Assignment 8

Inderpreet Singh Chera - 160101035

Producer Consumer Problem:

Multiple producers and multiple consumers share a single bounded queue. A producer adds an integer (that is, element) at the end of the queue. A consumer removes the number (say x) from the front of the queue and sleeps for x seconds. A producer cannot add an item if the queue is full. Similarly, a consumer can consume an item only when the queue has at least one item.
Bounded lock free queue was used for the same purpose.

System Settings

All tests were done on **Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz** processor. This computer is **dual core** where each core has **2 threads**. Also during each experiment it was insured that **no other applications** were running in the background, so that we don't have any biased readings.

Implementations

Bounded Lock Free Queue

As given in Micheal and Scotts paper "**Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms**", lock free queue was used in our algorithm. Pseudo code of the algorithm as given in paper is:

```
structure pointer_t        {ptr: pointer to node_t, count: unsigned integer}
structure node_t           {value: data type, next: pointer_t}
structure queue_t          {Head: pointer_t, Tail: pointer_t}

initialize(Q: pointer to queue_t)
        node = new_node()                    # Allocate a free node
        node->next.ptr = NULL                # Make it the only node in the linked list
        Q->Head = Q->Tail = node             # Both Head and Tail point to it
```

```
enqueue(Q: pointer to queue_t, value: data type)
E1:     node = new_node()                                   # Allocate a new node from the free list
E2:     node->value = value                                 # Copy enqueued value into node
E3:     node->next.ptr = NULL                               # Set next pointer of node to NULL
E4:     loop                                                # Keep trying until Enqueue is done
E5:         tail = Q->Tail                                  # Read Tail.ptr and Tail.count together
E6:         next = tail.ptr->next                           # Read next ptr and count fields together
E7:         if tail == Q->Tail                              # Are tail and next consistent?
E8:             if next.ptr == NULL                         # Was Tail pointing to the last node?
E9:                 if CAS(&tail.ptr->next, next, <node, next.count+1>)  # Try to link node at the end of the linked list
E10:                    break                               # Enqueue is done. Exit loop
E11:                endif
E12:            else                                        # Tail was not pointing to the last node
E13:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)  # Try to swing Tail to the next node
E14:            endif
E15:        endif
E16:    endloop
E17:    CAS(&Q->Tail, tail, <node, tail.count+1>)           # Enqueue is done. Try to swing Tail to the inserted node
```

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:     loop                                                # Keep trying until Dequeue is done
D2:         head = Q->Head                                  # Read Head
D3:         tail = Q->Tail                                  # Read Tail
D4:         next = head->next                               # Read Head.ptr->next
D5:         if head == Q->Head                              # Are head, tail, and next consistent?
D6:             if head.ptr == tail.ptr                     # Is queue empty or Tail falling behind?
D7:                 if next.ptr == NULL                     # Is queue empty?
D8:                     return FALSE                        # Queue is empty, couldn't dequeue
D9:                 endif
D10:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)  # Tail is falling behind. Try to advance it
D11:            else                                        # No need to deal with Tail
                    # Read value before CAS, otherwise another dequeue might free the next node
D12:                *pvalue = next.ptr->value
D13:                if CAS(&Q->Head, head, <next.ptr, head.count+1>)  # Try to swing Head to the next node
D14:                    break                               # Dequeue is done. Exit loop
D15:                endif
D16:            endif
D17:        endif
D18:    endloop
D19:    free(head.ptr)                                      # It is safe now to free the old dummy node
D20:    return TRUE                                         # Queue was not empty, dequeue succeeded
```

After going through the algorithm a bug was found in this algorithm due to free use in this algorithm. Essentially, the problem is that **Tail.ptr** can't be dereferenced without first loading **Tail**, and **dequeue** frees **Head.ptr**. If the queue is empty at line E5, then **tail == Head** since when the queue is empty **Tail == Head**. If, on other threads, a single **enqueue** followed by a single **dequeue** occurs before line E6, then **tail.ptr** will be **freed** because **dequeue frees head.ptr**, **head == Head**, and **tail == Head**. Then line E6 will proceed to dereference it, which is a NULL value. After a little research, a more recent paper on **Hazard pointers** was able to tackle this problem. But, as this problem occurs when one of the processors is very slow and the other is very fast which was not in our case as all CPU cores are equally efficient, I went with Micheal and Scott's lock free queue.

Now, to make it a bounded queue, another variable **nodeNum** was added to the pointer struct. nodeNum variable is always incremented on addition of queue, and hence size of the queue can be found out by difference of tail and head pointers (Note that +1 is not required for the size of the queue, as head pointer always points to dummy node). Size check was added between E6 and E7 in above pseudo code.

Now, for the implementation of producer consumer problem, inputs as number of producers, number of consumers, max size of queue, and total number of enqueue operations are taken from the user. Then this total number of enqueue operations are divided equally among the producers. On the other hand, consumers go on consuming until they see that queue is empty. Now, if the consumers start their job before even producers can produce something, they will see the queue is empty and hence will exit. To overcome this, another bool variable **hasProducerStartedProducing** is added and when the producer starts producing this variable is changed to true. Consumers wait till this variable is made true. Also, if the queue gets full, producers keep trying to add into the queue in a **random backoff** (of max 2 seconds) manner to avoid contentions.

Also, it being the simulation of producer consumer problem, each second is considered as milliseconds in the code, and also, each producer generates work of ((it's id)%2+1).

Throughput Calculations: Throughput is calculated as:

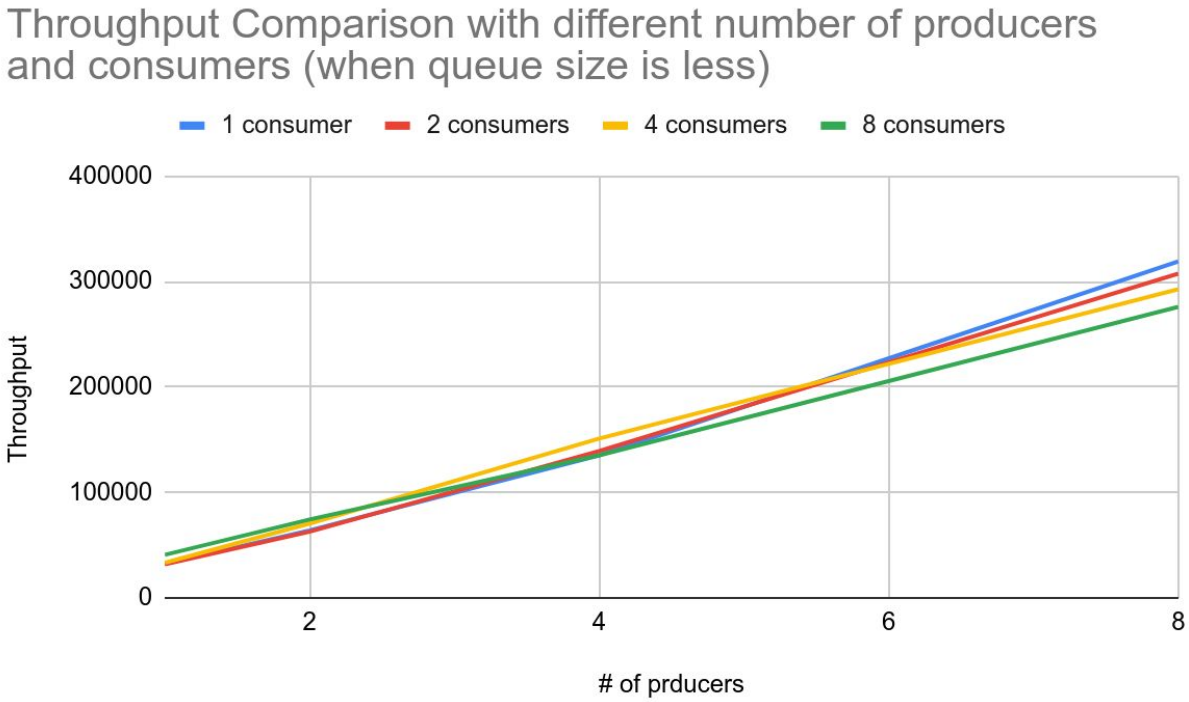
Throughput = 2*(Total Number of Enqueue Operations)/(Time taken by program).

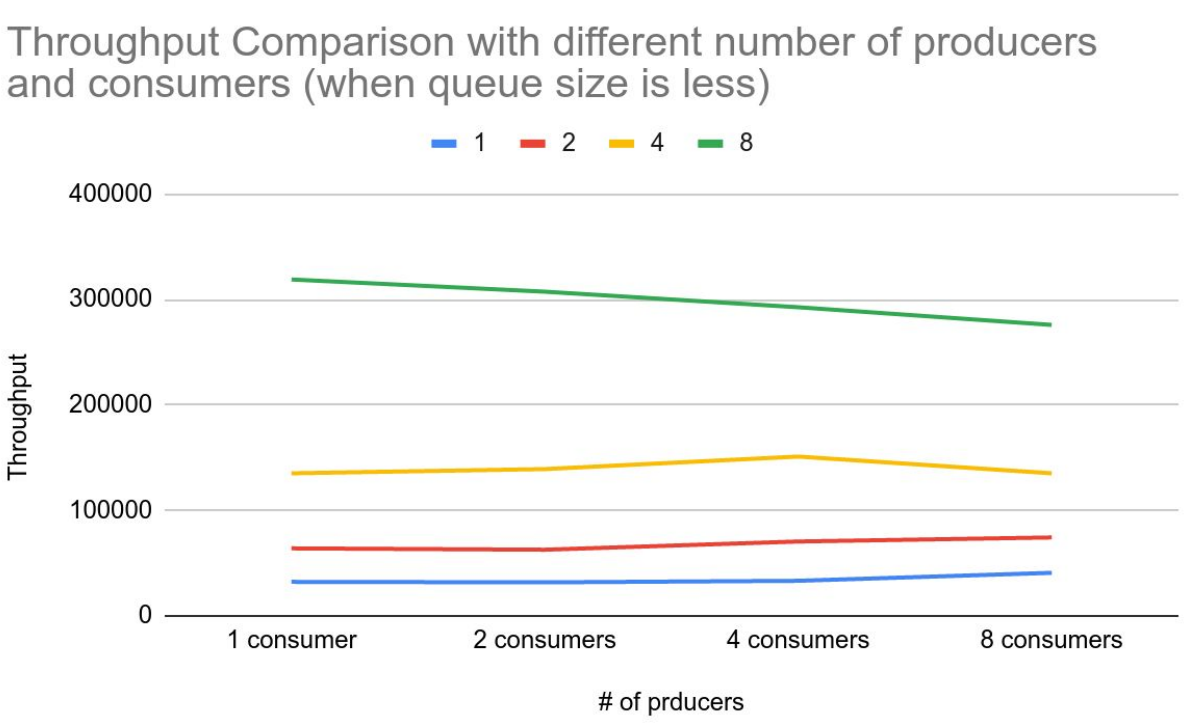
Total number of enqueue operations is given as input to the program .

Experiments

Throughput by changing number of producers and consumers with queue size = 10^5 and total number of operations = 2^{20} (around 10^6) [Here queue size is less than total number of enqueue operations, hence producers must went through random backoff]

# of producers	1 consumer	2 consumers	4 consumers	8 consumers
1	32272.8	31786	33273	40831.1
2	64032.4	62761.4	70553.3	74396.7
4	135259	139252	151131	135250
8	319104	307533	292815	276030


Throughput Comparison with different number of producers and consumers (when queue size is less)
1 consumer
2 consumers
4 consumers
8 consumers
400000
300000
200000
100000
0
Throughput
2
4
6
8
# of prducers

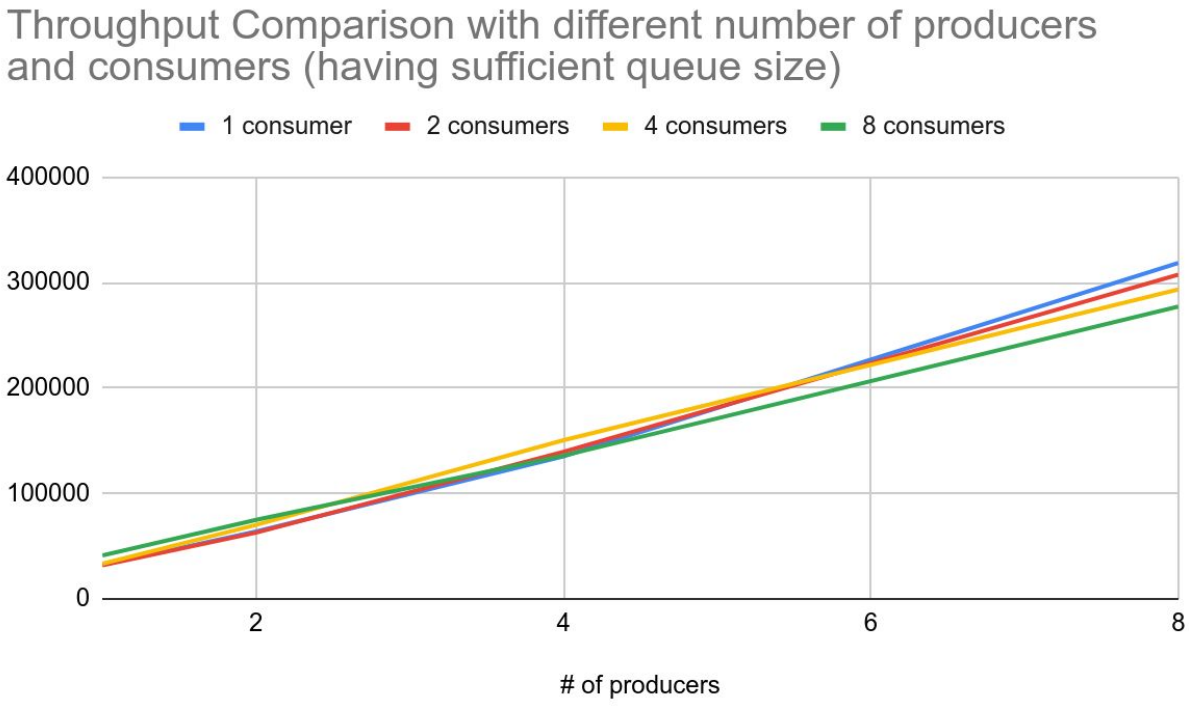

Throughput Comparison with different number of producers and consumers (when queue size is less)
1
2
4
8
400000
300000
200000
100000
0
Throughput
1 consumer
2 consumers
4 consumers
8 consumers
# of prducers

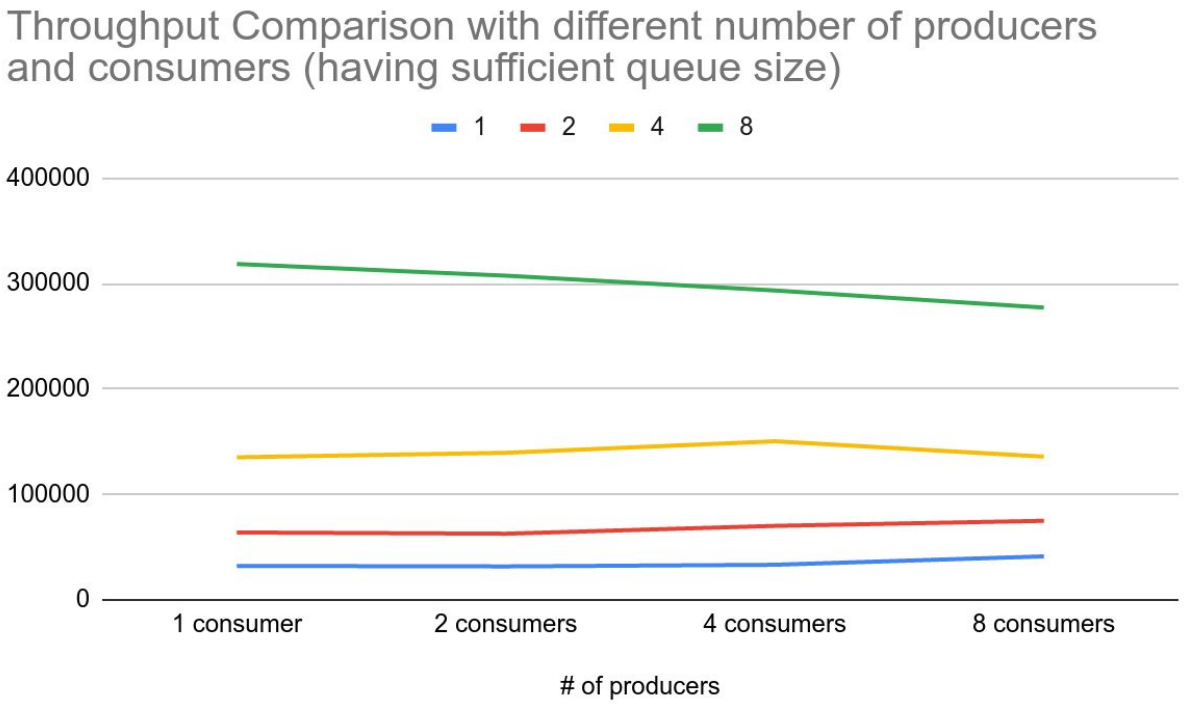
Observations:

- From the first graph, on fixing the number of producers there is no significant change in throughput among the consumers.
- From the second graph, we observe that on fixing the number of consumers, throughput increases when the number of producers are increased.

Throughput by changing number of producers and consumers with queue size = 2^{21} and total number of operations = 2^{20} (around 10^6) [Here queue size is more than total number of enqueue operations, hence no problem for producers]

# of producers	1 consumer	2 consumers	4 consumers	8 consumers
1	32240.8	31745.4	33292	41277.8
2	63945.5	62758.4	70181.3	74984.2
4	135248	139525	150469	135885
8	318646	307547	293489	277287



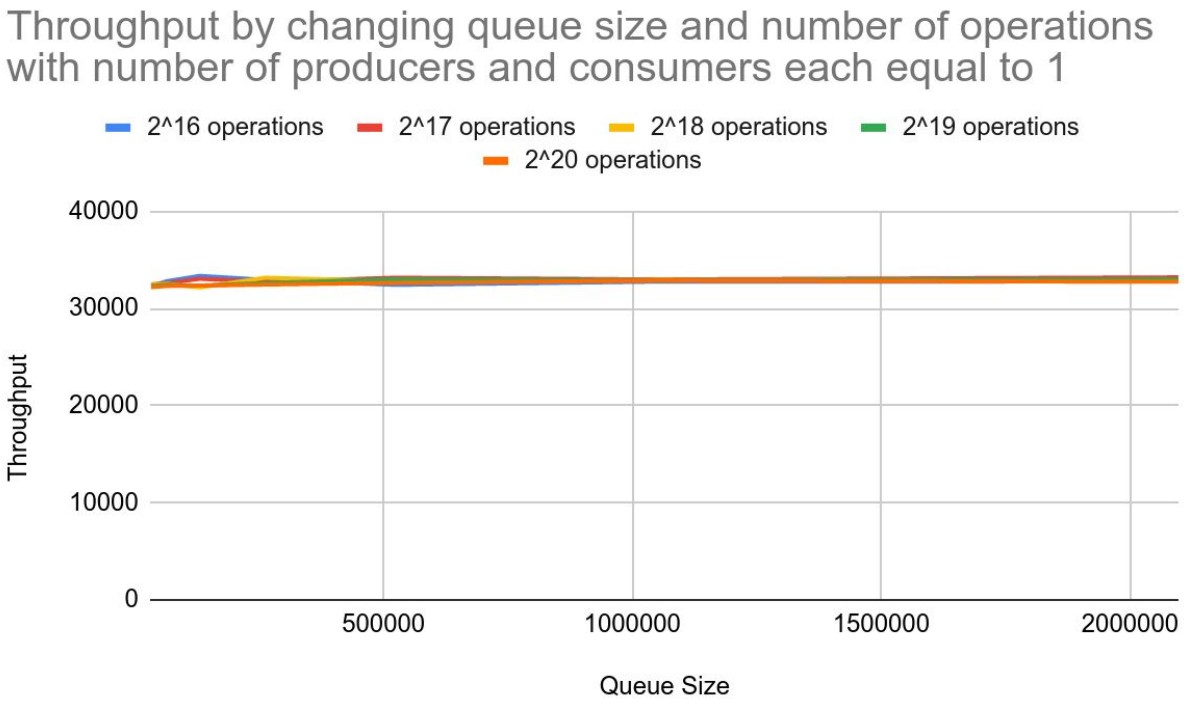


Observations:

The results and throughput are similar as when the queue size was small then total number of operations. So, for now we may assume that queue size doesn't effect throughput.

Throughput by changing queue size and number of operations with number of producers and consumers each equal to 1.

Queue Size	2^16 operations	2^17 operations	2^18 operations	2^19 operations	2^20 operations
32768	32306.3	32475.4	32433	32304.7	32207.2
65536	32822.4	32621.9	32544.2	32448	32323.9
131072	33363.1	33061.2	32169.5	32359.2	32361
262144	32949.6	32752.4	33166.5	32624.4	32466.9
524288	32439.5	33172.4	32825.3	33008	32646.5
1048576	32769.2	32996.1	32943.1	32881.2	32870.5
2097152	32786.6	33212.9	32890.7	33004	32755.7

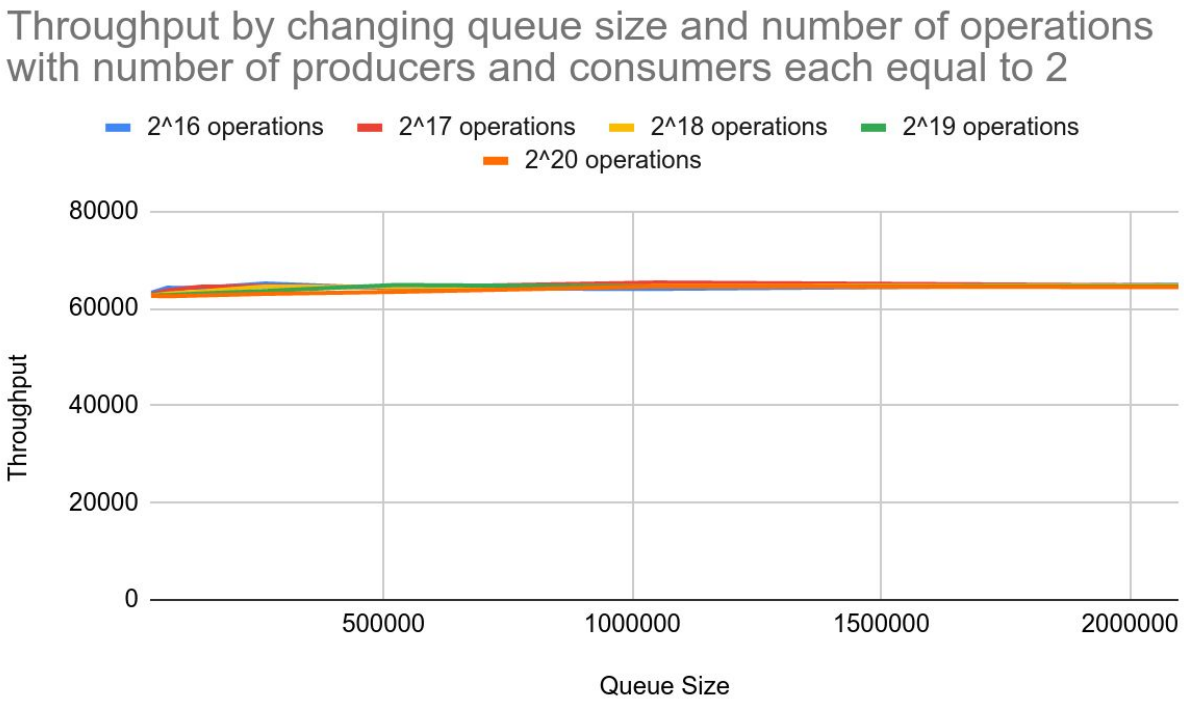


Observations:

As observed in the earlier graph, throughput is almost the same on increasing queue size or number of operations. Let's see if increasing number of producers and consumers affect the graph.

Throughput by changing queue size and number of operations with number of producers and consumers each equal to 2.

Queue Size	2^16 operations	2^17 operations	2^18 operations	2^19 operations	2^20 operations
32768	63225.4	62914.9	62699.6	62707.4	62604.3
65536	64286.5	63711	63074.7	62751.2	62521.7
131072	64227.2	64491.7	63502	63081.4	62689.7
262144	65156.2	64628.9	64546.3	63614	63019.3
524288	64138.4	64405.8	64428.8	64852.5	63430
1048576	64037.7	65360.8	64604.1	64500.8	64542.4
2097152	64907.2	64739.3	64768	64579.4	64409.8



Observations:

Now, we can safely assume that the number of operations and queue size doesn't matter for current implementation of code, probably because of random backoff used for the producer that reduces the contention at the producer side. Also, throughput has increased from the previous case.

Conclusion:

- With An increasing number of producers, throughput increases.
- There is no effect on throughput on increasing the number of consumers. (Although it should have mattered theoretically.)
- There is no effect on increasing the number of operations or queue size on the throughput.