Aayush Kumar
CO21BTECH11002
Assignment 2

CLQ:

Coarse Grained Locking Queue uses locks for making enqueue and dequeue operations mutually exclusive. In the enqueue function, each thread acquires a lock before reading/modifying the value of tail. Then it checks whether the queue is full or not. A value is enqueued accordingly and the lock is released. Similarly in the dequeue function a lock is acquired before reading/modifying the value of head. A value is dequeued depending on whether the queue is empty or not. Then the lock is released.

NLQ:

Non Locking Queue uses atomic operations for making enqueue and dequeue operations mutually exclusive. In the enqueue function, each thread uses atomic operations to read and modify the value of tail. Then it atomically sets the value at index tail to the value to be enqueued. Similarly in the dequeue function, each thread uses atomic operations to read value of tail. Then it dequeues the first non zero value from the queue and atomically sets the value at that index to 0.

testThread() function:

This function is called by each thread. It simulates enqueue and dequeue operations. It also calculates the time taken by each thread to complete numOps operations and stores the enqueue, dequeue and total time taken by each thread in enqTime, deqTime and thTime respectively. To ensure that the threads are not synchronized, each thread sleeps for a random time after each operation. The random time is generated using exponential distribution with mean lambda. The random number generator is seeded with the thread id and time(NULL). Hence, each thread has a different random number generator. The random number generator is used to generate a random number between 0 and 1. If the random number is less than rndLt, then enqueue operation is performed, else dequeue operation is performed. The value to be enqueued is generated using uniform distribution between 1 and 1000000. The time taken by each operation is calculated using the gettimeofday() function.

main() function:

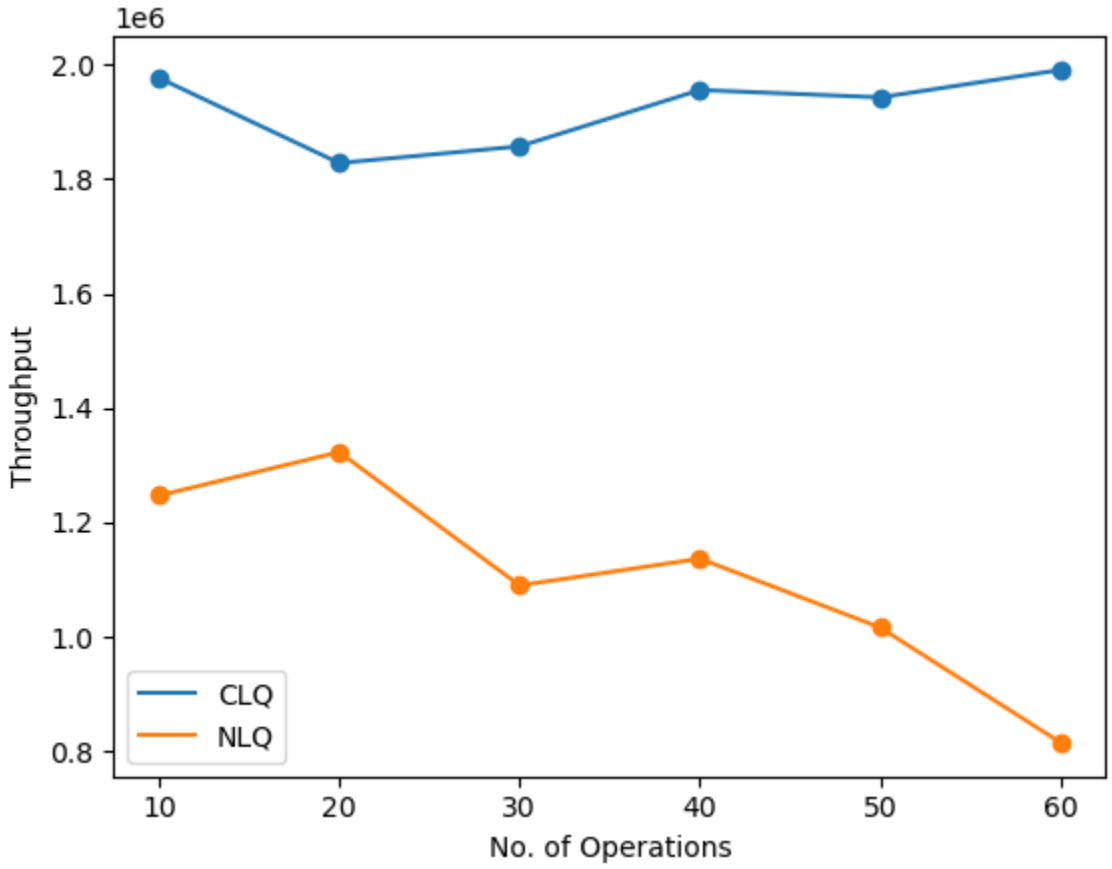
The main function reads the input from the file and creates n threads. It then waits for all the threads to finish and then computes the average time taken by each thread to complete numOps operations, average time taken by each thread to complete all enqueue operations, average time taken by each thread to complete all dequeue operations and throughput. It also writes the average time taken by each thread to complete numOps operations to the output file using the computeStats() function.

Note:

rndLt is taken 0.7 for NLQ to reduce the probability of the program getting stuck in an infinite loop because of all threads calling the dequeue function on an empty queue. To keep consistency, it's 0.7 for CLQ as well.

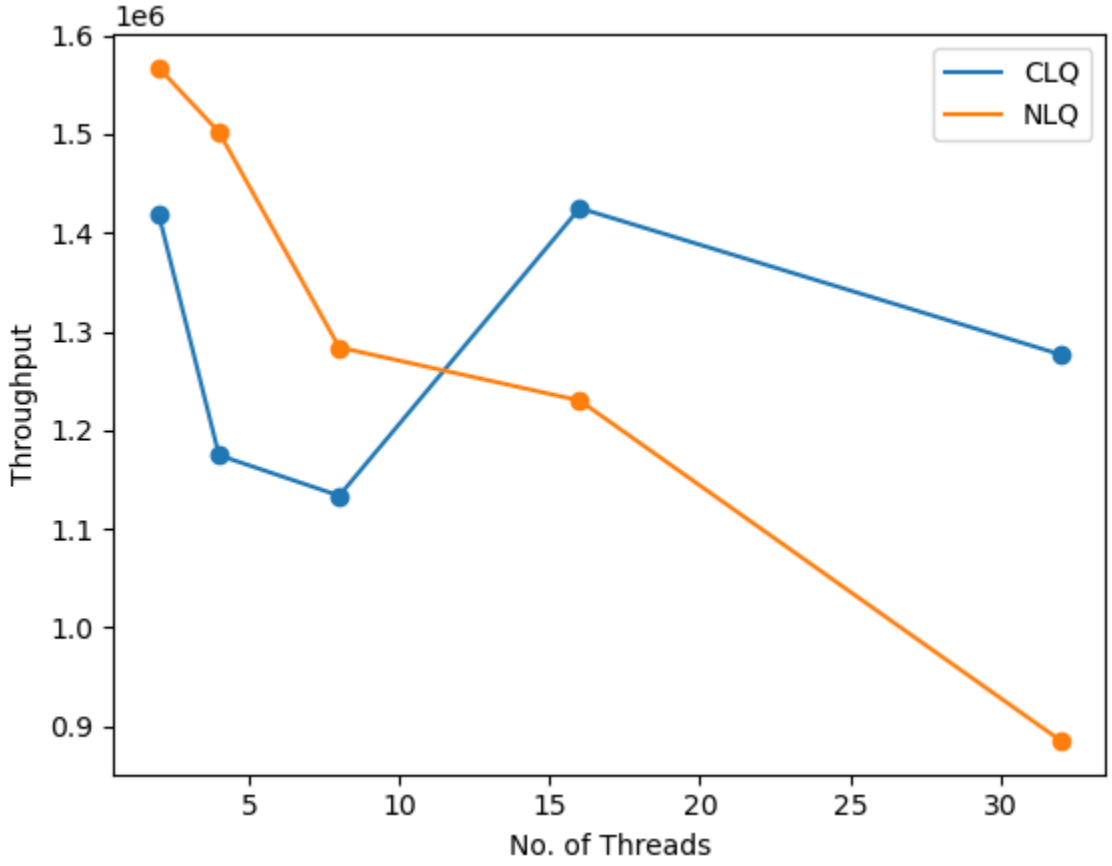
Results:

Throughput vs number of operations (keeping n = 16, rndLt = 0.7, lambda = 5)



- As the number of operations increase, throughput for CLQ almost remains the same. This is because both enqueue and dequeue in CLQ take constant time and hence the total time increases with the same factor.
- For NLQ, there is a decrease in throughput with increase in operations. This is because as more elements are added, the value of tail will increase and hence for dequeuing, more elements will have to be checked.
- NLQ in general is faster than CLQ. A possible reason for this can be slow dequeue operations in NLQ.

Throughput vs number of threads (keeping numOps = 60, rndLt = 0.7, lambda = 5)



- As the number of threads increases, throughput for CLQ almost remains the same. There is a slight decrease in the beginning. This can be because of more waiting time for threads to acquire the lock.
- For NLQ, there is a decrease in throughput with increase in threads. This is because with more threads, more elements are added, the value of tail will increase and hence for dequeuing, more elements will have to be checked.
- Initially NLQ is faster than CLQ. This is because enqueue is faster in NLQ than in CLQ (NLQ performs less operations in enqueue functions) ans rndLt is 0.7 hence more enqueues are performed. But as thread and hence number of operations are increased, dequeues also increase making NLQ slower.