1 Related Work

1.1 List-based Queues

In the following paragraphs, we look at previous lock-free queues. Michael and Scott [?] DBLP:conf/podc/MichaelS96 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 1). fig::msq 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.

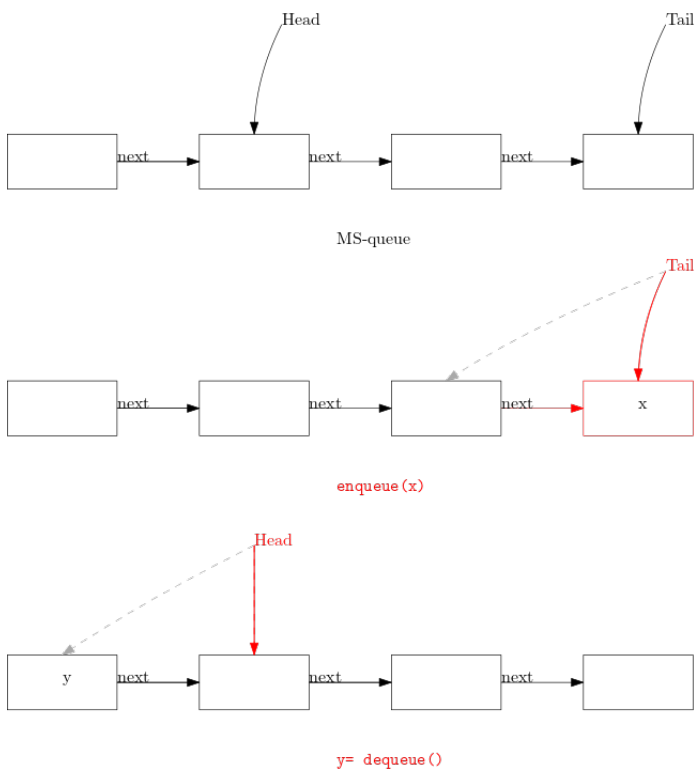

fig::msq Figure 1: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

Moir, Nussbaum, and Shalev [?] DBLP:conf/spaa/MoirNSS05 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.

Hoffman, Shalev, and Shavit [?] DBLP:conf/opodis/HoffmanSS07 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.

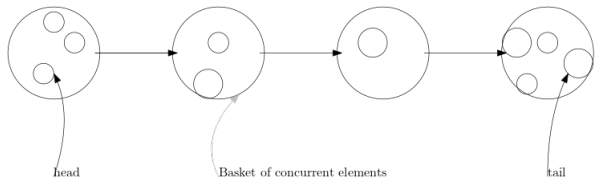

Figure 2: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do `CAS`. To order the operations in a basket, the mechanism in the algorithm for processes is to `CAS` again. The successful process will be the next one in the basket and so on.

Ladan-Mozes and Shavit [?] DBLP:journals/dc/Ladan-MozesS08 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 still $\Omega(p)$ `CAS`es.

Hendler et al. [?] 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 [?] 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 3). Their data structure is lock-free. Still, if the head array is empty and p processes try to enqueue simultaneously, the step complexity remains $\Omega(p)$.

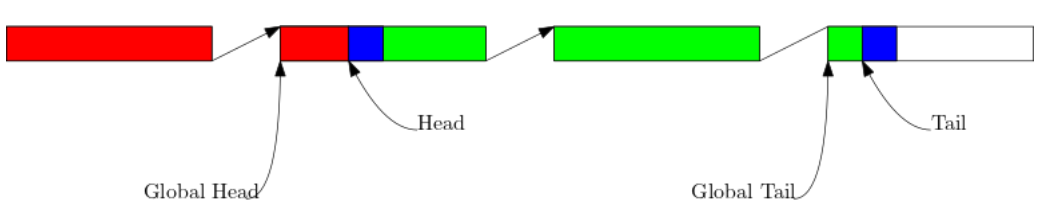

fig::sundell Figure 3: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

Kogan and Petrank [?] 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, 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* [?]. It is not limited to list-based queues and array-based queues share the CAS retry problem as well [?, ?, ?] . We are focusing on seeing if we can implement a queue in sublinear steps in terms of p or not.

1.2 Universal Constructions

Herlihy discussed the possibility of implementing shared objects from other objects [?]. 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 [?]. He also introduced a construction that achieves $O(\log^2 p)$ shared accesses [?]. 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 [?]. Their idea has similarities to Jayanti's construction, and they represent the value of the Fetch&Inc using the history of successful operations.

2 Adiitional ones

2.1 Tsigas and Zhang

Tsigas and Zhang [?] present a cyclic array queue that allows the head and tail to lag at most m nodes behind the actual head and tail of the queue, so the amortized number of CAS executions per operation is 1 + 1/m. Their algorithm is limited to bounded queues due to their static allocation.

2.2 Kogan and Herlihy

Previous works present concurrent constructs that combine multiple operations into a single operation on the shared object. We chose to combine operations and apply them as batches, to increase scalability. The work of Kogan and Herlihy [?] is the closest to this work. They propose alternative definitions for linearizability of executions with batches, including MF-linearizability,

which we use. They describe very simple implementations of stacks, queues, and linked lists that demonstrate the benefits of using futures.

2.3 Yang and Mellor-Crummey

Because of the problem of failure of CASes one way is to use other primitive objects other than CAS objects. Yang and Mellor-Crummey [?] DBLP:conf/ppopp/YangM16 utilize fetch-and-add as well, to form a wait-free queue. They present the first linearizable and wait-free FIFO queue based on fetch-and-add (FAA). While compare-and-swap (CAS) based non-blocking algorithms may perform poorly due to work wasted by CAS failures, algorithms that coordinate using FAA, which is guaranteed to succeed, can in principle perform better under high contention.

I could not compute its complexity but I guess it is hard or not important