# OS-2

## Programming Assignment-3 Report

Somalaraju Bhavya Shloka

CS22BTECH11056

**Introduction:**

The Main objective of this assignment is to Compare the performance of different Mutual Exclusion algorithms in evaluating the Square of a given Matrix of size N , and using K threads to divide the task into smaller parts.

**Description:**

- The program first reads the input from the input file “inp.txt” and stores the values of N(no.of rows), K(no.of threads), rowInc, and the entries of the input matrix into the Input array.
- An array of K threads are declared using pthread_t keyword, and created threads using the pthread_create() function from the POSIX threads library .
- Each thread is given the threadFunc() function to start executing with.

**TAS:**

➔ Lock variable, Counter variable are globalized as all the threads need to access them while executing.

➔ Counter variable which is of int data type is initialized to 0 and locker_var is an atomic_flag type variable in the atomic library.

➔ In the threadFunc() , an infinite while loop is present to iterate through all threads until counter becomes N (i.e, all rows have been allocated to the K threads) and another while loop with the condition locker_var.test_and_set() function (from the atomic library) to check whether the lock is free or held so that utmost 1 thread can be in the critical section at a time.

➔ In the critical section , the counter is checked whether it is N or not and if it is N, then the thread should set the lock free for another thread to enter

the critical section and it should be breaked out of the loop. If the counter is not N then it is incremented by rowInc.

➔ The remainder section calculates the rows with indices (counter-rowInc) to (counter). Dot product function is declared and defined before the threadFunc().

**CAS:**

➔ The same process is done as above but, to check whether the lock is held or free, __sync_val_compare_and_swap built-in function from the atomic library is used.

**Bounded CAS:**

➔ The same __sync_val_compare_and_swap function used above in the case of CAS algorithm is used . But the difference is that a vector of size K named waiting is created to avoid the problem of starvation (i.e, some of the threads waiting to enter the critical section remain in waiting state).

➔ The Bounded CAS algorithm from the Textbook as reference.

➔ The waiting array stores the boolean values of each thread whether they are waiting or not . The waiting array is cyclically checked whether there are any threads waiting to enter the critical section and if there is a thread waiting then, the lock is set free and the waiting thread is given access to the critical section.

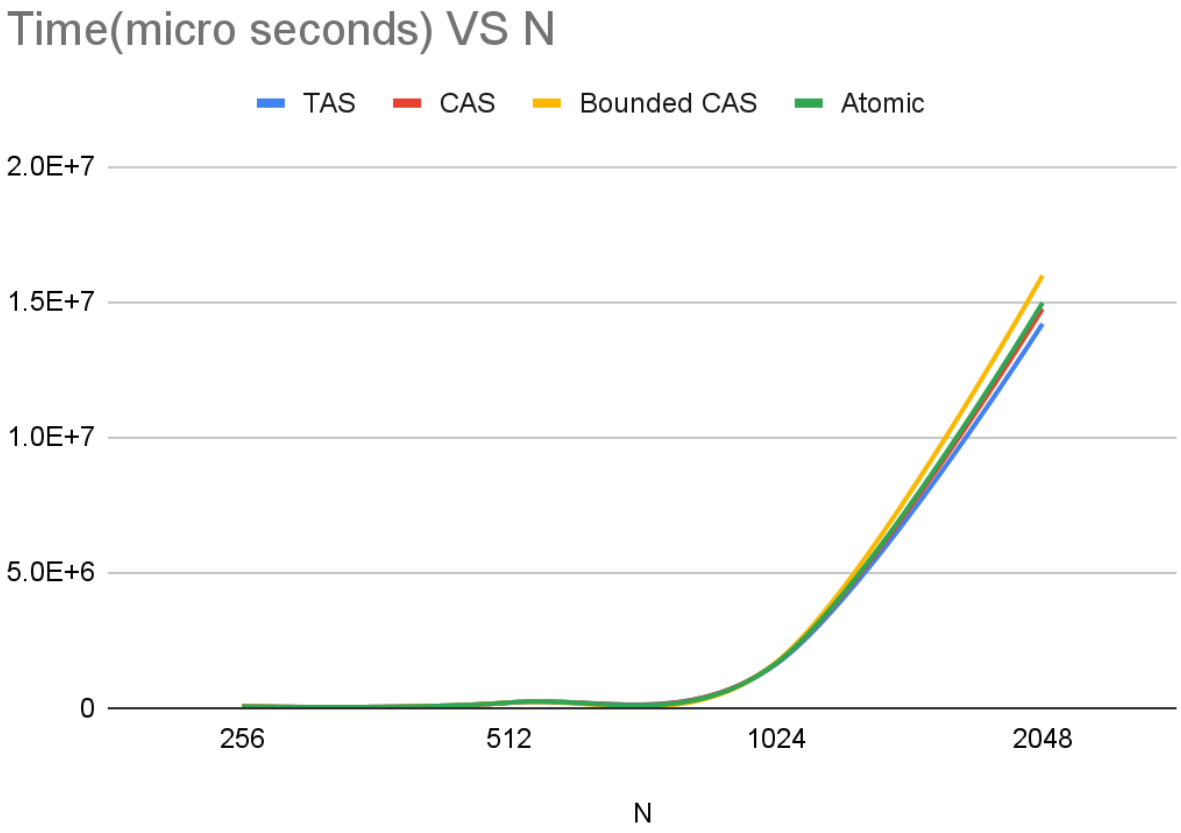
**Atomic Increment:**

➔ The counter is declared as an atomic variable ,so that it is incremented as a single instruction , it can be incremented as a whole or not at all.

➔ fetch_add () function from the atomic library is used to increment the counter variable by rowInc atomically, i.e, counter cannot be incremented by two threads at the same time, it is considered as a single instruction as we declared it as atomic variable.

- Now, after all the rows are assigned to K threads and after all of them calculate them , these rows are written into an output array and written into a file (out_TAS.txt,out_CAS.txt,...).

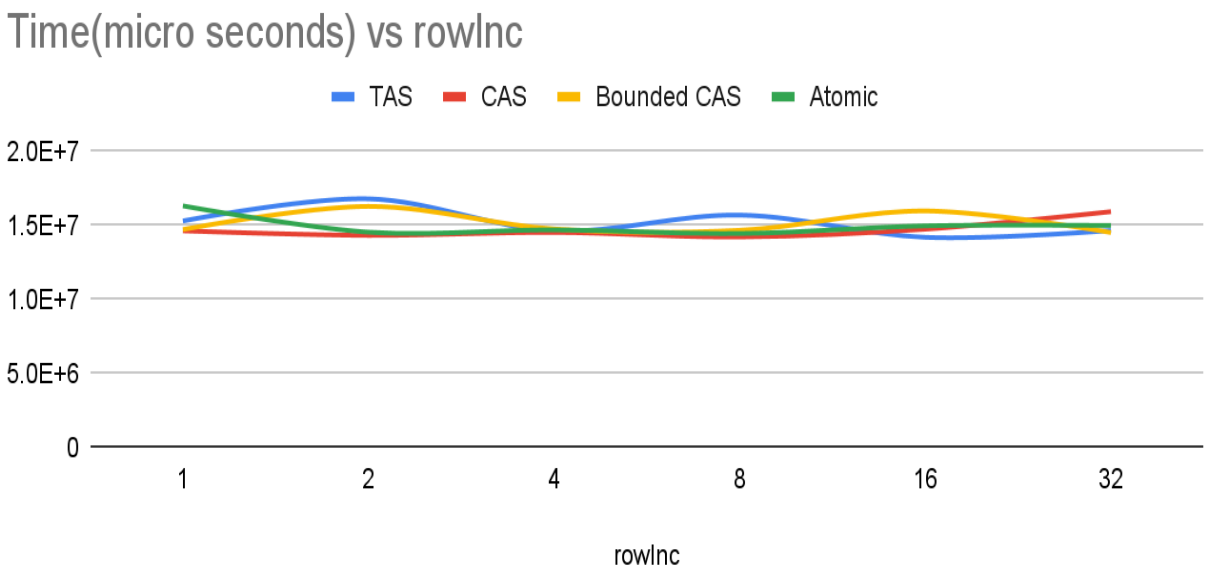
## Graph Plots and their Analysis:




| N | Time in TAS Algorithm | Time in CAS Algorithm | Time in Bounded CAS Algorithm | Time in Atomic Algorithm |
|---|---|---|---|---|
| 256 | 45036 | 50280 | 47284 | 46900 |
| 512 | 187920 | 188933 | 190549 | 189358 |
| 1024 | 1595738 | 1637328 | 1663816 | 1629210 |
| 2048 | 14186175 | 1473750 | 15968620 | 14959960 |

**Observations:**

As we are increasing the value of N ,i.e, the number of rows then the workload on each thread also increases as K is fixed so, it is obvious that the performance of all the algorithms is the same . As, N increased then the time taken by all the algorithms increased.

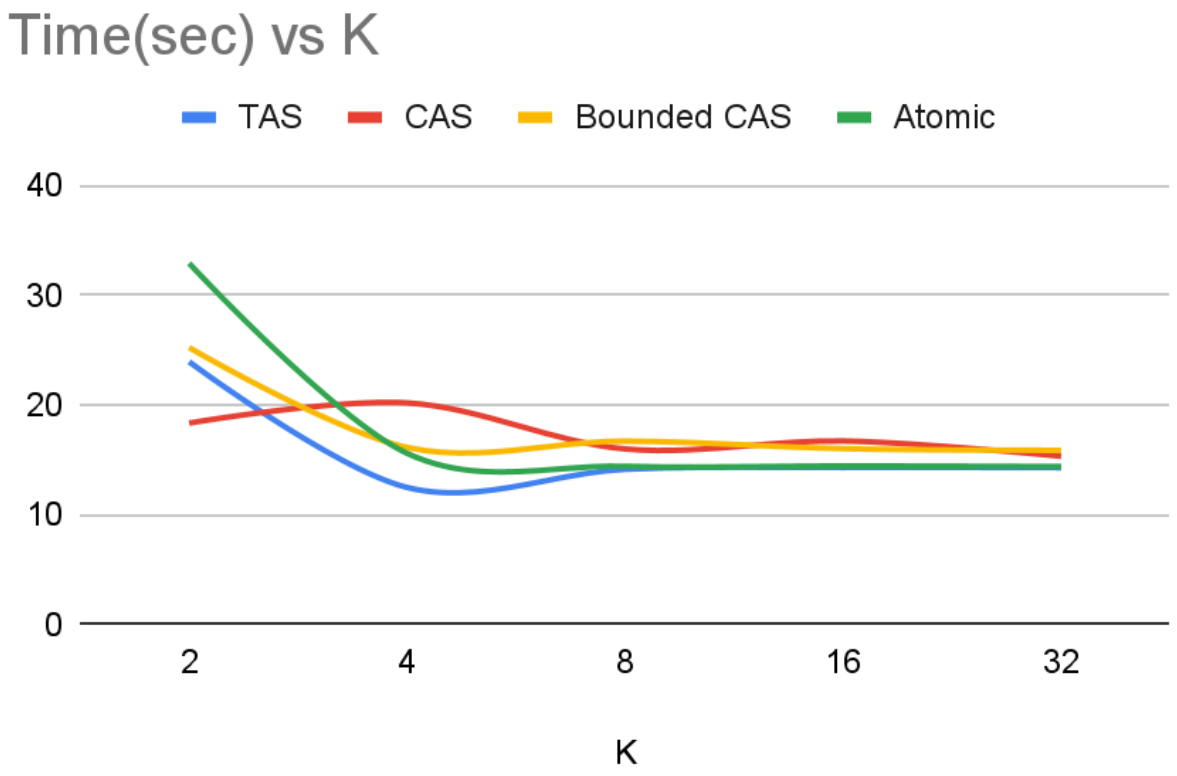



| rowInc | TAS | CAS | Bounded CAS | Atomic |
|---|---|---|---|---|
| 1 | 15289850 | 14625800 | 14711500 | 16313525 |
| 2 | 16789933 | 14318033 | 16277716 | 14533566 |
| 4 | 14625500 | 14519850 | 14748633 | 14688633 |
| 8 | 15689266 | 14210366 | 14661666 | 14442633 |
| 16 | 14186175 | 14737500 | 15968620 | 14959960 |
| 32 | 14648333 | 15920900 | 14510100 | 14985733 |

**Observations:**

As the value of rowInc is changed while both N and K being fixed , there isn't much of a change in the performance of the threads.As the value of rowInc is increasing the number of times the statement (counter+=rowInc) executes increases ,which doesn't make much difference as it is the only statement changing with rowInc increasing. In any of the algorithms as the workload on

each thread is almost the same, the time taken by each algorithm did not change.

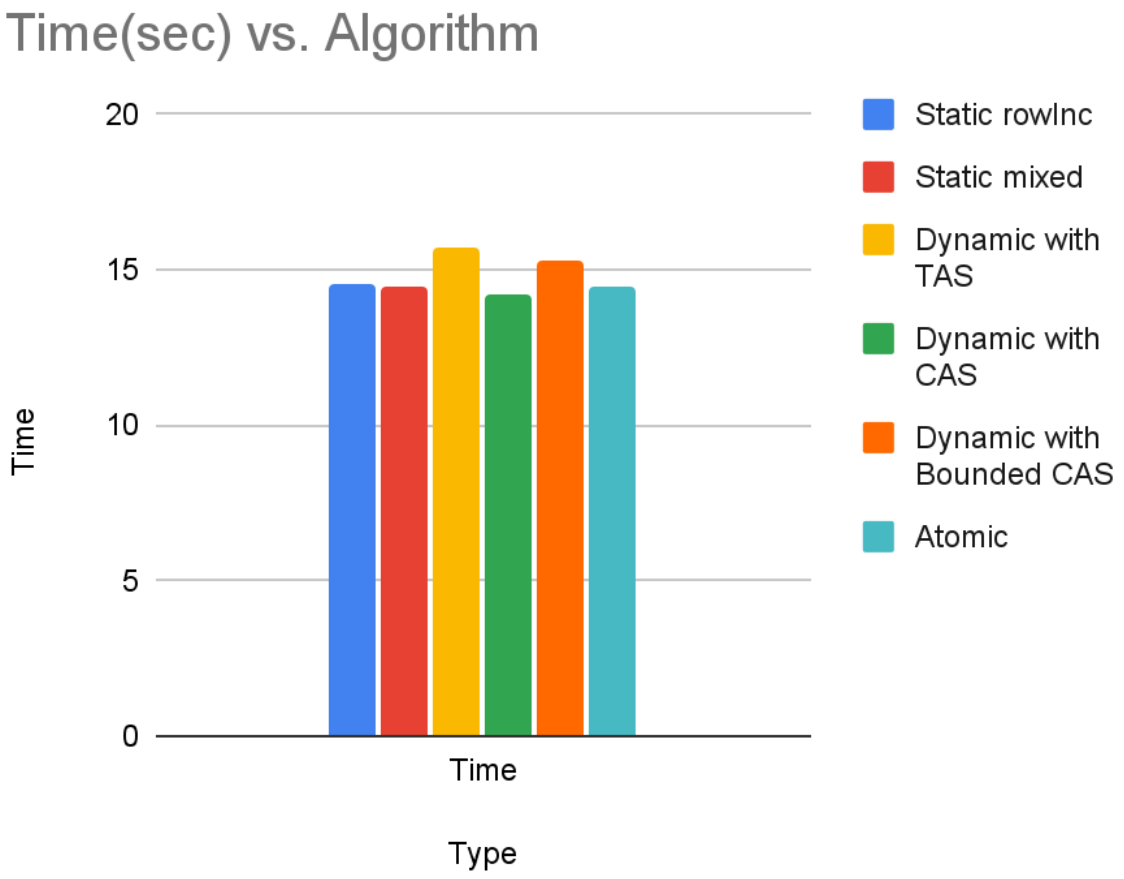



| K | TAS | CAS | Bounded CAS | Atomic |
|---|---|---|---|---|
| 2 | 23.9509 | 18.3629 | 25.2465 | 32.92615 |
| 4 | 12.49455 | 20.2057 | 16.1438 | 15.604 |
| 8 | 14.1529 | 16.00615 | 16.7201 | 14.42775 |
| 16 | 14.3191 | 16.7325 | 16.0306 | 14.45865 |
| 32 | 14.3018 | 15.339 | 15.885 | 14.403 |

**Observations:**

As we expect, when the value of N is fixed and the number of threads i.e, the value of K is increased, the workload on each thread decreases . So, in the graph above we can see that all the algorithms performance enhances by increasing the number of threads. In the graph above, Atomic algorithm seems to have a steeper curve and has better performance when compared to others, because there isn't any while loop in it to check continuously whether a lock is

held or not. Which implies in better performance compared to others where there is a while loop which continuously checks for the lock to give access to the waiting threads.




| Algorithm Type | Average Time |
|---|---|
| Static rowInc | 14.51798 |
| Static mixed | 14.4823 |
| Dynamic with TAS | 15.7408 |
| Dynamic with CAS | 14.24496 |
| Dynamic with Bounded CAS | 15.32284 |
| Atomic | 14.47836 |

**Observations:**

In the bar graph above, we can see that CAS and Atomic algorithms are giving better performance than others. CAS and atomic increment operations

allow for fine-grained synchronization without the need for locking entire sections of code . Atomic operations are often implemented directly in hardware or have optimized software implementations, making them faster and more efficient. These might be the reasons for better performance