Wait-free Queues with Polylogarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

August 8, 2022

1 Introduction

In this work, we are going to introduce a novel lock-free queue implementation. Linearizability and lock-freedom are standard requirements for designing shared data structures. All existing linearizable, lock-free queues in the literature have a common problem in their worst case. Our algorithm solves this problem, performing better than them.

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 designed for one process. For example, consider two processes trying to insert some values into a sequential linked list simultaneously. Processes p, q read the same tail node, p changes the next pointer of the tail node to its new node and after that q does the same. In this run, p's update is overwritten. One solution is to use locks; whenever a process wants to do an update or query on a data structure, the process 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 the one process holding the lock can make progress.

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 property* requires 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 an execution on a linearizable queue that is initially empty. The arrow shows time, and each rectangle shows the time between the invocation and the termination of an operation. Since `Enqueue(A)` and `Enqueue(B)` are concurrent, `Enqueue(B)` may or may not take effect before `Enqueue(A)`. The execution in Figure 2 is not linearizable since `A` has been enqueued before `B`, so it has to be dequeued first.

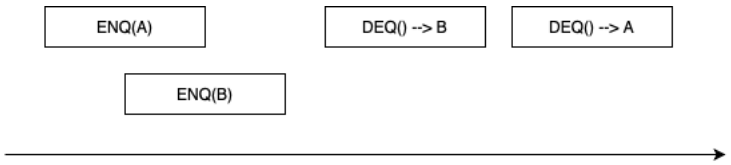


Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

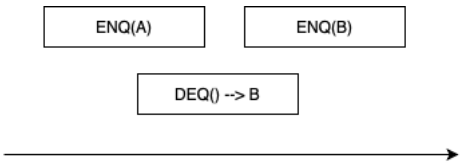


Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

There are various progress properties; the strongest is wait-freedom, and the more common is lock-freedom. An algorithm is *wait-free* if each operation terminates after a finite number of its own 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 but not vice versa; in an infinite run of a lock-free algorithm there might be an operation that takes infinitely many steps but never terminates.

2 Related Work

2.1 List-based Queues

In the following paragraphs, we look at previous lock-free queues. Michael and Scott [1] introduced a lock-free queue which we refer to as the MS-queue. A version of it is included in the standard Java Concurrency Package. Their idea is to store the queue elements in a singly-linked list (see Figure 3). Head points to the first node in the linked list that has not been dequeued, and Tail points to the last element in the queue. To insert a node into 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 amortized number of steps to be $\Omega(p)$ per enqueue. Similarly, dequeue can take $\Omega(p)$ steps.

Moir, Nussbaum, and Shalev [2] presented a more sophisticated queue by using the elimination technique. The elimination mechanism has 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 pairs of concurrent operations such as an enqueue and a dequeue to exchange values when the queue is empty or when concurrent operations can be linearized to empty the queue. Their algorithm makes it possible for long-running operations to eliminate an opposing operation. The empirical evaluation showed the throughput of their work is better than the 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.

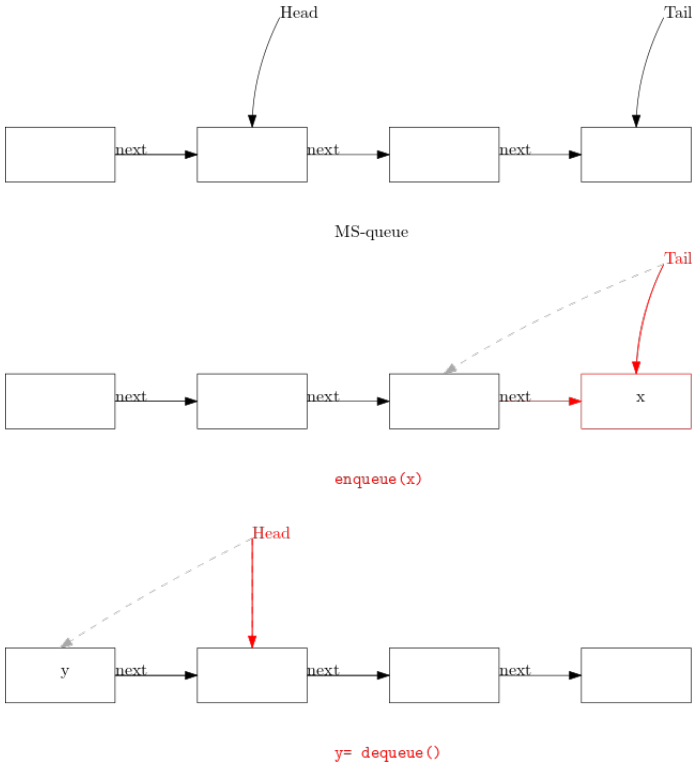


Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

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. Enqueues 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 step complexity remains $\Omega(p)$ per operation.

Ladan-Mozes and Shavit [4] presented an Optimistic Approach to Lock-Free FIFO Queues. They use a doubly-linked list and do fewer `CAS` operations than

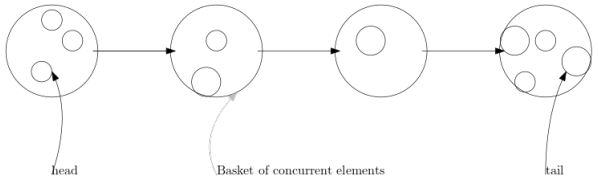


Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do `CAS`. To order the operations in a basket, the mechanism in the algorithm for processes is to `CAS` again. The successful process will be the next one in the basket and so on.

MS-queue. But as before, the worst case is when there are p concurrent enqueues which have to be enqueued one by one. The amortized worst-case 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. Their main idea is that with knowledge of all the history of operations, it might be possible to answer queries faster than doing them one by one. In our work we also maintain the whole history. They present experiments that show their algorithm performs well in some situations.

Gidenstam, Sundell, and Tsigas [6] introduced a new algorithm using a linked list of arrays. Global head and tail pointers point to arrays containing the first and last elements in the queue. Global pointers are up to date, but head and tail pointers may be behind in time. An enqueue or a dequeue searches in the head array or tail array to find the first unmarked element or last written element (see Figure 5). Their data structure is lock-free. Still, if the head array is empty and p processes try to enqueue simultaneously, the step complexity remains $\Omega(p)$.

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

In the worst-case step complexity of all the list-based queues discussed above,

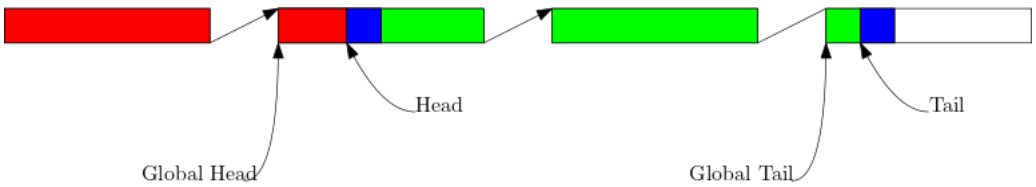


Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

there is a p term that comes from the case all p processes try to do an enqueue simultaneously. Morrison and Afek call this the *CAS retry problem* [8]. It is not limited to list-based queues and array-based queues share the CAS retry problem as well [9, 10, 11] . We are focusing on seeing if we can implement a queue in sublinear steps in terms of p or not.

2.2 Universal Constructions

Herlihy discussed the possibility of implementing shared objects from other objects [12]. A *universal construction* is an algorithm that can implement a shared version of any given sequential object. We can implement a concurrent queue using a universal construction. Jayanti proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of p-process universal constructions [13]. He also introduced a construction that achieves $\mathrm{O}(\log^2 p)$ shared accesses [14]. His universal construction can be used to create any data structure, but its implementation is not practical because of using unreasonably large-sized `CAS` operations.

Ellen and Woelfel introduced an implementation of a Fetch&Inc object with step complexity of $O(\log p)$ using $O(\log n)$-bit `LL/SC` objects, where n is the number of operations [15]. Their idea has similarities to Jayanti's construction, and they represent the value of the Fetch&Inc using the history of successful operations.

3 Our Contribution

In this work, we design a queue with $O(\log^2 p + \log q)$ steps per operation, where q is the size of the queue, avoiding the $\Omega(p)$ worst-case step complexity of existing shared queues based on linked lists or arrays. We introduce a data structure that allows processes to agree on the linearization ordering of their operations using $\mathrm{O}(\log p)$ `CAS` per operation called a *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 6). Each process has a leaf, and it appends its operations to its leaf. After that, the process tries to propagate its new operation up to the tree's root. An ordering of operations propagated up to a node is stored in that node. All processes agree on the sequence stored in the root and this is used as the linearization ordering. Our idea is similar to Jayanti and Petrovic's multi-enqueuer single-dequeuer Queue [16], but we do not use `CAS` operations with big words and do not put a limit on the number of concurrent dequeues.

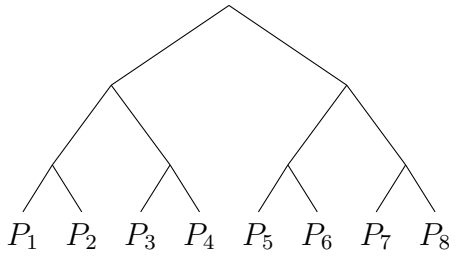
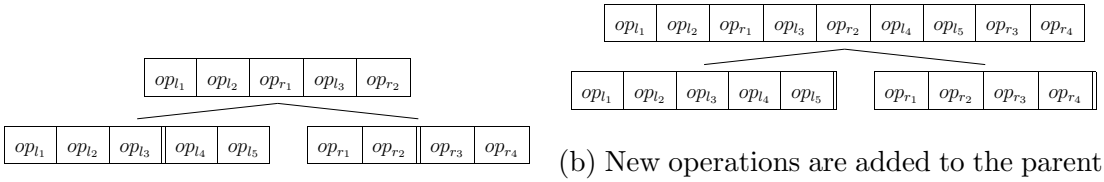


Figure 6: In the block tree each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root, which stores the final ordering of all operations.

The goal here is to ensure that in each propagate step the new operations are propagated up to the parent in $O(\log p)$ steps (see Figure 7). Then, a dequeue operation uses the linearization ordering to compute its answer.

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 a `CAS` operation. The general idea is that if we call REFRESH(n) twice, the operations in n's children before the first REFRESH(n) are guaranteed to be in n. 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 the left and the 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 7: Successful `Refresh`, operations in children after || are new.

We also implement methods `Get(i)`, `Index(op)` to get the ith propagated operation and compute the rank of a propagated operation in the linearization. `Get(i)` finds the block containing the ith operation in the root and then finds its sub-block recursively to reach a leaf. `Index()` is similar but more complicated, finding super-blocks from a leaf to the root. The main challenge in each level of `Get(i)` and `Index(op)` is that it should take polylogarithmic steps with respect to p. After appending operations to the root, processes can find out information about the linearization ordering using `Get(i)` and `Index(op)`. Now we can implement Enqueue and Dequeue using our block tree. An `Enqueue(e)` appends an operation with input argument `e` in the block tree. To do a `Dequeue()`, process p first appends a `DEQ` operation to the tree. Then p finds the rank of the `DEQ` using

`Index()`, the rank of the `DEQ` and the information stored in the root about the queue p computes the rank of the `ENQ` having the answer of the `DEQ`. Finally p returns the argument of that `ENQ` using `Get(i)`.

`Get()` and `Index()` search among blocks in each level of the tree to find the sub-block or super-block containing the given operation. Each block stores a constant amount of information (like prefix sums) to allow binary searches to find the required block in a node quickly.

The algorithm works as a queue, but `Get(i)` may take a long time since it has to find the block containing the ith operation at the root level. We came up with a garbage collection only for the root level to reduce the time to find this block to $O(\log q)$. The main idea is to remove a block after its operations are no longer needed. We will store the blocks in the root in a persistent red-black tree to be able only to maintain the remaining information of the queue. It is possible to search in persistent red-black trees with size n in $O(\log n)$ steps and split the tree by a key in $O(\log n)$ steps. Finally, the algorithm works in $O(\log^2 p + \log q)$ 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 have not yet considered space usage, 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 August 2021. The thesis will contain:

- More detailed discussion of previous work
- Description of the algorithm, layer by layer
- Pseudocode of the algorithm
- Proof of linearizability and time analysis

We expect to submit a paper of our work to *36th International 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 Principles 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] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013* (A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, eds.), pp. 103–112, ACM, 2013.

[9] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001* (A. L. Rosenberg, ed.), pp. 134–143, ACM, 2001.

[10] N. Shafiei, "Non-blocking array-based algorithms for stacks and queues," in *Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings* (V. K. Garg, R. Wattenhofer, and K. Kothapalli, eds.), vol. 5408 of *Lecture Notes in Computer Science*, pp. 55–66, Springer, 2009.

[11] R. Colvin and L. Groves, "Formal verification of an array-based nonblocking queue," in *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*, pp. 507–516, IEEE Computer Society, 2005.

[12] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, p. 124149, jan 1991.

[13] 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.

[14] 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.

[15] F. Ellen and P. Woelfel, "An optimal implementation of fetch-and-increment," in *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, (Berlin, Heidelberg), p. 284298, Springer-Verlag, 2013.

[16] 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.