Article title: Re-Implementation of BQ: A Lock-Free Queue with Batching Using MR Locks

Authors: Benjamin Rhiner[1], Joshua Howell[2], Rocco DiGiorgio[3]

Affiliations: University of Central Florida[1]

Orcid ids: 0000-0002-6176-6988[3]

Contact e-mail: cldigiorgio@knights.ucf.edu

License information: This work has been published open access under Creative Commons Attribution License <http://creativecommons.org/licenses/by/4.0/>, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. Conditions, terms of use and publishing policy can be found at <https://www.scienceopen.com/>.

Preprint statement: This article is a preprint and has not been peer-reviewed, under consideration and submitted to ScienceOpen Preprints for open peer review.

DOI: 10.14293/S2199-1006.1.SOR-.PP5TSMV.v1

Preprint first posted online: 23 March 2020

Keywords: parallelism, data structure, parallel, queue

Re-Implementation of “BQ: A Lock-Free Queue with Batching” Using MR Locks

Benjamin Rhiner

University of Central Florida

brhiner@knights.ucf.edu

Joshua Howell

University of Central Florida

Josh.howell@knights.ucf.edu

Rocco DiGiorgio V

University of Central Florida

cldigiorgio@knights.ucf.edu

Introduction

This implementation is based on a publication by Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank in 2018. [1] Titled “BQ: A Lock-Free Queue with Batching”, they describe the data structure’s functionality and correctness, along with the previous work it is based on. [1] All implementation details are from this paper unless otherwise stated. In this series of papers, we will be reimplementing and performance testing their data structure. This first paper covers an initial implementation involving no concurrency, using an MR Lock [2] to restrict threads’ access to the data structure. This ensures correctness as the operations on the queue are guaranteed to be mutually exclusive.

Our source code is available here: <https://github.com/ClayDiGiorgio/ParallelTeam24>

To compile the following implementations, use the command-line arguments below:

```
g++ BQSequential.cpp -std=c++11
g++ -pthread BQLocked.cpp -std=c++11
```

Description of the Data Structure

A BQ is a thread-safe queue that differs from a normal thread-safe queue in that it supports batch operations. Batch operations work like this: a thread reports to the queue that it would like to enqueue a value or dequeue a value at some point. The queue returns a log (known as a “future”) of this request. When the thread is ready to execute those future operations, (or when it wants to enqueue or dequeue immediately) the queue performs those requests with the help of other threads. The original thread can then get the results of its operations from the futures originally returned by the queue. [1]

Structures

The queue is stored in a linked list that maintains a global-accessible head and tail pointer. It also contains the count of the head and tail to keep track of the size of the list. The head node in this structure is used as a sentinel node, meaning that the data stored in the node is not a part of the queue itself. [1]

Each individual thread has its own ThreadData struct, which contains its own collection of operations, and its own linked list of enqueued elements. It also keeps track of the number of excessDequeues, which is used to ensure that linearizability is maintained. [1]

List of BQ's Public Methods

`enq()`

Immediate enqueue. Executes stored batch before executing itself.

`deq()`

Immediate dequeue. Executes stored batch before executing itself.

`futureEnq()`

Report to the queue that an enqueue will happen at some point. Returns a log of this notification.

`futureDeq()`

Report to the queue that a dequeue will happen at some point. Returns a log of this notification.

`execute()`

Performs the saved `futureEnq()` and `futureDeq()` operations, clears the batch. To do this, it calls two functions, `executeBatch()` and `pairFuturesWithResults()`. It then resets all the counts and pointers of the thread

List of BQ's Private Methods

`executeBatch()`

Appends the threads enqueue list to the end of the shared queue, effectively enqueueing all of them at once, then calls `updateHead()`

`updateHead()`

moves the shared thread's head down a number of nodes equivalent to the number of successful dequeue operations. This leaves the shared thread up to date with all of the futures that this thread has called.

`pairFuturesWithResults()`

Iterates through the list of futures and marks them as completed, as well as matching the dequeue futures with their result. It does this in a way that the order of the calls is maintained, including returning null for dequeue calls that would have been called on an empty queue.

`init()`

Performs setup of all local threads and sets initial values of the BQ.

BQLocked.cpp

The modification of the BQ data structure to work non-concurrently with an MR Lock is our original work, though the BQ and the MR Lock definitions are by their respective authors. [1] [2] BQLocked.cpp is an almost carbon copy of BQSequential but with a few alterations to implement locking functionality using the provided mrlock.h and bitset.h header files. [2] The init() function has been updated to include an instantiation of the MRLock class with two resources. As there are only two resources, head and tail, which require locking, the resources are controlled through an enum, LockType, defined in BQStructs.h. This enum maps the head, tail, and both, to 1, 2, and 3, respectively. The enq() and deq() now lock the TAIL and HEAD, whereas a call to execute(), which handles batch operations, locks BATCH (both head and tail).

Performance

BQLocked.cpp and BQSequential.cpp also include a main() function with code for performance testing. In both, we initialize an array of values from which we can select for enqueue operations. Each test randomly selects between performing a single enqueue, dequeue, or batch of futures 400 times. For BQSequential, this is done in the main function. In BQLocked, each of the 4 spawned threads does $\frac{1}{4}$ of these total operations.

This test function initializes an array of values for future enqueue operations and four threads with a test function. The test function performs a random mix of operations between individual enqueues, dequeues, and batches. Whenever a batch is performed, a randomized 10-length sequence of enqueue and dequeue Futures are created and executed. It is impossible to preallocate Futures prior to testing without substantially altering the data structure. As such, Futures are allocated as futureEnq() and futureDeq() operations are called, as the structure defines, and the execute() function will execute all futures in the thread. With pre-allocation, threads would be able to update and change the Futures of other threads.

Sequential vs Parallel Execution of BQ

Each main() method was called 10 times and the average runtimes for each is graphed to the left.

As we can see, the sequential version of the queue performed substantially better than the parallel version. We attribute this to the wait time experienced due to locking. However, we can see that the wait time overhead introduced by locking is substantially reduced as operations increase. Where both implementations scale more or less linearly, the linear implementation has a steeper slope.

References

- [1] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18). Association for Computing Machinery, New York, NY, USA, 99–109. DOI:<https://doi.org/10.1145/3210377.3210388>
- [2] Deli Zhang, Brendan Lynch, Damian Dechev, Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors, In Proceedings of 17th International Conference on Principles of Distributed Systems (OPODIS 2013), Nice, France, December 2013.
- [3] Deli Zhang, Pierre LaBorde, Lance Lebanoff, Damian Dechev, Lock-free Transactional Transformation, ACM Transactions on Parallel Computing (ACM TOPC), Vol. 5, No. 1, Article 6, June 2018.
- [4] Christina Peterson, Pierre LaBorde, Damian Dechev, CCSpec: A Correctness Condition Specification Tool, In the 27th IEEE/ACM International Conference on Program Comprehension (ICPC '19), Montreal, QC, Canada, May 2019.
- [5] Ramin Izadpanah, Steven Feldman, Damian Dechev, A Methodology For Performance Analysis of Non-Blocking Algorithms Using Hardware and Software Metrics, In Proceedings of the 19th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (IEEE ISORC 2016), York, UK, May 2016.
- [6] Christina Peterson, Victor Cook, Damian Dechev, Practical Progress Verification of Descriptor-Based Non-Blocking Data Structures, In Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019), Rennes, France, October 22-25, 2019.
- [7] Steven Feldman, Pierre LaBorde, Damian Dechev, *Tervel: A Unification of Descriptor-based Techniques for Non-blocking Programming*, In Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV), Samos, Greece, July 2015.