Dynamic Matrix Squaring

Patel Yash Jigneshkumar

CS22BTECH11047

Contents

Chapter 1

Introduction

Parallel matrix multiplication is a fundamental operation in many scientific and engineering applications, including image processing, numerical simulations, and machine learning. The ability to perform matrix multiplication efficiently can significantly impact the performance of these applications, particularly when dealing with large datasets.

In this report, we present our implementation of parallel matrix multiplication using dynamic row allocation and different mutual exclusion methods in C++. The goal of this assignment is to explore the performance implications of various mutual exclusion techniques when executing parallel matrix multiplication.

The primary focus of this report is to investigate the impact of mutual exclusion methods, namely Test-And-Set (TAS), Compare-And-Swap (CAS), Bounded CAS, and Atomic Increment, on the overall performance of parallel matrix multiplication. We aim to measure and analyze the execution time of each method and provide insights into their effectiveness in achieving efficient parallelization.

Chapter 2

Low-level Design

The low-level design of our parallel matrix multiplication program involves several key components and considerations, including data structures, synchronization mechanisms, and thread management. In this section, we provide an overview of the design choices made for our implementation.

2.1 Data Structures

Our program utilizes two main data structures: the input matrix A and the result matrix. Both matrices are represented as two-dimensional vectors of integers (`vector<vector<int>>`). The input matrix A is read from an input file in row-major order, while the result matrix is initialized to zeros and populated with the computed values during the matrix multiplication process.

2.2 Thread Management

We employ a multi-threaded approach to parallelize the matrix multiplication task. The number of threads used is determined by the user-specified parameter K. Each thread is responsible for computing a subset of rows in the result matrix.

To dynamically allocate rows to threads, we implement a shared counter (`currentRow` or `currentRowAtomic`) that keeps track of the next available row index to process. Threads increment this counter atomically to claim a set of rows for computation. The size of each row chunk claimed by a thread is determined by the `rowInc` parameter.

2.3 Synchronization Mechanisms

Synchronization is crucial to ensure correct and efficient concurrent access to shared resources, such as the shared row counter and the result matrix. We explore four different mutual exclusion methods to synchronize access to the shared counter:

1. Test-And-Set (TAS) Mutex: Uses an atomic flag to implement a simple test-and-set lock.
2. Compare-And-Swap (CAS) Mutex: Utilizes atomic compare-and-swap operations to update the shared counter.
3. Bounded CAS Mutex: Extends CAS with a limit on the number of rows that can be claimed in each iteration.
4. Atomic Increment Mutex: Uses C++ atomic operations to atomically increment the shared counter.

Each mutual exclusion method has its advantages and trade-offs in terms of simplicity, contention, and performance.

Chapter 3

Results and Analysis

3.1 Time vs. Size (N)

3.1.1 Plots

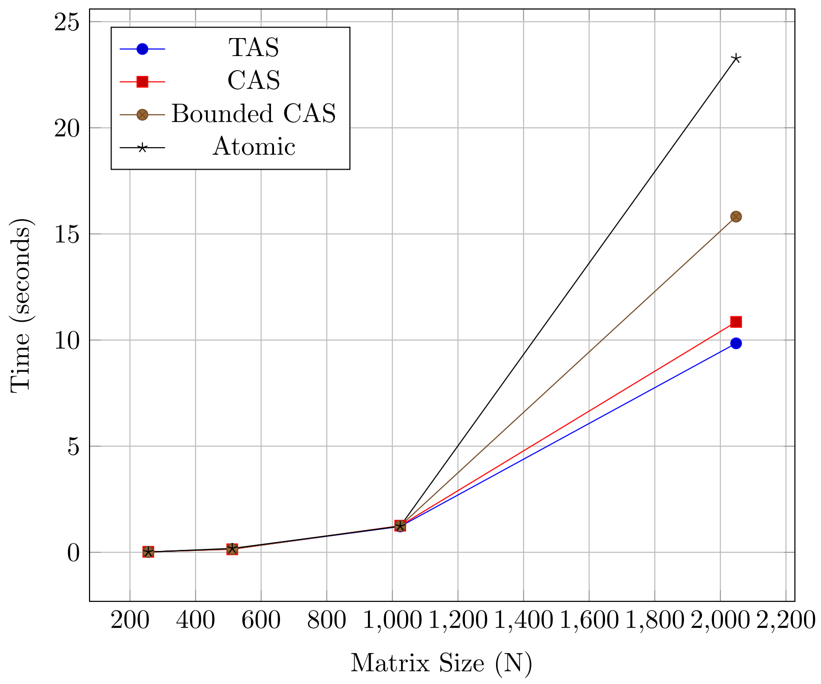

Figure 3.1: Time vs. Size plot for different mutual exclusion methods

3.1.2 Data Points for Performance Measurements

Table 3.1: Data points for Time vs. Size plot

Matrix Size (N)	**TAS Time**	**CAS Time**	**Bounded CAS Time**	**Atomic Time**
256	0.0194	0.0229	0.016375	0.01477
512	0.158186	0.141913	0.152678	0.1841
1024	1.2179	1.25451	1.24168	1.22894
2048	9.84539	10.8549	15.815	23.2714

3.1.3 Observations

- As the size of the input matrix (N) increases, the time taken to compute the square matrix also increases for all mutual exclusion methods.
- Among the different mutual exclusion methods, TAS (Test-And-Set) consistently exhibits the lowest computation time across all matrix sizes.
- The Atomic method, implemented using C++ atomic operations, consistently exhibits the highest computation time among all mutual exclusion methods across all matrix sizes.
- For larger matrix sizes (e.g., N = 2048), the computation time for all methods increases significantly compared to smaller matrix sizes (e.g., N = 256), highlighting the impact of matrix size on computation complexity.
- The increase in computation time with larger matrix sizes suggests that scalability may become a challenge when processing very large datasets using parallel matrix multiplication techniques.

3.2 Time vs. Row Increment

3.2.1 Plots

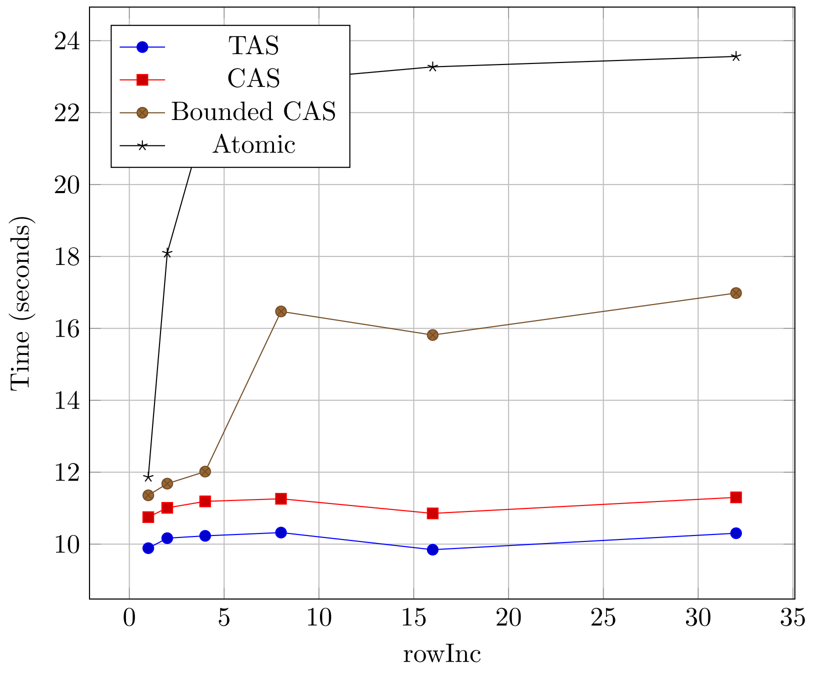

Figure 3.2: Time vs. rowInc plot for different mutual exclusion methods

3.2.2 Data Points for Performance Measurements

Table 3.2: Data points for Time vs. rowInc plot

rowInc	TAS Time	CAS Time	Bounded CAS Time	Atomic Time
1	9.88774	10.7511	11.3572	11.8632
2	10.1655	11.0094	11.6822	18.0953
4	10.2306	11.189	12.0174	21.8198
8	10.3213	11.2613	16.4708	22.8978
16	9.84539	10.8549	15.815	23.2714
32	10.3034	11.3004	16.9821	23.5624

3.2.3 Observations

- The computation time for all mutual exclusion methods generally increases as the row increment (rowInc) increases.

- TAS and CAS method exhibits relatively stable computation time across different row increments, showing minimal variation.
- The Atomic method, implemented using C++ atomic operations, demonstrates a significant increase in computation time with larger row increments, suggesting that higher levels of concurrency may lead to increased contention and synchronization overhead.
- For smaller row increments (e.g., rowInc = 1, 2, 4), TAS method generally outperforms other methods in terms of computation time, highlighting its efficiency in managing finer-grained concurrency.

3.3 Time vs. Number of threads, K

3.3.1 Plots

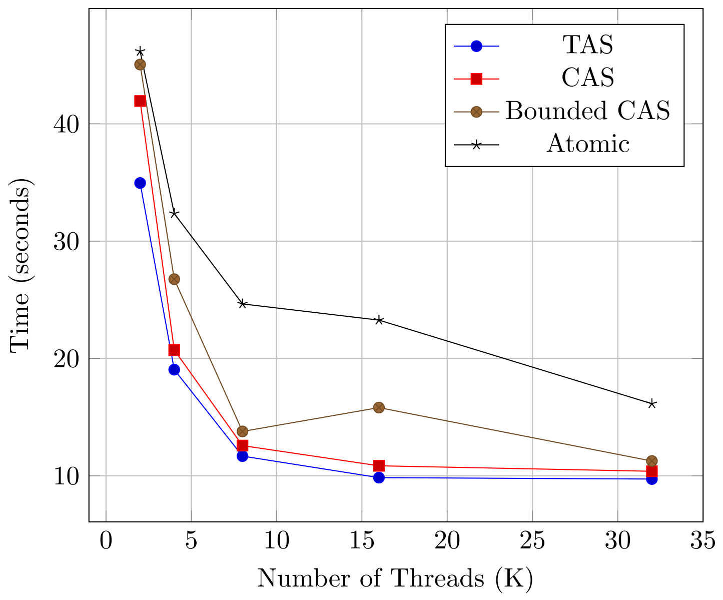

Figure 3.3: Time vs. Number of threads plot for different mutual exclusion methods

3.3.2 Data Points for Performance Measurements

Table 3.3: Data points for Time vs. Number of Threads plot

Threads (K)	**TAS Time**	**CAS Time**	**Bounded CAS Time**	**Atomic Time**
2	34.9544	41.944	45.0507	46.1826
4	19.0477	20.7247	26.759	32.355
8	11.6808	12.5765	13.7814	24.6522
16	9.84539	10.8549	15.815	23.2714
32	9.72482	10.3878	11.2601	16.1475

3.3.3 Observations

- As the number of threads (K) increases, the computation time generally decreases for all mutual exclusion methods.
- TAS and CAS method exhibits relatively stable computation time across different numbers of threads, with minor fluctuations.
- Bounded CAS (Bounded Compare-And-Swap) methods also show decreasing computation time with more threads, although the reduction in time is less pronounced compared to TAS.
- The Atomic method, implemented using C++ atomic operations, demonstrates a significant reduction in computation time as the number of threads increases, indicating the scalability of atomic operations for parallel computation.
- As the number of threads increases beyond a certain threshold (e.g., K = 16, 32), the performance gap between TAS and other methods narrows down, indicating that TAS may not scale as efficiently with very high thread counts compared to other methods.

3.4 Time vs. Algorithms

3.4.1 Plots

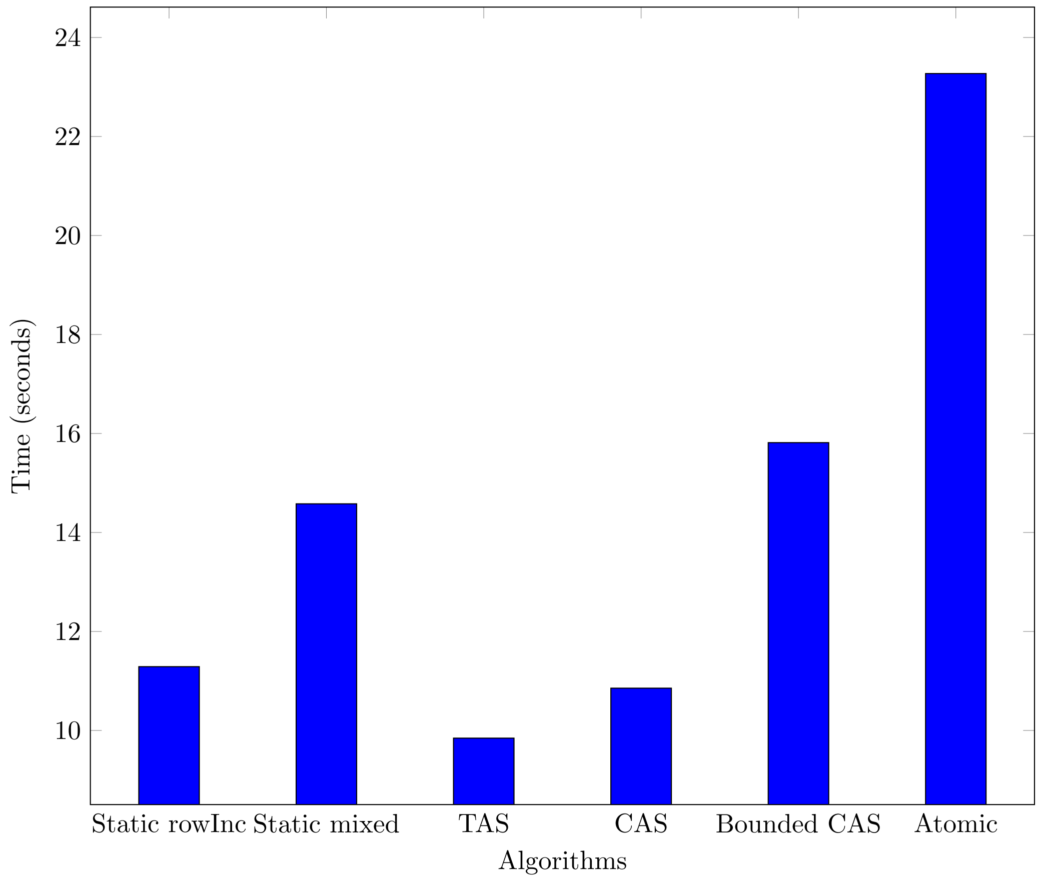

Figure 3.4: Time vs. Algorithms plot

3.4.2 Data Points for Performance Measurements

Table 3.4: Data points for Time vs. Algorithms plot

Algorithm	**Time (seconds)**
Static rowInc	11.289
Static mixed	14.5788
TAS	9.84539
CAS	10.8549
Bounded CAS	15.815
Atomic	23.2714

3.4.3 Observations

- The computation time varies significantly across different mutual exclusion methods and concurrency strategies.
- The Mixed method (Static mixed) shows slightly higher computation time compared to the Chunk method, suggesting that the variability in chunk sizes may introduce additional overhead in task distribution.
- Dynamic concurrency strategies, such as TAS (Test-And-Set) and CAS (Compare-And-Swap), demonstrate competitive performance with relatively low computation times compared to static methods.
- Bounded CAS (Bounded Compare-And-Swap) exhibits higher computation time compared to TAS and CAS methods, indicating potential overhead associated with the additional constraint on memory access.
- The Atomic method, implemented using C++ atomic operations, demonstrates the highest computation time among all methods, highlighting the overhead associated with fine-grained synchronization using atomic operations.