Comparing Queue Implementations

Aaryan, CO21BTECH11001

Important Note:

The value of $rndLt$ is taken as 0.75 in all the executions. The reason, which was also discussed with professor sir, is to avoid a halt in NLQ's execution, which can occur if the queue is empty and all the threads have invoked the dequeue method. Note that it also affects the average time taken to complete one operation since the number of enqueues are more than the number of dequeues. However, there will not be a major effect on the average time taken to complete one enqueue or one dequeue operation individually.
Also, the value of λ is taken as 0.7 in all the executions, which is to make sure that the program executes in a reasonable amount of time.

Program Design:

1. Two global arrays named `thrEnqTimes` and `thrDeqTimes` are made.
   $thrEnqTimes[i] = (t[i], num[i])$
   Where, $num[i]$ = Number of enqueue operations done by i^{th} thread and $t[i]$ = time taken by i^{th} thread to execute all these operations.
   Similarly for `thrDeqTimes`.

2. In the implementation of CLQ, a semaphore is used to ensure mutual exclusion.

3. In the implementation of NLQ, the array named `items` is initialized with -1. When `deq()` is called, the element which is not equal to `-1` is set to -1.

4. The function `computeStats` calculates the average enqueue time, average dequeue time and average operation time (in seconds) using the arrays `thrEnqTimes` and `thrDeqTimes.`

5. To ensure that all the threads produce a different sequence of random numbers (the variable `p`), the random number generator is seeded with a value equal to the product of the thread id and the current time.

- **Impact of the number of operations on throughput:**

The data obtained from CLQ implementation:

CLQ							
numOps	Time1	Time2	Time3	Time4	Time5	Average Time	Throughput
10	1.11E-06	1.43E-06	1.41E-06	1.27E-06	1.50E-06	1.34E-06	7.44E+05
20	1.47E-06	1.52E-06	1.74E-06	1.62E-06	1.62E-06	1.59E-06	6.28E+05
30	1.97E-06	1.45E-06	1.66E-06	1.30E-06	1.53E-06	1.58E-06	6.32E+05
40	1.56E-06	1.65E-06	1.60E-06	1.53E-06	1.50E-06	1.57E-06	6.38E+05
50	1.93E-06	1.51E-06	1.56E-06	1.32E-06	1.55E-06	1.57E-06	6.35E+05

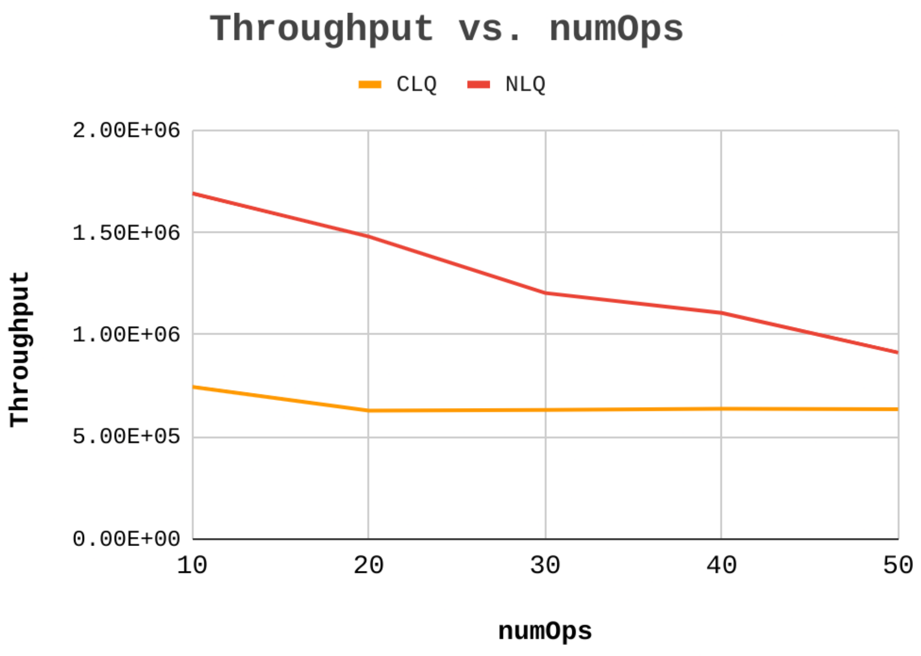
As we can see, throughput doesn't vary significantly with the number of operations. The possible reason is that in this implementation, time taken to execute an operation is primarily dependent on the availability of the lock. When the number of threads are kept constant, it takes almost the same amount of time to acquire the lock and hence complete the method call.

The data obtained from NLQ implementation:

NLQ							
numOps	Time1	Time2	Time3	Time4	Time5	Average Time	Throughput
10	1.15E-06	4.18E-07	3.06E-07	5.87E-07	4.93E-07	5.91E-07	1.69E+06
20	5.71E-07	7.78E-07	5.25E-07	8.81E-07	6.21E-07	6.75E-07	1.48E+06
30	8.83E-07	7.47E-07	1.10E-06	6.91E-07	7.32E-07	8.31E-07	1.20E+06
40	8.45E-07	8.70E-07	9.46E-07	9.71E-07	8.85E-07	9.03E-07	1.11E+06
50	1.18E-06	1.21E-06	9.70E-07	9.01E-07	1.22E-06	1.10E-06	9.12E+05

As we can see, throughput decreases when the number of operations are increased. The reason being that the amount of time taken by a dequeue method call is primarily dependent on the number of elements in the array named `items`. When the number of operations are more, the number of elements in the array increases and therefore it takes more time for a dequeue operation to search for a non-null value and return it.

Here is the graph for the same:



- **Impact of the number of threads on throughput:**

The data obtained from CLQ implementation:

CLQ							
Threads	Time1	Time2	Time3	Time4	Time5	Average Time	Throughput
2	1.45E-06	1.71E-06	1.51E-06	1.51E-06	1.25E-06	1.49E-06	6.73E+05
4	1.40E-06	1.49E-06	1.55E-06	1.71E-06	1.75E-06	1.58E-06	6.33E+05
8	1.51E-06	1.80E-06	1.62E-06	1.55E-06	1.55E-06	1.61E-06	6.23E+05
16	1.43E-06	1.51E-06	1.71E-06	1.85E-06	1.66E-06	1.63E-06	6.13E+05
32	1.82E-06	1.61E-06	1.73E-06	1.48E-06	1.62E-06	1.65E-06	6.05E+05

Note that the throughput decreases slightly when the number of threads are increased. The reason is the dependence of time taken on the availability of the lock. When the number of threads are increased, the waiting time to acquire the lock increases, hence the throughput decreases.

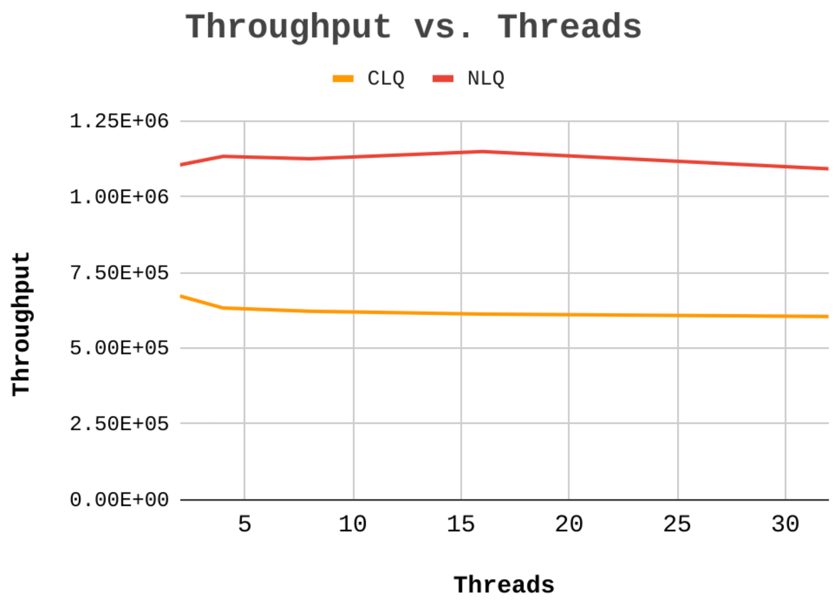
However there is no significant change in the throughput because the total number of operations ($n * numOps$) are very less. Therefore, it brings very little change to be observed.

The data obtained from NLQ implementation:

NLQ							
Threads	Time1	Time2	Time3	Time4	Time5	Average Time	Throughput
2	9.16E-07	9.17E-07	8.33E-07	9.33E-07	9.22E-07	9.04E-07	1.11E+06
4	8.58E-07	8.55E-07	8.91E-07	9.33E-07	8.73E-07	8.82E-07	1.13E+06
8	9.75E-07	8.42E-07	8.87E-07	8.85E-07	8.50E-07	8.88E-07	1.13E+06
16	7.77E-07	8.72E-07	1.01E-06	8.60E-07	8.30E-07	8.70E-07	1.15E+06
32	9.28E-07	8.09E-07	9.02E-07	9.36E-07	1.00E-06	9.15E-07	1.09E+06

Notice that there is no significant change in throughput when the number of threads are increased. The reason is that the change in total number of operations ($n * numOps$) is very less and such a small change doesn't affect the time taken to execute. Since the time taken is primarily dependent on the total number of operations, therefore there is no significant change in throughput.

Here is the graph for the same:

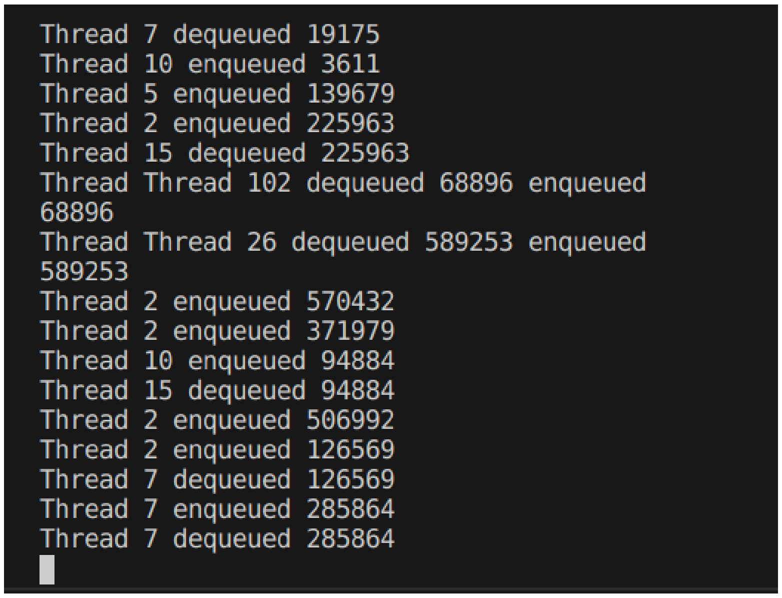


Note: In all of the cases mentioned above, the throughput of NLQ is higher than the throughput of CLQ. The reason for this is that CLQ is implemented using a lock and a large amount of time is consumed in waiting to acquire the lock.
Although NLQ is also not lock free, the use of atomic variables naturally increases the throughput as compared to a lock implementation.

Important note for program execution:

It is less probable, although possible, that the program for NLQ may come to a halt when the queue is empty and every thread has invoked `deq()` method call. Therefore, it is recommended to comment out lines 108 and 123 of the file `NLQ_co21btech11001.cpp` so that you will be able to see if the program has come to a halt. If the program has come to a halt, then you can terminate the program by pressing Ctrl+C.

Here is a scenario where the NLQ program comes to a halt *(Notice that the last method call is a dequeue call)*



```
Thread 7 dequeued 19175
Thread 10 enqueued 3611
Thread 5 enqueued 139679
Thread 2 enqueued 225963
Thread 15 dequeued 225963
Thread Thread 102 dequeued 68896 enqueued
68896
Thread Thread 26 dequeued 589253 enqueued
589253
Thread 2 enqueued 570432
Thread 2 enqueued 371979
Thread 10 enqueued 94884
Thread 15 dequeued 94884
Thread 2 enqueued 506992
Thread 2 enqueued 126569
Thread 7 dequeued 126569
Thread 7 enqueued 285864
Thread 7 dequeued 285864
```

Of course, this is not a problem with the CLQ implementation.