# Report-ASSIGNMENT 3 OS-2 –

(CE21BTECH11008)

Reading The file My input file name is “inp.txt” and output file name is “out.txt” . For OutFile I have made a file pointer that is called at the end of the program and getting written using answer matrix.

For input file using ifstream from fstream library and taking input in the main () function. “In.txt” contains a line. Extracting the N and K from this by simple reading and called “read matrix” functionFrom reading determining the N and K and rowinc value and making “matrix” 2d vector of N x N size and storing the value of K and N using pass by reference. “rowinc” is used to increase the count by rownc and thread which won the bid of critical section will get these much consecutive rows at a time.

To measure Time: Used “auto start_time = std::chrono::high_resolution_clock::now();” found in time.h library . As we are intended to find the time taken by threads to execute on different set of assigned rows. So, I am starting the time for each thread when it is entering in that function and end_time when that thread is just about to call return. Taking difference to find interval. Now made one vector name “time” that is keeping track record of the time taken by each thread. It is free from race conditions because every thread has its unique index. Finally, taking the average of the time taken by each thread.

Threading Methodology I have used Posix thread and called pthread_create function with its arguments and for parameter passing I have made struct data type. That is containing N and K and 2d vector matrix to be passed as this is the data used by all. For passing pthread_create I have made array of pthread_t of size k where the function instance will be stored. Also, I am passing ans matrix that is keeping track of the result. And last have called pthread_join function as no return value is there so passed its argument as null and thread id th[i].

**For test and set**

I have use function name test and set

“flag.test_and_set(std::memory_order_acquire”

which is built in c++. This function takes the value of the lock. It will return the value of lock i.e. input and change the value to 1 if it is 0 and not changed if already 1 but it will return the previous value. Also, this is hardware implanted to make sure that race conditions should not take place.

At last to unlock this lock I have used

“flag.clear(std::memory_order_release)” it will release the lock.

**For compare and swap**

I have use function name test and set

```
   bool expected = false;
   bool desired = true;
   while (!lock1.compare_exchange_strong(expected, desired)) {
    expected = false;
     desired = true;
```

which is built in c++. As compare_and_swap() is depricited in x86 system so I have used

compare_exchange_strong() this will take two value expected and desired initially have value false and true. Initial value of lock will be false that will show this lock has not be given to anyone. The thread which comes first and will acquire the lock. This function take the lock and return true if given value of lock is false and viceversa. It will also change the expected value to true if false so we again have to setup.

At last to unlock this lock I have used

“lock1.store(false); it will release the lock by setting the value of lock.

**For compare and swap(bounded)**

**I have used same function as previous**

```
        bool expected = false
        key = !lock1.compare_exchange_strong(expected, true);
```

for bounded waiting I have taken inspiration from the Galvin book and also teaches by sir. It is maintaining waiting[K] that is declared globally to avoid conflict. In each thread have thread unique thread id and waiting function will tell those thread are actually waiting for critical section of not. If yes then it will be true.This function will work exactly same as previous one.Key value is also added that will be decide that which thread is going to run if so many threads are there waiting.

**Ad**’s while (*count1 <N-1)

```
waiting[thread_no] = true;

key = true;


while (waiting[thread_no] && key==true)

{

  bool expected = false;


  key = !lock1.compare_exchange_strong(expected, true);

}

  break;

  }
```

Looping into the circal to check which thread wants to acquire the critical section, this will ensure that every one will get turn by after n-1 turns definitely. This solve the problem of bounded waiting.

```
int j = (thread_no + 1) % K;

while ((j != thread_no) && !waiting[j])

{

  j = (j + 1) % K;

}
```

For unlocking same function is used.

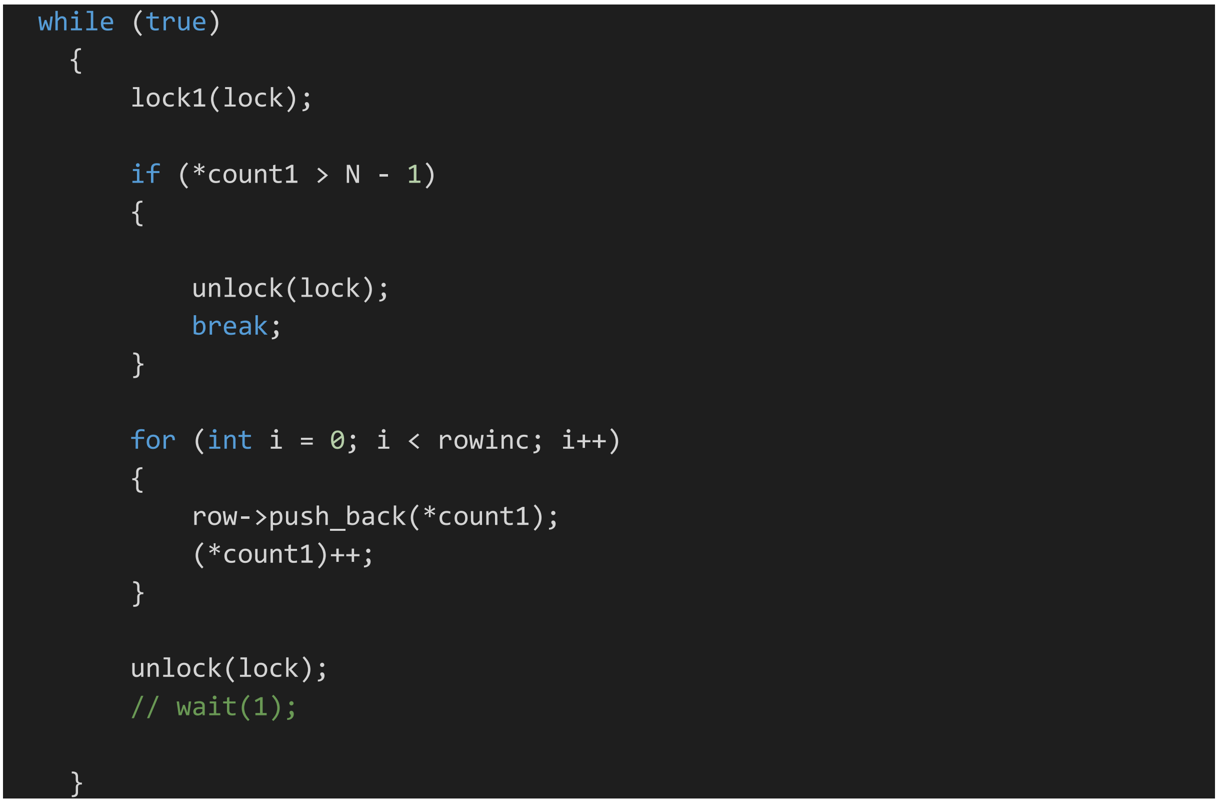
```
  lock1.store(false);
```

**Atomic c++**

For this I have used

```
__sync_fetch_and_add(&lock->ticket, 1);
```

And uses the methodology of ticket and turn. The one who raises the ticket will get turn after some time to execute in critical section. This is also a bounded waiting algoritnm.



```
while (true)
  {
      lock1(lock);

      if (*count1 > N - 1)
      {

          unlock(lock);
          break;
      }

      for (int i = 0; i < rowinc; i++)
      {
          row->push_back(*count1);
          (*count1)++;
      }

      unlock(lock);
      // wait(1);

  }
```

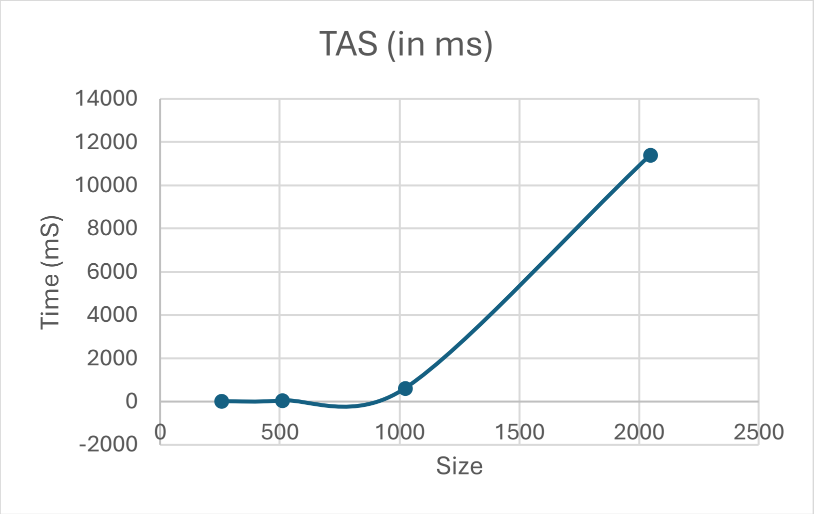
1. Time vs. Size, N: The y-axis will show the time taken to compute the square matrix in this

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. Please have the

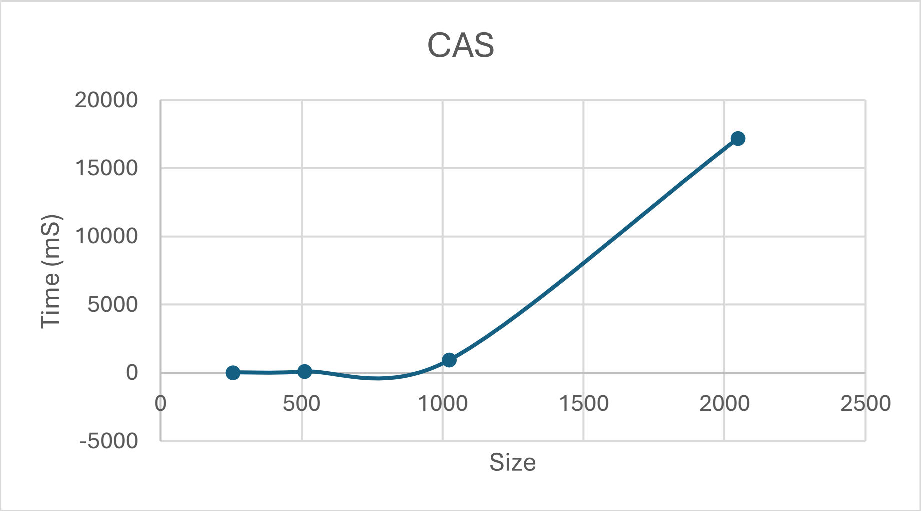
rowInc fixed at 16 and K at 16 for all these experiments.

**1.Test and set (Time vs size)**



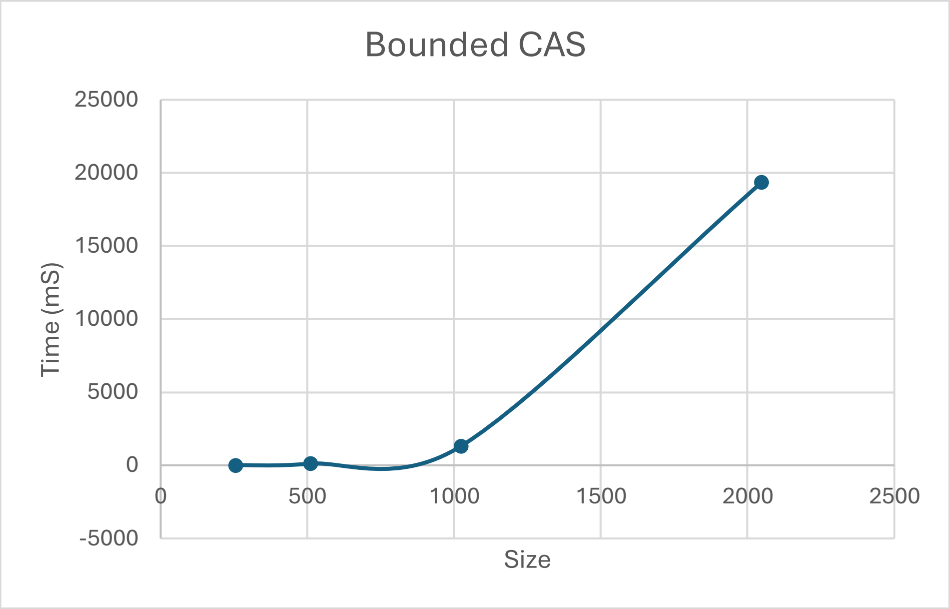
Time is increasing exponentially with respect to size. For the size of 2048 matrix it taking approximately 12 second and for the whole code it is taking a minute. In this number of core was kept fixed to 16 and rowinc also 16.

**Compare and swap**

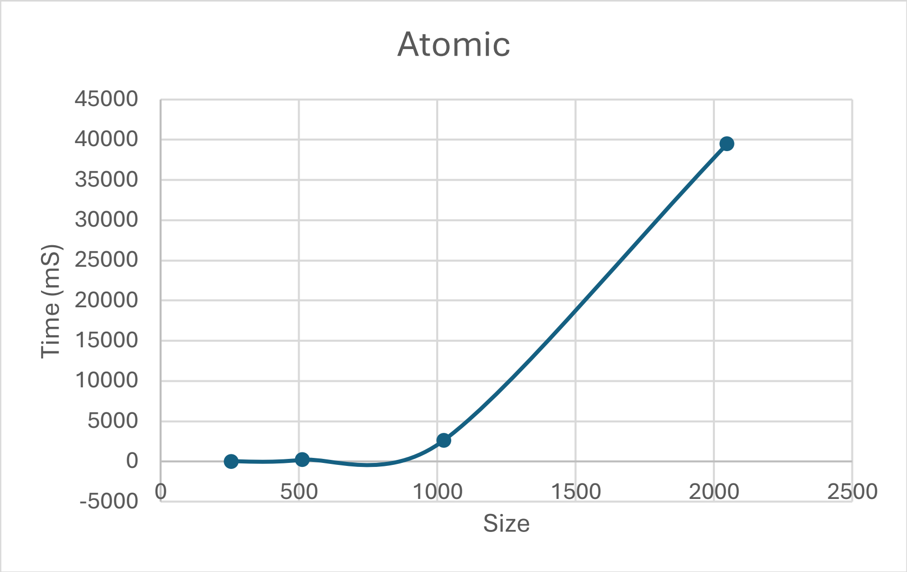


For the compare and swap same trend is following but the time taken by CAS is comparitevly more than the test and set. This is conflicting our idea that CAS typically requires less memory overhead than TAS. In TAS, you need to maintain a separate flag or lock to indicate whether the value is being accessed or modified. This can increase memory consumption and complexity. CAS, on the other hand, doesn't require any additional memory overhead as it's an atomic operation. This opposite order may be attributed to the use of different compare_and_swap i.e. "compare_and_exchange_strong()". As the name suggest this is the strong form and thus it guarantees the atomicity and lead to morse overhead and thus leading to more time.

**Compare and swap(bounded)**



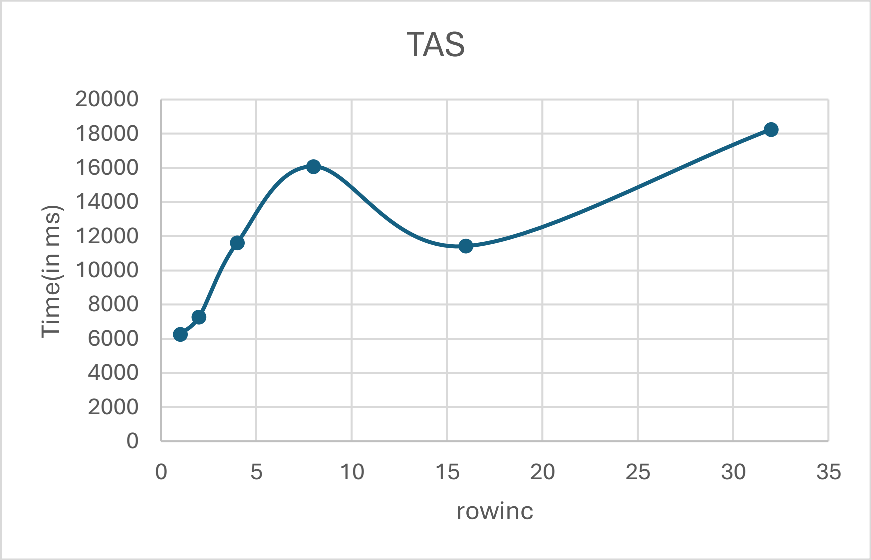
As expected Compare and swap with bounded waiting will little bit increase the time because overhead to mentain this algorithm is higher than previous, By looking the graph it has not actually benefit when each thread have have bounded waiting. It may be more efficient in the situation when there is chance that some thread of matrix multiplication will starve. So it is likely that if such matrix are given as input then it will perform better than those as starving of thread may lead to very high time.

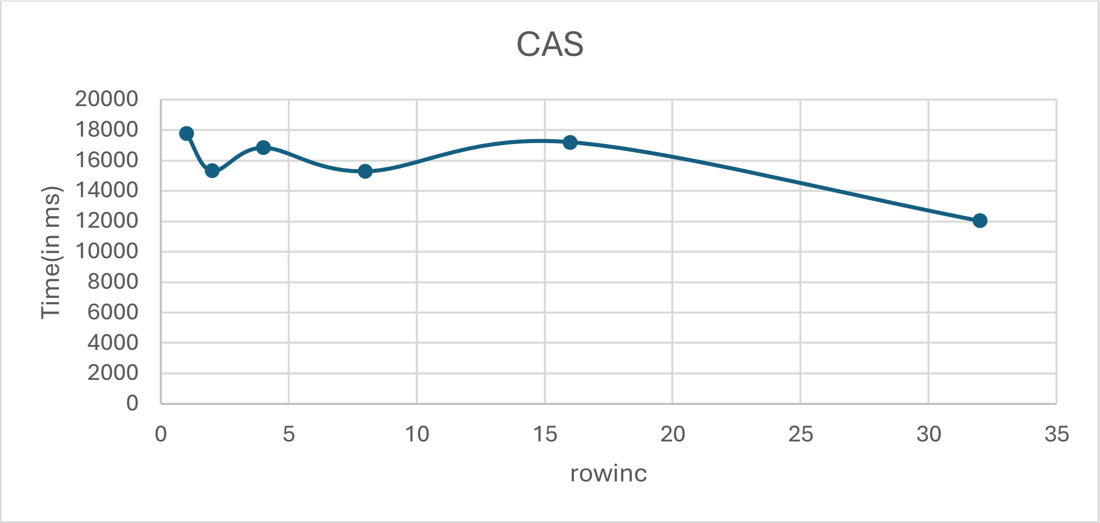


C++ atomic is taking much higher time than all of the other algorithms. This may be due to complex algorithms atomic will follow. This will lead to the much higher time than previous.

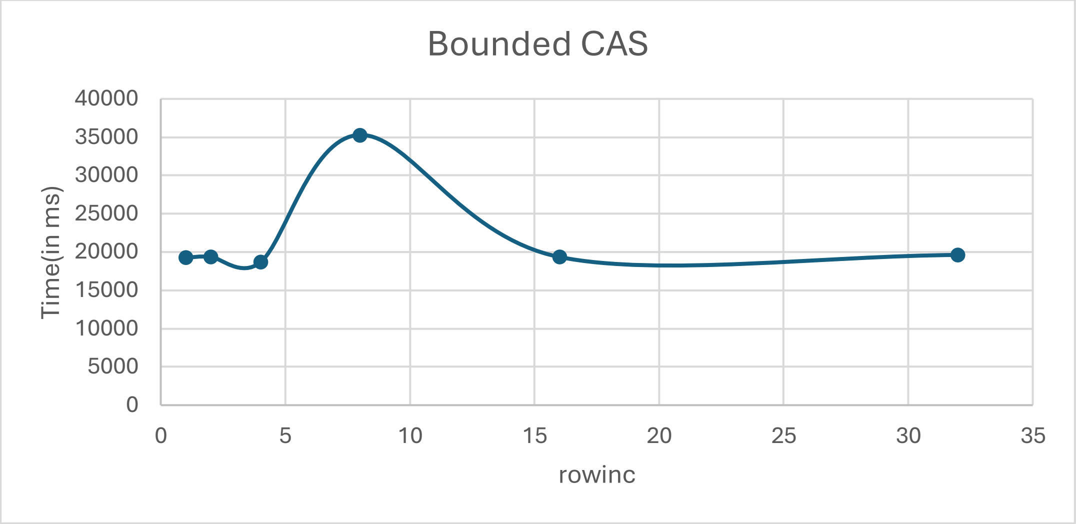
2. Time vs. rowInc, row Increment: Like the previous graph, the y-axis will show the time to compute the square matrix. The x-axis will be the rowInc varying from 1 to 32 (in powers of 2, i.e., 1,2,4,8,16,32). Please fix N at 2048 and K at 16 for all these experiments.

Ans—Keeping N= 2048 and K =16

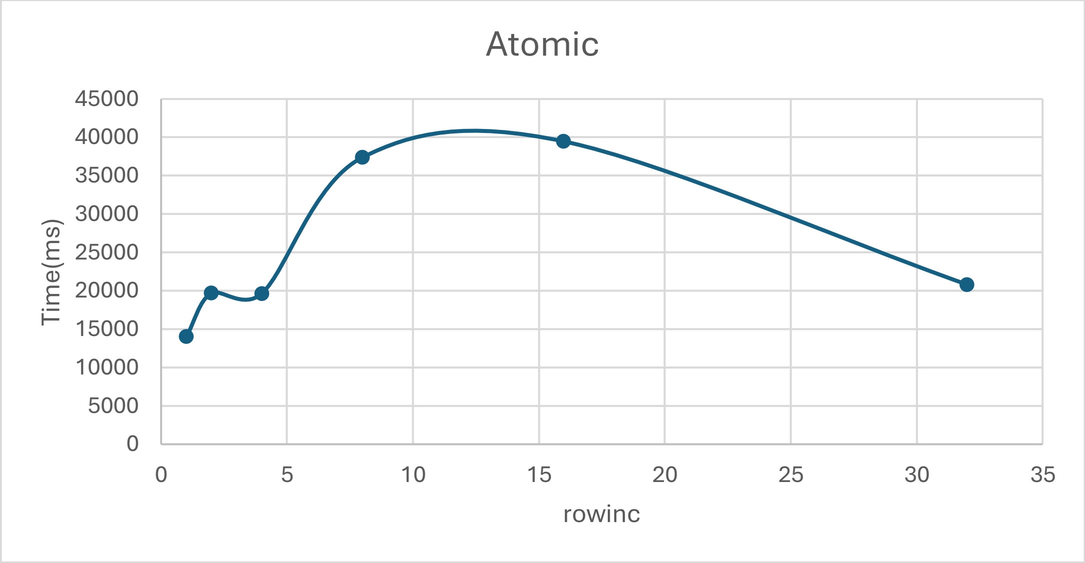




CAS
Time(in ms)
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
5
10
15
20
25
30
35
rowinc



Bounded CAS
Time(in ms)
40000
35000
30000
25000
20000
15000
10000
5000
0
0
5
10
15
20
25
30
35
rowinc



Atomic
Time(ms)
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
0
5
10
15
20
25
30
35
rowinc

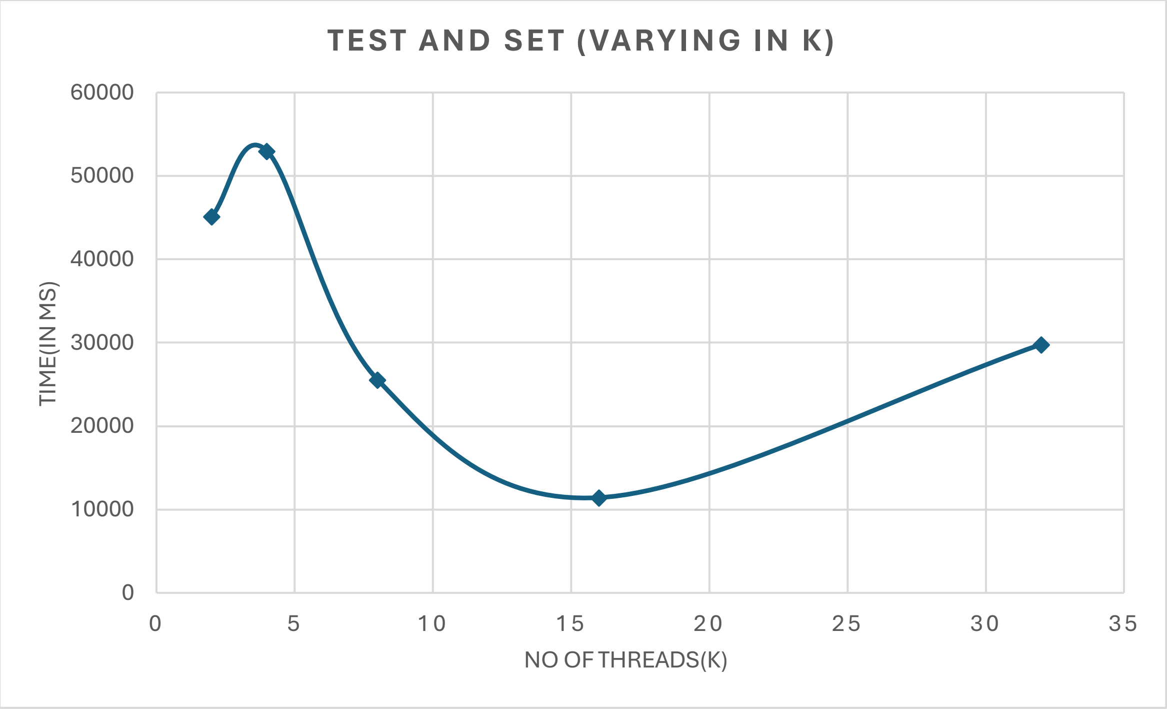
Observation- Variation of the graph is not even when rowinc is small but increasing the number of rowinc is decreasing the time and reaching to saturated level . As chunk increase the chance of cache miss due to locality of the data. So this trend may be attributed to that . Saturation is also taking place it may be because locality will take the date of fixed byte like generally 64 bytes. Thus increasing the size of rowinc may not actually increase the performance thus leading to some sort of saturation. Continuing the previous trend of time between these

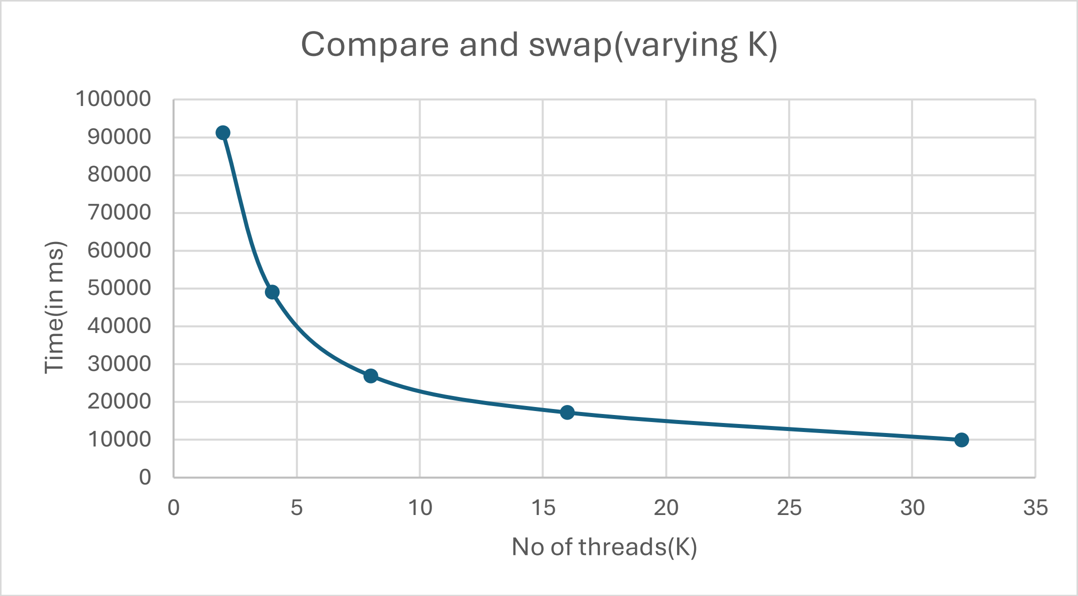
On an average, atomic>comper_and_swap_with bounded waiting > compare and swap>test and set.

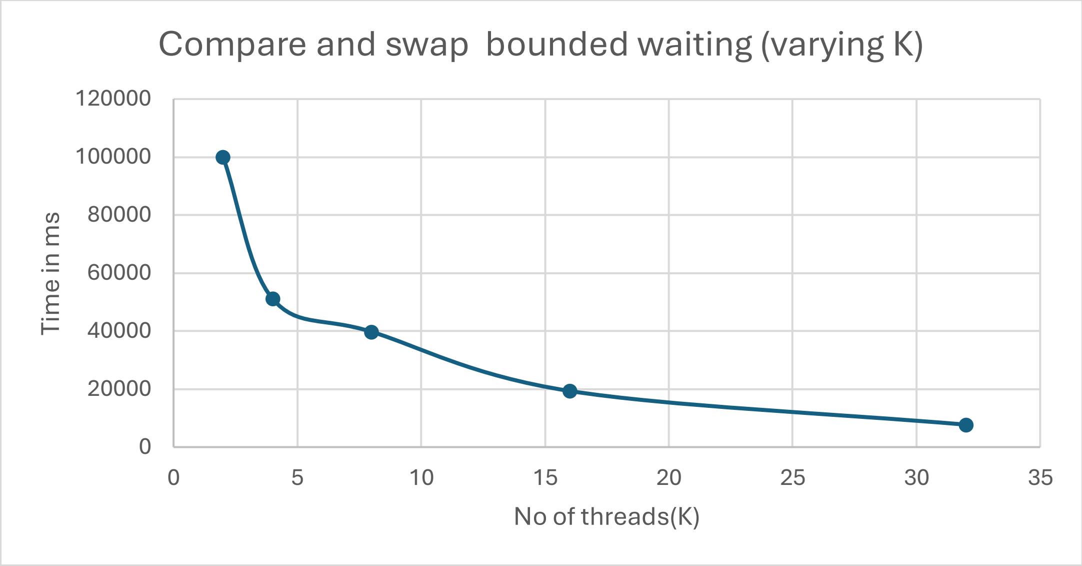
Similar arguments as the previous experiment can be given . Some of the behaviour in this graph is not explinable. It is hard to tell why actually this pattern are coming but these may have dependency that how these are implemented on hardware label and also how scheduler acts with these atomicity. Also dependency of the type of input matrix cant be neglected. So this may be the probable reason for these uneven trends.

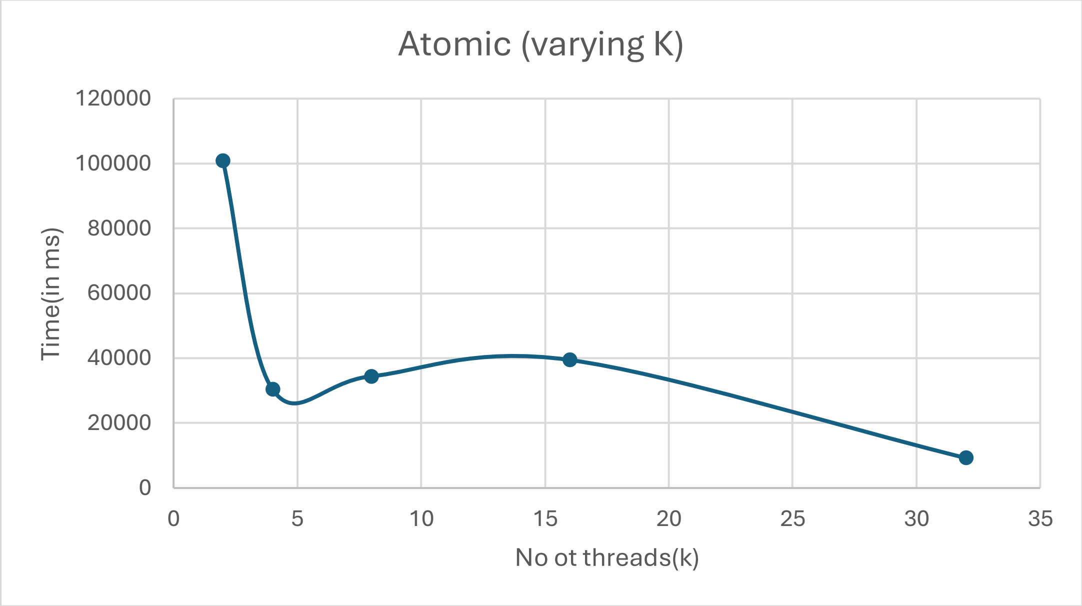
3. Time vs. Number of threads, K: Again, like the previous graphs, the y-axis will show the time taken to do the matrix squaring, and the x-axis will be the values of K, the number of threads varying from 2 to 32 (in powers of 2, i.e., 2,4,8,16,32). Please have N fixed at 2048 and rowInc at 16 for all these experiments.

Ans:--




Compare and swap(varying K)
Time(in ms)
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
5
10
15
20
25
30
35
No of threads(K)


Compare and swap  bounded waiting (varying K)
Time in ms
120000
100000
80000
60000
40000
20000
0
0
5
10
15
20
25
30
35
No of threads(K)


Atomic (varying K)
Time(in ms)
120000
100000
80000
60000
40000
20000
0
0
5
10
15
20
25
30
35
No ot threads(k)

Observation:-- All the experiments(except test and set) it is found that as number of thread are increasing time is decreasing. This may be due to increasing the number of threads increasing the parallelism and thus increasing more distribution of rows in between this is leading to decrease in the number of row that should be calculated by a single row on an average thus leading to better timing.

While somehow diverted trend has been seen test and set. As the number of thread decreasing the time and further increase after 8 is increasing the time . This may be due to overhead in the number of threads and its maintenance.

If we observer the timing then on an average test and set is still doing better while other there test and set ,test and set with bounding and atomic of c++ is approximately giving same result.

4. Time vs. Algorithms: Again, like the previous graphs, 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

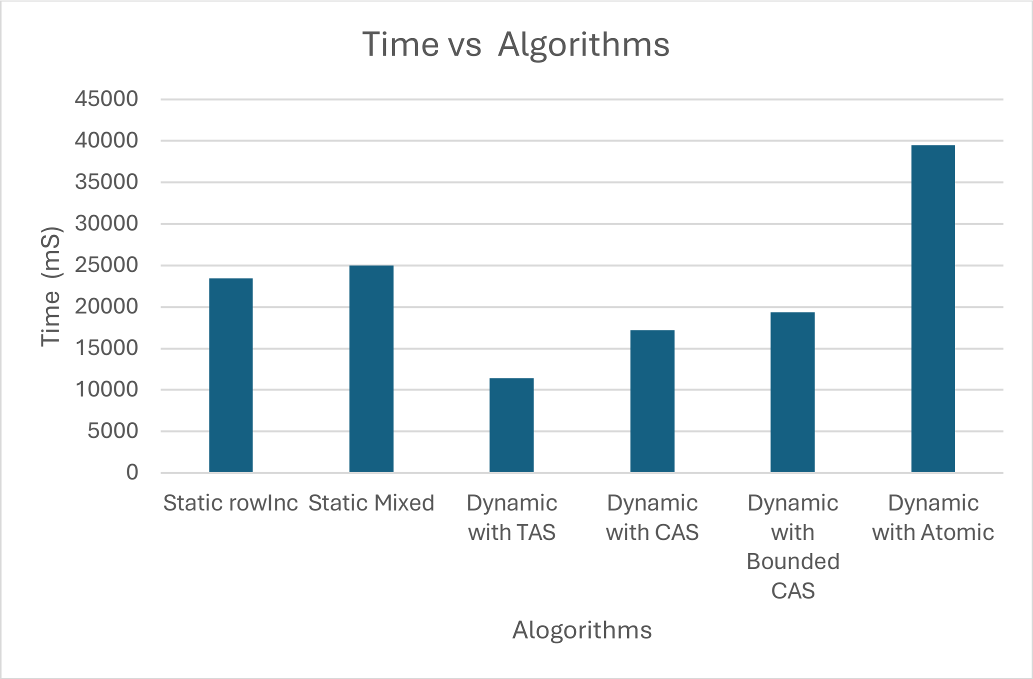
d) Dynamic with CAS

e) Dynamic with Bounded CAS

f) Dynamic with Atomic.

Ans:-

| | |
|---|---|
| Static rowInc | 25010 |
| Static Mixed | 23484 |
| Dynamic with TAS | 11420 |
| Dynamic with CAS | 17200 |
| Dynamic with Bounded CAS | 19350 |
| Dynamic with Atomic | 39470 |
| | |



Obseravations:--

It is observed that  TAS is working more efficiently than all other algorithms. Also inspite of busy waiting (spin lock) in dynamic algorithms TAS,CAS, CAS with bounded waiting is still working better than static algorithms. This may be due to benefit of dynamicity as no extra timing is going on statically allocate those. In dynamic then which is coming first to get the row is getting the row first. As in static this was the main problem like one thread has already been completed the work while other has not been completed yet. Atomic of c++ is still taking more time than all of the other this may be due to internal implementation such that spin lock is encountering the benefit of the dynamicity. Also static rowinc (chunk) is providing better time than mixed. This may be attributed to more cache hit benefit in chunk in comparison to mixed.

**Measuring observations:-**

1.when I reboot my PC same algorithm started showing different timing than what It was taking before rebooting.

2.Also when laptop was in battery saver mode it was reducing its processor speed and thus changing the observed time significantly.

Conclusion: --In this experiment I have tried to take every possible aspect that caused this time but at many points I was unable to make correlate the behaviour. Also, It is good to measure the interval duration at same time if possible otherwise there very high chances that the processor condition change and the process will not give the actual time what it should as scheduler will schedule the things accordingly on number of process available. Due to hardware involvement is hard to tell the some particular trends.