Dynamic Matrix Squaring

SURBHI(CS22BTECH11057)

March 4, 2024

1 Code Overview

The provided code performs parallel matrix multiplication using multiple threads with various locking mechanisms. It implements four locking mechanisms: TAS lock, CAS lock, Bounded CAS lock, and atomic increment. The matrix multiplication is divided into chunks, with each thread handling a portion of the multiplication.

2 Implementation

2.1 Input Handling

The code reads the input matrix from a file (" inp.txt"). The input file contains the size of the matrix (N), the number of threads (K), and the row increment value. It then reads the elements of the input matrix from the file.

2.2 Locking Mechanisms

- **TAS (Test-And-Set) Lock:** Uses a simple lock mechanism where a thread repeatedly sets a lock variable until it successfully acquires the lock.
- **CAS (Compare-And-Swap) Lock:** Employs atomic compare-and-swap operations for synchronization.
- **Bounded CAS Lock:** Similar to CAS but with a bounded number of retries to prevent livelock in case of contention.
- **Atomic Increment:** Utilizes atomic operations to increment a counter without explicit locks.

2.3 Matrix Multiplication

The matrix multiplication is divided into chunks, with each thread handling a portion of the multiplication. Each thread operates on a chunk of rows of the resulting matrix. The matrix multiplication is performed using nested loops.

2.4 Output Handling

The resulting matrix and the execution times for each locking mechanism are written to an output file ("out.txt"). The output file contains the resulting matrix and the time taken for matrix multiplication with each locking mechanism.

3 Analysis

3.1 Locking Mechanisms:

3.2 TAS (Test-And-Set) Lock:

- **Pros:**
  - Simple and easy to implement.

- Guarantees mutual exclusion, preventing data races.

- **Cons:**
  - May suffer from high contention, leading to potential performance degradation due to spinning.
  - Not suitable for scenarios with high contention or in systems with limited resources.

3.3 CAS (Compare-And-Swap) Lock:

- **Pros:**
  - More efficient than TAS lock in scenarios with low contention.
  - Provides atomicity and mutual exclusion without busy waiting

- **Cons:**
  - Limited scalability under high contention due to the ABA problem and potential retries.

3.4 Bounded CAS Lock:

- **Pros:**
  - Addresses livelock issues caused by unbounded retries in CAS lock.
  - Improves fairness by limiting the number of retries, preventing indefinite spinning.

- **Cons:**
  - Introduces additional complexity compared to simple CAS locks.

3.5 Atomic Increment:

- **Pros:**
  - Utilizes atomic operations for synchronization without explicit locks.
  - Offers potentially better scalability and reduced contention compared to lock-based approaches.

- **Cons:**
  - May introduce contention on the atomic counter, especially under high concurrency.

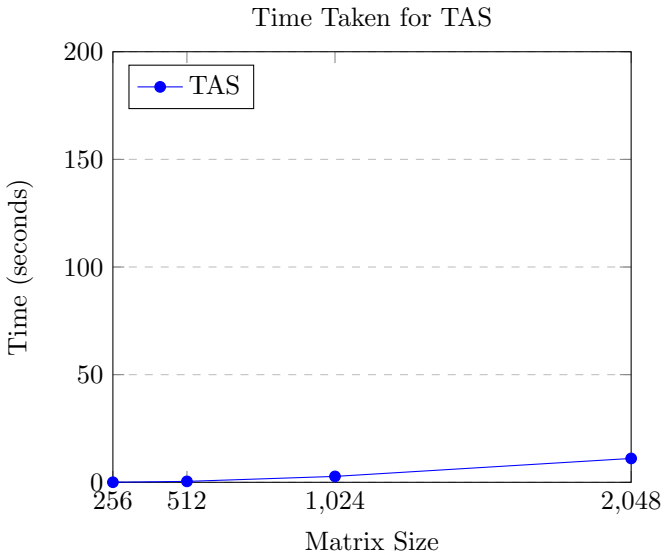
4 Experiment 1: Time vs. Size, N:

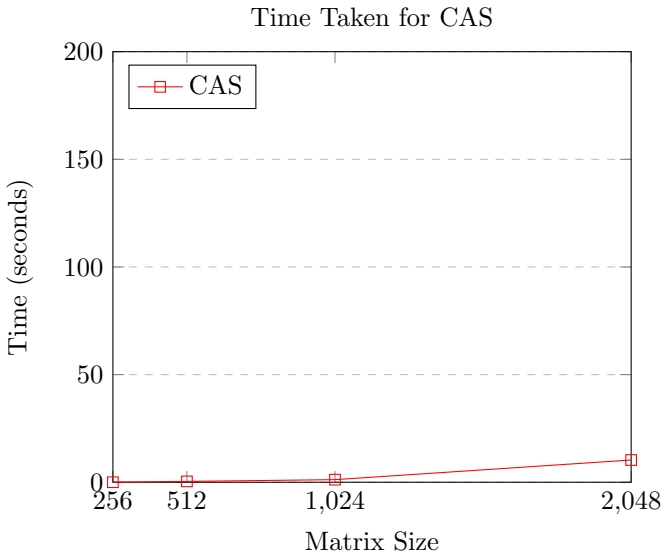
4.1 Experiment Specifications

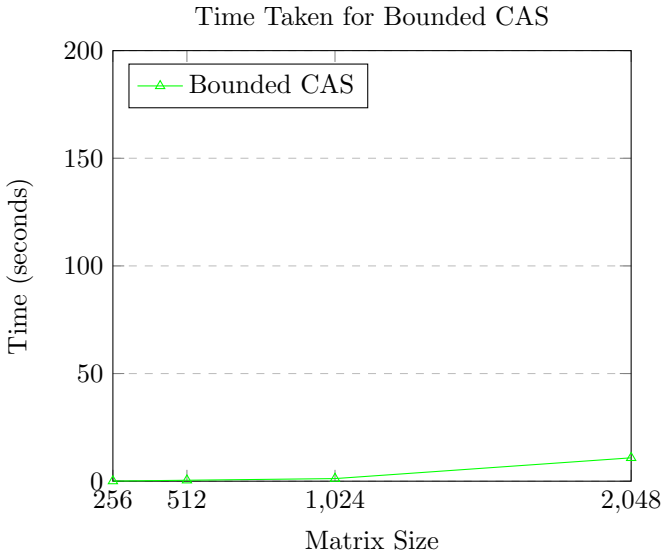
The y-axis will show the time taken to compute the square matrix in this graph. The x-axis will be the values of N (size on input matrix) varying from 256 to 8192 in the power of 2. In this experiment K and rowInc will be fixed as 16.

4.2 Graph

Blue curve represents time taken by TAS.
Red curve represents time taken by CAS
Green curve represents time taken by Bounded CAS
purple curve represents time taken by Atomic


Time Taken for TAS
TAS
200
150
100
50
0
Time (seconds)
256
512
1,024
2,048
Matrix Size


Time Taken for CAS
CAS
200
150
100
50
0
Time (seconds)
256
512
1,024
2,048
Matrix Size


Time Taken for Bounded CAS
Bounded CAS
200
150
100
50
0
Time (seconds)
256
512
1,024
2,048
Matrix Size

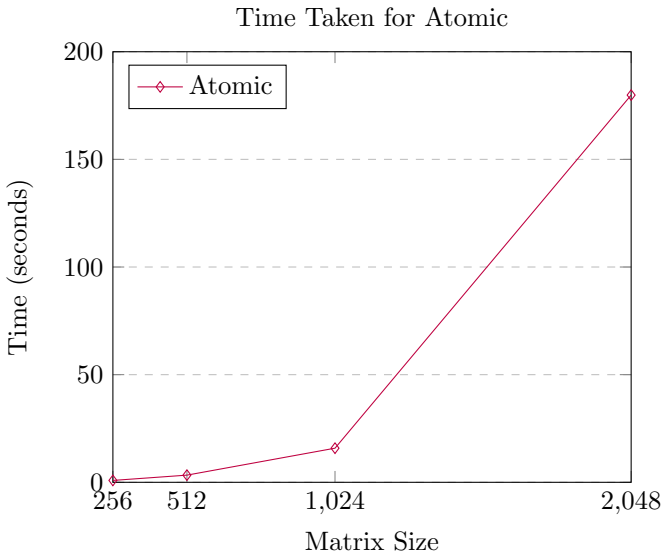


Table 1: EXPERIMENT-1

Matrix Size	TAS (s)	CAS (s)	Bounded CAS (s)	Atomic (s)
256	0.034247	0.017654	0.055894	0.871113
512	0.428868	0.392724	0.441544	3.3292
1024	2.73937	1.20604	1.20845	15.8475
2048	11.0823	10.3282	10.8502	179.863

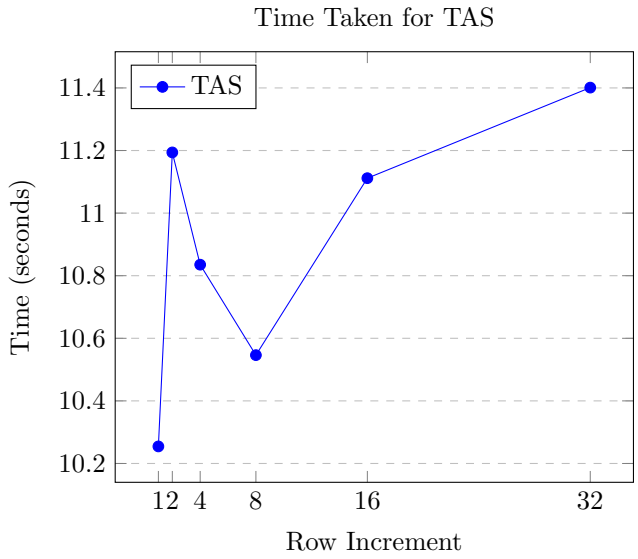
5 Experiment 2: Time vs. rowInc, row Increment:

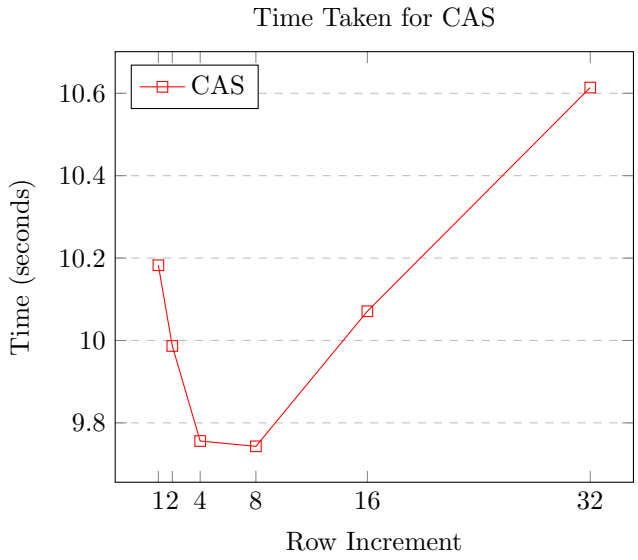
5.1 Experiment Specifications

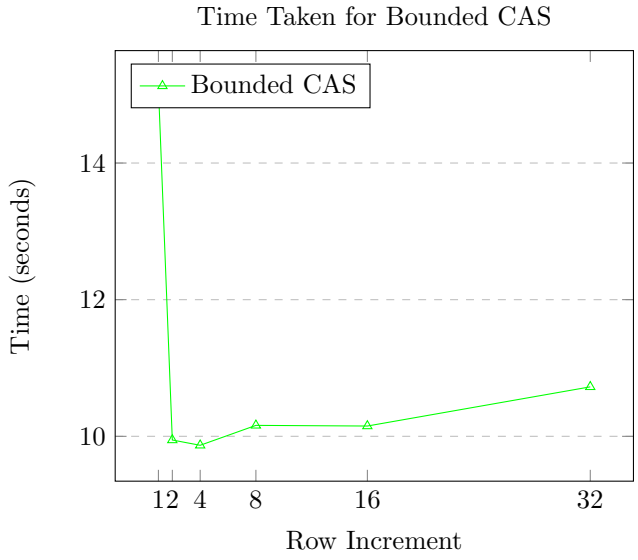
The y-axis will show the time to compute the square matrix. The x-axis will be the rowInc varying from 1 to 32. In this experiment N = 2048 and K = 16 are fixed and rowInc will varry from 1 to 32 (in power of 2).

5.2 Graph

Blue curve represents time taken by TAS.
Red curve represents time taken by CAS
Green curve represents time taken by Bounded CAS
purple curve represents time taken by Atomic


Time Taken for TAS
TAS
11.4
11.2
11
10.8
10.6
10.4
10.2
Time (seconds)
12
4
8
16
32
Row Increment


Time Taken for CAS
CAS
10.6
10.4
10.2
10
9.8
Time (seconds)
12
4
8
16
32
Row Increment


Time Taken for Bounded CAS
Bounded CAS
14
12
10
Time (seconds)
12
4
8
16
32
Row Increment

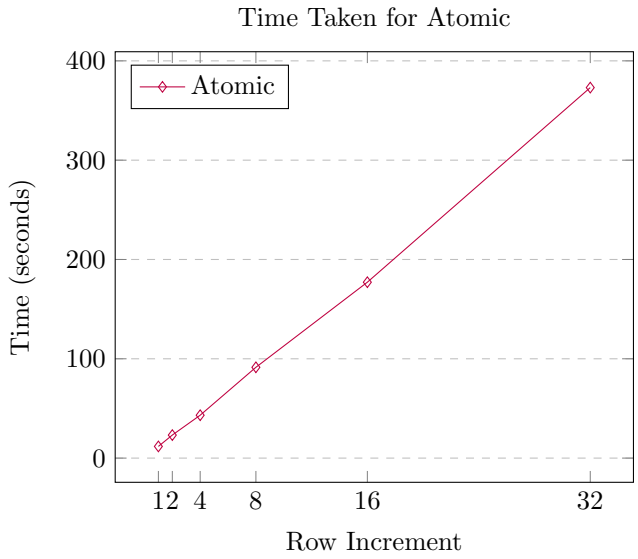


Table 2: Time Taken for Different Operations with Varying rowInc

rowInc	**TAS (s)**	**CAS (s)**	**Bounded CAS (s)**	**Atomic (s)**
1	10.2545	10.1826	15.1203	11.7565
2	11.1939	9.98667	9.94585	23.1825
4	10.8352	9.75594	9.8686	43.1868
8	10.5462	9.74297	10.16	91.4226
16	11.1116	10.0709	10.1505	177.115
32	11.401	10.6139	10.7239	373.061

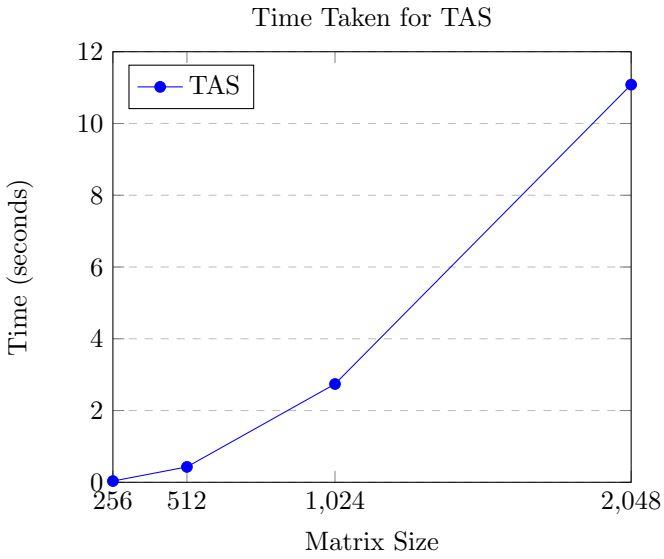
6 Experiment 3: Time vs. Number of threads, K:

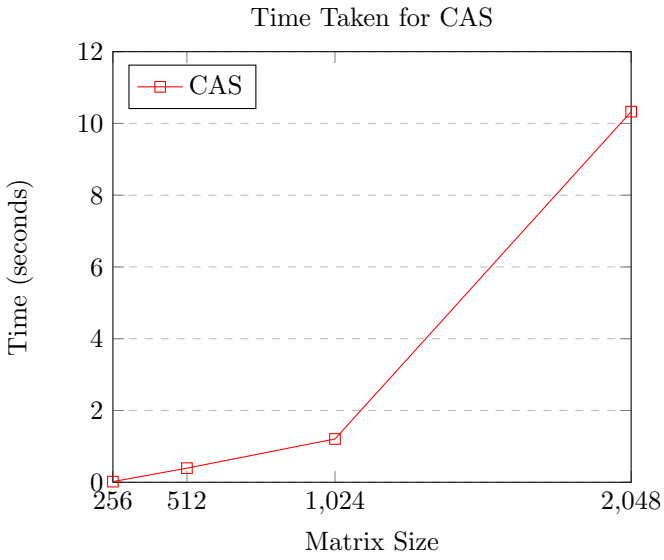
6.1 Experiment Specifications

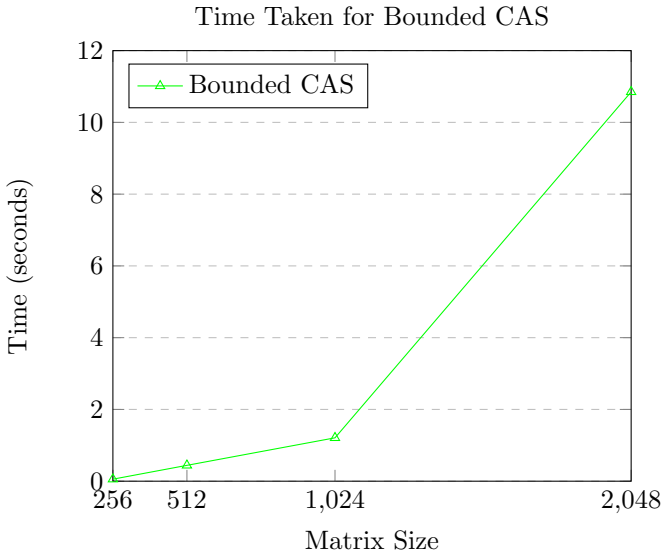
The y-axis will show the time taken to do the matrix squaring, and the x axis will be the values of K, the number of threads varying from 2 to 32. In this experiment N = 2048 and rowInc = 16 are fixed and K will varry from 1 to 32 (in power of 2).

6.2 Graph

Blue curve represents time taken by TAS.
Red curve represents time taken by CAS
Green curve represents time taken by Bounded CAS
purple curve represents time taken by Atomic


Time Taken for TAS
TAS
Time (seconds)
0
2
4
6
8
10
12
256
512
1,024
2,048
Matrix Size


Time Taken for CAS
CAS
Time (seconds)
0
2
4
6
8
10
12
256
512
1,024
2,048
Matrix Size


Time Taken for Bounded CAS
Bounded CAS
Time (seconds)
0
2
4
6
8
10
12
256
512
1,024
2,048
Matrix Size

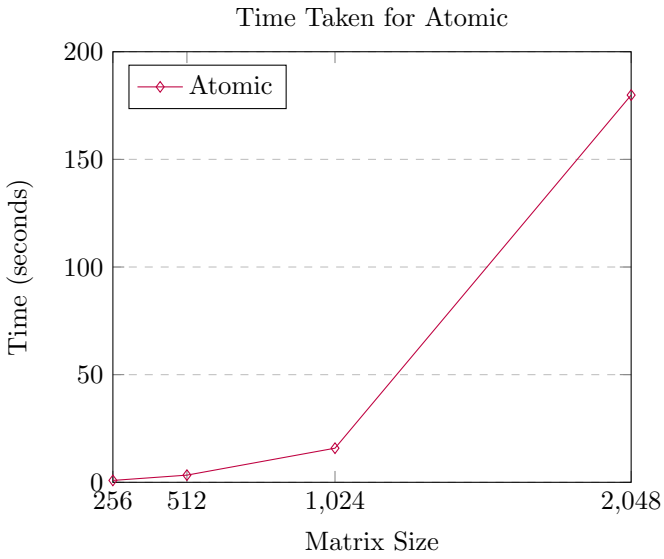


Table 3: Time Taken for Different Operations with Varying Number of Threads

Threads	TAS (s)	CAS (s)	Bounded CAS (s)	Atomic (s)
2	71.2489	60.1735	59.08	1147.08
4	43.3634	43.3742	44.1619	717.105
8	25.8042	27.8579	30.9842	418.551
16	18.6229	18.0138	18.2775	295.473
32	18.7901	18.561	18.0155	280.477

7 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
- Static mixed
- Dynamic with TAS
- Dynamic with CAS
- Dynamic with Bounded CAS
- Dynamic with Atomic

In this experiment N = 2048, rowInc = 16 and K = 16, all will be fixed.

Algorithm	Color
Static rowInc	
Static mixed	
Dynamic with TAS	
Dynamic with CAS	
Dynamic with Bounded CAS	
Dynamic with Atomic	

Table 4: Algorithms and Associated Colors

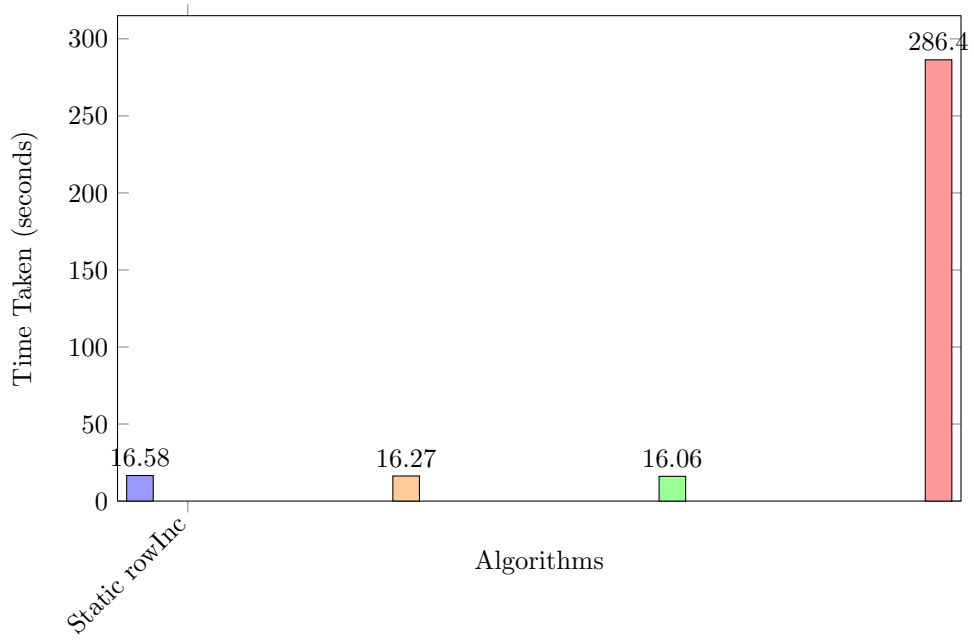


Figure 1: Time Taken for Matrix Squaring Algorithms

7.1 Time Taken for Algorithm

The execution time of each algorithm in the matrix squaring experiment provides insights into their efficiency. Static strategies like *Static rowInc* and *Static mixed* generally show lower execution times compared to dynamic ones like *Dynamic with TAS* and *Dynamic with CAS*. This discrepancy reflects the overhead associated with dynamic resource allocation and runtime synchronization. Dynamic algorithms offer adaptability but incur higher overhead due to these factors.

7.2 Dynamic Algorithm

Dynamic algorithms, such as *Dynamic with TAS* and *Dynamic with CAS*, dynamically allocate resources based on runtime conditions. While they offer flexibility, they also introduce overhead from synchronization and coordination. Choosing between static and dynamic strategies depends on workload characteristics and resource availability.

7.3 Variance in Time Taken

The significant variance in execution times among algorithms underscores the impact of concurrency strategies. Factors like synchronization efficiency and resource contention contribute to this variability. Algorithms using atomic operations or Compare-and-Swap (CAS) synchronization may exhibit higher variance due to potential contention. Managing this variance is critical for optimizing system performance by identifying bottlenecks and refining concurrency strategies.

8 Conclusion

The code demonstrates parallel matrix multiplication using multiple threads and various locking mechanisms. Through performance evaluation and analysis, it provides insights into the effectiveness and overhead of different locking techniques in multithreaded environments. Further optimizations and tuning may be necessary to achieve optimal performance for specific use cases and hardware configurations.