OS2 CS3523

Name: Swapnil Bag

Roll no: ES22BTECH11034

Aim: This assignment aims to perform parallel matrix multiplication through a Dynamic mechanism in C++.

Low level design

- First the input matrix and output matrices is globally defined to have a max size of 4096
- A structure _th_param_ is created that stores the matrix size N, the thread id (0,1,2….) , row increment value and the max number of threads K
- In main function value of N, K, rowInc and matrix is read through command line argument
- 4 Worker thread arrays are created of size K to create and run the threads.

- **Test and set** : First the thread arguments are dereferenced to extract N, rowInc. Atomic TAS_lock(flag) is initialised as well a row counter count_TAS to 0. In a while loop test_and_set is performed and if the lock is acquired it returns false (previously set value) and sets lock to true (so that other threads trying to enter CS would be in a spin lock), enters critical section where the start row of the current thread is determined from count_TAS and count_TAS is incremented. Then the TAS_lock is cleared and reset after CS and the thread computes the matrix within the designated range in the remainder section.

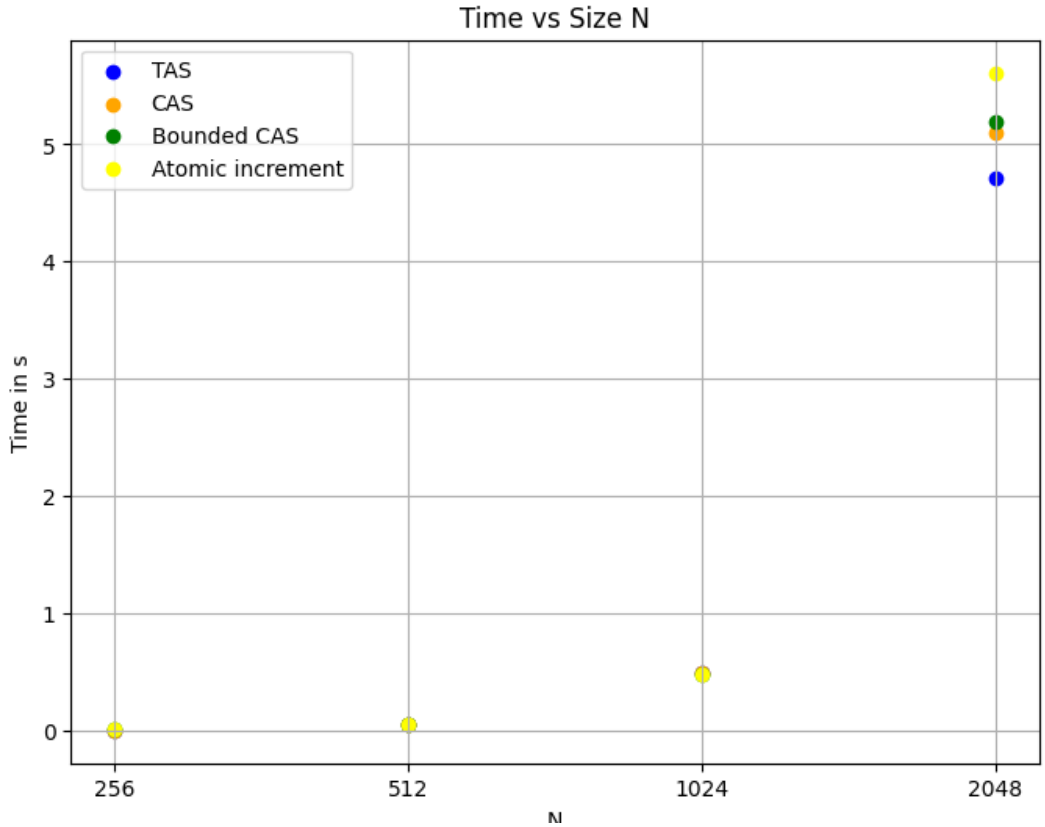
- **Compare and swap** : First the thread arguments are dereferenced to extract N, rowInc. Atomic CAS_lock is initialised to 0. The start variable loads the value of the CAS_lock. In a while loop using compare_exchange_strong the CAS_lock is checked if its value is the start value loaded, if yes then it returns true, increments the CAS_lock value (acts like a row counter) and uses it to compute the range, if false it changes value of start to CAS_lock enters while loop and keeps loading the lock to check if the value has changed or not. Then in the remainder section matrix is computed for specific range of rows

- **Bounded waiting with CAS** : First the thread arguments are dereferenced to extract N, rowInc. waitining array initialised to false, count_bndCas =0, and atomic lock CAS_bnd_lock = false. The thread that is ready is set to true in the waiting array and key set to 0. In a while loop CAS is performed to acquire the lock. After the thread acquires the lock it sets waiting to false and enters CS. The next thread to enter is decided by seeing which thread ready and ensure it's not the same thread that just entered to impose bounded waiting. If some other thread enters then its waiting state is set to false or if the previous thread renters, then the lock is set to false to break out of spinlock. In the critical section the range is determined and remainder section those rows are computed.

- **Atomic Increment**: First the thread arguments are dereferenced to extract N, rowInc. Atomic counter count_atm =0. Here each thread increments the shared counter values using fetch_add, to determines its row range.

- In the main program the threads are created and joined calling their respective functions. The final output is printed to out.txt

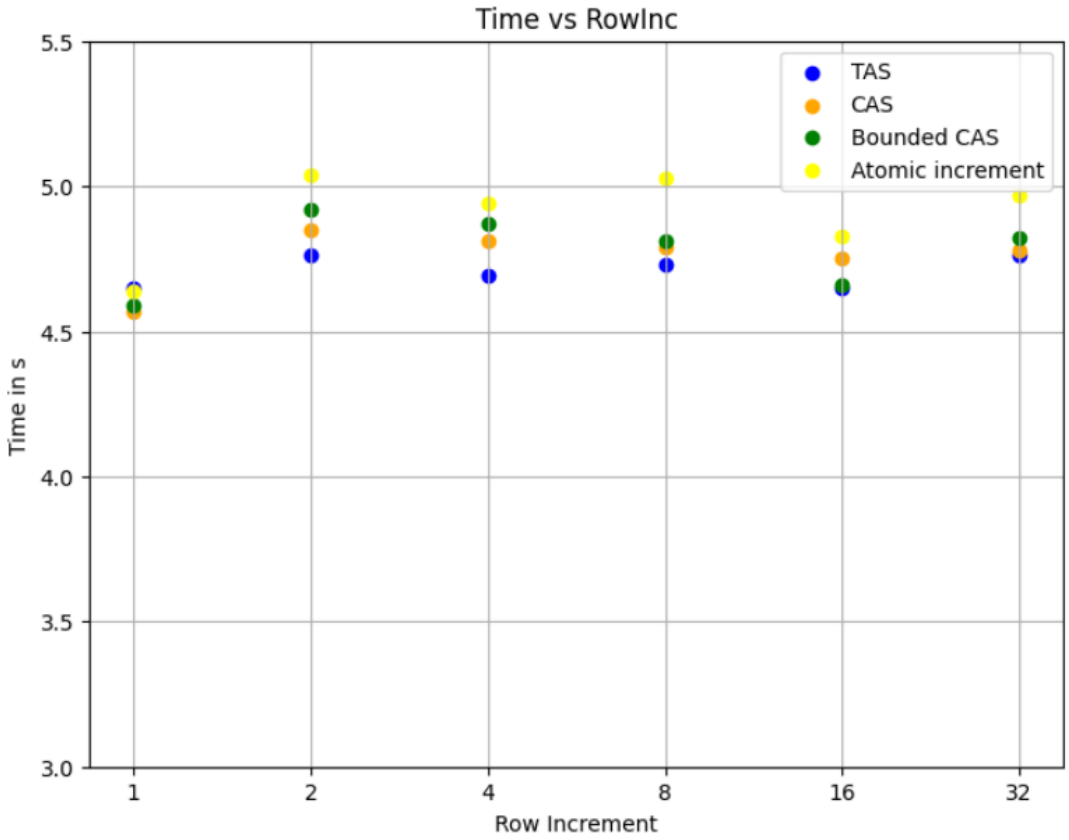
Time vs. Size, N:



	Size N	TAS	CAS	Bounded CAS	Atomic increment
0	256	0.006	0.007	0.015	0.009
1	512	0.057	0.051	0.050	0.049
2	1024	0.498	0.494	0.489	0.487
3	2048	4.710	5.100	5.190	5.610

Here the points appear yellow because the points overlap. The time taken increases on increasing value of N due to higher workload.

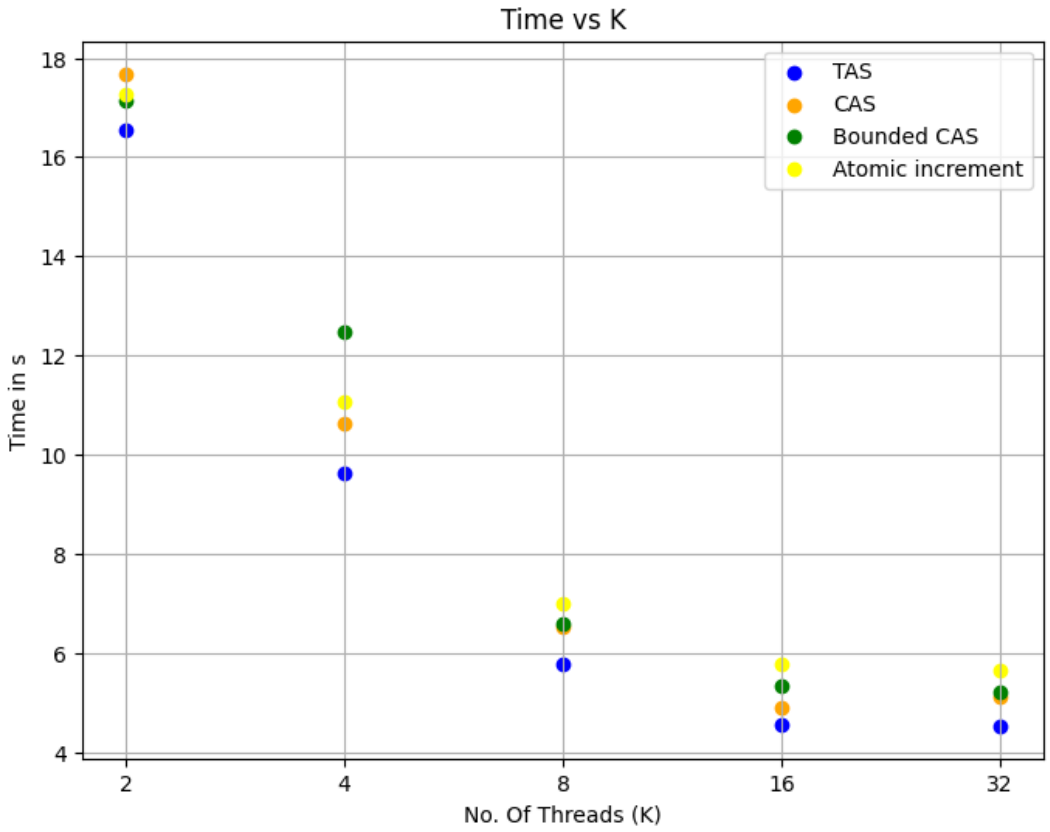
Time vs. rowInc, row Increment



	Row Inc	TAS	CAS	Bounded CAS	Atomic increment
0	1	4.65	4.57	4.59	4.64
1	2	4.76	4.85	4.92	5.04
2	4	4.69	4.81	4.87	4.94
3	8	4.73	4.79	4.81	5.03
4	16	4.65	4.75	4.66	4.83
5	32	4.76	4.78	4.82	4.97

Here as row increment increases the time taken slightly increases and then decreases.

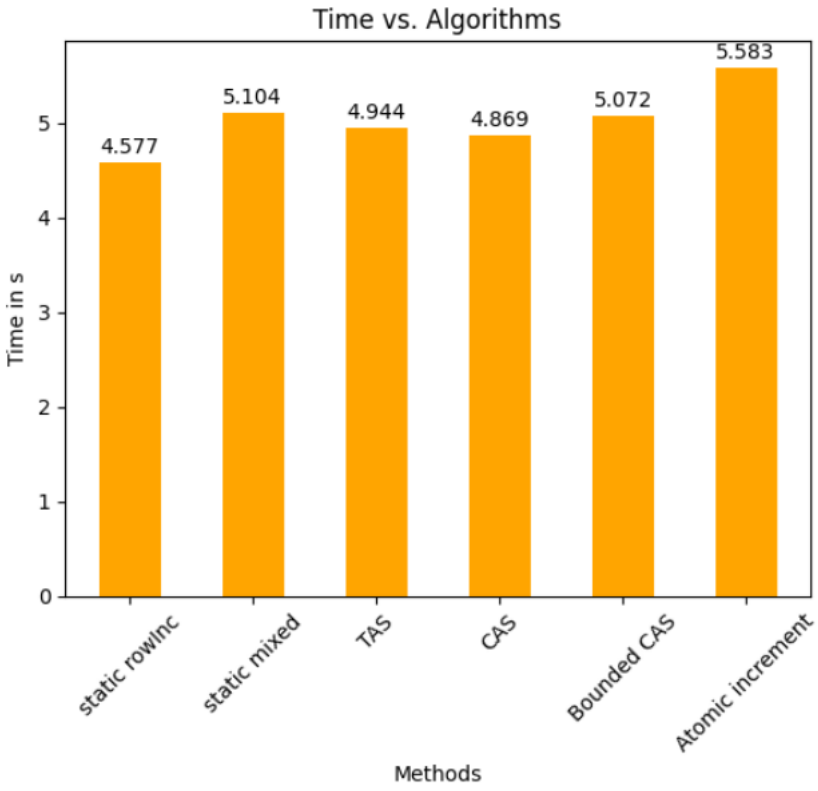
Time vs. Number of threads, K:



	K	TAS	CAS	Bounded CAS	Atomic increment
0	2	16.55	17.68	17.14	17.27
1	4	9.64	10.63	12.47	11.07
2	8	5.77	6.51	6.60	7.01
3	16	4.56	4.91	5.35	5.76
4	32	4.53	5.12	5.21	5.65

Here the time decreases as the value of K increases as more threads are available to compute the rows parallelly

Time vs. Algorithms



The static method takes the least time probably because there's no need for locking and synchronisation tasks and atomic lock takes the maximum amount of time because of overhead incurred in atomic instruction execution