MA513

Parallel Computing

Assignment-8 Report

Paranjay Bagga
160101050

Problem Statement :-

Write a **Producer-Consumer** program using Pthreads, where a single finite sized queue is shared among them, while maximizing the achieved throughput by varying the no. of Producers, Consumers, max Queue Size & total no. of Operations involved.

System Specifications :-

All the experiments have been performed on **Dell Inspiron 5559** laptop with **Intel i5-6200U dual core processor** and **8 GB RAM**. Each core has **2 threads.** Also, it was made insured that no other applications were running in the background during each experiment which could have incurred biased readings.

Implementation Details :-

1. The input to the program are - no. of producer threads, no. of consumer threads, FullQueue size and the total no. of operations to be done by all the producers, as well as consumers, as a whole. It must be noted that each producer's or consumer's pthread is running independently in the system.
2. In the program, it is considered that the total operations are equally divided among all the producers (and the consumers as well). Also, it is guaranteed

that the required no. operations (of enqueue/dequeue) per thread are definitely performed by each individual thread, even if it involves waiting upto a time until the required condition is reached. Thus in every possible case, the total no. of operations performed, by producers (and consumers too) as a whole, are the same as given in the input.

3. Each producer thread performs an enqueue operation (of an integer having its value as either *1* or *2*) until the queue size is full, after which it *sleeps* for a predefined back-off time of *2 microseconds* before re-attempting the enqueue operation. This process is repeated until the queue is not full or it has completed the no. of operations assigned to it. Also, after all the producer threads complete their required operations a global boolean variable ***prod_ended*** is set true.
4. Similarly, each Consumer thread performs a dequeue operation until the queue is not empty and the global boolean variable ***prod_ended*** is not set to true. In case either of the above conditions fail, the respective consumer thread repetitively keeps on checking for the non-emptiness of the queue, until it succeeds. The above procedure is repeated until the thread has completed the no. of operations assigned to it. Also, each time a successful dequeue operation is performed, the corresponding consumer thread sleeps for that much integral value of microseconds that is dequeued by it, before it could proceed to the next operation.
5. **Parallel Queue :** The parallel non-blocking lockless queue used in the program is referred from the paper "**Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms - Michael, Scott - 1992**". The queue is modified to have a finite length by introducing a new variable ***queueSize*** which stores the index of each node in the queue. At any instant, the length of the queue is given by *tail.queueSize - head.queueSize*. Whenever a thread tries to enqueue, the current size of the queue is

checked and if it is less than *FULLSIZE*, the thread proceeds further else enqueue operation is cancelled.

Correctness: **If a thread passes the check: *tail.queueSize - head.queueSize < FULLSIZE* and if the enqueue operation happens, then that operation is always correct and does not produce any inconsistency.**
Let's suppose that a particular thread A passes the test and after which, it goes into sleep. Now, let's assume, meanwhile, other threads do some dequeue and enqueue operations.

a. If a dequeue operation happens, then the size of the queue further decreases and hence if the thread A successfully enqueues after it, the enqueue operation is valid and correct.
b. Similarly, if some enqueue operation happens in between and the queue becomes full, in that case, the *tail* gets modified and the enqueue operation by the thread A gets cancelled, as per the algorithm mentioned in the Michael Scott paper, and as a result, thread A loops back to get the updated *tail* value. After that, it again checks the queue size until the *tail* actually points to the last node and only then enqueues its held value. Hence, the queue remains consistent.
c. In case nothing happens in between, the enqueue successfully proceeds and hence is again a valid operation.

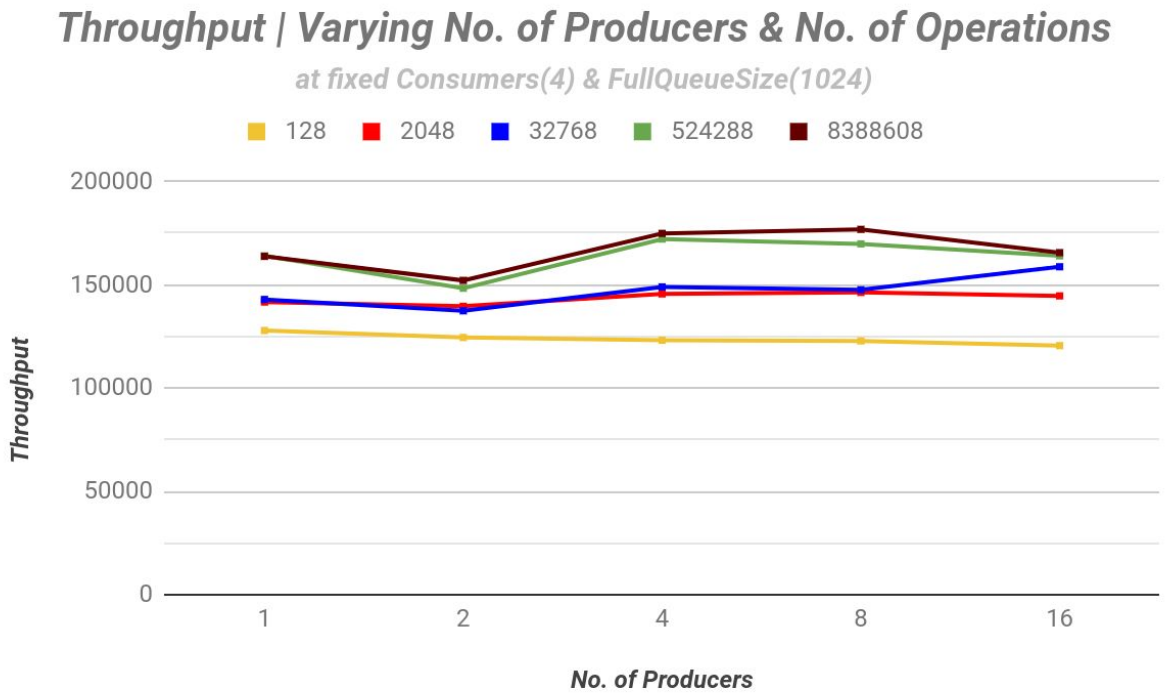
6. **Throughput Calculation :** The throughput of the system is defined by :-
Total number of operations (enqueues+dequeues) per second
i.e. (2**Total_Ops* / *time_taken* by whole process)
Similarly, Throughput per Thread is defined by dividing by the throughput value by the no. of parallel threads used.

Experiments *:-*

The value of the no. of producers and consumers is taken as **1**,**2**,**4**,**8** & **16.**
The size of the queue is taken as **64**,**256**,**1024**,**4096** & **16384**.
The total no. of operations are taken as **128**,**2048**,**32768**,**524288** & **8388608**.
The experiment results are shown in the tables and graphs below.
The values in the table are average values of 8 (or 1 for larger N) executions to avoid any unexpected behaviour introduced as a result of thread switching.

1. **Throughput** | **Varying No. of Producers & No. of Operations** | **at fixed Consumers(4) & FullQueueSize(1024) :-**

No. of Producers	No. of Operations				
	128	**2048**	**32768**	**524288**	**8388608**
1	127888.0979	141666.3784	142925.9184	164044.5944	163783.1461
2	124498.4802	139617.1266	137428.1913	148358.6791	152132.3104
4	123143.5271	145612.2576	148984.2243	172167.1889	174904.2590
8	122848.0595	146330.1359	147577.3455	169819.9715	176858.8642
16	120555.6864	144648.0913	158746.3745	164026.6059	165589.6078

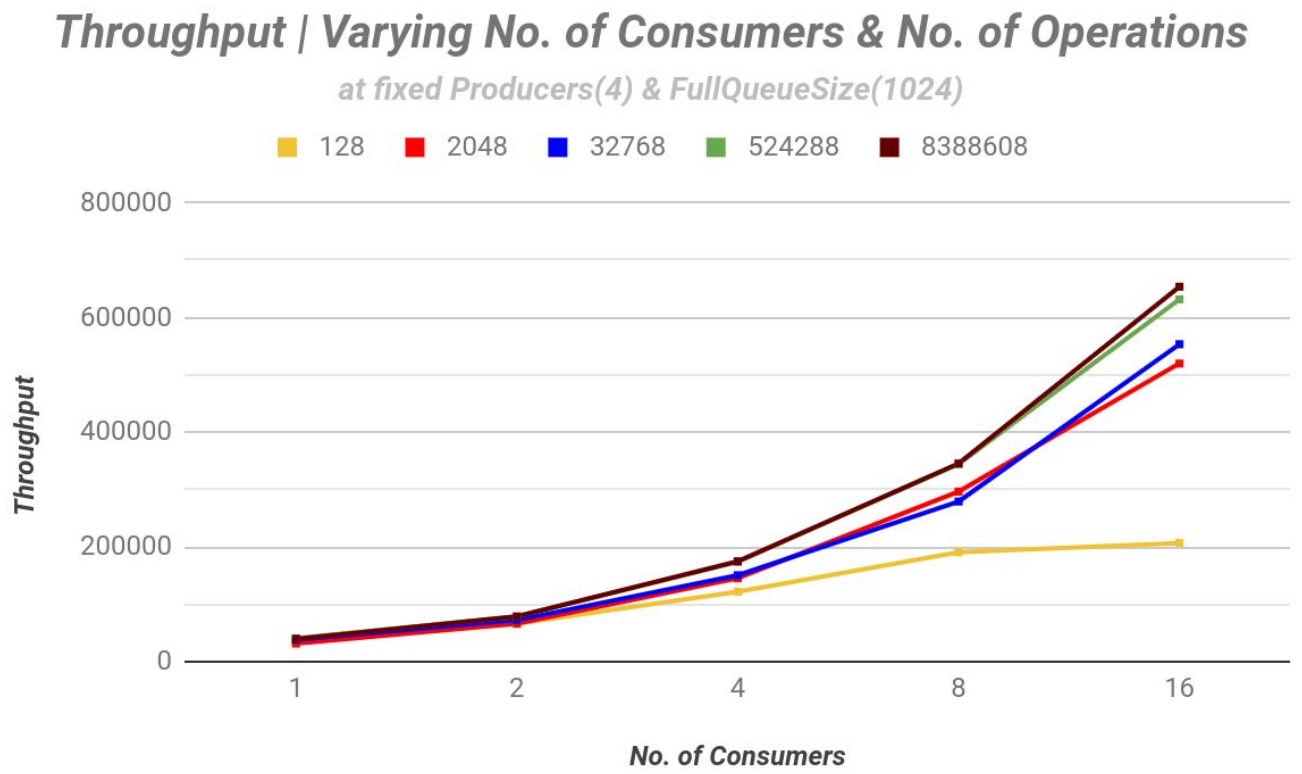


Observations :-

1. From the above graph, when the no. of producers are increased along with only the no. of operations, while keeping other parameters constant, it is observed that the *Throughput* remains nearly constant even upon increasing producer threads. This also signifies that the *Throughput per thread* decreases with increase in no. of producers.
2. As a result, this points that the queue remains full frequently and hence the producers have to wait more frequently to carry on successful enqueue operations, thus causing similar throughput values even upon increase in producers when no. of operations remain the same.
3. On the other hand, as expected, when the no. of operations increase, then the throughput values too, irrespective of how many the no. of producers are.
4. Also, it can be seen that maximum throughput is reached when the no. of producers are **4**, the reason behind which may be attributed to the fact that the system on which the experiment is performed has a total of 4 threads!

2. Throughput | Varying No. of Consumers & No. of Operations | at fixed Producers(4) & FullQueueSize(1024) :-

No. of Consumers	No. of Operations				
	128	2048	32768	524288	8388608
1	33912.3379	31476.4580	38137.3171	40476.5284	39698.5030
2	66377.1310	65973.6693	72433.4379	78715.2914	78538.9901
4	121970.1030	145462.9395	150576.9472	174080.0986	174876.3053
8	190564.8088	296502.7372	279166.0077	344655.2399	345316.0316
16	206806.0184	519772.2190	553428.2089	631633.4779	653697.2597

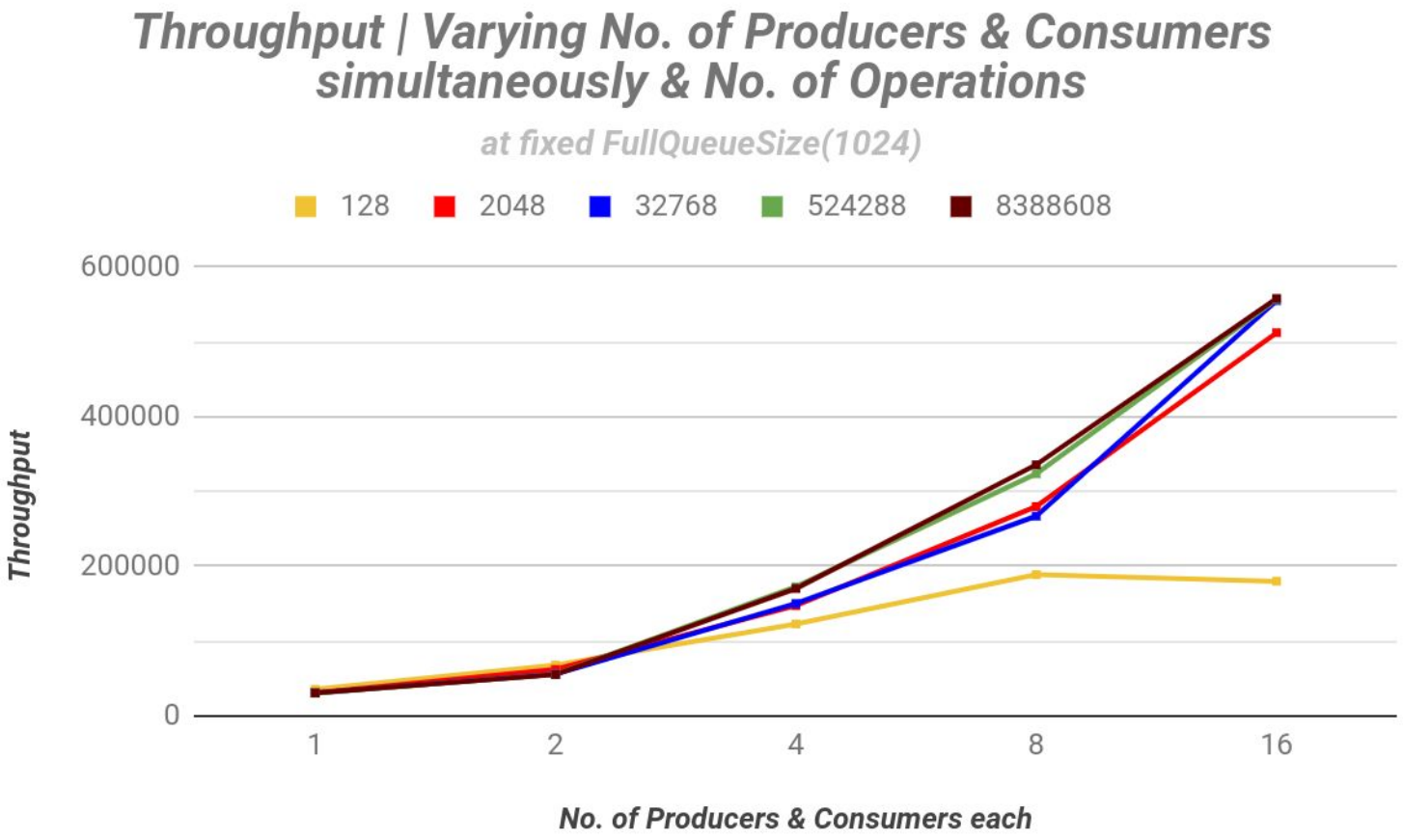


Observations *:-*

1. From the above graph, when the no. of consumers are increased along with only the no. of operations, while keeping other parameters constant, it is observed that the *Throughput* increases manifold (nearly by twice every time no. of consumers are doubled) upon increasing consumer threads for any value of no. of operations. This also signifies that the *Throughput per thread* too increases with increase in no. of consumers, as was opposed in the previous case of producers.
2. As a result, this points that more consumers try their best in not making the queue being full frequently and hence, avoiding the producers to wait more often for carrying successful enqueue operations, thus causing larger throughput values upon increase in consumers when no. of operations remain the same. Thus, it can be seen that maximum throughput is reached when the no. of consumers are **16**.
3. Again, as expected, when the no. of operations increase, then the throughput values too.

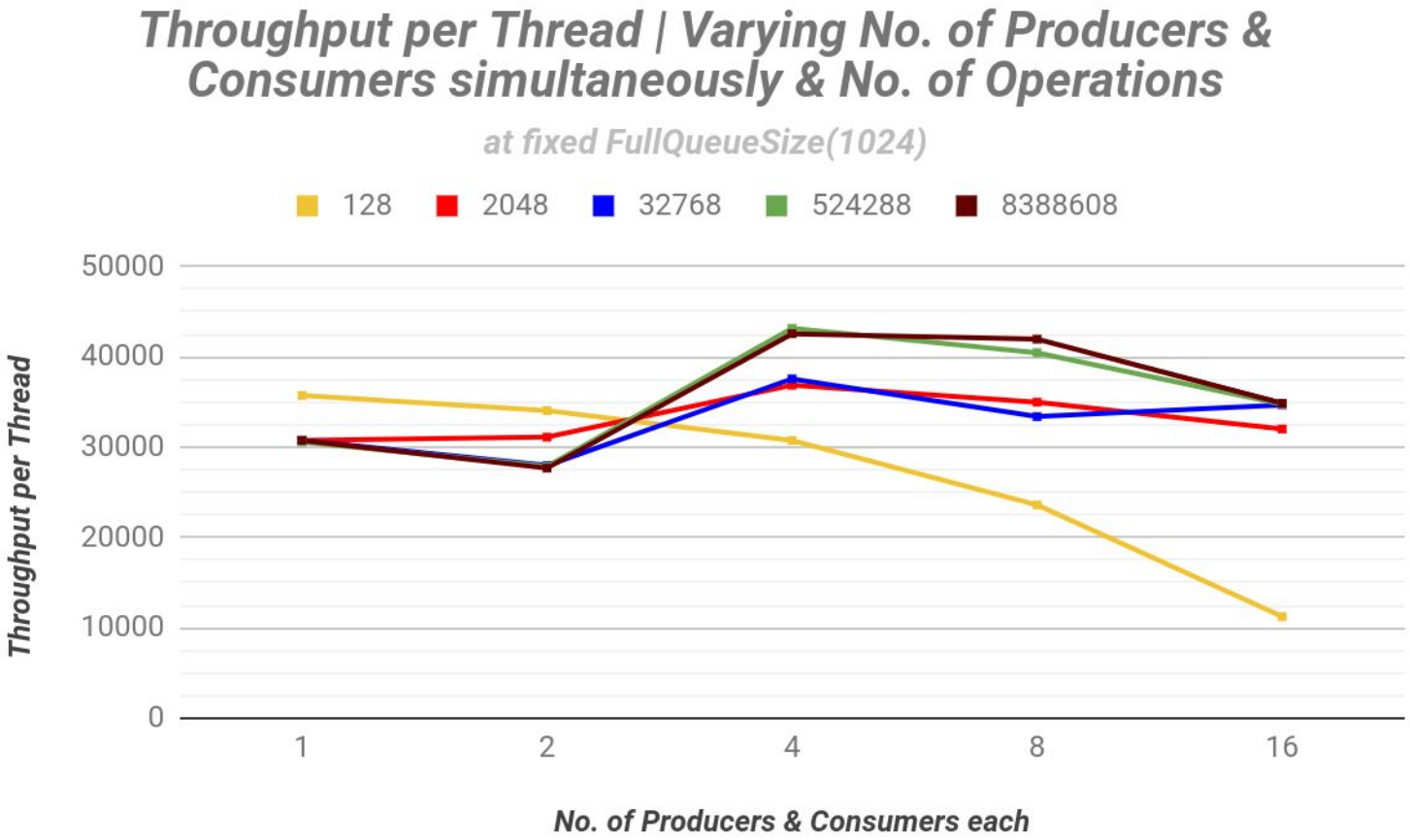
3. **Throughput** | Varying No. of Producers & Consumers simultaneously as well as No. of Operations | at fixed FullQueueSize(1024) :-

Equal No. of Producers & Consumers	No. of Operations				
	128	**2048**	**32768**	**524288**	**8388608**
1	35723.0072	30733.3018	30761.0293	30557.8473	30751.7328
2	68073.7909	62210.3586	55951.6024	55763.3804	55350.0762
4	122980.8443	147426.2936	150242.7925	172632.1273	170233.9990
8	188877.6169	279793.3655	267077.3187	323617.4881	335634.3103
16	180012.3055	512128.0320	554910.1624	556430.7805	557822.1589



4. <u>Throughput per Thread</u> | Varying No. of Producers & Consumers simultaneously as well as No. of Operations | at fixed FullQueueSize(1024) :-

Equal No. of Producers & Consumers	No. of Operations				
	128	2048	32768	524288	8388608
1	35723.0072	30733.3018	30761.0293	30557.8473	30751.7328
2	34036.8955	31105.1793	27975.8012	27881.6902	27675.0381
4	30745.2111	36856.5734	37560.6981	43158.0318	42558.4997
8	23609.7021	34974.1707	33384.6648	40452.1860	41954.2888
16	11250.7691	32008.0020	34681.8852	34776.9238	34863.8849

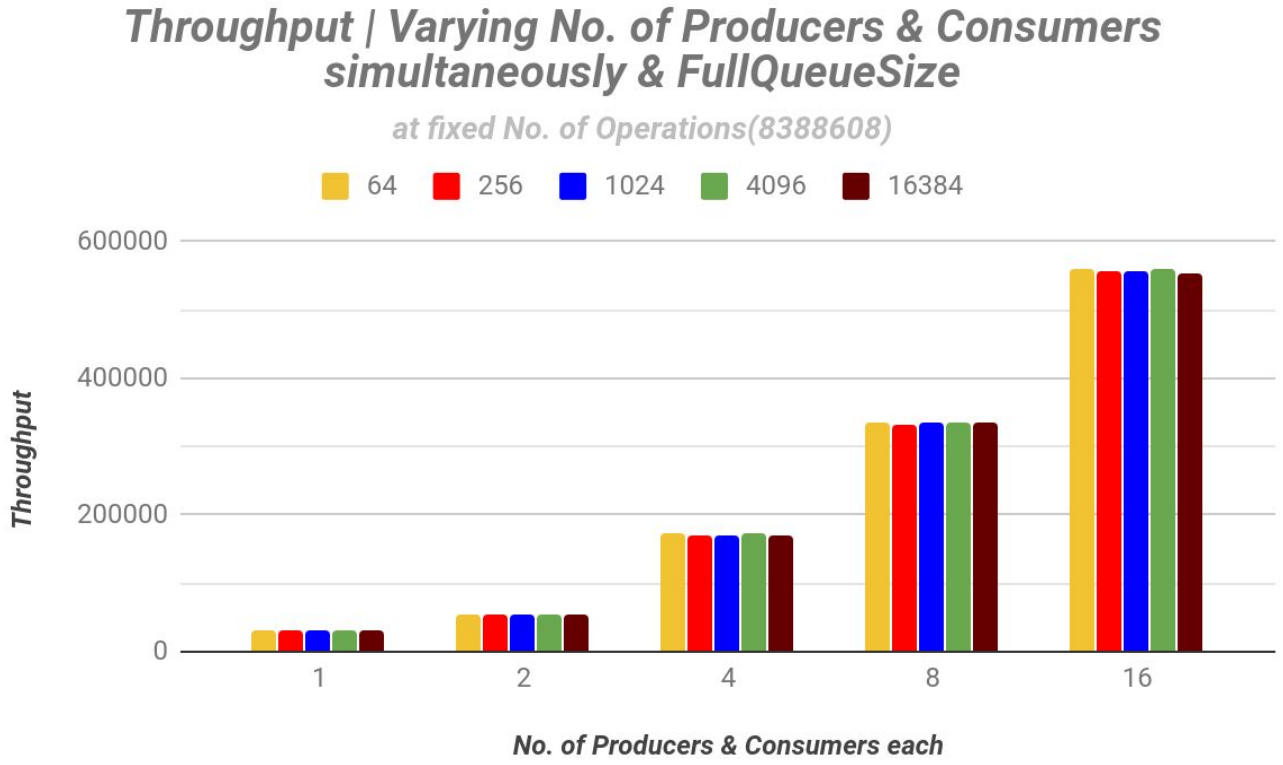


Observations *:-*

1. From the above two graphs (3&4), when the no. of producers and consumers are simultaneously increased along with the no. of operations, while keeping the fullQueueSize as constant, it is observed that the *Throughput* increases manifold upon increasing no. of threads. Also it can be seen that the *Throughput per thread* increases the most while using **4** threads and reaches the minimum value in case of **2** threads, with the exception when no. of operations were *128*, in which case the throughput value keeps on decreasing with increase in threads. This exception can be attributed to the fact that fullQueueSize used is *1024* which is only greater than the no. of operations value when they are *128*, and is less for all others.
2. On the other hand, as expected, when the no. of operations increase, then the throughput values too. Also, the *Throughput per thread* values too increase with the no. of operations but only when the no. of threads are **>=4**.
3. Also, it can be seen that maximum *Throughput* is reached when the no. of threads are **16**(since it increases with inc. in threads), while the maximum *Throughput per thread* is reached when the thread count is **4**. The reason behind this may be attributed to the fact that the system on which the experiment is performed has a total of 4 threads, and hence could result in a better parallelism.

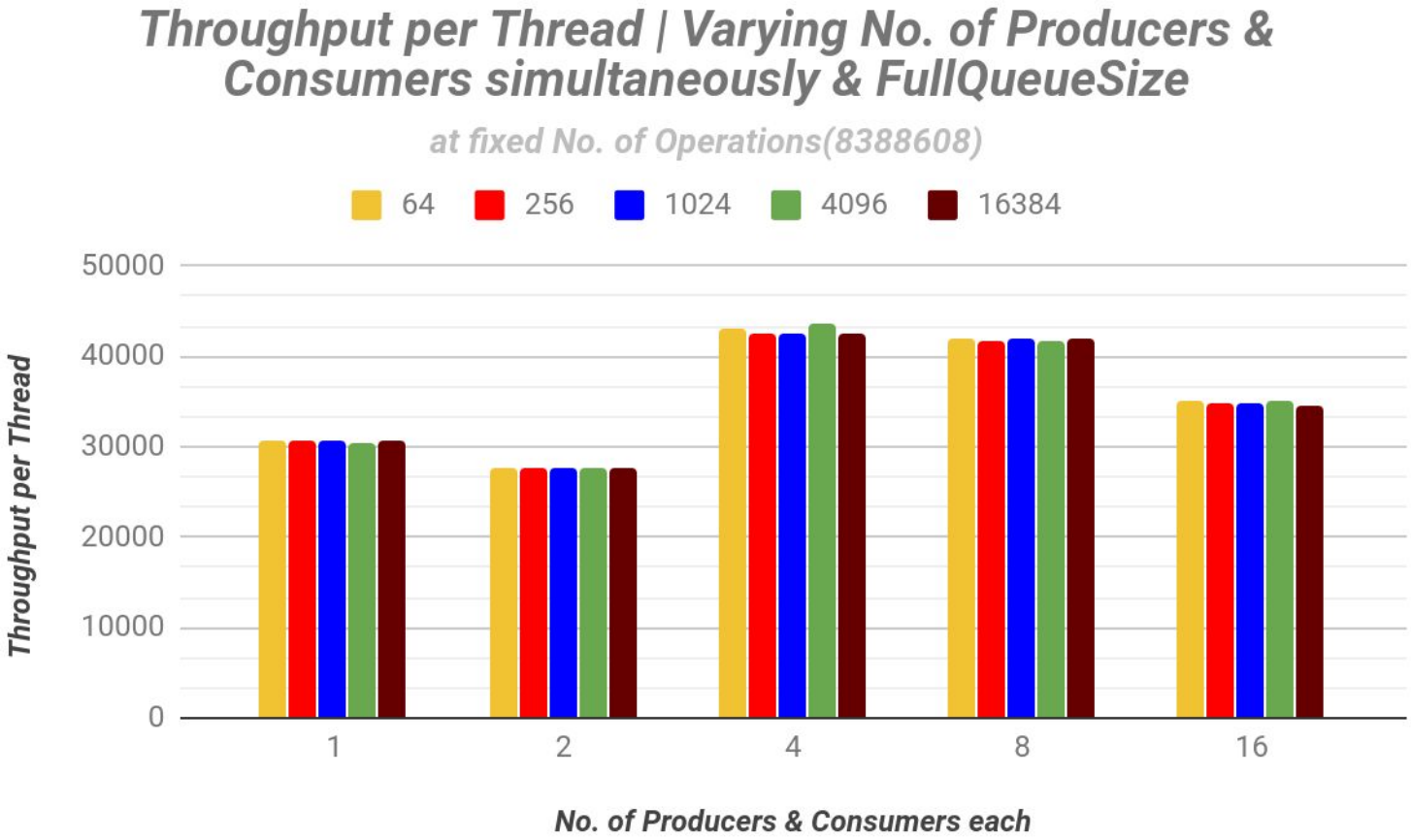
5. **Throughput** | Varying No. of Producers & Consumers simultaneously as well as FullQueueSize | at fixed No. of Operations(8388608) :-

Equal No. of Producers & Consumers	Queue Size				
	64	256	1024	4096	16384
1	30589.4577	30594.9341	30751.7328	30469.6644	30562.2243
2	55171.9468	55194.9928	55350.0762	55229.7940	55203.5197
4	172589.0407	170067.7601	170233.9990	174841.1940	169595.8325
8	334615.2207	333320.8235	335634.3103	333882.5199	334697.1346
16	559275.5067	556570.2946	557822.1589	560834.7247	552270.8385



6. <u>Throughput per Thread</u> | Varying No. of Producers & Consumers simultaneously as well as FullQueueSize | at fixed No. of Operations(8388608) :-

Equal No. of Producers & Consumers	Queue Size				
	64	256	1024	4096	16384
1	30589.4577	30594.9341	30751.7328	30469.6644	30562.2243
2	27585.9734	27597.4964	27675.0381	27614.8970	27601.7598
4	43147.2602	42516.9400	42558.4997	43710.2985	42398.9581
8	41826.9026	41665.1029	41954.2888	41735.3150	41837.1418
16	34954.7192	34785.6434	34863.8849	35052.1703	34516.9274

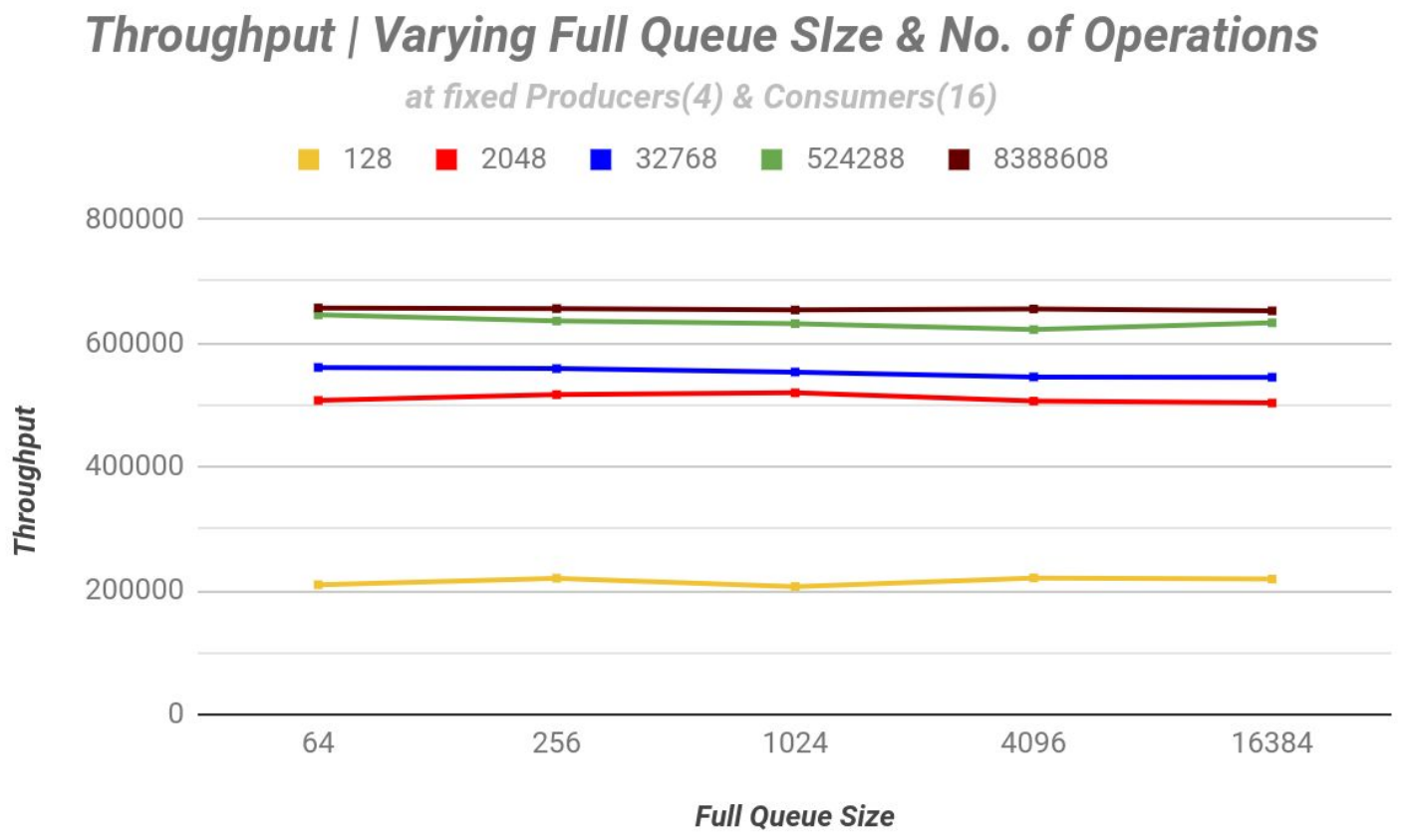


Observations *:-*

1. From the above two graphs (5&6), when the no. of producers and consumers are simultaneously increased along with the fullQueueSize, while keeping the no. of operations as constant, it is observed that the *Throughput* increases manifold(max by thrice factor every time no. of threads are doubled) upon increasing no. of threads. Also, it can be seen that the *Throughput per thread* increases the most while using **4** threads and reaches the minimum value in case of **2** threads.
2. On the other hand, when the fullQueueSize increases, then the throughput values remain nearly the same, irrespective of the no. of threads used.
3. Also, it can be seen that maximum *Throughput* is reached when the no. of threads are **16**(since it increases with inc. in threads), while the maximum *Throughput per thread* is reached when the thread count is **4**. The reason behind this may be again attributed to the fact that the system on which the experiment is performed has a total of 4 threads, and hence could result in a better parallelism.

7. **Throughput** | **Varying FullQueueSize & No. of Operations | at fixed Producers(4) & Consumers(16) :-**

Full Queue Size	**No. of Operations**				
	128	**2048**	**32768**	**524288**	**8388608**
64	209771.5866	507511.6935	560809.6038	645836.7337	656970.9389
256	220167.7059	516829.1221	558940.6430	635867.4823	655889.5027
1024	206806.0184	519772.2190	553428.2089	631633.4779	653697.2597
4096	220665.8765	506421.4512	545485.0017	622020.4798	655236.6056
16384	218850.1817	503356.4264	544983.1917	632964.1461	652243.5244



Observations :-

1. From the above graph, when the no. of operations and fullQueueSize are increased, while keeping the no. of producers and consumers as constant(taking them as their inferred individual best values), it is observed that the *Throughput* increases upon increasing no. of operations, irrespective of the fullQueueSize used.
2. On the other hand, when the fullQueueSize increases, then the throughput values remain nearly the same, irrespective of the no. of operations performed.
3. Also, it can be seen that maximum *Throughput* is reached when the no. of operations are **8388608**(since it increases with inc. in operations).