Thesis Proposal

Hossein Naderi Eric Ruppert

March 1, 2022

1 Introduction

Shared data structures have become an essential field in distributed algorithms research. We are reaching the physical limits of how many transistors we can place on a CPU core. The industry solution to provide more computational power is to increase the number of cores of the CPU. This is why distributed algorithms have become important. It is not hard to see why multiple processes cannot update sequential data structures; for example, consider two processes trying to insert some values to a linked list simultaneously. One solution is to use locks; whenever a process wants to do an update query on a data structure, it locks it, and others cannot use it until the lock is released. Using locks has some disadvantages; for example, one process might be slow, and holding a lock for a long time prevents other processes from progressing. Moreover, locks do not allow complete parallelism since only one process holding the lock can make progress in the moment. Our approach is to create a lock-free data structure that developers can safely use without the disadvantages of locks.

The question that may arise is, "What properties matter for a lock-free data structure?". Since executions on a shared data structure are different from sequential ones, the correctness conditions also differ. To prove a concurrent object works perfectly, we have to show it satisfies safety and progress conditions. A safety condition tells us that the data structure does not return wrong responses, and a progress condition indicates that operations eventually terminate.

The standard safety condition is called *linearizability*, which ensures that for any concurrent execution on a linearizable object, each operation should appear to take effect instantaneously at some moment between its invocation and response. Figure 1 is an example of execution on a linearizable empty queue. The arrow shows time, and rectangles show the time between invocation and termination of an operation. Since `Enqueue(A)` and `Enqueue(B)` are concurrent, `Enqueue(B)`

may or may not have taken effect before `Enqueue(A)`. Execution in figure 2 is not consistent since `A` has been enqueued before `B`, so it has to be dequeued first.

Figure 1: An example of a linearizable execution. Both could take effect first since `Enqueue(A)`, and `Enqueue(B)` are concurrent.

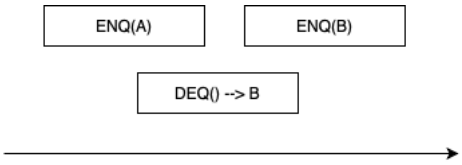


Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A`.

An algorithm is *wait-free* if each operation terminates after a finite number of steps. We call an algorithm *lock-free* if, after a sufficient number of steps, one operation terminates. A wait-free algorithm is also lock-free.

2 Previous Work

In the following paragraphs, we look at previous lock-free queues. Michael & Scott [1] introduced a lock-free queue which we refer to as the MS-queue. It is included in the standard Java Concurrency Package. Their idea is to store the queue elements in a singly linked list. Head points to the first not dequeued node in the linked list and tail points to the last element in the queue. To insert a node to the linked list, they use atomic primitive operations like LL/SC or `CAS`. If p processes try to enqueue simultaneously, only one can succeed, and the others

have to retry. This makes the worst-CASe number of steps to be $\Omega(p)$ per enqueue. Similarly, dequeue can take $\Omega(p)$ steps.

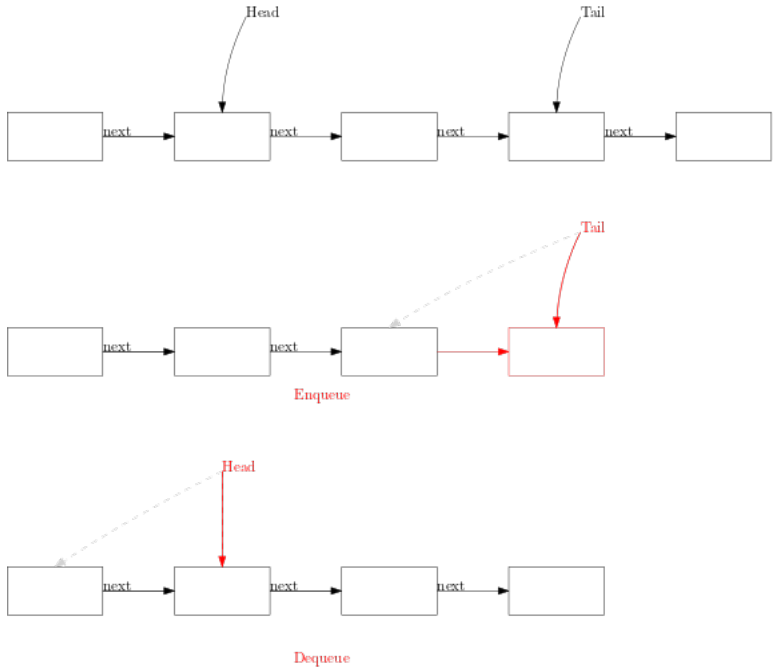


Figure 3: MS-queue structure, Enqueue and Dequeue operations

Moir, Nussbaum, and Shalev [2] presented a more sophisticated queue by using the elimination idea. The elimination mechanism benefits the dual purpose of allowing operations to complete in parallel and reducing contention for the queue. An Elimination Queue consists of an MS-queue augmented with an elimination array. Elimination works by allowing opposing operations such as consecutive enqueues and dequeues in the elimination array to exchange values when the queue is empty. Their algorithm makes it possible for failed aged operations to complement. Empirical evaluation of their work is better than MS-queue, but the worst CASe is still the same; in CASe there are p concurrent enqueues, their algorithm is not better than MS-queue.

Hoffman, Shalev, and Shavit [3] tried to make the MS-queue more parallel by

introducing the Baskets Queue. Their idea is to allow more parallelism by treating the simultaneous enqueue operations as a basket. Each basket has a time interval in which all its nodes enqueue operations overlap. Since the operations in a basket are concurrent, we can order them in any way. Operations in a basket try to find their order in the basket one by one by using `CAS` operations. However, like the previous algorithms, if there are still p concurrent enqueue operations in a basket, the amortized complexity remains $\Omega(p)$.

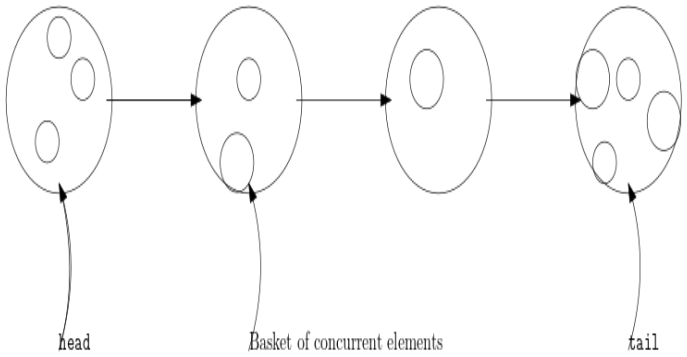


Figure 4: Baskets queue idea

Ladan-Mozes and Shavit [4] presented an Optimistic Approach to Lock-Free FIFO Queues. They use a doubly linked list and do fewer compare and swap operations than MS-queue. But as before, the worst `CAS`e is when there are p concurrent enqueues which have to be enqueued one by one. The amortized worst-`CAS`e complexity is still $\Omega(p)$ `CAS`es.

Hendler et al. [5] proposed a new paradigm called flat combining. Their queue is linearizable but not lock-free. After adding an operation acquiring the lock, they maintain history in publication records and compute all active operations responses. They believe their algorithm in real-world assumptions works well.

Gidenstam, Sundell and Tsigas [6] introduced a new algorithm using a linked list of arrays. Global head and tail point to arrays and content of arrays are marked if dequeued and written by `CAS` operations. Their data structure is lock-free, but

it updates lazily. Threads have a cache view on the queue and update it if it gets old.

Kogan and Perank [7] introduced wait-free queues based on MS-queue and use Herlihy's helping technique to achieve wait-freeness. Their step complexity is $\Omega(p)$ because of the helping mechanism.

Milman et al. [8] designed BQ: A Lock-Free Queue with Batching. Their idea of batching allows a sequence of operations to be submitted as a batch for later execution. It supports a new notion introduced by the authors called Extended Medium Futures Linearizability.

In the worst-`CAS`e step complexity of all the papers discussed above, there is a p term which comes from the `CAS`e all p processes try to do an enqueue at the same time. We are focusing to see if we can do this in sublinear steps in terms of p or not.

3 Our contribution

In this work, we design a queue with $\log^2 p + Q$ steps per operation, outperforming queues built by the LinkedList approach. Our idea is similar to Jayanti 's Multi Enqueuer-Single Dequeuer Queue [9], but we do not use `CAS` operations with big words. The new work about our idea is that even if there are p processes trying to enqueue or dequeue simultaneously, each operation takes $\log^2 p + Q$. Jayanti proved an $\Omega(\log p)$ lower bound on the worst-`CAS`e shared-access time complexity of p-process universal constructions [10]. He also introduced a construction that achieves $O(\log^2 p)$ shared accesses [11]. We introduce a data structure that makes processes agree on the linearization ordering using $O(\log p)$ `CAS` per operation called *Block tree*. Then we use the block tree as a stepping stone towards our queue algorithm. A block tree is a tournament tree shared among p processes (See

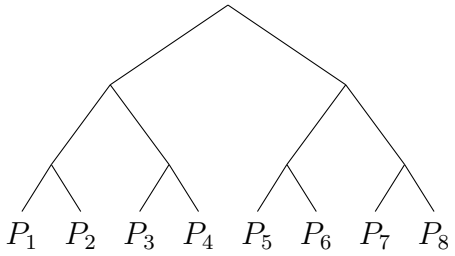
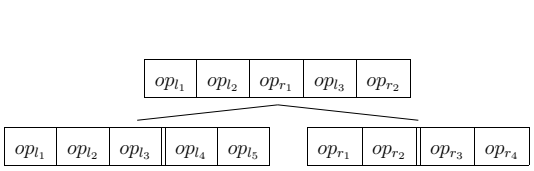


Figure 5: In the block tree each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root which the final ordering is stored.

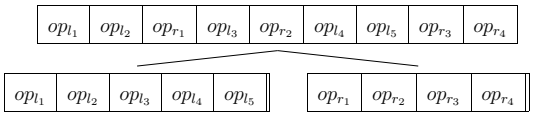
Figure 5). Each process has a leaf, and it appends its operations to its leaf. After that, the process tries to propagate up to the tree's root. An ordering is stored in each node of the tree of operations propagated to the node. All processes agree on the sequence stored in the root (Linearization ordering).

The goal here is to ensure in each propagate step, the new operations are propagated up to the parent in $\log p$ steps (See Figure 6a). And then use the linearization ordering to answer the dequeue operations.

In each propagate step, our algorithm uses a subroutine Refresh(n) that aggregates new operations from node n's children (that have not already been propagated to n) and tries to append them into n using `CAS` operations. The general idea is that if we call Refresh(n) twice, the operations in the first Refresh(n) are guaranteed to be in n. Since there is a successful instance of Refresh(n) after the first Refresh(n), which has appended the talked operations. Instead of storing operations explicitly in the nodes, we only keep track of the number of them. This allows us to `CAS` fixed-size objects in each Refresh(n). To do that, we introduce blocks that only contain the number of operations from left and right child in a Refresh() procedure and only propagate the block of the new operations.



(a) Operations after || are new.



(b) New operations are added to the parent node.

Figure 6: Propagate Step, operations in children after —— are new.

We also implement methods Get(i), Index(op) to get the ith propagated operation and compute the order of a propagated operation in the linearization. Get(i) finds the block containing the ith in the root and then finds its sub-block recursively to reach a leaf. Index() is somewhat similar but more complicated from bottom to top; the main challenge is that each level of the tree should take poly-logarithmic steps with respect to p. Having methods Get(i), Index(op), we can have the linearization shared among the processes. Now we can implement Enqueue and Dequeue using our block tree. An `Enqueue(e)` can be done using just an `Append()` operation with input argument `e` in the block tree. `Dequeue()` consists of an `Append()`, `Index()`, compute the corresponding response, and return it using `Get()`.

`Get()` and `Index()` search among blocks in each level of the tree to find the containing sub-block or super-block of the given operation. We use prefix sum for binary search and some extra constant information that we do not discuss here.

However, the algorithm works as a queue, but the Get(i) may take a long time since it has to find the block containing the ith operation at the root level. We come with a garbage collection only for the root level to reach $\log Q$. The main idea is to make use of the fast split of persistent red-black trees. So our algorithm works in $O(\log^2 p + Q)$ steps.

4 Next steps

The current algorithm only works as a queue, but other data structures like Stacks share similarities. It might be possible to come up with a generalization for all these types. We do not care about the space, and we use infinite-sized arrays that are not practical. Our future work is to manage the memory of the tree and make it practical.

5 Planning

The plan is to finish the writing of the thesis by Aug 2021. The thesis will contain:

- More detailed discussion of previous work
- Description of the algorithm, layer by layer
- Pseudocode of the algorithm in two parts; regarding memory management
- Proof of linearizability and time analysis

We expect to submit a paper of our work to *36th Int-— Symposium on Distributed Computing October 25-27, 2022, Augusta, Georgia, USA* by May 13.

References

[1] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996* (J. E. Burns and Y. Moses, eds.), pp. 267–275, ACM, 1996.

[2] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free FIFO queues," in *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA* (P. B. Gibbons and P. G. Spirakis, eds.), pp. 253–262, ACM, 2005.

[3] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings* (E. Tovar, P. Tsigas, and H. Fouchal, eds.), vol. 4878 of *Lecture Notes in Computer Science*, pp. 401–414, Springer, 2007.

[4] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free FIFO queues," *Distributed Comput.*, vol. 20, no. 5, pp. 323–341, 2008.

[5] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010* (F. M. auf der Heide and C. A. Phillips, eds.), pp. 355–364, ACM, 2010.

[6] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings* (C. Lu, T. Masuzawa, and M. Mosbah, eds.), vol. 6490 of *Lecture Notes in Computer Science*, pp. 302–317, Springer, 2010.

[7] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM SIGPLAN Symposium on Prin-*

ciples and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011 (C. Cascaval and P. Yew, eds.), pp. 223–234, ACM, 2011.

[8] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank, "BQ: A lock-free queue with batching," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018* (C. Scheideler and J. T. Fineman, eds.), pp. 99–109, ACM, 2018.

[9] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks," in *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings* (R. Ramanujam and S. Sen, eds.), vol. 3821 of *Lecture Notes in Computer Science*, pp. 408–419, Springer, 2005.

[10] P. Jayanti, "A time complexity lower bound for randomized implementations of some shared objects," in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998* (B. A. Coan and Y. Afek, eds.), pp. 201–210, ACM, 1998.

[11] T. D. Chandra, P. Jayanti, and K. Tan, "A polylog time wait-free construction for closed objects," in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998* (B. A. Coan and Y. Afek, eds.), pp. 287–296, ACM, 1998.