REPORT
CS22BTECH11006

Low-Level Design Report

Overview

The implementation of parallel matrix multiplication using different mutual exclusion mechanisms: Test-and-Set (TAS), Compare-and-Swap (CAS), Bounded Compare-and-Swap (Bounded CAS), and Atomic Increment. Each code segment utilizes multithreading to distribute the workload across multiple threads and achieve parallelism in matrix multiplication.

1. Test-and-Set (TAS) Implementation

- Shared Variables: The TAS implementation uses an atomic flag named `lock` to manage mutual exclusion. Additionally, an atomic integer `counter` is employed to keep track of the current row being processed.
- Worker Function: The `worker` function is responsible for performing matrix multiplication for a portion of rows. It uses a while loop to acquire the lock using Test-and-Set and then proceeds to process rows based on the current counter value.

- Main Function: In the main function, threads are created to execute the worker function. Once all threads complete execution, the resulting matrix and execution time are written to an output file.

2. Compare-and-Swap (CAS) Implementation

- Shared Variables: Similar to the TAS implementation, CAS utilizes an atomic flag `lock` and an atomic integer `counter` to coordinate access to shared resources.
- Worker Function: The `worker` function employs a while loop with Compare-and-Swap (CAS) to acquire the lock. It then proceeds to process rows for matrix multiplication.
- Main Function: Threads are created and joined in the main function to execute the worker function. The resulting matrix and execution time are written to an output file.

3. Bounded Compare-and-Swap (Bounded CAS) Implementation

- Shared Variables: Bounded CAS utilizes an atomic integer `counter` and a boolean atomic flag `lock` to coordinate access to shared resources.
- Bounded CAS Function: A custom function `boundedCAS` simulates the Bounded

Compare-and-Swap operation by limiting retries to acquire the lock.

- Worker Function: Similar to the CAS implementation, the worker function employs CAS to acquire the lock and performs matrix multiplication.
- Main Function: Threads are created and joined in the main function to execute the worker function. The resulting matrix and execution time are written to an output file.

4. Atomic Increment Implementation

- Shared Variables: This implementation uses an atomic integer `counter` for synchronization.
- Worker Function: The `worker` function employs atomic fetch-and-add operations to increment the counter and distribute row processing among threads.
- Main Function: Threads are created and joined to execute the worker function. The resulting matrix and execution time are written to an output file.

Conclusion

Overall, demonstration of the implementation of different mutual exclusion mechanisms for parallel matrix multiplication. Each implementation offers unique advantages and trade-offs in terms of performance, scalability, and complexity. By analyzing the execution time and behavior of each mechanism, one can gain insights into their suitability for specific parallel computing scenarios. Additionally, optimizations and further enhancements can be explored to improve the efficiency and robustness of these implementations.

GRAPHS

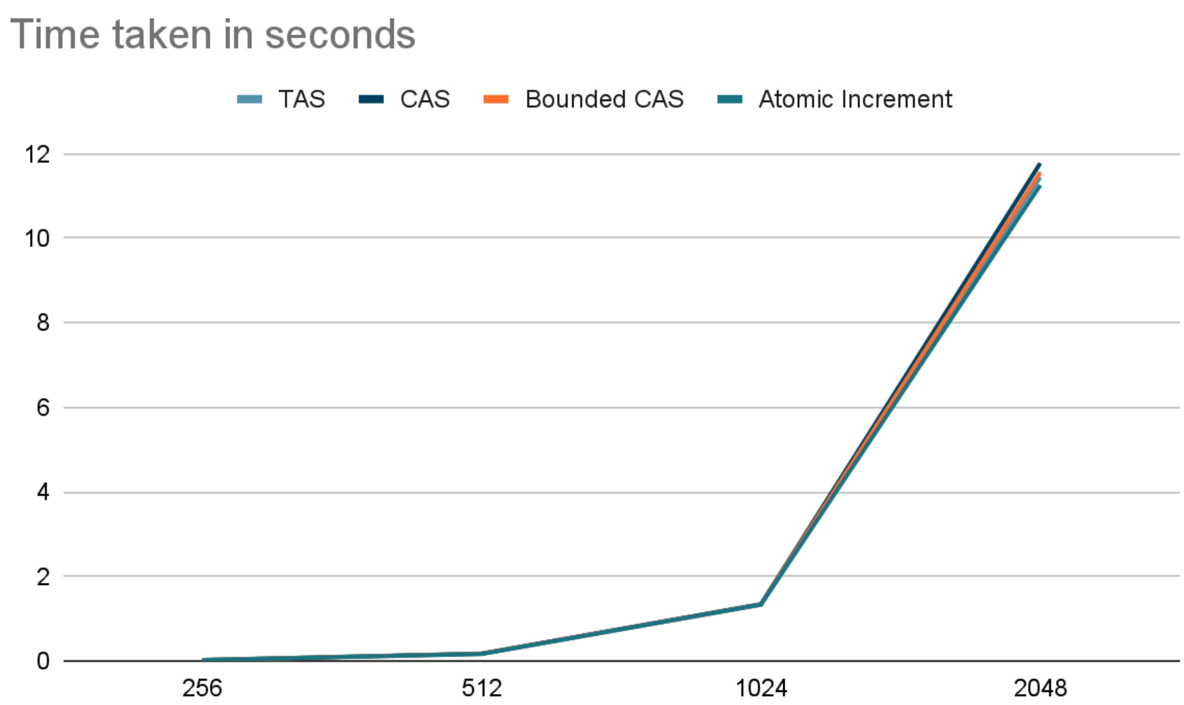
EXPERIMENT 1:

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 2048 (size of the matrix will vary as 256*256, 512*512, 1024*1024, ...) in the power of 2. Please have the rowInc fixed at 16 and K at 16 for all these experiments.

Table1 :N,time taken for TAS,CAS

Bounded CAS ,Atomic Increment in seconds

N	TAS	CAS	Bounded CAS	Atomic Increment
256	0.020568	0.0217	0.019778	0.019643
512	0.17066	0.174237	0.168616	0.163978
1024	1.34313	1.34313	1.3331	1.33052
2048	11.449	11.7788	11.5675	11.2596



Observations :

1. The execution times for TAS, CAS, Bounded CAS, and Atomic Increment show marginal differences across varying matrix sizes, indicating comparable performance.
2. As the matrix size increases, execution times also rise proportionally for all mechanisms, though not linearly.
3. Atomic Increment consistently exhibits slightly better efficiency compared to other methods, with the lowest execution times observed.
4. Despite differences, the overall performance characteristics of TAS, CAS, Bounded CAS, and Atomic Increment remain stable and comparable.

5. While execution time is a factor, considerations such as implementation complexity and adaptability should guide the choice of mutual exclusion mechanisms.
6. The scalability and efficiency of Atomic Increment suggest its potential advantage over TAS, CAS, and Bounded CAS for managing shared resources in parallel computing tasks.

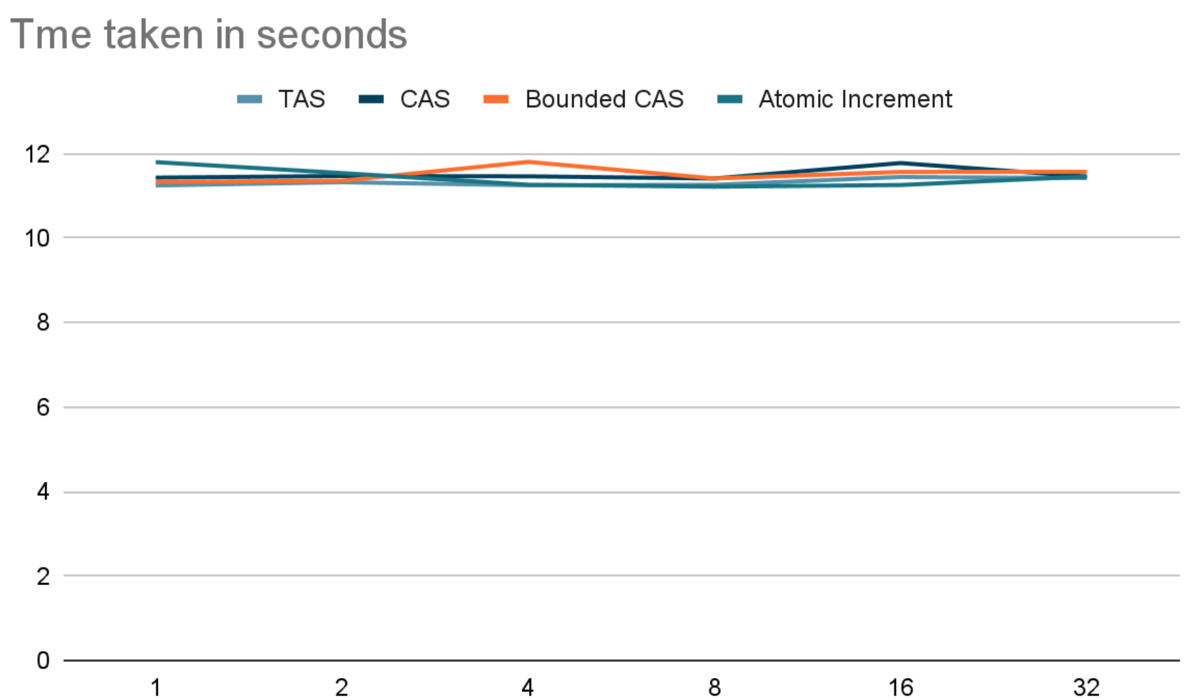
Experiment - 2:

Time vs. rowInc: 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). Please fix N at 2048 and K at 16 for all these experiments.

Table2: RowInc, time taken for TAS,CAS Bounded CAS ,Atomic Increment in seconds

RowInc	TAS	CAS	Bounded CAS	Atomic Increment
1	11.2491	11.4334	11.331	11.8009
2	11.3262	11.4729	11.3608	11.5416
4	11.253	11.4646	11.8067	11.2657
8	11.2639	11.4135	11.4133	11.2196
16	11.449	11.7788	11.5695	11.2596

32	11.4243	11.4586	11.5724	11.4607



Observations :

1. Increasing the `RowInc` parameter leads to varied execution times across all mutual exclusion mechanisms, suggesting sensitivity to workload distribution.
2. TAS, CAS, and Bounded CAS show consistent trends with `RowInc` increments, exhibiting minor fluctuations in execution times.
3. Atomic Increment demonstrates stable performance across different `RowInc` values, showcasing its resilience to variations in workload distribution.

4. The choice of mutual exclusion mechanism should consider both execution time and scalability, as seen in the fluctuating trends with `RowInc` adjustments.
5. TAS, CAS, Bounded CAS, and Atomic Increment offer distinct trade-offs in performance and adaptability under varying workload configurations.
6. Fine-tuning parameters like `RowInc` alongside careful selection of mutual exclusion mechanisms can optimize parallel computing tasks for efficiency and scalability.

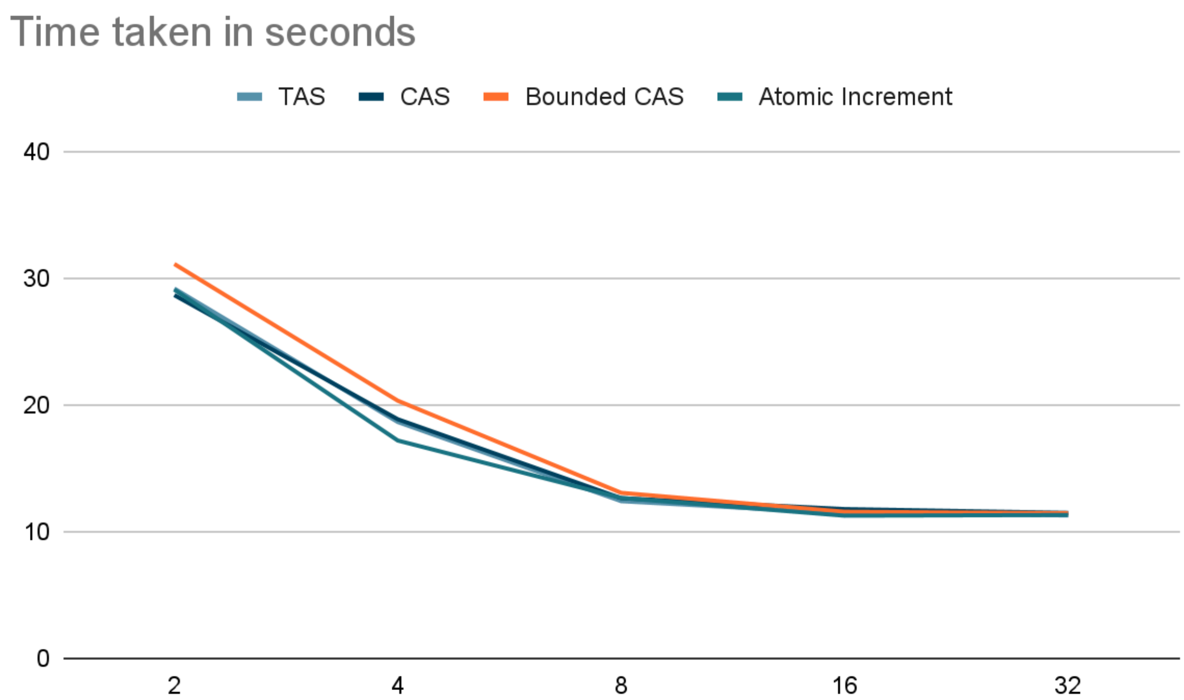
Experiment - 3 :

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). Please have N fixed at 2048 and rowInc at 16 for all these experiments.

Table3: K, time taken for TAS,CAS Bounded CAS ,Atomic Increment in seconds

K	TAS	CAS	Bounded CAS	Atomic Increment
2	29.1752	28.6597	31.1147	29.0931
4	18.6482	18.8658	20.329	17.1887
8	12.4042	12.6376	13.062	12.6582
16	11.449	11.7788	11.5695	11.2596

32	11.2835	11.482	11.4535	11.3179



Observations :

1. As the number of threads (K) increases, TAS, CAS, and Bounded CAS exhibit decreasing execution times, reflecting improved parallelism.
2. Atomic Increment consistently maintains relatively lower execution times across varying thread counts, highlighting its efficiency in handling concurrent operations.
3. While TAS, CAS, and Bounded CAS demonstrate similar performance trends, Atomic Increment stands out for its stable and comparatively lower execution times.

4. The reduction in execution times with increasing thread counts underscores the effectiveness of parallelization in optimizing computation-intensive tasks.
5. TAS, CAS, and Bounded CAS may encounter contention issues as the number of threads escalates, potentially impacting their scalability compared to Atomic Increment.
6. The choice between TAS, CAS, Bounded CAS, and Atomic Increment should consider factors like scalability, contention, and overhead to maximize performance in parallel computing scenarios

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 -

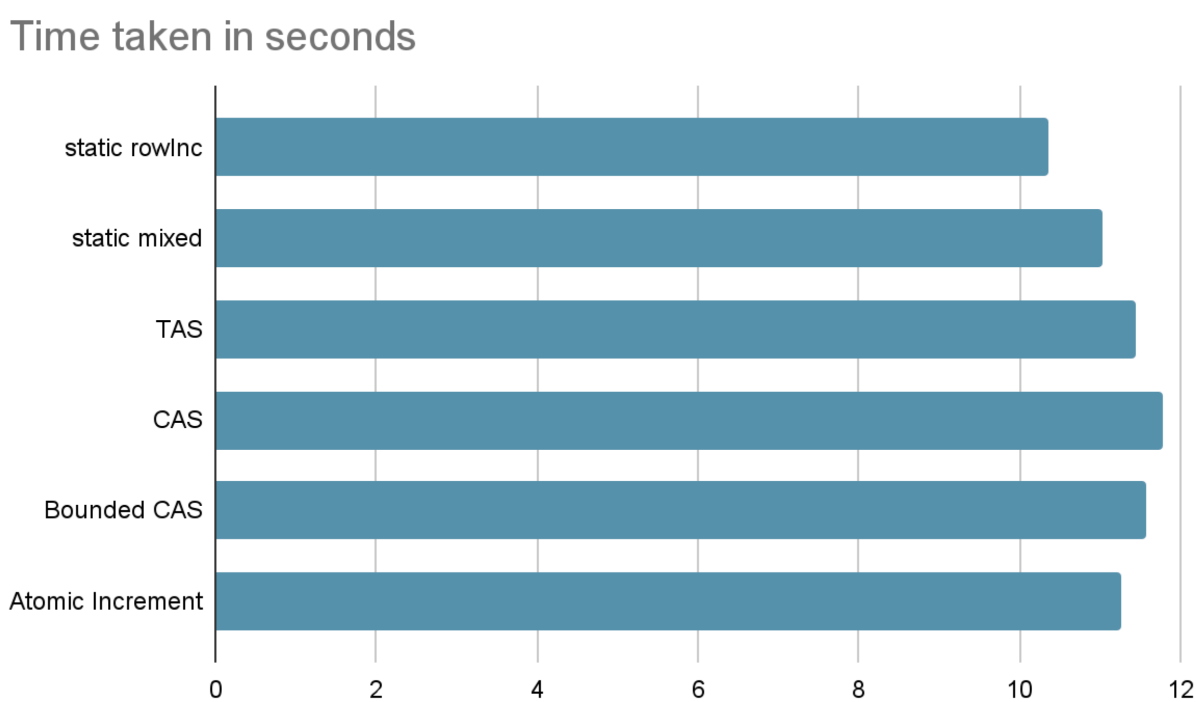
a) Static rowInc
b) Static mixed
c) Dynamic with TAS
d) Dynamic with CAS
e) Dynamic with Bounded CAS
f) Dynamic with Atomic.

N fixed at 2048, K fixed at 16, and rowInc at 16 for all these experiments. Please plot a bar graph for this experiment.

Table4: time taken for static rowInc,static mixed TAS,CAS, Bounded CAS ,Atomic Increment in seconds

Algorithm	Time taken in seconds

static rowInc	10.362196
static mixed	11.021407
TAS	11.449
CAS	11.7788
Bounded CAS	11.5695
Atomic Increment	11.2596



Observations :

1. The static row increment exhibits the fastest execution time at 10.362196 seconds, suggesting its efficiency in managing parallel operations.

2. The static mixed approach follows closely with a time of 11.021407 seconds, indicating its competitive performance in comparison to dynamic strategies like TAS, CAS, Bounded CAS, and Atomic Increment.
3. Among the dynamic approaches, TAS, CAS, and Bounded CAS exhibit relatively higher execution times, with CAS recording the longest time at 11.7788 seconds.
4. Bounded CAS presents a slightly lower execution time compared to CAS, indicating its improved performance in managing concurrent operations.
5. Atomic Increment emerges as the most efficient dynamic approach, boasting a relatively lower execution time of 11.2596 seconds compared to TAS, CAS, and Bounded CAS.
6. Overall, while dynamic approaches offer flexibility, the static row increment and mixed strategies demonstrate superior performance, emphasizing the significance of algorithmic design in optimizing parallel computation tasks.