Assign_3Report_ES22BTECH11026

04 March 2024 15:32

Hello. Before I start explaining my Code & Graphs, Here are the changes I made to the assignment for Experiment 1:

CHANGES MADE

- Have Calculated time for the number of threads, **N = 256, 512, 1024, 2048.**
- **Didn't analyse** the ode for larger values of N, since it was taking a lot of time(approx. **3.5 minutes** for N= 4096, even more for N = 8192). Allowed by the TA to do so.

EXPLANATION

1. **Dynamic with TAS**

```
38  void* dynamic_TAS(void* arg){
39      intermediary* inner = (intermediary*)arg;
40      int local_row_counter = 0;
41      do{
42          while(flag.test_and_set())
43          ; //do nothing//
44
45          //critical section
46          int starting_row = C;
47          C += rowInc;
48          flag.clear();
49
50          //remainder section
51          for(int i = starting_row; i < starting_row + rowInc; i++){
52              for(int j = 0; j < N; j++){
53                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
54              }
55              local_row_counter++;
56          }
57          if(local_row_counter == N/K){
58              break;
59          }
60      }while(true);
61      return NULL;
62  }
```

(Done earlier in the code)
`atomic_flag flag = ATOMIC_FLAG_INIT;` //This initializes flag with the initial value "clear", meaning it's initially false

Since the inbuilt function **'test_and_set()' is executed atomically**, the first thread which accesses it sets the flag to 1(i.e. **acquires the lock**), & enters the critical section. The other threads meanwhile are stuck in the while loop(**spinning**). One of the threads exits the while loop only once the previous thread exits the critical section & executes **line 48.**

(flag.clear() again sets the value of flag to 0**,** i.e. **releasing the lock)**

This GUARANTEES **mutual exclusion.**

A **'local_row_counter' variable has been set which keeps account of the number of rows each thread has computed**. This has been done to prevent **starvation** and to prevent only some threads computing majority rows while some threads **stuck in the while loop on line 42.**

2. **Dynamic with CAS**

```
39  void* dynamic_CAS(void* arg){
40      intermediary* inner = (intermediary*)arg;
41      int local_row_counter = 0;
42      do{
43          int expected = 0;
44          while(!flag.compare_exchange_strong(expected, 1)){
45              expected = 0;
46          } //do nothing//
47
48          //critical section
49          int starting_row = C;
50          C += rowInc;
51
52          flag = 0;
53
54          //remainder section
55          for(int i = starting_row; i < starting_row + rowInc; i++){
56              for(int j = 0; j < N; j++){
57                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
58              }
59              local_row_counter++;
60          }
61          // cout<<inner->thread_no<<" "<<starting_row<<endl;
62          if(local_row_counter == N/K){
63              break;
64          }
65      }while(true);
66      return NULL;
67  }
```

(Written in the actual Src file)

atomic<int> flag(0); //This initializes flag with the initial value "0", meaning it's initially unlocked.

I learnt something new while executing the inbuilt 'compare_exchange_strong()' function.

The only **difference** between the inbuilt **compare_exchange** function and the **compare_&_swap function** taught in class is that the former returns whether the "**value of the flag equals the expected value(i.e. 0)**" while the latter returns **"the original value of the flag"**.

I had to accordingly manipulate the code on **line 44** to address this ambiguity.

The function is executed atomically. The thread which first accesses the line 44 sees that the value of flag is **0,** & the function compare_exchange_strong() returns false since: **"flag = expected",** so the

thread exits the loop and enters the critical section. It internally sets **"flag = 1",** meaning other threads then keep **spinning** in the while loop.

It sets **"flag = 0"** on line 52 as it exits the critical section, guaranteeing mutual exclusion.

Rest is same as the previous.

3. Dynamic with Bounded CAS

```
39  void* dynamic_Bounded_CAS(void* arg){
40      intermediary* inner = (intermediary*)arg;
41      int local_row_counter = 0;
42      while(true){
43          waiting[inner->thread_no] = true;
44          int key = 1;
45          int expected = 0;
46          while(waiting[inner->thread_no] && key==1){
47              key = !(flag.compare_exchange_strong(expected, 1));
48          }
49          waiting[inner->thread_no] = false;
50
51          //Critical section
52          int starting_row = C;
53          C += rowInc;
54
55
56          int j = (inner->thread_no + 1)%K;
57          while((j != inner->thread_no)  && !waiting[j]){
58              j = (j + 1) % K;
59          }
60
61          if(j == inner->thread_no){
62              flag = 0;
63          }
64          else{
65              waiting[j] = false;
66          }
67
68          //  remainder section
69          for(int i = starting_row; i < starting_row + rowInc; i++){
70              for(int j = 0; j < N; j++){
71                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
72              }
73              local_row_counter++;
74          }
75          if(local_row_counter == N/K){
76              break;
77          }
78      }
79      return NULL;
80  }
```

Improvement over CAS, since it **satisfies the bounded waiting requirement**. CAS involved **busy waiting,** i.e. there was possibility of some threads getting starved. But here that's not the case.

We maintain two data structures:

```
atomic<int> flag(0);  //This initializes flag with the initial value "clear", meaning it's initially false
bool waiting[32];
```

As the threads enter the while loop, they set **"waiting = true"** inside the while loop. The first thread which accesses **line 47** sets the value of **flag to one(i.e acquires the lock**) and exits the while loop. Rest of the threads keep spinning. **The only thing different here** is that as the thread exits the critical section it **checks whether the next thread wants to enter the critical section** and this cycle repeats. Hence, **every** thread gets a chance to execute itself and **enter the critical section within N - 1 turns.**

Later, we will see this does better than CAS in terms of time taken for computation.

4. **Dynamic with Atomic**

```
39  void* dynamic_Atomic(void* arg){
40      intermediary* inner = (intermediary*)arg;
41      int local_row_counter = 0;
42      do
43      {
44          /* code */
45          int my_ticket = ticket.fetch_add(1, memory_order_relaxed);
46          while (my_ticket != turn.load(memory_order_relaxed)) {
47              // Spin until it's our turn
48          }
49
50          // Critical section
51          int starting_row = C;
52          C += rowInc;
53
54          // Exit the critical section
55          turn.fetch_add(1, memory_order_relaxed);
56          for(int i = starting_row; i < starting_row + rowInc; i++){
57              for(int j = 0; j < N; j++){
58                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
59              }
60              local_row_counter++;
61          }
62          if(local_row_counter == N/K){
63              break;
64          }
65          // cout<<"hi from thread "<<inner->thread_no<<endl;
66      }while (true);
67
68
69      return NULL;
70  }
```

```
std:: atomic<int> ticket(0);  //A ticket number for each thread, initialised to 0
std:: atomic<int> turn(0);   //Determines the turn of the thread whose ticket_no matches with turn
```

Broadly, each thread follows the below steps:

Atomically increment ticket to get its unique ticket number.

Spin (busy-wait) until its **ticket number matches the value of another atomic variable called "turn"**.

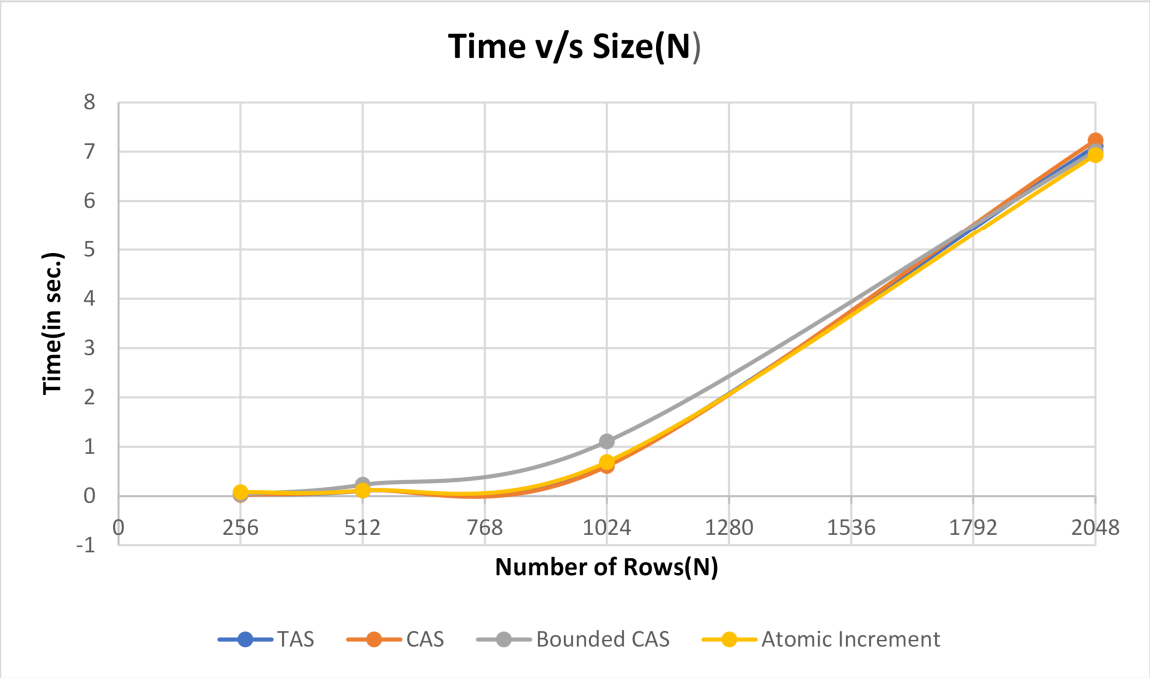
Enter the critical section.

After completing the critical section, **increment "turn"** to allow the **next thread to proceed.**

Unlock is accomplished simply by incrementing the turn such that the next waiting thread(if there is one)can now enter the critical section.

Later, we will see this performs the **best** among the 4 algorithms.

EXPERIMENT 1



N	Time(in seconds)			
	TAS	CAS	Bounded_CAS	Atomic
256	0.026	0.024	0.036	0.079
512	0.109	0.115	0.23	0.113
1024	0.66	0.607	1.108	0.69
2048	7.11	7.23	7.01	6.93

It is evident from the graph that all the **4 algorithms take nearly the same amount of time** to execute the code for the same size "N.

STILL..

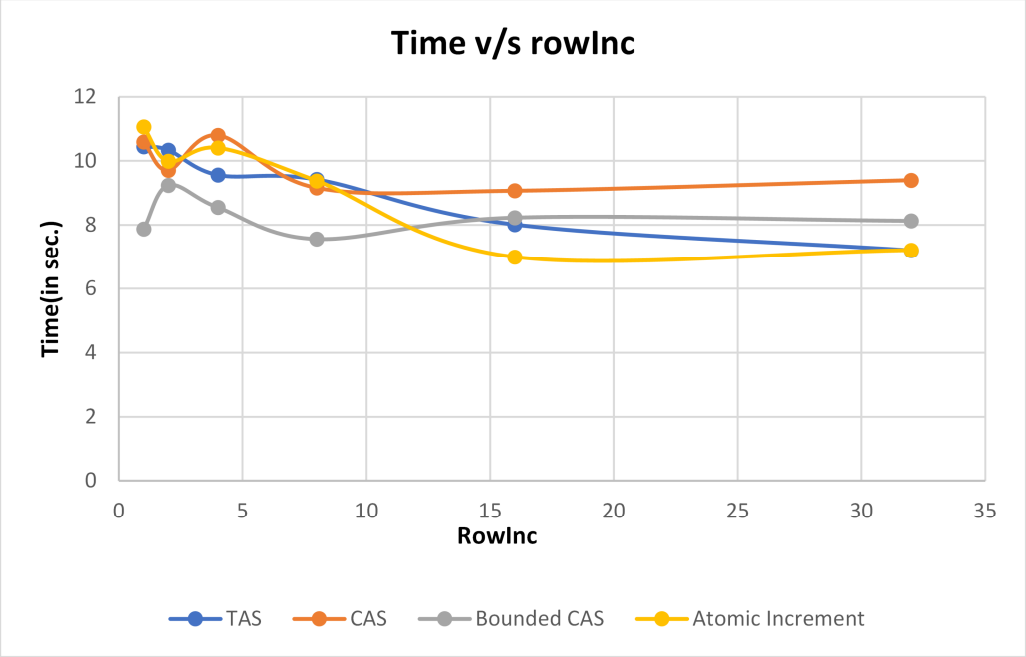
Since we can see from the graph, **CAS takes slightly less time** than the other 3 for **smaller inputs**. We can think of a reason:
The reason why the CAS method might have taken less time to execute compared to the test and set method is that **CAS operates directly on the memory location**, allowing for atomic operations **without** needing to use additional locking mechanisms.
This can result in **better performance**, as CAS avoids the overhead of acquiring and releasing locks.

But for the largest input, i.e. N= 2048, **Atomic does the best.**
We can think why:
Atomic ensures progress for all the threads.(as does Bounded CAS too!)
Once a thread is assigned its ticket value, it will be **scheduled** at some point in the future (once those in front of it have passed through the critical section and released the lock). In our previous attempts, **no such guarantee existed**; a thread spinning on test-and-set (for example) could **spin forever** even as other threads acquire and release the lock.

The Atomic code is also a lot **less complex** than the other 3..

Overall, as the size "N" increases, **it takes increasingly more & more time** for the threads to compute the final square matrix. This trend is **followed by all the 4 algorithms.**

EXPERIMENT 2:



	Time(in sec.)			
rowInc	TAS	CAS	Bounded_CAS	Atomic
1	10.45	10.59	7.87	11.06
2	10.34	9.71	9.24	9.99
4	9.56	10.8	8.55	10.41
8	9.42	9.16	7.56	9.38
16	8.01	9.07	8.23	6.99
32	7.2	9.4	8.13	7.21

This graph doesn't follow a fixed pattern unlike the others**.** But we can **guarantee** one thing from the graph**,** that **rowInc = 1 takes lot more time to execute than rowInc = 32.**

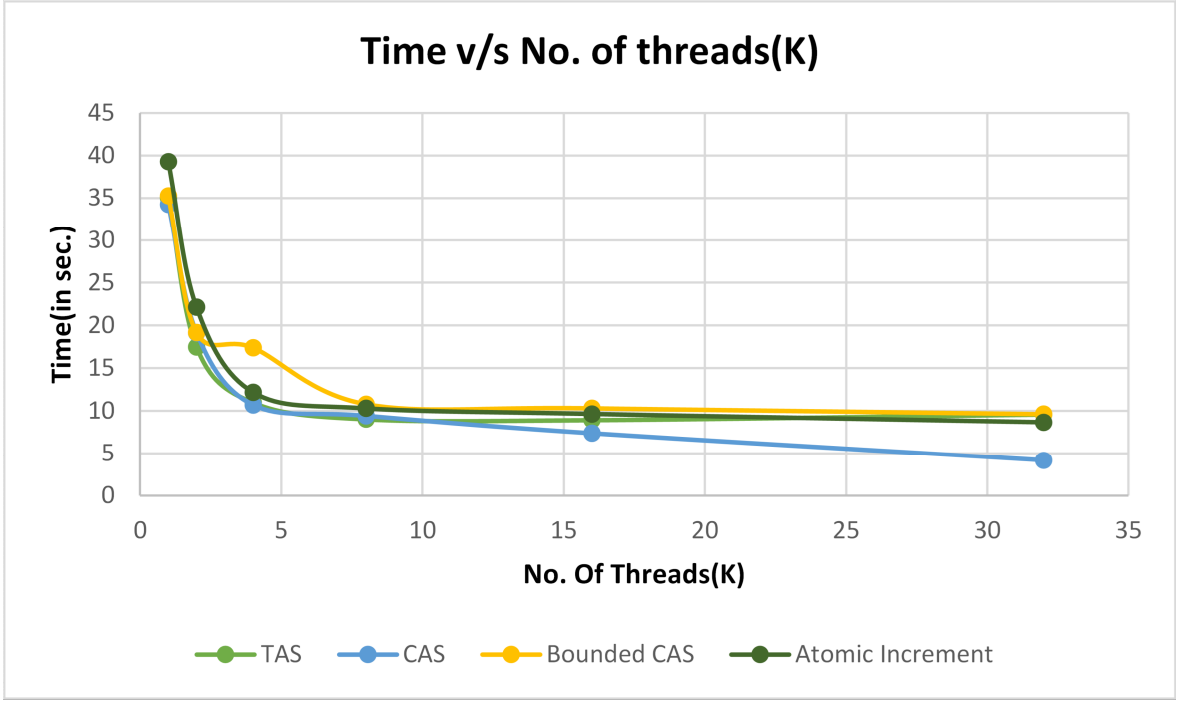
The reason is simple. **Lower the rowInc, lower the time taken by the thread to execute in the**

remainder section, and lower the time it takes to re-enter the loop. It keeps **spinning** then in the while loop until other thread exits its critical section.

However, **as rowInc increases, the overhead incurred due to spinning significantly reduces** & thread spends more time doing effective work(i.e. computing the dot product of the rows).

Rest, all the 4 algorithms take almost equal time to execute.

EXPERIMENT 3

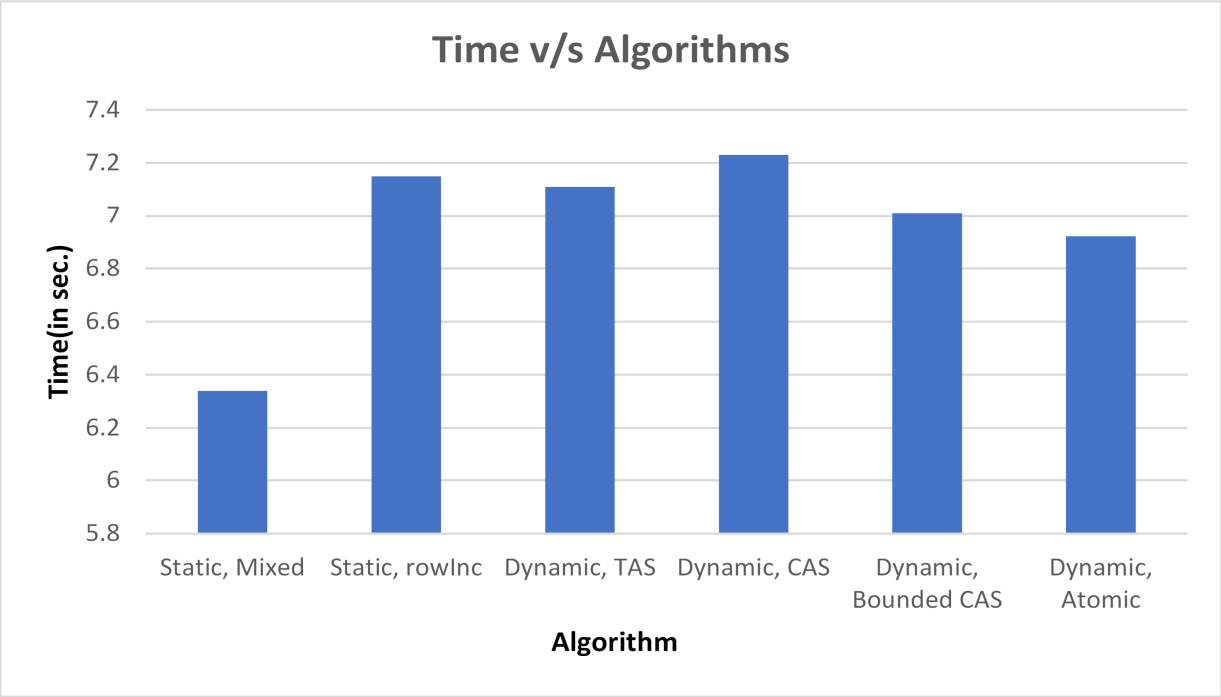


K	Time(in sec.) TAS	CAS	Bounded_CAS	Atomic
1	35.042	34.23	35.23	39.28
2	17.48	19.13	19.18	22.12
4	10.98	10.7	17.39	12.18
8	9.03	9.41	10.78	10.29
16	8.93	7.36	10.3	9.64
32	9.59	4.16	9.61	8.66

Here also, **CAS does better** than the other 3 owing to the same reason discussed In expt. 1.

Also, as the no. of threads increases, the execution time for all the 4 algorithms decreases(which is obvious by now!).

EXPERIMENT 4



STATIC_MIXED IS THE WINNER!!

But Why?
One of the reasons can be **accredited to spinning overhead created in the 4 Dynamic Algorithms**. But in Mixed, **each thread gets an equal workload**, and **no thread sits idle** as each has a set number of rows to work upon unlike in the other 4 dynamic algorithms.

Also for larger inputs, we see that TAS BEATS CAS. One of the simple reasons might be that since CAS has three operands to work with, i.e. expected, new_value, and flag, some overhead might be incurred in working with these.
Meanwhile in TAS, test_and_set is an atomic operation that directly sets a lock variable to a predetermined value.