Brief Announcement: MultiLane - A Concurrent Blocking Multiset

Dave Dice
Oracle Labs
dave.dice@oracle.com

Oleksandr Otenko
Oracle Corporation
oleksandr.otenko@oracle.com

ABSTRACT

We introduce an extremely simple transformation that allows composition of a more scalable concurrent blocking multiset, or *bag*, from multiple "lanes" of a potentially less scalable underlying multiset. Our design disperses accesses over the various lanes, reducing contention and memory coherence hot spots. Implemented in Java, for instance, we construct a multiset from multiple lanes of *java.util.concurrent.SynchronousQueue* [9] that yields more than 8 times the aggregate throughput of a single instance of *SynchronousQueue* when run on a 64-way Sun Niagara-2 system with 16 producer threads and 16 consumer threads. We experimented with various queues from *java.util.conconcurrent* and found that in general a MultiLane form will outperform its underlying counterpart.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Concurrency

General Terms

Performance, experiments, algorithms

Keywords

Concurrency, queues, bags, concurrent multisets, resource pools, producer-consumer, message passing

1. INTRODUCTION

Concurrent multisets allowing multiple producers and multiple consumers can implement message passing mechanisms or when provisioned with elements representing resources they can be used as concurrent resource pools. They are found with increasing frequency in modern software. Our construct exposes blocking *non-total* `take` and `put` accessor methods. `Take` waits for an element to become available and then removes and returns that element to caller, while `put` inserts a new element, respecting the collection's capacity bound, if any, by first stalling until the collection is not at full capacity before adding the element. All of the JDK packages mentioned above expose blocking `take` and `put` operators except *ConcurrentLinkedQueue*, which is based on Michael and Scott's classic non-blocking queue [8]. For that specific construct we emulate `take` and `put` by spinning on `offer` and `poll`.

A MultiLane collection has a *put cursor* and a *take cursor*, which reflect the lanes to which arriving `put` and `take` operations will be dispatched, and an array of lanes. Each lane, in turn, consists of an instance of an underlying blocking sub-collection. `Put` and `take` operations on a MultiLane collection will first increment the appropriate cursor. These are the only centralized read-write variables in our algorithm and are accessed with a simple atomic fetch-and-add primitive. Both cursors are initially 0. After advancing a cursor the operation uses the cursor value to select a lane index and then invokes `put` or `take`, respectively on the underlying sub-collection. Because of the disjunct between advancing the cursor and then accessing the sub-collection identified by the cursor value, our collection are not FIFO even if the sub-collections happen to be. In practice, however, many applications do not require FIFO ordering. MultiLane multisets can be either unbounded or bounded depending on the underlying sub-collection. The progress properties of the constituent lanes are not necessarily reflected in the aggregate multilane collection.

A MultiLane collection is work-conserving if the underlying collections are work-conserving. Specifically, we say a collection has a surplus of `takes` when the number of `take` invocations on that collection exceeds the number of `put` operations. If we have a surplus of `take` operations on a MultiLane collection then the put cursor will select a lane that itself has a surplus of `take` operations, facilitating the expeditious pairing of `put` and `take` operations. Complementary statements hold for the take cursor. The general approach is similar to that of a ring buffer, except that there are no locks at the top level but only atomically updated cursors, and the ring elements are themselves blocking concurrent collections instead of storage locations.

The key benefits to our approach, as compared to existing collections, are (a) reduced coherence traffic as we distribute operations over the lanes, and (b) by dispersing operations we lessen the impact of critical sections or optimistic concurrent windows that might exist within those underlying collections. Relatively simple and less-scalable thread-safe collections can be easily composed into scalable MultiLane collections. The atomically-accessed cursor fields certainly constitute a coherence hot-spot and impediment to ultimate scalability, but they appear to admit more scaling than the centralized data in the underlying collections. Furthermore, atomic fetch-and-add may confer performance advantages relative to compare-and-swap [3].

We note too that a multilane semaphore can be readily constructed from multiple lanes of potentially less scalable underlying semaphores.

2. RELATED WORK

The literature is rich with scalable concurrent queue algorithms. Of late and of particular interest, Hendler et al. [5][6] show to use *flat combining* to construct scalable synchronous queues. Afek et al. [2] introduce the concept of *Quasi-linearizable* data structures and use the concept to construct relaxed and highly scalable

SPAA'11, June 4–6, 2011, San Jose, California, USA.
ACM 978-1-4503-0743-7/11/06.

queues. Most relevant to our work are the Elimination-Diffraction trees (*ED-Trees*) of Afek et al. [1] that, like our approach, uses multiple sub-collections. Mellor-Crummey [7] and others [4][10] used atomic fetch-and-ϕ to implement non-blocking queues but their algorithms are more complex and require additional read-write accesses to central variables.

3. PERFORMANCE

In Figure 1 we report the performance of a microbenchmark that runs a number of concurrent producer and consumer threads and measures aggregate message throughput rates with varying multi-set implementations. Data was collected on a single-socket T5120 UltraSPARC-T2^{TM}"Niagara" system having 64 logical processors and 8 cores. The UltraSPARC-T2 has only two pipelines per core, so scaling above 16 threads is modest and arises only from memory-level parallelism. The MultiLane forms were configured with 8 lanes. Interestingly, we see that *LinkedTransferQueue* is faster than its MultiLane counterpart with 16 producer threads and 4 consumer threads. Further investigation showed that many messages were simultaneously in-flight under the MultiLane form, and that garbage collection activity dominated the measurement interval. This behavior arises because the underlying collection is unbounded and the underlying implementation allocates "container" nodes for each message, illustrating a potential confounding factor if we have unbounded collections and producer-consumer rate imbalance. (Augmenting the *LinkedTransferQueue* instances with semaphores to create bounded sub-collections provided relief, and, despite the overhead of the semaphores actually improved performance). Garbage collection overhead was negligible in all the other reported runs. To enable fair comparison, if the underlying form was bounded, as in the case of *ArrayBlockingQueue*, we reduced the bound by a factor of 8 for the MultiLane variation thereof, so the aggregate MultiLane would have the same effective bound at 8 lanes.

Implementation	4P : 16C		16P : 4C		16P : 16C	
	Base	MultiLane	Base	MultiLane	Base	MultiLane
ArrayBlockingQueue	509	653	1002	881	1017	11338
LinkedBlockingQueue	1312	4245	1183	4008	1123	10604
LinkedTransferQueue	6288	6737	6601	4425	5424	11376
ConcurrentLinkedQueue	2061	6404	3066	6326	1542	12134
SynchronousQueue	783	4945	671	5137	1177	9747

Table 1: Aggregate message throughput results shown in transfers completed per millisecond with varying implementations and producer:consumer ratios

4. REFERENCES

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. Euro-Par'10. http://dx.doi.org/10.1007/978-3-642-15291-7_16.

[2] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. PODC 2010.

[3] D. Dice. *Dave Dice's blog*, 2011 (accessed Feb 15, 2011). `http://blogs.sun.com/dave/entry/atomic_fetch_and_add_vs`.

[4] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5:164–189, April 1983.

[5] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA '10. http://doi.acm.org/10.1145/1810479.1810540.

[6] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing*. 2010. http://dx.doi.org/10.1007/978-3-642-15763-9_8.

[7] J. M. Mellor-Crummey. Concurrent queues: Practical fetch-and-ϕ algorithms. 1987. University of Rochester Computer Science Technical Report # 229.

[8] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. PODC '96. http://doi.acm.org/10.1145/248052.248106.

[9] W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable synchronous queues. *Commun. ACM*, 52:100–111, May 2009.

[10] J. Wilson. *Operating System Data structures for Shared-Memory MIMD Machines with Fetch-and-Add*, 1988. PhD Dissertation, New York University.

APPENDIX

Listing 1: MultiLane Algorithm

```
1  public class MultiLane< T > {
2    // Implements a concurrent blocking multiset −− bag
3    // Transforms existing blocking multisets into multilane forms
4    // Exposes take() and put() accessor methods.
5
6    // Lanes: Array of underlying blocking concurrent collections ...
7    // Possible examples of sub−collection types include :
8    // ArrayBlockingQueue, LinkedBlockingQueue, LinkedTransferQueue,
9    // SynchronousQueue etc
10   // This particular example employs SynchronousQueue.
11   private final SynchronousQueue<T> [] Lanes;
12
13   // PutCursor and TakeCursor are write and read "cursors" that chase
14   // each other.
15   // These are the only central read−write variables in our algorithm.
16   // Invariant: the generated indices must follow the same trajectory
17   // The stream of indexes generated by PutCursor and TakeCursor does _not
18   // need to be strictly cyclic, and in fact will not be when the PutCursor
19   // and TakeCursor overflow and change sign. That's benign.
20   // Progress property : obstruction within a lane impacts only that lane.
21   // Invariant: if there are any "written" lanes in the MultiLane collection then
22   // the lane identified by TakeCursor is written. "Written" means that the
23   // sub−collection at that specified lane has at least one available element,
24   // or that some arriving put() has advanced PutCursor and is poised to put()
25   // into that lane. Take() may thus pair−up promptly if put messages are
26   // available. Complementary invariants exist for readers.
27   // The general approach is similar to that of a ring buffer, except that
28   // there are no locks at the top level but only atomically updated cursors, and
29   // the ring elements are themselves blocking concurrent collections
30   private final AtomicInteger PutCursor = new AtomicInteger();
31   private final AtomicInteger TakeCursor = new AtomicInteger();
32
33   public MultiLane (int Width) {
34     // For brevity of explication require power−of−two Width value
35     // That allows efficient indexing of the form : Lanes[i & (Lanes.length−1)]
36     assert (Width & (Width−1)) == 0 && Width > 0 ;
37     Lanes = (SynchronousQueue< T >[]) new SynchronousQueue[Width];
38     for (int i = 0; i < Width; i++ ) {
39       Lanes[i] = new SynchronousQueue<T>();
40     }
41   }
42
43   public void put (T v) {
44     final int curs = PutCursor.getAndIncrement() ; // atomic fetch−and−add
45     Lanes [curs & (Lanes.length−1)].put(v) ; // put() is blocking
46   }
47
48   public T take() {
49     final int curs = TakeCursor.getAndIncrement(); // atomic fetch−and−add
50     return Lanes[curs & (Lanes.length−1)].take() ; // take() is blocking
51   }
52 }
```