# OS ASSIGNMENT 3

Suvedh Deva

Roll No:CS22BTECH11016

## INTRODUTION:

The objective of the code is to calculate square of given matrix efficiently by parallelizing the task using threads but the portion of matrix the threads would be working on is allocated dynamically now. There are four code files, one for each method used for mutual exclusion of threads. The structure of main function in these files is almost same, just the function the threads would execute is different

## Code Explanations and Low-Level Designs

### 1. TAS Algorithm (`tas_worker_function.cpp`)

**Low-Level Design:**

The TAS (Test-and-Set) algorithm is implemented using a simple atomic flag. The worker function (`tas_worker_function`) utilizes the `std::atomic_flag` to implement mutual exclusion. The atomic flag is set and cleared to ensure exclusive access during matrix multiplication.

**Code Explanation:**

```
// Code for tas_worker_function.cpp

#include <atomic>


void tas_worker_function(int local_rowInc) {
    for (int i = 0; i < local_rowInc; ++i) {
        // Acquire lock using TAS
        while (tas_flag.test_and_set(std::memory_order_acquire)) {
            // Wait for lock to be released
        }

        // Perform matrix multiplication
        multiply_matrix();

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

### Explanation:

tas-flag is an instance of std::atomic-flag used as a lock for mutual exclusion. test-and-set is an atomic operation that atomically sets the flag and returns its previous value. The loop ensures that the thread waits if the flag is already set (indicating another thread holds the lock). The lock is released using clear when the matrix multiplication is completed.

## 2. CAS Algorithm (`cas_worker_function.cpp`)

**Low-Level Design:**

The CAS (Compare-and-Swap) algorithm is implemented using `std::atomic` operations. The worker function (`cas_worker_function`) uses `std::atomic<int>` as a counter, and CAS is employed to ensure exclusive access during matrix multiplication.

**Code Explanation:**

```
// Code for cas_worker_function.cpp

#include <atomic>


void cas_worker_function(int local_rowInc) {
    for (int i = 0; i < local_rowInc; ++i) {
        // Acquire lock using CAS
        int expected = 0;
        while (!cas_counter.compare_exchange_weak(expected, 1,
                                                std::memory_order_acquire,
                                                std::memory_order_relaxed)) {
            // Wait for lock to be released
            expected = 0;
        }

        // Perform matrix multiplication
        multiply_matrix();

        // Release lock
        cas_counter.store(0, std::memory_order_release);
    }
}
```

## Explanation:

cas-counter is an instance of std::atomic¡int¿ used as a counter and lock. compare-exchange-weak is a CAS operation that attempts to atomically compare and swap the counter's value. The loop ensures that the thread waits if the counter is already set (indicating another thread holds the lock). The lock is released using store when the matrix multiplication is completed.

## 3. Bounded CAS Algorithm (`bounded_cas_worker_function.cpp`)

**Low-Level Design:**

The Bounded CAS algorithm is implemented using additional variables for controlling access. The worker function (`bounded_cas_worker_function`) utilizes `std::atomic` variables and a waiting protocol to achieve bounded CAS.

**Code Explanation:**

```
// Code for bounded_cas_worker_function.cpp

#include <atomic>


void bounded_cas_worker_function(int i, int local_rowInc) {
    for (int j = 0; j < local_rowInc; ++j) {
        // Acquire lock using Bounded CAS
        waiting[i] = true;
```

```
        j = bounded_cas_lock.exchange(i, std::memory_order_acquire);

        // Perform matrix multiplication
        multiply_matrix();

        // Release lock
        waiting[i] = false;
        bounded_cas_lock.store(-1, std::memory_order_release);
    }
}
```

## Explanation:

waiting is an array of std::atomic¡bool¿ used to signal if a thread is waiting for the lock. bounded-cas-counter is an instance of std::atomic¡int¿ used as a counter and lock. The thread sets its waiting flag, then uses exchange to atomically set the counter to 1 and retrieve its previous value. If the previous value was non-zero, the thread waits, otherwise, it continues with matrix multiplication. The lock is released using store when the matrix multiplication is completed.

### 4. Atomic Algorithm (`atomic_worker_function.cpp`)

**Low-Level Design:**

The Atomic algorithm is implemented using `std::atomic` operations for counter increments. The worker function (`atomic_worker_function`) uses `std::atomic<int>` to ensure atomic increments and avoid race conditions.

**Code Explanation:**

```
// Code for atomic_worker_function.cpp

#include <atomic>


void atomic_worker_function(int local_rowInc) {
    for (int i = 0; i < local_rowInc; ++i) {
        // Acquire lock using Atomic operation
        int current_counter = counter.fetch_add(1, std::memory_order_relaxed);

        // Perform matrix multiplication
        multiply_matrix();

        // Release lock
        // (No explicit release needed as fetch_add ensures atomic increment)
    }
}
```

## Explanation:

atomic-lock is an instance of std::atomic-flag used as a lock for mutual exclusion. atomic-flag-test-and-set-explicit is an atomic operation that sets the flag and returns its previous value. The loop ensures that the thread waits if the flag is already set (indicating another thread holds the lock). The lock is released using atomic-flag-clear-explicit when the matrix multiplication is completed.

## Experiment 1: Time (s) vs Size of input (N)

### Observations:

The execution time for the TAS, CAS, Bounded CAS, and Atomic methods increases as the size of the input matrix (N) grows. TAS and CAS exhibit similar performance, while Bounded CAS and Atomic methods show better scalability with larger matrix sizes.

Table 1: Time (s) vs Size of input (N)

| **Size of input (N)** | **Time (s)** | | | |
|---|---|---|---|---|
| | **TAS** | **CAS** | **Bounded CAS** | **Atomic** |
| 256 | 0.028971 | 0.030064 | 0.027137 | 0.032248 |
| 512 | 0.235332 | 0.234574 | 0.210878 | 0.218715 |
| 1024 | 1.896776 | 1.838367 | 1.883301 | 1.742939 |
| 2048 | 17.484722 | 17.5543 | 17.413124 | 14.373580 |
| 4096 | 113.448983 | 114.401097 | 112.264892 | 113.137393 |

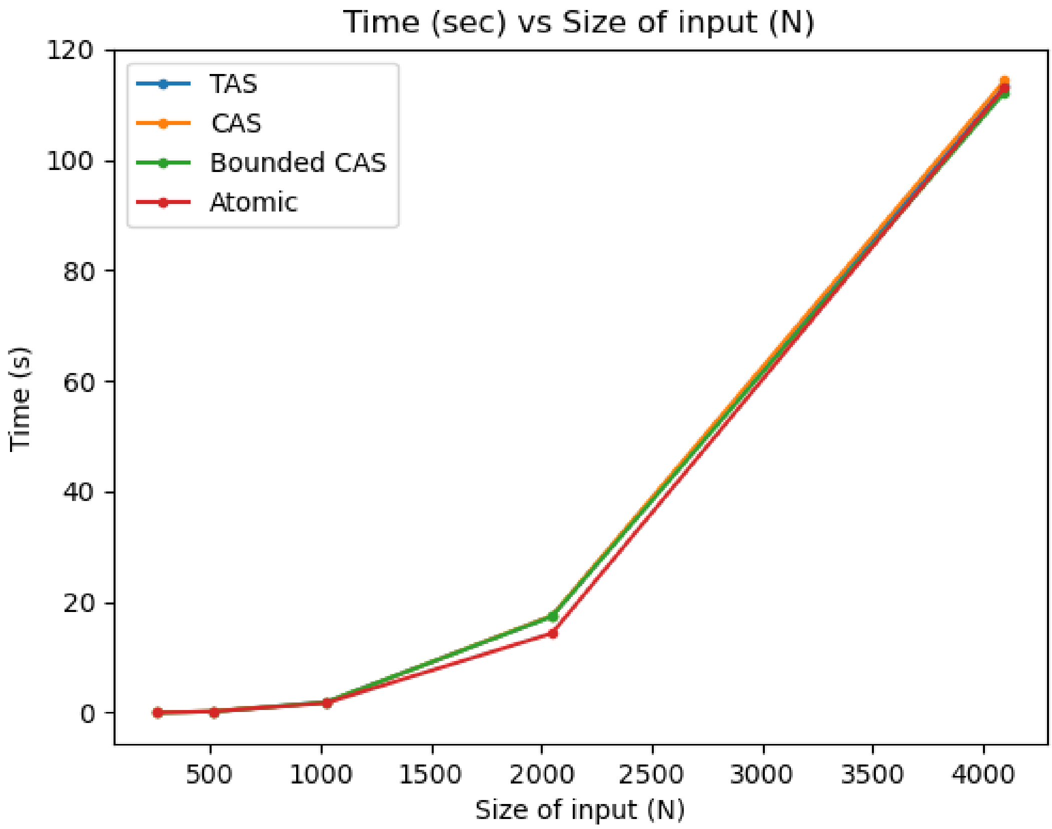


Figure 1: Time vs Size of input (N)

## Experiment 2: Time (s) vs rowInc

### Observations:

The impact of changing the row increment (rowInc) on the execution time is evident. The TAS and CAS methods show relatively consistent performance, while Bounded CAS and Atomic methods demonstrate sensitivity to rowInc changes.

Table 2: Time (s) vs rowInc

| **rowInc** | **Time (s)** | | | |
|---|---|---|---|---|
| | **TAS** | **CAS** | **Bounded CAS** | **Atomic** |
| 1 | 13.903124 | 13.266910 | 13.390052 | 13.063435 |
| 2 | 13.036897 | 13.186222 | 13.023201 | 13.095582 |
| 4 | 13.568577 | 13.053321 | 13.122132 | 13.140845 |
| 8 | 13.104747 | 13.067507 | 13.179161 | 13.117014 |
| 16 | 13.213482 | 13.248782 | 13.265411 | 13.266193 |
| 32 | 13.354862 | 13.081951 | 13.178456 | 13.144958 |

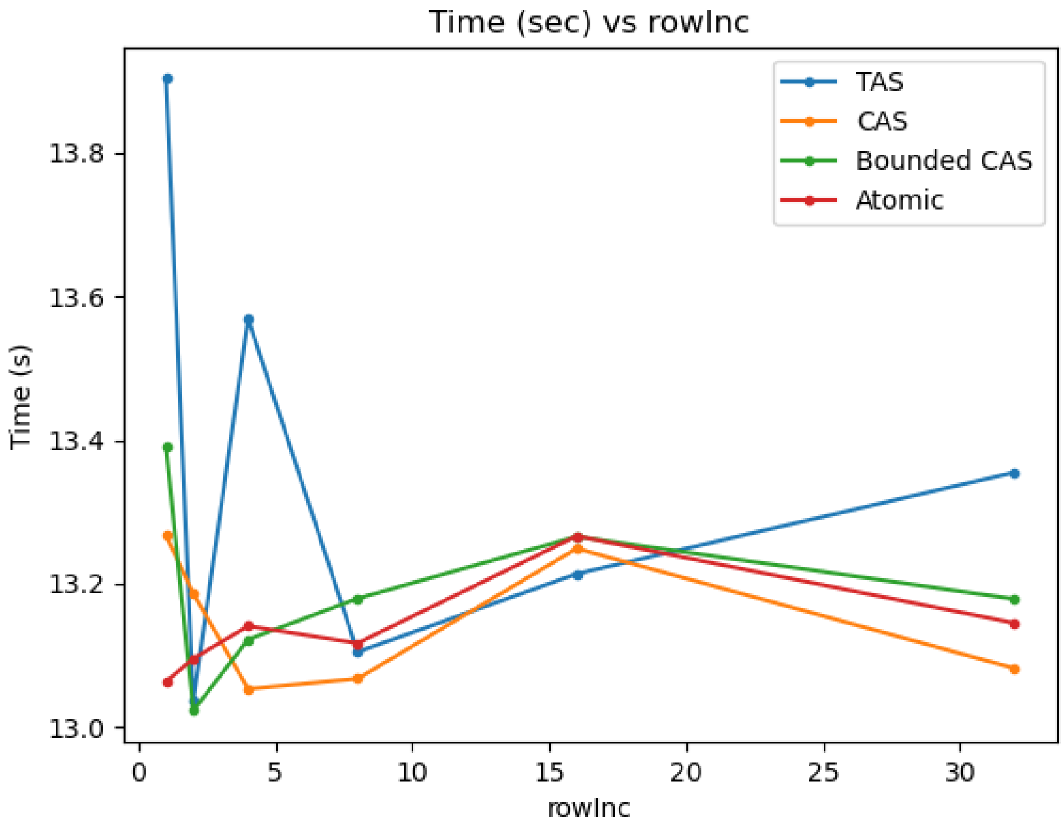


Figure 2: Time vs rowInc

## Experiment 3: Time (s) vs Number of Threads (K)

### Observations:

The impact of the number of threads (K) on execution time is apparent. TAS, CAS, and Bounded CAS show improvements with more threads, while Atomic method exhibits consistent performance.

Table 3: Time (s) vs Number of Threads (K)

| **Number of Threads (K)** | **Time (s)** | | | |
|---|---|---|---|---|
| | **TAS** | **CAS** | **Bounded CAS** | **Atomic** |
| 2 | 42.783392 | 41.381823 | 41.746440 | 42.836923 |
| 4 | 22.656044 | 22.910142 | 23.051607 | 22.189255 |
| 8 | 14.295684 | 14.366868 | 14.207872 | 14.673512 |
| 16 | 13.278628 | 13.236062 | 13.565804 | 13.605325 |
| 32 | 13.221209 | 13.611026 | 13.193409 | 13.245979 |

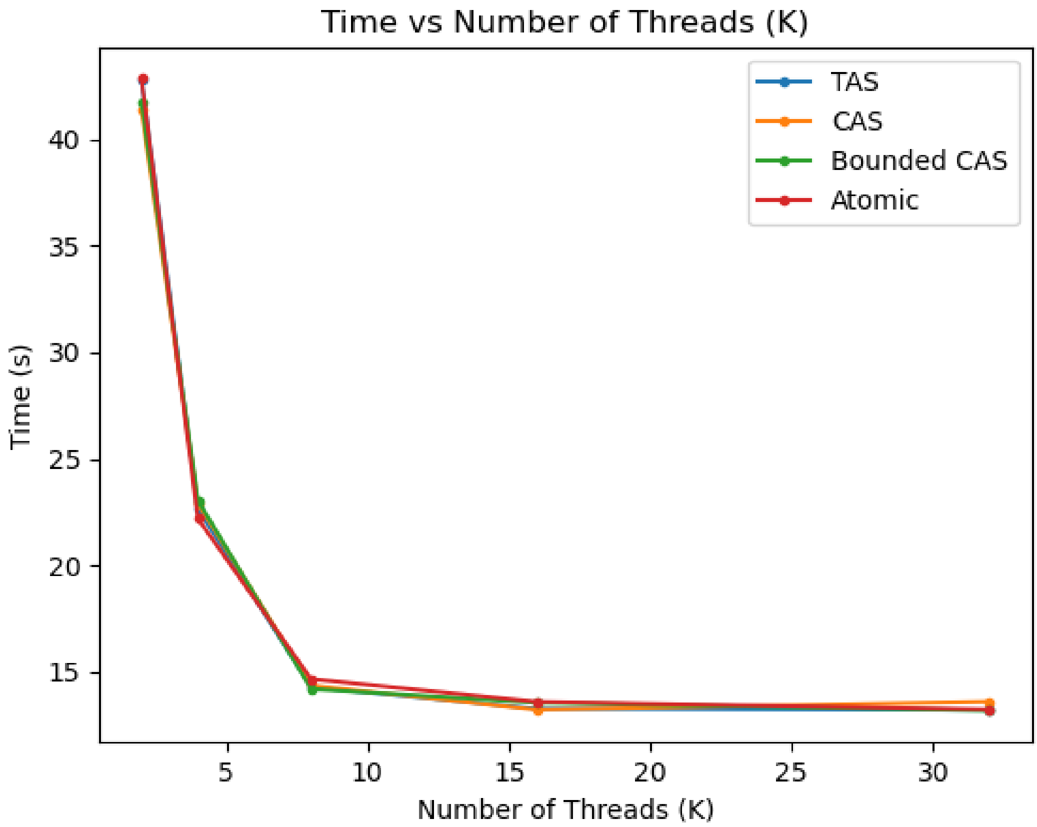


Figure 3: Time vs Number of Threads (K)

### Experiment 4: Time vs Algorithm

#### Observations:

The execution time varies across different parallel matrix multiplication algorithms. The CHUNK and MIXED methods exhibit similar performance, while TAS and CAS methods show slightly higher execution times. Bounded CAS and Atomic methods demonstrate better efficiency.

Table 4: Time vs Algorithm

| Algorithm | Time |
|---|---|
| CHUNK | 17.498166 |
| MIXED | 16.094472 |
| TAS | 17.484722 |
| CAS | 17.5543 |
| BCAS | 17.413124 |
| ATOMIC INC | 14.373580 |

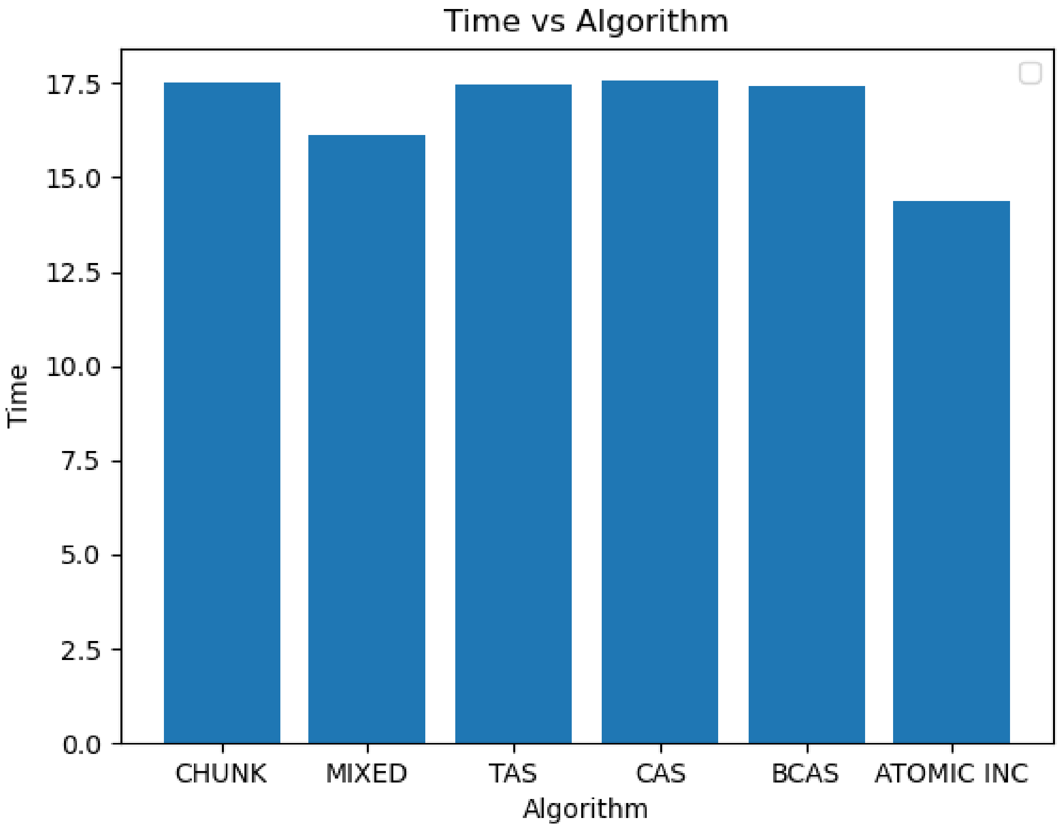


Figure 4: Time vs Algorithm