Inter-thread communication efficiency

Jarosław Sagan[1]

[1] *Institute of Computer Science, Maria Curie-Sklodowska University, pl. M. Curie-Sklodowskiej 5, 20-031 Lublin, Poland*

Abstract – In this paper I compare inter-thread communication methods: blocking queue and LMAX Disruptor without synchronization according to a number of threads (CPU cores) and consumer rate. The research is carried out using a multiprocessor machine with Non Uniformed Memory and Oracle Java Runtime Environment. I determine if processing on many multi-core CPUs with NUMA is faster than on single multi-core CPU or vice versa.

Keywords: *multi-threading, multi-thread processing*

1 Introduction

A lot of computer systems use multi-thread processing. In the thread and locks concurrency model threads communicate with each other. In most cases parallel processing is much more efficient than single thread execution. But when thread tasks are short (high frequency computing), the thread communication time can be longer than the task execution time. It causes that the multi-thread execution time is longer or close to that of thread. To solve the problem the LMAX Disruptor pattern can be used. The LMAX Disruptor is a concurrent framework created to improve multi-thread high frequency processing efficiency. It dispose of synchronization which is time consuming. The thread synchronization problem is greater when time of consumer execution is short and there are many consumers. In this case, the time spent on synchronizing threads is close to the consumer execution time. Moreover, thread management by the operating system kernel causes loss of time. The kernel switches execution to another thread so CPU spends time on another thread and replaces some cache lines with the preemptive thread data. This causes that the total time of parallel execution is longer than the single thread processing. The disruptor was designed to be CPU cache-friendly [**1**].

The LMAX Disruptor technical paper contains comparison of array blocking queue and the LMAX Disruptor on a single CPU multi-core machine. There are five configurations of producer consumer connection: one producer to one consumer (unicast), one producer to pipeline of three consumers (pipeline), three producers to one consumer (sequencer), one producer to three consumers (multicast), one producer to two consumers that produces data to third consumer (diamond). In the LMAX Disruptor technical the paper are presented throughput results for the above five configurations without time consuming operations inside of the consumer handling [**1**]. I compare communication between producer and consumer via blocking queue with the Disruptor pattern according to a number of CPU cores. I focus on the total processing time, CPU usage and impact of multiprocessor NUMA environment on thread scaling. The research takes into account the consumer execution time.

2 Other concurrency models

There are many known concurrency models, but in this paper only one "threads and locks" implementation of producer consumer pattern is described. For example, instead of the "threads and locks" concurrency model one of the models: actors, communicating sequential process, data parallelism, map reduce can be used in concurrent application [**2**].

2.1 Producer consumer pattern

The standard inter-thread communication method commonly used in Java is the communication between the producer and consumer via a blocking queue. Using a blocking queue is simpler and less prone to bugs than waiting and notifing inside the producer and consumer [**3**]. Using the blocking queue pattern is presented in Figure 1. The producer thread calculates or obtains data from a resource and puts it into a queue to perform further processing parallel. Consumer threads obtain data from queue and process it. Each element from a queue is processed by only one consumer thread.

The most efficient queue in adding and removing elements is the `ArrayBlockingQueue`. The `ArrayBlockingQueue` size is fixed so queue internal object creation methods are not used. In most of the applications queues are empty or full. This is due to the fact that producer thread throughput is different from the total consumer threads throughput. When the queue is empty and consumer thread call take element method consumer thread is blocked until an element is added to the queue. When

the queue is full and producer thread call put element, producer thread is blocked until a consumer thread takes an element from the queue. `ArrayBlockingQueue` uses a lock to block taking and adding elements.

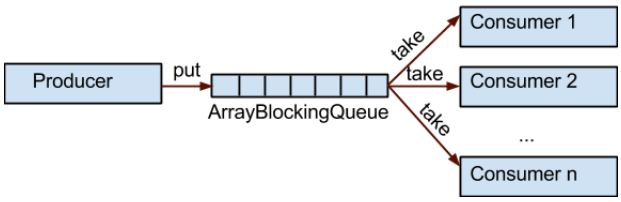

FIGURE 1. Producer consumer inter threads communication using `ArrayBlockingQueue`

2.2 LMAX Disruptor

Programming inter-thread communication with the LMAX Disruptor is completely different from that using the standard method. Instead of `ArrayBlockingQueue` Disruptor uses Ring buffer. The Ring buffer consists of Object array with the fixed size. The ring buffer size should be an power of 2 to make faster modulo operation on array index calculation. Ring buffer array elements are preinitialized on the disruptor start up. Array elements are created to increase the chance of continuity of data in memory. If sequentially processing data are continuous in memory, CPU cache is able to pre-fetch data to processing by CPU. The disruptor processing schema is presented on figure 2 [**1**].

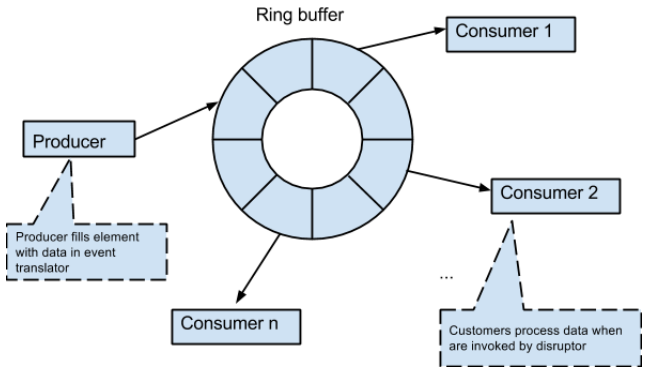

FIGURE 2. Producer consumer inter thread communication using the disruptor.

Implementation of producer using the disruptor pattern requires change of programmer thinking. The producer thread does not create data and pushes it into the queue. The producer call publish event with a given event translator. Than the even translator is called by disruptor to fill element data with new values. The event translator does not create a new object but only fills existing one with new data. This strategy reduces garbage creation (temporary objects to be removed by Garbage Collector in the future). It causes that the ring buffer data remain consistent in memory independent of producer call times [**1**].

The event consuming method (consumer) is also called by disruptor. The programmer must implement work handler. Work handler receives in the parameter data an element to be processed and the element sequence number. The consumer method should not create many local objects because of Garbage Collector. But when data are processing, other library methods are called. Those libraries may produce objects in young generation memory. In this research I created a worker that generates object in young generation memory.

3 Material and methods

The research is carried out on the scientific cluster node with two Intel(R) Xeon(R) CPU X5650 2.67GHz processor with Hyper Threading technology. Two processors are two NUMA nodes. Threads task does not perform IO operation. For each element received from the producer consumer threads creates a new local integer array with 100 consecutive numbers and sort it 1 or 15 or 300 times depending on configuration. It causes that this data must be swept by Java Garbage Collector. Local thread data are stored in young generation of Java Heap Space. These data should not be moved to old gen.

The research is carried out for 1 (high frequency), 15 (medium frequency) and 300 (low frequency) array sort times for both LMAX Disruptor, blocking queue and sequential. Sequential processing is single thread processing where the producer creates event and executes the consumer. The number of consumer threads increases from 1 to 11. Each thread is bound to next core. The producer thread is also bound to one core. When the consumer thread count is in the range from 1 to 5 only one NUMA node is used. When consumer thread count is larger than 5 processing is performed on two NUMA nodes. To bind threads to CPUs I use Java-Thread-Affinity library [**4**]. Hyper Threading technology creates two logical processors for each core. It provides increase of efficiency when threads perform I/O operation. In this research I/O operations are reduced so I use only one virtual CPU per core. Using one thread per core makes results analysis simpler. Measured values are wall-time and CPU usage. The CPU usage is a system plus user time divided by wall-time. In all configuration producer creates 40 000 000 events.

3.1 Results

In the Figure 3 execution time is compared for disruptor, standard and sequential for high frequency processing. High frequency processing using the standard blocking queue is less efficient than sequential processing. Processing using the LMAX disruptor is much faster than sequential and uses as many whole cores as threads. The blocking queue method can not use whole the CPU power, which is shown on Figure 6. Processing using one NUMA node is more efficient than using both nodes for the standard method and LMAX disruptor. The total processing time increases when processing on two NUMA nodes compared to processing on a single node. When processing on a single node all data or most of data are stored in the CPU cache. When processing on two nodes, data are shared between two processors. Processors may invalidate their cash lines mutually so they must fetch data from memory, which is time consuming, especially on NUMA.

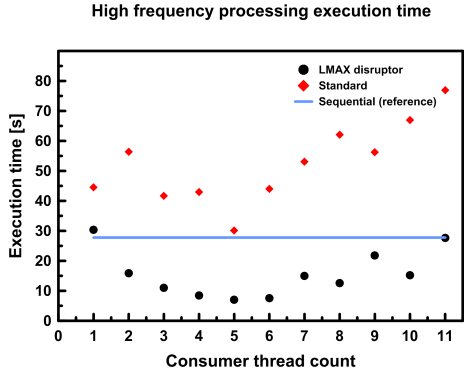

FIGURE 3. High frequency processing execution time comparison.

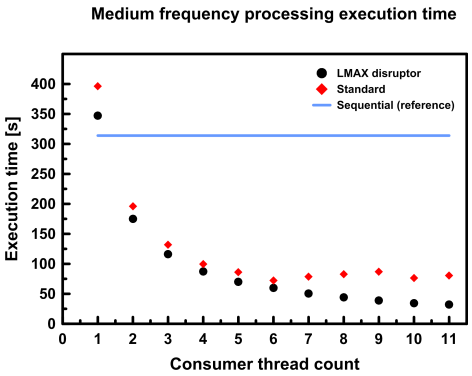

FIGURE 4. Medium frequency processing execution time compariso.

Medium frequency processing using the standard blocking queue is more efficient than sequential processing if only there is more than one consumer which is presented in Figure 4. Processing using the LMAX disruptor is a little more efficient than the standard method and uses as many whole cores as threads. The blocking queue method can not use the whole CPU power when processing on both NUMA nodes, as shown in Figure 7. When processing on one NUMA node the standard CPU usage method is as high as the disruptor one. Low frequency processing using the standard blocking queue is more efficient than the sequential processing if only there is more than one consumer as presented in Figure 5. Processing using the blocking queue is as that using the LMAX disruptor.

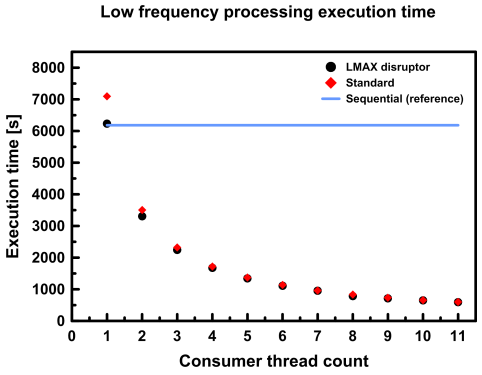

FIGURE 5. Low frequency processing execution time comparison.

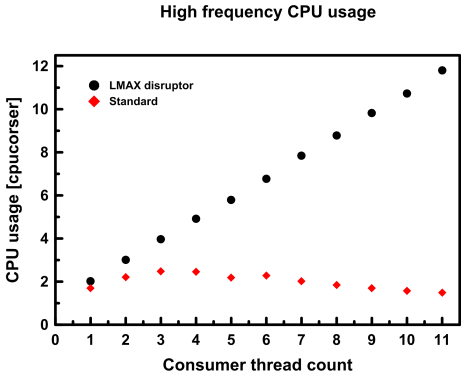

FIGURE 6. High frequency processing CPU usage.

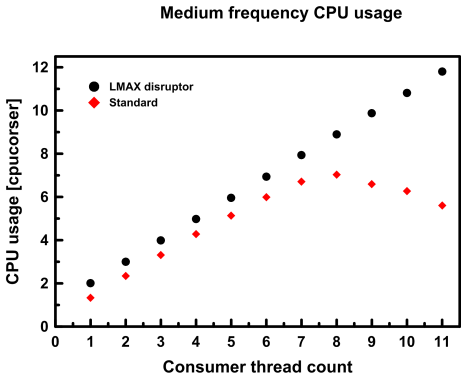

FIGURE 7. Medium frequency processing CPU usage.

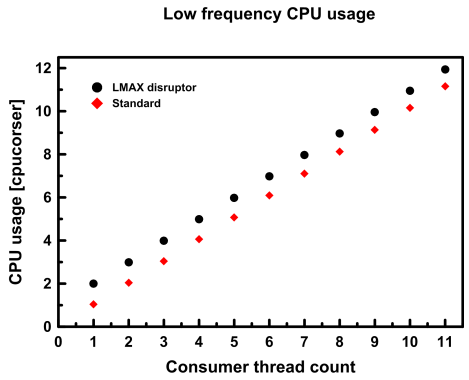

FIGURE 8. Low frequency processing CPU usage.

The standard method CPU usage is as high as disruptor (Figure 8).

4 Conclusions

Performance tests in this research are performed on two multi-core processors NUMA machine. Nowadays NUMA is a very popular architecture but it is worth performing similar tests on the uniform memory access multi-core multiprocessor machine.

For high frequency processing parallelization using the standard method is not as efficient as sequential processing or disruptor. The LMAX Disruptor is very efficient in high frequency processing. Binding all threads to only one NUMA node is more efficient than processing on two nodes. Processing using the LMAX Disruptor exploits as many whole CPU cores as threads. High frequency processing using the blocking queue does not exploit th CPU power. The LMAX Disruptor always uses as many whole cores as threads. The LMAX disruptor is better in high frequency processing. Medium and low frequency processing using the LMAX Disruptor is comparable to the blocking queue. Also in this case the LMAX Disruptor is slightly faster.

References

[1] Thompson Martin, Farley Dave, Barker Michael, and Gee Patricia andStewart Andrew. *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*. 2011.

[2] Butcher Paul. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Pragmatic Programmers, LLC, 2014.

[3] Eckel Bruce. *Thinking in Java. Fourth Edition*. Prentice Hall PTR Upper Saddle River, 2005.

[4] Java thread affinity https://github.com/peter-lawrey/java-thread-affinity.