Programming Assignment-3
Report

-Anirudh Saikrishnan(CS22BTECH11004)

Introduction:

This report presents the implementation and performance analysis of parallel matrix multiplication using a dynamic mechanism with different mutual exclusion algorithms in C++. The goal was to compute the square of a matrix A in parallel and measure the performance across various parameters such as matrix size (N), row increment (rowInc), number of threads (K), and mutual exclusion algorithms.

Implementation Details:

The program reads input parameters N, K, rowInc, and the matrix A from an input file named inp.txt. It then computes the square matrix using dynamic allocation of rows to threads with mutual exclusion mechanisms such as TAS, CAS, Bounded CAS, and atomic increment provided by the C++ atomic library following which it prints the square matrix to its respective output file along with the time taken for the computation.

Critical Section Analysis:

The main method creates K threads and sends them all to execute a function named criticalSection. The important thing to note is that the actual critical section is only a subset of the commands within this function.

The general idea for the dynamic allocation of threads is the following:

- There exists a global counter C and a global variable rowInc.
- The variable C keeps track of how many rows have already been assigned and rowInc specifies how many rows every thread handles at a time.
- Whenever a thread is free, it accesses the counter C. It picks up rowInc rows starting at C (Basically, the rows C to C+rowInc-1). Since this counter is global, i.e, shared by all threads and we have no idea which thread will access it at what point, the allocation of threads to rows is completely dynamic. Furthermore, since this C will need to be updated every time a thread accesses it, there are chances for race conditions to occur.
- Thus, the actual critical section of the code is the lines which allot rows to each thread along with the counter updation.

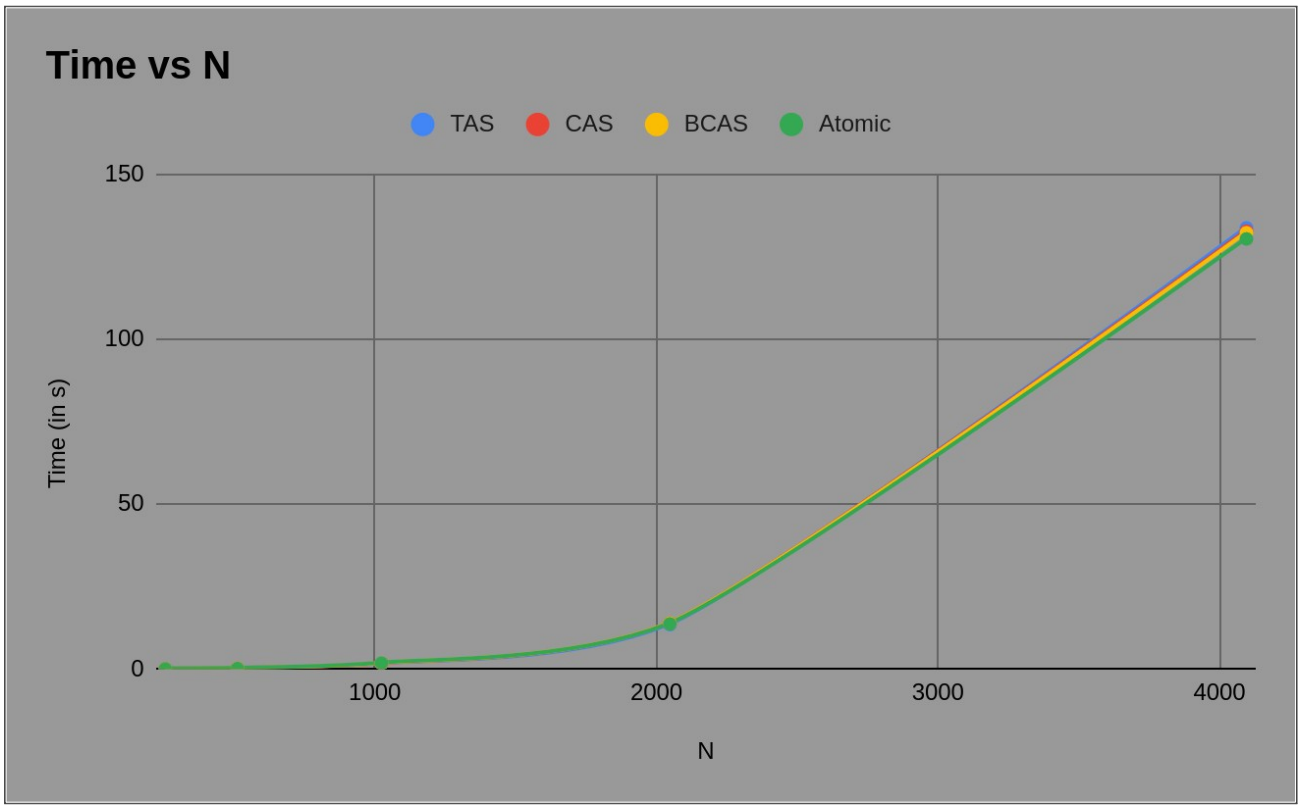
The threads all exit the critical section function only once C hits the value N. Thus, till this point, they keep repeating this function over and over. Basically, when a thread finishes the rows its assigned, it will repeat this function and pick up a new set of lines. The 4 mutual exclusion mechanisms are implemented in the standard fashion and don't affect the majority of the remaining code, which stays similar for all 4 mechanisms.

Data Analysis

1. Time vs. Size, N:

	TAS	CAS	BCAS	Atomic
256	26.2	26.8	28.2	27.6
512	224.2	231	232.6	229.2
1024	1691.2	1691.2	1775.8	1794
2048	13487.6	13906	13930.8	13750
4096	133773.4	132855.6	132327.2	130430

(in ms)

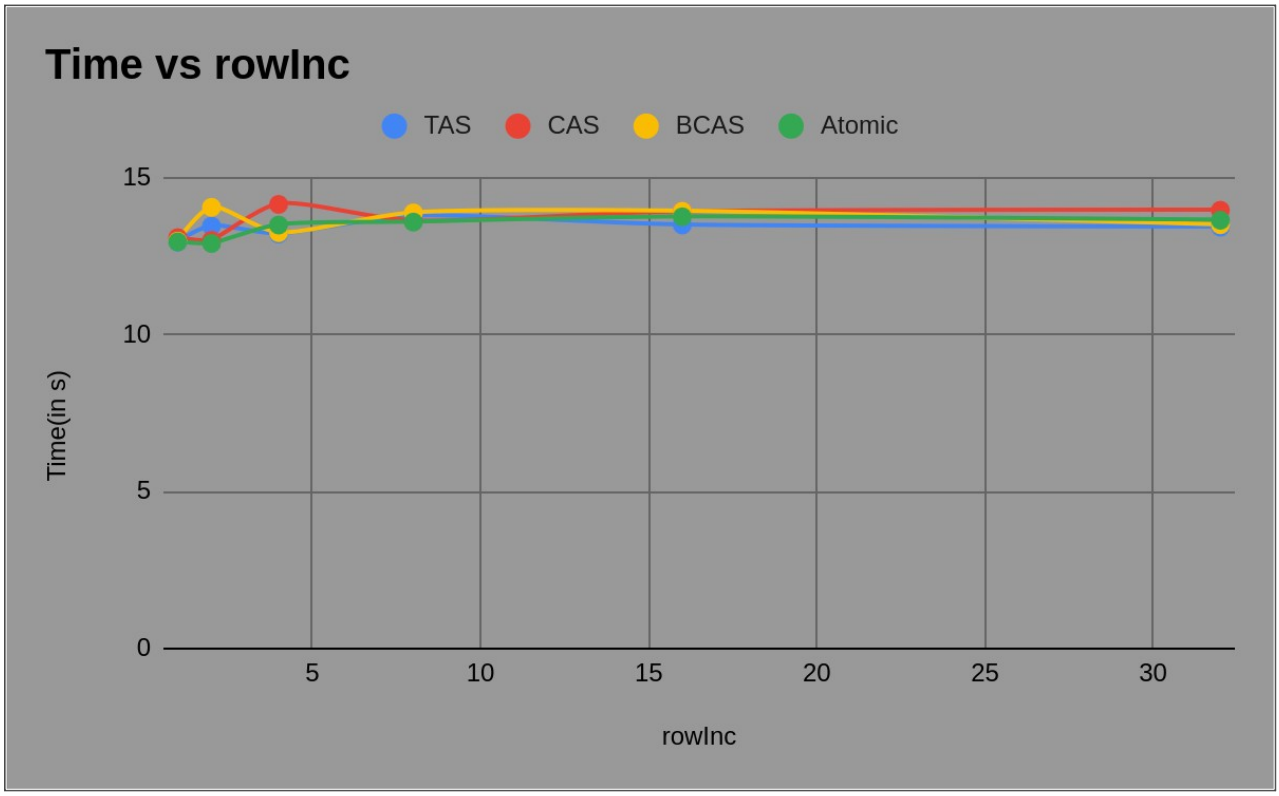


- As shown in the plot, the time taken to compute the square matrix increases with the size of the input matrix.
- All mutual exclusion methods exhibit similar trends, but Bounded CAS and atomic increment tend to perform slightly better as N increases since
  - Atomic operations minimize contention and synchronization overhead by allowing threads to access shared variables concurrently without explicit locking.
  - BCAS reduces contention by limiting the number of retries, thereby improving overall scalability and performance.
- An interesting observation is that the time taken grows non-linearly, in fact, using the data points, it is clear that it grows in the order of O(N^3) *{Increasing N by a factor of 2 increases time of execution approximately by a factor of 8}*, which stems from the fact that the matrix multiplication algorithm we implemented is an O(N^3) algorithm.

2. Time vs. rowInc, row Increment:

	TAS	CAS	BCAS	Atomic
1	12943	13090	12976.8	12953.4
2	13456.4	13005.2	14052.4	12901.6
4	13206.8	14147.8	13247.8	13489
8	13771	13614.6	13873.6	13583.8
16	13487.6	13906	13930.8	13750
32	13423.8	13958.6	13494.2	13644.2

(in ms)

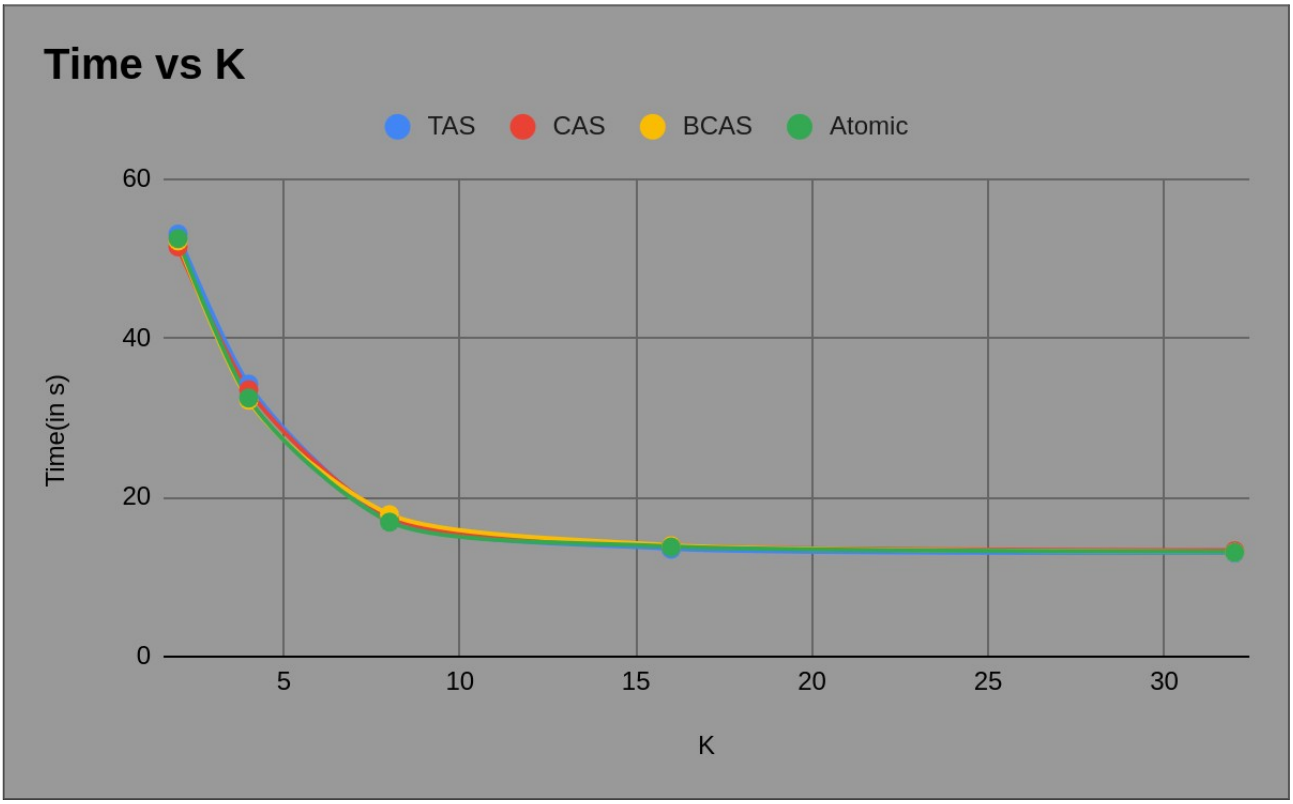


- The plot illustrates that as row increment increases, the time taken to compute the square matrix varies wildly initially, then stabilizes.
- However, the execution time stays within a specific range irrespective of the value of rowInc. This is because in a reasonably random matrix, probabilistically, every thread will still be assigned a very similar number of rows irrespective of the value of rowInc. This is why the values also fall into a very similar range.

3. Time vs. Number of threads, K:

	TAS	CAS	BCAS	Atomic
2	53081.6	51466.6	52206.8	52499.8
4	34235.8	33527.2	32231.2	32479.8
8	17291.4	17520.4	17832.2	16902.4
16	13487.6	13906	13930.8	13750
32	12988.4	13290	13096.8	13088

(in ms)

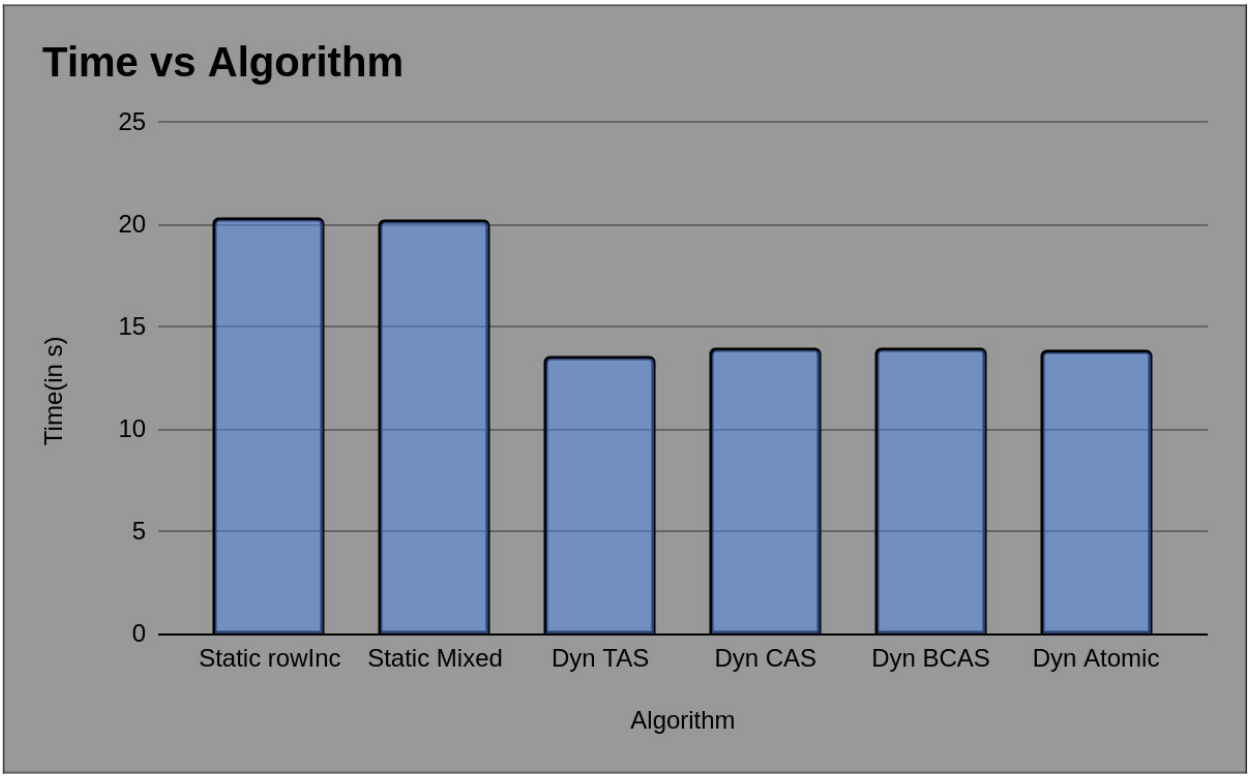


- The plot demonstrates that the time taken decreases as the number of threads increases initially, indicating better parallelization.
- However, beyond a certain point, further increasing the number of threads leads to diminishing returns, as seen for K=16 to K=32.
- TAS, CAS, and Bounded CAS exhibit similar trends in performance with varying thread counts, while atomic increment tends to perform slightly better with higher thread counts, again, due to its lock-free operations.

4. Time vs. Algorithms:

Static rowInc	20253.8
Static Mixed	20178.4
Dyn TAS	13487.6
Dyn CAS	13906
Dyn BCAS	13930.8
Dyn Atomic	13750

(in ms)



- The bar graph compares the performance of different mutual exclusion algorithms and static approaches.
- It is evident that dynamic mechanisms with CAS, Bounded CAS, and atomic increment outperform static approaches significantly.
- This performance gulf is specifically emphasized since N is large in this case and dynamic algorithms are significantly better than static ones for dense matrices of larger size since chances of threads sitting idle are minimal.
- Among the dynamic mechanisms, theoretically, atomic increment generally performs the best, followed by Bounded CAS, CAS, and TAS. Practically, the difference in execution times is minimal.