Magical Concurrent Double-Ended Queue

A re-implementation of "An Unbounded Nonblocking Double-Ended Queue"

Soliman Alnaizy Ryan Dozier Natasha Zdravkovic

University of Central Florida
Orlando, Florida, 32816

1 Abstract

The investigated algorithm modifies the existing bounded deque method implemented by Herlihy, Luchangco, and Moir to be obstruction free. It avoids interference on the ends of the queue, which allow this new algorithm to be unbounded. This algorithm supports the correctness and progress principle by prescribing compare-and-swap for any steps that may lead to an incorrect result. Due to the double-sided implementation of this queue, the actions performed at one side are symmetric to the actions performed to the other.

2 Related Works

Due to the abstract nature of queues, there are numerous different ways to implement a double-ended queue. The one that's most similar to our implementation would be the HLM array-based double ended queue [1]. The HLM deque is considered to be bounded and does not allow for expansion once it reaches capacity. Our implementation will borrow from their techniques and apply it to a double linked list, resulting in an unbounded deque that could unconditionally expand and shrink to accommodate for the number of *push()* calls.

3 Introduction

A double-ended queue supports push and pop operations at both ends. However, this queue differs from previous implementations due to its unbounded nature. This queue supports expansion due to the linkage between nodes, and every node stores a linear array of slots which is similar to the HLM queue that it is based on.

The slots contain left-null and right-null tuples on its boundaries, in effect encompassing the data tuples which are continuous on its interior. The borders of the node contain either a NULL or a pointer to an adjacent node. It is in this way that the nodes are connected to allow for an unbounded queue.

With such expansion of the queue possible, the boundary conditions are the areas in which the most problems arise when dealing with concurrency. The interior of the queue follows the simple storage as the push and pop operations deal with the ends. Thus, most of the algorithm will focus on the boundaries at the end of the whole queue and the linkages between the nodes. It is in these boundary conditions that the principles of correctness and progress must be analyzed to assure that the the implementation of the queue is adequate.

The advantages of such an implementation lie in its unbounded nature. It provides a lock-free environment that deals with only the simplest of hardwares, focusing intensely on compare-and-swap. This allows the algorithm to have portability as most machinery supports these operations.

The major disadvantage of this implementation is that it is only obstruction-free, which requires a live-lock prevention manager. Live-lock is when the states of the thread are constantly changing but no threads are allowed to make progress. This guarantee is the weakest of the guarantees (the others being deadlock free and starvation free). Moreover, the paper dictates that this obstruction-free guarantee is only valid when there is no contention, which is likely to happen when dealing with multiple threads that are attempting to access the same object.

However, the advantages that this algorithm brings outweighs the disadvantage of a slightly weaker guarantee. There is a high overhead due to keeping track of all the edges and the connections between the slots. Due to this, another implementation would go towards creating a doubly-linked list that would perform like a queue. This would eliminate the need for the edge hints, and it would also eliminate the arrays of slots inside each node. Instead, each slot would become a node itself. With this doubly-linked list, there is more references to create the linkages in the nodes, but there is decrease in algorithm complexity. With this in mind, we also attempted to replicate a concurrent doubly-linked list.

4 Algorithm Analysis

The unbounded queue modifies an existing queue by retracting its boundaries. While it still holds the array-style of slots, the new algorithm places an array in each node, carefully linking the nodes together in a doubly-linked list. With this technique, the overall dequeue can span multiple nodes, and the only indication of a current boundary is the placement of LN (left-null) and RN (right-null) in the data slots. Each node will either have a pointer to an adjacent node or will have a pointer to NULL indicating that it is the end of the dequeue.

To find the edges between datum and null, the algorithm introduces a new method called oracle() which will find the exact slot for the next push to be contained, or the slot at which a pop will reset to NULL. The algorithm also introduces sealed nodes, which will prevent outward growth in their direction. In effect, these sealed nodes are nodes that are labelled as logically deleted.

4.1 Progress

The algorithm implements concurrent behavior without the calling for a lock. This indicates that it can be labelled as "lock-free" which states that some thread will be able to complete a method call in a set amount of steps.

To ensure against deadlock, an informal proof was written while analyzing the three mandatory loops that form the push, pop, and oracle method. In these loops, only the push operation would result in a retry when the node it is accessing is a sealed one. However, in the other methods, multiple transitions are given which allow each one to end in a successful method.

Thus the paper concludes that in the absence of contention, the algorithm is obstruction-free as methods in all times will point the thread back to the active chain and lead to successful operations.

In our implementation of this algorithm, we also did not use a lock. We followed the paper's description and used a compare-and-set architecture and allowed each thread to continually loop for access until they are successful in finishing their operation. Our implementation of this created an infinite while loop with multiple compare-and-swaps which allow the thread to continually make progress.

4.2 Correctness

To prove that the algorithm follows the principle of safety, it is necessary to determine whether the outcome of the algorithm in parallel would produce the same outcome in sequential code. For this, the identification of linearization points will demonstrate that the algorithm has the same observable behavior as a sequential algorithm.

The algorithm relies on a double Compare and Set (CAS) protocol to ensure the correctness condition. The counter in the next or previous Node is incremented to mark that it is in use, thus failing any in progress CAS calls onto that node, then the references are swapped. After the swap is made a further check is done to ensure that the new Node points back the the previous one.

Our implementation of this algorithm introduced multiple compare-and-swaps which allowed for the threads to check that they are in the appropriate position. With the algorithm in the paper, it is necessary for each thread to compare the object that they are performing an operation on, and also update the hints and the edge indexes almost simultaneously. Thus, we established multiple atomic operations to ensure that the thread is producing correct results.

4.3 Synchronization Techniques

The algorithm does not depend on a lock, which is why it can be labelled as non-blocking. However, it does require compare-and-set architecture which will allow certain lines to run atomically. To run atomically means that certain lines are synchronized so that no other thread can enter memory to update those values. The use of atomic structures also prevents threads from storing certain variables in their local cache where the value can be invalidated quickly. While this increases time, due to having to access main memory, it is necessary to prevent incorrect values in memory and invalidation of the local thread's cache.

In our implementation, we introduced AtomicReference Objects which would encapsulate the ends of the double-ended queue. This ensures that, since the ends are the places where the operations will be performed, that only one thread has access to modify such a datum.

5 Implementation

Our first implementation began with a reconstruction of the HLM queue since this unbounded double-ended queue is essentially a linkage of HLM queues. The HLM algorithm consisted of a slot class which contained the data that is being inputted and the number corresponding to the number of operations done towards the data. The actual HLM deque consisted of an array of these slots and the placement of the first data was in the middle of the queue as per instructions of the overview in the paper. This placement allowed for the queue to be double ended as pushes and pops could be done from both directions. With the pop operations, special consideration was taken into account to rebalance the queue and replace the nulls on the appropriate sides.

The next step in our implementation methodology was the linkage of the HLM data structure to, in effect, create a sequential version of the double-ended queue described in this paper. With the linkages of such nodes that contain the HLM data, it was important to implement hints which would keep track of which node was being accessed for the appropriate push and pop operations. In our implementation, we constructed a new class called Hint which would store the queue and the location that it is referring to, which is a deviation from the pseudo code in the paper.

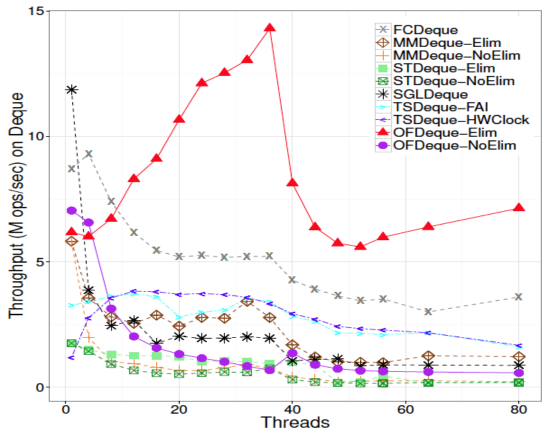
Finally, we ended on the creation of a concurrent version. The concurrent version makes use of AtomicReference objects to store the hints necessary for determining the location of pushes and pops. By using the atomic references, we take advantage of the compareAndSet method for swapping the hints with every change of a push and pop. We also use atomic references for the slots to ensure that only one thread is accessing the data at the endpoints at a time.

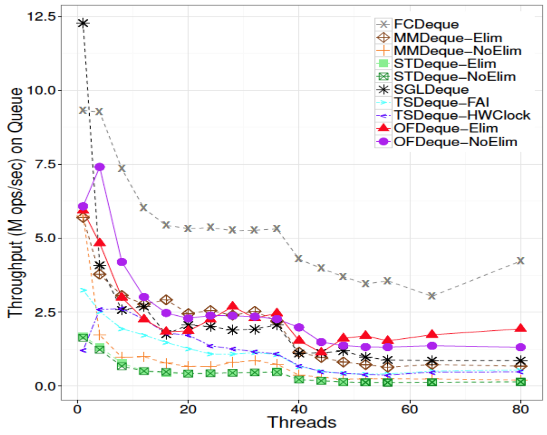
5.1 Obstacles

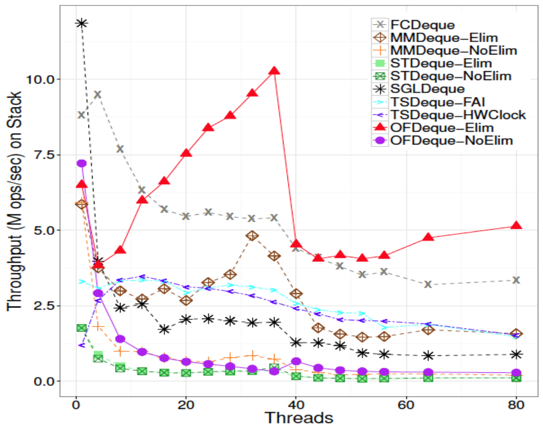
The supplied pseudo-code in our paper gives a general idea of how to implement the data structure. However, there are sections which are hard to understand as some functions were only given a brief summary instead of a true algorithmic approach. Another difficulty was determining what fields needed to be atomic as the algorithm does not explicitly state which fields need this protection. Through some deduction and trial/error we were able to determine what pieces of data must behave in an atomic way to ensure the correctness condition.

5.2 Performance Evaluation

The paper describing the algorithm presented their own experimental results comparing the algorithm's performance in regards to similar algorithms' performances. We attempted to copy their testing procedure; however, the exact proportions of their procedure were vague and unclear. Instead, we produced a randomized testing evaluation where the results plotted where the average of five trials. The paper tested the push operations solely, and we randomly selected with side to push on our deque. The paper also tested pop operations, and we randomly selected which side to perform an operation on and which operation to perform. And finally, the paper performed an evaluation on just the stack capabilities of the deque. In this evaluation, we randomly pushed and popped only on the left side of the deque. The results of our re-implementation followed closely with the paper's.







The above graphs are the performance results from the referenced paper. They represent a varying number of threads which multiple different algorithms of a deque. The paper's algorithm is OFDeque-NoElim.

For our testing environment, we used Windows 10 operating system with a quad core Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz and 8.00GB of RAM. We also made sure that the every time we ran the program, the same background apps were running.

6 STM Implementation

For the Software Transactional Memory Library, our source code implements Multiverse. Inside the Multiverse library is the TxnOject which every object must implement to ensure that there is atomicity, consistency and isolation. The actual mechanism runs in-memory which is how it satisfies these properties. Every operation that our re-implementation uses must be placed in a critical section so that only one thread can access a specific object at a time. The method that this uses is the StmUtils.atomic() method. Only one thread should be able to execute these sections at a time, and this method will guard it. Our implementation makes use of the Maven Dependency which allows insertion of the multiverse into our pom, and thus our program. With the Maven Dependency it is necessary to convert the algorithm's files into a project and then run the project.

6.1 Obstacles

A major obstacle encountered during the implementation of STM was the lack of recent resources for STM in Java. The Multiverse library, along with the Maven Dependency, was last edited in April of 2012. All other similar libraries were visited less recently. Taking into consideration the fast development time of software, the old creation date brought apprehension.

Furthermore, there was a lack of resources on implementing the Multiverse as it is not a library funded by Java. This created a lot of pitfalls as we attempted to install all the necessary components to ensure that the library could be functional in our testing environment. The lack of resources also lead to confusion on where the files should be placed as downloading the Maven Dependency immediately created multiple folders in the project.

6.2 Performance Evaluation

When we increase the size of the transactions, the graph still follows roughly the same pattern, with a logistic growth curve. The increase in the beginning due to the increasing number of threads results in exponential growth before the graph becomes asymptotic with the spawning of more threads. As the transaction size increases, this slope becomes shallower and the graphs seem to approach more of a linear approach. This can be attributed to the fact that STM provides synchronized blocks. With an increase in transaction size, more operations are done within a critical section that can only be accessed by one thread. At this point, there is a lack of contention for access as one thread is only allowed to perform at a time. Only when one thread completes their block can another thread wish to enter the same section.

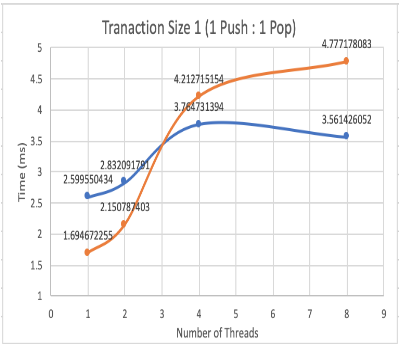
6.3 Modification

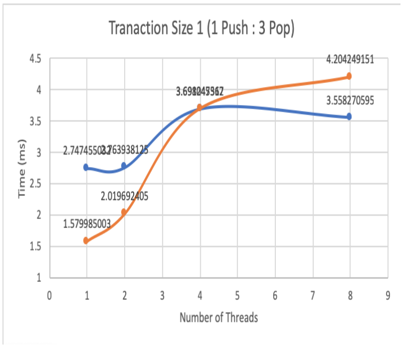
We attempted to try to improve the performance of this algorithm by removing the array in each slots and instead making each slot a node. This would allow less overhead in searching for edges, and we had hoped that this would allow the STM to be more efficient as each transaction would only call for one main operation instead of searching. Our results are shown below. We optimized the *push()* and *pop()* functions by reducing the transaction size.

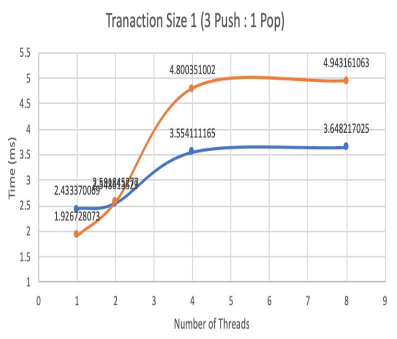
The usage of STM implies that there is a guarantee of progress and correctness. Only the head and tail will ever be modified in this method, similarly to how only the outer array slots would be modified in the original STM version. The modification does not modify which elements must be atomically minded, but instead only focuses on eliminating transaction size by removing the need for searching for the outer edge. The elimination of the Hint class results in only a necessary atomic pointer to the head and tail.

7 Comparison

We performed the same tests on both the concurrent version of the algorithm and the version of the algorithm that is implementing STM. We also observed that when there is a minimum number of threads, the STM outperforms the concurrent version. When the thread count reaches a critical amount, between two to four, the concurrent version outperforms the STM version. This can be attributed to the fact that at small thread amounts the STM handles the contention easier since it has to account for less comparisons with the threads. However, at a higher thread count, the concurrent outperforms due to being able to easily allow the thread to enter the function and pause instead of not being able to enter at all. The blue lines represent the STM version, while the orange represents the concurrent version.

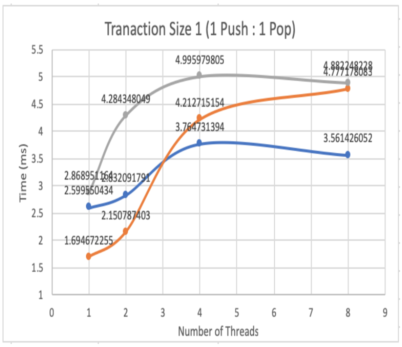


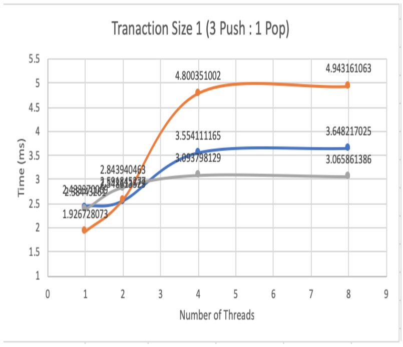


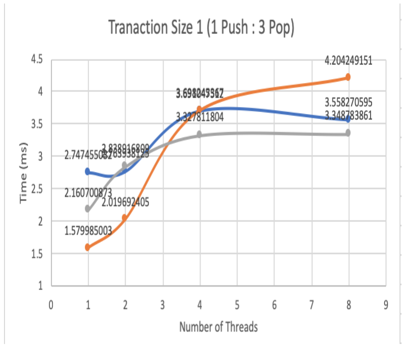


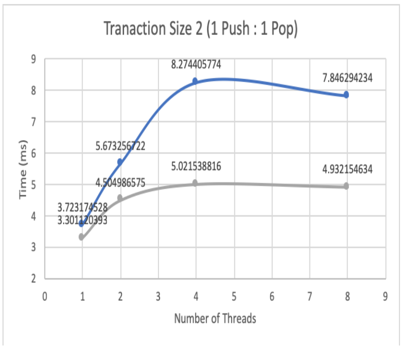
Based on additional performance data, our modified STM approach outperforms the original STM in some aspects, but not in others. However, the results follow similar curvatures which indicate that the time increase evolves from increasing thread sizes. Based on the sporadic ratio of when the modified outperforms the original, we cannot at this time address a pattern for the performance. We can only conclude that the potential ordering of pushes and pops create a difference in performance.

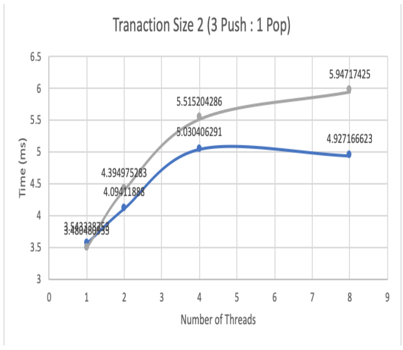
8 Performance Data and Analysis

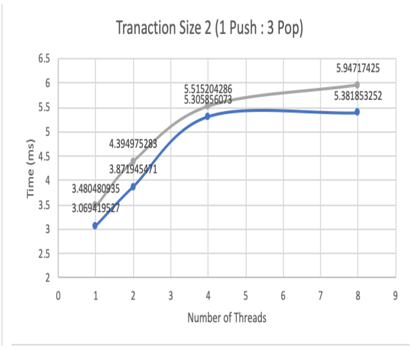


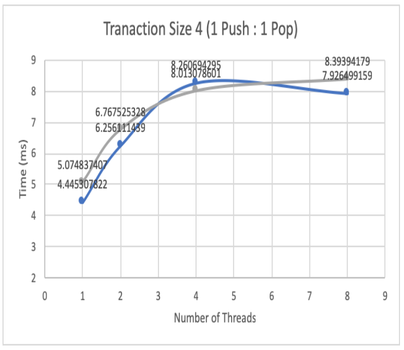


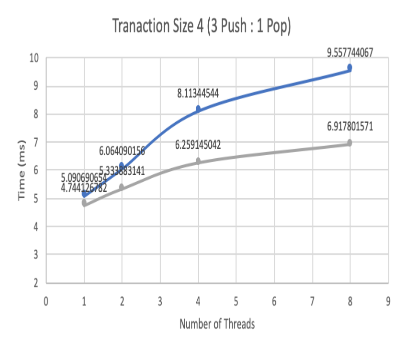


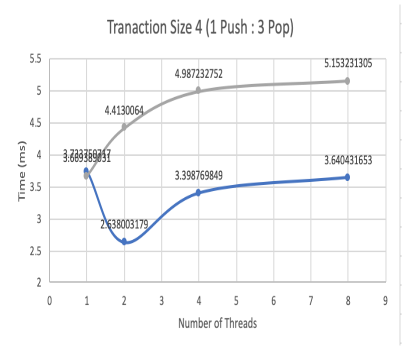












The orange lines indicate the concurrent method. The gray lines represent the unmodified STM. The blue lines represent our modified STM approach.

9 References

[1] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in Proc. of the 23rd Intl. Conf. on Distributed Computing Systems, ser. ICDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 522–.

[2] M. Graichen, J. Izraelevitz and M. L. Scott, "An Unbounded Nonblocking Double-Ended Queue," 2016 45th International Conference on Parallel Processing (ICPP), Philadelphia, PA, 2016, pp. 217-226.

[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.