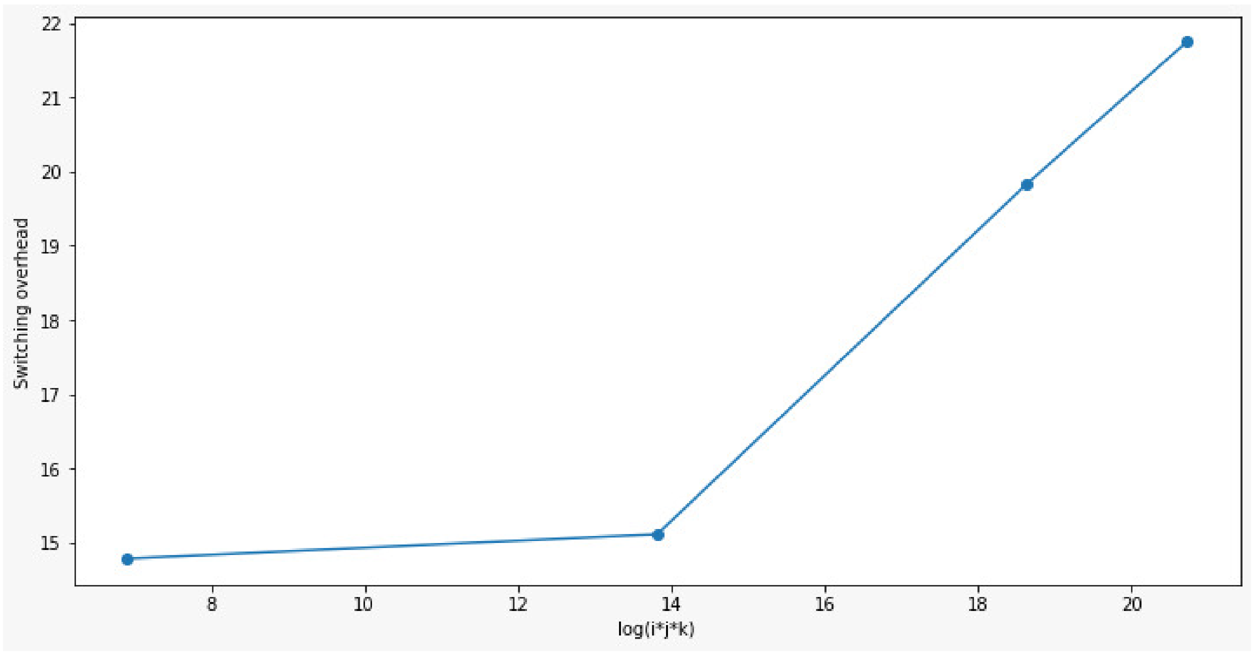
Plot of waiting time vs workload size for quanta = 1ms for P1



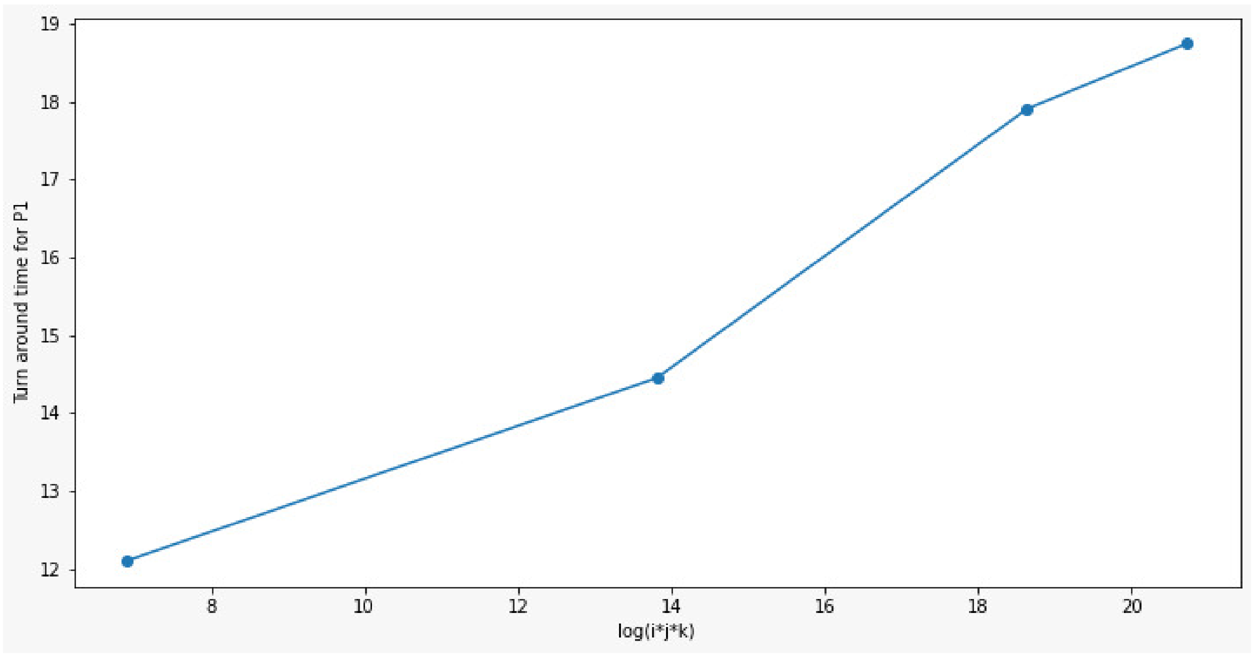
Plot of waiting time vs workload size for quanta = 1ms for P2



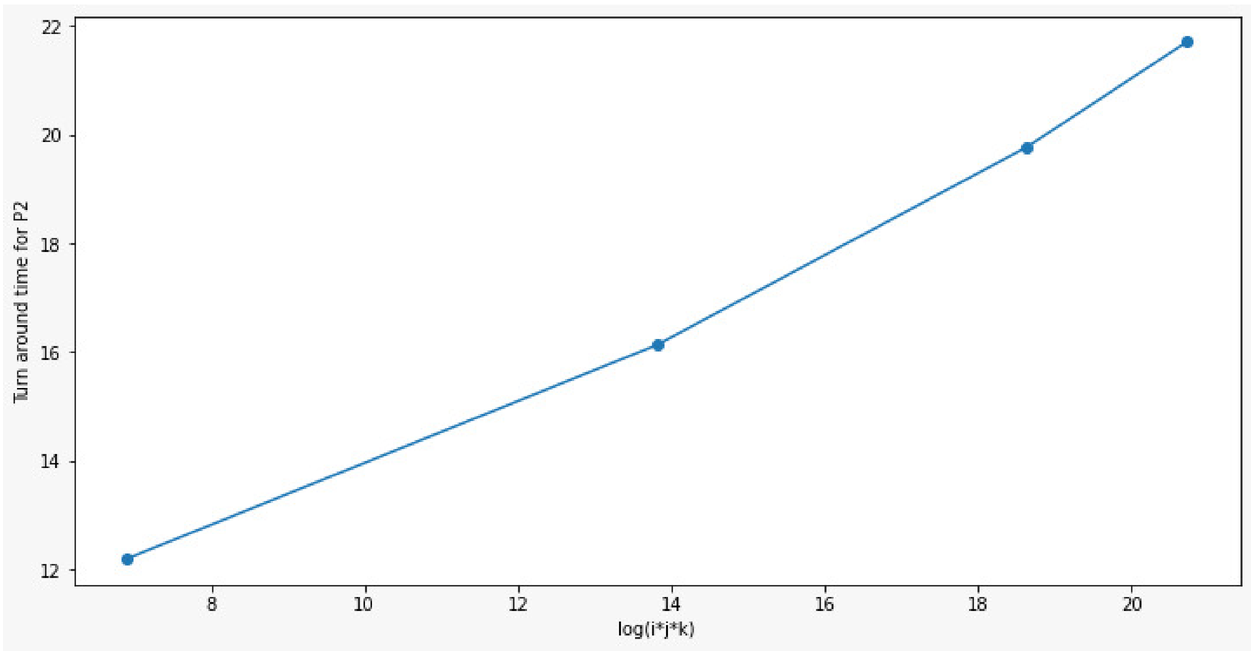
Plot of switching overhead vs workload size for quanta = 1ms



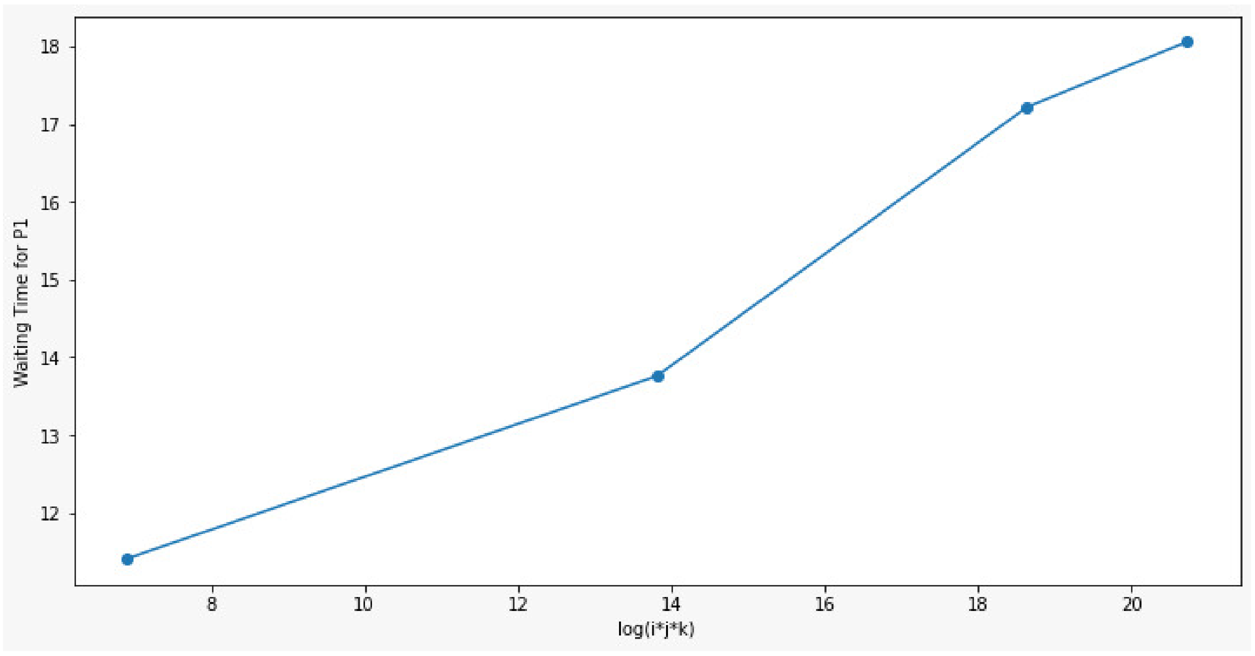
Plot of turnaround time vs workload size for quanta = 2ms for P1



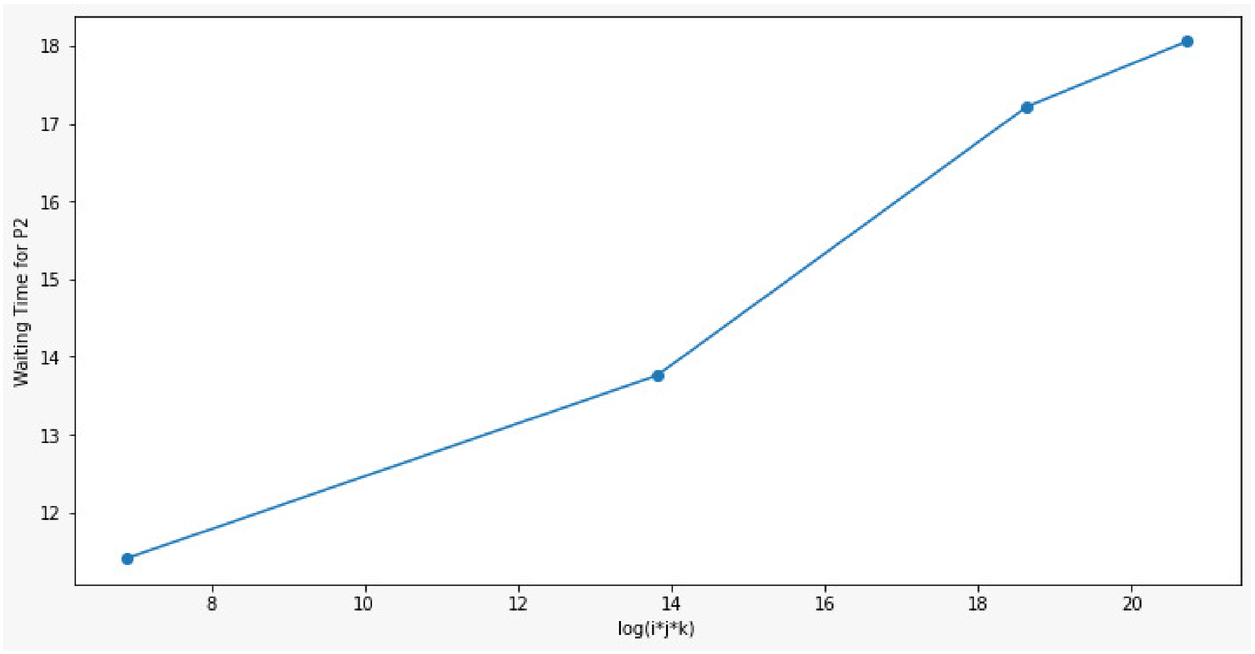
Plot of turnaround time vs workload size for quanta = 2ms for P1



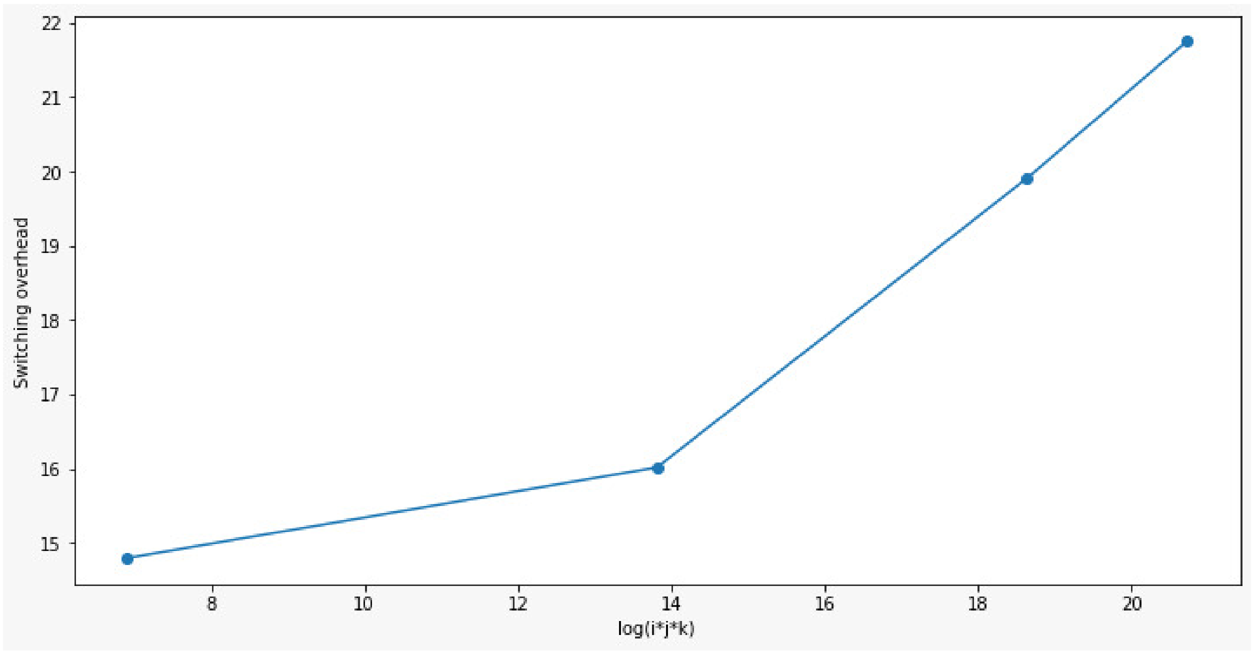
Plot of waiting time vs workload size for quanta = 2ms for P1



Plot of waiting time vs workload size for quanta = 2ms for P2



Plot of switching overhead vs workload size for quanta = 2ms



Inferences:

- For the scheduler, we notice that the switching overhead is more when the time quantum = 1ms. On the contrary, for q = 2ms, it behaves as we expect. This is owing to the fact that the time spent for context switches are more for lesser time quantum, and hence the overhead increases.
- In addition to this, we come to the conclusion that Matrix Multiplication is an algorithm which can be more parallelized as compared to file read(which is in order of O(n^2) though it is more time complexity wise(O(n^3)). Hence, the turnaround time is less for P2 as compared to P1 in general. But, as the workload size increases, the turnaround time of P2 can be more than that of P1 due to higher asymptotic time complexity.