OS Programming Assignment 3 – Report

Dynamic

Design Of Program

I have submitted a total of 4 programs – one for each of the 4 dynamic methods. The general structure of all the programs is as follows –

1. Taking input from the inp.txt file – all the important parameters such as n, k, rowInc are taken from here
2. After this, we create the array of structures – threads[k] – to store the data to be passed on to each of the threads
3. After this, the threads are created. Now – the dynamic approaches come into picture. I will explain all of my 4 approaches one by one.

Test and Set

I have explained every line of the critical section – and how the test_and_set works with the help of comments:

Here, the test_and_set inbuilt function returns the value of the lock. If the lock is initially 0, this indicates that it is free, and can be acquired by the present thread. Hence, lock is marked 1, test_and_set returns 0, and we break out of the busy waiting condition – and execute the critical section.

If, the lock was initially 1, it means that it has already been acquired by some other thread, and hence we just keep busy waiting in this thread, unless control of lock is gained.

```
void *runner(void *param)
{

    struct calcInfo *thread = (struct calcInfo *)param;
    while (true)
    {
        while (lock.test_and_set(memory_order_acquire))//Test and set returns the value of the lock
            ;                                           //If the lock was initially 0, this indicates that it
                                                        //is free and can be aquired. In this case, we break out
                                                        //of the while loop, and set the lock to 1.
                                                        //If the lock was initially 1, it means it has been taken
                                                        //By some other thread already, and this thread
                                                        //Stays in busy waiting condition

        // critical section
        if (counter == thread->totalNumberOfRows)//Checking if we have already reached the value -
        {                                         //totalNumberOfRows, and exiting if it is so
            lock.clear(memory_order_release);
            pthread_exit(0);
        }

        // Now starts the caputring of the lock and picking up the chunk
        int rowStart = counter; //Allotting the current value of counter to rowStart
        counter += thread->sizeOfChunks;//Incrementing counter by rowInc
        int rowEnd = counter;//The rows from rowStart to rowEnd will be computed by this thread

        lock.clear(memory_order_release);
        //Critical Section ends here
```

Compare and Swap

Below is the code – I have used compare_exchange_strong here:

Compare_exchange_strong returns whether the exchange was successful or not. We set the expected value of lock to be 0, and if it is indeed 0, we make it 1. Compare and exchange returns a 1 if the exchange was successful, and in this case we break out of the busy waiting – and enter the critical section.

If the exchange was unsuccessful, it implies that lock was 1 – and it Is acquired by some other thread. In this case, we are stuck with this thread in busy waiting condition

```
void *runner(void *param)
{

    struct calcInfo *thread = (struct calcInfo *)param;
    while (true)
    {
        int expected = 0;
        while (!lock.compare_exchange_strong(expected, 1))//If the exchange is succesfull, it means that the value
        {                                                 //Of lock was same as the expected value - i.e. 0, and
            expected = 0;//Reinitializing to 0            //hence, this thread can acquire the lock. If the exchange
        }                                                 //Was not successful, It would return a 0, and we would be
            ;                                             //Stuck in busy waiting

        // critical section - entering it once the exchange is made
        if (counter == thread->totalNumberOfRows)//If the counter has already reached the total number
        {                                        //of threads, we exit, as all the work is over
            lock.store(0);
            pthread_exit(0);
        }

        // Now starts the caputring of the lock and picking up the chunk
        int rowStart = counter;//Storing the initial counter value into rowStart
        counter += thread->sizeOfChunks;//Incrementing the counter
        int rowEnd = counter;//Giving rowEnd the new value of counter

        lock.store(0);//Making the lock 0 so that other threads can aquire it
        //Critical Section ends
```

Compare and Swap – With Bounded Waiting

Below is the code. I have explained each line of the code with the help of comments.
Here – we store the waiting status of each of the executing threads in the waiting array.
We initially set the waiting status of the thread to be true – as it has not gained entr into the critical section yet.
The variable key stores !(value of CAS). Hence, we enter the critical section in either one of the two cases – 1. Some other thread exits the CS allows this thread to enter 2. The exchange is successful – i.e. this thread acquires the lock successfully

Once the computation of this thread is over, it looks for some other thread which is in waiting condition – and allows it to enter the CS. If no thread is waiting, we release the lock and exit this thread.

```
void *runner(void *param)
{

    struct calcInfo *thread = (struct calcInfo *)param;
    while (true)
    {
        int i = thread->threadNumber;
        int j;
        waiting[i] = true;//Making the initial waiting condition of this thread to be true
        int expected = 0;
        int key = 1;
        while (waiting[i] && key == 1)//Keeping it in busy waiting until key becomes 0(Exchange is done),
        {                              // or some other thread makes its waiting condition to be false
            key = !lock.compare_exchange_strong(expected, 1);//Making the value of the key to be the value returned by CAS
            expected = 0;//reinitializing the expected value
        }
        waiting[i] = false;//Marking it to be non waiting now

        // critical section;
        if (counter == thread->totalNumberOfRows)//Checking whether we have already reached the total number of threads
        {                                        //In that case, we release the lock and exit
            waiting[i] = false;
            lock.store(0);
            pthread_exit(0);
        }

        // Now starts the caputring of the lock and picking up the chunk
        int rowStart = counter;//Assigning the initial value of counter to rowStart
        counter += thread->sizeOfChunks;//Incrementing the value of the counter
        int rowEnd = counter;//Allotting the update value of counter to rowEnd

        j = (i + 1) % thread->numberOfThreads;
        while (j != i && !waiting[j])//We keep looking for a thread trhat is in waiting condition
        {
            j = (j + 1) % thread->numberOfThreads;//Keep incrementing until we find a thread
        }

        if (j == i)//If no threads are in waiting, we come back to i. In this case, we release the lock
        {
            lock.store(0);
        }
        else
        {
            waiting[j] = false;//We find a thread whose waiting is true. We make its waiting condition to be false
        }                      // - allowing it to now enter its CS

        //End of Critical Section
```

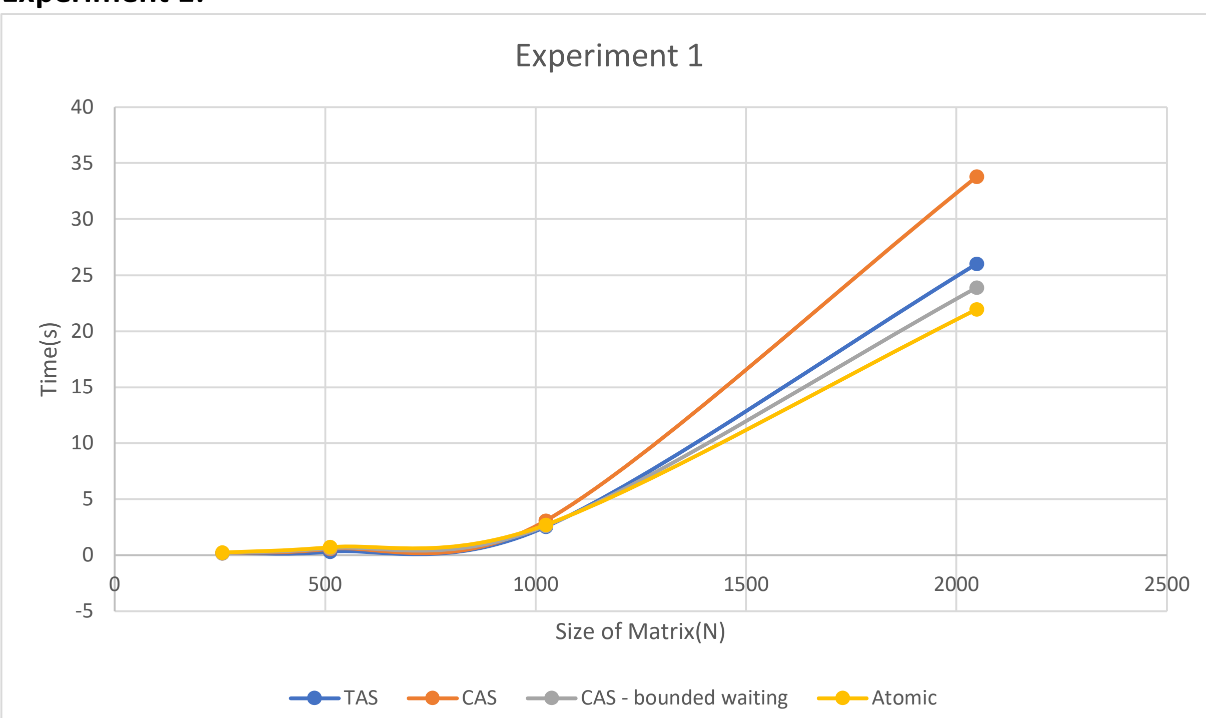
Using Atomic Increment

This was the easiest to implement among all the methods – we simply had to use the inbuilt fetch_and_add function to atomically increase the globally shared counter. The fetch_and_add function as well as the load() function ensure that the access to the counter is atomic – i.e. atmost 1 thread can access it at any instant of time. We just use it and increment it.

```
void *runner(void *param)
{
    while(true)
    {

    struct calcInfo *thread = (struct calcInfo *)param;


        // critical section;


        // Now starts the picking up of the chunk
        int rowStart = counter,load(); //Loading the initial value of the counter
        counter.fetch_add(thread->sizeOfChunks); //Atomically incrementing the value of counter
        int rowEnd = counter.load();//Loading the updated value of counter

        if(rowStart >= thread->totalNumberOfRows)//Checking if we have already crossed the
        {                                         //total number of threads. In this case,
            pthread_exit(0);                      //we can exit this thread
        }
        // End of critical section
```

Graphs

Experiment 1:



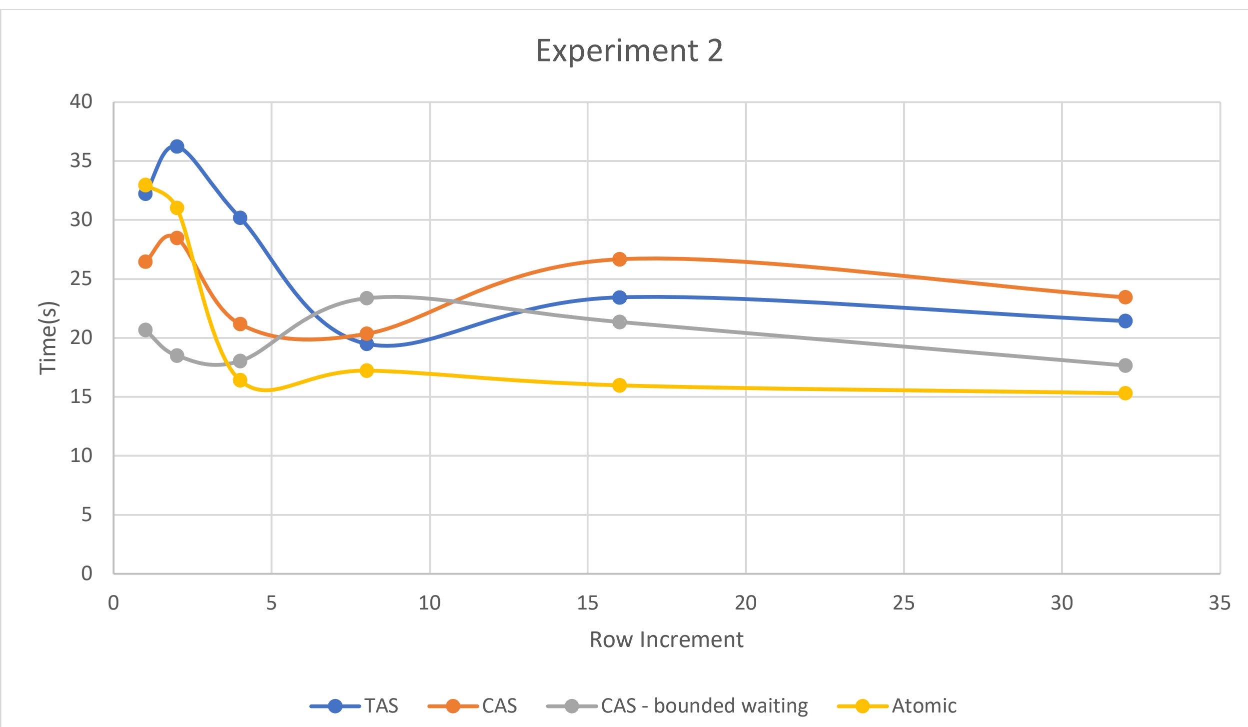
Conclusions derived from the Graph: As we can clearly see, as N – the size of matrix increases, the time taken to compute its square increases.

Also, as far as the methods are concerned, Atomic Method gives the best performance, and compare and exchange gives the worst performance. Compare and exchange gives a bad performance because the thread continuously spins in the busy waiting loop when it has yet not acquired the lock. Whereas – Atomic method gives a great performance because we have not given any explicit condition for locking – it is all done by the fetch_and_add() and load() functions – which also save time.

Why does compare and swap with bounded waiting give a better performance though? The reason is – the threads that are not immediately able to acquire the lock, wait for a short time before retrying, and do not continuously spin in the busy waiting condition.

They enter a bounded waiting loop – by setting waiting[i] = true, and it becomes false when some other thread whose execution is over exits. Thus – the threads that fail to get the lock in first try do get it after a finite time, and they do not spin for an unbounded amount of time.

Graph 2

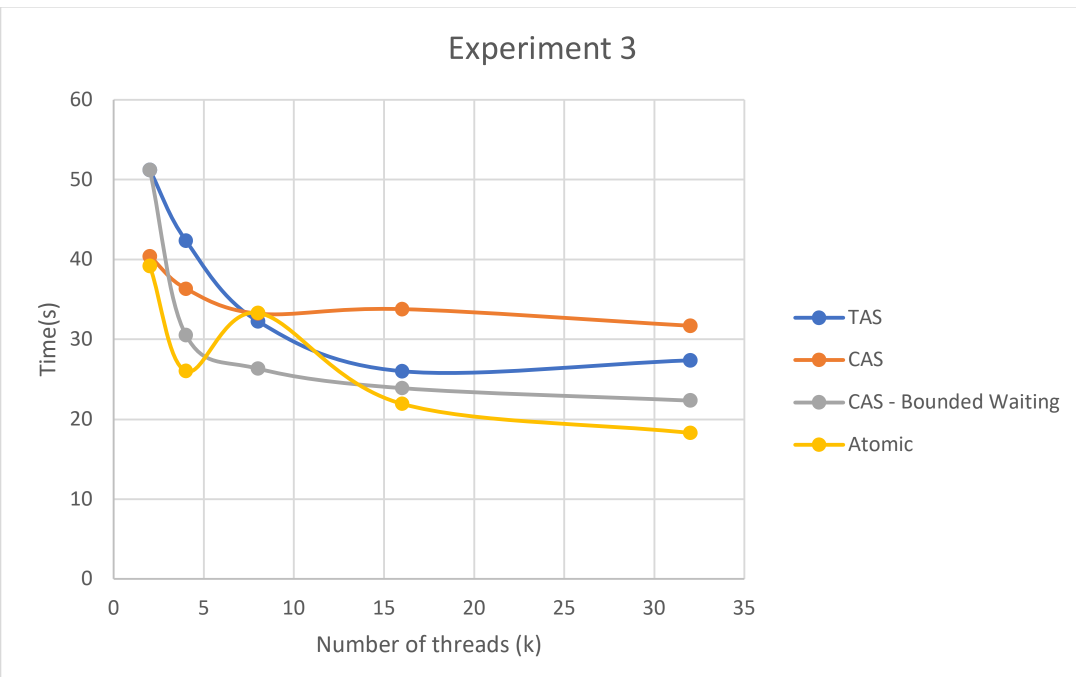


In this graph, we see a decrease in time as the chunk size increases. The reason for this is –

Each thread on an average will get N/k = 2048/16 = 128 rows to compute. If the value of rowInc is small, such as 1, each thread increments the counter only by 1, and computes only 1 row at once. After this – it runs the loop again, enters the critical section again, gets control of counter again, and multiplies the rows.
Instead of this, if each thread is allotted larger chunks at once, we do not have to go over the critical section again and again – and we end up saving a lot of time. One important thing to note is – by increasing rowInc, we are NOT LOSING PARRELISM. Each thread is supposed to compute roughly 128 threads, so as long as we keep rowInc<128, we will not lose Parnellism, as all the threads will have some work to do.

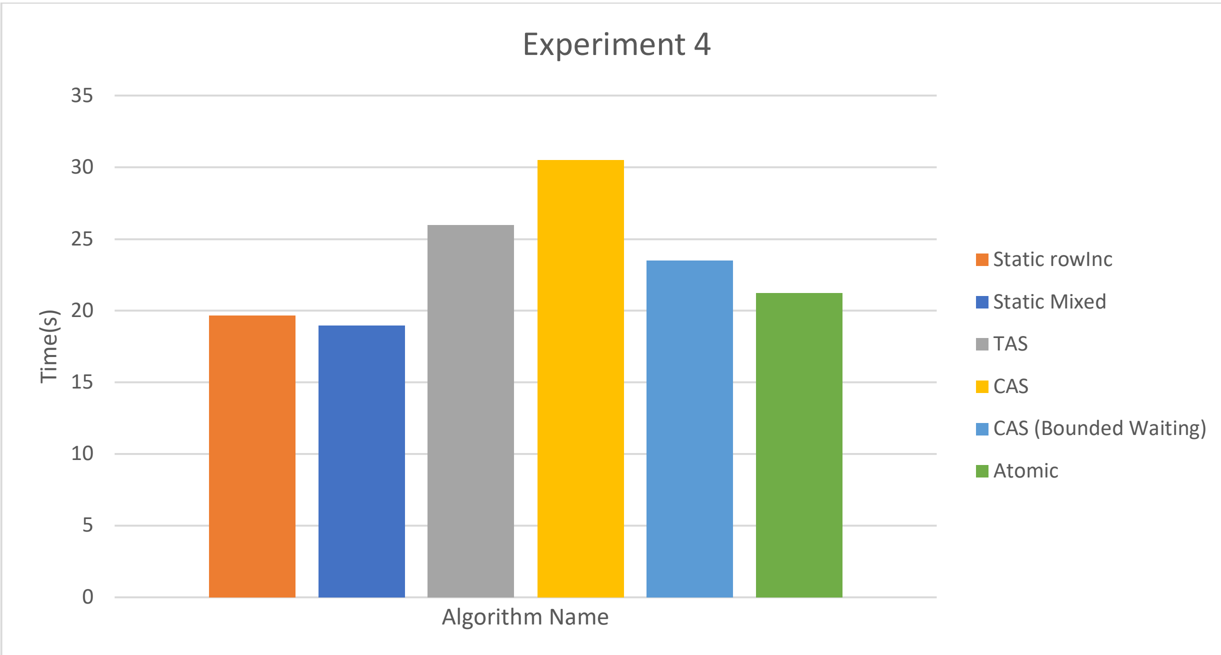
Amongst the methods, the best performance is given by Atomic increment– and the worst performance is given by – normal Compare and Swap. The atomic increment code has a very concise critical section – leading to its improved performance. Since the counter itself is atomic, it ensures that no two threads access the value of counter, and we do not need to write additional conditions for it, which in turn saves time

Graph 3



Conclusions from the graph – We can clearly see that as the number of threads increase, the performance increases – which is pretty obvious now, as the parallelism increases.
The best performance is by atomic, and worst by CAS – we saw the reasons for that earlier. Also, bounded CAS performs better than CAS – whose reason I have mentioned in experiment 1.

Graph 4



As seen from the graph, the order of time taken is as follows –

CAS > TAS > CAS(Bounded Wating) > Atomic > Static rowInc > Static Mixed
The static methods are faster because time is not wasted in contention for the critical section, and in busy waiting. All the threads are equally allocated their work, and hence not much time is wasted once the threads start working. While dynamic methods have their own advantages, in this case – static method gives a better performance.
Amongst the dynamic methods, atomic gives the best performance, and CAS the worst – the reason is discussed before.