REPORT

Code implementation:

Threads are created using pthread_create and the attributes are the following

```
struct threadAtt{
    int i; //index of thread
    int n;
    int k;
    int **inpMatrix;
    int **resMatrix;
    char *mem; //mutual exlusion method
};
```

Method is the mutual exclusion method that is given through the command line while running the program.

To implement the mutual exclusion method I am changing the thread function.

To reuse threads created: A do-while loop is implemented to ensure that the threads run till all the rows of the resultant matrix are completely calculated which is checked using the condition c>=n . If so this is then the thread is exited using pthread_exit() function.

Critical section: The thread calculates rows from rowStart to rowsStart+rowInc where rowStart is initialised with c at that time. These values are initialised in the critical section and also c value is incremented with rowInc here.

Remainder section: Further calculation of these rows is done outside the critical section i.e in the remainder section.

Entry section: Different mutual exclusion methods are implemented here based on the input method given.

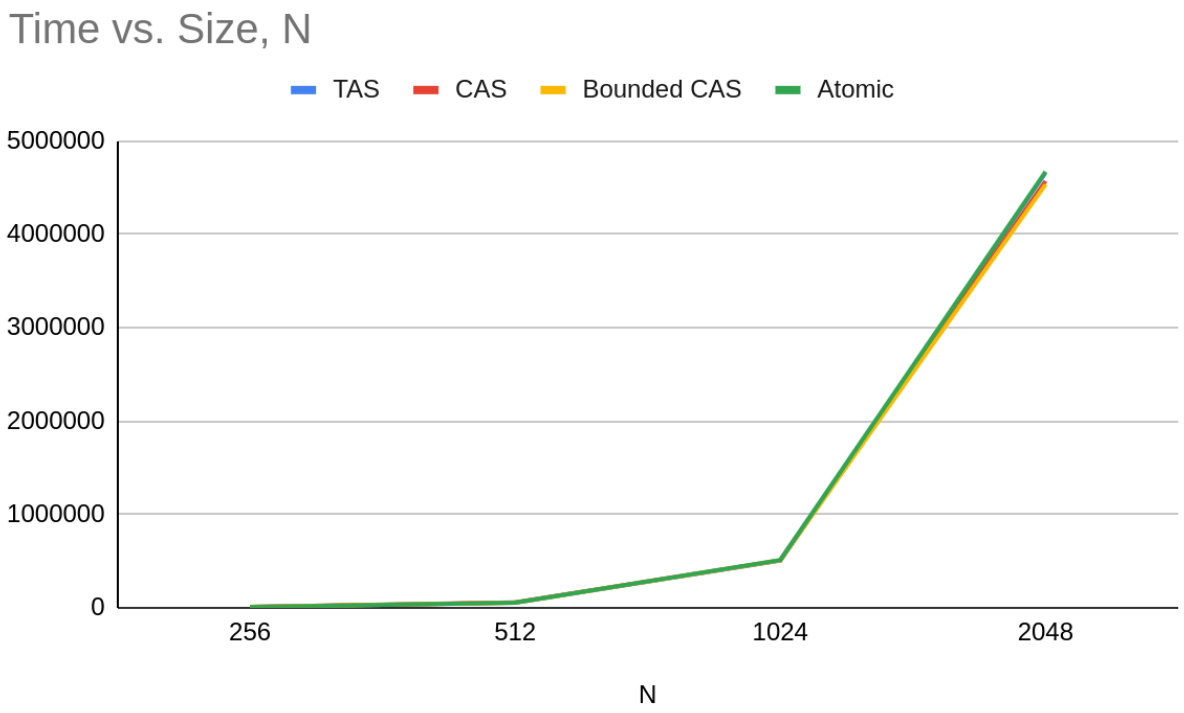
Test and Set (TAS): I have used an in-built function which is in the atomic library i.e test_and_set() . To implement this I have declared an atomic flag lock_TAS, which is initialized to 0,to which this a method.
When the lock is 0 the thread attains a chance to go to the critical section while changing the lock_TAS to 1. The waiting threads remain in the while loop and after the lock being set to 0 gives a chance to one of the other threads that are waiting.

Compare and Swap(CAS): A function is compareAndSwap is used to implement this which uses an in-built function which is in the atomic library *compare_exchange_weak* is used in this. This works very similarly to TAS which releases lock once after ending the critical section and returns 1 and stays in the while loop for the threads that are waiting.

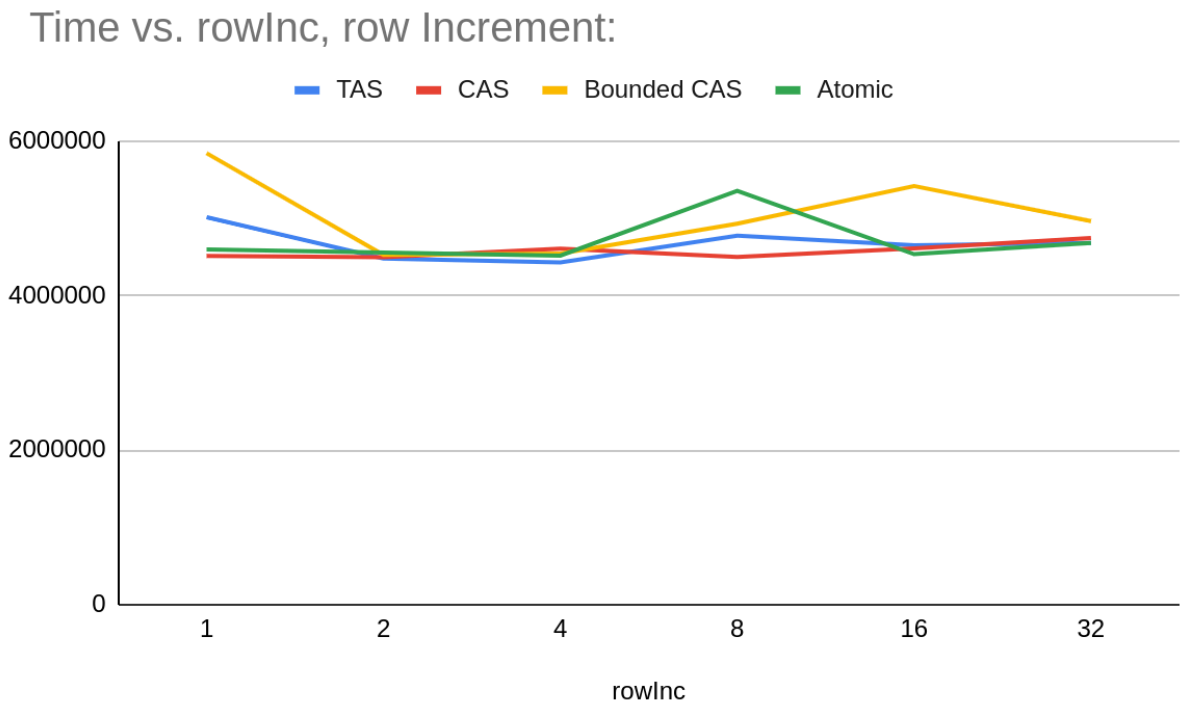
Bounded Compare and Swap : It uses the previously implemented CAS and also maintains a waiting vector for all processes to get a chance at least once within k-1 chances after requesting for the critical section (where k is the number of threads). I used a waiting vector which is resized in the main and in critical section the thread to which next chance is to be given is decided based on the waiting values of threads and also the thread i.e currently running. Then the lock is released or waiting is made false based on the thread chosen and hence only this would break from the while loop and enter the critical section.

Atomic: The variable c_atomic is defined atomically and used directly for giving rowStart and is incremented directly as the underlying implementation of atomic variable in the library takes care of the synchronisation and mutual exclusion part.

Graphs:

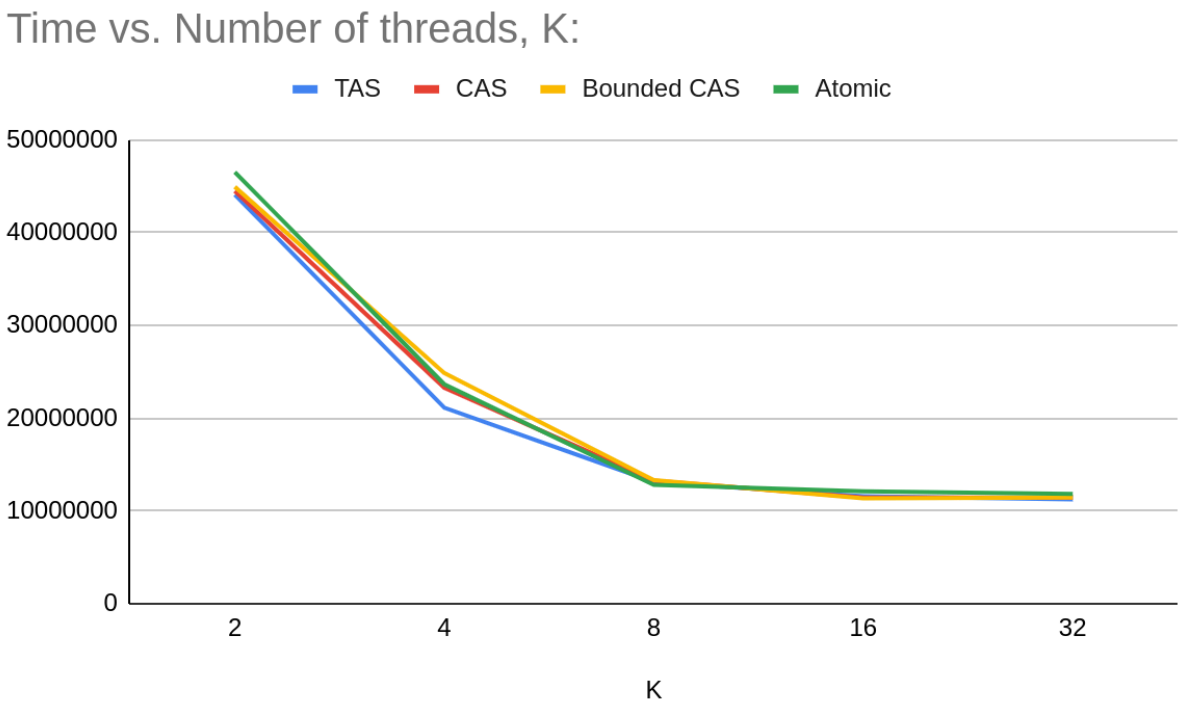


Observation and analysis 1 : As the value of N is increasing the time is also increased for all graphs. This needs to happen as workload increases as thread number is constant.
All the algorithms are performing the same mostly except that bounded CAS has some value less than that of remaining (but not with much difference).

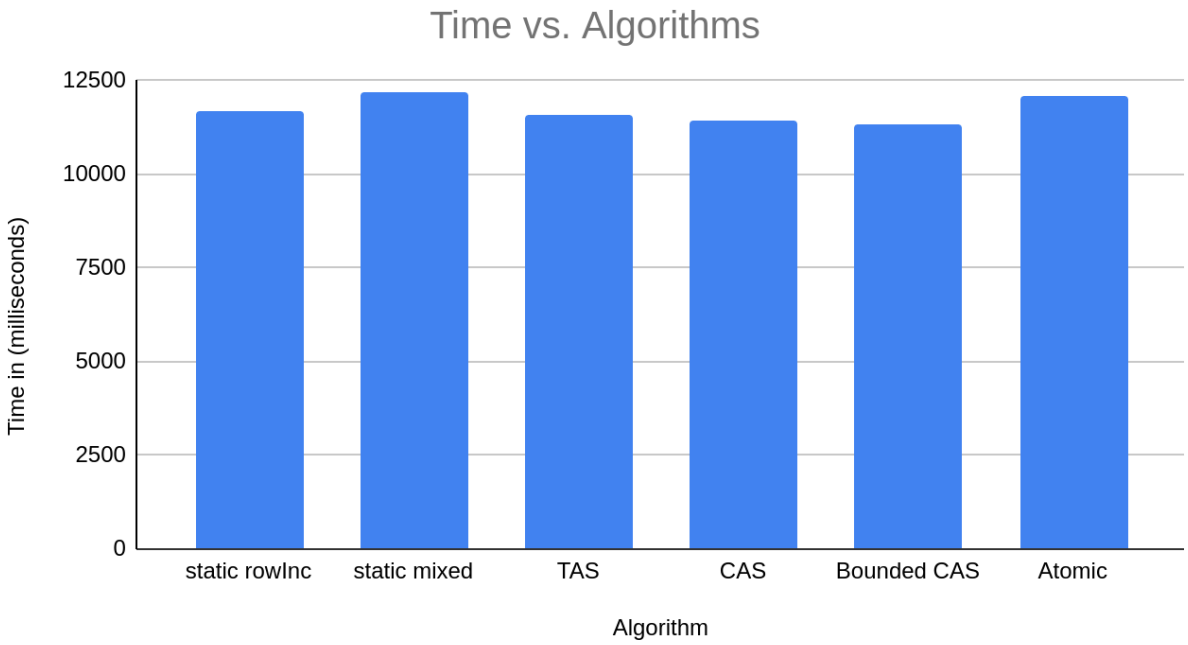


Observation and analysis 2 : As the value of row increment is increasing the change in time is not seen for all graphs. There is no fixed pattern in the trends as well as all the algorithms work in the same way except giving a chance to one thread at a time to the critical section and then computation

for this thread is independent of others.



Observation and analysis 3 : As the value of K is increasing the time taken is decreased as the workload per thread decreases and also due to parallelism of the threads after 8 the value remains almost constant as the parallelism at its maximum can be seen when it is 12 as there are 12 cores in my system, after this the remaining would not much participate in parallelism hence till 8 we can see sharp decrease.



Observation and analysis 4 : The values remain almost the same as the threads do the same work except that in the last four algorithms they have to wait to enter the critical section to get threads assigned to them.