Operating Systems–2: Spring 2024 Programming Assignment 3: Dynamic Matrix Squaring

Roll No: CS22BTECH11046

Name:P.Yasaswini

Program's low-level design:

- **1.Variables declared Initially :**
  - Output_matrix is declared globally using the 2-D vector variable "omatrix" , for updating the matrix where the matrix has updated in the overall program.
  - Declared N, K, and rowInc globally.
  - Struct thread_data is used to store thread data i.e input matrix.
  - Lock is declared globally for CAS and Bounded_CAS ,I have taken Lock as atomic so it can prevent race conditions. A waiting vector is declared in Bounded_CAS to set a particular process False/True.
- **2.Main_Function:**
  - Initialized C =0;
  - Opened "inp.txt" file using ifstream() and stored the respective input matrix in "imatrix" 2-D vector and K,N,rowInc.
  - We will resize the omatrix using resize().
  - Now we will start the time using the function which was in the chrono library.

    - Created threads using posix pthreads. As same as in assignment 1&2 we will create threads and send it to the function sq_multiplication.
    - After completion of the squaring matrix, we will join and terminate all threads using pthread.join( ).
    - We will end time and time is stored in the time_taken variable in milliseconds.
    - We will print the output matrix and time in the out.txt file.
- **TAS_Method:**
    - It was implemented in sq_multiplication, inbuilt c++ TAS was in the atomic library .
    - Lock is declared atomically so that it cannot be altered by two threads simultaneously.
    - As every thread goes into the function all will enter into the do_while loop.
    - In do_while loop:
        - while(lock.test_and_set()); Here if one process has acquired lock remaining all threads will wait until the lock has been released.
        - We will release the lock using lock.clear() after the critical section.
        - In the critical section if C is greater than equal to n then we will release the lock of that thread and break out of the loop, else we will declare the “start” variable inside the sq_multiplication function to store the C value for that particular thread.
        - As “start” is a local variable to every thread ,so there is no race condition.
        - We will implement the remainder section i.e matrix multiplication.

        - If start <n and start+rowInc >n then we will assign rows from start to n to the particular thread
    - We will end the “do_while” loop after every thread breaks out of the loop.
- **CAS Method:**
    - I have implemented the compare_and_swap function. I have used the built in compare_and_swap function called compare_exchange_strong.
    - It is the same as TAS. In CAS we compare the expected value with the current value. If both are equal then we set new_value to the current value, but it returns the old value of the current value.
    - In while(true) loop:
        - while(lock.compare_exhange_strong(temp,1)); Here it will compare the lock with 0 (temp is local variable to every thread and initialised with 0) if the lock is 0 then it will update lock=1 and the thread will come out and remaining threads will be waiting in loop until lock of this thread is set to 0.
        - Same as TAS, we will write the critical section followed by the remainder section.
- **Bounded CAS Method:**
    - Waiting vector is declared globally and resized to k in the main function and sets all waiting time of threads False.
    - Here struct thread data has input matrix + its ID(index).
    - In sq_multiplication we have implemented Bounded CAS.
    - In while loop:

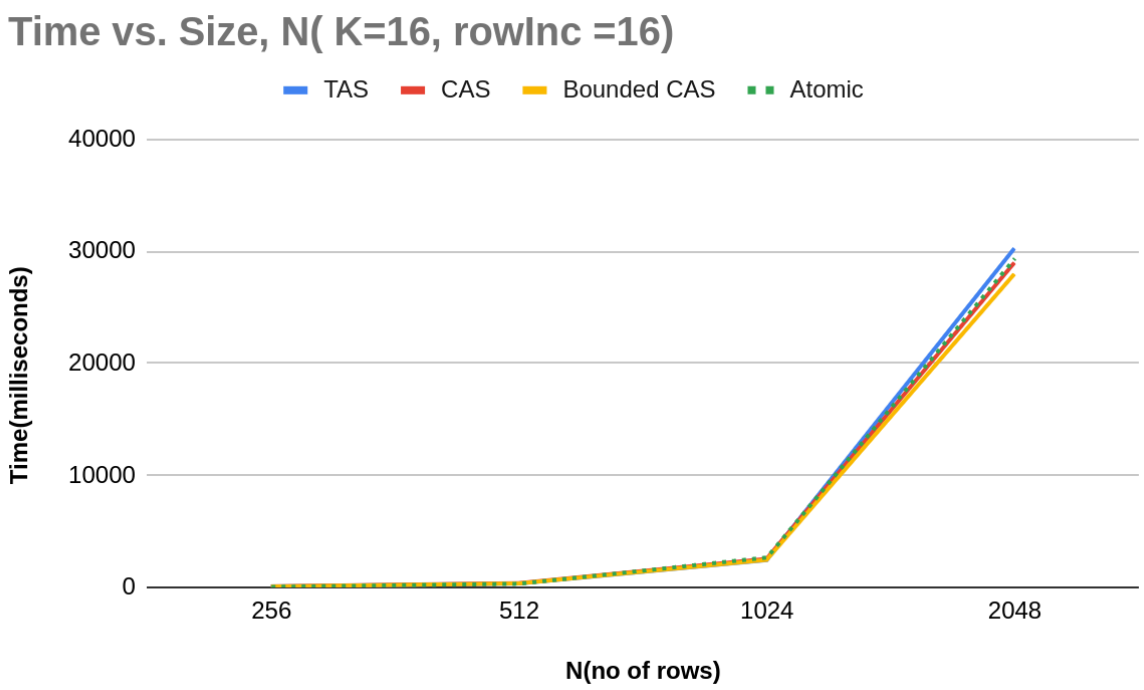
        - The threads which were going through a loop, their waiting time is set to true and key=1;(key is local variable to each thread,it will allow the other threads to wait).
        - In the inner while loop we will do compar_and_swap with the lock . At first thread k will be 0 and for remaining threads k=1. The first thread will do the critical section and others will wait until this thread lock of the first thread is set to 0.
        - Waiting time of the first thread will be set to false because it should wait until all threads had completed the critical section otherwise it may come twice or thrice before the remaining waiting threads which have not even started the critical section for the first time.
        - To ensure the above mentioned problem we implement a second while to check that every process has done the critical section or not, if the thread is waiting then it will set its waiting time False so that it can execute critical section i.e first inner while loop break if either waiting time is false or key =0.
        - Next it will do the remainder section and all will break out of the loop if C>=(Number of rows).
- **Atomic increment :**
    - We will set **C** as an atomic variable because it should not be altered by two or more threads simultaneously, as it is uninterruptible.
    - There is an inbuilt function fetch_add, I have incremented C by value rowInc in sq_multiplication.
    - Atomic will ensure C to not have any race conditions.
    - The Rest is the same.

Experiment 1: Time vs. Size, N:

Table:

	TAS	CAS	Bounded CAS	Atomic
256	36	37	35	40.25
512	304.5	324	314.5	320
1024	2391.5	2495	2414	2664
2048	30215.6	28934.8	27940.8	29346

Graph:



Observations:

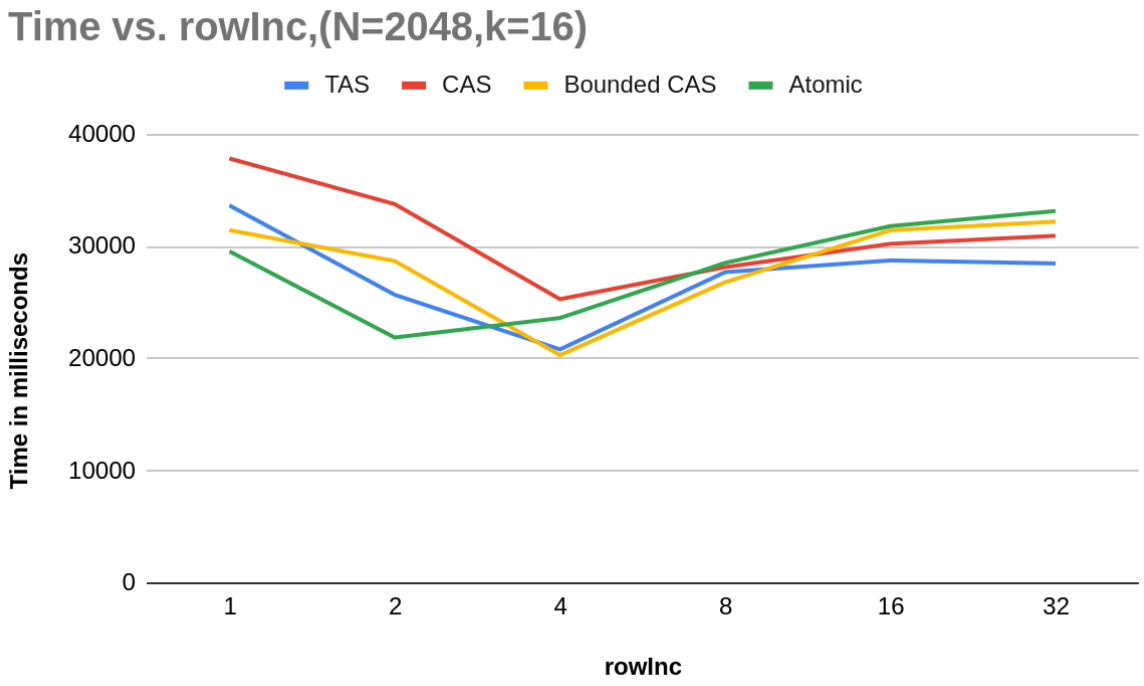
- All methods are showing significant growth of time on increasing the number of rows of a matrix.
- Because computing a bigger matrix takes time, irrespective of the method.

- Here on increasing ‘N’ bounded CAS is showing slightly better performance because it ensures every thread goes through the critical section at least once so here we are using threads in a more efficient way.

Experiment2:Time vs. rowInc, row Increment:
Table:

rowInc	TAS	CAS	Bounded CAS	Atomic
1	33615	37809	31429	29526.6
2	25656	33741.6	28683.2	21850
4	20788	25275.2	20290.6	23594.2
8	27679.6	28126	26793.2	28507
16	28721.4	30201	31420	31782.8
32	28448.2	30917	32194.5	33125.2

Graph:



Observations:

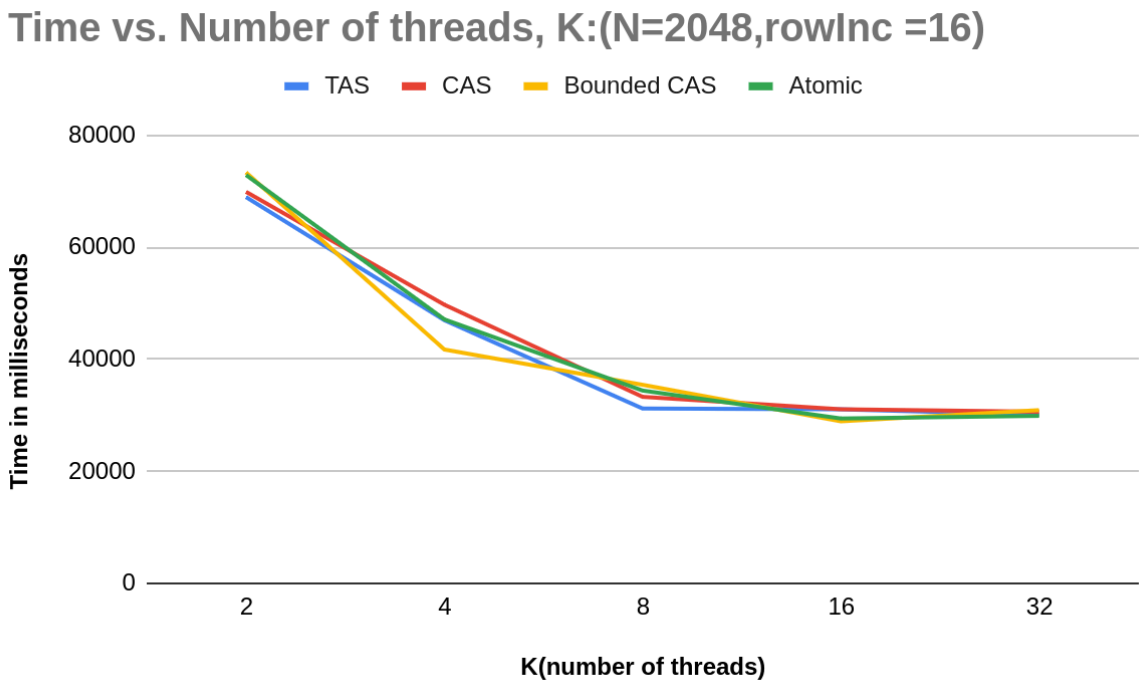
- With smaller rowInc value atomic is showing better performance and CAS is showing least performance.
- Actually, TAS and CAS should show almost the same performance.
- On increasing rowInc almost all shows same performance but Atomic and bounded CAS are higher than TAS and CAS.
- In bounded CAS the time is higher because if the first thread has completed the rows multiplication but it will wait until all threads had reached critical section once as matrix bigger and rowInc is bigger, the computation takes more time and waiting time will also increase.

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

Table:

	TAS	CAS	Bounded CAS	Atomic
2	68865.2	69790	73259	72796.6
4	46902.2	49644	41665.4	47029
8	31133.8	33205.4	35375.6	34324
16	30989	30979.4	28856.4	29346
32	30030.4	30546.4	30850.8	29816

Graph:



Observations:

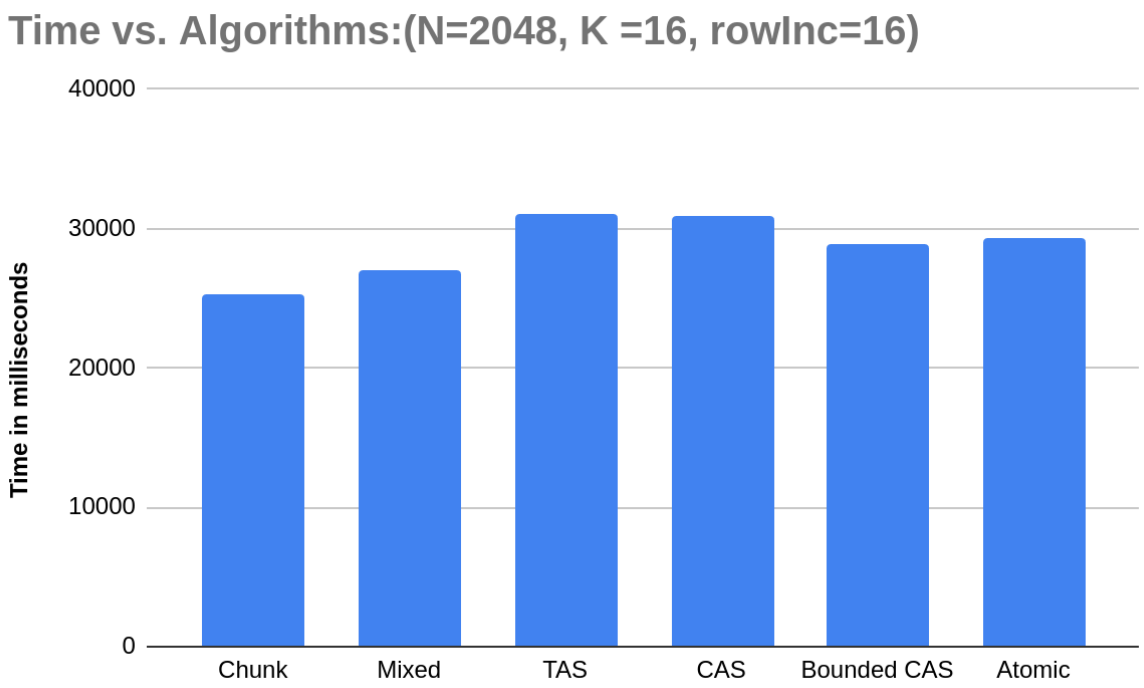
- Here all methods show almost the same performance.
- On increasing threads the workload to each thread decreases implies work distributed evenly so the efficiency increases on increasing number of threads.
- Here Bounded CSA shows somewhat better performance than others because it ensures all threads are doing their tasks without more waiting.
- Conclusion: irrespective of method the performance will increases on increasing number of threads.

Experiment 4:Time vs. Algorithms:

Table:

Chunk	Mixed	TAS	CAS	Bounded CAS	Atomic
25326	27068	30989	30979.4	28856.4	29346

Graph:



Observations:

- Static chunk shows better performance than all methods because we are fixing threads statically so there is no race condition occurs.
- Chunk is better than mixed because we are assigning rows contiguous.
- TAS and CAS show almost the same performance because implementing these methods are almost the same.
- Bounded CAS is less, this is Obvious because we are not waiting one thread more time than the other thread.
- Atomic shows between CAS/TAS and Bounded CAS Here every thread will access Counter atomically.