Distributed Queues in Shared Memory

Multicore Performance and Scalability through Quantitative Relaxation

Andreas Haas
University of Salzburg
ahaas@cs.uni-salzburg.at

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Michael Lippautz
University of Salzburg
mlippautz@cs.uni-salzburg.at

Hannes Payer
Google
hpayer@google.com

Ali Sezgin
IST Austria
asezgin@ist.ac.at

Ana Sokolova
University of Salzburg
anas@cs.uni-salzburg.at

ABSTRACT

A prominent remedy to multicore scalability issues in concurrent data structure implementations is to relax the sequential specification of the data structure. We present distributed queues (DQ), a new family of relaxed concurrent queue implementations. DQs implement relaxed queues with linearizable emptiness check and either configurable or bounded out-of-order behavior or pool behavior. Our experiments show that DQs outperform and outscale in micro- and macrobenchmarks all strict and relaxed queue as well as pool implementations that we considered.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Algorithms, Measurement, Performance

Keywords

Inexact computing, load balancing, LRU

1. INTRODUCTION

A concurrent data structure implementation is desired to be correct, fast, and scalable. Correctness, specified as a consistency condition such as linearizability [10], constrains the set of allowed behaviors. A mapping from a sequence of concurrent method calls and returns to a set of sequences in which methods are assumed to execute sequentially defines allowed behaviors. Scalability, on the other hand, is the ability to increase throughput as the number of concurrent threads increases. An ideally scalable implementation would increase the throughput n-fold on a data structure as the number of cores and threads using the data structure increases n times.

Recent results [4] dictate that correctness and scalability conflict, especially so for ordered data structures such as queues or stacks. This is due to the impossibility of having a truly distributed update mechanism for the logical state of a data structure. For instance, all threads trying to enqueue an element into a shared queue have to, in some form or another, acquire the ownership of the tail of the queue and update it accordingly. As the number of concurrent threads trying to enqueue increases, all threads progressing in their own tasks becomes less likely, resulting in poor scalability.

In order to obtain scalability, recent research has focused on expanding the set of allowed behaviors, e.g. by relaxing the sequential specification of a data structure, but mostly implicitly and qualitatively, such as replacing a queue with a pool. With the work on quantitative relaxation [9], one can now define and implement new data structures which are k-relaxed, for any $k \geq 0$, relative to a given data structure. For instance, a 1-relaxed queue behaves like a queue with the exception that a dequeue is allowed to return either the oldest or the second oldest element of the queue. Quantitative relaxation of queues and other data structures is an example of a recent trend towards inexact computing [13].

Our new, relaxed queue implementations are instances of a distributed queue (DQ) which consists of multiple FIFO queues, called partial queues, whose operations are derived from the Michael-Scott queue [18]. There are two DQ subfamilies depending on the way of controlling the access to the partial queues: (1) DQs that use load balancers to decide, upon an enqueue or a dequeue call, in which partial queue the element should be enqueued or dequeued, and (2) least-recently-used DQs where ABA counters are used to make that decision. The least-recently-used DQs implement k-relaxed queues for configurable k. For the load-balanced DQs k cannot be configured but in some cases still be bounded in the number of threads. The DQs for which k cannot be bounded effectively implement pool semantics. However, despite being relaxed, all DQs provide a linearizable emptiness check, i.e., return empty if and only if all partial queues are empty.

In a number of experiments, we analyze a broad range of DQs,

CF'13, May 14–16, 2013, Ischia, Italy.

both via different load balancers and ABA counters, and compare them with a large variety of existing queue and pool implementations. In our micro- and macrobenchmarks DQs provide better performance and scalability than any of the queue or pool implementations we considered. We also measure the degree of relaxation that actually happens when running different implementations on a given workload. Interestingly, the actual behavior of some implementations stays well below their relaxation bounds.

The structure of the paper is as follows. In Section 2 we informally recall the definitions of pool and k-relaxed queue semantics. In Section 3 we present DQs and their algorithms and design details, and demonstrate correctness. In Section 4 we discuss related work and introduce existing pools and queues that are then evaluated against the DQs in Section 5. Section 6 concludes the paper.

2. POOLS AND K-RELAXED QUEUES

We informally recall the sequential specifications of a pool and a k-relaxed queue to facilitate the correctness proofs in Section 3.3 where DQs are shown to be linearizable with respect to pool semantics and, in the case of some load-balanced DQs and the least-recently-used DQs, k-relaxed queue semantics.

The sequential specification of a pool is informally defined as follows. A pool has a put and a get operation which we call enqueue and dequeue here for simplicity. The enqueue operation inserts an element into the pool. The dequeue operation removes and returns previously inserted elements in no particular order and returns `null` when the pool is empty.

A k-relaxed queue is a restricted out-of-order k-relaxation of a FIFO queue as defined in the framework for quantitative relaxation of concurrent data structures [9]. We informally discuss the sequential specification of a k-relaxed queue.

Let Σ bet the set of queue methods including their input and output values defined as

$$\Sigma = \{\texttt{enq}(x), \texttt{deq}(x) \mid x \in D\} \cup \{\texttt{deq}(\texttt{null})\}$$

where D is the set of elements that can be enqueued in and dequeued from the queue and $\texttt{deq}(\texttt{null})$ represents a dequeue returning empty. We refer to sequences in Σ^* as queue sequences.

The sequential specification of a FIFO queue is the set $S \subseteq \Sigma^*$ that contains all valid FIFO queue sequences. Informally, valid FIFO queue sequences are sequences where elements are dequeued in the same order as they are enqueued. Furthermore, a $\texttt{deq}(\texttt{null})$ only happens if the queue is empty at the time of $\texttt{deq}(\texttt{null})$, i.e., every element which gets enqueued before $\texttt{deq}(\texttt{null})$ also gets dequeued before $\texttt{deq}(\texttt{null})$. For example, the queue sequence

$$\texttt{enq(a)enq(b)deq(a)enq(c)deq(b)deq(c)}$$

belongs to S whereas

$$\texttt{enq(a)enq(b)deq(b)enq(c)deq(a)deq(c)}$$

does not.

A restricted out-of-order k-relaxation of a FIFO queue is the set $S_k \subseteq \Sigma^*$ containing all sequences with a distance of at most k to the sequential specification S of the queue. Informally, the distance is the number of elements overtaking each other in the queue, i.e., an element e may overtake at most k elements and may be overtaken by at most k other elements before it is dequeued. A 0-relaxed FIFO queue thus corresponds to a regular FIFO queue. The above example sequences are therefore within the specifications of a regular FIFO queue and a 1-relaxed FIFO queue, respectively. We call k the relaxation bound of a k-relaxed queue.

3. DISTRIBUTED QUEUES

We introduce two distributed queue (DQ) algorithms called load-balanced DQ and least-recently-used (LRU) DQ. Both algorithms implement a shared array of $p \geq 1$ so-called partial queues whose implementation is based on Michael-Scott (MS) FIFO queues [18]. Upon an enqueue or dequeue operation one out of the p partial queues is selected for performing the actual operation without any further coordination with the other $p-1$ partial queues. Selection is done either by a load balancer, hence load-balanced DQ, or by an LRU-style algorithm that uses the ABA counters in the head and tail pointers of the partial queues (for dequeue and enqueue, respectively) to identify the less recently used partial queues. For dequeueing, both algorithms implement an emptiness check that checks all partial queues up to two times before returning empty. If all queues are found to be empty during the first pass the queues are checked again. If no queues are found that have performed an enqueue operation since the beginning of the first pass, recognized by the ABA counters in their tail pointers, the algorithms correctly return empty.

The concept of distributing access to different queues is not new. Scal Queues [15, 14] already use (special cases of) the load balancers we discuss here but do not implement an emptiness check, neither do Multilane Multisets [8]. As a consequence, applications with termination conditions based on emptiness may be impossible to implement using Scal Queues or Multilane Multisets.

Similar to other algorithms such as MS, we use a double-word `compare-and-swap` (`CAS`) [1] instruction to change a pointer and its ABA counter atomically. Whenever we speak of a pointer below we therefore mean the pointer together with its ABA counter. Note that using double-word `CAS` does not limit the generality of DQ since ABA counters can alternatively be embedded into the pointer itself (although with less precision). However, embedding ABA counters requires knowledge of the memory allocation boundaries since the ABA counter has to reside in a part of the pointer that is not needed for addressing.

3.1 Load-balanced DQ Algorithm

Listing 1 shows the pseudo code of the load-balanced DQ algorithm. The `enqueue` method (line 1) calls the load balancer using the method `load_balancer` (line 2) which determines the partial queue at `index` in the shared `DQ` array of partial queues for the actual `enqueue` operation (line 3).

Similarly, the `dequeue` method (line 5) also calls the load balancer using `load_balancer` (line 6) to obtain an initial index `start`. The method then searches `DQ` for a non-empty partial queue beginning at `start` and wrapping around at $p-1$. Using the MS algorithm the `MS_dequeue` method retrieves the oldest element from the partial queue at `index` if the queue is not empty (line 10). In this case, `dequeue` returns the `element` (line 14). The `MS_dequeue` method is slightly different from the standard MS dequeue operation since it also returns the `current_tail` pointer of the partial queue, which is saved in a thread-local array of size p called `tail_old` when the partial queue is empty (line 12). If all partial queues have been found to be empty, the second pass of the emptiness check begins (line 16–21). If a partial queue is found to have a tail pointer that differs from the tail pointer saved in `tail_old` (line 17) at least one new element has been enqueued into that queue. In this case, `dequeue` retries to find a non-empty partial

[1] `compare-and-swap` is a CPU instruction that atomically swaps the value stored in some memory location if that location contains an expected value. The operation also returns whether it was successful or not.

Listing 1: Lock-free load-balanced distributed queue algorithm

```
1  enqueue(element):
2    index = load_balancer();
3    DQ[index].MS_enqueue(element);
4
5  dequeue():
6    start = load_balancer();
7    while true:
8      for i in 0 to p-1:
9        index = (start + i) % p;
10       element, current_tail = DQ[index].MS_dequeue();
11       if element == null:
12         tail_old[index] = current_tail;
13       else:
14         return element;
15
16     for i in 0 to p-1:
17       if DQ[i].get_tail() != tail_old[i]:
18         start = i;
19         break;
20       if i == p-1:
21         return null;
```

queue starting with that queue (line 18–19). Otherwise, `dequeue` returns `null` (line 21).

Load Balancers

We use two load balancers in our experiments called d-RA and b-RR. The d-RA load balancer randomly selects $d \geq 1$ queues out of the p partial queues and then returns (the index of) the queue with the least elements among the d queues when called by an enqueue operation. Symmetrically, the load balancer returns the queue with the most elements when called by a dequeue operation. However, for better performance, the number of elements in partial queues is only approximated by computing the differences between ABA counters of the queues' head and tail pointers non-atomically. Random numbers are generated thread-locally using thread-local seeds. The d-RA load balancer has already been described elsewhere [15].

The b-RR load balancer maintains $b \geq 1$ pairs of shared round-robin counters that are associated with threads such that each thread is permanently assigned to exactly one pair and all pairs have approximately the same number of threads assigned. If b is equal to (or greater than) the number of threads, the pairs are thus thread-local. In this case, we call the load balancer TL-RR. The counters in a pair keep track of which partial queue was selected for the most recent enqueue and dequeue operation, respectively, called by any of the threads assigned to the pair. Upon an enqueue or dequeue operation the respective counter is atomically incremented and returned by a `fetch-and-add` [2] instruction identifying in round-robin fashion (the index of) the partial queue to be used for the operation.

Note that the d-RA DQs do not provide relaxation bounds but the b-RR DQs do as shown in Section 3.3.

3.2 Least-recently-used DQ Algorithm

Listing 2 shows the pseudo code of the least-recently-used DQ algorithm. The key invariant maintained by the algorithm is that the maximum difference of the ABA counters of the head pointers of the p partial queues is at most one. The same holds for the ABA counters of the tail pointers. Moreover, the algorithm always enqueues into a partial queue whose tail pointer has an ABA counter with the lowest value among the ABA counters of the tail pointers of all partial queues. Similarly, the algorithm always dequeues from a partial queue whose head pointer has an ABA counter with the lowest value. Note that the ABA counters of the head/tail pointers may only differ by one but we anyway speak of the lowest value here.

The `enqueue` method (line 1) starts by calling a `random` number generator that returns the index `start` of one of the p partial queues (line 2). Then, by calling the `lowest_aba_tail` method, the partial queues are checked, beginning at `start` and wrapping around at $p-1$, for a queue whose tail pointer has an ABA counter with the lowest value among the ABA counters in the tail pointers of all partial queues. The `lowest_aba_tail` method returns the index of such a queue and the ABA counter of its tail pointer which are then stored in `aba_index` and `aba_count`, respectively (line 4). Note that as soon as two different ABA counters are found the search may stop since the ABA counter with the lower value is guaranteed to contain the lowest value because of the algorithm's invariant. After finding `aba_count` the algorithm again iterates over all partial queues starting at `aba_index` (lines 5–6). Since there may be multiple queues whose tail pointers have ABA counters with the same value as `aba_count`, all such queues are candidates for enqueueing the `element` (lines 7-8). Using the MS algorithm the `try_MS_enqueue` method enqueues the `element` into the queue at `index` if the tail pointer of the queue is still equal to `current_tail` (line 9). In this case, the `enqueue` method returns (line 10). If in the meantime some other thread successfully enqueued an element, however, and thus changed the tail pointer of the queue, `try_MS_enqueue` fails causing the algorithm to look for another partial queue for enqueueing whose tail pointer is equal to `aba_count`. If there are none, the algorithm retries (back to line 3).

The `dequeue` method (line 12) begins like the `enqueue` method except that it looks for a partial queue whose head pointer has the lowest ABA counter among the head pointers of all partial queues.

[2] `fetch-and-add` is a CPU instruction that atomically increments the value of a memory location and returns the old value.

Listing 2: Lock-free least-recently-used distributed queue algorithm

```
1 enqueue(element):
2   start = random();
3   while true:
4     aba_index, aba_count = lowest_aba_tail(start);
5     for i in 0 to p-1:
6       index = (aba_index + i) % p;
7       current_tail = DQ[index].get_tail();
8       if current_tail.aba == aba_count &&
9          DQ[index].try_MS_enqueue(element, current_tail):
10        return;
11
12 dequeue():
13   start = random();
14   while true:
15     aba_index, aba_count = lowest_aba_head(start);
16     empty_count = 0;
17     for i in 0 to p-1:
18       index = (aba_index + i) % p;
19       current_head = DQ[index].get_head();
20       if current_head.aba == aba_count:
21         element, current_tail =
22           DQ[index].try_MS_dequeue(current_head);
23         if element == null:
24           tail_old[index] = current_tail;
25           empty_count++;
26         else if element != FAILED:
27           return element;
28
29     if empty_count == p:
30       for i in 0 to p-1:
31         if DQ[i].get_tail() != tail_old[i]:
32           start = i;
33           break;
34         if i == p-1:
35           return null;
```

Then the method initializes a thread-local counter `empty_count` to zero (line 16). The purpose of the counter is explained below. Next, the method works like the `enqueue` method except that it calls the `try_MS_dequeue` method, which is again using the MS algorithm, to dequeue an element from a partial queue whose head pointer has the lowest ABA counter (lines 17–22). If an `element` is successfully dequeued it is returned (line 27). Similar to the load-balanced DQ algorithm, if the `try_MS_dequeue` method returns `null` the algorithm saves the tail pointer (line 24) for checking in the second pass of the emptiness check whether new elements were enqueued in the meantime (lines 30–35). The only difference is that `empty_count` is incremented (line 25), making sure that it eventually reaches the value of p if all partial queues are found to be empty. Lastly, if the `try_MS_dequeue` method failed, which happens when in the meantime another thread dequeued an element from the same queue, the for-loop over the partial queues continues (line 17). Note that in this case `empty_count` will not reach the value of p thus preventing the second pass of the emptiness check (line 29) and have the algorithm eventually retry (back to line 14) in case no element could be dequeued. This is necessary because partial queues on which dequeueing failed may still contain elements, which the second pass would nevertheless not detect since it only checks if new elements were enqueued in the meantime.

Note that overflows of ABA counters need to be handled in the `lowest_aba_tail`, `get_tail`, `lowest_aba_head`, and `get_head` methods. Whenever some but not all ABA counters have overflowed we consider temporarily decremented and thus not-overflowed copies of all ABA counters in arithmetic comparisons.

3.3 Correctness

PROPOSITION 3.1. *The load-balanced and least-recently-used DQs are linearizable with respect to the sequential specification of a pool.*

PROOF. Without loss of generality we assume that enqueued elements are unique. We begin by identifying the linearization points of each method and then show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the specification of a pool. Note that the load balancers of a load-balanced DQ do not have an effect on linearizability as long as they identify any of the partial queues.

For the load-balanced DQ (Listing 1) the linearization point of the `enqueue` method is the linearization point of the `MS_enqueue` method of the MS queue (line 3). Similarly, the linearization point of the `dequeue` method is the linearization point of the `MS_dequeue` method of the MS queue (line 10) if it returns an element. The linearization point of a `null`-returning `dequeue` method is the last

`null`-returning `MS_dequeue` method of the MS queue (line 10) after observing all other $p-1$ partial queues to be empty and if the subsequent tail consistency checks do not find any changed tail pointers (lines 16–21).

For the LRU DQ (Listing 2) the linearization point of the `enqueue` method is the linearization point of the `try_MS_enqueue` method which is based on MS but only tries to enqueue once (line 9). The linearization point of the `dequeue` method, if it returns an element, is the linearization point of the `try_MS_dequeue` method which is also based on MS and tries to dequeue only once (line 22). Analogous to the load-balanced DQs the linearization point of a `null`-returning `dequeue` method is the `try_MS_dequeue` method that last returns `null` (line 22) before all other partial queues are still found to be empty by the tail consistency checks (lines 30–35).

1. *An element is enqueued exactly once.* This is a consequence of the assumption that elements are unique and the fact that the `enqueue` method directly passes an element to a partial queue, which is an MS queue that enqueues the element exactly once [18].

2. *An element is dequeued at most once.* Similar to the previous argument this is a consequence of the fact that an MS queue dequeues an element at most once [18].

3. *If a dequeue method returns empty, then during its execution there must exist a point in time in which there are no elements in the DQ.* Since returning `null` is without any side-effect on the DQ, it suffices to prove the existence of a point in time which corresponds to a logically empty queue. The second pass over all partial queues (lines 16–21) makes sure that after observing an empty partial queue for the first time no other thread enqueued an element. If the tail pointers, including their ABA counters, of all partial queues did not change (line 17), then the DQ was indeed empty after the first pass (lines 8–14).

Similarly, 1. through 3. also hold for the LRU DQ because its partial queues are also implemented by MS queues which only try to enqueue and dequeue once. □

PROPOSITION 3.2. *The b-RR DQs are linearizable with respect to the sequential specification of a k-relaxed queue with k bounded in the number of round-robin counters b, the number of partial queues p, and the number of threads n.*

PROOF. Given a b-RR DQ, the maximum difference of how many times any two partial queues Q_1 and Q_2 of the DQ may have been selected for an operation by a single round-robin counter is one. This is true even if Q_1 is selected for a dequeue operation that finds Q_1 to be empty and then the emptiness check selects and finds Q_2 to be non-empty instead, and thus actually dequeues from Q_2. In this case, we are safe to pretend that the operation dequeued from Q_1 since the emptiness check also searches partial queues in round-robin fashion which implies that Q_2 has been selected exactly one time less often than Q_1.

With b round-robin counters for enqueue operations and another b round-robin counters for dequeue operations the maximum difference of how many times any two partial queues may have been selected for any operation is therefore $2b$. Since threads may block after selection before performing the actual operation, the maximum imbalance among all partial queues, i.e., the maximum difference in the number of elements in any two partial queues, may be at most $2b+n$ where n is the number of threads.

Let m be $(2b+n)\times(p-1)$ elements where p is the number of partial queues.

1. *An element e may overtake at most* $3m$ *other elements.* Suppose e is enqueued in a partial queue Q that contains q elements before enqueueing e. The DQ may thus contain at most $p\times q+m$ older elements in total before e is enqueued. To dequeue e the q elements need to be dequeued from Q first which requires dequeueing a total of $p\times q-2m$ elements from the DQ leaving $3m$ elements in the DQ when e may be dequeued.

2. *An element e may be overtaken by at most* $3m$ *other elements.* Suppose again e is enqueued in a partial queue Q that contains q elements before enqueueing e. The DQ may thus contain at least $p\times q-m$ older elements in total before e is enqueued. The element e will then be dequeued after at most $p\times q+2m$ elements have been dequeued before e, provided that at least $3m$ younger elements have been enqueued after e was enqueued.

Thus $k=3m$ holds. □

PROPOSITION 3.3. *The LRU DQs are linearizable with respect to the sequential specification of a k-relaxed queue with* $k=p-1$ *where p is the number of partial queues.*

PROOF. With the LRU DQ a partial queue Q is selected after at most $p-1$ operations on other partial queues when Q becomes the least-recently-used queue. If the thread that selected Q blocks before performing the actual operation and thus not incrementing an ABA counter of Q, any other threads that have not yet selected a partial queue will also select Q. Thus $p-1$ is the maximum number of enqueue or dequeue operations that may be performed on all but one partial queue Q before an enqueue or dequeue operation, respectively, must be performed on Q.

1. *An element e may overtake at most* $p-1$ *other elements.* Suppose e is enqueued in a partial queue Q that contains q elements before enqueueing e. The DQ may thus contain at most $p\times q+(p-1)$ older elements in total before e is enqueued. To dequeue e the q elements need to be dequeued from Q first which requires dequeueing a total of $p\times q$ elements from the DQ leaving at most $p-1$ elements in the DQ when e may be dequeued.

2. *An element e may be overtaken by at most* $p-1$ *other elements.* Suppose again e is enqueued in a partial queue Q that contains q elements before enqueueing e. The DQ may thus contain $p\times q-(p-x)$ older elements in total before e is enqueued for some $0<x\leq p$. The element e will then be dequeued after at most $p\times q+x-1$ elements have been dequeued before e, provided that at least $p-1$ younger elements have been enqueued after e was enqueued.

Thus $k=p-1$ holds. □

PROPOSITION 3.4. *The load-balanced and least-recently-used DQ algorithms are lock-free provided the underlying partial queues are lock-free.* □

Lock-freedom can be shown by arguing that whenever an operation of the queue loops, another concurrent operation of the queue makes progress, an argument already used for the same purpose elsewhere [18].

4. RELATED WORK

Implementing concurrent data structures requires expensive synchronization mechanisms which may prevent multicore scalability in high contention scenarios [4]. A way to improve scalability is relaxing the sequential specification of data structures [19] quantitatively [9].

The concept of distributing concurrent access across multiple queues is not new. There exist algorithms for concurrent multilane multisets [8] and for load balancing multiple queues [15]. While both approaches share the concept of partial queues with DQs, only DQs provide a linearizable emptiness check and that without negative impact on performance and scalability. Furthermore the LRU DQ algorithm is, to the best of our knowledge, the first algorithm using multiple queues that provides a configurable bound for out-of-order behavior independent of the number of threads.

We relate our DQ algorithms to existing strict and relaxed queue as well as pool algorithms, which we have implemented and evaluated as well. We introduce these algorithms in detail to prepare for the experiments in Section 5.

The following algorithms implement strict FIFO queues: a lock-based queue (LB), the lock-free Michael-Scott queue (MS) [18], the flat-combining queue (FC) [11], and a wait-free extension (WF) of MS [16]. LB uses a single lock for each data structure operation. With MS each thread needs at least two `CAS` operations to insert an element into the queue and at least one `CAS` operation to remove an element from the queue. FC is based on the idea of helping, where a single thread performs the queue operations of multiple other threads by locking the whole queue, collecting operations, and applying them to the queue in a sequential manner. WF extends MS to a wait-free algorithm. This is done by dividing an operation into single steps that can be performed concurrently by multiple threads. The state of an operation is stored in a so-called descriptor. Each descriptor receives a unique priority and is published in a descriptor table. All threads can see all descriptors and work together by processing them one-by-one, highest priority first.

The lock-free bounded-size FIFO queue (BS) [7] is based on an array of fixed size where elements get inserted and removed circularly and enqueue operations fail when the queue is full, i.e., each array slot contains an element.

Existing relaxed queues are the unbounded-size and bounded-size k-FIFO queues (BS and US k-FIFO) [12], the segment queue (SQ) [3] and the random dequeue queue (RD) [3]. Both the US and BS k-FIFO queues implement a queue of segments of size k. In the bounded case the underlying queue is a fixed-size array. Elements get inserted in the head segment and removed from the tail segment at random positions, thus elements may overtake each other within a segment. Both algorithms provide a linearizable emptiness check (and full check in the bounded-size case) and are linearizable with respect to a $(k-1)$-relaxed queue. These algorithms perform and scale very well in general. SQ is closely related to the US k-FIFO queue, logically they both implement the same algorithm: a queue of segments. However, the implementations are significantly different. In the dequeue operation SQ eagerly tries to dequeue at all positions in the oldest segment using a `CAS` operation. SQ has better peak performance than strict queues, but does not scale better than MS [3]. Compared to k-FIFO, its performance and scalability is significantly worse. Also, SQ does not provide a linearizable emptiness check. RD is based on MS where the dequeue operation was modified in a way that, given a configurable integer r, a random number in $[0, r-1]$ determines which element is returned starting from the oldest element. Although RD does perform better than MS, it does not scale better than MS in experiments reported on elsewhere [3].

As any queue is also a pool, we compare to state-of-the-art pool implementations as well. The lock-free linearizable pool (BAG) [20] is based on thread-local lists of blocks of elements. Each block is capable of storing up to a constant number of elements. A thread performing an enqueue operation inserts an element into the first block of its thread-local list. Once the block is full, a new block is inserted at the head of the list. A thread performing a dequeue operation tries to find an element in the thread- local blocks first. If the thread-local list is empty, work-stealing from other threads' lists is used to find an element. The algorithm implements a linearizable emptiness check by repeatedly scanning all threads' lists for elements and marking already scanned blocks. These marks are cleared upon inserting an element, making the change visible to all scanning threads. The algorithm works only for a fixed number of threads.

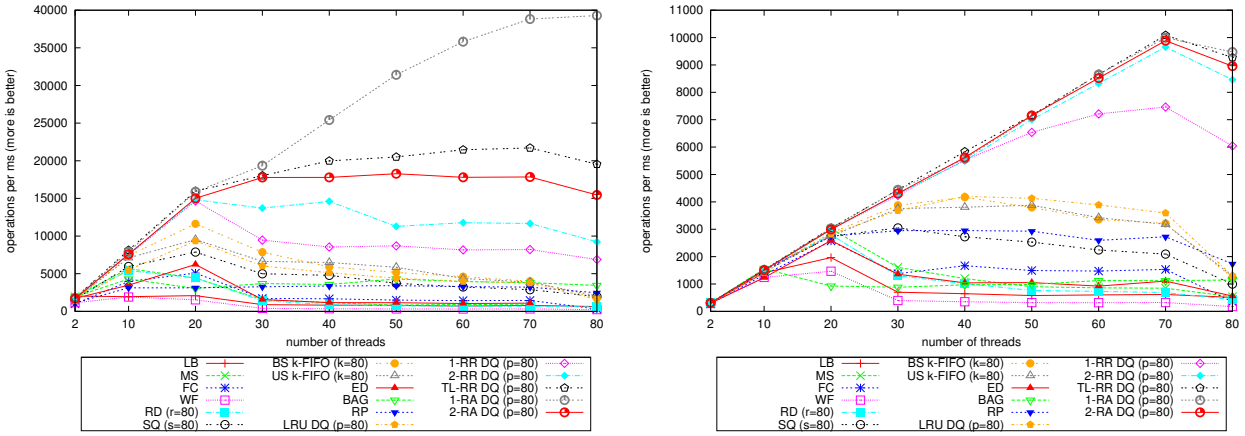
The lock-free elimination-diffraction pool (ED) [2] uses strict FIFO queues to store elements. Access to these queues is balanced using elimination arrays and a diffraction tree. While the diffraction tree acts as a distributed counter balancing access to the queues, elimination arrays in each node of the counting tree increase disjoint-access parallelism. Operations hitting the same index in an elimination array can either directly exchange their data (enqueue meets dequeue), or avoid hitting the counter in the node that contains the array (enqueue meets enqueue or dequeue meets dequeue). If based on non-blocking FIFO queues, the presented algorithm lacks a linearizable emptiness check. If based on blocking queues, there is no emptiness state at all. Parameters such as elimination waiting time, number of retries, array size, tree depth, number of queues, and queue polling time need to be configured to adjust ED to different workloads.

The synchronous rendezvousing pool (RP) [1] implements a single elimination array using a ring buffer. Both enqueue and dequeue operations are synchronous. A dequeue operation marks a slot identified by its thread identifier and waits for an enqueue operation to insert an element. An enqueue operation traverses the ring buffer to find a waiting dequeue operation. As soon as it finds a dequeue operation they exchange values and return. There exist adaptive and non-adaptive versions of the pool, where in the former case the ring buffer size is adapted to the workload.

5. EXPERIMENTS

We evaluate performance and scalability of the DQ algorithms, described in Section 3, and the LB, MS, FC, WF queue algorithms, the BS k-FIFO, US k-FIFO, SQ, RD relaxed queue algorithms, and the ED, BAG, and RP pool algorithms, described in Section 4.

All algorithms as well as the benchmarking code were implemented from scratch by us in C/C++ and compiled with gcc 4.4.3 and `-O3` optimizations. The experiments ran on an Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (40 cores, 2 hyper-threads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. We exercised the system with up to 80 benchmarking threads (40 cores times 2 hyper-threads) using the Linux round-robin scheduler with priorities. The priority of the benchmarking threads was set to just below the priority of kernel interrupts and the CPU governor was disabled. For heap management we implemented our own allocator based on thread-local allocation buffers [6] that initializes each benchmarking thread with a private pool of hot pages and guarantees that all allocated memory is page- and cache-aligned, all at the expense of memory fragmentation which may not be tolerable in stock allocators but is irrelevant here. We profiled our time and memory management not to introduce any scheduling, locking, paging, and



(a) High contention producer-consumer microbenchmark ($c = 250$) (b) Low contention producer-consumer microbenchmark ($c = 2000$)

Figure 1: Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyper-threads per core) server machine

caching artifacts into the data that are unrelated to the benchmarked implementations.

The algorithms are configured as follows. With d-RA we only evaluated $d = 1$ and $d = 2$ since in our benchmarks any value greater than two results in worse performance. With b-RR we evaluated $b = 1$ and $b = 2$ as well as the fully thread-local configuration TL-RR where b is set to the number of benchmarking threads. In our benchmarks 1-RR and TL-RR result in worst and best performance, respectively. Values for b greater than one and smaller than the number of benchmarking threads result in performance in between. We included 2-RR to demonstrate that but omitted other values not to overload the figures. The RD, SQ, k-FIFO, and DQ implementations are configured to $r = s = k = p = 80$ (see Sections 3 and 4), enabling up to 80 parallel enqueues and 80 parallel dequeues. We determined experimentally that 80 is the lowest value that results in overall best performance and scalability of the involved implementations in all our benchmarks.

5.1 Microbenchmarking Performance and Scalability

For measuring and comparing performance and scalability we designed a microbenchmark that emulates a multi-threaded producer-consumer workload where half of the threads are producers and the other half are consumers. Each thread performs one million queue or pool operations. We evaluate high and low contention scenarios by having each thread compute π iteratively between any two consecutive operations in $c = 250$ iterations (high contention) and $c = 2000$ iterations (low contention), respectively. As reference, $c = 1000$ iterations take on average 2.3 microseconds on the server machine. Higher contention with computational load down to $c = 0$ exposes machine-related artifacts resulting in meaningless data. The presented data is averaged over five runs. Note that we use this microbenchmark again in Section 5.3 to study the out-of-order behavior of all considered queue and pool algorithms.

Figures 1a and 1b show performance in operations per millisecond and scalability with an increasing number of threads for the high and low contention scenarios, respectively. The key observation when comparing high and low contention is that all implementations perform and scale better under low contention but still perform and scale similarly in relative terms in both scenarios. Overall 1-RA performs and scales best, followed by the other DQ implementations, which all outperform and outscale the other implementations including the pool implementations. Under high contention all implementations except TL-RR, 1-RA, and 2-RA scale negatively beyond 20 threads. The performance of 2-RR is in between the performance of 1-RR and TL-RR.

5.2 Macrobenchmarking Performance and Scalability

We evaluate performance and scalability with three macrobenchmarks based on spanning tree and transitive closure graph algorithms [5], and a Mandelbrot algorithm [17]. All presented data is averaged over ten runs.

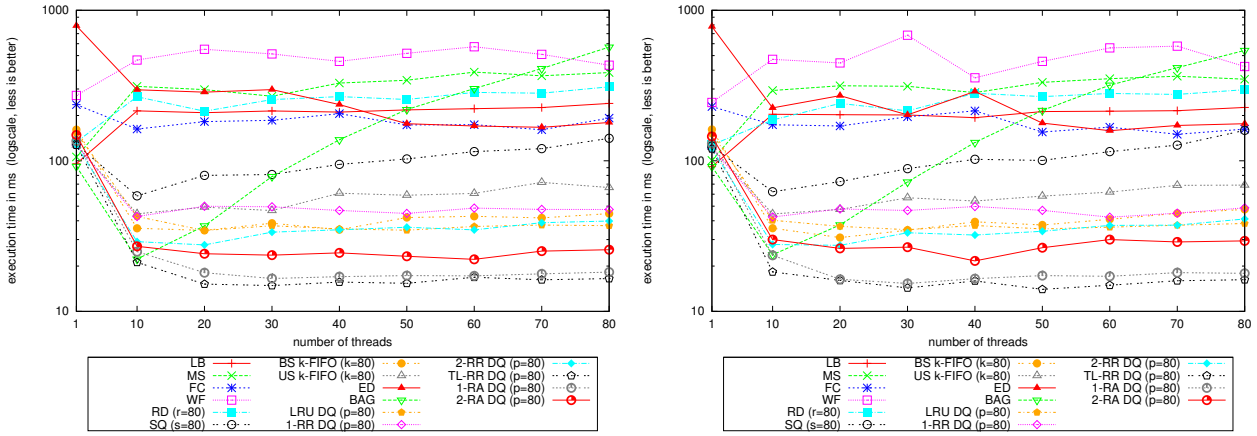
Spanning Tree and Transitive Closure Benchmarks

We ran the spanning tree and transitive closure graph algorithms on graphs consisting of a hundred thousand vertices and ten million randomly generated unique edges. Both algorithms use a shared queue or pool of vertices to distribute work among multiple threads. Initially, the shared queue or pool is prefilled with 160 randomly determined vertices. Each thread dequeues a vertex and then iterates over its immediate neighbors to process them (transitive closure or spanning tree operation). If a neighboring vertex already got processed by a different thread then the vertex is ignored. Otherwise, the vertex is processed and then enqueued. When a thread processed all neighbors it dequeues another vertex. The algorithms terminate when the shared queue or pool is empty.

Figures 2a and 2b show performance in terms of total execution time in milliseconds and scalability with an increasing number of threads. In both benchmarks, the DQ implementations perform and scale best. While most implementations are on par with DQ up to ten threads, only the BS and US k-FIFO queue implementations scale as much as DQ. Despite its thread-local storage BAG scales negatively because the connectivity of the graph makes it likely to hit already processed nodes and thus requires work-stealing. Note that RP is not shown because RP, due to its synchronous behavior, cannot handle a workload where producers are also consumers.

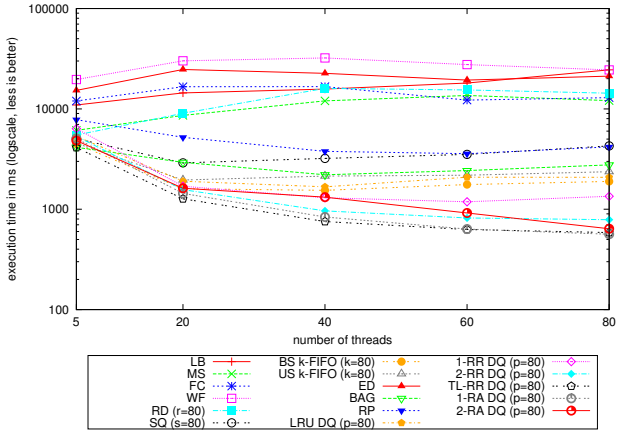
Mandelbrot Benchmark

The Mandelbrot benchmark renders an image by dividing it into blocks of $4x4$ pixels that are distributed by producer threads through a shared queue or pool to consumer threads for parallel process-

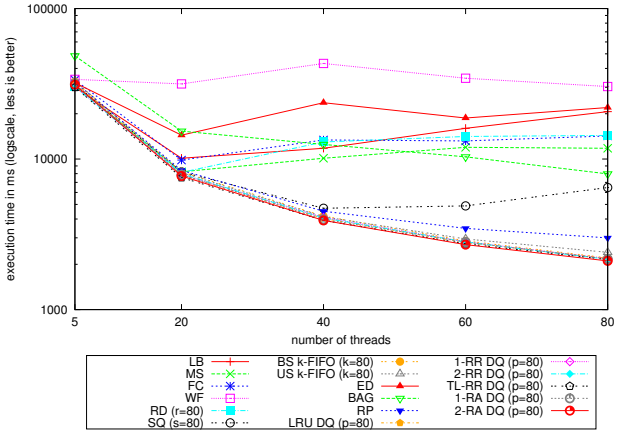


(a) Spanning tree benchmark

(b) Transitive closure benchmark



(c) Mandelbrot benchmark with low computational load



(d) Mandelbrot benchmark with high computational load

Figure 2: Macrobenchmarks for performance and scalability with an increasing number of threads on a 40-core (2 hyper-threads per core) server machine

ing. For each producer there are four consumers. We distinguish low and high computational load scenarios by using images whose blocks are mostly rendered either relatively fast or relatively slow, respectively.

Figures 2c and 2d show performance in terms of total execution time in milliseconds and scalability with an increasing number of threads. Best performance and scalability is achieved by TL-RR and 1-RA followed by the remaining DQ and the BS and US k-FIFO implementations. In the high computational load scenario all DQ implementations show identical performance and scalability. The BS and US k-FIFO implementations also perform and scale competitively.

5.3 Analyzing Relaxation

The results in Section 5.1 and Section 5.2 show that DQs outperform and outscale all strict and relaxed queues as well as pools we considered. In this section we analyze the out-of-order behavior of DQs and the other queues and pools in the high contention producer-consumer benchmark.

We instrumented all implementations such that the linearization points of all queue and pool operations are time-stamped using the globally synchronized TSC register of x86 processors which may be read with low overhead. Note that time-stamping linearization points may only be approximative but still close enough for our purposes. We record the timestamps of the linearization points together with the type and parameter of the operations and then compute the linearization of the run. We then compute for each element e the number of elements it overtakes in the linearization, called the relaxation distance of e. Finally, we compute the average relaxation distance of all elements in linearizations of runs of the high contention producer-consumer benchmark for all queue and pool implementations with an increasing number of threads. Figure 3 shows the highest average relaxation distance among five runs of the same configuration.

To improve the readability of Figure 3 we limit the y-axis to the relaxation distance of 100. The TL-RR DQ, the 1-RA DQ, the ED pool, and the BAG pool have higher average relaxation distances than 100 so that their lines are not always visible in the figure. The TL-RR DQ shows values up to 216, the 1-RA DQ up to 1288, the ED pool up to 4787, and the BAG pool up to 92671.

With LB, MS, FC, and WF elements do not overtake each other as these implementations are linearizable with respect to strict FIFO queue semantics. Their relaxation distance is therefore always zero in all configurations except when time-stamping was inaccurate.

The average relaxation distance of the 2-RA DQ is below 40 in all but one run although it provides no relaxation bound. The 1-RR DQ shows an average relaxation distance of at most 23 when accessed by 80 threads concurrently, and of at most 4 when accessed only by 30 threads. The relaxation bound of the LRU DQ is configured to be 79 in all benchmark executions. On average, however,

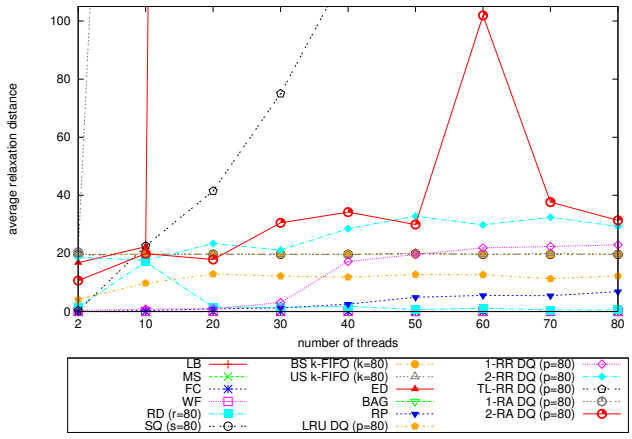


Figure 3: Average relaxation distance of all elements in the high contention producer-consumer microbenchmark ($c = 250$)

the relaxation distance of all elements is not more than 13. The k-FIFO queues, which have the same relaxation bound as the LRU DQ, show an average relaxation distance of about 20.

6. CONCLUSIONS

We presented a new family of distributed queue (DQ) algorithms that implement pools and relaxed queues via multiple strict queues. Access to each of the strict queues is coordinated either by a load balancer or through ABA counters that provide distributed synchronization. The semantics of DQs varies accordingly, from k-relaxed queues with configurable or bounded k to pools. All DQ algorithms provide a linearizable emptiness check. We demonstrated in micro- and macrobenchmarks that DQs outperform and outscale all strict and relaxed queues as well as pools that we considered. Interesting future work may consider the impact of replacing strict queues with DQs in applications that tolerate relaxation by relating application features to the degree of relaxation.

7. ACKNOWLEDGEMENTS

This work has been supported by the European Research Council advanced grant on Quantitative Reactive Modeling (QUAREM) and the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11402-N23 and S11404-N23).

8. REFERENCES

[1] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Proc. International Conference on Distributed Computing (DISC)*, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 151–162. Springer, 2010.

[3] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.

[4] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of Principles of Programming Languages (POPL)*, pages 487–498. ACM, 2011.

[5] D. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Journal of Parallel and Distributed Computing*, 65:994–1006, 2005.

[6] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proc. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128. ACM, 2000.

[7] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *Proc. Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 507–516. IEEE, 2005.

[8] D. Dice and O. Otenko. Brief announcement: multilane - a concurrent blocking multiset. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 313–314. ACM, 2011.

[9] T. Henzinger, C. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proc. Symposium on Principles of Programming Languages (POPL)*. ACM, 2013.

[10] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[11] D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.

[12] C. Kirsch, M. Lippautz, and H. Payer. Fast and scalable k-fifo queues. Technical Report 2012-04, Department of Computer Sciences, University of Salzburg, June 2012.

[13] C. Kirsch and H. Payer. Incorrect systems: It's not the problem, it's the solution. In *Proc. Design Automation Conference (DAC)*. ACM, 2012.

[14] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Brief announcement: Scalability versus semantics of concurrent FIFO queues. In *Proc. Symposium on Principles of Distributed Computing (PODC)*. ACM, 2011.

[15] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, LNCS. Springer, 2012.

[16] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 223–234. ACM, 2011.

[17] B. Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. *Annals of the New York Academy of Sciences*, 357:249–259, Dec. 1980.

[18] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[19] N. Shavit. Data structures in the multicore age. *Communications ACM*, 54:76–84, March 2011.

[20] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, New York, NY, USA, 2011. ACM.