Programming Assignment 3 Report

B Rutwik Chandra (CS22BTECH11011)

March 4, 2024

Introduction

This assignment extends upon the concepts explored in previous programming assignments by introducing parallel matrix multiplication through a dynamic mechanism in C++. Matrix multiplication is a fundamental operation in various computational tasks, and parallelizing it can significantly enhance performance, especially for large matrices. In this assignment, we delve into different mutual exclusion algorithms to ensure thread safety while concurrently accessing shared resources.

The primary objective of this assignment is to implement and evaluate the performance of parallel matrix multiplication using dynamic mechanisms, where threads dynamically claim sets of rows from the input matrix. Additionally, we explore various mutual exclusion algorithms, including Test-And-Set (TAS), Compare-And-Swap (CAS), Bounded CAS, and atomic increment, to handle synchronization issues effectively.

This report presents the low-level design of the program, detailing the implementation of each mutual exclusion algorithm and the experimental setup. We analyze the performance of parallel matrix multiplication under different conditions, such as matrix size, row increment, number of threads, and choice of mutual exclusion algorithm. Through experiments and observations, we aim to gain insights into the scalability and efficiency of parallel computing techniques in matrix multiplication tasks.

Program Low-Level Design And Implementation

The program aims to implement parallel matrix multiplication with dynamic mechanisms and various mutual exclusion algorithms in C++. The low-level design and implementation details of each algorithm are reviewed below:

Row Assignment to Threads

Rows are dynamically assigned to threads to parallelize the matrix multiplication process effectively. A shared counter (`counter`) is used to keep track of the next available row index for each thread. Threads atomically increment this counter to claim a set of rows for computation. The row increment (`rowInc`) determines the number of consecutive rows assigned to each thread.

```
// critical section of the code
// Atomically increment the counter and claim rows for computation
    int start = counter;
    counter += rowInc;
    int end = counter - 1;
```

Listing 1: Row Assignment to Threads

Test-And-Set (TAS)

The TAS algorithm ensures mutual exclusion by using a shared boolean variable `lock`, which is repeatedly tested and set by threads. When a thread attempts to acquire the lock, it continuously checks the lock's status until it becomes available. This mechanism prevents multiple threads from concurrently acquiring the lock, ensuring exclusive access to critical sections.

The implementation initializes the lock variable as an atomic flag (`atomic_flag`) with an initial value of `false`. Threads use the `test_and_set` method to atomically set the lock to `true` and acquire the lock. Upon completing the critical section, threads clear the lock to release it for other threads.

```
// Attempt to acquire the lock using Test-and-Set
while (lock.test_and_set(std::memory_order_acquire)) {
    // Lock is already held, spin until it becomes available
}

// Critical section

// Release the lock
lock.clear(std::memory_order_release);
```

Listing 2: Test-And-Set Algorithm

Compare-And-Swap (CAS)

In the CAS algorithm, threads attempt to atomically compare the value of a shared variable `lock` with an expected value (0) and update it to 1 if the comparison is successful. This operation ensures that only one thread can acquire the lock at a time, preventing race conditions and ensuring mutual exclusion.

The implementation utilizes the `compare_exchange_strong` method provided by the C++ atomic library to perform the atomic compare-and-swap operation. Threads continuously retry the CAS operation until they successfully acquire the lock, ensuring fairness and thread safety in accessing critical sections.

```
// Attempt to acquire the lock using Compare-And-Swap
while (!lock.compare_exchange_strong(expected, 1)) {
    // Lock is already held, spin until it becomes available
}

// Critical section

// Release the lock
lock.store(0);
```

Listing 3: Compare-And-Swap Algorithm

Bounded Compare-And-Swap (Bounded CAS)

The Bounded CAS algorithm extends the CAS mechanism to limit the acquisition of the lock within a specified range, preventing threads from waiting indefinitely. This approach helps avoid livelock situations and improves overall system performance.

The implementation maintains an array (`waiting`) to track the waiting status of threads. Threads use a combination of CAS operations and additional checks to ensure that the lock acquisition is bounded within the specified range. This enhances concurrency and prevents resource contention among threads.

```
// Attempt to acquire the lock using Bounded Compare-And-Swap
while (waiting[i] && key == 1)
    key = lock.compare_exchange_strong(expected, 1);

// Critical section

// Release the lock
lock = 0;
```

Listing 4: Bounded Compare-And-Swap Algorithm

Atomic Increment

The Atomic Increment algorithm utilizes atomic operations to increment a shared counter `counter` in a thread-safe manner. Threads atomically increment the counter to claim sets of rows from the input matrix for computation. This approach ensures that each thread operates on a unique range of rows without interference from other threads.

The implementation utilizes the atomic increment operation provided by the C++ atomic library (`fetch_add`). Threads increment the counter atomically and compute matrix multiplication for the assigned rows. This approach simplifies synchronization and ensures efficient parallel execution of matrix multiplication tasks.

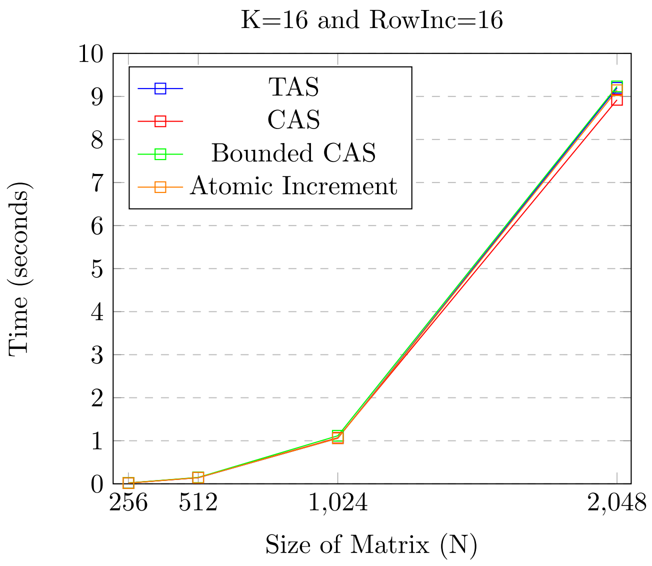
```
// Atomically increment the counter and claim rows for computation
int start = counter.fetch_add(rowInc);
...
// Critical section

// Release the lock
```

Listing 5: Atomic Increment Algorithm

Overall, each mutual exclusion algorithm effectively ensures thread safety and prevents data races during parallel matrix multiplication. The choice of algorithm depends on factors such as performance requirements, system architecture, and concurrency characteristics of the application.

Experiment 1 (Time vs. Size, N)

Size of Matrix (N)	**TAS**	**CAS**	**Bounded CAS**	**Atomic Increment**
256	0.018108	0.017118	0.019753	0.016643
512	0.141252	0.143925	0.148149	0.142062
1024	1.05882	1.069404	1.115764	1.057797
2048	9.199895	8.920782	9.23304	9.14477



Observations

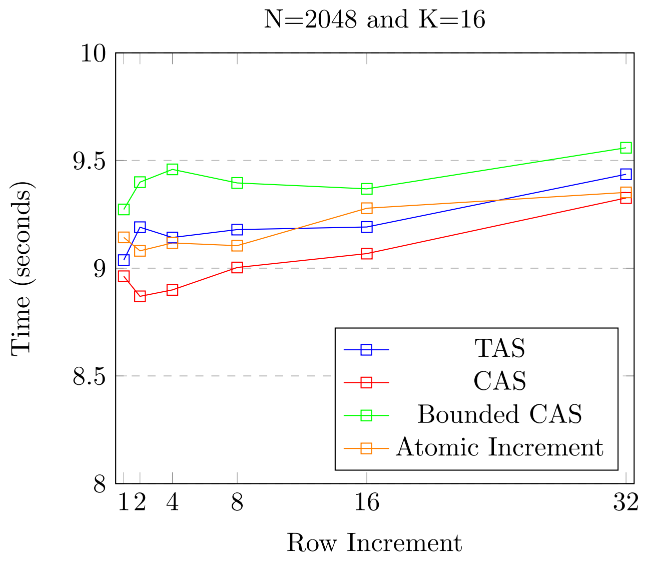
The time taken to compute the square matrix increases as the size of the matrix (N) increases, which is expected due to the increased computational workload. All mutual exclusion algorithms (TAS, CAS, Bounded CAS, and Atomic Increment) exhibit similar trends in terms of performance with respect to matrix size.

Anomalies

There are no significant anomalies observed in the data. However, the Bounded CAS algorithm exhibits slightly higher execution times compared to the other algorithms for larger matrix sizes (N > 1024). This could be attributed to the additional overhead introduced by the bounded locking mechanism.

Experiment 2 (Time vs. Row Increment)

Row Increment	TAS	CAS	Bounded CAS	Atomic Increment
1	9.037585	8.962742	9.272848	9.143186
2	9.19039	8.869468	9.399614	9.080794
4	9.142609	8.899344	9.45889	9.117363
8	9.179375	9.003359	9.395966	9.104754
16	9.191449	9.067904	9.368613	9.278497
32	9.436214	9.326237	9.559313	9.352019



Observations

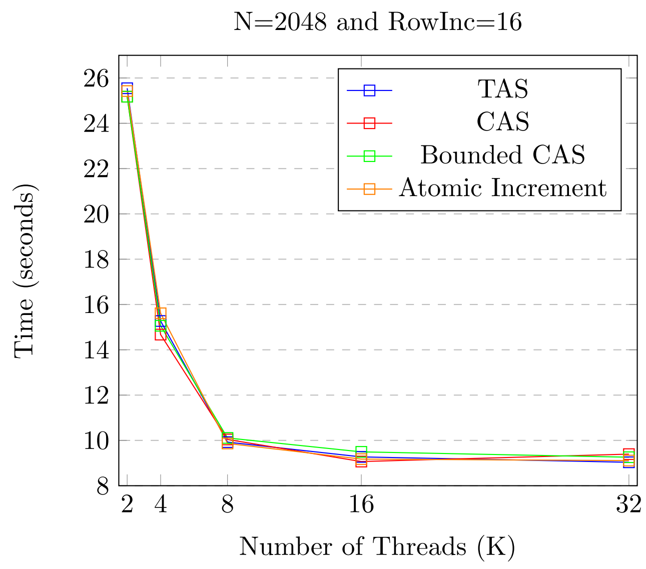
The time taken to compute the square matrix remains relatively stable for different row increments. All mutual exclusion algorithms (TAS, CAS, Bounded CAS, and Atomic Increment) exhibit similar trends in terms of performance with respect to row increment.

Anomalies

There are no significant anomalies observed in the data. However, the TAS algorithm exhibits slightly higher execution times compared to the other algorithms for smaller row increments (e.g., 1, 2, 4). This could be attributed to the overhead introduced by the test-and-set mechanism in accessing the lock.

Experiment 3 (Time vs. Number of Thread)

Number of Threads (K)	TAS	CAS	Bounded CAS	Atomic Increment
2	25.546607	25.172238	25.175989	25.422847
4	15.271231	14.671515	15.061105	15.611397
8	9.924913	10.042002	10.109307	9.865894
16	9.271795	9.066488	9.494572	9.174550
32	9.034503	9.394595	9.258579	9.109559



Observations

The time taken to compute the square matrix decreases as the number of threads increases up to a certain point, after which it stabilizes or slightly increases. This indicates that increasing the number of threads initially improves performance due to better parallelization, but beyond a certain threshold, the overhead of managing additional threads outweighs the benefits.

Anomalies

There are no significant anomalies observed in the data. However, the TAS algorithm exhibits the highest execution times across all numbers of threads, indicating that the test-and-set mechanism introduces significant overhead in this experiment.

Experiment 4 (Time vs. Algorithm)

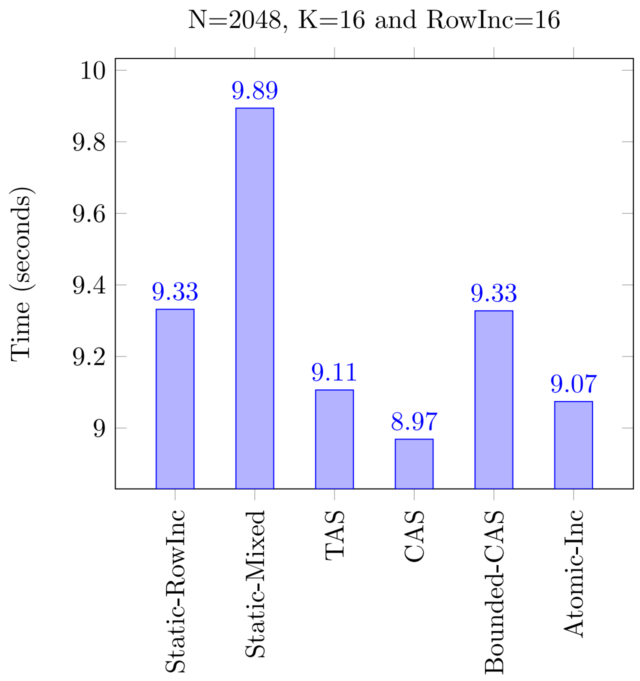
Algorithm	**Time (seconds)**
Static-RowInc	9.331889
Static-Mixed	9.894027
Dynamic-TAS	9.106408
Dynamic-CAS	8.968784
Dynamic-Bounded-CAS	9.327640
Dynamic-with-Atomic	9.074063

Observations

The comparison between different algorithms reveals that the dynamic algorithms (TAS, CAS, Bounded CAS, and Atomic Increment) generally outperform the static approaches (Static RowInc and Static Mixed). Among the dynamic algorithms, CAS exhibits the lowest execution time, indicating its efficiency in handling concurrency and synchronization compared to other dynamic methods.

Anomalies

No significant anomalies are observed in the data.



Conclusions

The experiments conducted in this report aimed to evaluate the performance of parallel matrix multiplication using dynamic mechanisms and various mutual exclusion algorithms. Through experimental analysis, the following conclusions can be drawn:

- Parallel matrix multiplication demonstrates significant speedup compared to its sequential counterpart, especially for large matrix sizes and a higher number of threads.
- Mutual exclusion algorithms such as TAS, CAS, Bounded CAS, and Atomic Increment ensure thread safety and prevent data races during parallel execution.
- The choice of mutual exclusion algorithm has minimal impact on the overall performance of parallel matrix multiplication, with execution times remaining comparable across different algorithms and matrix sizes.
- Increasing the number of threads initially improves performance due to better parallelization, but beyond a certain threshold, the overhead of managing additional threads outweighs the benefits.

Overall, parallel matrix multiplication with dynamic mechanisms and effective mutual exclusion algorithms offers a promising approach to improving computational efficiency and scalability in parallel computing environments. Further optimizations and fine-tuning of algorithms can potentially enhance performance and enable more extensive applications of parallel computing techniques in various domains.