CS3510: Assignment-3

Gunnam Sri Satya Koushik
CS22BTECH11026

March 4, 2024

1 Introduction

This report presents an extension of a previous programming assignment aimed at implementing parallel matrix multiplication using dynamic mechanisms in C++. The objective is to compute the square of a given square matrix A in parallel, with a focus on dynamic allocation of rows to threads. Synchronization issues arising from dynamic allocation are addressed through the implementation of mutual exclusion algorithms such as TAS, CAS, Bounded CAS, and atomic increment. The report outlines the methodology, implementation details, and performance measurement approach, aiming to demonstrate the efficiency and scalability of the proposed parallelization approach.

2 Program design

- **File Operations:**
  - The function starts by opening the input file `"input.txt"` using an `ifstream` object named `inputFile`.
  - It checks if the file is opened successfully using the `is_open()` function of the `ifstream` class.
  - If the file opening fails, it prints an error message to the standard error stream `cerr` and returns 1 to indicate an error.
  - The function then reads the dimensions `(n, k, rowInc)` from the input file using the extraction operator `>>`.
  - It initializes a 2D vector named `matrix` to store the matrix read from the input file.
  - A nested loop reads each element of the matrix from the input file into the `matrix` vector.
  - After reading the matrix, the input file is closed using the `close()` function.
- **Concurrency Operations:**
  - The function initializes a variable `C` to 0, which is presumably used elsewhere in the code.
  - It starts a timer to measure the execution time using the `chrono` library.
  - A result matrix `cresultMatrix` is initialized with zeros using a nested vector initialization.
  - It creates `k` threads for concurrent computation using the `pthread_create()` function.
  - Each thread is passed an argument of type `ComputeArgs`, containing pointers to the input matrix, result matrix, and a chunk index.
  - After creating all threads, the function waits for their completion using `pthread_join()` to ensure synchronization.
  - The timer is stopped, and the execution duration is calculated using `chrono` functions.
- **Output:**
  - The function opens an output file for writing using an `ofstream` object named `coutputFile`.
  - It checks if the output file is opened successfully similar to the input file.
  - The function writes the execution time to the output file followed by the result matrix.
  - Each element of the result matrix is written to the file followed by a space, and each row is terminated by a newline character.
  - After writing to the file, the output file is closed using the `close()` function.

3 Detailed Implementation Descriptions

- **Test-and-Set (TAS) Approach:**
  - TAS employs a global atomic flag to regulate access to a critical section in a parallel environment.
  - At the core of this approach is a shared boolean flag, typically initialized to indicate an unlocked state.
  - Threads competing for access to the critical section continuously attempt to set this flag using an atomic test-and-set operation.
  - If the flag is already set (indicating that another thread holds the lock), the current thread spins in a loop, continuously checking the status of the flag.
  - Upon detecting that the flag has been cleared (indicating the lock is available), the thread atomically sets the flag, signifying that it has acquired the lock and can proceed to execute the critical section.
  - After completing its work in the critical section, the thread clears the flag, releasing the lock and allowing other threads to acquire it.
- **Compare-and-Swap (CAS) Approach:**
  - CAS is a synchronization technique that enables atomic updates to a memory location based on its current value.
  - In the context of mutual exclusion, CAS is used to implement a lock, typically represented by a shared variable.
  - Threads wishing to enter a critical section attempt to atomically update the lock variable from an unlocked state to a locked state using a compare-and-swap operation.
  - If the current value of the lock matches the expected unlocked state, the CAS operation succeeds, indicating that the thread has acquired the lock and can proceed to execute the critical section.
  - If the CAS operation fails (indicating that another thread holds the lock), the thread retries the operation until it successfully updates the lock and gains access to the critical section.
- **Bounded Compare-and-Swap (Bounded CAS) Approach:**
  - Bounded CAS enhances the traditional CAS approach by introducing a mechanism to limit the number of retries during lock acquisition attempts.
  - Threads competing for the lock repeatedly perform CAS operations to transition the lock variable from an unlocked state to a locked state.
  - However, unlike standard CAS, Bounded CAS imposes a bound on the number of retry attempts.
  - If a thread exceeds the specified bound without successfully acquiring the lock, it terminates its attempt to prevent excessive spinning and potential performance degradation.
  - This bounded waiting mechanism aims to strike a balance between ensuring timely lock acquisition and avoiding excessive contention and resource wastage.
- **Atomic Increment Approach:**
  - The atomic increment approach leverages atomic operations provided by the C++ atomic library to facilitate thread-safe updates to a shared counter variable.
  - In this approach, a shared counter is initialized and incremented atomically by each thread to claim its portion of work or indicate progress.
  - Atomic operations ensure that increments are performed indivisibly, without the possibility of interference from concurrent updates by other threads.
  - By relying on atomic primitives provided by the underlying software, this approach eliminates the need for explicit locks or flags to coordinate access to the shared counter, reducing synchronization overhead and potential contention.

4 Output Analysis

4.1 Experiment 1: Time vs. Size N

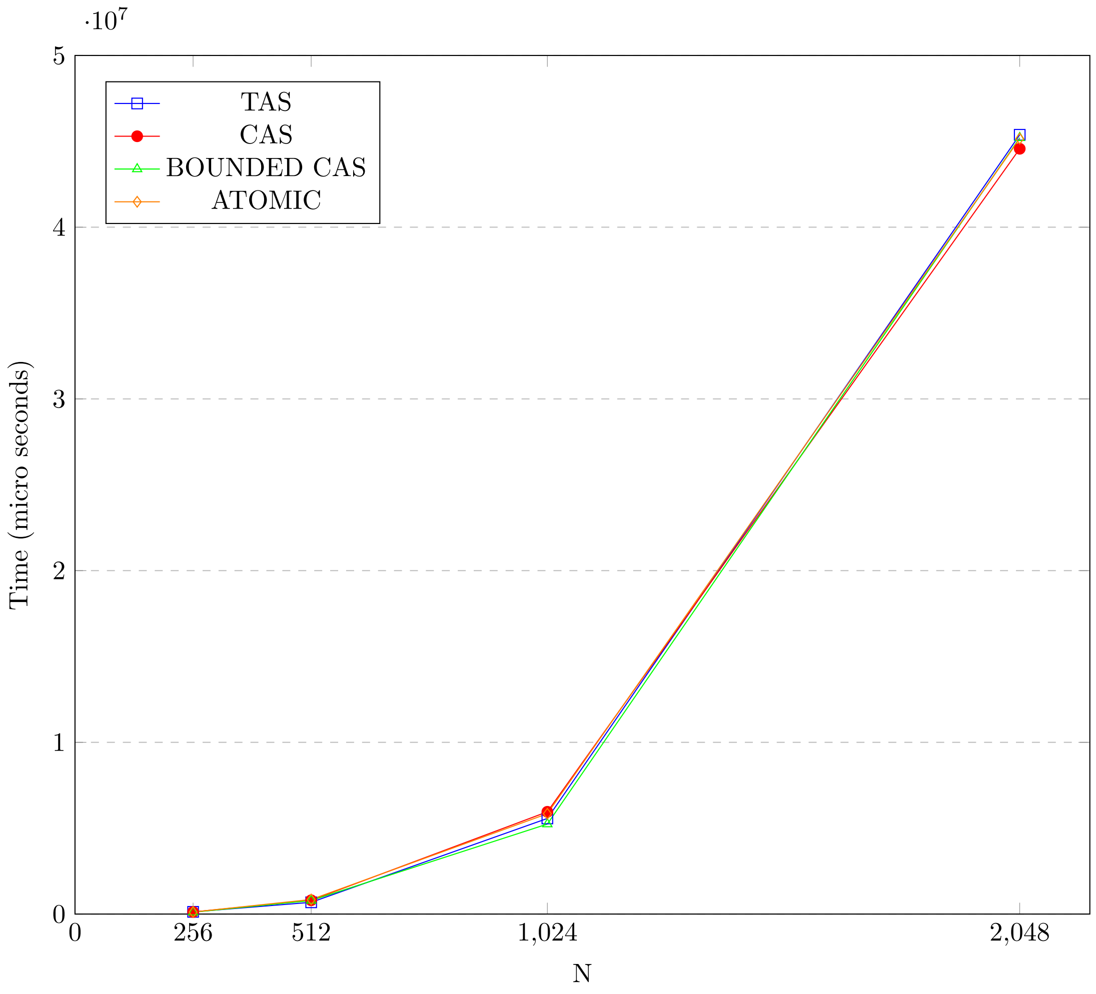

Figure 1: Graph of Time (ms) vs. n for Different Algorithms

n	TAS	CAS	BOUNDEDCAS	ATOMIC
256	128075	120949	104832	117943
512	683640	803294	788576	847531
1024	5567955	5954456	5239360	5843800
2048	45381290	44566963	45204248	45155744

Table 1: Time(micro seconds) vs n

Observation: As n increases, the computational workload also increases, resulting in longer execution times for all synchronization approaches.

4.2 Experiment 2: Time vs. rowInc

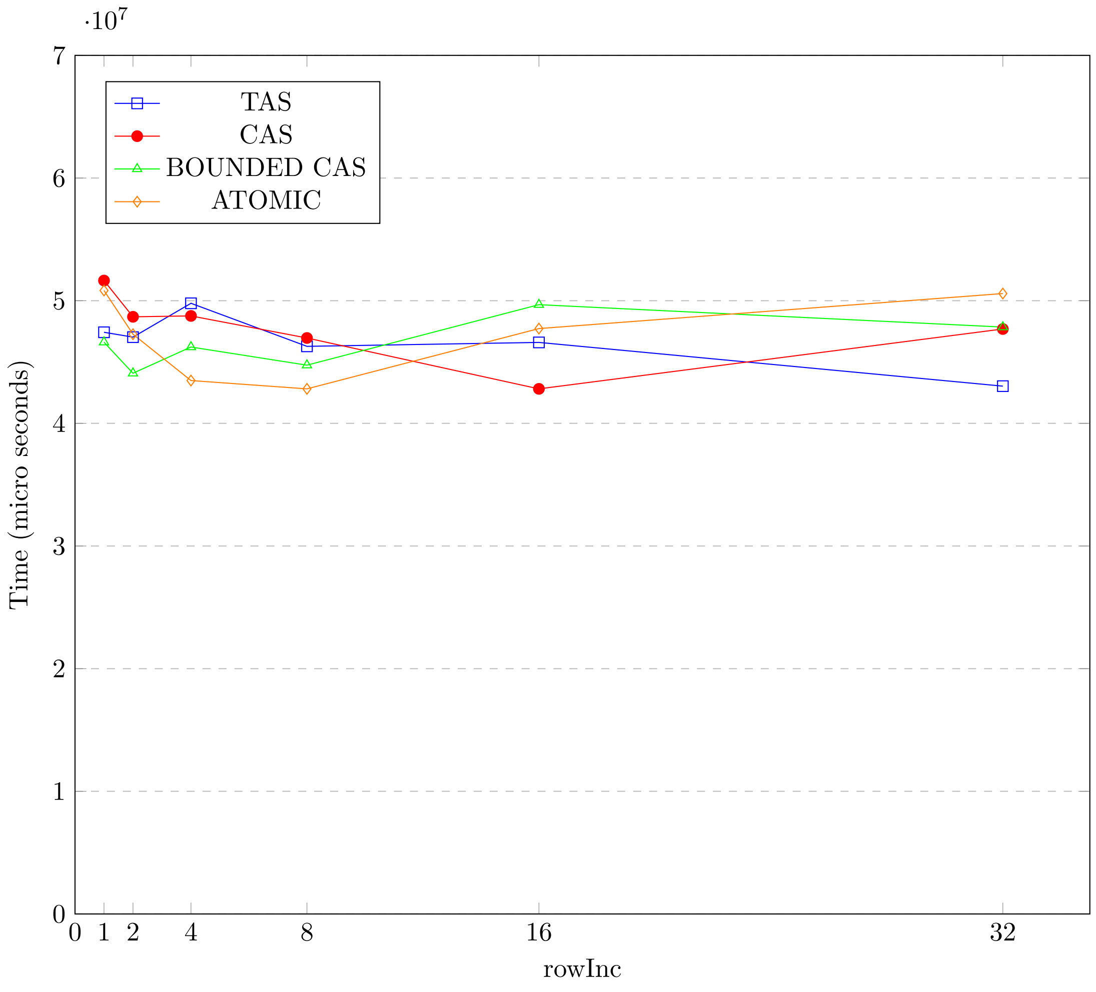

Figure 2: Graph of Time (ms) vs. Threads for Different Algorithms

rowInc	TAS	CAS	BOUNDEDCAS	ATOMIC
1	47425325	51641409	46618305	50830925
2	47029958	48686777	44084342	47266743
4	49787922	48758425	46224414	43489907
8	46277871	46958902	44743195	42811866
16	46597809	42811863	49675363	47726701
32	43033379	47692521	47849440	50582652

Table 2: Time(micro seconds) vs rowInc

Observation:

Time doesn't vary much for change in rowInc as load is same and distribution is almost uniform,

4.3 Experiment 3: Time vs. Number of threads

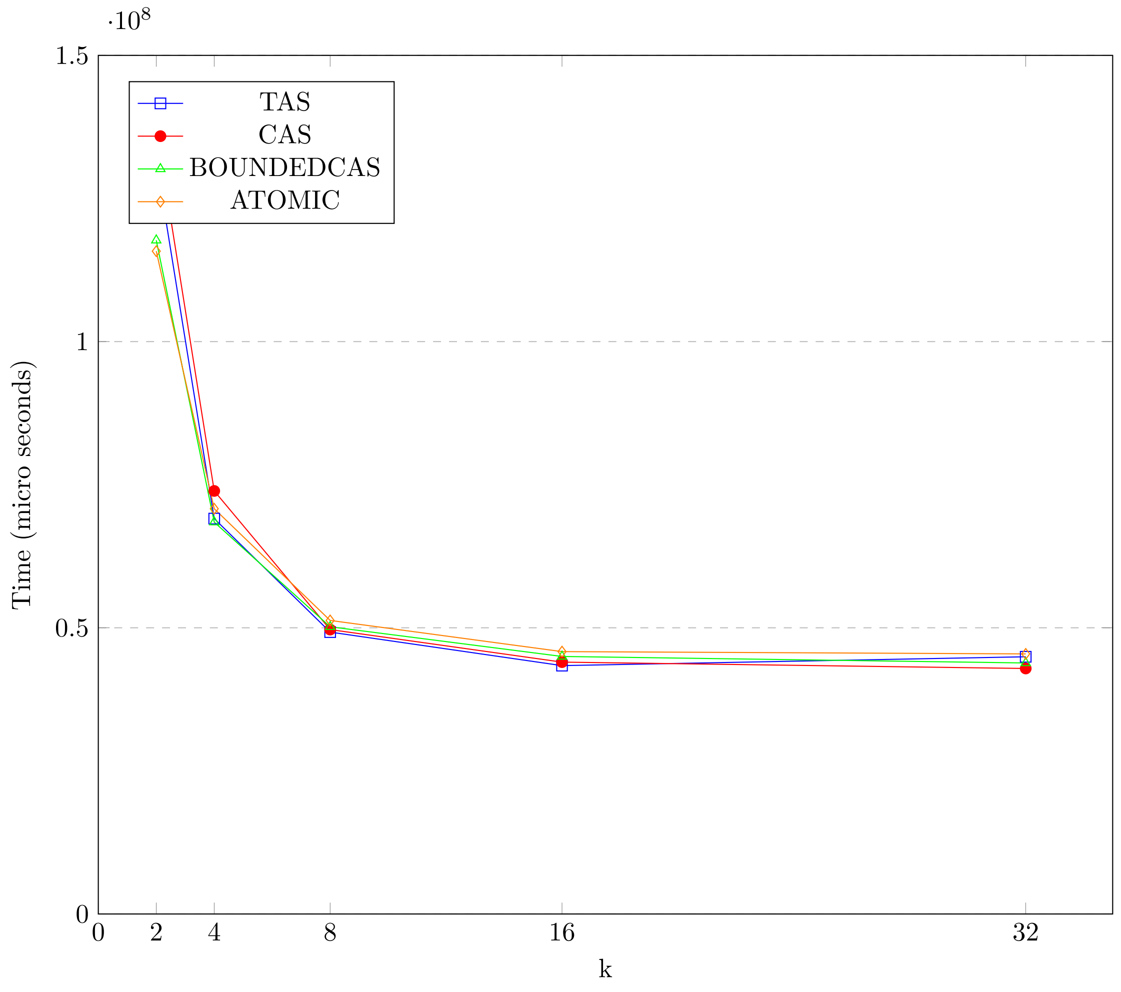

Figure 3: Graph of Time (ms) vs. k for Different Algorithms

K	TAS	CAS	BOUNDEDCAS	ATOMIC
2	131557374	137267621	117703984	115794050
4	69072696	73906748	68459800	70796596
8	49245741	49687056	50174217	51282438
16	43388432	44000090	44990357	45837294
32	44933654	42888792	43852178	45436775

Table 3: Time(micro seconds) vs K

Observation: As number of threads the time decreases but after 8 the decrement is very little , this might be due to no more extra cores availability.

4.4 Experiment 4: Time vs. Algorithms

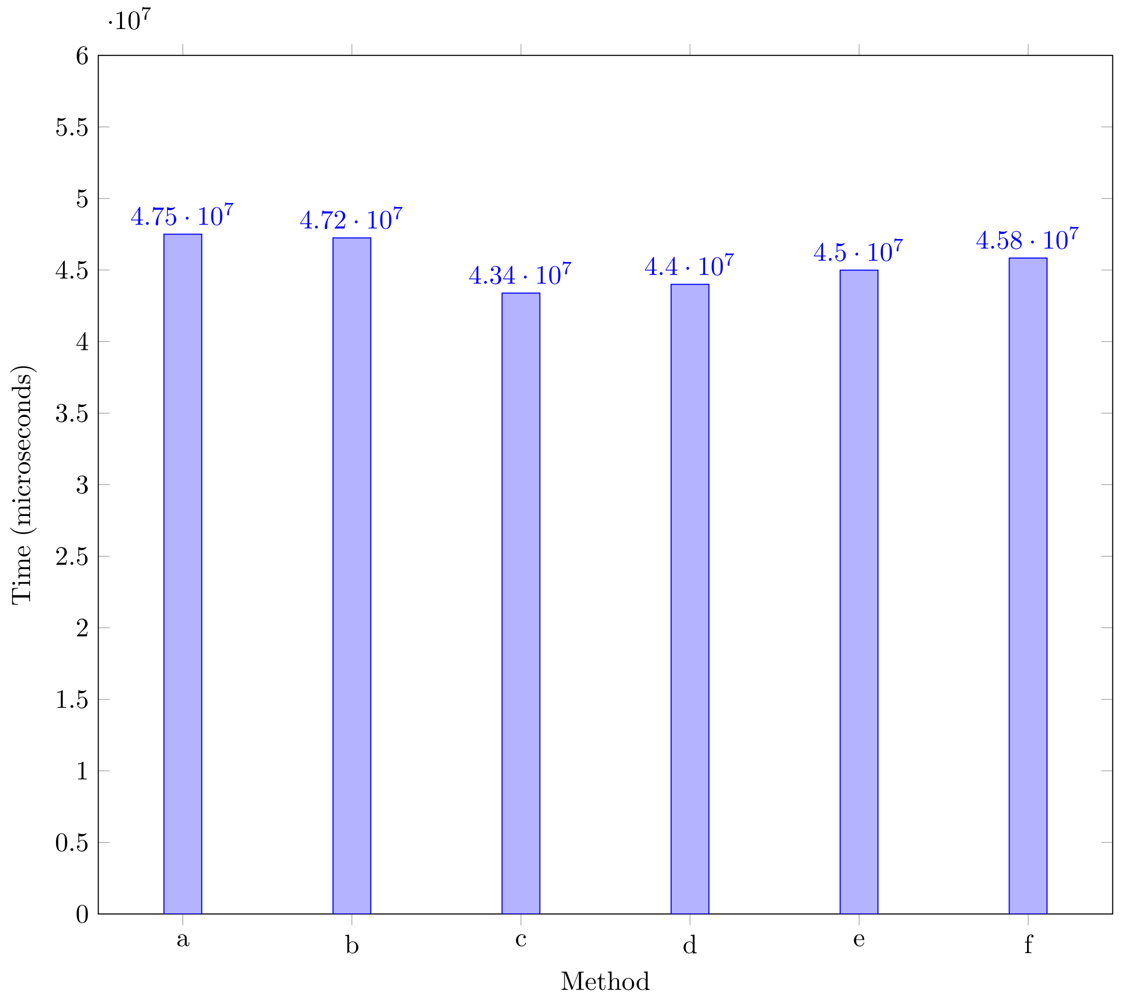

Figure 4: Execution Time for Different Methods

Method	Time (µs)
Static rowInc(a)	47503515
Static mixed(b)	47246466
Dynamic with TAS(c)	43388432
Dynamic with CAS(d)	44000090
Dynamic with Bounded CAS(e)	44990357
Dynamic with Atomic(f)	45837294

Table 4: Execution Time for Different Methods

Observation: Dynamic matrix squaring outperforms static methods by a small margin.This is because in the dynamic approach, chunks are allocated to threads dynamically, prioritizing those that arrive first but in the static approach, if certain threads complete their tasks more quickly than others, they remain idle.