Assignment 3 Report - CS5300
Implementing Multi Reader Multi Writer Register

Thupten Dukpa - ES20BTECH11029

StampedValue class

This class is the implementation of Fig 4.10 from the book. The StampedValue is a data structure that associates a value of type T with a timestamp, allowing for atomic read and write operations with timestamps.

It contains two attribute variables: *stamp* of long datatype and *value* of a custom type. There is a static method $max()$ which compares two StampedValue objects and returns the object with the greater stamp value.

AtomicMRMWRegister class

This class is the implementation of Fig 4.14 from the book. The class variable for this implementation is a vector of atomic StampedValue of T datatype called *a_table*.

Its constructor initializes *a_table* with *capacity* elements, each containing a $StampedValue$ of type T initialized with the *init* value.

The $write()$ function is used to write a new value to a specific element in an AtomicMRMWRegister. Each thread is associated with a unique identifier and uses this identifier to determine its position in the register. The thread's unique identifier is obtained using *thread_id_map* which is a global map. A maximum timestamp value is determined by iterating through all elements in the register. The specified value is written to the register with an updated timestamp to ensure it is recorded as the latest value.

The $read()$ function is used to read the latest value from an AtomicMRMWRegister. It retrieves the value associated with the maximum timestamp from the elements within the register, ensuring the most recent value is read. It initializes a max variable with a minimum timestamp and iterates through all elements in the register to find the one with the maximum timestamp. The value associated with the maximum timestamp is returned as the result.

testAtomic

This function tests the behavior and performance of an AtomicMRMWRegister in a multithreaded context. It simulates concurrent read and write operations on the register, records operation details, and measures execution times. The function performs the following steps:

- Maps the global unique ID of each thread to a range from 0 to capacity-1 in *thread_id_map*.
- Initializes an output log file for recording operation details.
- It uses the random library to generate two types of values from the uniform distribution from [0, 1] and exponential distribution of parameter λ.
- A for loop iterates for *numOps* number of times performing either read or write action randomly.
- Records the request time, action, and thread information in the output file.
- Performs the operation on the AtomicMRMWRegister (read or write) and logs the result.
- Records the completion time of the operation.
- Simulates additional work by sleeping for a random duration(in millisec).
- Accumulates execution times in the global atomic *totalTime* variable to measure performance.
- Closes the output file when testing is complete.

There is also the *testAtomic*2() function which is similar to *testAtomic*() but performs read and write operations on the inbuilt atomic variable.

main method

The following variables are declared globally:

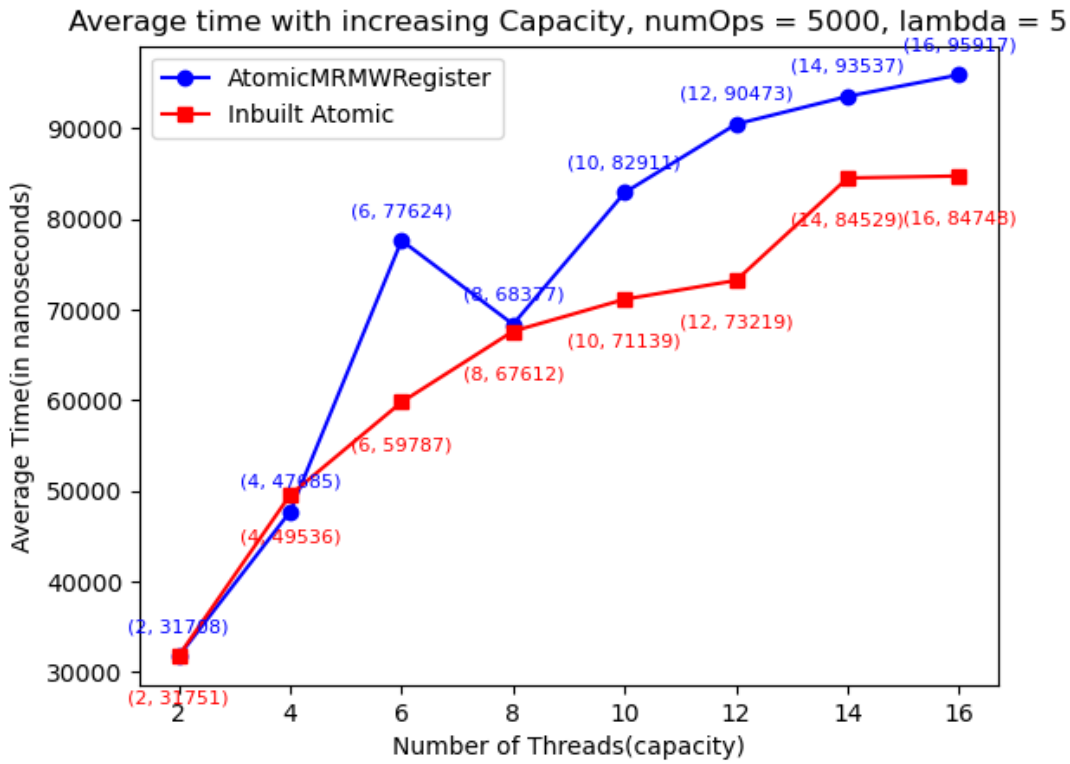
- Input variables: capacity, numOps, lambda.
- thread_id_map: maps the unique ID of a thread to a range from 0 to capacity-1
- mutex mutx: a lock that is used when writing to LogFile.txt.
- totalTime: an atomic variable to store the total time taken by all threads for all operations
- output_file: output file handle for writing the output.

In the main() function, the inputs are read and an object of class AtomicMRMWRegister or a variable of inbuilt atomic module is initialized. A for loop creates n threads each of which executes the testAtomic() function or testAtomic2() function(in case of testing for inbuilt atomic register). It waits for all threads to finish before joining them and finally calculates the average time taken per operation per thread outputs them in a text file.

The graphs obtained after testing are displayed in the next pages.

Impact of average time with increasing Capacity

$numOps = 5000$, $\lambda = 5$

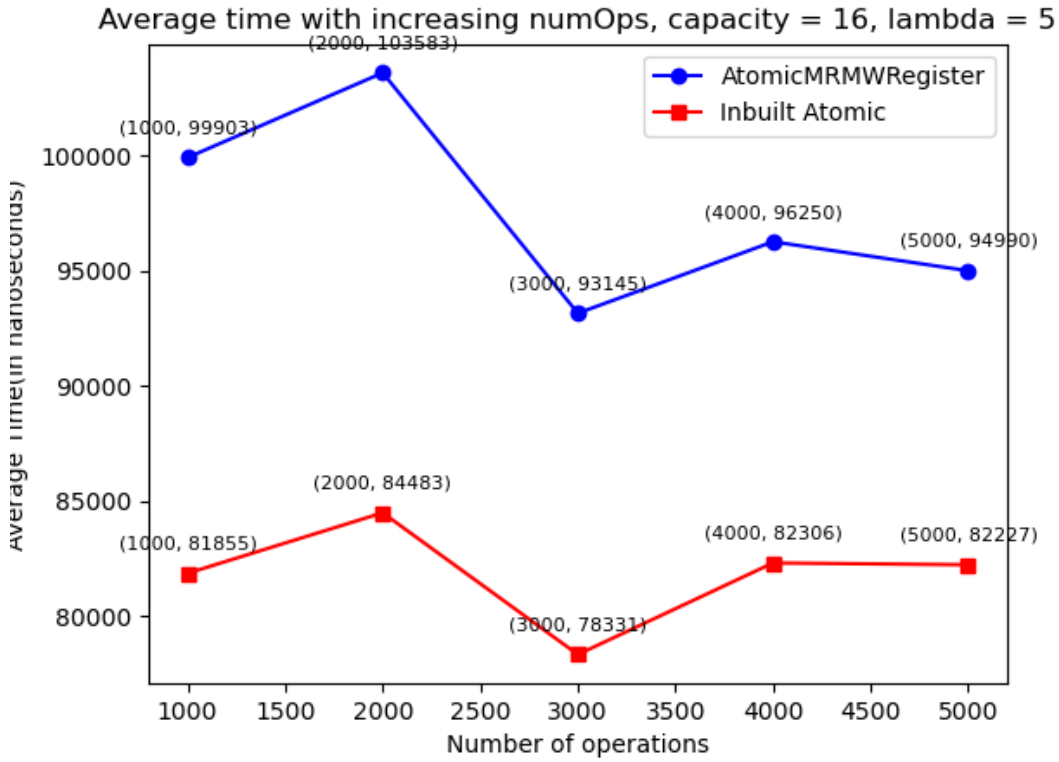


capacity	AtomicMRMWRegister	Inbuilt atomic
2	31708	31751
4	47685	49536
6	77624	59787
8	68377	67612
10	82911	71139
12	90473	73219
14	93537	84529
16	95917	84748

The average time(in nanoseconds) was calculated as the total time taken divided by the product of the number of threads and number of operations which is given as output and was averaged over 5 runs each time. As seen from the graph, the average time taken for both implementations increase with increase in number of threads. Both have similar performance when number of threads < 6, but the inbuilt atomic performs slightly better when number of threads $>= 6$.

Impact of average time with increasing numOps

capacity $= 16$, $\lambda = 5$



numOps	AtomicMRMWRegister	InbuiltAtomic
1000	99903	81855
2000	103583	84483
3000	93145	78331
4000	96250	82306
5000	94990	82227

In this case, the average time taken for both implementations increases slightly with increase in number of operations and then have a slight decreasing trend with their best performance when numOps = 3000. Each value was averaged over 5 runs.