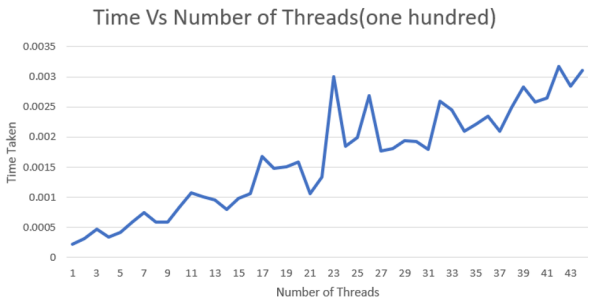
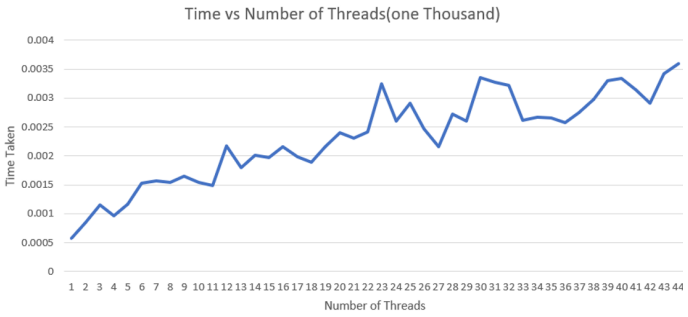
Group-19 Assignment 2 -Report

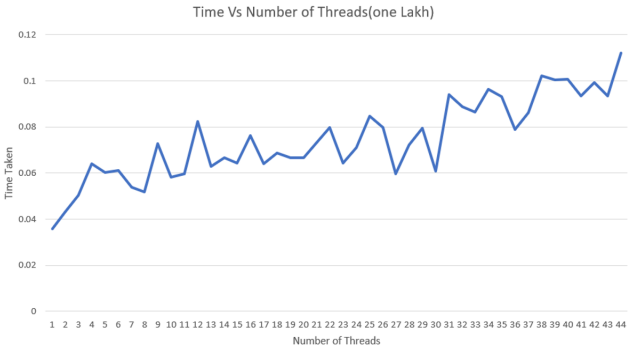
Report Compiled by

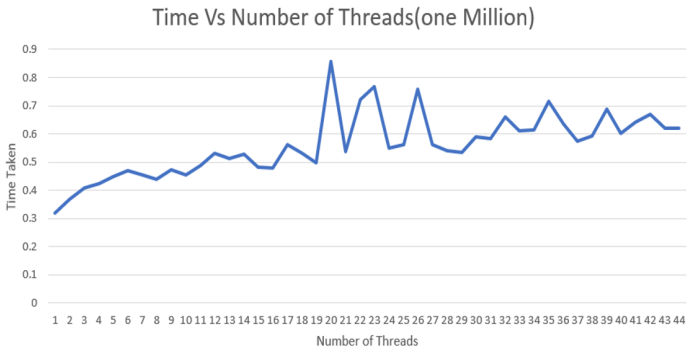
- Karthikeya , 2019AAPS0276H
- Raghuram , 2019A3PS0357H
- MIhir , 2019AAPS0306H
- Sanjeeva, 2019A3PS0485H
- Sharan , 2019A30423H
- Viswa Sai , 2019AAPS0275H
- Anjan , 2019A8PS0367H
- Yasovar , 2019AAPS0226H

Part (A)







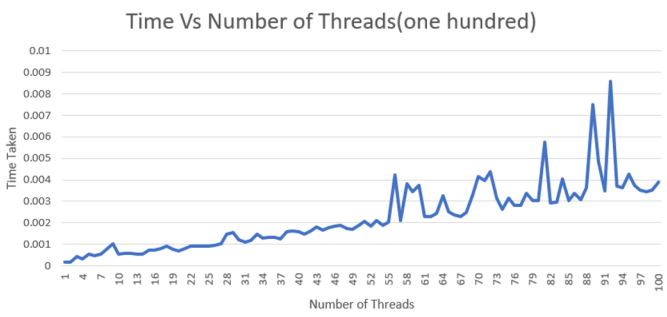


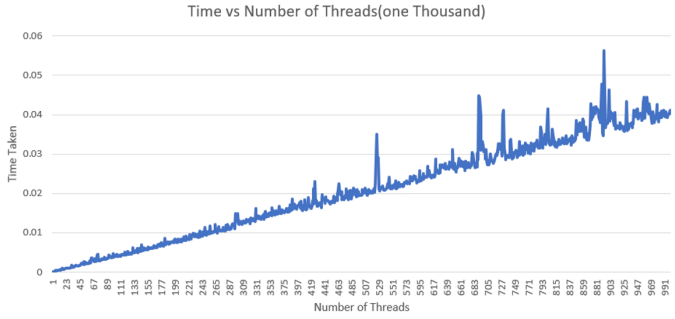
In this assignment, we are to read integers from a file, meanwhile maximizing the amount of threads to solve the problem. In our approach, we created a dynamic array using ***malloc()*** of size 1 for each thread where first element of each array contains number elements present in that particular array. When thread reads a new element, the size of dynamic array is increased by 1 using ***realloc()***. New element is added to this newly created space and size of array is updated in first element of the array.

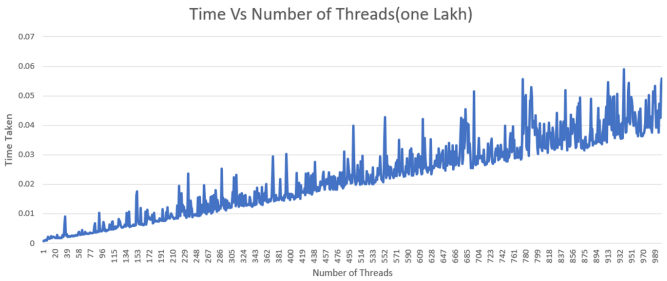
When ***realloc()*** is used for increasing size, it has two options to choose from. One, is add adject continuous free memory blocks to current continuous memory block to cater new size requirements or to search for entirely new memory location of required size and copy old data to new location.

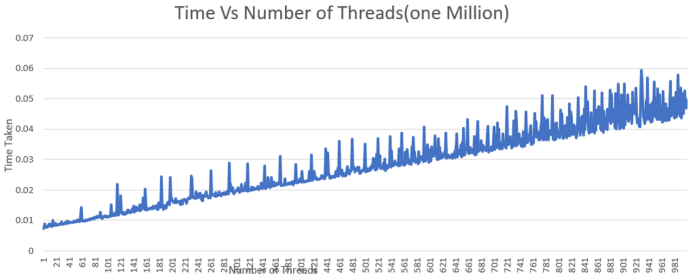
Our analysis from the graph is that as threads are increasing, time taken is increasing .Our reasoning for this is since we are using single core so threads may not run parallel they may run concurrently and we are additionally spending time to create threads and pass the arguments this in turn is resulting in increase of time taken and we are using realloc() from above para we can see that there might be some time that realloc option 1 does not work and go ahead with option2 because of multiple threads this may occur many number of times which might be the reason for the increase of overall time.

Part (B)



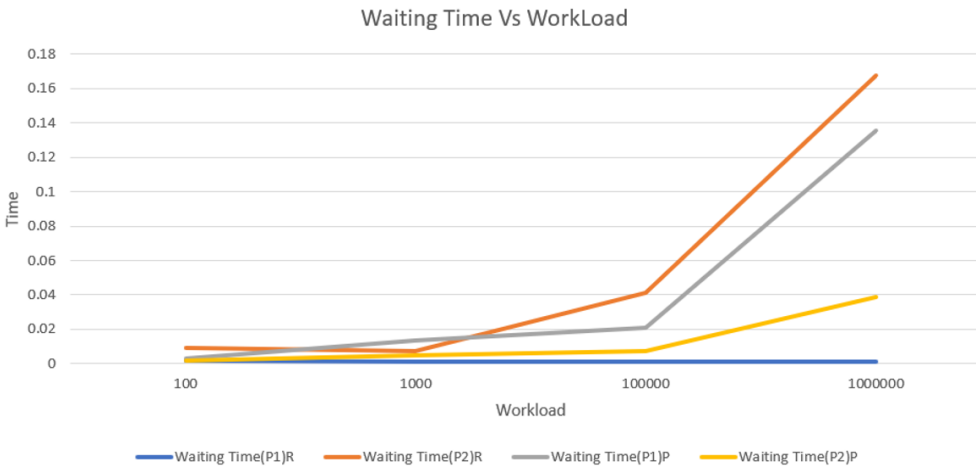


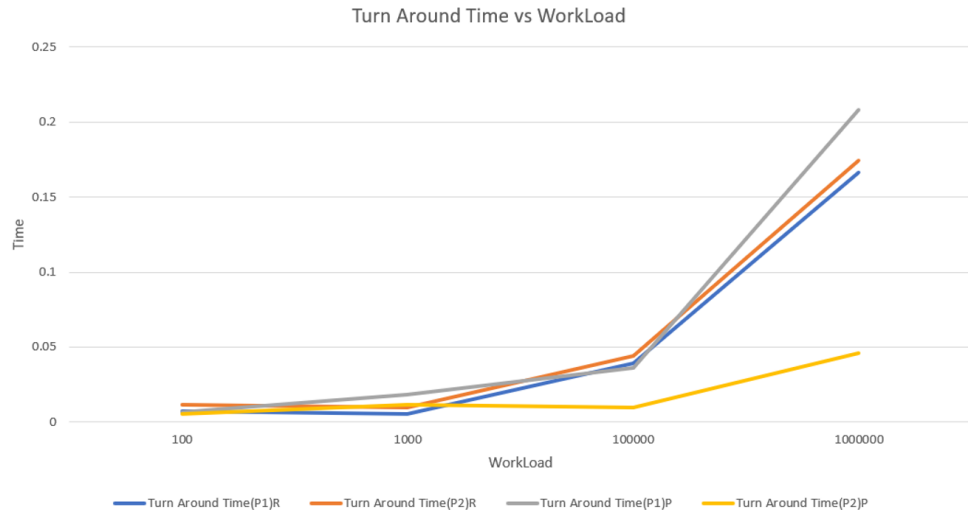




In P2 after receiving the numbers from p1 ,we have used threads and mutex to sum the numbers. In the usage of threads we have divide a part of grid to each thread and used mutex while summing. The final sum is in the critical section that is, only one thread can modify sum at a point of time .From above graph we can infer that as threads are increasing, time taken is increasing our reasoning is that this might be of 2 reasons. 1st reason is same as of p1 reason ie threads may be running concurrently and as we are additionally spending time to create threads and pass the arguments this in turn is resulting in increase of time takenand and other is since the final sum can be accessed by one thread at a time increasing number of threads leads to increase in time.

Part(C)





In the third part of the problem, we are meant to use the processes P1 and P2 created in parts A and B. We are to create a uniprocessor scheduler, S, that simulates round-robin scheduling and Preemptive priority scheduling.

In graph 2, TAT vs Workload is shown for P1 and P2 for Round robin and preemptive, respectively. Observations that in the round-robin way, both have the same values, only P1 lags behind P2, as there are only two processes the code switched between P1 and P2 alternatively for each 1ms of time quanta. In the pre-emptive method to solve the problem as workload increases, P1 takes more time to complete than P2 as the rate-limiting step is the reading of the variables from the file, and P2 only receives the values once they are read.

In graph 1 Wa vs workload P1 has lesser WT than P2 as WT = TAT - Burst time as P2 takes lesser TAT from in graph 2.

Conclusion ;

Since there are only two processes, round-robin and pre-emptive priority scheduling look similar when we look at it theoretically, but when we execute the code, we can see that there is a difference in TAT and WT; this might be due to the anomaly which process executes first and also the difference in switching overhead in each of the scheduling algorithms.