CS3523: Programming Assignment-3

Manohar Maripe
CS22BTECH11036

March 4, 2024

1 Introduction

This report provides an analysis of a C++ program for performing matrix multiplication using multithreading, aiming to leverage parallel processing capabilities for improved computational efficiency. The main objective of this code is to demonstrate the parallelization of matrix multiplication for improved performance. Matrix multiplication is a computationally intensive task that can benefit significantly from parallelization.

This report discusses the implementation of a parallel matrix multiplication program in C++ with a dynamic mechanism for thread allocation. Various mutual exclusion algorithms are employed to synchronize thread access to shared resources (thread coordination). Synchronization between threads is achieved using different mutual exclusion algorithms: Test-and-Set (TAS), Compare-and-Swap (CAS), Bounded CAS, and atomic increment provided by the C++ atomic library. The performance of the program is evaluated through experiments measuring execution time under different conditions.

2 Code Overview

The C++ implementation for multithreaded matrix multiplication comprises several key components that collectively contribute to its functionality. The code is designed to efficiently multiply two matrices using a multithreading approach, by using different algorithms in which rows are assigned to threads dynamically. Here is a detailed overview of the major components:

2.1 Header Inclusions

- `<fstream>`: Provides facilities for file input and output.
- `<thread>`: Provides classes and functions for working with threads.
- `<ctime>`: Defines various functions to manipulate date and time information.
- `<sys/time.h>`: Defines structures for time-related operations.
- `<pthread.h>`: POSIX threads library for multithreading support.
- `<sched.h>`: Provides functions for setting scheduling parameters for processes and threads.
- `<atomic>`: Atomic operations for thread synchronization.

2.2 File Input

The code begins by reading input matrices from a file named `inp.txt`. This part of the implementation ensures flexibility by allowing users to provide matrices of varying sizes for multiplication and Row Increment value . The file format is assumed to contain the matrix dimensions (n, k, rowInc) followed by the matrix elements.

2.3 Main Function

The `main` function is the entry point of the program. It reads input parameters from the file `"inp.txt"`, initializes matrices, measures execution time, performs matrix multiplication using threads, and writes the result to the output file `"out_[algorithm].txt"` according to the algorithm implemented.

2.4 Matrix Multiplication Function

This function takes two matrix references (`matrix` and `finalmatrix`), along with their dimensions (`n`), an index for thread-specific computation, and the number of columns (`k`). It iterates over rows of the matrix, computing partial products based on the provided index, and accumulates the results in the final output matrix (`finalmatrix`).

2.5 Multithreading Implementation

To exploit parallelism, the code utilizes the C++ `std::thread` library. Threads are created based on the user-defined parameter k, which represents the number of threads to be employed. Each thread executes the `multiplymulti` function, targeting a distinct subset of rows in the input matrices. This multithreading approach aims to maximize CPU utilization and accelerate the overall matrix multiplication process.

2.6 Thread Synchronization:

Here we implement four different algorithms i.e.(Test and set, Compare and swap, Bounded Compare and Swap Atomic), these algorithms are employed to ensure proper synchronization among threads. This is achieved using atomic operations and a shared variable `locker`. The basic idea is that no two threads can access the shared variable among the threads simultaneously. We can achieve using atomic variables and certain functions.

2.7 Time Calculation

The program utilizes the gettimeofday function to calculate the time taken for matrix multiplication.After all threads complete their execution, the total time taken by the program is calculated by taking the difference between the start time and end time obtained using gettimeofday.

2.8 File Output

The final multiplication result is stored in a matrix named `finalmatrix`, which is then written to an output file named `out_[algorithm].txt`. The resulting matrix is presented in a format suitable for easy interpretation and further analysis. Additionally, the execution time is recorded and appended to the output file for performance evaluation.

3 Multithreading Implementation

The multithreading implementation in the code leverages the capabilities of the C++ `std::thread` library to parallelize the matrix multiplication process. The key objective is to distribute the workload among multiple threads, thereby potentially reducing the overall execution time.

In this implementation:

- A vector of threads, `threads`, is created to store instances of the `std::thread` class.
- A loop is used to create threads, and a lambda function is passed as an argument to each thread. The lambda function captures the necessary variables by reference and calls the `multiplymulti` function with the appropriate parameters.
- The threads are then stored in the `threads` vector.
- Another loop is used to join all the threads. The `join()` function is called on each thread to ensure the main program waits for all threads to complete before proceeding further.

This multithreading approach divides the matrix multiplication workload into smaller chunks assigned to different threads. The concurrent execution of these threads can lead to improved performance, especially on systems with multiple processing cores. The subsequent sections of this report will delve into the results obtained from this multithreading approach, providing insights into the efficiency gains achieved and potential considerations for further optimization.

3.1 Chunk

The matrix A is divided into chunks of size p (rowIter). Where any of the threads can `thread i` can compute subset of p rows in A where subsets are divided in 1 to p , p+1 to 2*p and soo on.

4 Thread Synchronization Techniques

Every thread synchronization technique allows only one thread to enter the critical session simultaneously.

4.1 Test and Set (TAS)

- In Test and Set, a thread attempts to set a lock variable to a specific value (usually 1) and returns the previous value of the lock variable.
- If the previous value was 0, it means the lock was successfully acquired; otherwise, it was already held by another thread.
- The basic algorithm can be described in pseudo-code as follows:

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}

while test_and_set(lock) == 1:
    // Wait or perform some other action if the lock is already held
    // Critical section
    release(lock);
```

4.2 Compare and Swap (CAS)

- Compare and Swap is similar to Test and Set but provides more flexibility by allowing a thread to compare the current value of a variable with an expected value before performing the swap.
- If the current value matches the expected value, the swap is performed, usually updating the variable to a new value.
- A global variable (lock) is declared and is initialized to 0. The first process that invokes compare and swap() will set lock to 1. It will then enter its critical section, because the original value of lock was equal to the expected value of 0. Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0. When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.
- The basic algorithm can be described in pseudo-code as follows:

```
int compare_and_swap(int *value, int expected, int new value) {
    int temp = *value;
    if (*value == expected)
        *value = new value;
    return temp;
}

while (true) {
while (compare_and_swap(&lock, 0, 1) != 0);  /* do nothing */
    /* critical section */
    lock = 0;
/* remainder section */
}
```

The above two techniques are typically implemented using atomic hardware instructions to ensure that the operation is performed atomically without interference from other threads.

4.3 Bounded Compare and Swap (BCAS)

Bounded Compare and Swap (BCAS) is a synchronization technique used to ensure mutual exclusion, progress, and bounded waiting in concurrent systems. It is based on the Compare and Swap (CAS) operation, which is an atomic instruction commonly provided by modern processors.

1. **Initialization**: BCAS typically involves initializing some data structures, such as a boolean array `waiting` to keep track of whether processes are waiting to enter their critical sections, and a boolean variable `lock` to indicate whether a process has acquired the lock.
2. **Entering Critical Section**:
   - When a process wants to enter its critical section, it sets its corresponding `waiting` flag to true.
   - It then attempts to acquire the lock using the CAS operation. If the lock is currently false (indicating it's available), the CAS operation will atomically set the lock to true and allow the process to proceed into the critical section. If another process manages to acquire the lock first, the current process will loop and try again.
   - Once inside the critical section, the process performs its critical operations.
3. **Exiting Critical Section**:
   - After completing its critical section, the process updates the `waiting` array and releases the lock.
   - It scans the `waiting` array in a cyclic order to find the next process waiting to enter its critical section. This ensures fairness among waiting processes.
   - If no waiting process is found, the lock is released, allowing other processes to acquire it. Otherwise, the process designates the next waiting process to enter its critical section by setting its `waiting` flag to false.

4.4 Atomic Algorithm

- Atomic algorithms encompass are the techniques that perform operations indivisibly, ensuring they are not interrupted by other threads.
- Atomic operations are indivisible, meaning they are executed entirely or not at all. This ensures that no other thread can observe an intermediate state of the operation.
- Increment operation on atomic variable is atomic.
- By guaranteeing indivisibility, atomic operations eliminate the need for explicit locking mechanisms, simplifying thread synchronization and reducing the risk of competing conditions.
- Consider a scenario where multiple threads are updating a shared counter. Instead of using locks or other synchronization primitives, an atomic increment operation can be employed to ensure that the counter is incremented atomically, without interference from other threads.
- The basic algorithm can be described in pseudo-code as follows:

```
// Initialize a shared atomic variable
atomic_int counter = 0;

// Function to increment the counter atomically
    counter.fetch_add(&counter, 1);

// Matrix Multiplication
```

5 Graph Analysis

5.1 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 8192 (size of the matrix will vary as 256*256, 512*512, 1024*1024, ...) in the power of 2.
- We Fix the rowInc at 16 and K at 16 for all these algorithms.

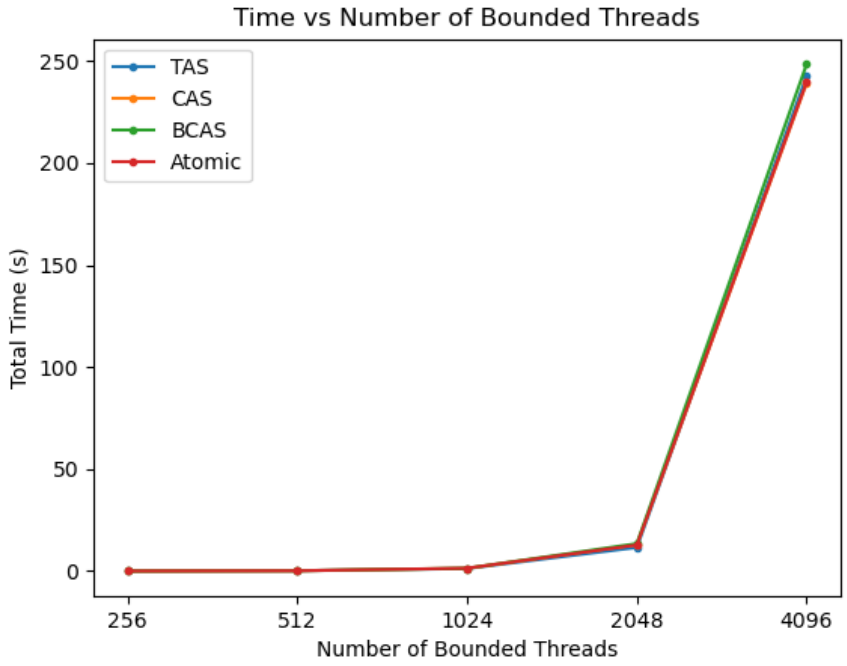

Figure 1: Time vs. Size(N)

N	TAS	CAS	B_CAS	Atomic
256	0.021868	0.022826	0.021221	0.022322
512	0.164252	0.152031	0.162464	0.148295
1024	1.24832	1.31997	1.34189	1.39131
2048	11.5556	12.9381	13.4655	12.7699
4096	242.612	239.303	248.434	239.612

Table 1: Total Time with varying N in seconds.

Analysis:

- Generally, larger matrix sizes result in longer execution times due to increased computational complexity.
- On increasing size, The number of row multiplications will increase and size of each row also increases so, the time taken will also increase gradually as we need to compute more number of rows. This is irrespective of algorithm.
- All the algorithms have almost same execution time.

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

- The y-axis will show the time taken to compute the square matrix in this graph.
- The x-axis will be the rowInc varying from 1 to 32 (in powers of 2, i.e., 1,2,4,8,16,32).
- We fix N at 2048 and K at 16 for all these algorithms.

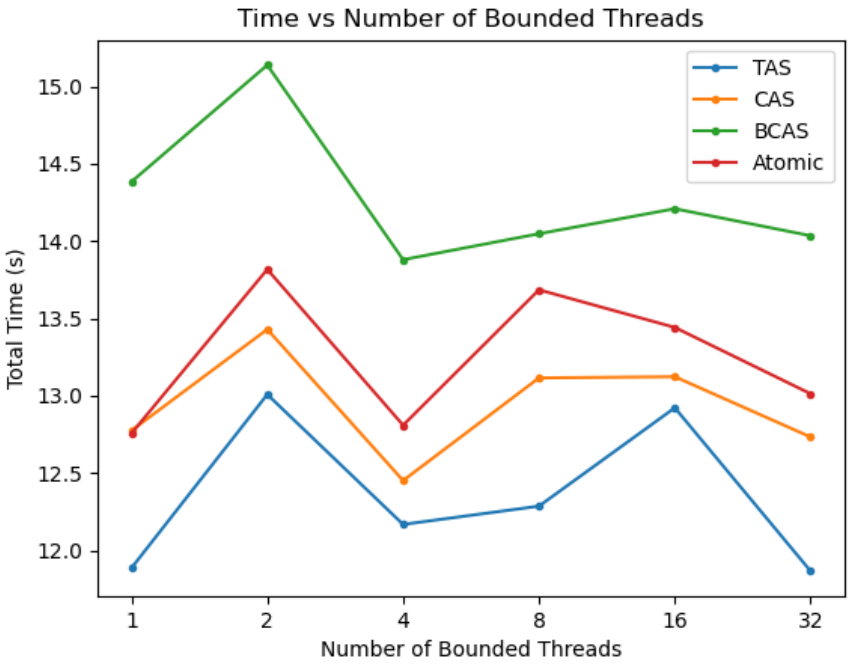

Figure 2: Time vs. rowInc.

Row Inc	TAS	CAS	B_CAS	Atomic
1	11.885	12.7742	14.384	12.7561
2	13.0079	13.4291	15.139	13.8172
4	12.167	12.4508	13.8799	12.8089
8	12.2852	13.115	14.0477	13.6849
16	12.9216	13.1226	14.2096	13.4416
32	11.8646	12.7325	14.0348	13.0122

Table 2: Total Time with varying rowIter in seconds

Analysis:

- Here in this case there is some inconsistency for all the algorithms. On observing the performance the trend, it is found out to be irregular (random).
- Performance between algorithms is also irregular.

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

- The y-axis will show the time taken to compute the square matrix in this graph.
- 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).
- We fix N at 2048 and rowInc at 16 for all these experiments.

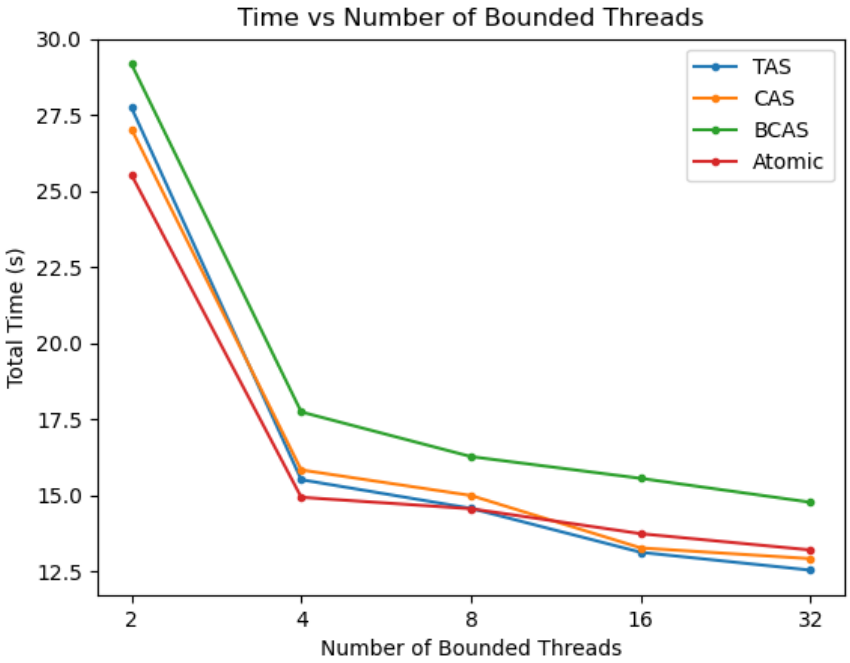

Figure 3: Time vs. Number of Threads (K).

No of Threads	TAS	CAS	B_CAS	Atomic
2	27.7422	27.0408	29.1753	25.5345
4	15.5107	15.8305	17.7372	14.9327
8	14.5673	14.9908	16.2692	14.5556
16	13.1226	13.2667	15.5539	13.7346
32	12.5360	12.9134	14.7677	13.1994

Table 3: Total Time with varying number of threads in seconds

Analysis:

- On increasing number of threads, The number of row computed parallelly will increase so, the time taken will decrease as we can compute more rows parallely. This is irrespective of algorithm.
- However, beyond a certain threshold, adding more threads may result in diminishing returns or even increased overhead due to contention for shared resources.
- A deviation from expected behavior is noted: the performance of BCAS algorithms appears to slightly low (time taken is more) with smaller values of K. This occured phenomenon might be due to occurrence of thread starvation issues for unbounded technique is comparitively low. Consequently, the implementation of bounded waiting might introduce unnecessary overhead in such scenarios.

5.4 Experiment 4: Time vs. Algorithms:

- The y-axis will show the time taken to compute the square matrix in this graph.
- 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.

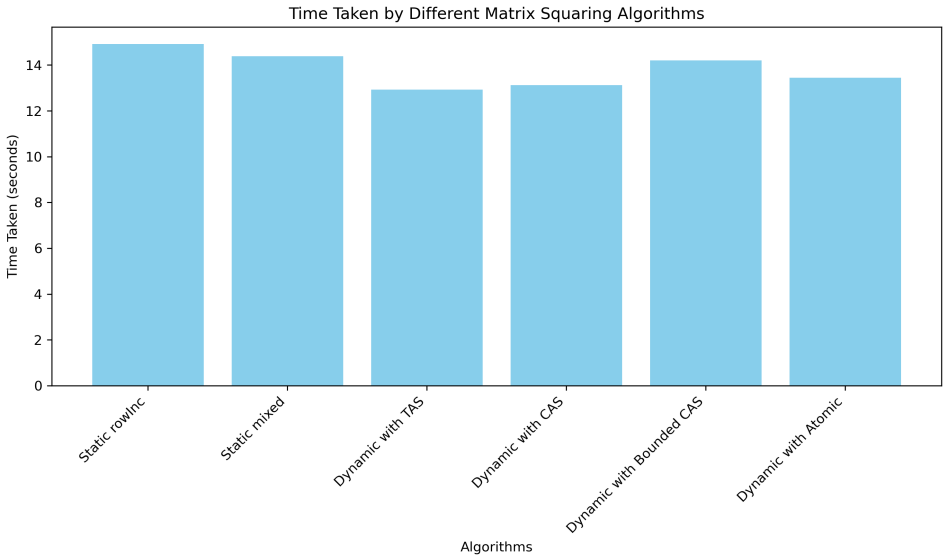

Figure 4: Time vs. Algorithms.

Algorithm	Time taken
Static rowInc	14.9153
Static Mixed	14.3917
Dynamic with TAS	13.1226
Dynamic with CAS	14.2096
Dynamic with B_CAS	13.4416
Dynamic with Atomic	12.9216

Table 4: Total Time for different algorithms in seconds.

Analysis:

- We can see that, Dynamic algorithms are performing better than Static algorithms. This is due to dynamic allocation of work to threads.
- There is some inconsistency in diving work in static allocation between threads as the work might not be equal among threads, some threads may complete early and some threads lags.
- In Static allocation after completing the work assigned to a thread for first time, the thread doesnot compete with other threads to work again.
- This is overcomed by dynamic allocation, in dynamic allocation even though after completing the work assigned for the thread the thread again competes with other threads so performance increases.

6 Conclusion

- In this experiment, we investigated the performance of dynamically allocating work among threads in parallel matrix squaring using different mutual exclusion algorithms, where each thread claimed a set of rows of the matrix to compute the square.
- Overall, the dynamic implementation of work division among threads, coupled with suitable mutual exclusion algorithms, demonstrated improved scalability and efficiency in parallel matrix squaring. The choice of algorithm significantly impacted performance, with more sophisticated techniques such as CAS and atomic increment outperforming simpler alternatives like TAS.