REPORT

Assignment 2

By

Group 21

Under The Supervision Of

Dr. Dipanjan Chakraborty

Birla Institute Of Technology And Science, Pilani

Hyderabad Campus

(November 2022)

CS F372

OPERATING SYSTEMS

Group 21

Contributors

Name	Email-ID
Parth Kulkarni	f20190706@hyderabad.bits-pilani.ac.in
Vidhani Hiten Jitendra	f20190812@hyderabad.bits-pilani.ac.in
Arjun Muthiah	f20190374@hyderabad.bits-pilani.ac.in
Sailesh Duddupudi	f20190632@hyderabad.bits-pilani.ac.in
Vibha Narendra	f20191302@hyderabad.bits-pilani.ac.in

TABLE OF CONTENTS

Title page……………….……………………..……………….…………1

Optimizations and Race conditions for P1……………………………….…………5

Optimizations and Race conditions for P2………..……….……………….………6

Optimizations and Race conditions for Scheduler…….……………….……………8

Analysis of Total Turnaround Time v/s Workload Size……………..……………10

Analysis of Waiting Time v/s Workload Size…..……………………..…………12

Commenting on Switching Overhead…………….…………..……..………13

Analysis of Number of Threads v/s Time in P1 and P2…...……………..……15

References……………….……………………..……………….…………18

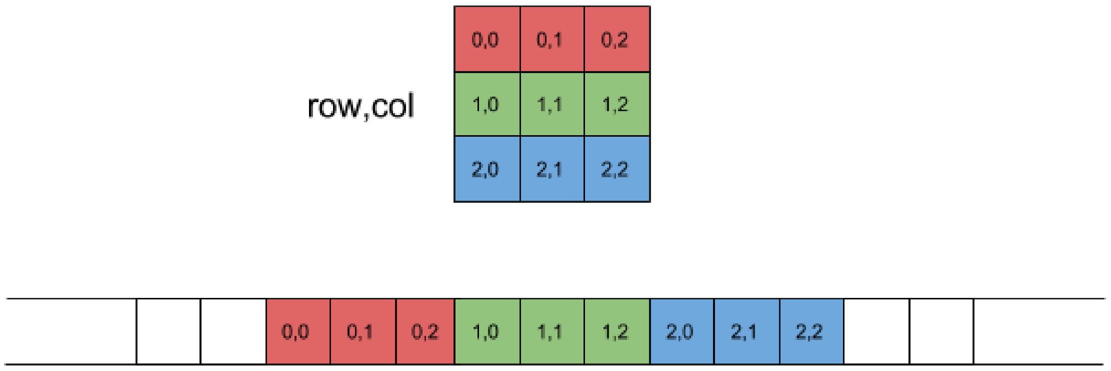
Notations

- Input matrices - A, B
- Output matrix - C
- Number of threads - n_t
- Dimensions of the matrix A - dim1 x dim2
- Dimensions of matrix B - dim2 x dim3

Optimizations and Race conditions for P1

Optimizations for P1 :

For matrix reading, we are first making a pass through the matrices to store the indices of the first elements of each row which we later pass as arguments to the thread function. This way, we can parallelly read the matrices by calling threads for rows. We spawn one thread for each input matrix and after that vary the number of threads for matrices A and B to plot the times. We initialize matrix A and B in shared memory and initialize all of its elements to -1. As each thread reads a row, it updates the shared memory values. We have allocated shared memory for the matrices here where we denote the matrices with contiguous 1-D arrays as shown in the figure below. We use this technique to allocate and access shared memory faster. These matrices are initialized to -1 when unread. On reading we update these values to the corresponding values from the files of the input.



RACE Condition for P1:

Since each thread is accessing different parts of the matrices so there is no race condition. Any number of threads can access the same matrix as long as they don't write over the same matrix.

Optimizations and Race conditions for P2

Optimizations for P2 :

We read the matrices A and B shared memory written by P1.

For matrix multiplication using threading, there are broadly two ways to implement it. One way is to make a thread for calculation of each row of C (corresponds to one row of A and the entire matrix B) and the other way is to make a thread for calculation of each element of C (corresponds to one row of A and the one column of matrix B). We went with the first method considering the fact that threads take a lot of time and resources to create and join, and hence the second method will be slower. Also, due to the limitations of c and the input format of the matrices, we cannot read the matrices column by column which means that the first method has no advantage over the second method in reading and calculation speed.

In the above explanation we considered one thread for each row of the C matrix. Now to implement it for a different number of threads, One way to do this is by creating **n_t** threads for calculating the first **n_t** rows, joining them and then looping the entire process until all rows are processed. But this way, every time a thread function runs it calculates only one row which is slow and also, for every iteration of the loop pthread_join has to be called which will also be slow. Hence in our implementation we distributed the rows equally among the threads. We wrote an algorithm for the same. Say,

ceil = ceiling(number of rows of C / number of threads)
flr = floor(number of rows of C / number of threads)
rem = number of rows of C % number of threads

Then the split is such that :
rem threads calculate **ceil** number of rows each and the rest of the threads calculate **flr** number of rows each. (**rem*ceil** + (**n_t-rem**)***flr** = **dim1**)
The starting index and ending index of the rows that should be read in a matrix is passed as an argument to the thread function.

RACE Condition (for P2) :

Multiple threads are accessing the same matrices but are accessing different parts of them, therefore, there is no need to handle a race condition. But we have to make sure that a thread doesn’t try to access a part of the matrix that hasn’t been written into yet. To handle this problem, we initialize all the elements of the input matrix to -1. Then to check whether a particular row is read, we can simply check whether the last element of that row is -1. NOTE : The B matrix will have to be read fully before any pthread executes since for any calculation, we would need the entire matrix.

Optimizations and Race conditions for Scheduler

Notations (used in code):

- Run time of P1: run1
- Wait time of P1: wait1
- Turn around time of P1: TAT1
- Run time of P2: run2
- Wait time of P2: wait2
- Turn around time of P2: TAT2
- Total Run time: run12
- Total wait time: wait12
- Total turnaround time: totTAT
- Total time: totTime
- Switching overhead: switch_ovhd

The way the scheduler works is that we fork twice in it to generate 2 children, and then in each of the children, we place the binaries of P1 and P2 respectively by using execlp as mentioned in the problem statement.

After that, we start and stop the processes P1 and P2 by using SIGSTOP and SIGCONT signals which will start and stop the process along with saving their states at the time the signals are sent, so that execution continues from where the process was terminated.
This starting and stopping of processes is done in a while loop, to simulate round robin scheduling, where each process is run for the amount of time specified by the time quantum(1ms and 2ms) and then stopped and the next process in the circular queue (which here contains just P1 and P2) and is run and so on.
Within the while loop, we're checking if the respective process has ended or not before sending a signal to it or making the other process wait, which ensures that if one process has ended, the other doesn't still wait for the time quantum amount of time before being executed again.

For calculating the times, we're using the clock_gettime() function, which gives a resolution of 100 ns as we need to specify time in granularity of nanoseconds. We can store the time at the call of the function in a timespec structure using this function and then we can calculate the time elapsed between any 2 events by calculating the difference between the time stored in the timespec structures corresponding to the respective events.

Turnaround Time (TAT) : We calculate this by calculating the difference between the time when the end signal is sent by either of the programs back to the main scheduler program and the time when the respective process was first submitted to the scheduler program.

For calculating Total Turnaround Time(TTAT) we take the maximum of TAT1 And TAT2, i.e. TTAT = max(TAT1,TAT2) because the total time between submission and completion of the entire process of reading and multiplying the matrices finishes only when the longer of the two processes finishes.
Run Time: We calculate run time of each process by maintaining a global variable and adding to that, the execution time of that process whenever the process runs according to the Round Robin(RR) schedule.

Total Run Time: The sum of runtimes of P1 and P2.

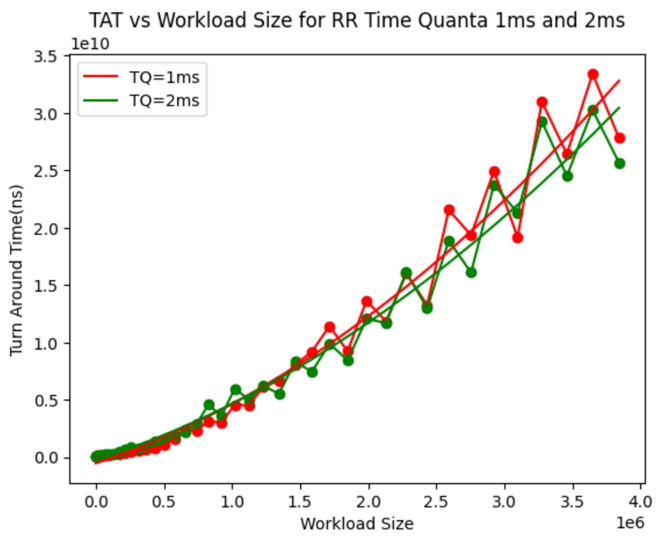
Waiting Time: We calculate the waiting time of each process by maintaining a global variable and adding to that, the wait time of that process whenever the process is swapped out according to the Round Robin(RR) schedule.

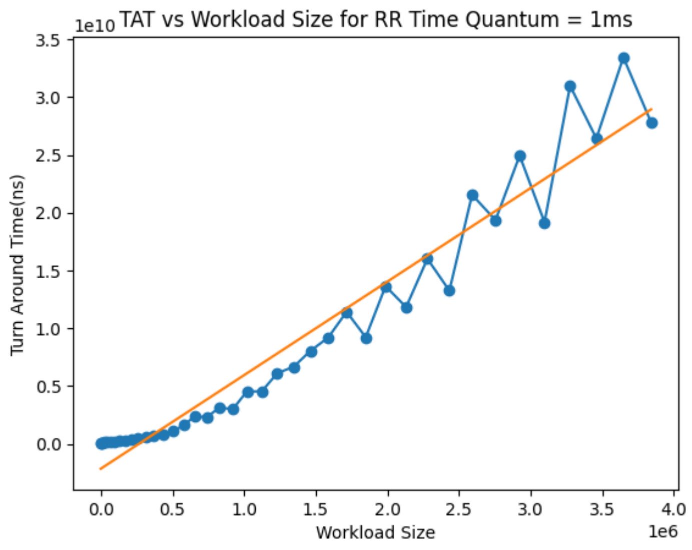
Total Time: We calculate total time by calculating the difference between the times when the while loop of scheduler ends and when the first process was forked.

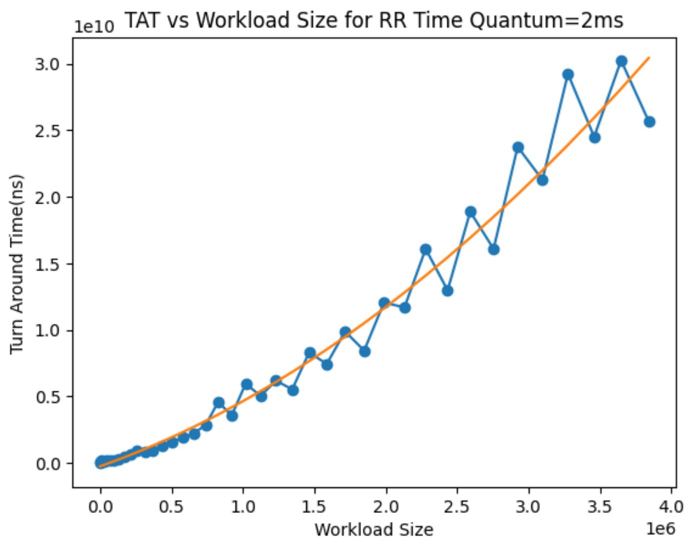
Switching overhead: Difference between Total time and Total Run Time. Here we are calculating the total switching overhead of the entire schedule.

Analysis of Total Turnaround Time v/s Workload Size

Graphs :





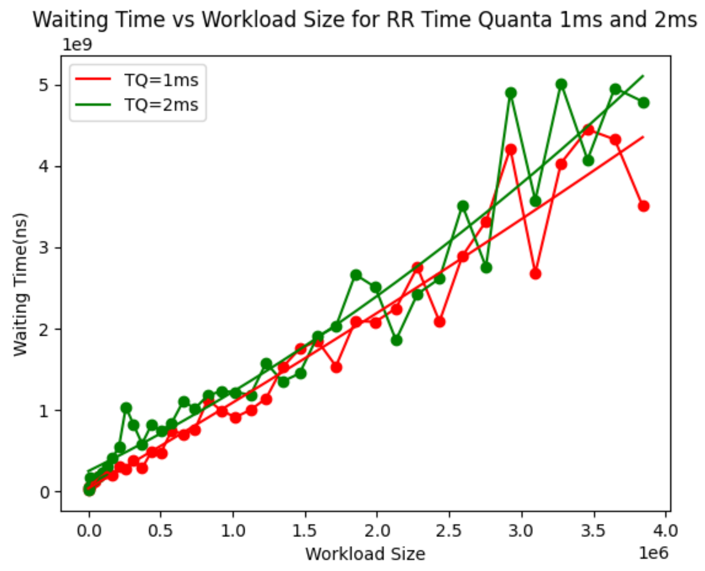


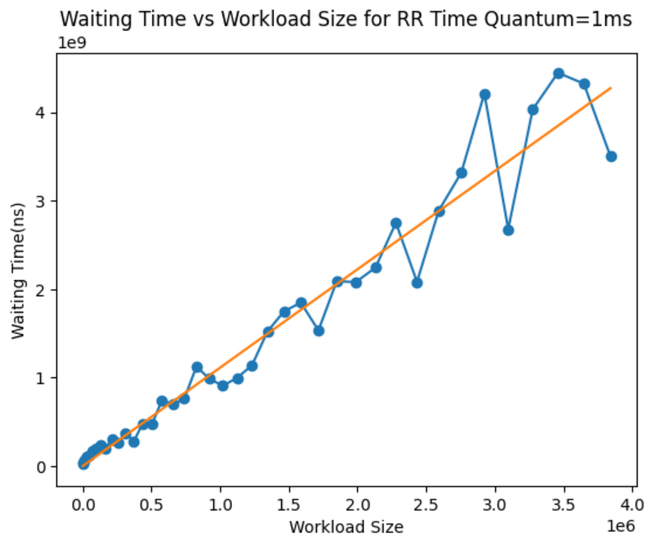
Total turnaround time shows an increasing trend with workload size, as expected, because the more the workload, the more time it'll take for a process to be completed.

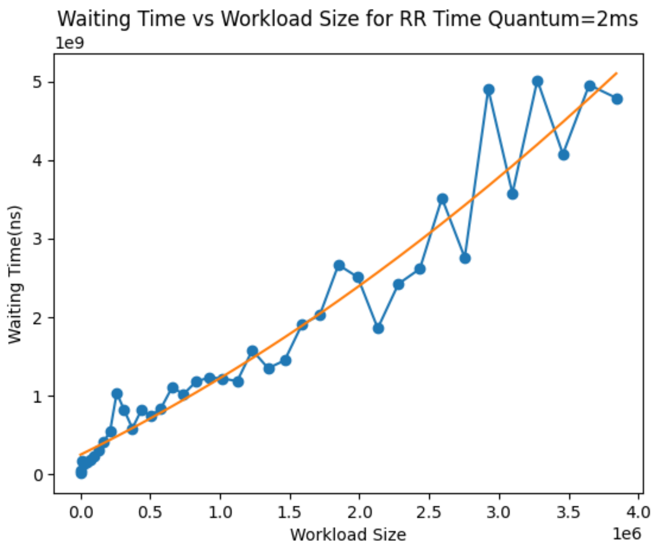
For the same workload size, the turnaround time is, in general, greater for the scheduler with time quantum = 1ms because the process is run for a shorter time in each burst, thus requiring more number of bursts for the process to finish, as there'll be more context switches in case of time quantum = 1ms.

In some cases the total TAT is coming out to be lesser for time quantum = 1ms than for time quantum = 2ms because after a point, when P1 ends and P2 is the only process executing, the total TAT depends also on how fast P2 finishes and not just on the number of context switches. (This exceptional case may also be a result of our scheduler just being a simulator of the round robin scheduling algorithm and inefficiencies creeping in because it's not executing at the kernel level)

Analysis of Waiting Time v/s Workload Size





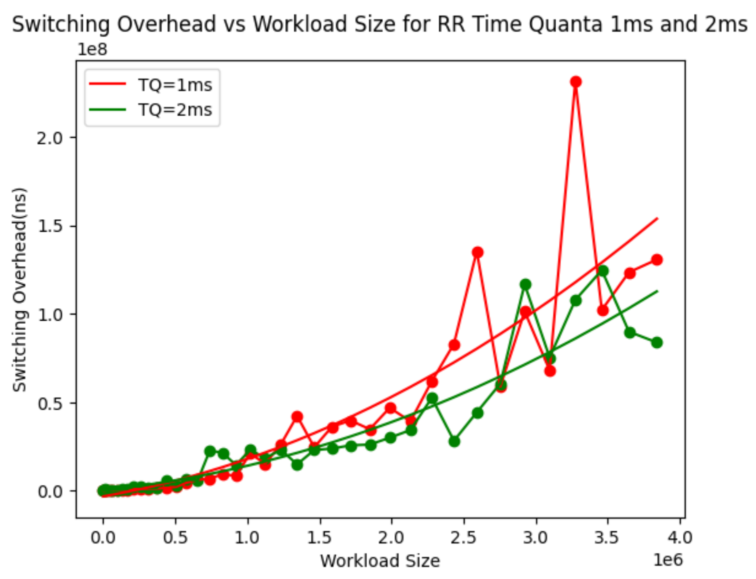


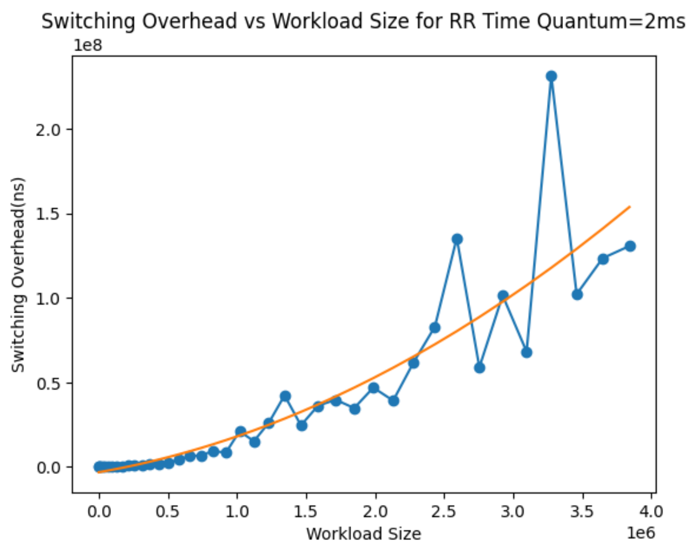
Total waiting time also shows an increasing trend with workload size, as expected, because as the workload increases, both the processes P1 and P2 need more time to run, thus increasing the waiting time of the other.

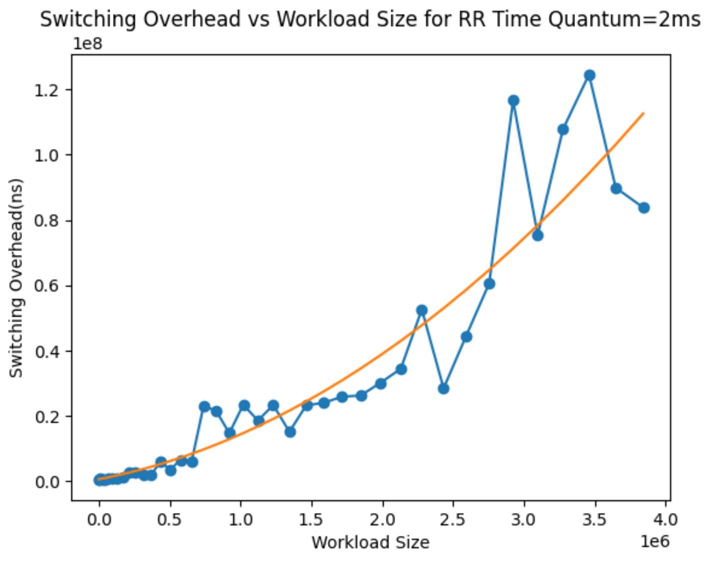
For the same workload size, **total waiting time is, in general, greater for the scheduler with time quantum = 2ms** because each process now waits for more time while the other is executing than in the case of 1ms.

In some cases the total waiting time in case of time quantum = 1ms is coming out to be lesser than for time quantum = 2ms because P1 is able to finish faster and thus waiting time of P1 is lesser, which reduces the total waiting time. (This exceptional case may also be a result of our scheduler just being a simulator of the round robin scheduling algorithm and inefficiencies creeping in because it’s not executing at the kernel level).

Commenting on Switching Overhead





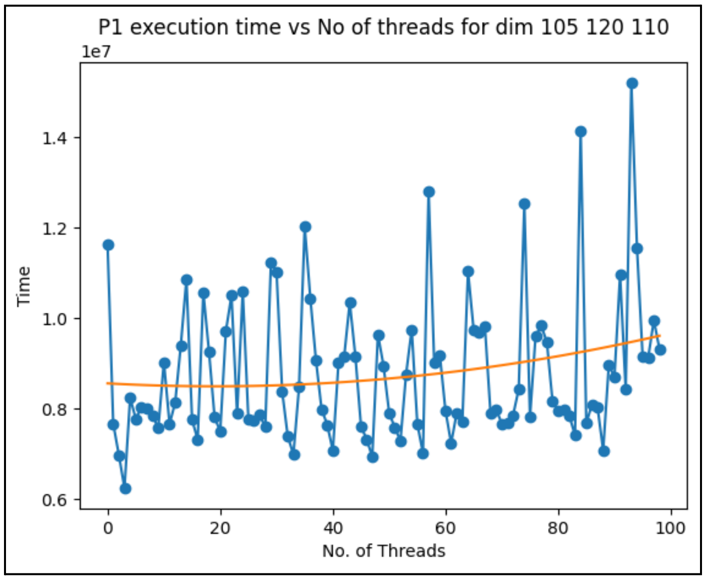


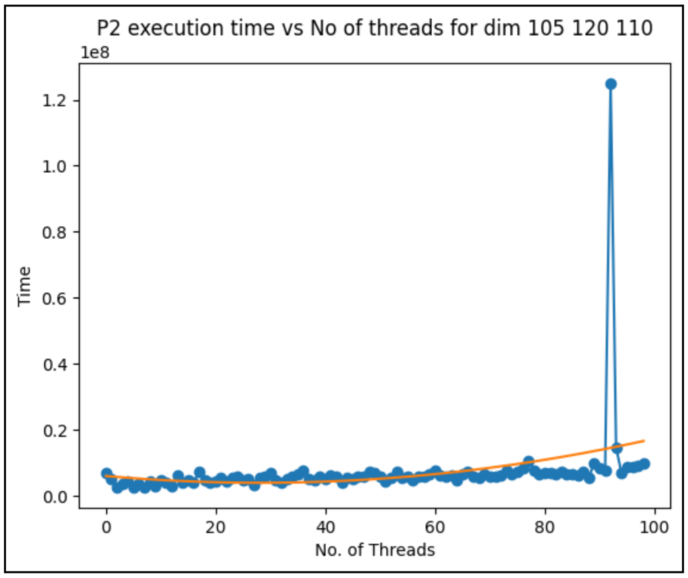
Total switching overhead shows an increasing trend with workload size, as expected, because as workload size increases, both P1 and P2 will take longer to run, thus there will be more context switches and thus there is more switching overhead in case of a greater workload size.

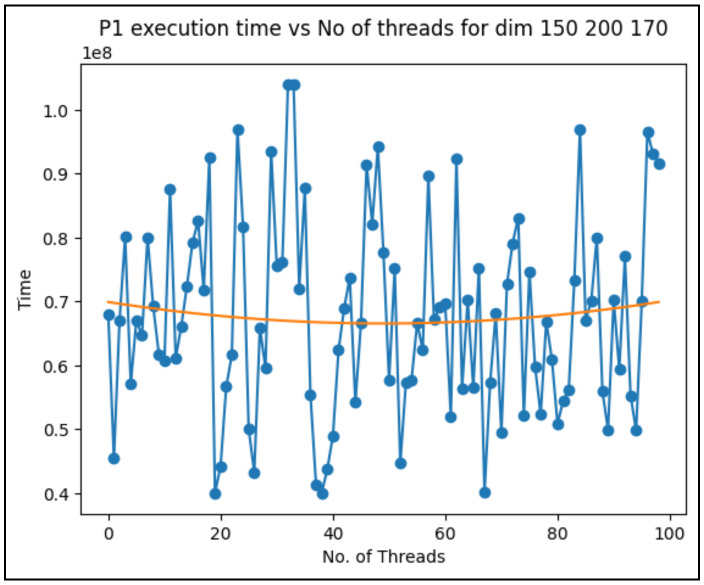
For the same workload size, in general, the switching overhead is greater for the scheduler with time quantum = 1ms because there will be more context switches in case of time quantum = 1ms than for time quantum = 2ms, thus there will be more switching overhead in case of time quantum = 1ms.

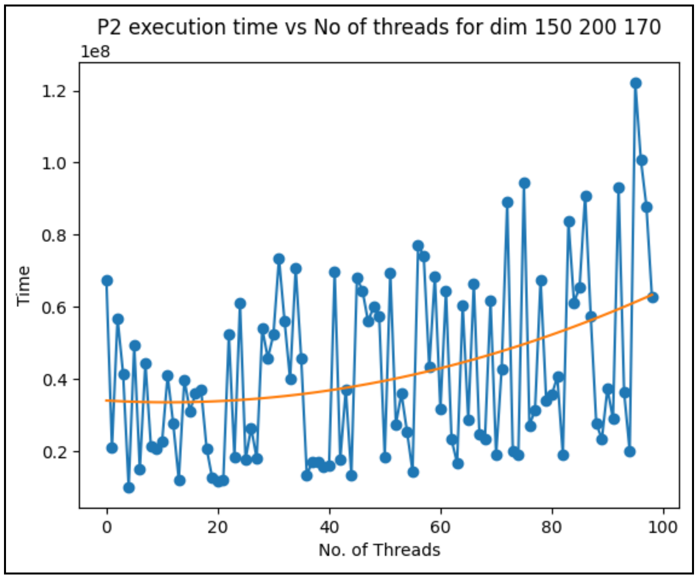
The cases where the switching overhead of time quantum = 1ms is lesser than that of time quantum = 2ms is probably because after a point, when P1 ends and P2 is the only process executing, there's not much switching overhead during that time, and it'll only depend on how fast P2 is able to finish its execution. (This exceptional case may also be a result of our scheduler just being a simulator of the round robin scheduling algorithm and inefficiencies creeping in because it's not executing at the kernel level)

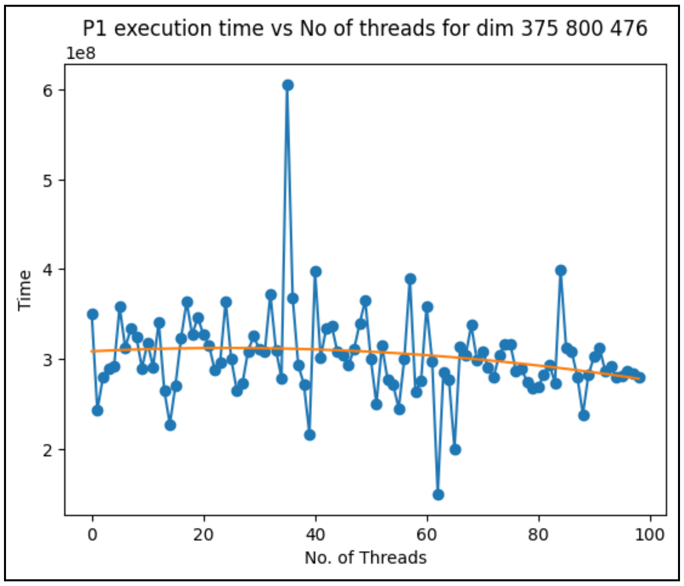
Analysis of Number of Threads v/s Time in P1 and P2

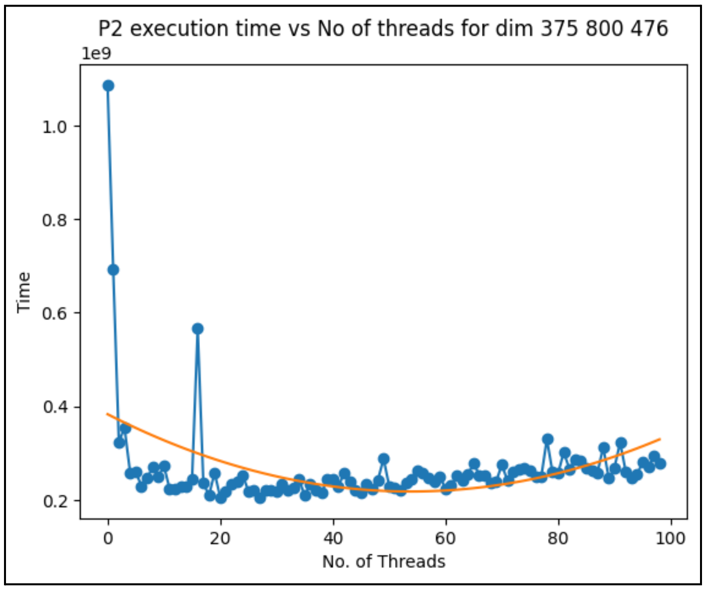


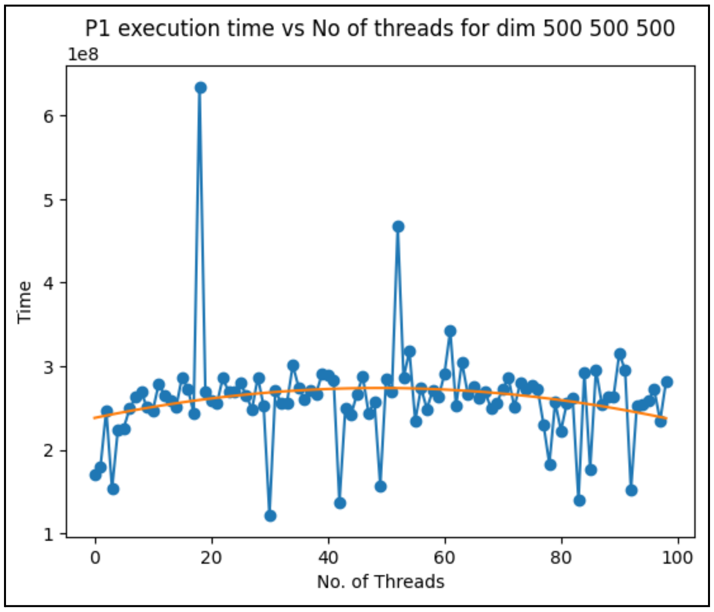


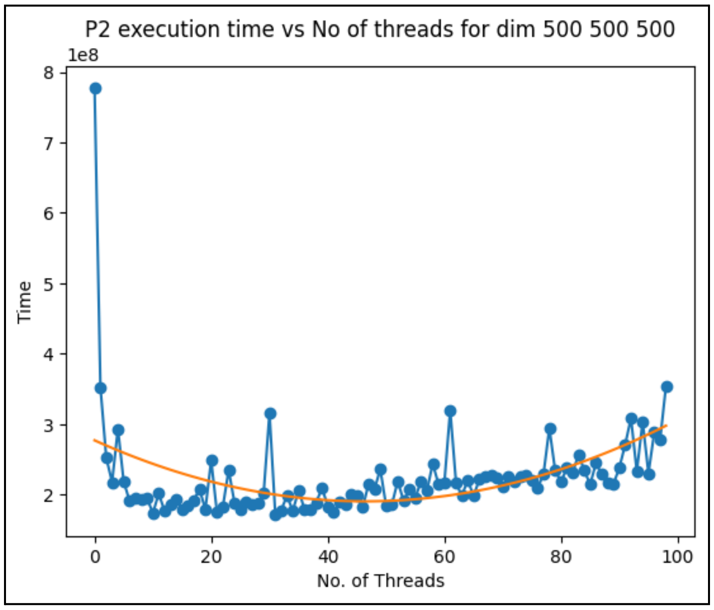


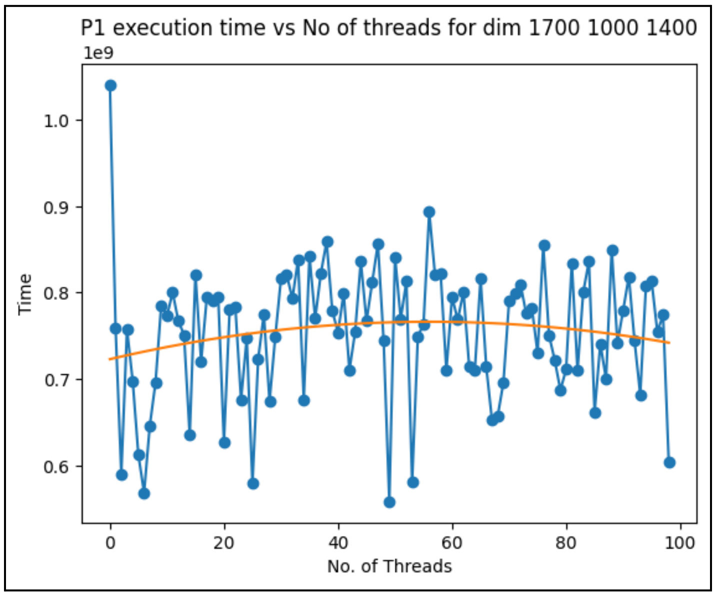


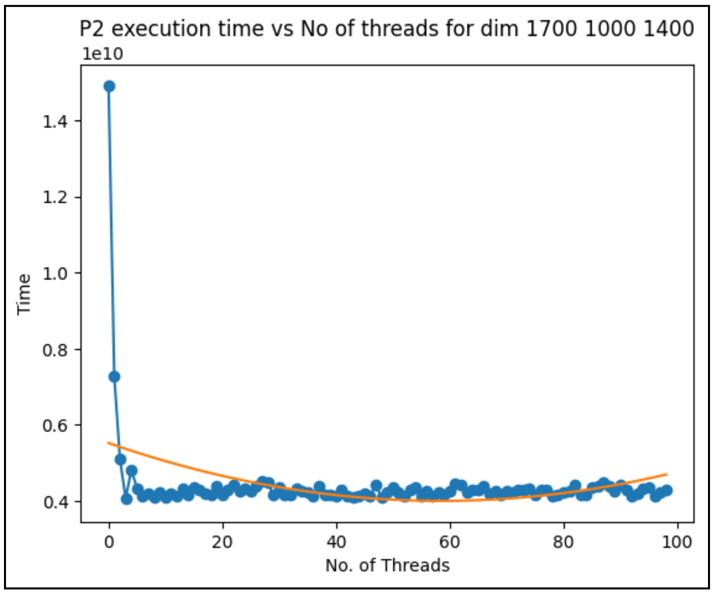

P1 execution time vs No of threads for dim 375 800 476
1e8
Time
No. of Threads


P2 execution time vs No of threads for dim 375 800 476
1e9
Time
No. of Threads


P1 execution time vs No of threads for dim 500 500 500
1e8
Time
No. of Threads


P2 execution time vs No of threads for dim 500 500 500
1e8
Time
No. of Threads


P1 execution time vs No of threads for dim 1700 1000 1400
1e9
Time
No. of Threads


P2 execution time vs No of threads for dim 1700 1000 1400
1e10
Time
No. of Threads

Explanation:

We can look at the trend lines to see the impact of the number of threads on the time taken to execute. We wrote a python wrapper code to automatically generate matrices and plot graphs for P1 and P2. When we plotted the graphs, we noticed that they followed a quadratic trend (be it concave or convex), hence we plotted a quadratic trendline fitted to each graph. (the graphs are sorted in ascending order of dimension sizes)

When we look at the P1 graphs (i.e., reading the matrices) we see that for smaller numbers of rows in matrix 1 (dim1), it has a convex trend while at higher dim1, it has a concave trend. For the smaller sizes, a convex trend implies that increasing the threads initially decreases the time taken to execute until a minimum after which time taken increases with increasing number of threads. This is due to the fact that too many threads would lead to usage of more resources and time as we need to create and join more threads. For larger sizes, there is a concave trend, this happens when dim1 is too high, initially as the number of threads increase, the cost to create and join threads doesn't make up for the benefit of parallelizing. As the number of threads becomes very high, parallelization takes over the cost of creating and joining threads are required to read the matrices quicker.

All the P2 graphs are convex (i.e, calculation and writing). This means that increasing the threads initially decreases the time taken to execute until a minimum after which time taken increases with increasing number of threads. This is because when the thread size is too large, creating and joining threads consumes a lot of resources and time which the gain from parallelization doesn't make up for. This also means that the minima of the curve would be the optimal number of threads for a matrix of that particular dimension.

Inferences to note:

1) In higher dimension sizes in P2 graphs, we notice the steep decrease in time taken to execute the program as soon as we increase the number of threads from 1. This shows the tremendous gain in speed of program due to parallelization
2) Even for concave graphs of P1, we might see a convex trend if we keep increasing the number of threads, their execution time may increase due to the same reasons as mentioned above. But due to the CPU having a limited number of cores, it is pointless to keep the number of threads too high as well.

References

https://docs.python.org/3/library/subprocess.html

https://en.wikipedia.org/wiki/Signal_(IPC)#POSIX_signals

https://www.ibm.com/docs/en/i/7.4?topic=ssw_ibm_i_74/apis/sigkill.html

https://man7.org/linux/man-pages/man2/shmctl.2.html

https://www.educative.io/answers/what-is-timespec-in-c

https://www.geeksforgeeks.org/memset-c-example/