CS5300 - Parallel & Concurrent Programming
Assignment 3
Vaishnavi W AI20BTECH11025

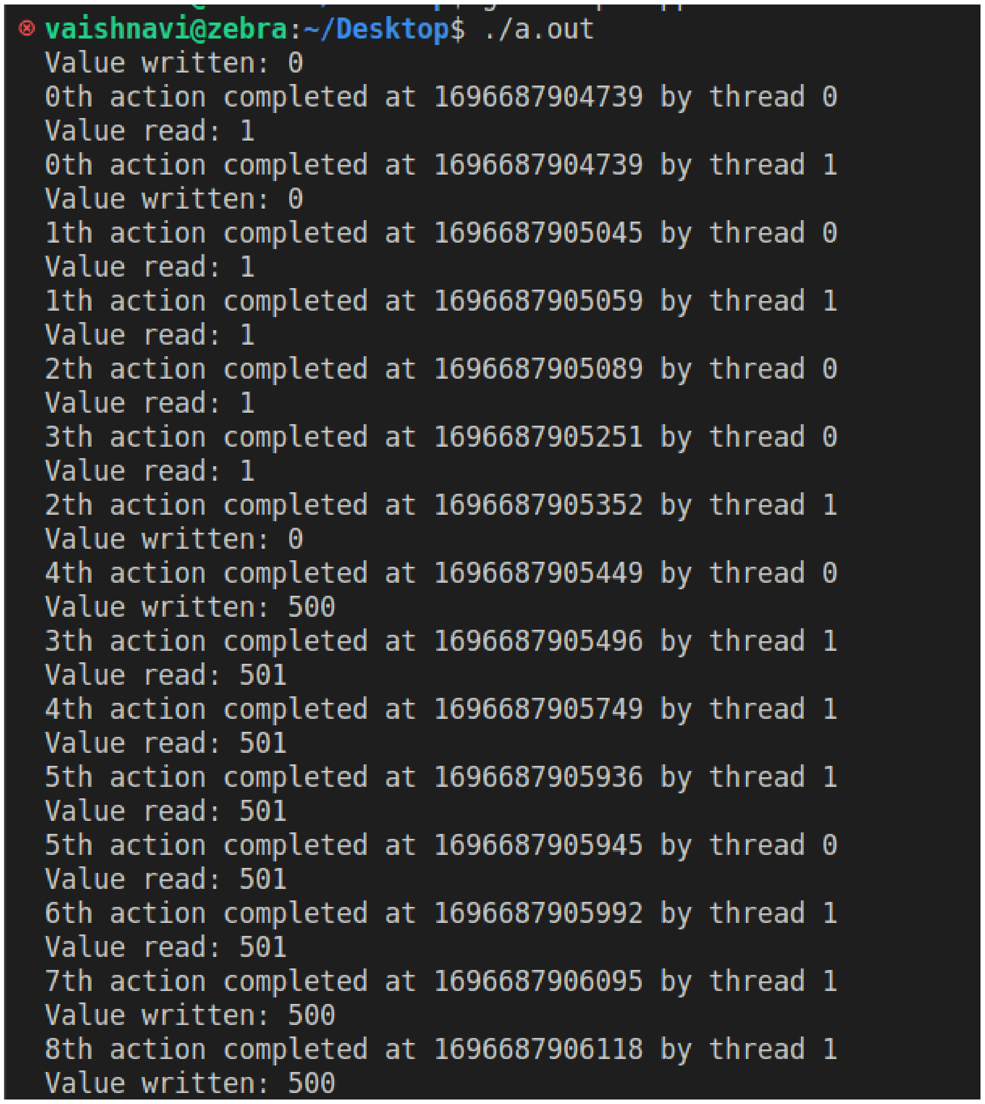
Program Design

- A struct StampedValue is created to replicate the StampedValue class from the textbook. It has two values: long stamp and int value. It has a default constructor, to help declare an array of StampedValues for each of the threads. It can also be initialized with stamp = 0 and a declared init value. It can later be filled with values specified in write functions.
- a_table: An array of 16 (the maximum possible thread number) StampedValues is declared.
- write function: The max_stamp value is initialized to -1. All the stamp values of the other threads are checked for the maximum stamp. A StampedValue with stamp one greater than the obtained maximum stamp along with the write value is written to the a_table.
- read function: A temp StampedValue is created with stamp -1. As with the case of the write function, maximum stamp is obtained. The value corresponding to the StampedValue that has the maximum stamp is returned by the function.
- testAtomic function: action is a random double number between 0 and 1. If action < p read is performed by the thread, else write of a value is performed.
- The value of p is chosen to be 0.5 to have an equal number of read and write operations by the threads.
- Function 'random_expo' is used to sample a random number from an exponential distribution.
- The time for the operations is measured using the *chronos* library. After each operation, the time is stored in the thread's thrTimes_enq and thrTimes_deq vectors. These vectors are not atomic as each of the memory locations is accessed by a single thread, thus no conflicts arise.
- After each operation, a thread is put to sleep for a time generated from an exp distribution.The objective of having these time delays is to simulate that these threads are performing some complicated time consuming tasks.
- The final average time is found by summing up all the times taken by all the threads by capacity and numOps.

Atomic inbuilt implementation:

- A global atomic int variable 'a' is declared.
- A read by a thread is performed by using the atomic.load() operation.
- A write by a thread is performed by using the atomic.store(value) operation.

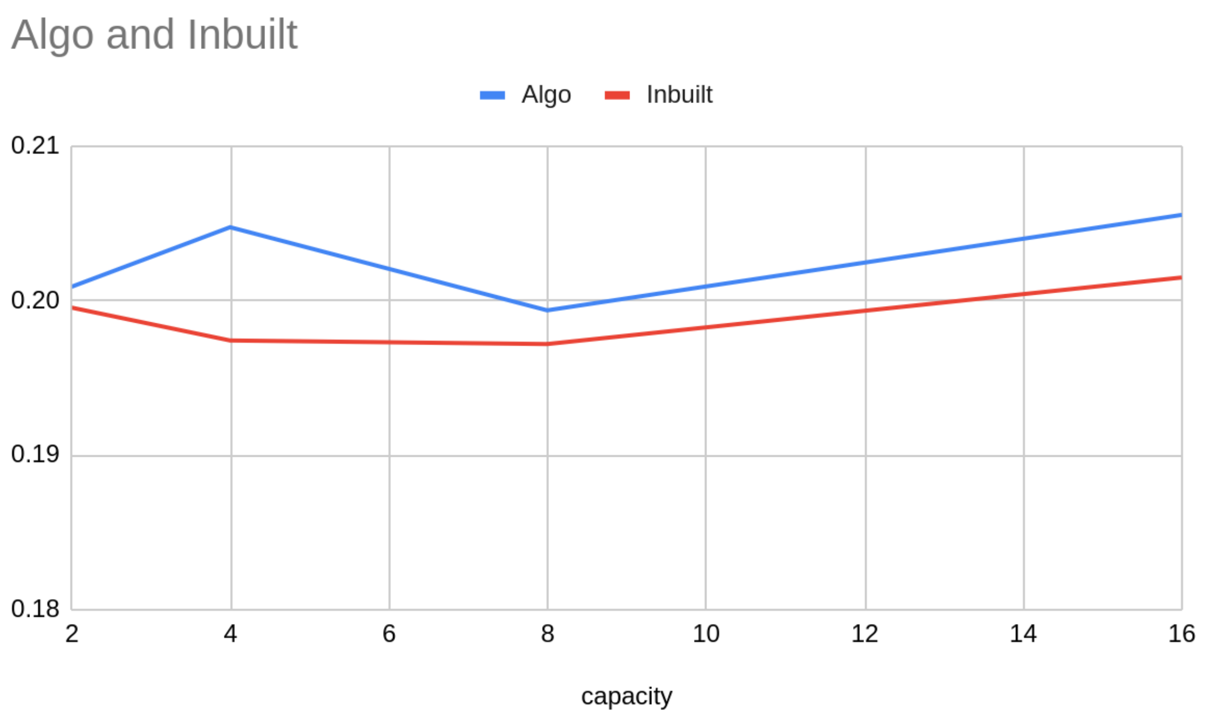
Sample log file obtained by using 2 threads



```
⊗ vaishnavi@zebra:~/Desktop$ ./a.out
  Value written: 0
  0th action completed at 1696687904739 by thread 0
  Value read: 1
  0th action completed at 1696687904739 by thread 1
  Value written: 0
  1th action completed at 1696687905045 by thread 0
  Value read: 1
  1th action completed at 1696687905059 by thread 1
  Value read: 1
  2th action completed at 1696687905089 by thread 0
  Value read: 1
  3th action completed at 1696687905251 by thread 0
  Value read: 1
  2th action completed at 1696687905352 by thread 1
  Value written: 0
  4th action completed at 1696687905449 by thread 0
  Value written: 500
  3th action completed at 1696687905496 by thread 1
  Value read: 501
  4th action completed at 1696687905749 by thread 1
  Value read: 501
  5th action completed at 1696687905936 by thread 1
  Value read: 501
  5th action completed at 1696687905945 by thread 0
  Value read: 501
  6th action completed at 1696687905992 by thread 1
  Value read: 501
  7th action completed at 1696687906095 by thread 1
  Value written: 500
  8th action completed at 1696687906118 by thread 1
  Value written: 500
```

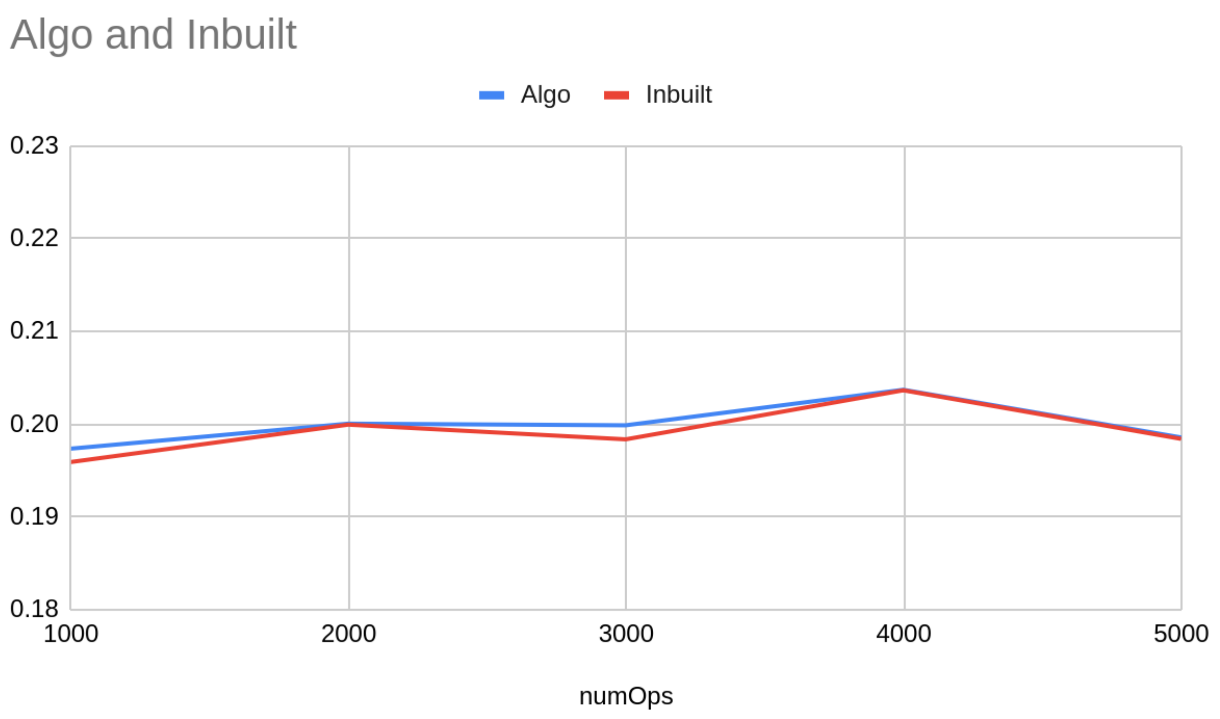
The Graphs obtained and the corresponding analysis

1. Impact of average time with increasing Capacity:



From the graph, we observe that the inbuilt algorithm was performing slightly better than the implemented algorithm. This increased average time for read and write by the algorithm can be attributed to the read of the entire a_table stamp values to find the maximum stamp, both for read and write. This process of finding the maximum stamp can be the overhead for the implemented algorithm.

2. Impact of average time with increasing numOps:



We observe that the average time taken by a thread to perform an operation doesn't depend upon the numOps. It is almost the same, as seen in the graph. Of Course the total time increases linearly with it as we are increasing the number of reads and writes by each of the threads. It was evident by the increasing time taken during execution of the program with the numOps.