CS3523: Programming Assignment-3

Kartikeya Mandapati
CS22BTECH11032

March 1, 2024

1 Introduction

to perform parallel matrix multiplication through a Dynamic mechanism in C++.

2 Code Overview

This program is written in C++ and follows modular structure. It is divided into four separate files one for each of TAS, CAS, Bounded CAS, atomic increment. However all the codes follow a common structure.

2.1 Main Program

The main program will read N, K, rowInc, and the matrix A from an input file ”input.txt”. It then calls the respective functions to perform each of their tasks. The code flow is as follows:

1. Read the input values N , K , $rowInc$ and the matrix A from `inp.txt` file.
2. Initialize the result matrix and other variables.
3. Creates K threads using a for loop, each starts to run from the chunkMethod( ) function.
4. Based on the respective lock type, the threads wait till they can increment the common variable and then claim the respective set of rows.
5. Wait for all the threads to finish .
6. Print the output matrix and time taken into `chunkOut.txt` using the `printOutput()` function.

The code utilizes the following functions along with functions `pthreadcreate()`, `pthreadsetaffinitynp()` and `pthreadjoin()` provided by the `pthread` and `atomic` library to create and join threads:

1. `chunkMethod()` function to perform multiplication using chunking method.
2. `joinThreads()` function to join the threads.
3. `printOutput()` function to print the resultant matrix and time taken into the out file.

All the multiplication methods same way for multiplying, using a three for loops, one for iterating over the row, one for iterating the column and the other over the column to multiply the row with column.

2.2 Chunking Method

1. **Main Loop**: The function runs in a loop until all rows of the matrix have been evaluated. This loop is controlled by the commonV variable, which tracks the next row to be evaluated.
2. **Critical Section**: Within the loop, a critical section is implemented. This critical section determines the rows to be evaluated by assigning the current value of commonV to the rowNumber variable, then increments commonV by the rowInc.

3. **Matrix Multiplication**: After determining the rows to be evaluated, each thread performs matrix multiplication on its assigned chunk of rows. The thread iterates over the rows within its chunk (start to end) and multiplies each element of the resulting square matrix (square) by performing the appropriate dot product of corresponding rows and columns of the input matrix (mat).

4. **Remainder Section**: After completing the matrix multiplication for its chunk, the thread checks if all rows have been evaluated by comparing commonV with the total number of rows N. If all rows have been evaluated, the thread exits the loop and terminates. Otherwise, it continues to the next iteration of the loop to process the remaining rows. Finally, once all rows have been evaluated, the thread exits by calling pthreadexit(0), indicating successful completion.

2.3 Test And Swap (TAS)

1. **Initialization**: The `atomic_flag` variable `lock` is initialized with `ATOMIC_FLAG_INIT`, which sets it to clear state (unlocked).

2. **Test and Set Technique**: The critical section is guarded by a loop that repeatedly attempts to set the atomic flag using the `test_and_set` method. This method atomically tests the flag and sets it to the set state (locked) if it was previously clear. It returns the previous value of the flag, allowing the thread to determine if it successfully acquired the lock.

3. **Critical Section**: Within the loop, the thread enters the critical section after successfully acquiring the lock. In this section, it determines the rows to be evaluated by assigning the current value of `commonV` to the `rowNumber` variable, then increments `commonV` by the chunk size `P`. This critical section is protected from concurrent access by the test and set mechanism.

4. **Clearing the Lock**: After completing the critical section, the thread clears the lock by calling the `clear` method on the `atomic_flag`. This action releases the lock, allowing other threads to acquire it.

5. **Remainder Section**: The remainder of the code (matrix multiplication part) is executed outside the critical section.

6. **Loop Termination**: The loop continues until all rows have been evaluated (i.e., `commonV` is greater than or equal to `N`), at which point the loop terminates.

7. **Thread Exit**: Once all rows have been evaluated, the thread exits by calling `pthread_exit(0)`, indicating successful completion.

```
atomic_flag lock = ATOMIC_FLAG_INIT;
do{
    int rowNumber=0;
    while(lock.test_and_set());
    /* CRITICAL SECTION*/

    /* CRITICAL SECTION END*/
    lock.clear();
    /*REMAINDER SECTION*/
    if(commonV>=N) break;
} while (commonV<N);
```

2.4 Compare And Swap (CAS)

1. **Loop Until All Rows are Evaluated**: The function begins with a while loop that continues until all rows are evaluated. This loop is controlled by the `commonV` variable, which tracks the progress of row evaluation.

2. **Atomic Operation with Compare and Swap**: Within the loop, the function employs the `__sync_val_compare_and_swap` function to atomically perform a compare and swap operation on the `lock` variable. This operation attempts to set the value of `lock` to 1 only if its current value is 0. The loop continues until the CAS operation successfully acquires the lock (i.e., sets `lock` to 1).

3. **Critical Section**: Once the lock is acquired, the thread enters the critical section. Inside the critical section, the thread determines the rows to be evaluated by assigning the current value of `commonV` to the `rowNumber` variable and then increments `commonV` by the chunk size `P`.
4. **Remainder Section**: After completing the critical section, the thread proceeds to the remainder section of the code, which performs the matrix multiplication task. This section operates outside the critical section and can be executed concurrently by multiple threads.
5. **Loop Termination**: The loop continues until all rows have been evaluated (i.e., `commonV` is greater than or equal to `N`). If all rows are evaluated, the loop breaks, and the thread proceeds to exit.
6. **Thread Exit**: Finally, the thread exits by calling `pthread_exit(0)`, indicating successful completion of its task.

```
while(!__sync_val_compare_and_swap(&lock,0,1));
/* CRITICAL SECTION*/

/* CRITICAL SECTION END*/
lock=0;

/*REMAINDER SECTION*/
}
//checking if all the rows are evaluated
if(commonV>=N) break;
}
```

2.5 Bounded Compare And Swap (BCAS)

- **Lock Acquisition** Each thread sets its `waiting` flag to `true` and attempts to acquire the lock using BCAS. If the lock is already acquired by another thread (`lock == 1`), the thread waits until the lock becomes available (`lock == 0`). This ensures that only one thread can enter the critical section at a time.
- **Critical Section**: Once a thread acquires the lock, it sets its `waiting` flag to `false` and enters the critical section to perform its task (e.g., matrix multiplication).
- **Scheduling Next Thread**: After completing its critical section, the thread is responsible for scheduling the next thread to execute. The next thread to execute is determined by iterating through the `waiting` flags of all threads. If the next waiting thread is the current thread itself, it releases the lock (`lock = 0`) to allow other threads to acquire it. If the next waiting thread is not the current thread, the lock is released to that next waiting thread to allow it to enter its critical section.
- **Loop Termination**: The thread continues executing until all rows are evaluated (`commonV >= N`).
- **Thread Exit**: Once all rows are evaluated, the thread exits.

```
do{
    waiting[t] = true;
    int lock = 1;
    //waiting for the lock to be free
    while (waiting[t] && lock == 1)
        lock = __sync_val_compare_and_swap(&lock, 0, 1);
    //if the lock is free, then the thread enters the critical section
    waiting[t] = false;
    /* CRITICAL SECTION*/

    /* CRITICAL SECTION END*/

    //finding the next thread to be executed
```

```
        int ntid = (t+1)%K;
        while((ntid!=t) && !waiting[ntid])
            ntid = (ntid+1)%K;
        if(ntid==t){
            lock=0;
        } else {
            waiting[ntid]=false;
        }
        if(commonV>=N) break;
    } while(common<V<N)
```

2.6 Atomic Increment

- **Initialization**: The function begins with a while loop that continues indefinitely (`while(true)`).
- **Critical Section**: Inside the loop, the thread enters the critical section where it updates the `commonV` variable atomically.
  - The variable `commonV` represents the current row number being processed.
  - Within the critical section, the thread reads the current value of `commonV` and assigns it to the `rowNumber` variable.
  - The thread then increments `commonV` by the chunk size `P`, ensuring atomicity of the operation to prevent race conditions.
- **Remainder Section**: After updating `commonV`, the thread proceeds to the remainder section where it performs matrix multiplication.
- **Loop Termination**: The loop continues indefinitely until all rows are evaluated (`commonV >= N`).

```
atomic<int> commonV{0};
do{
    /* CRITICAL SECTION*/

    /* CRITICAL SECTION END*/
    /*REMAINDER SECTION*/

    if(commonV>=N) break;
}while(commonV<N)
```

2.7 Printing the Output

The printOutput function is passed the the output file name and the result matrix and the time elapsed. It creates or opens the output file and prints the resultant matrix along which the time in it.

2.8 Joining the threads

We pass in the vector or array of the threads and it iterates over the threads using a for loop and uses pthreadjoin() to join them.

2.9 Calculating Time

To calculate the time taken by bounded and normal Threads , I used two vectors that store the time taken by each thread respectively for bounded and normal threads. In the corresponding chunk or mixed function, once the thread execution completes, it updates its time in the vector.
At the end, the times are averaged using the vector. For unbounded threads, the time is calculated same as in Assignment-1.

2.10 Complications

- **Joining threads** we shall wait till the threads are joined else the result would be incomplete.
- **Inconsistent Data** the data doesn't stay consistent as it depends on various aspects like number of current processes running in the system, and also varies drastically due to battery charge as cores get suspended.
- Variation of performance from system to system.
- Synchronization issues leading to exceptions when common variables are being modified.

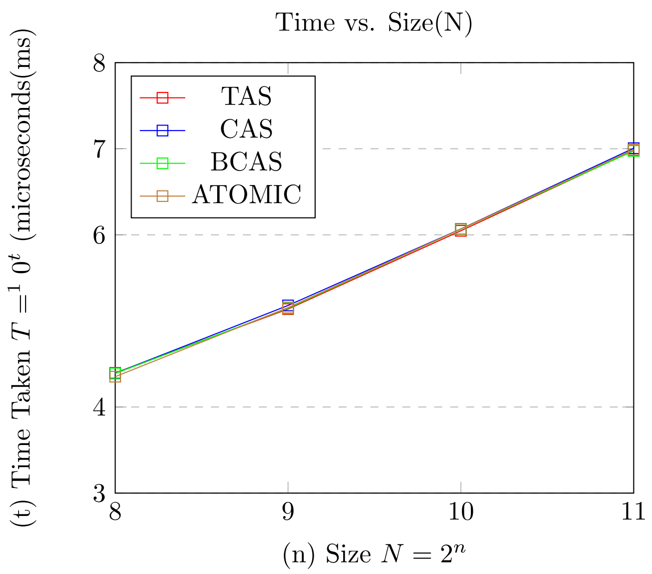
3 Experiment-1

3.1 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. rowInc=16 and K=16 is fixed for all values of this experiment. The graph contains four curves:

Size(N)	TAS	CAS	BCAS	ATOMIC
256	0.024944	0.024621	0.024525	0.022526
512	0.137114	0.151179	0.140444	0.142544
1024	1.115222	1.171398	1.164376	1.169454
2048	9.524440	10.148207	9.442876	9.772656

Table 1: Time taken (seconds (s)) with varying Size (N)



3.2 Analysis

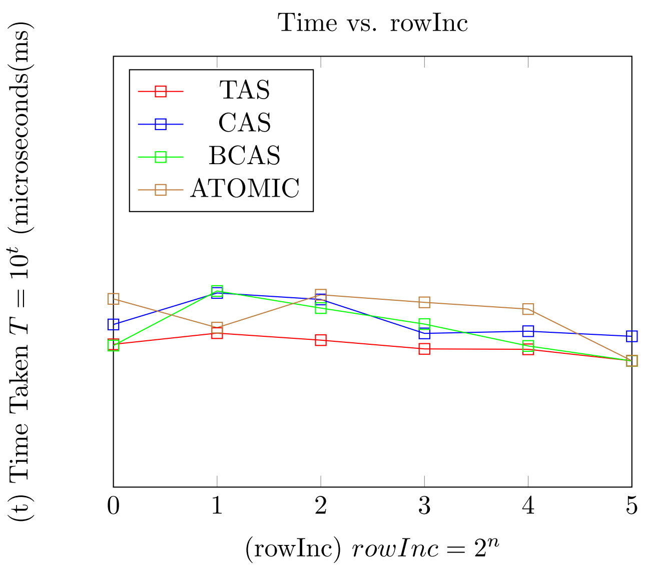
Clearly as the size (N) increases, the work to be done increases. Since we fixed the number of threads. The time taken increases exponentially. Also as we don't find much difference between the algorithms as the increase in N dominates the benefits offered by the algorithm.

3.3 Time vs. rowInc (row Increment):

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). N=2048 and K=16 is fixed for all values of this experiment.

rowInc	TAS	CAS	BCAS	ATOMIC
1	9.977872	10.622953	9.961542	11.543323
2	10.098882	11.735301	11.825551	10.548976
4	11.880926	11.515385	11.210244	11.670883
8	9.814329	10.348495	10.660145	11.435825
16	9.813056	10.415440	9.933842	11.128657
32	9.460341	10.233162	9.818353	9.457286

Table 2: Time taken (seconds (s)) with varying rowInc value



3.4 Analysis

Initially the time taken increases but later as we increase the rowInc value, but later as we increase rowInc value, we are increasing the chunk size, which is reducing the repeated execution of threads one after the other, resulting in reduced waiting , ultimately better efficiency.

3.5 Time VS Number of Threads plot(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). N=2048 and rowInc=16 is fixed for all values of this experiment.

Number of Threads(K)	TAS	CAS	BCAS	ATOMIC
1	87.055031	85.513570	78.290194	79.086042
2	38.085705	38.833772	37.362654	37.671668
4	20.669828	21.077207	21.079813	23.745618
8	15.111723	18.389898	15.550128	16.091192
16	11.662050	12.655745	12.021014	11.943737
32	9.512086	10.084137	9.127446	9.261386

Table 3: Time taken (seconds (s)) with varying K value

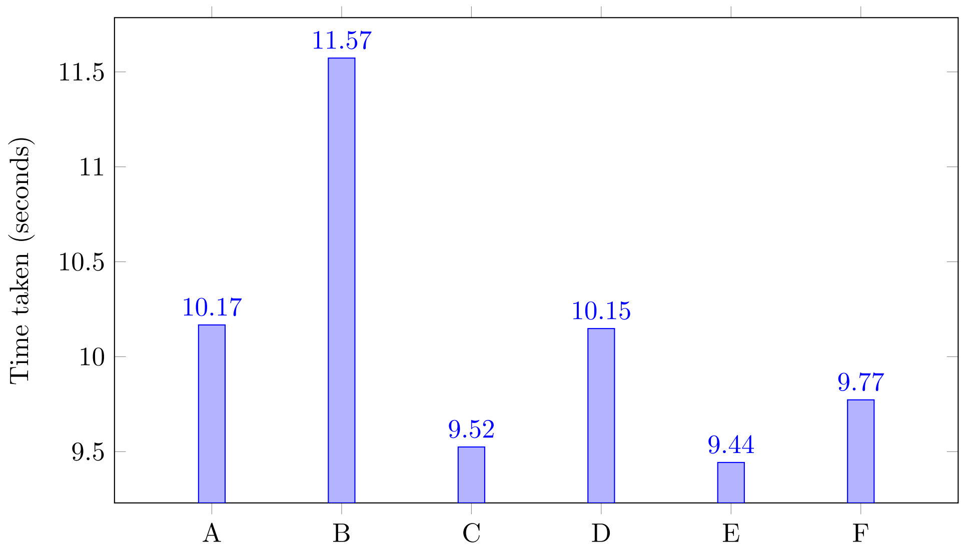
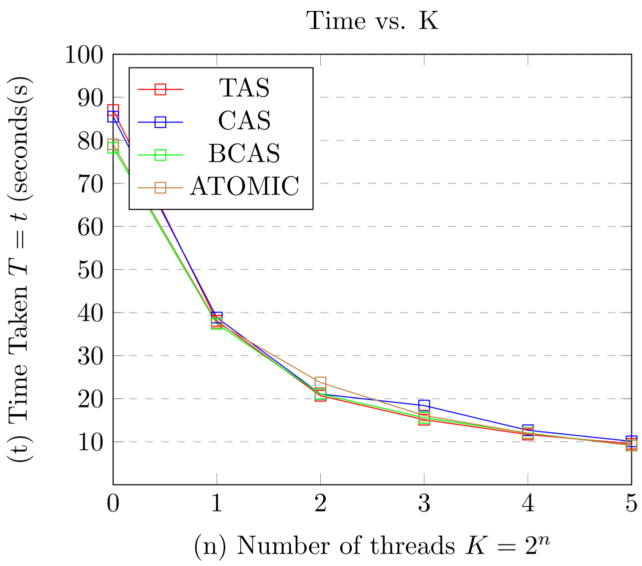


Figure 1: Time taken for different operations



3.6 Analysis

As we increase the number of threads, we are using parallelism in much better way. So as we increase the number of threads , work is getting divided and resulting in less time of execution.

3.7 Time vs Algorithm

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.

3.8 Analysis:

Dynamic appears to perform slightly better than static. But the difference is very small. With dynamic as we are allocating the chunks dynamically, threads that have completed faster take up the left over work reducing the time, whereas in case of static, as we fix the the work, the threads that have completed faster do not offer any benefit. The difference could be seen more clearly if the work being divided is not uniform, in this case if the matrix values are not distributed uniformly.
The instructions to compile and run the program are given in the README file.