Implementing Multi Reader Multi Writer Register

Aaryan, CO21BTECH11001

Note:
The value of λ is taken as 2.0 for all the executions, so that the program would run in a reasonable amount of time.

Program Design:

1. As mentioned in the assignment, it is assumed that the class `StampedValue` serves as an atomic MRSW register.
   The methods `read()` and `write()` of the class `AtomicMRMWRegister` are implemented in accordance with Figure 4.14 of the textbook.

2. To ensure that all the threads produce a different sequence of random numbers , the random number generator is seeded with a value equal to the product of the thread id and the current time. This also ensures that different random numbers are generated in different executions of the program.

3. A global array named `operationTime` is made where `operationTime[i]` stores the sum of times (in microsecond) taken by read/write operations performed by thread `i`. Here an array is used instead of a single variable to achieve correctness and synchronization. Using a single variable would cause issues if multiple threads are trying to increment it at the same time.

4. When all threads have completed, the program sums all the values of the array `operationTime` and divides it with the number of read/write operations performed, which is equal to the product of number of threads and the variable `numOps`. Hence the average operation time is calculated.

5. Upon completion of the program, it outputs a log file containing the sequence of operations with their timestamps.

- **Impact of the capacity (number of threads) on average operation time:**

The data obtained from custom implementation of MRMW register:

Custom MRMW Register Average Operation Time (Micro Seconds)						
Capacity	Time1	Time2	Time3	Time4	Time5	Average Time
2	20.9797	21.006	21.4909	21.1718	21.0538	21.14044
4	21.8852	21.263	21.7532	21.2878	21.7971	21.59726
6	21.9123	21.8202	22.2535	22.0504	22.3036	22.068
8	22.166	22.3346	22.46	22.1727	22.1374	22.25414
10	22.7886	23.1574	22.7213	22.8777	23.2071	22.95042
12	23.2119	23.2795	23.0066	22.8243	22.8537	23.0352
14	23.3591	23.54	23.5277	26.1036	23.6291	24.0319
16	23.9021	26.2618	24.2115	23.8844	28.2155	25.29506

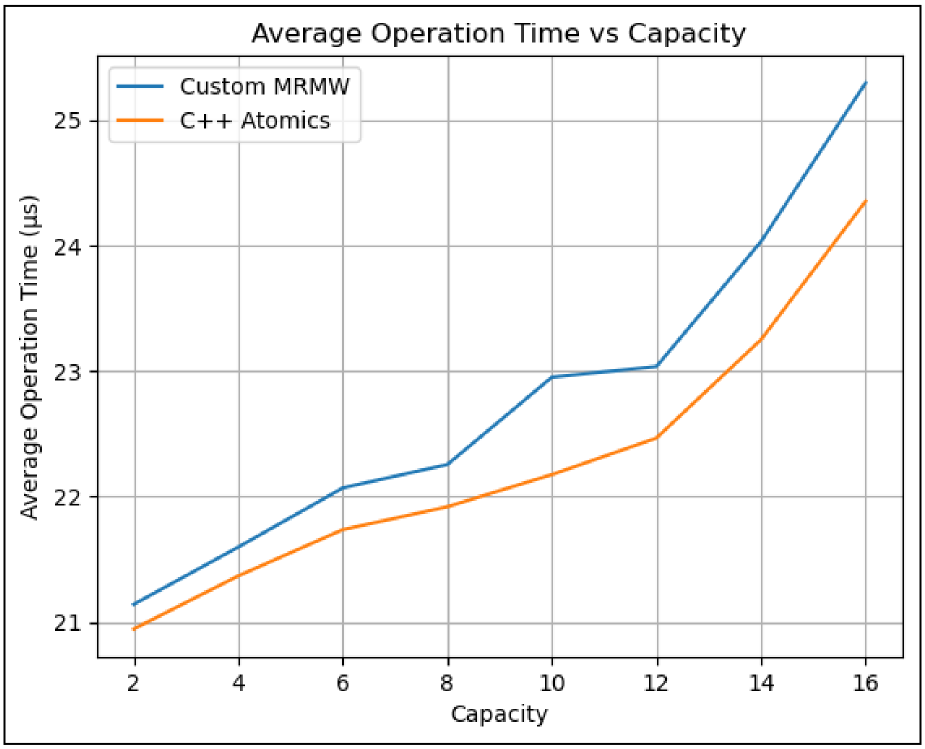
As we can see, the average operation time increases with increasing capacity. The reason is that read/write operations in the custom implementation takes $O(capacity)$ time to execute. Therefore, when capacity is increased, the average operation time increases by a few microseconds.

The data obtained from C++ atomics MRMW register:

C++ Atomics Average Operation Time (Micro Seconds)						
Capacity	Time1	Time2	Time3	Time4	Time5	Average Time
2	20.8467	20.8764	20.984	21.0245	20.9878	20.94388
4	21.3445	21.4848	21.4491	21.3373	21.2219	21.36752
6	21.6697	21.6225	21.8397	21.7941	21.7549	21.73618
8	22.0514	21.7796	21.9837	21.9304	21.8475	21.91852
10	22.2978	22.2287	22.3287	21.7056	22.3086	22.17388
12	22.3374	22.3823	22.7051	22.2721	22.6306	22.4655
14	22.8357	22.7181	23.0611	24.8132	22.8245	23.25052
16	23.4764	27.7889	23.5723	23.4603	23.4622	24.35202

As we can see, the average operation time increases with increasing capacity. The possible reason is that the synchronization overhead increases when the number of threads (particularly writer threads) increases. To ensure synchronization, the implementation of C++ atomics would spend more time when the number of threads are more. Therefore, when capacity is increased, the average operation time increases by a few microseconds.

Here is the plot for the same:



- **Impact of the $numOps$ on average operation time:**

The data obtained from custom implementation of MRMW register:

Custom MRMW Register Average Operation Time (Micro Seconds)						
numOps	Time1	Time2	Time3	Time4	Time5	Average Time
1000	24.5927	24.2728	24.0686	24.5771	24.8532	24.4729
2000	24.4734	24.2556	24.3571	24.0707	24.2342	24.2782
3000	24.6605	24.1075	24.1877	23.5645	24.3907	24.1822
4000	24.2518	23.9353	24.2424	23.6851	24.0821	24.0393
5000	24.0761	23.9016	24.3881	23.9671	23.8860	24.0438

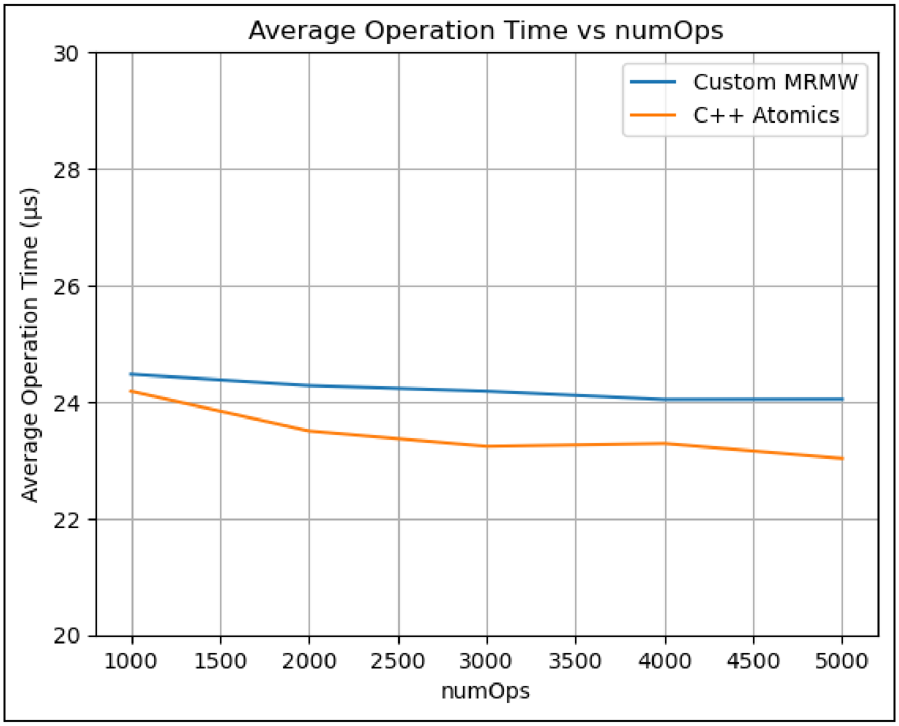
We can see that the average operation time remains almost the same (*changes in order of* 0.1 *microseconds, which is insignificant compared to the average time taken to execute one operation*) when $numOps$ is increased. The reason is that the time taken to execute one read/write operation is dependent on the number of threads ($O(capacity)$), which is constant.

The data obtained from C++ atomics MRMW register:

C++ Atomics Average Operation Time (Micro Seconds)						
numOps	Time1	Time2	Time3	Time4	Time5	Average Time
1000	24.4164	24.1424	24.1636	24.034	24.1534	24.18196
2000	23.8709	23.7492	23.3599	23.0706	23.4416	23.49844
3000	23.2962	23.2913	23.6541	22.8993	23.0312	23.23442
4000	23.4176	23.3158	23.47	23.3254	22.8917	23.2841
5000	22.9695	23.582	23.2257	22.7076	22.6604	23.02904

We can see that the average operation time remains almost the same (*changes in order of* 0.1 *microseconds, which is insignificant compared to the average time taken to execute one operation*) when $numOps$ is increased. The possible reason is that the time taken for the inbuilt implementation of C++ atomics, which ensures synchronization among threads (particularly the writer threads), is primarily dependent on the number of threads it is “handling”. Since the number of threads are constant in this case, the average operation time remains almost constant.

Here is the plot for the same:



Note that in both of the cases above, the time taken by C++ atomics implementation is always lower than the custom implementation.
The reason being that the custom implementation is a relatively inefficient implementation. It takes a linear amount of time for a read/write operation to execute.
However, C++ atomics implementations, which are implemented by experienced programmers, are definitely more efficient than this particular implementation.