Operating System -2 Programming Assignment 3

Name : Ahmik Virani
Roll Number : ES22BTECH11001

Low Level Design :

The entire code is divided into 5 parts :

1. The include header files :

```
#include<iostream>
#include<fstream>
#include<pthread.h>
#include<cmath>
#include<sys/time.h>
#include<atomic>
#include<thread>
#include<vector>
```

2. The global variables and structures

```
atomic_flag lock_stream = ATOMIC_FLAG_INIT;
atomic<bool> cnt(false);

//Code to calculate time of algorithm in micro seconds
long getTime()…

//Global variable with the same variable names as given in the problem statement
int N, K, rowInc;       //These will be input from the input file
int sharedCounter = 0;  //This will indicate the current row being processeed

/* Creating a 2D array */
int **A;  //Input
int **C;  //Output
```

3. The multiply Function :

```
//Compute the value of the rowNo row
void multiply(int rowNo)
{
    int ans;
    for(int i = 0 ; i < N ; i++)
    {
        ans = 0;
        for(int j = 0 ; j < N ; j++)
        {
            ans += A[rowNo][j] * A[j][i];
        }

        C[rowNo][i] = ans;
    }
}
```

4. The runner function : This will be explained in more detail below
5. The main function

Lets go through them one by one

1. The include headers are common in all my source codes
2. The global variables are same in all source codes with small exceptions of the atomic variables.
3. The multiply function is same in all codes and takes in a rowNo, then calculates the corresponding row of the output matrix
4. The runner functions : Each runner function is different, however they all follow a common pattern : they all have
    a. Code to acquire the lock
    b. Critical Section
    c. Code to release the lock
    d. Remainder section
5. In the main function, I have used C++ inbuilt functions to declare threads

```
vector<thread> threads;
for(int i = 0 ; i < K ; i++)
{
    threadNumber[i] = i;
    threads.push_back(thread(runner, &threadNumber[i]));
}

//We wait for all threads to finish
for(auto &th : threads)
{
    th.join();
}
```

Here, the **std::thread** constructor is used to create threads, and it handles thread attributes implicitly. The thread is created with default attributes, and you don't explicitly initialize or modify thread attributes.

Lets take a look into each of the runner functions :

1. TAS

```
//dividing the work to threads, the input parameter shows the thread number
void *runner(void *param)
{
    //A loop that runs till all the rows of the output matrix has been calculated
    do
    {
        // Acquire the lock
        while(lock_stream.test_and_set()) {} // Wait until the lock is available
        int i;

        // Critical section
        {
            i = sharedCounter;
            sharedCounter += rowInc;
        }

        // Release the lock
        lock_stream.clear();

        // Remainder section
        for(int j = i ; j < rowInc + i && j < N ; j++)
        {
            multiply(j);
        }


    } while (sharedCounter < N);

    pthread_exit(0);
}
```

Here, I used the inbuilt C++ function test_and_set to spin the while loop to cause busy waiting.

Then I am releasing the lock using the clear() function.

In the critical section, I am first storing the current value of shared counter then I am updating the value of the shared counter.

In the remainder function, I am assigning the threads rows starting from ‘i’ to ‘i + rowInc’, i.e. a total of rowInc rows.

2. CAS

```
//dividing the work to threads, the input parameter shows the thread number
void *runner(void *param)
{
    int k = (*(int *)param);            //Using the input parameter as the thread number and storing it in the integer data type

    bool old_value = false;


    //A loop that runs till all the rows of the output matrix has been calculated
    do
    {
        int i;

        //Acquire the lock
        while(!cnt.compare_exchange_strong(old_value, true)) {
            old_value = false;
        };

        //Critical Section
        {
            i = sharedCounter;
            sharedCounter += rowInc;
        }

        //Free Critical Section
        cnt = false;

        //Remainder Section
        for(int j = i ; j < rowInc + i && j < N ; j++)
        {
            multiply(j);
        }

    }while(sharedCounter < N);

    pthread_exit(0);
}
```

Here, I used the inbuilt C++ function compare_exchange_strong to spin the while loop to cause busy waiting.

The critical section and remainder sections are same as above.
Note that in the while(!cnt.compare_exchange_strong(old_value, true)) loop, I make sure that old_value gets flase again so that it can cause busy waiting.

3. Bounded CAS

```
//dividing the work to threads, the input parameter shows the thread number
void *runner(void *param)
{
    int k = (*(int *)param);           //Using the input parameter as the thread number and storing it in the integer data type

    bool old_value = false;

    //A variable to store the current value of shared counter so we could use it for our matrix computation
    int i;

    //A loop that runs till all the rows of the output matrix has been calculated
    do
    {
        waiting[k] = true;
        bool key = true;

        //acquiring the lock
        while(waiting[k] && key)
        {
            key = cnt.compare_exchange_strong(old_value, true);
            old_value = false;
        }

        waiting[k] = false;

        //critical section
        {
            i = sharedCounter;
            sharedCounter += rowInc;
        }

        //exit condition
        int j = (k + 1) % K;
        while((j != k) && !waiting[j])
        {
            j = (j + 1) % K;
        }

        if(j == k)  cnt = false;
        else waiting[j] = false;

        //remainder section
        for(int j = i ; j < rowInc + i && j < N ; j++)
        {
            multiply(j);
        }


    }while(sharedCounter < N);


    pthread_exit(0);
}
```

We need an additional waiting array to check the status of each thread. Next, we have key to store the return value of the while loop, it is initialized to true to ensure at least one iteration of the while loop.

Critical and remainder sections are same as the above codes.

Here the exit condition ensures that the next thread Tj enters its critical section since waiting[j] is set to false and it allows it to break out of the spinning while loop.

4. Atomic

```
atomic<int> sharedCounter(0);   //This will indicate the current row being processeed, unlike other codes, here it will be atomic
```

```
//dividing the work to threads, the input parameter shows the thread number
void *runner(void *param)
{
    int k = (*(int *)param);            //Using the input parameter as the thread number and storing it in the integer data type

    //A loop that runs till all the rows of the output matrix has been calculated
    do
    {
        int i;

        //Critical Section
        {
            i = sharedCounter.fetch_add(rowInc);
        }

        //Remainder Section
        for(int j = i ; j < rowInc + i && j < N ; j++)
        {
            multiply(j);
        }

    }while(sharedCounter < N);

    pthread_exit(0);
}
```

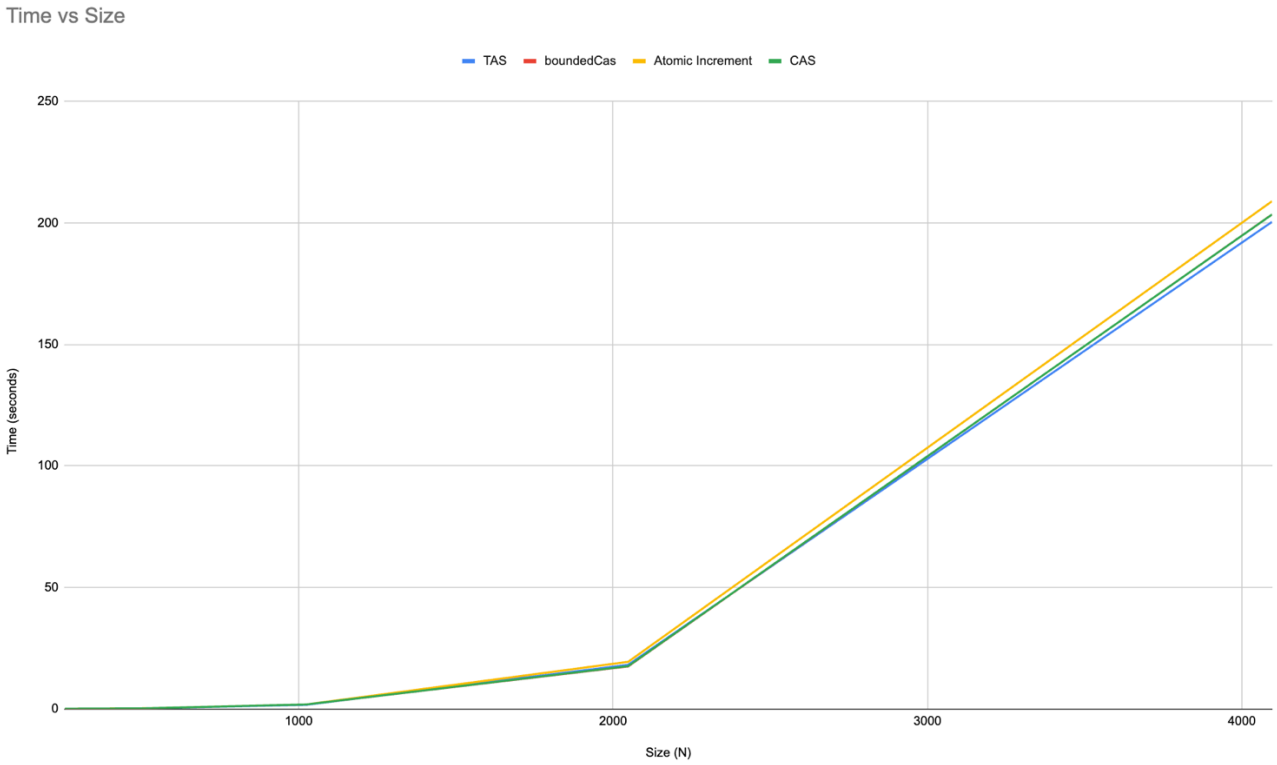
Atomic as default acts as a lock, and here we just increase it value by rowInc in the critical section. The value of shared counter is first stored in the ‘i’ variable and then increased by rowInc simultaneously i.e. atomically.

Remainder section is same as above.

Algorithm Analysis

1. Time vs Size

N	TAS	CAS	bounded CAS	Atomic
256	0.0445382	0.0434076	0.0434984	0.0432726
512	0.1968054	0.1946796	0.2036926	0.211245
1024	1.6437086	1.7796008	1.7766564	1.788418
2048	18.1087546	17.554764	17.4620894	19.3403112
4096	200.506112	203.51731	203.55563	209.00538



General trend : As the size (N) increases, the time taken also increases. The reason for this is trivial because as the input size increases, we have to calculate more values for the output matrix, (of the order O(N^2)), therefore as N increases, the time also increases

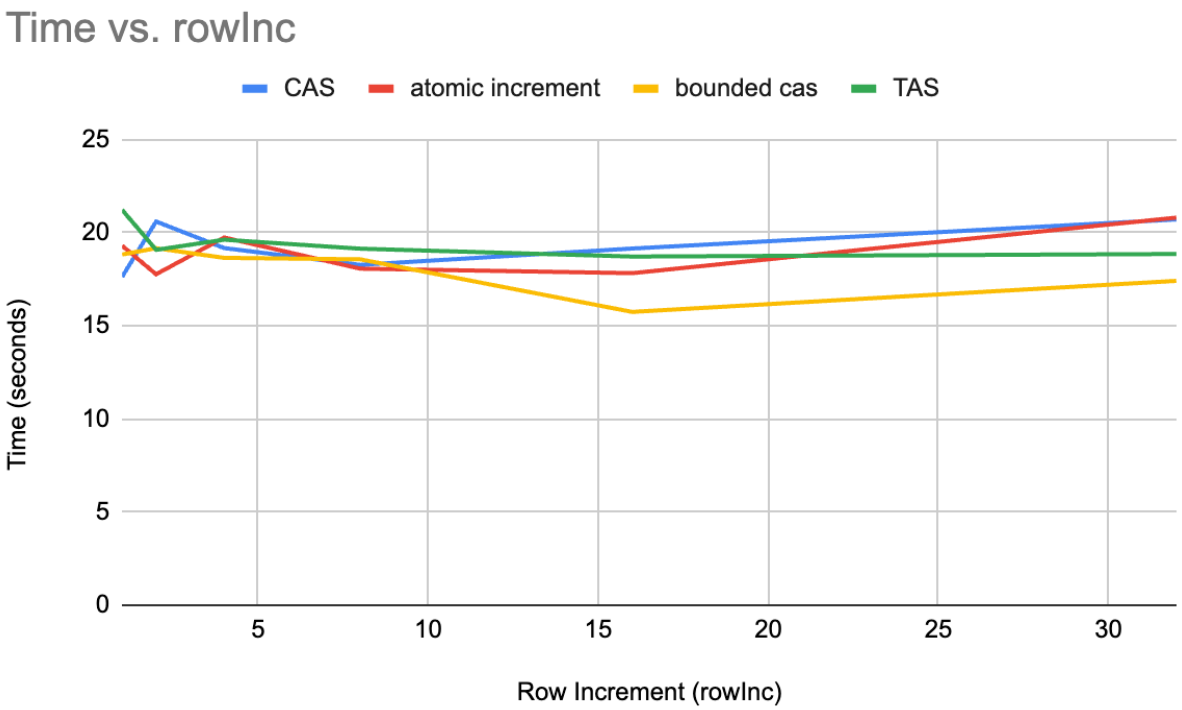
Note : In my graph, the boundedCAS and CAS values are almost same and overlapping.

Algorithm comparision : Atomic increment is taking the maximum time and TAS is taking the minimum time. TAS may be explained to be running fast due to the fact that it has low overhead

and quickly grabs the lock and proceeds with the computation. Next, compare_and_swap needs to observed values against the expected values, adding a little more overhead than TAS. It is to be noted that TAS and CAS both have busy waiting. While bounded waiting prevents starvation, due to the need for thread suspension and resumption, thereby overall having a similar time to CAS. Atomic increment is being used for synchronization in the critical section, and cause more contention among threads leading to higher execution times.

2. Time vs rowInc

rowInc	TAS	CAS	bounded CAS	Atomic
1	21.2272127	17.6310997	18.828226	19.3003867
2	19.0791643	20.600616	19.1668573	17.7685023
4	19.6229625	19.1740877	18.6487727	19.742065
8	19.1412177	18.2727017	18.569549	18.068469
16	18.7186417	19.1489433	15.746017	17.8268943
32	18.855484	20.7111535	17.4134563	20.8081463



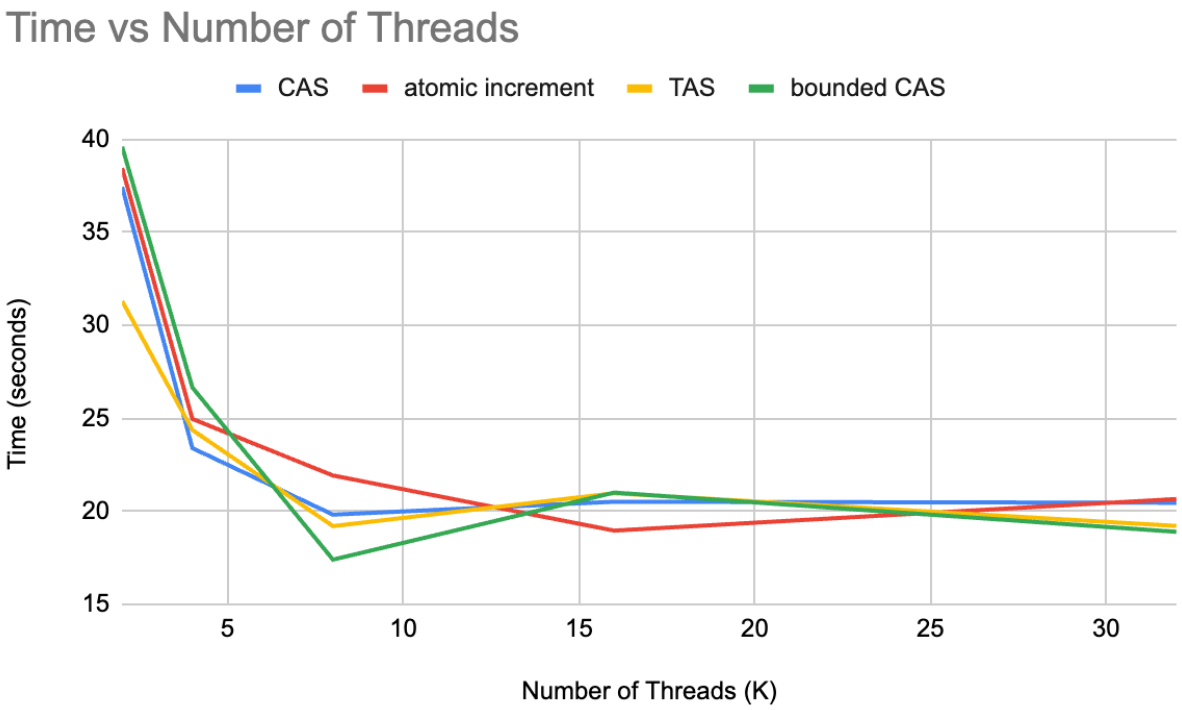
There is no clear pattern observed in this, it can just be noted that the times almost remain constant as the rowInc changes value, and the reason to this may be due to almost constant work being done.

The algorithm pattern for TAS, CAS and Atomic increment is same as above and a similar explanation could be given for these.

One reason why bounded waiting may contribute to optimal time is related to fairness and avoiding starvation.

3. Time vs Number of Threads

K	TAS	CAS	bounded CAS	Atomic
2	31.3054373	37.4440237	39.6054797	38.4610727
4	24.3743773	23.399991	26.657839	24.9680443
8	19.2055697	19.821745	17.4042053	21.9272927
16	20.9741633	20.5158473	21.007857	18.963647
32	19.2144467	20.4607437	18.8984493	20.6623333

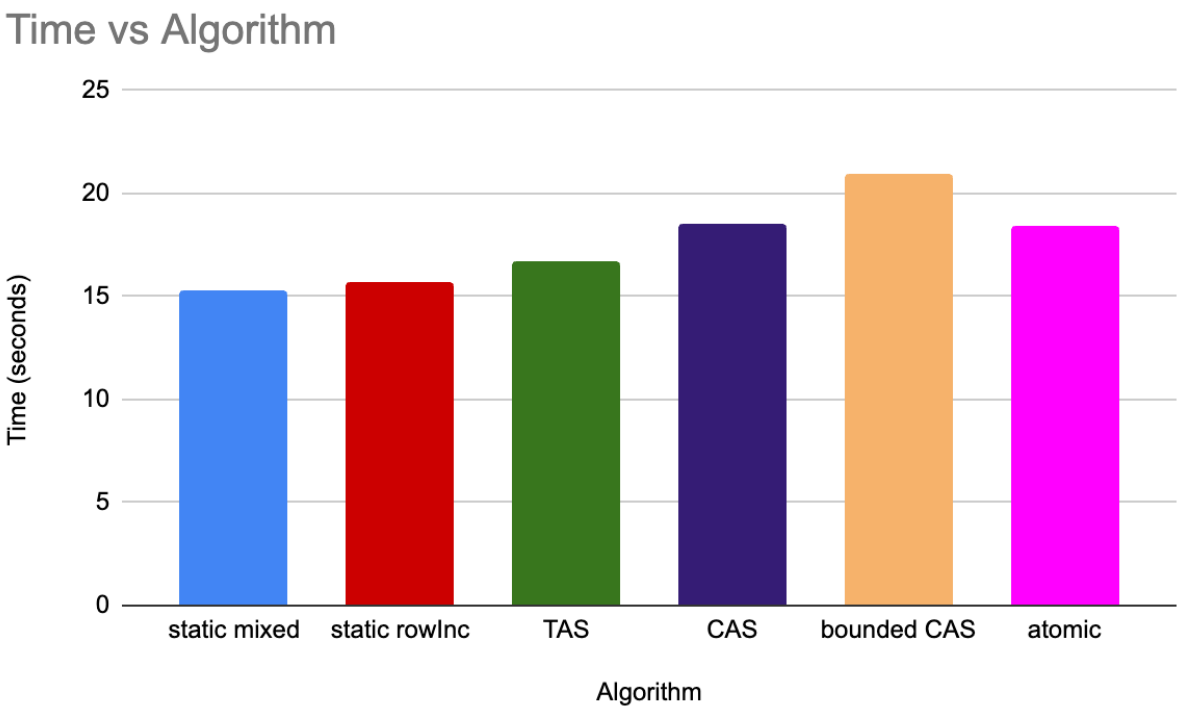


General trend: As number of threads increases, more parallelization occurs leading to decrease in time. It decreases exponentially and then eventually becomes constant.

Algorithm trend: Bounded takes most time, and the trend for the other three are observed as before. Implementing a bounded CAS mechanism typically involves additional complexity in the form of loop control, tracking the number of attempts, and handling backoff strategies. This additional overhead can contribute to increased execution time.

4. Time vs Algorithms

Algorithm	Time
static mixed	15.29121433
static rowInc	15.71063933
TAS	16.727417
CAS	18.57589967
bounded CAS	20.97113833
atomic	18.44753667



We can see that when comparing static and dynamic programs, static tend to be faster, this is because they assign a fixed value to the OS in compile time, whereas for dynamic programs, the value is assigned in run time, this involves the CPU taking (causing less predictability and increased workload for the OS). Hence static is better time than dynamic. In terms of runtime.

Final Conclusion

According to my experiments, I can conclude the following results

1. Static has better runtime than dynamic programs
2. Test and Set algorithm has the lowest runtime compared to other dynamic programs in this assignment
3. Compare and Swap and atomic are the next in the list in terms of runtime
4. Bounded runtime is not giving a clear trend in my experiments – and one possible reason for this is - bounded waiting with CAS is more about ensuring fairness and preventing situations where a thread is blocked indefinitely. It doesn't necessarily guarantee the fastest possible execution time.

Please note that I am using a macOS m1 chip, and due to the change of OS the runtimes and trends may be different in the TA's laptop.