Choosing a High performance Queue

Roshan Naik
Hortonworks
Jan 19th 2017

STORM-2306

1 Storm queuing requirements

Since queues are the de-facto building blocks for inter-thread message passing, we need to carefully select a high-speed queue that can satisfy Storm's needs well. But first, let us define Storm's requirements for inter-thread messaging queues.

- **Multi Producer:** A bolt executor can receive data from one or more upstream executors. Also it is worth noting that generally the number of producers is a fixed and relatively small number that is known in advance.
- **Single Consumer:** Don't need multi consumer support. Concurrent queues that don't have to support multi-consumer can operate much faster.
- **Bounded capacity:** Queue size must be bounded. It should not grow arbitrarily and consume all available memory.
- **Array based:** queues are preferable to link list based ones. This reduces amount dynamic allocation and garbage collection.

For state-of-the-art performance, lock based queues are essentially not worthy of consideration as locks are too expensive. *Lock-free*, or preferably *wait-free* queues are desirable. Fortunately there are some battle-tested options available that can satisfy Storm's needs. There are two that I was able to test drive. LMAX Disruptor, which is already being used in Storm, is one popular option. The other is MpscArrayQueue included in JCTools library, built by the performance wizards behind the Zing JVM.

In the next few sections we shall provide some published or self-measured numbers of various queuing options. Some of these measurements have also been mentioned in the document attached to STORM-2284

2 Java - ArrayBlockingQueue:

Java provides this queue out of the box so it was worth benchmarking it for reference. It is a lock-based queue.

Throughput: 4 million/sec.

As we shall see, this is clearly suboptimal.

3 Java - ConcurrentLinkedQueue

Java also provides this *wait-free* queue out of the box. Unfortunately it is not a bounded queue, which makes it unsuitable for Storm. Secondly it is linked list based.

4 JCTools - MPSCArrayQ:

This queue is designed specifically for "multi producer single consumer" use. In this benchmark we have one or more producers threads continuously inserting a Long value into the queue and a Consumer thread is draining the queue. Throughput was measured on the Consumer thread. Although this queue supports batching, it was not employed here as we are interested in how efficiently the queue is synchronizing.

Producer count	Throughput (mill/sec)
1	74
2	59
3	43
4	40
6	56
8	65
10	66
15	68
20	68

5 LMAX - Disruptor:

Performance of LMAX Disruptor Queue is dependent on the WaitStrategy used. These numbers were taken for various wait strategies using 1 producer and 1 consumer only. On the consumer side BatchEventProcessor was used, due to lack of documentation on how to consume without batching.

WaitStrategy	Producer Mode	Throughput (millon/sec)
BlockWaitStrategy	SINGLE	11
	MULTI	10
TimeOutBlockingWait	SINGLE	10
	MULTI	6
LiteBlockingWaitStrategy	SINGLE	56
	MULTI	24
BusySpinWaitStrategy	SINGLE	85
	MULTI	25
SleepingWaitStrategy	SINGLE	88
	MULTI	28
YieldingWaitStrategy	SINGLE	88
	MULTI	**42**

Not surprising to see that SINGLE producer mode is faster than MULTI producer mode, even though only 1 producer was employed. In practice, for Storm only the multi-producer mode is applicable. YieldingWaitStrategy delivered the best performance at 42million/sec. Given the large number of wait strategies involved and since Disruptor was trailing in performance with 1 producer thread, I decided to skip benchmarking with more producers.

6 AKKA:

Strictly speaking Akka is neither a low-level message queue nor a high level streaming solution. It is a library that supports programming in terms of the actor model where concurrent actors are communicating with each other. It has its own internal queuing. Since it has built-in support for messaging between concurrent actors, there is inter-thread messaging happening. Some streaming solutions like Flink and Gearpump use AKKA to handle inter-thread messaging among other things. Consequently they are bound by AKKA's performance.

Throughput: As per AKKAs own published numbers, performance peaks at 50 million messages/sec using 96 actors (on 48 threads) on a single machine. Notice that with just 2 threads, Disruptor approaches this number and JCTools exceeds it.

7 JCTools vs Disruptor:

Although Disruptor is a fast queue, micro-benchmarking suggests it still trails behind JCTools. An important problem with Disruptor lies in its 'one-queue-for-all' design approach. This complicates the programming model a lot for end-users. For most queues, all you need to know is two methods: insert() and remove(). This is not the case with Disruptor.

To deal with Disruptor you need to understand RingBuffer, EventFactory, Executors, Wait strategies, EventProcessor, Sequences, SequenceBarriers, Disruptor and then optionally some functional/lambda style programming to minimize lines of code and make things readable. In newer versions, additional concepts like EventTranslators and EventPublishers have been introduced to manage the complexity ☺

JCTools takes a simpler approach. It simply provides different queues for different use cases. There is a different Q for each of these use cases: 1producer-1consumer, Nproducers-1consumer, 1producer-Nconsumers, Mproducers-Nconsumers. Most importantly, it retains the simple insert()/remove() programming model that we like and understand. It also is important to notice how the performance remains stable when concurrency level is increased.

8 Conclusion

This combination of state-of-the-art performance plus simple programming model makes JCTools a great choice.