Parallel Computing – Assignment 2

Name: Hrishikesh Deshpande
PUID: 0028994843
Email: hdeshpa@purdue.edu

Question 1.

All tests for Q1 were performed on the machine mc18.cs.purdue.edu that has 32 cores as shown by the command below:

```
mc18 53 $ lscpu
Architecture:        x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              32
```

NOTE about results for Question 1: Each of these throughput nos were obtained by repeating the test for about 50 times with a sleep time of 10 seconds using an automated script and then getting the max. So, the results may not be directly reproducible with 5-10 runs.

Also, we create producer and consumer threads and sleep for a fixed amount of time (10 seconds for this test) and then compute the throughput.

This program implements a producer consumer buffer (with buffer-size=1000). The producer inserts items into the buffer if it is not full and consumer extracts items from the buffer if it's non-empty. The producer produces values p, p+k, p+2k, … and the consumer consumes the values. We have k no of producers and consumers where k = no of threads. We run this program with buffer size = 1000 and for no of producer-consumer threads varying from k = 1,2,4, 8,……,256 and evaluate the producer insertion throughput, consumer extraction throughput and the overall total throughput.

a) Using read-write locks

Implementation: In part a, the buffer used is a single ended buffer meaning all producer will insert items at one end and all consumer extract items from the same end as producers.

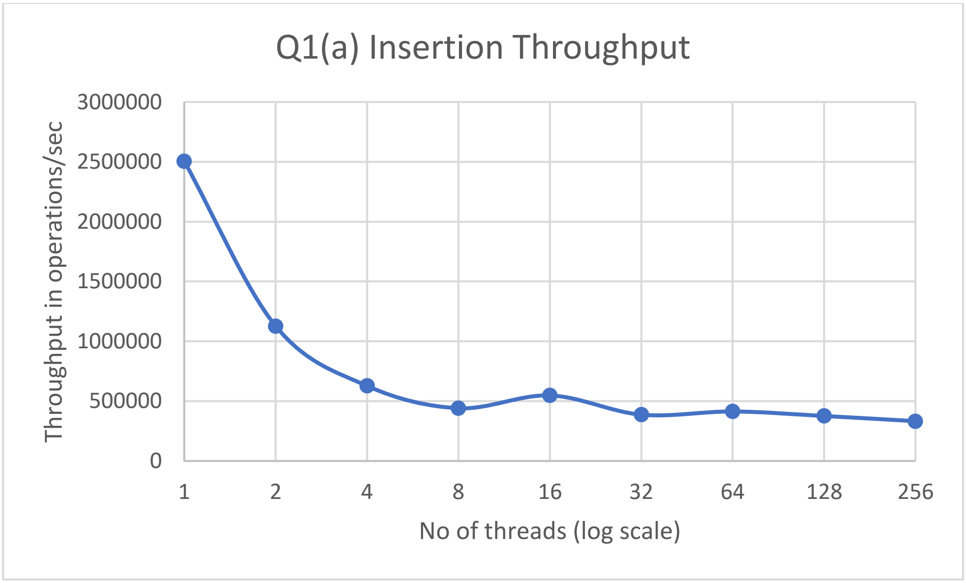
Synchronization: Synchronization between the producer and consumer threads is established using read-write locks.

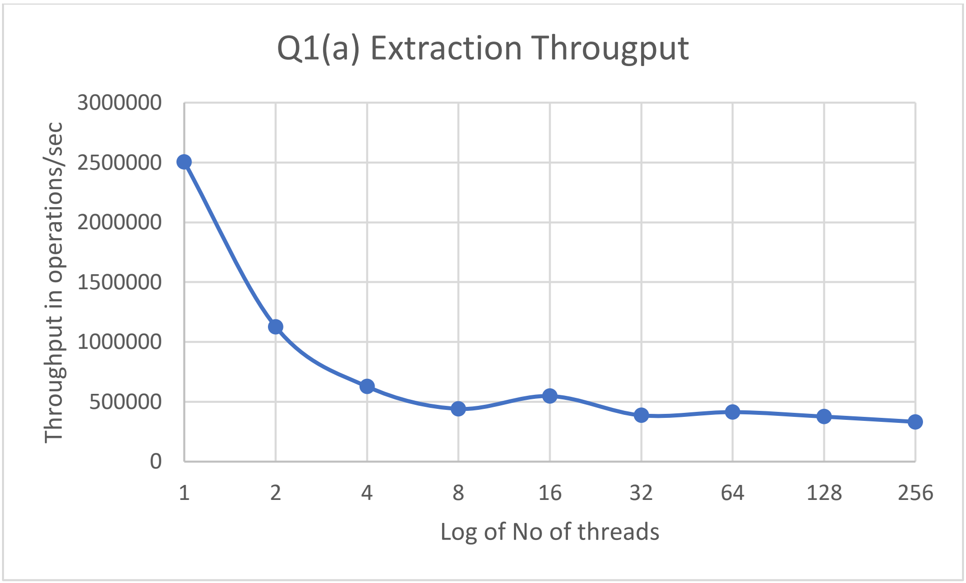
Buffer: The buffer used is a single ended buffer (stack).

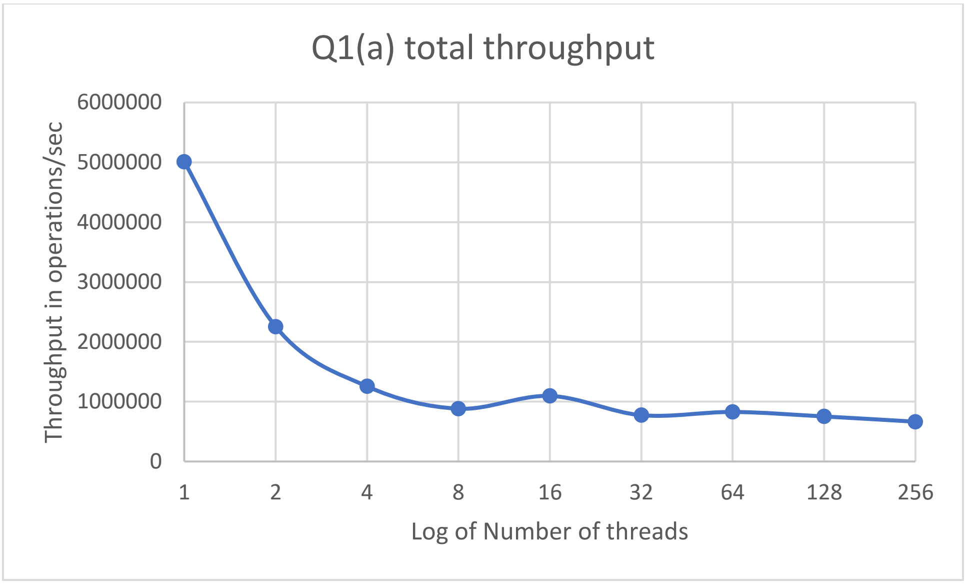
The insertion, extraction and total throughput results for varying no threads k = 1,2,4,…256 are tabulated as below:

Sl. No	No of threads	Insertion throughput (ops/sec)	Extraction throughput (ops/sec)	Total throughput (ops/sec)
1.	1	2505584.960905	2505527.661306	5011112.622212
2.	2	1127647.361763	1127556.062220	2255203.423983
3.	4	627731.743105	627632.643808	1255364.386913
4.	8	441176.555882	441176.055885	882352.611767

5.	16	547802.746260	547794.446321	1095597.192581
6.	32	386913.827307	386816.828102	773730.655409
7.	64	413943.343355	413878.143994	827821.487349
8.	128	376253.138314	376205.439235	752458.577549
9.	256	332581.919612	332547.820502	665129.740114







b) Using condition variables:

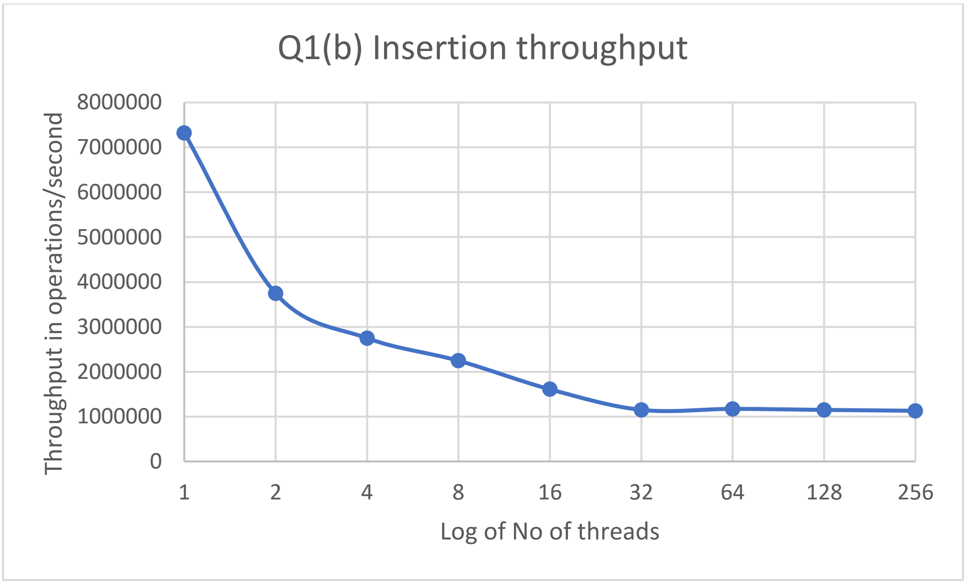
Implementation: Part b of question 1 is basically similar to part a in that we again use a single ended buffer for insertion and extraction of items by producers and consumers respectively.

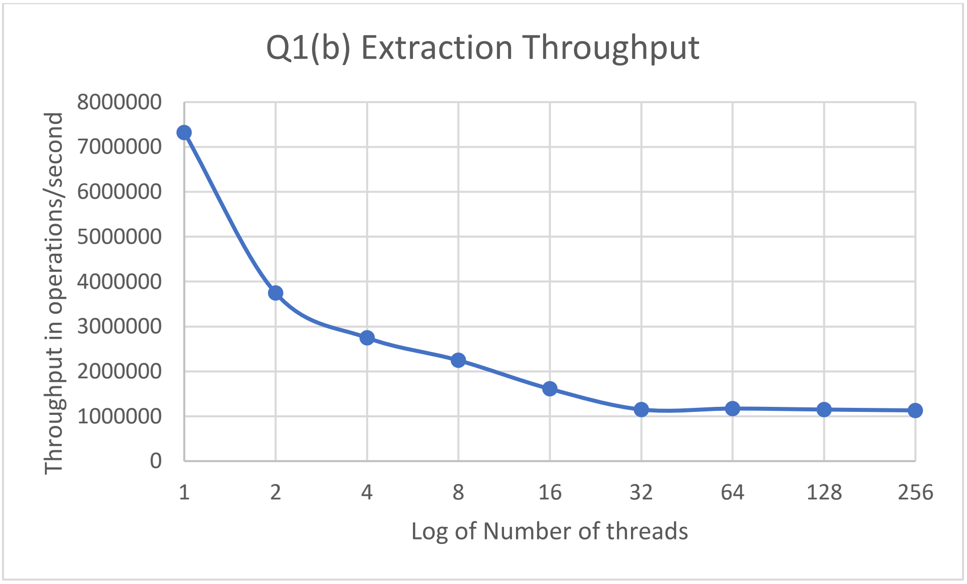
Synchronization: However, synchronization between producers and consumers is achieved through condition variables. So, if a producer finds that the buffer is full, it performs a condition wait. Similarly, a consumer performs a condition wait if the buffer is empty.

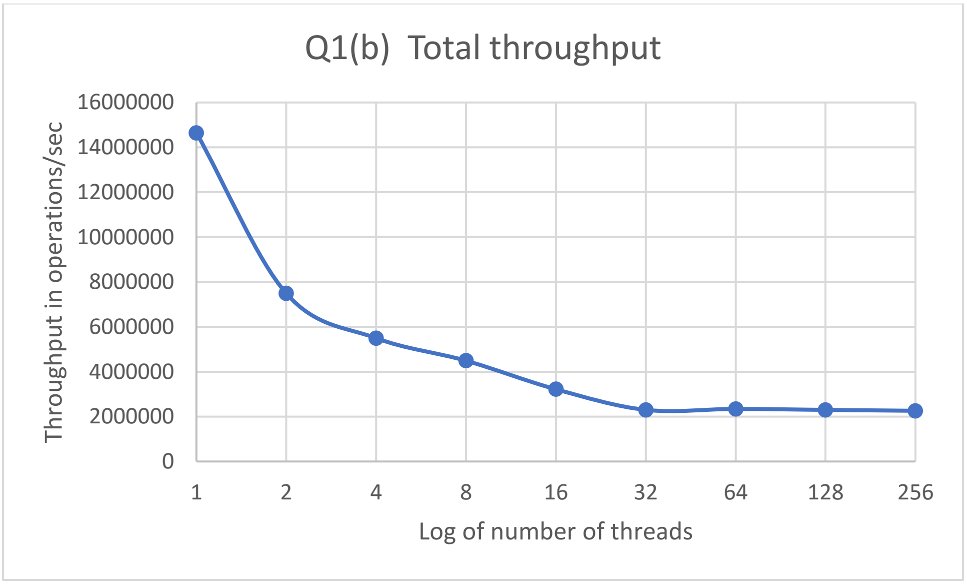
Buffer: The buffer used is a single ended buffer (stack).

The insertion, extraction and total throughput results for varying no threads k = 1,2,4,…256 are tabulated as below:

Sl. No	No of threads	Insertion throughput (ops/sec)	Extraction throughput (ops/sec)	Total throughput (ops/sec)
1.	1	7324932.722951	7324844.623541	14649777.346492
2.	2	3744781.212053	3744716.412513	7489497.624567
3.	4	2748483.034253	2748383.034863	5496866.069117
4.	8	2245648.983938	2245565.684588	4491214.668526
5.	16	1609413.107519	1609352.607985	3218765.715504
6.	32	1149954.680399	1149901.980863	2299856.661261
7.	64	1171798.307140	1171772.707437	2343571.014576
8.	128	1149511.028410	1149492.928722	2299003.957132
9.	256	1130121.645518	1130115.945635	2260237.591153


Q1(b) Insertion throughput
Throughput in operations/second
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
1
2
4
8
16
32
64
128
256
Log of No of threads


Q1(b) Extraction Throughput
Throughput in operations/second
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
1
2
4
8
16
32
64
128
256
Log of Number of threads



c) Circular Buffer with condition variables

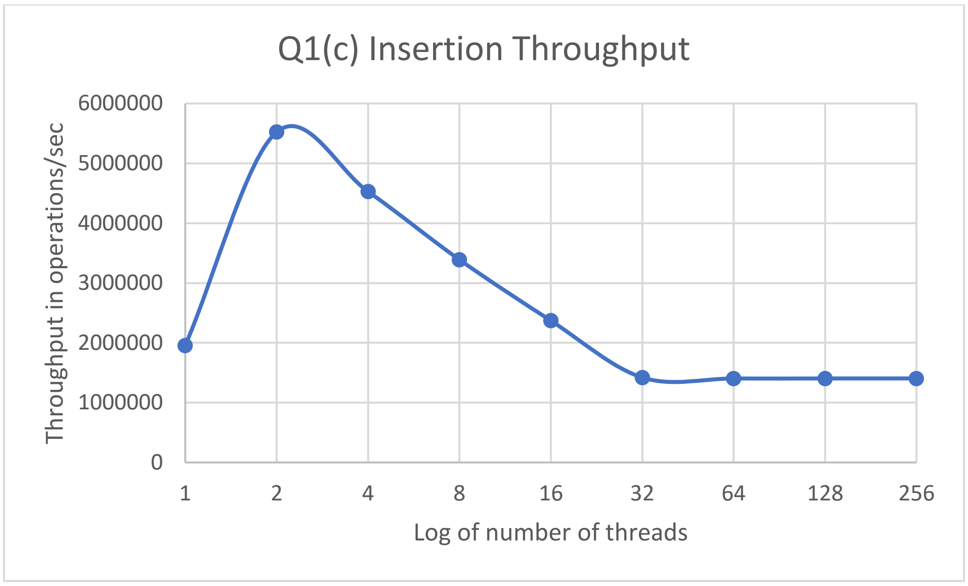
Implementation: In part 3 of question 1, we use a circular buffer as a shared resource between producers and consumers. This means that producers insert items at one end and consumer extract items from the other end.

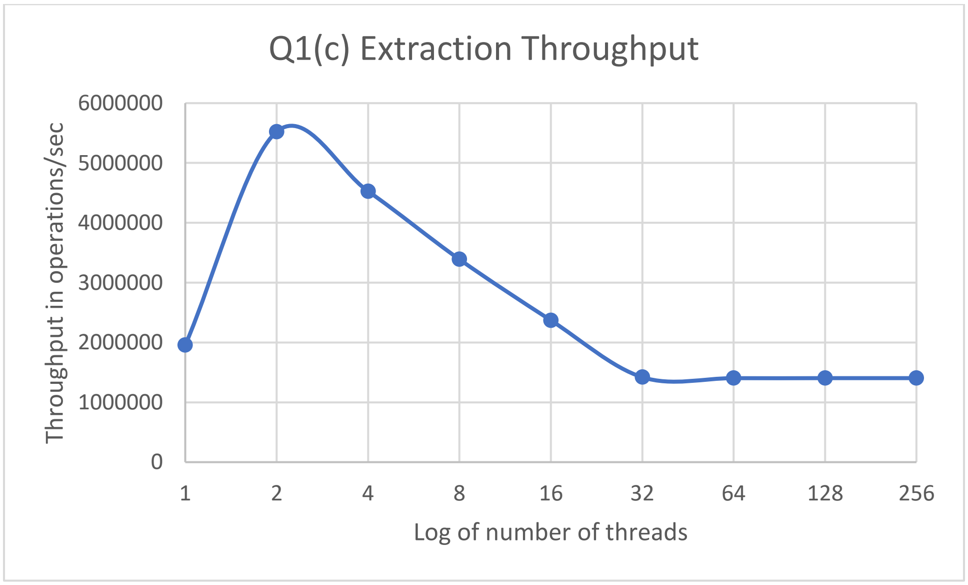
Locking mechanism: The synchronization between producers and consumers is achieved using condition variables. We use 2 condition variables one for producer and another for consumer which are controlled using a mutex variable.

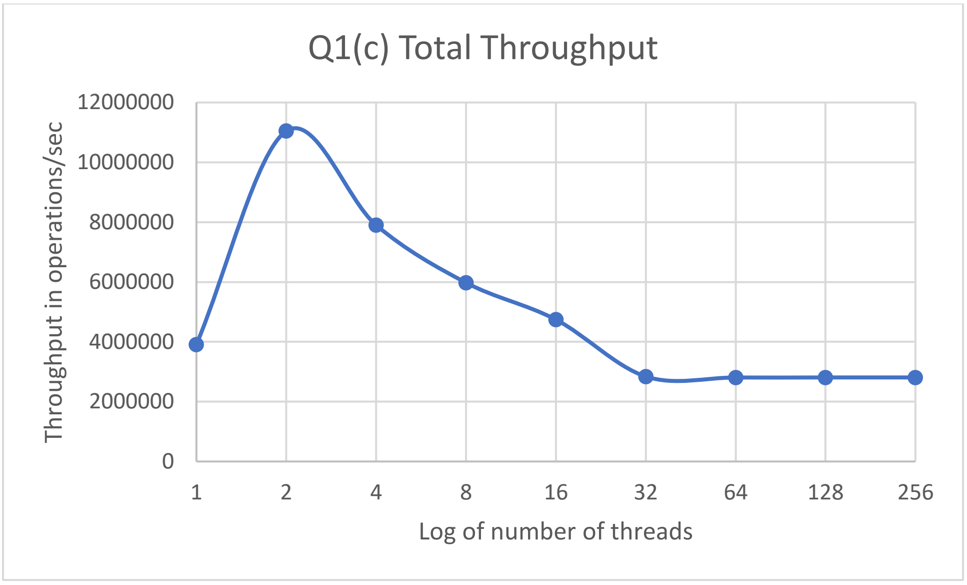
Buffer: We use a circular queue to implement Part c.

The insertion, extraction and total throughput results for varying no threads k = 1,2,4,...256 are tabulated as below:

Sl. No	No of threads	Insertion throughput (ops/sec)	Extraction throughput (ops/sec)	Total throughput (ops/sec)
1.	1	1954267.101837	1954250.001950	3908517.103787
2.	2	5522305.060407	5522269.260711	11044574.321118
3.	4	4527930.979070	4527924.379134	7899783.711752
4.	8	3389231.780762	3389161.081292	5976007.375146
5.	16	2370954.669539	2370950.269601	4741905.039139
6.	32	1420431.910415	1420431.310420	2840863.220835
7.	64	1403856.022982	1403770.623802	2807626.646784
8.	128	1404322.443808	1404322.443808	2808644.887616
9.	256	1404322.443808	1404322.443808	2808644.887616


Q1(c) Insertion Throughput
Throughput in operations/sec
6000000
5000000
4000000
3000000
2000000
1000000
0
1
2
4
8
16
32
64
128
256
Log of number of threads


Q1(c) Extraction Throughput
Throughput in operations/sec
6000000
5000000
4000000
3000000
2000000
1000000
0
1
2
4
8
16
32
64
128
256
Log of number of threads



Comparison of performance for Parts a and b:

a) Producer-consumer implementation using a single ended buffer and read-write locks gives us the least performance. This is due to the fact that both producer and consumer will have to lock a common buffer end using a write lock. This results in long wait periods.

b) In part b, we use two condition variables one for producer and another consumer. When the buffer is full, producers sleep and when it's empty, consumers sleep. This ensures that the other party gets a shot with the performing an operation while the other party sleeps. This explains a drastic increase in throughput in part b as compared to part a.

As

Based on tabulated data, performance of part b (condition variables) is almost 3 times that of using read locks.

Representative examples:

Threads = 4: Part a total tput= 1255364.386913, Part b total tput = 5496866.069117

Threads = 256: Part a total tput= 665129.740114 Part b total tput = 2260237.591153

Comparison of performance of parts b and c:

From the data tabled, we can easily guess that throughput with a double ended queue of part c gives us a significant throughput jump in comparison with the single ended buffer of part b. Producers compete among each other for insertion at tail. Consumers compete among each other for extraction from head. But producers and consumers do not compete among each other at the same end. This explains the significant increase in performance in part c as compared to part b.

The performance speedup of part c w.r.t part b is approx by a factor of 1.5 when the no of threads is <= 16.

i.e. performance speed with circular buffer = 1.5* performance of normal buffer

For no of threads >= 32 and upto 256, the speedup is approximately 1.2.

Reason: For all parts a,b and c as we increase the no of threads, there will be increased contention among producer and consumer threads resulting in a slowdown.

Representative examples:

No of threads = 4, Total tput part b = 4491214.668526 Total tput Part c = 7899783.711752

No of threads = 256, Total tput part b = 2260237.591153, Total tput part c = 2808644.887616

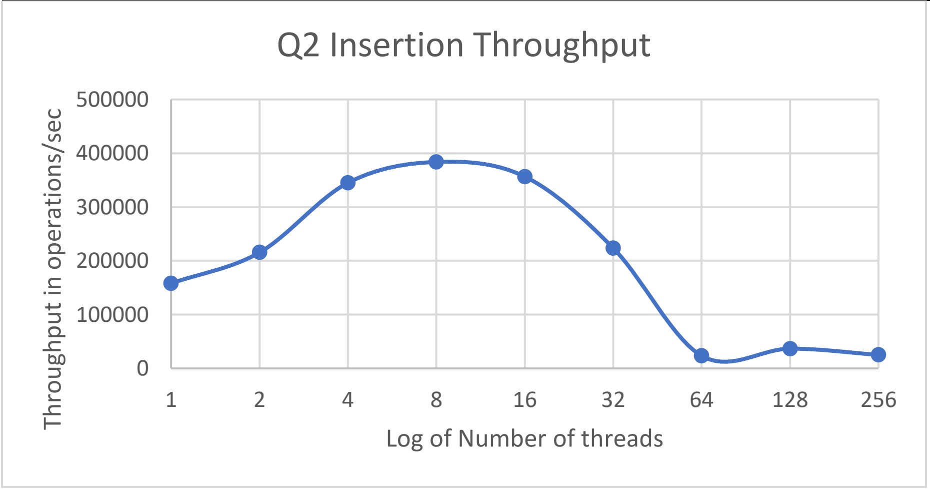
Question 2:

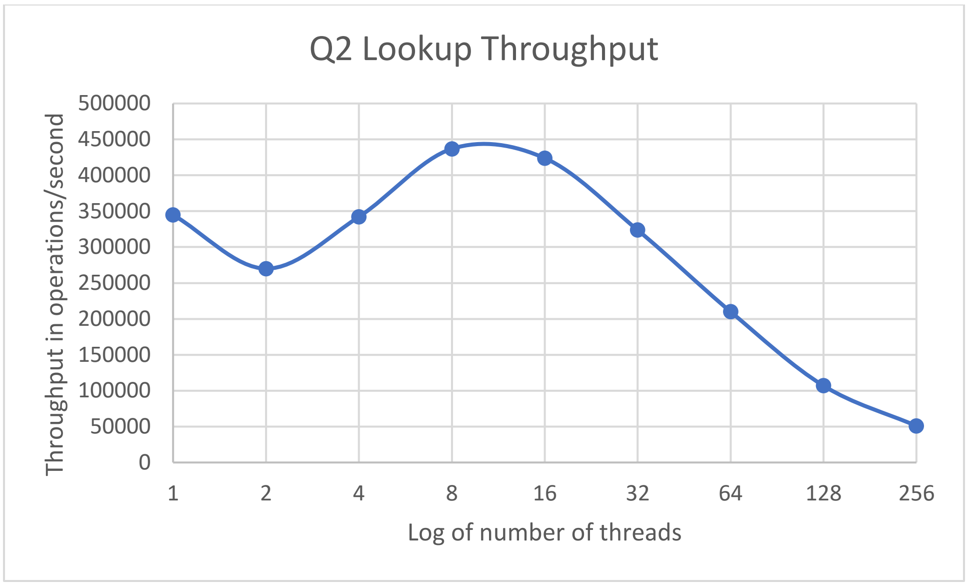
For question 2, we use the machine mc17.cs.purdue.edu which has 24 cores on the machine as shown by the following command:

```
mc17 51 $ lscpu
Architecture:        x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):             24
```

NOTE about results: Each of these throughput no's were obtained by repeating the test for about 200 times using an automated script and then getting the max. So the results may not be directly reproducible with 5-10 runs.

Sl. No	No of threads	Insertion throughput (ops/sec)	Lookup Throughput (ops/sec)
1.	1	158024.691358	344827.586207
2.	2	215488.215488	270042.194093
3.	4	345013.477089	342245.989305
4.	8	383808.095952	436860.068259
5.	16	356545.961003	423841.059603
6.	32	223776.223776	323640.960809
7.	64	23561.895996	210008.203445
8.	128	36755.204594	106889.352818
9.	256	25149.818253	50935.137286





Observations:

1. We insert only 256 values ranging from 1-256.
2. In case of insertions, as we increase the number of threads from 2-16 the throughput increases. Since this is a 24-core machine, beyond 16 threads we see a dip in performance. As such we are only inserting 256 values. So, increasing the no of threads beyond 32 and up to 256, means that each thread has to insert only a few values.
3. With 256 threads, each thread is to insert only 1 value. With a high no of threads in comparison to the no of values to be inserted, we will observe an increased degradation in performance due to increased contention between threads for read/write locks.
4. A similar result can be observed with lookups. However, lookups do not require any write locks. So, with only read locks, we can obviously find lookups working much faster than insertions.
5. Even in case of lookups, with an increase in no of threads there's an increased overhead of acquiring read locks (although there's no contention among read locks since multiple reads can proceed in parallel). As the no of threads increases from 32 and beyond, performance degrades since each thread must lookup a very small no of values.