Dynamic Multi-Threaded Matrix Multiplication using Mutual Exclusion: Report

Om Dave (CO22BTECH11006)

March 4, 2024

1 Introduction

This assignment explores parallel matrix multiplication using a dynamic allocation mechanism in C++. It focuses on dynamically distributing matrix rows among threads and addressing synchronization issues through various mutual exclusion algorithms: Test-and-Set (TAS), Compare-and-Swap (CAS), Bounded CAS, and atomic increments from the C++ atomic library. The goal is to enhance efficiency and understanding of parallel computing techniques in matrix operations.

2 Program Design and Implementation

2.1 Initial Setup

- **Input Parameters:** The program takes input parameters N, K, $rowInc$, and $N \times N$ matrix A. Here, K denotes the number of threads and $rowInc$ is the row increment value for dynamic allocation.
- **Thread Creation:** It creates K threads for parallel computation, in line with the input value of K.
- **Dynamic Row Allocation:** A shared counter C is used to dynamically allocate rows of matrix A to threads. Each thread increments C by $rowInc$ to claim the next set of rows for processing.
- **Synchronization Mechanism:** To manage access to C, different mutual exclusion algorithms are implemented: Test-and-Set (TAS), Compare-and-Swap (CAS), Bounded CAS, and atomic increments using the C++ atomic library.

3 Mutual Exclusion Alogrithms

3.1 Test and Set (TAS)

3.1.1 Overview

The Test-and-Set (TAS) algorithm operates by atomically checking and setting a flag variable, ensuring that only one thread can enter its critical section at a time. This approach is particularly useful in scenarios where threads compete for a shared resource, like in the case of dynamically allocating rows of a matrix to threads for parallel processing. TAS prevents race conditions by ensuring that critical sections of the code are executed by only one thread at a time.

3.1.2 Implementation

The provided code snippet demonstrates the implementation of the TAS mutual exclusion algorithm:

Listing 1: TAS Implementation

```
int C; // Shared counter to keep track of the number of threads that have
    ↪  completed their work
atomic_flag lock_ = ATOMIC_FLAG_INIT; // TAS lock

// Thread function to compute the certain rows (rowInc) of the square
    ↪ matrix
void* Compute_TAS(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;

    while(true) {

        if(C>n) break; //All rows of product matrix have been computed

        while(atomic_flag_test_and_set(&lock_)); //spin untill lock is
            ↪ acquired

        /* Critical Section */

        //Getting the start index and manually incrementing the shared
            ↪ counter C
        int start = C; //returns the previous value of C
        C+=rowInc; //Increment the counter
        int end = min(start + rowInc, n);
        cout << C << " " << start << " " << end << " Thread: " <<
            ↪ thread_id << "\n";

        //CRITICAL SECTION ENDS
        atomic_flag_clear(&lock_); //release the lock

        /* Remainder Section */
        for (int i = start; i < end; i++) {
            //computing row i of product matrix
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    prod[i][j] += A[i][k] * A[k][j];
                }
            }
        }

    }

    free(args);
    pthread_exit(NULL);
}
```

- **Atomic Flag Initialization:** The *lock_* is an atomic flag initialized to the unlocked state (*ATOMIC_FLAG_INIT*). It ensures that the TAS operation can be performed atomically.
- **Lock Acquisition:** Threads enter a spinlock loop, repeatedly calling *atomic_flag_test_and_set* on *lock_*. This atomic operation checks if the lock is already set (indicating the lock is taken) and sets it if not. The thread spins (continues to check) until it successfully acquires the lock.
- **Critical Section:** Once the lock is acquired, the thread computes the start and end indices for the rows it will process. It increments the shared counter C by $rowInc$, ensuring no other thread can modify C simultaneously. This section includes accessing and updating shared resources, which requires mutual exclusion.
- **Lock Release:** After updating C and determining the rows to compute, the thread releases the lock by clearing the atomic flag with *atomic_flag_clear*. This action allows other threads to acquire the lock and proceed with their computation.
- **Matrix Computation:** Outside the critical section, the thread computes the specified rows of the matrix product. This computation does not require mutual exclusion as each thread works on a distinct portion of the matrix.

3.2 Compare and Swap (CAS)

3.2.1 Overview

CAS works by atomically comparing the value of a memory location to a given value and, only if they match, modifying the memory location to a new given value. This atomicity guarantees that concurrent modifications are safely coordinated, making CAS particularly suitable for implementing lock-free data structures and algorithms where threads need to synchronize their access to shared resources without using traditional locking mechanisms.

3.2.2 Implementation

The provided code snippet demonstrates the implementation of the CAS mutual exclusion algorithm

Listing 2: CAS Implementation

```
int C; // Shared counter to keep track of the number of threads that have
    ↪ completed their work
atomic<bool> lock_(false); // CAS lock initialized to false

// Thread function to compute the certain rows (rowInc) of the square
    ↪ matrix
void* Compute_CAS(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;

    while(true) {

        if(C>n) break; //All rows of product matrix have been computed

        bool expected = false;
        while (!lock_.compare_exchange_strong(expected, true)) {
            expected = false; // Reset expected after failed exchange
        }

        /* Critical Section */

        //Getting the start index and manually incrementing the shared
            ↪ counter C
        int start = C; //returns the previous value of C
        C+=rowInc; //Increment the counter
        int end = min(start + rowInc, n);

        cout << C << " " << start << " " << end << " Thread: " <<
            ↪ thread_id << "\n";

        // CRITICAL SECTION ENDS
        lock_.store(false); // Release the lock
        /* Remainder Section */

        for (int i = start; i < end; i++) {
            //computing row i of product matrix
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    prod[i][j] += A[i][k] * A[k][j];
                }
            }
        }

    }

    free(args);
    pthread_exit(NULL);
}
```

- **Atomic Boolean Initialization:** The *lock_* is initialized to false, indicating that the lock is available.
- **Lock Acquisition:** To enter the critical section, a thread sets *expected* to false and attempts to swap *lock_* to true using *compare_exchange_strong*. If *lock_* is already true (indicating another thread holds the lock), *compare_exchange_strong* fails, *expected* is reset to false, and the thread retries until it successfully acquires the lock.
- **Critical Section:** Upon successfully acquiring the lock, the thread computes the start and end indices for the rows it will process by incrementing C by $rowInc$. This ensures exclusive access to modify C, thereby avoiding race conditions.
- **Lock Release:** After completing its updates, the thread releases the lock by setting *lock_* back to false, allowing other threads to enter the critical section.
- **Matrix Computation:** Outside the critical section, the thread computes the specified rows of the matrix product. This computation does not require mutual exclusion as each thread works on a distinct portion of the matrix.

3.3 Bounded Compare and Swap (BCAS)

3.3.1 Overview

The Bounded Compare-and-Swap (BCAS) mechanism enhances the basic Compare-and-Swap (CAS) algorithm by introducing a bounded waiting scheme to manage thread synchronization more efficiently. Unlike traditional CAS, which allows threads to compete freely for the lock, BCAS uses a bounded approach to limit the contention and potentially reduce the waiting time for threads. This is achieved by maintaining a waiting list (or vector) that tracks which threads are attempting to acquire the lock. Threads check this list in a cyclic manner, ensuring that after a thread completes its critical section, it hands off the lock to the next waiting thread, if any.

3.3.2 Implementation

The BCAS implementation is provided in the code below:

Listing 3: BCAS Implementation

```
int C; // Shared counter to keep track of the number of threads that have
   ↪  completed their work
atomic<bool> lock_(false); // BCAS lock initialized to false
vector<bool> waiting; // vector to keep track of threads waiting in
   ↪ Bounded CAS
waiting.resize(k,false);

// Thread function to compute the certain rows (rowInc) of the square
   ↪ matrix
void* Compute_BCAS(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;

    while (true) {

        if (C > n) break; // All rows of product matrix have been computed

        waiting[thread_id]=true; // Thread is waiting
        while(waiting[thread_id]) { //spin untill thread is not waiting
            bool expected = false;
            if(lock_.compare_exchange_strong(expected, true)) {
                break; //lock acquired, break out from this loop and enter
                   ↪  critical section
            }
        }
        waiting[thread_id]=false; // Current thread can enter critical
           ↪ section, it is not waiting any more

        /* Critical Section */

        //Getting the start index and manually incrementing the shared
           ↪ counter C
        int start = C; //returns the previous value of C
        C+=rowInc; //Increment the counter
        int end = min(start + rowInc, n);

        cout << C << " " << start << " " << end << " Thread: " <<
            ↪ thread_id << "\n";

        // CRITICAL SECTION ENDS

        //cyclically search for next waiting thread
        int j = (thread_id + 1) % k;
        while(j != thread_id && !waiting[j]) {
            j = (j + 1) % k;
        }

        if(j == thread_id) {
            lock_.store(false); //No waiting thread found, release the
                ↪ lock
        }
        else {
            waiting[j]=false; //Found a waiting thread, set it to false so
                ↪  it can entere critical section now
        }

        /* Remainder Section */

        for (int i = start; i < end; i++) {
            //computing row i of product matrix
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    prod[i][j] += A[i][k] * A[k][j];
                }
            }
        }

    }

    free(args);
    pthread_exit(NULL);
}
}
```

- **Initialization:** The *lock_* is an atomic boolean initialized to false, indicating the lock is available. The *waiting* vector is resized to k, where k is the number of threads, and initialized to false, indicating that initially, no threads are waiting.
- **Termination Check:** Before attempting to acquire the lock, each thread checks if all rows of the matrix have been processed by evaluating ‘if(C ¿= n) break;‘.

This condition ensures that threads cease their computation efforts once the entire matrix has been processed, effectively preventing unnecessary work.

- **Lock Acquisition:** Threads declare themselves as waiting by setting their respective *waiting*[*thread_id*] to true. They then attempt to acquire the lock through CAS. If successful, the thread exits the waiting state and proceeds to the critical section; otherwise, it continues to spin, checking its waiting status (If any other threads sets the its waiting = false, then it breaks out of the loop and enters critical section).
- **Critical Section:** Once in the critical section, a thread computes the rows of the matrix it is responsible for, based on the shared counter C, incrementing C by $rowInc$ for the next thread.
- **Lock Transfer or Release:** After completing its computation, the thread attempts to find the next waiting thread by cyclically checking the *waiting* vector. If no other thread is waiting, it releases the lock by setting *lock_* to false. If another waiting thread is found, the current thread enables it to proceed by setting its waiting status to false.

3.4 Atomic Increment

3.4.1 Overview

The Atomic Increment operation utilizes built-in C++ atomic operations for lock-free synchronization of shared variables. Atomic operations are indivisible; they complete in a single step relative to other threads. This means that no other thread can observe the operation at an intermediate state, and at the end of the operation, the change appears instantaneously.

3.4.2 Implementation

The provided code snippet illustrates the use of an atomic increment operation for parallel matrix multiplication.

Listing 4: CPP Atomic increment Implementation

```cpp
atomic<int> C(0); // Atomic Shared counter to keep track of the number of
    ↪ threads that have completed their work

// Thread function to compute the certain rows (rowInc) of the square
    ↪ matrix
void* Compute_ATOMIC(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;

    while(true) {

        if(C>n) break; //All rows of product matrix have been computed

        /* Critical Section */
        int start = C.fetch_add(rowInc); //Atomically returns the previous
            ↪  value of C counter and adds rowInc to it, basically gives
            ↪ the starting index for this to calculate
        int end = min(start + rowInc, n);
        cout << C << " " << start << " " << end << " Thread: " <<
            ↪ thread_id << "\n";

        /* Remainder Section */

        for (int i = start; i < end; i++) {
            //computing row i of product matrix
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    prod[i][j] += A[i][k] * A[k][j];
                }
            }
        }
    }

    free(args);
    pthread_exit(NULL);
}
```

- **Atomic Counter Initialization:** The atomic shared counter C is initialized to 0, ensuring that it starts from a known state before any thread begins computation.
- **Termination Check:** Each thread checks if all rows of the matrix have been processed by evaluating ‘if(C greather than equal to n) break;‘. This condition helps to terminate the thread’s execution once the entire matrix has been processed, optimizing resource use.
- **Critical Section:** The core of the atomic increment operation is in the critical section where $C.fetch\_add(rowInc)$ is called. This atomic operation performs two actions: it returns the current value of C for the thread to use as the starting index for its computation, and it atomically increments C by $rowInc$. This ensures that each thread works on a unique segment of the matrix without overlap.
- **Matrix Computation:** With the start and end indices determined, each thread computes its assigned rows of the matrix product. This computation is performed outside of any lock mechanism but is coordinated through the atomic increment of C, ensuring that each thread has a unique portion of the matrix to process.

4 Performance Analysis

4.1 Experiment 1: Time vs Size of Matrix (N)

4.1.1 Methodology

In this experiment, we evaluated the performance of four different mutual exclusion algorithms—Test and Set (TAS), Compare and Swap (CAS), Bounded Compare and Swap (BCAS), and Atomic—in a parallel matrix multiplication context. The matrix size (N) was varied in powers of two, starting from 256 × 256 to 2048 × 2048, to understand the algorithms' scalability with increasing workload. Both the number of threads (K) and the row increment (rowInc) were fixed at 16. It is important to note that the matrix size was capped at 2048 × 2048 due to computational constraints as larger matrices resulted in significantly longer computation times.

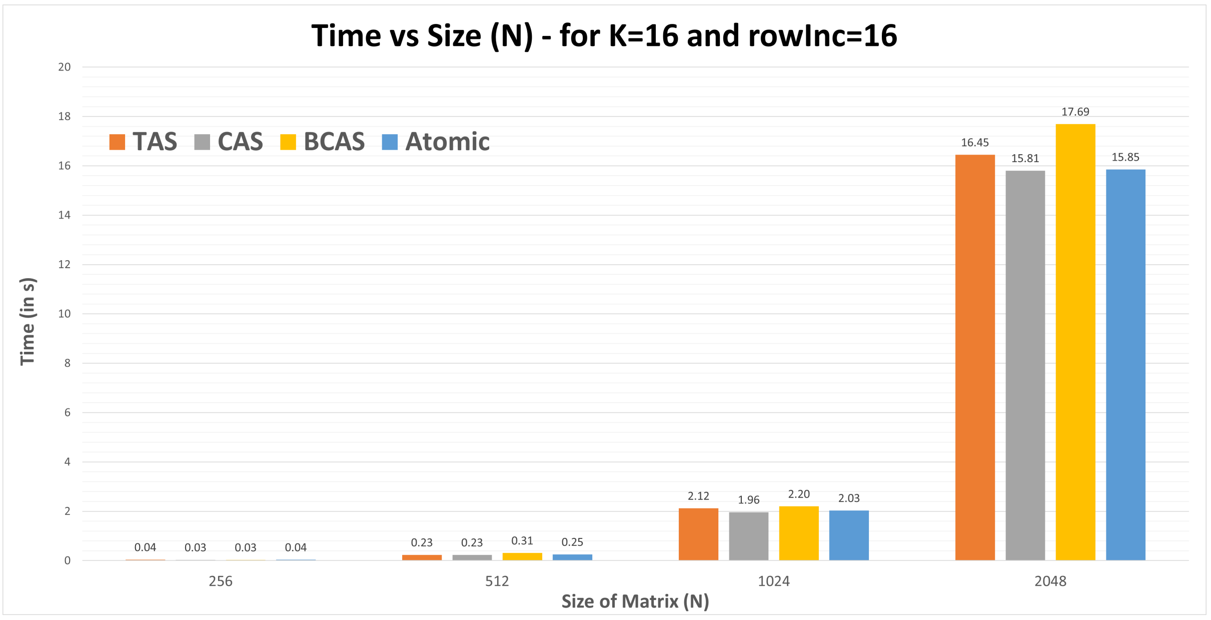
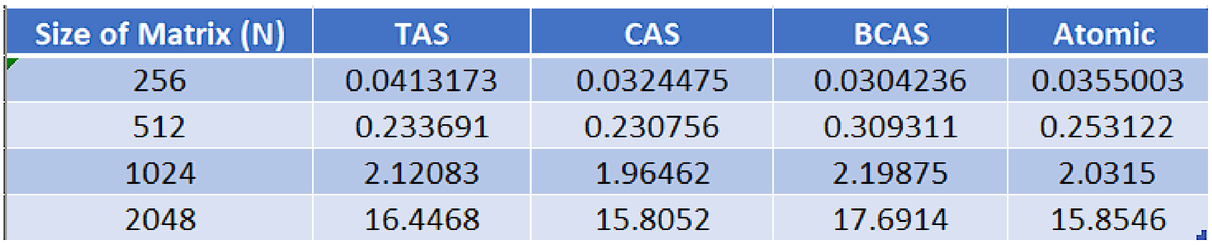


Figure 1: Total time vs Size of Matrix (N) for K=16 and rowInc=16



Size of Matrix (N)	TAS	CAS	BCAS	Atomic
256	0.0413173	0.0324475	0.0304236	0.0355003
512	0.233691	0.230756	0.309311	0.253122
1024	2.12083	1.96462	2.19875	2.0315
2048	16.4468	15.8052	17.6914	15.8546

Figure 2: Time vs Size of Matrix (N) for different synchronization algorithms

4.1.2 Observations

Analysis of the results indicates that all algorithms exhibit a polynomial time complexity (N^3), with time taken to compute the square of the matrix increasing significantly with the size of the matrix. The Atomic algorithm consistently outperformed TAS and BCAS across all matrix sizes, while CAS and Atomic algorithms showed close performance, especially with larger matrices.

4.1.3 Reasoning

The Atomic operations provided by modern CPUs are highly optimized for performance, which likely explains the efficiency of the Atomic algorithm. CAS, which also benefits from hardware-level optimization, similarly demonstrates efficient synchronization for concurrent operations. TAS, while straightforward, incurs more overhead due to spin-waiting, explaining its relatively slower performance. BCAS, designed to minimize waiting time, did not perform as expected, potentially due to the overhead associated with managing the bounded waiting mechanism. The performance trends observed are consistent with the theoretical complexity of matrix multiplication (N^3) and the overhead associated with each synchronization mechanism.

4.2 Experiment 2: Time vs Row Increment (rowInc)

4.2.1 Methodology

This experiment aimed to measure the impact of the row increment value (rowInc) on the time taken to compute the square of a matrix using different mutual exclusion methods. The row increment value represents the number of rows of the matrix that a thread claims and computes in each step, and it was varied from 1 to 32 in powers of 2. The size of the matrix (N) was set to 1024, and the number of threads (K) was fixed at 16. The initial plan was to set N to 2048, but due to extended computation times, it was adjusted to 1024.

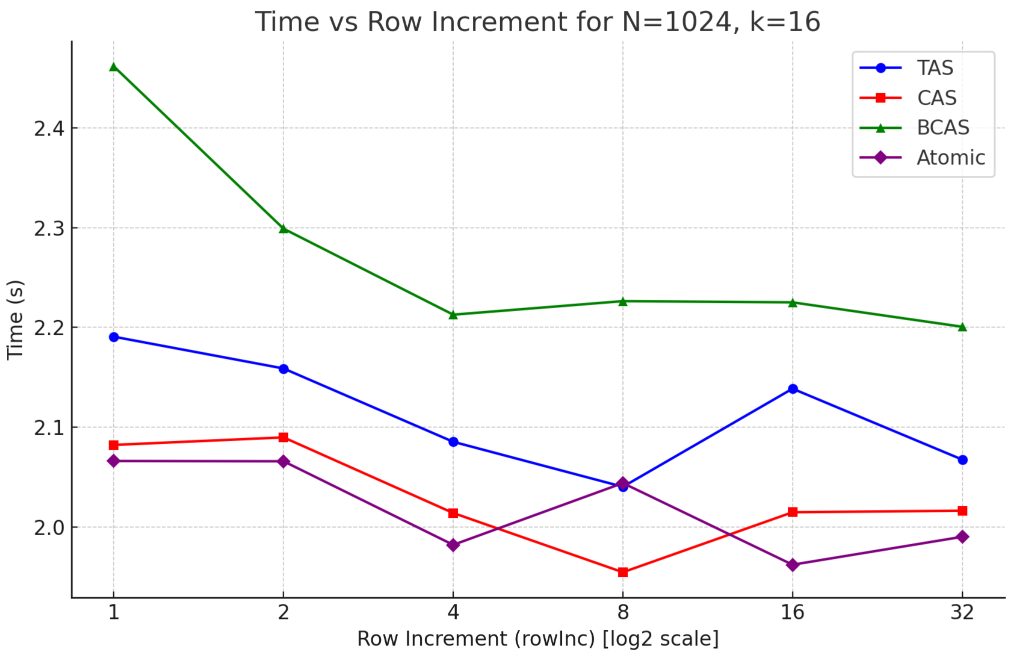
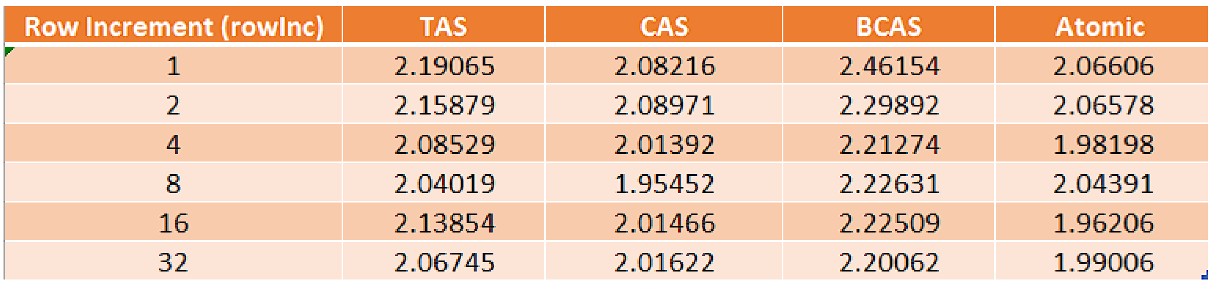


Figure 3: Time vs rowInc (row Increment) N=1024 k=16



Row Increment (rowInc)	TAS	CAS	BCAS	Atomic
1	2.19065	2.08216	2.46154	2.06606
2	2.15879	2.08971	2.29892	2.06578
4	2.08529	2.01392	2.21274	1.98198
8	2.04019	1.95452	2.22631	2.04391
16	2.13854	2.01466	2.22509	1.96206
32	2.06745	2.01622	2.20062	1.99006

Figure 4: Time vs rowInc(row Increment) for different algorithms

4.2.2 Observations

From the table and graph presented, it is observed that there is a general trend of decreasing computation time with increasing row increments across all synchronization methods. The Atomic synchronization method shows a consistent advantage in lower row increments, but exhibits a slight increase in computation time at the highest row increment of 32. TAS generally displays higher computation times initially but shows improvement as the row increment increases. BCAS starts as the least efficient method, but it improves with increasing rowInc. CAS shows similar trend like atomic with less times comparing to other methods.

4.2.3 Reasoning

The general trend of decreasing computation times as row increments increase can be attributed to several factors. Larger row increments mean that threads can perform more work before needing to synchronize with each other, effectively reducing the overhead associated with synchronization. This reduction in synchronization frequency allows threads to utilize CPU time more efficiently, leading to faster computation times. Additionally, larger workloads per thread can lead to better cache utilization, as more data is processed in sequence, which can also contribute to reduced computation times. The Atomic method benefits from hardware-level support for atomic operations, leading to faster execution times, especially with larger row increments where the overhead of locking is minimized. The improvement in TAS performance with larger row increments could be due to reduced contention, as threads claim more work per lock acquisition. CAS's performance is less affected by row increments due to its efficient lock-free synchronization. BCAS's initial poor performance may be due to the overhead of managing the bounded waiting scheme, which becomes less significant with larger chunks of work (higher rowInc values).

4.3 Experiment 3: Time vs Number of Threads (K)

4.3.1 Methodology

In the third experiment, our objective was to assess the performance impact of varying the number of threads (K) on the time taken to square a matrix. We incremented K from 2 to 32 in powers of 2, while keeping the matrix size (N) constant at 1024 and the row increment (rowInc) at 16. The intention was to fix N at 2048; however, due to longer computation times, N was set to 1024.

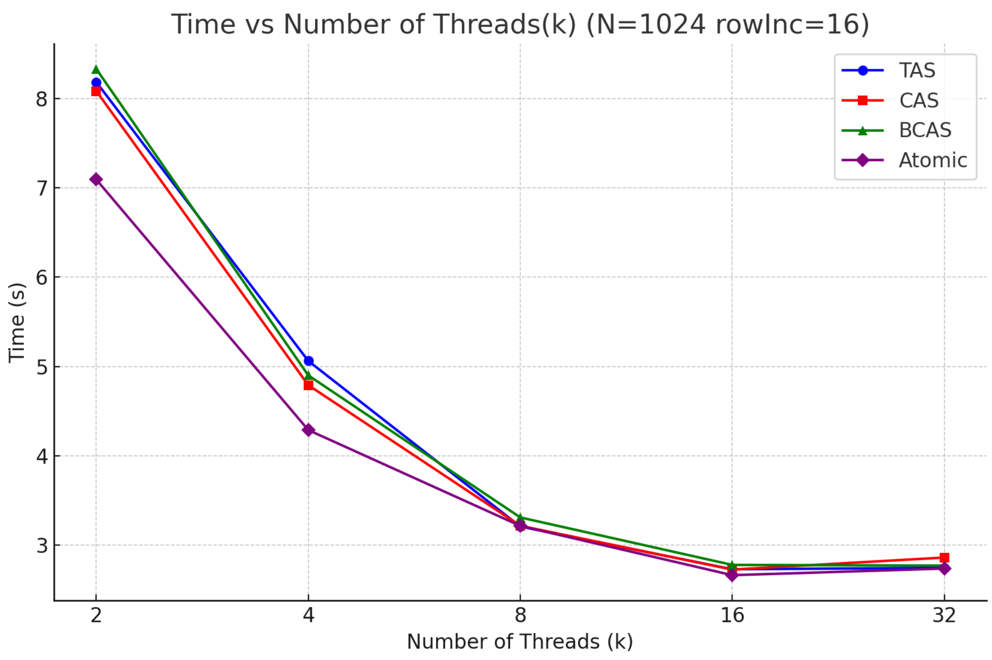
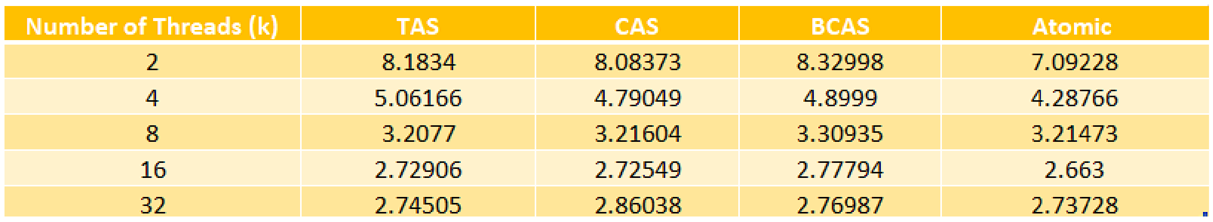


Figure 5: Time vs Number of Threads (K) N=1024 rowInc=16



Number of Threads (k)	TAS	CAS	BCAS	Atomic
2	8.1834	8.08373	8.32998	7.09228
4	5.06166	4.79049	4.8999	4.28766
8	3.2077	3.21604	3.30935	3.21473
16	2.72906	2.72549	2.77794	2.663
32	2.74505	2.86038	2.76987	2.73728

Figure 6: Time vs Number of Threads (K) for different algorithms

4.3.2 Observations

From the plotted data, it is observed that as the number of threads increases from 2 to 16, the time taken for matrix squaring decreases for all algorithms, indicating improved performance due to parallel execution. However, an increase in execution time is noted as the number of threads exceeds 16 and reaches 32. The Atomic algorithm shows the best performance consistently, followed closely by CAS. TAS and BCAS also improve with more threads, but their performance gain plateaus beyond 16 threads.

4.3.3 Reasoning

The decrease in computation time up to 16 threads can be attributed to the efficient parallelization of the matrix squaring task, where the workload is effectively distributed among the threads. However, since my system is a hexacore with hyperthreading support, allowing for 12 true parallel threads, the observed increase in computation time from 16 to 32 threads aligns with the system's hardware limitations. The additional threads do not correspond to additional cores and therefore do not contribute to parallelism. Instead, they introduce overhead associated with thread management and context switching, resulting in longer computation times. The Atomic operation, being hardware-supported, exhibits the most significant performance improvement, likely due to efficient lock-free synchronization that minimizes overhead. CAS also shows improved performance as it leverages compare-and-swap operations that tend to be more efficient than lock-based synchronization in multi-threaded environments. TAS experiences a more modest performance gain due to its simpler but more contention-prone mechanism. The BCAS algorithm, designed to limit the contention by introducing a bounded waiting scheme, does not show a significant performance advantage, which might be due to the overhead of managing the waiting queue, especially as the number of threads increases. This experiment underscores the need for a balanced approach to parallelization, where the number of threads is optimized according to the computational task and the underlying hardware architecture.

4.4 Experiment 4: Time vs Algorithms

4.4.1 Methodology

The fourth experiment compares the time efficiency of various synchronization algorithms in matrix squaring operations. The algorithms tested include two static allocation methods (Static rowInc and Static Mixed) and four dynamic synchronization methods (TAS, CAS, BCAS, Atomic). The size of the matrix (N) was fixed at 1024 (For lesser computation time), the number of threads (K) at 16, and the row increment (rowInc) also at 16.

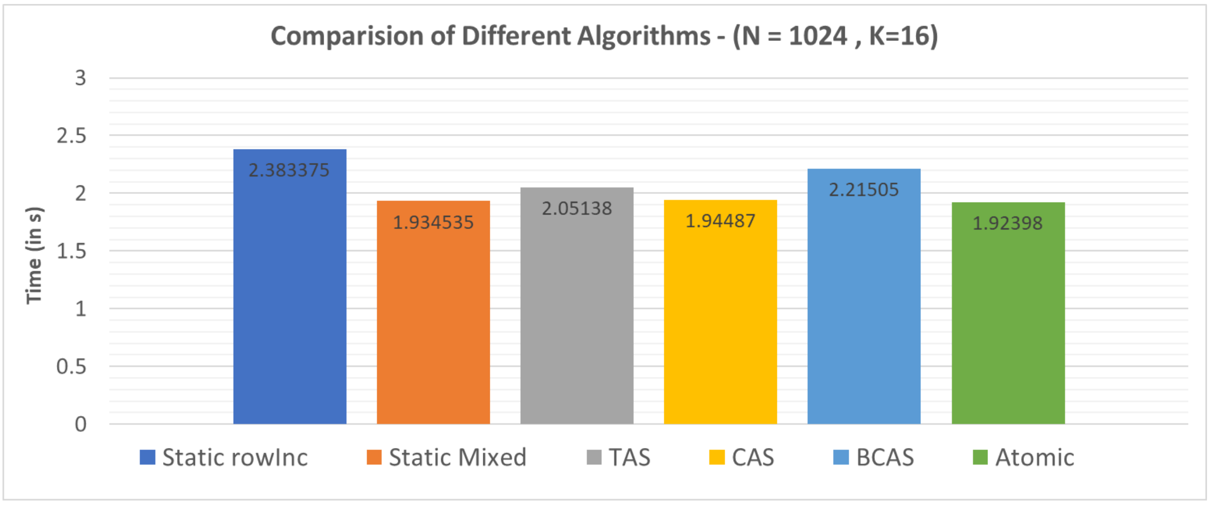
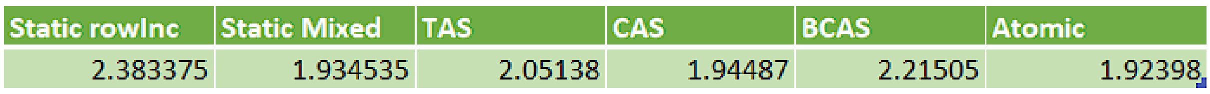


Figure 7: Time vs Algorithms (N=1024 K=16 rowInc=16)



Static rowInc	Static Mixed	TAS	CAS	BCAS	Atomic
2.383375	1.934535	2.05138	1.94487	2.21505	1.92398

Figure 8: Time vs Algorithms (N=1024 K=16 rowInc=16)

4.4.2 Observations

The bar graph illustrates that the Static Mixed allocation strategy yielded the best performance, closely followed by the Atomic synchronization method. The Static rowInc method was the least efficient. Dynamic synchronization methods show varying performance, with TAS and CAS being more competitive than BCAS.

4.4.3 Reasoning

The Static Mixed strategy's superior performance can be attributed to the reduced contention it creates by evenly distributing the work among the threads. This minimizes the synchronization overhead and maximizes the use of the CPU cache by following a pattern conducive to spatial locality.

The Atomic method's efficiency comes from its hardware-level support for atomic operations, which ensures minimal overhead for synchronization. TAS and CAS both benefit from a dynamic workload distribution, with CAS slightly outperforming TAS due to its more efficient handling of contention without the need for locking.

BCAS's slightly lower performance compared to TAS and CAS suggests that the bounded waiting mechanism introduces some overhead, which may not be fully compensated for by the reduced contention it provides.

The performance dip in the Static rowInc method might be due to the cyclic distribution of work, which could lead to scenarious where some threads might have to bear heavy work load than other thus increasing overall time.