Programming Assignment-3

Kandi Prudhvi
CS22BTECH11031

March 4, 2024

1 Introduction

This report describes the aims to perform parallel matrix multiplication through a Dynamic mechanism in C++.This code is written in C++ program.

1.1 FILE INPUT

The program reads integers N,K,rowInc and matrix A from an input file named inp.txt.N is the number of rows in matrix A. K is the number of threads. The value rowInc can be seen similarly to the chunk size of assignment 1, where the chunk size was fixed statically.

1.2 Low Level Design

The program reads the N,K,rowInc values from input file and store them in variables n,k,rowinc respectively. Then it reads the matrix A from input file and store it in 2D vector named v. There are four algorithms and each algorithm is written in different codes. So, main function will be same for for all four codes. Creates a vector of threads (threads) to perform matrix multiplication using the tas or cas or bcas or atom function. Each thread is assigned a chunk of rows to process concurrently.The main function waits for all threads to finish their computations by using join on each thread.

1: **TAS**: The tas function is designed to facilitate multithreaded matrix multiplication using the Test-and-Set (TAS) technique for thread synchronization. It takes as input a vector of vectors v representing an input matrix, the matrix size n, a reference to the result matrix vec, and an integer rowinc. The function utilizes atomic flags for thread synchronization. The primary functionality of the tas function involves assigning a chunk of rows to each thread and performing matrix multiplication for the assigned rows. The function employs a while loop to iterate until all rows are processed. Inside the loop, it uses an atomic flag (lock) to ensure exclusive access to the shared count variable, which keeps track of the next row to be processed. Once a thread acquires the lock, it determines the start and end indices for the rows it is responsible for and updates the count variable accordingly. For each assigned row in the chunk, the function performs matrix multiplication with the corresponding row of the input matrix (v). The result is then stored in the output matrix (vec). The use of thread synchronization and row assignments allows multiple threads to concurrently compute different portions of the matrix multiplication task, improving parallelism and potentially reducing computation time. Writes the resulting matrix (vec) and the elapsed time to an output file (out1.txt).

2: **CAS**: The cas function is designed to facilitate multithreaded matrix multiplication using the Compare-And-Swap (CAS) technique for thread synchronization. The function takes as input an input matrix represented by the vector of vectors v, along with the matrix size n, a reference to the result matrix vec, and an integer rowinc indicating the number of rows each thread is responsible for processing. The core of the function is a while loop that iterates until all rows have been processed. Inside this loop, a CAS operation is employed to atomically check and set the value of the lock variable, ensuring exclusive access to the critical section. The function effectively waits until it successfully acquires the lock before proceeding. Once the lock is obtained, the function determines the start and end indices for the rows it is responsible for and updates the count variable accordingly. Subsequently, it performs matrix multiplication for the assigned rows, iterating through the input matrix v and storing the results in the output matrix vec. After completing its computations for the assigned rows, the function releases the lock, setting lock back to 0. This allows other threads to enter their critical sections and proceed

with their designated tasks. The loop continues until all rows have been processed. Writes the resulting matrix (vec) and the elapsed time to an output file (out2.txt).

3: **Bounded CAS**: The bcas function is designed to facilitate multithreaded matrix multiplication using the Bounded CAS technique for thread synchronization.Operating within a loop until the global counter count surpasses the total number of rows (n), each thread sets its waiting flag, indicating readiness. The function employs a custom Bounded Compare-and-Swap (CAS) mechanism to synchronize access to shared variables, such as the counter count and the lock lock. This ensures that only one thread at a time can enter the critical section. After acquiring the lock, the thread determines the range of rows it's responsible for based on the global counter and the specified rowinc. Matrix multiplication is then performed on the assigned rows. Subsequently, the function releases the lock and looks for the next waiting thread, signaling it to proceed. If no waiting threads exist, it releases the lock to enable other threads to enter the critical section. Overall, the bcas function orchestrates the concurrent execution of matrix multiplication tasks, leveraging Bounded CAS for synchronization and optimizing the workload distribution among multiple threads. Writes the resulting matrix (vec) and the elapsed time to an output file (out3.txt).

4: **Atomic**: The atom function in the provided code is designed for multithreaded matrix multiplication using atomic operations, specifically the $atomic < int >$ type and the fetch_add method. This function is intended to be executed concurrently by multiple threads. The primary objective of the atom function is to perform matrix multiplication for a designated chunk of rows. The function uses an atomic integer (count) to efficiently manage the assignment of rows to different threads, ensuring synchronized and thread-safe access to the shared resource. Within the function, a while loop continues execution until all rows have been processed. The fetch_add method is employed on the atomic counter (count), providing an atomic increment of the counter by the specified rowinc value while simultaneously returning the original value. This atomic operation ensures that each thread receives a unique starting row index for processing. Subsequently, the function computes the start and end indices for the assigned rows, taking into account the chunk size (rowinc) and the matrix size (n). For each assigned row, the function performs matrix multiplication with the corresponding row of the input matrix (v). The results are then stored in the output matrix (vec). Writes the resulting matrix (vec) and the elapsed time to an output file (out4.txt).

1.3 Experiment 1: Time vs. Size, N:

1. Time vs. Size, N: The y-axis will show the time taken to compute the square matrix in this graph. The x-axis will be the values of N (size on input matrix) varying from 256 to 4096 (size of the matrix will vary as 256*256, 512*512, 1024*1024, ...) in the power of 2. rowInc fixed at 16 and K at 16 for all these experiments.

Size(N)	Time taken by TAS(in sec)	Time taken by CAS(in sec)	Time taken by Bounded CAS(in sec)	Time taken by atomic(in sec)
256	0.02102	0.01912	0.02151	0.02023
512	0.15452	0.14393	0.15302	0.14251
1024	1.20185	1.11815	1.15437	1.09001
2048	9.85128	9.55410	9.55495	9.22824
4096	131.611	140.765	135.602	137.170

Table 1: Time taken for 4 algorithms

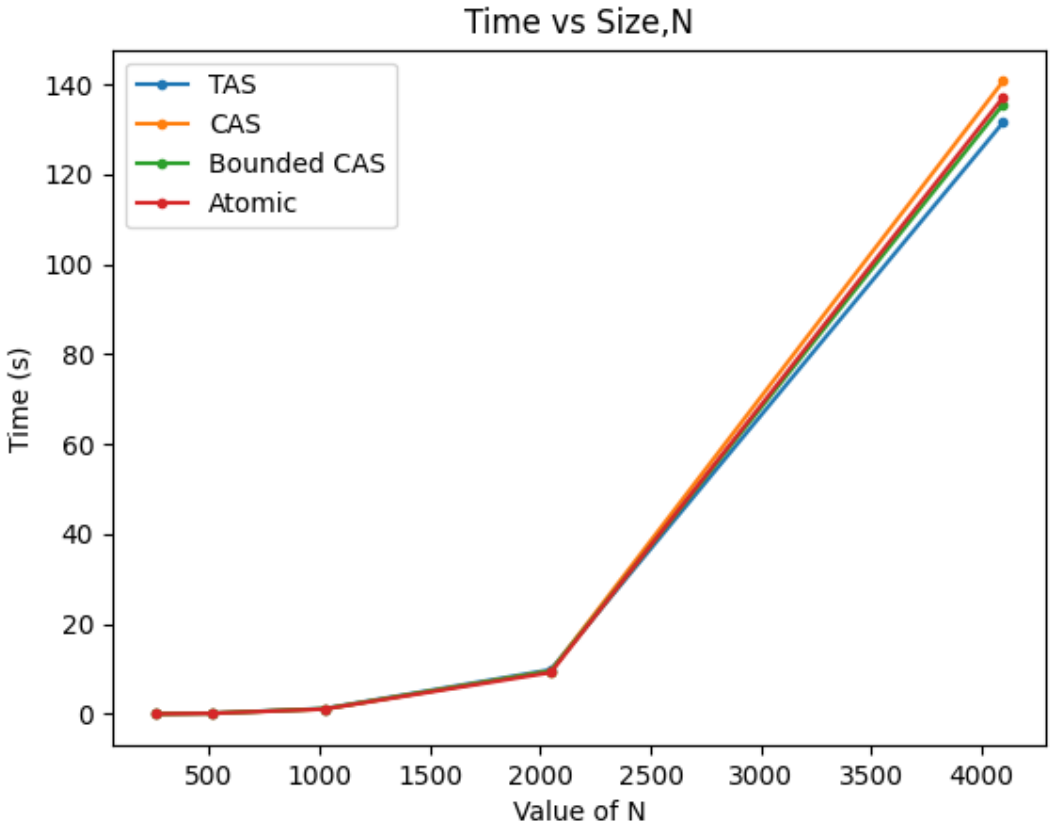

Figure 1: Time vs. Size, N

1.4 Experiment 2: Time vs. rowInc, row Increment:

The y-axis will show the time to compute the square matrix. The x-axis will be the rowInc varying from 1 to 32 (in powers of 2, i.e., 1,2,4,8,16,32). Fix N at 2048 and K at 16 for all these experiments.

rowInc	Time taken by TAS(in sec)	Time taken by CAS(in sec)	Time taken by Bounded CAS(in sec)	Time taken by atomic(in sec)
1	19.3773	19.6905	18.6440	17.9603
2	20.0370	20.9408	19.4490	18.2911
4	18.9162	18.9125	18.7731	17.9048
8	18.2324	17.9457	18.6695	17.7866
16	18.0408	17.5543	18.1197	17.2560
32	17.7843	17.3468	17.8751	17.0843

Table 2: Time taken for 4 algorithms

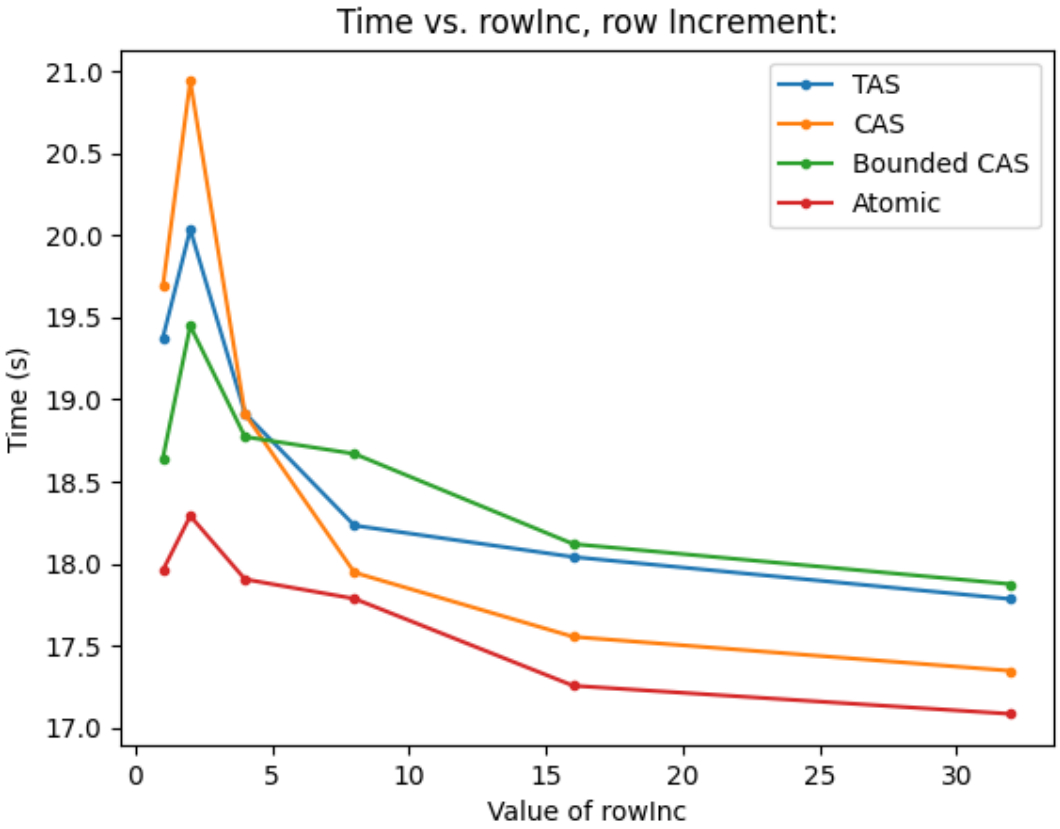

Figure 2: Time vs rowInc

1.5 Experiment 3: Time vs. Number of threads, K:

The y-axis will show the time taken to do the matrix squaring, and the x-axis will be the values of K, the number of threads varying from 2 to 32 (in powers of 2, i.e., 2,4,8,16,32). N fixed at 2048 and rowInc at 16 for all these experiments.

Number of threads(K)	Time taken by TAS(in sec)	Time taken by CAS(in sec)	Time taken by Bounded CAS(in sec)	Time taken by atomic(in sec)
2	44.9355	45.9363	46.1035	44.9673
4	34.4617	33.3901	33.9387	33.6327
8	22.3935	21.4037	23.3371	21.5497
16	21.97	19.2043	20.9399	19.9923
32	15.7361	15.4324	16.1928	15.6823

Table 3: Time taken for 4 algorithms

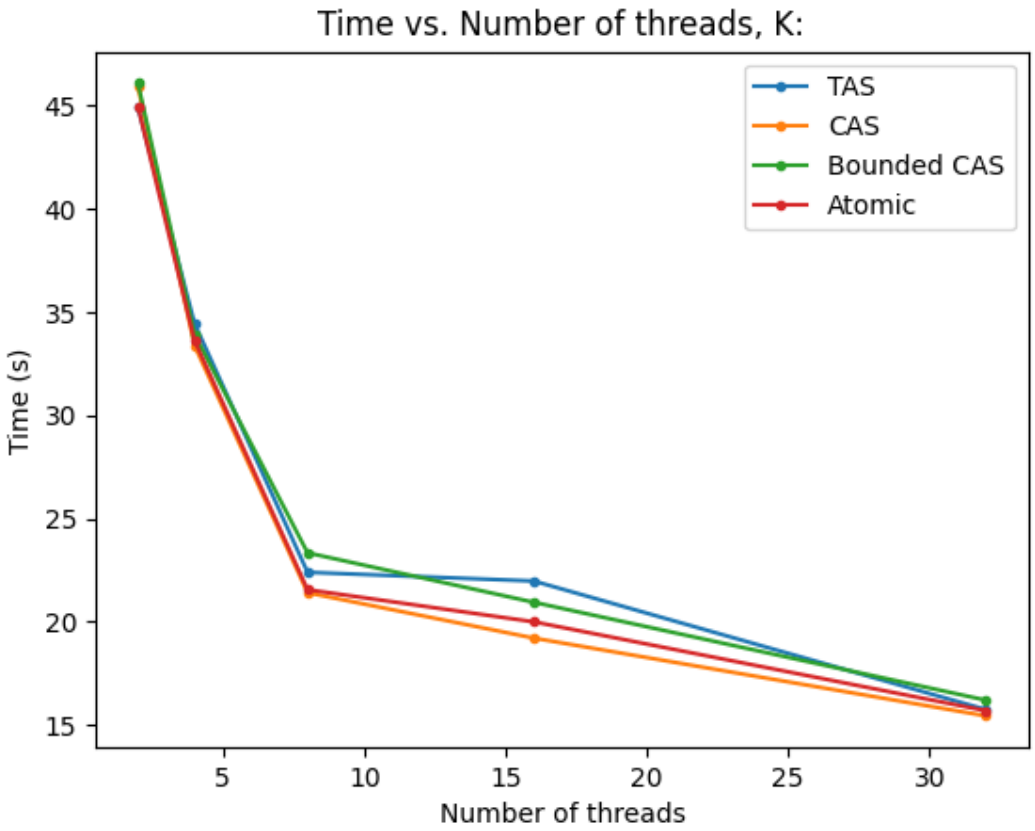

Figure 3: Time vs Number of threads

1.6 Experiment 4: Time vs. Algorithms:

The y-axis will show the time taken to do the matrix squaring, and the x-axis will be different algorithms.

Static rowInc - 23.4187 s
Static mixed - 21.9092 s
Dynamic with TAS - 18.1886 s
Dynamic with CAS - 17.9571 s
Dynamic with Bounded CAS - 18.8868 s
Dynamic with Atomic - 18.0339 s

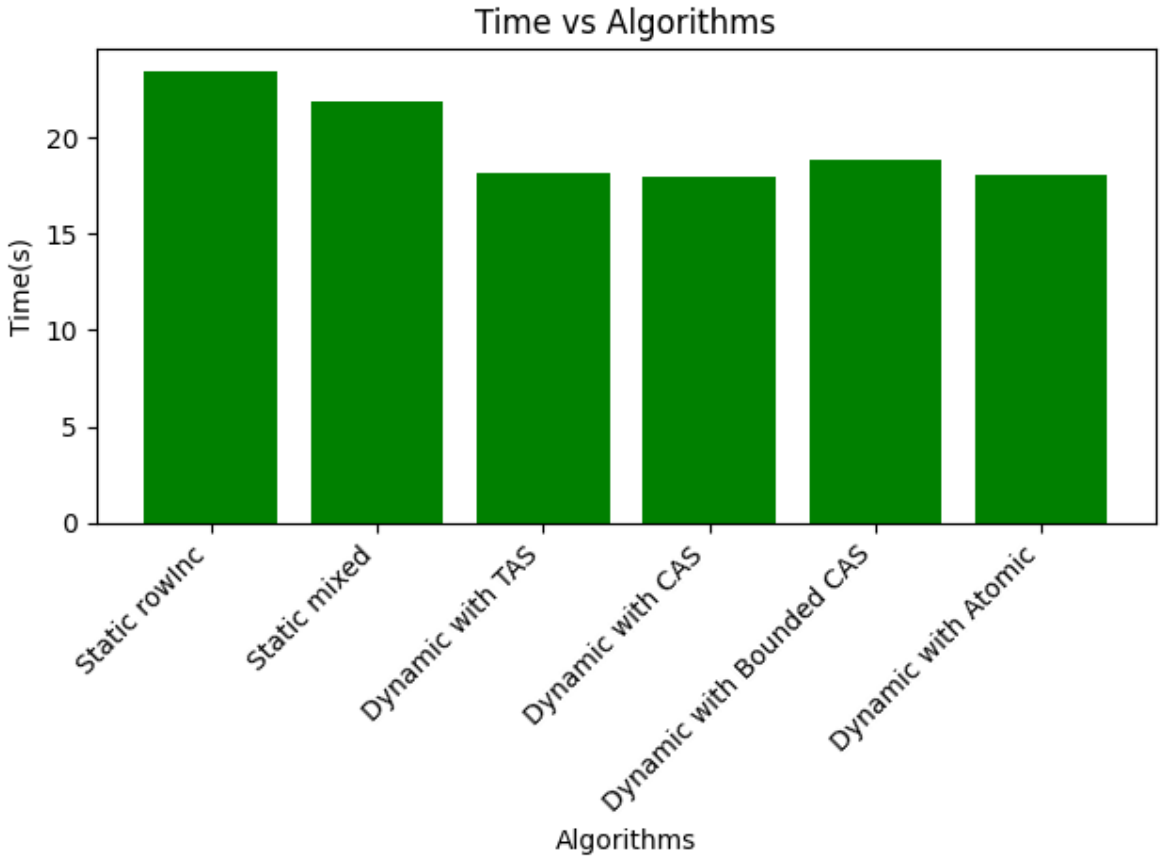

Figure 4: Time vs Algorithms

1.7 Analysis of Graphs :

Experiment 1: Time taken for all algorithms increases as the value of N increases. All algorithms take almost same time.

Experiment 2: Time taken is random and slightly decreases at the last as rowInc increases. Atomic took less time at the beginning but all algorithms take almost same time at the end.

Experiment 3: Time taken for all algorithms decreases as the value of K increases. Bounded CAS has taken more time than remaining algorithms. This phenomenon may be attributed to the reduced occurrence of thread starvation issues for lower thread counts. In situations where there are fewer threads, the challenges associated with unbounded waiting are less pronounced. Consequently, the incorporation of a mechanism for bounded waiting might introduce some overhead in the implementation.

Experiment 4: The superior performance of dynamic algorithms compared to static algorithms can be attributed to the fact that static algorithms may not achieve an equal distribution of work for certain matrix inputs. In such cases, some threads may complete their tasks early, leaving others still processing. This asymmetry in workload distribution leads to an inefficient utilization of resources. On the contrary, dynamic algorithms allocate work dynamically, allowing threads that finish early to receive new assignments. This adaptability ensures a more efficient utilization of computing resources, contributing to the observed improvement in performance.