Thesis Proposal

Hossein Naderi

supervised by Eric Ruppert

March 9, 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 sequential linked list simultaneously, processes p, q read the same tail node, p changes the pointer of tail 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 one process holding the lock can make progress. 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 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 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)`. The execution in Figure 2 is not consistent since `A` has been enqueued before `B`, so it has to be dequeued first.
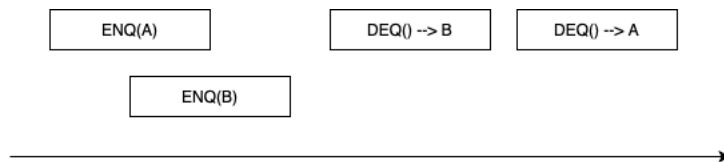

Figure 1: An example of a linearizable execution. Either could take effect first since `Enqueue(A)`, and `Enqueue(B)` are concurrent.
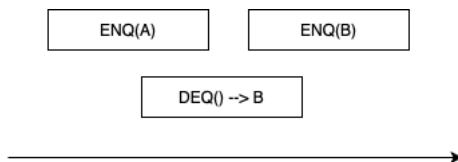

Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A`.

There are various progress properties; the strongest is wait-freedom, and the more common is loc-freedom. 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 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. 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 worst-case number of steps to be $\Omega(p)$ per enqueue. Similarly, dequeue can take $\Omega(p)$ steps.
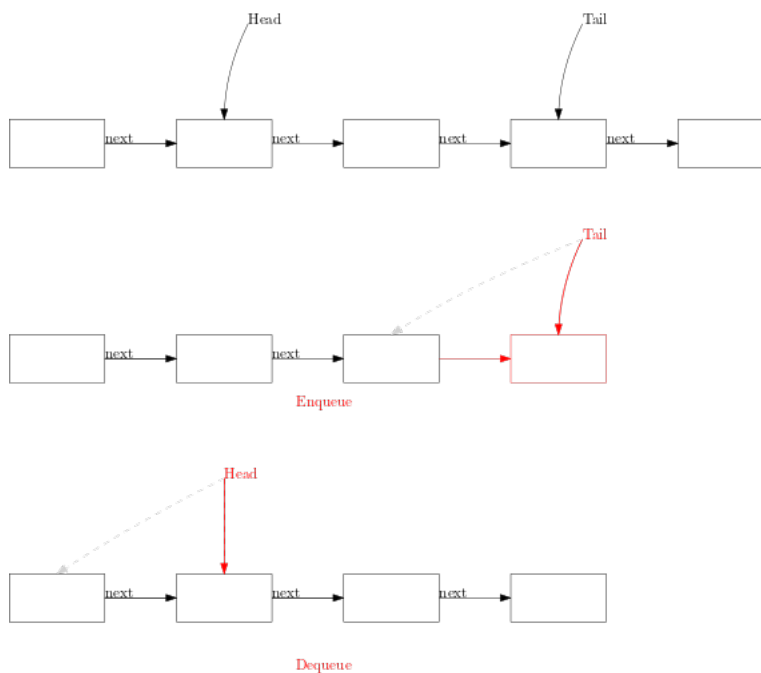
Figure 3: MS-queue structure, enqueue and dequeue operations.

Moir, Nussbaum, and Shalev [2] presented a more sophisticated queue by using

the elimination idea. The elimination mechanism has the dual purpose of allowing operations to be 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 failed aged operations to complement. The empirical evaluation showed the throughput 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. 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 complexity remains $\Omega(p)$ per operation.
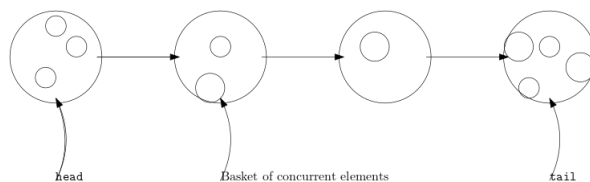

Figure 4: Baskets queue idea.

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

5

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 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 element in the queue and the elements when they are written by `CAS` operations and dequeued. Their data structure is lock-free. An enqueue or a dequeue searches for the array head or tail array to find the first unmarked element or last written element (see Figure 5). Still, if the head array is empty and p processes try to enqueue simultaneously, step complexity remains $\Omega(p)$.
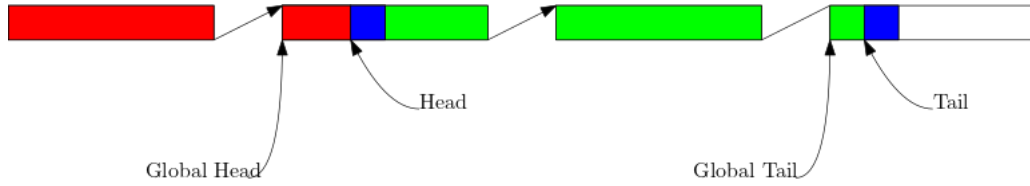

Figure 5: Global pointer point to arrays, dequeued elements are red and current elements are in green.

Kogan and Petrank [7] introduced wait-free queues based on 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, there is a p term that comes from the case all p processes try to do an enqueue simultaneously. Morrison and Afek introduce this execution as *CAS retry problem* [8]. 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 introduced the consensus hierarchy and discussed the possibility of implementation of shared objects from other objects [9]. A universal construction is an object which if it implements any other objects. 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 [10]. He also introduced a construction that achieves $O(\log^2 p)$ shared accesses [11]. His universal construction can be used to create any data structure, but his implementation is not practical because of using unreasonably sized `CAS` operations.

Ellen and Woelfel introduced an implementation of a Fetch&Inc object with step complexity of $\log p$ using $O(\log n)$bit `LL/SC` objects, which n is the number of operations [12]. 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 $\log^2 p + \log q$ steps per operation, which q is the size of the queue, outperforming queues built by the LinkedList approach. Our idea is similar to Jayanti's Multi Enqueuer-Single Dequeuer Queue [13], but we do not use `CAS` operations with big words and do not put a limit on the number of concurrent dequeues. The new feature of our implementation is that even if there are p processes trying to enqueue or dequeue simultaneously, each operation takes $O(\log^2 p + \log q)$ steps. We introduce a data structure that allows processes to agree on the linearization ordering of their operations using $O(\log p)$ `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
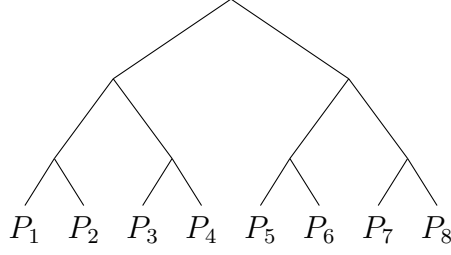
Figure 6: In the block tree each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root, which stores the final ordering.

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.

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 7a). Then a dequeue operation uses the linearization ordering to compute its answer.

In each propagate step, our algorithm uses a subroutine $\text{REFRESH}(n)$ 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 $\text{REFRESH}(n)$ twice, the operations in n's children before the first $\text{REFRESH}(n)$ 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 $\text{REFRESH}(n)$. To do that, we introduce blocks that only contain the number of operations from left and right child in a $\text{REFRESH}()$ procedure and only propagate the block of the new operations.

We also implement methods `Get(i)`, `Index(op)` to get the ith propagated

8

(a) Operations after || are new.
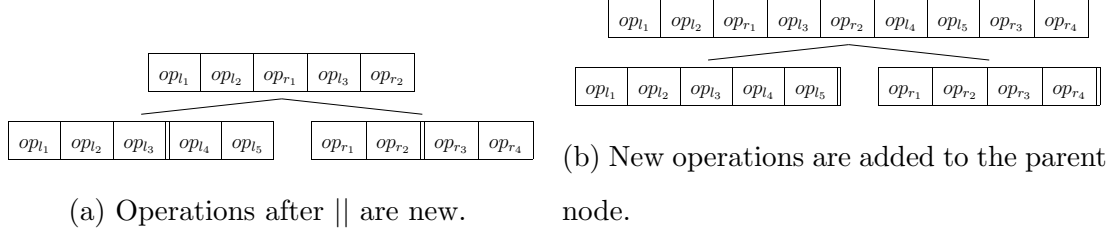
(b) New operations are added to the parent node.

Figure 7: Successful `Refresh`, operations in children after || are new.

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 superblocks from a leaf to the root. The main challenge in each level of the `Get(i)`, `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)`, `Index(op)`. Now we can implement Enqueue and Dequeue using our block tree. An `Enqueue(e)` is appending an operation with input argument `e` in the block tree. To do a `Dequeue()`, first process p first appends a `DEQ` operation to the tree. Then finds the rank of the `DEQ` using `Index()`, after knowing the rank of `DEQ` it computes the rank of the `ENQ` having the answer of the `DEQ`. Finally returns the argument of the answer `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. We use prefix sum for binary search and a constant amount of information that we do not discuss here.

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 finished. We

can 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. 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. So our 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 in two parts; regarding memory management

- 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 *Prin-*

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

[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.

[12] 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.

[13] 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.