Work-stealing Queues In Practice

Sune Alkærsig, Thomas Hallier Didriksen, and Christian Harrington
{sual, thdi, cnha}@itu.dk

IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

Abstract. There are many solutions for scheduling work across multiple threads, one of which is the work-stealing queue. In this paper we examine several implementations of work-stealing queues, such as the one described by Arora, Blumofe, and Plaxton, as well as several queues for idempotent work. We have implemented these queues for the Java Virtual Machine, and we benchmark them using four test cases, to be able to compare their relative performance. Our findings show that the simple Arora, Blumofe, and Plaxton queue is very competitive compared to later extensions. We also find that the idempotent queues' performance is greatly dependent on the task at hand, more so than the other implementations.

Keywords: Work-stealing queues

1 Introduction

A popular way of scheduling parallel workloads across multiple threads is through the use of work-stealing queues. Several variations of the canonical work-stealing queue exist, featuring different advantages and drawbacks. In this report we describe six approaches, in order present an overview of possible strategies for implementing work-stealing queues. We have also implemented these queues for the Java Virtual Machine (JVM), using the Scala programming language, along with several cases that use these queues. This is done in order to compare the performance of the different queues in different situations.

This report is structured as follows: In Section 2, we cover the necessary background, including different approaches to concurrency, as well as work-stealing queues themselves. We describe the six different queues we have implemented, along with their differences in Section 3. Section 4 describes the queue implementations, along with our benchmarking results for each case. An analysis of these results are presented in Section 5. Our testing approach is explained in Section 6, while we reflect on the project in Section 7. We conclude in Section 8.

2 Background

This section presents the concepts underlying the queue implementations and the subsequent analysis. In particular, different approaches to creating concurrent application are discussed.

2.1 Common Concurrency Problems

Many errors can arise which are directly caused by the presence of concurrent computations. One of the most important problems related to work-stealing queues is the *ABA problem*. This problem can occur in the context of a *compare-and-swap* operation, where the update is performed if the value of the given register is as expected. However, this provides no guarantees that the value has not changed since we last observed it, and then been changed back by another thread. Thus, the ABA problem gets its name from the notion that the register was A when we observed it, but then changed to B and back to A before our next observation. In some situations we need to prevent this from happening, which may require specific measures to be taken.

Another common problem is *check-then-act*. This problem can arise if a thread is supposed to perform some action as a consequence of the successful result of a check. If the thread first performs the check successfully, but is then suspended by the scheduler, then the result of the check might no longer be valid when the thread is later resumed, since other threads could have made changes to the shared state in the meantime. In such case, the invariants of the system could be violated if the planned action is performed by the resumed thread.

2.2 Approaches to Concurrency

The complexity of building concurrent applications arises mostly from the inherent need to manage concurrent access to shared state, in particular shared *mutable* state. With shared mutable state, any thread may write to any memory location at any time (in principle), which makes maintaining the invariants of the system a major concern. Keeping shared state immutable eliminates many such problems, as no threads will ever read different data from the same memory location at different moments in time. While it is arguably easier to reason about concurrent applications with immutable shared state, having mutable state can be desirable for several reasons, the most compelling of which is (possible) increased performance. In the following, we will discuss different approaches to handling concurrent access to shared mutable state.

Lock-based Synchronization Locking is a widely used mechanism for handling access to shared mutable state, especially in the realm of imperative programming. Locks can be used to grant exclusive access to one or more shared resources by associating that resource with a lock (assuming a locking scheme with *mutual exclusion locks*, where a given lock can only be held by one thread at a given point in time). The problem of synchronization is solved by a guarantee that no other threads will be accessing the shared resource as long as the lock is being held. Since any access to the resource must be exclusive, a lock must be obtained both when reading from and writing to it.

While the exclusive locking mechanism preserves the integrity of the shared state, this exclusivity is also one of the greatest disadvantages of locking. Whenever a thread tries to obtain a lock held by another thread, the call blocks and

the thread is suspended and must wait (potentially forever) for the current lock owner to finish its work. During this time, the waiting thread cannot do anything else. For locks under high contention, the performance degradation caused by the overhead of suspending and resuming threads might be very high, especially if the lock is only required for reading. For these situations, the use of non-blocking synchronization can be preferable.

Non-blocking Synchronization Instead of taking the pessimistic approach of making sure that exclusive access is obtained before performing any operations on a shared resource, non-blocking synchronization is more optimistic in nature [GPB+06]. Being *non-blocking* means that if a thread fails or is suspended, it cannot cause another thread to fail or be suspended, because threads accessing a shared resource never have to wait for another thread (contrary to the lock-based scheme, where a thread may have to wait for the release of a lock) [GPB+06]. A non-blocking algorithm can be said to be *lock-free*, which means that at any point of execution, some thread is making progress (and thus the term has nothing to do with lock-based synchronization).

In the non-blocking scheme, the integrity of a shared resource is ensured by only allowing updates to happen atomically from a consistent state, where all the invariants of the system hold. If some invariant does not hold, e.g. when a shared variable has another value than expected because it has been updated by another thread, a recovery mechanism is triggered instead. To perform atomic non-blocking updates, specific processor instructions (e.g. compare-and-swap) must be used to avoid common concurrency problems such as *check-then-act*.

Non-blocking synchronization provides a more fine-grained approach to synchronization, where the cost of suspending and resuming threads is replaced by the need for a recovery mechanism when faced with failed updates (retrying is a perfectly valid recovery mechanism). However, a non-blocking algorithm can be more complicated to design than a similar lock-based one [GPB+06], so using non-blocking synchronization might not always be desirable.

2.3 A Simple Solution

The simplest solution to the problem of concurrently processing n suspended computations (denoted "work units" or simply "work"), is to have m worker threads processing these in parallel from a central queue. To avoid two threads handling the same work unit, a lock is placed on this central queue, which must be held in order for a thread to take more work. Whenever two or more threads finish their current work simultaneously, they will all try to obtain the lock from the queue. However, only one of them will succeed, and the others will be suspended while waiting for the lock. In these situations, many threads will be idle at any given time when contention for the lock gets high. Consequently, this design leads to a situation where the central work queue becomes a bottleneck, and computational resources are wasted while threads are idle. Although not very efficient, a simple design such as this can serve as a good baseline for comparison with other approaches.

2.4 Work-stealing Queues

A work-stealing queue is a queue implementation that facilitates working with work-stealing scheduling algorithms [ABP98]. In a work-stealing scheduling algorithm, each thread is assigned a work-stealing queue, the elements of which are work units. This queue has the standard `push` and `take` operations for adding and removing elements from it, respectively, and only the thread owning the queue has access to these operations. Additionally, the queue exposes a `steal` operation, through which other threads may "steal work". An example of such a scenario is shown in Figure 1.

Work-stealing algorithms are especially well-suited for solving problems where one unit of work may generate more work (examples of which can be found in Section 4). The idea is that whenever the work unit processed by a given thread generates more work, this work is stored in the thread's local queue. When the thread finishes its current work, it first examines its own queue, and continues to process any work it might find. If no work is found locally, the thread begins to examine the queues of other threads, in effect "stealing" the work generated by these. This design leads to efficient utilization of computation resources, since a thread is never idle: it is either working or looking for more work.

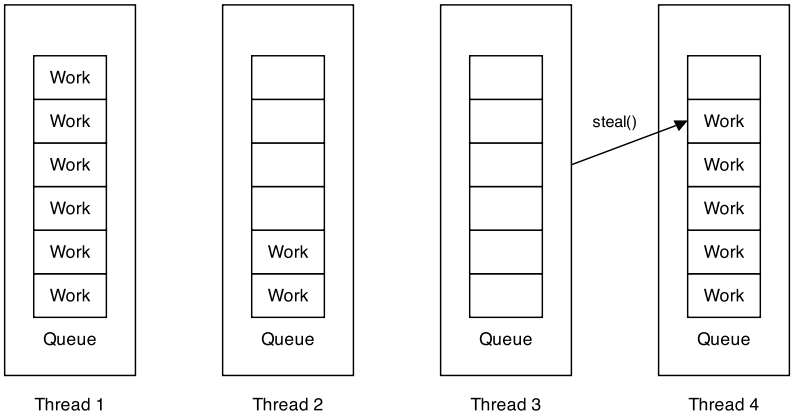


Fig. 1. An example situation from a work-stealing algorithm with four working threads. Each thread has its own queue. Thread 3 has completed all of its own work, and is therefore in the process of stealing work from thread 4.

2.5 Software Transactional Memory

Software Transactional Memory (STM) is an attempt to address the difficulty of designing concurrent algorithms and data structures, by enabling the user to

specify concurrent operations as transactions. Following the formulation of transactional memory presented by Herlihy and Moss [HM93], Shavit and Touitou proposed the first model for software-only transactional memory [ST95]. Using STM, shared memory can be accessed and updated by means of transactions, where a transaction is understood as an application of a finite sequence of primitive operations to that shared memory. Furthermore, the transactions must have the properties of *atomicity* (transactions appear to happen sequentially) and *serializability* (the sequential order of transactions is consistent with their real-time order) [ST95].

Ideally, designing concurrent algorithm with STM should shift much of the burden of worrying about concurrency from the user to the runtime system, without incurring any performance trade-offs. A study done by Dice and Shavit [DS06] shows that algorithms implemented using STM scale better than equivalent hand-crafted algorithms, both lock-based and non-blocking. However, opponents of STM claim that it causes a higher sequential overhead than more traditional shared-memory approaches [CBM+08], since transactions expand to many more load and store instructions, leading to performance losses.

In an improvement to their earlier algorithm, Dice, Shalev, and Shavit [DSS06] present the *Transactional Locking II* algorithm, which supposedly is "an order of magnitude faster than sequential code made concurrent using a single lock." This is achieved using a global version-clock, which allows for fast read-only transactions. Although these findings seem promising, the debate concerning the performance of concurrent algorithms implemented using STM has yet to reach consensus.

3 Implementations

In this section we will present the 6 queues we have implemented. The first three are based on the Arora-Blumofe-Plaxton (ABP) queue, with later versions bringing various improvements. The final three are so-called idempotent queues, which we will explain in Section 3.3.

3.1 Arora-Blumofe-Plaxton Queue

The ABP queue [ABP98] stores its tasks in a fixed size array, and uses `top` and `bottom` pointers to keep track of the beginning and end of the queue. Pushing a task to the queue (`push`) is straightforward. It is added to the array at the bottom, and `bottom` is incremented. Taking tasks from the queue requires more work to ensure that the same work is not extracted twice, as `take` and multiple different `steal` operations can happen simultaneously. Avoiding duplication is done by only allowing `steal` to return if the `top` has remained unchanged throughout the operation. This, however, is not sufficient [ABP98], as it is still susceptible to the ABA problem. If the queue is emptied and refilled to the same size between the read from the queue (Figure 2, line 9) and the comparison with `top` (Figure 2, line 12) the read might no longer be valid. To handle this, a `tag`

value is added, which is incremented whenever the queue is completely emptied. This `tag` together with `top` is now known as `age`. Now, instead of just the `top` having to remain unchanged, the entire `age` must be the same. While the ABP queue is relatively simple to implement, it suffers from its use of a fixed size array. Because of this, one must know the maximum number of tasks beforehand, or risk overflow errors.

```
1  final def steal(): Option[E] = {
2      val oldAge = age.get
3      val localBot = bottom
4
5      if (localBot <= oldAge.top) { // Queue is Empty
6        None
7      }
8      else {
9        val v = queue.get(oldAge.top)
10       val newAge = Age(oldAge.tag, oldAge.top + 1)
11
12       if (age.compareAndSet(oldAge, newAge)) {
13         // CAS for ABA prevention
14         Some(v)
15       }
16       else {
17         None
18       }
19     }
20   }
```

Fig. 2. ABP implementation of `steal`.

3.2 Chase-Lev Queue

Chase and Lev designed an improved version of the ABP queue using circular arrays [CL05]. The Chase-Lev implementation solves the fundamental problem from which the ABP queue suffers, namely the use of fixed-size arrays. Besides the risk of overflows, the fixed-size arrays also lead to inefficient memory usage: In a system with n threads and allocated memory m, any queue in the system can hold at most $\frac{m}{n}$ elements. Using dynamic circular arrays, any Chase-Lev queue can grow when necessary, which means that the shared memory does not have to be divided among the queues upfront. To improve memory usage, Chase and Lev also describe a shrinking operation for the underlying circular arrays, such that memory is reclaimed whenever the actual size of an array is less than the allocated size by some constant factor. Since the `top` pointer is never decremented, the size of a given Chase-Lev queue is only restricted to the size of `top` (Chase and Lev use a 64-bit integer). Contrary to the ABP queue, the Chase-Lev implementation does not need a tag to avoid the ABA problem. As a direct consequence of the `top` pointer never decrementing, any thread will always notice if the contents of the queue have changed (using a *compare-and-swap* operation on `top`). For this project we have implemented two versions of

the Chase-Lev queue, one with shrinking and one without. While all our tests have been run in a garbage-collected environment on the JVM, the Chase-Lev queue does not rely on garbage collection for memory management.

3.3 Idempotent Queues

The previously discussed queues guarantee that no task will be extracted twice. This can be relaxed in some cases to possibly improve performance [MVS09]. The rest of the queues do not provide this guarantee, which means that they can only be used in cases where the task is idempotent, i.e. they can be performed multiple time without changing the result.

We have implemented three different approaches to idempotent work stealing [MVS09]. Generally, all three work in a similar fashion, but the order in which tasks are extracted differ from queue to queue. All three queues keep their tasks in an array, but use different ways of keeping track of the ends of the queue When talking about these structures we use the term "queue" loosely to mean a data structure that keeps its elements in some order.

The Idempotent LIFO Queue [MVS09] does not need a special pointer to track the beginning of the queue, as tasks are always added to, and extracted from the end. The end of the queue is determined by a `tail` pointer, which always points to the last task in the queue. To prevent the ABA problem, the `tail` is bundled with a `tag` in an `anchor`. This `anchor` must remain unchanged between the read of a stolen element and the return of this element. It is worth noting that, in contrast to the other queues, this queue (along with the Idempotent FIFO queue) `steal` and `take` from the same end of the queue.

The Idempotent FIFO Queue [MVS09] adds tasks to the end of the queue, defined by a `tail` pointer, and removes from the beginning of the queue, determined by a `head` pointer. This means both `take` and `steal` operate on the same end of the queue. Interestingly, this implementation does not need ABA prevention at all. This is because `head` can only decrease in one case:

1. A `take` operation begins, and reads the `head` pointer.
2. Before the `take` completes, one or more threads `steal`, incrementing `head`.
3. The original `take` operation completes, incrementing `head` by 1 compared to its original value, the result of which is lower than the `head` set by the other threads.

However, this does not matter, as a decreasing head would only cause a problem if executed concurrently with a `push` operation. This is not possible, as only the thread owning the queue can perform the `take` and `push` operations.

The Idempotent Double-Ended Queue [MVS09] is unusual, as elements are pushed to and stolen from the beginning of the queue, while they are taken from the end. This means that this implementation also needs ABA prevention. This is done in a similar way as the LIFO queue, with a `anchor` packed with the `head`, `size`, and a `tag`.

3.4 Duplicating Queue

Similar to the idempotent queues, the duplicating queue [LSB09] can potentially return a pushed task more than once. When stealing from this queue, the stolen task is either removed from the queue or just duplicated. When a task is the only element in the queue, concurrent `steal` (stealing from the `head`) and `take` (taking from the `tail`) operations will access the same task. In this case we say that the task has been "duplicated". Two threads cannot steal the same element from the queue as stealing requires a lock.

Furthermore, if a new element is pushed just after the above duplication, the `head` and `tail` pointers are inconsistent [LSB09]. This is solved by another pointer `tailMin` indicating the minimal index at which a task has been taken. When checking if the queue is empty when taking, `head` is compared to the smallest value of `tailMin` and `tail`.

3.5 Comparison

Our implementations can be split into two categories: those that are idempotent, and those that are not. The idea behind the idempotent queues is that the queue operations themselves are simpler, but at the risk of performing the same task multiple times. This means that we can expect the idempotent queues to perform well in cases where there are many queue operations. They will, however, not work in cases where the same work is not allowed to be done twice.

Amongst the idempotent queues, the duplicating queue stands out. Since it is only idempotent when the owner and exactly one thief attempts to access a queue with exactly one element, duplication of work happens less often. This comes at the cost of slowing down concurrent `steal` operations, as the lock only lets one thread steal at a time.

The ABP-based queues are very similar, but the key difference is that the size of the ABP queue's array is fixed, whereas Chase-Lev can grow. We have also made an implementation of Chase-Lev that also shrinks the queue, which should be even more memory efficient, at the cost of performance. This means that while ABP is less memory efficient, it might perform better than the two Chase-Lev queues, as it does not need to grow or shrink the queue.

4 Benchmarks

To test our different queue implementations, we developed a small framework for parallelizing computations using work-stealing queues. There are three main Scala components: the `WorkerPool`, `Worker`, and `Node` traits. For each of the cases below, classes implementing these traits were created. The `WorkerPool` is in charge of starting the desired number of threads, and assigning each `Worker` to a thread. A `Worker` holds a queue, and contains the main logic for the case. `Node`s represent individual "work units", and are the elements contained in the queues.

4.1 Cases

To test our different work-stealing queues, we have implemented four different example cases.

Raw Queue Operations The first of these cases, which we call "Raw", is meant to do as little work as possible, and thus stress the queue implementations as much as possible. A tree of `Node`s is built, each of which hold an arbitrary number of children, and nothing else. Each `Worker` simply looks at a `Node`, and adds all its children to the queue, before moving to the next `Node`. This means that queue operations will be the deciding factor for how long a run will take. It should be noted that this is an unrealistic workload, as no actual work is done, besides traversing the tree.

Quick Sort Our second case is a simple quick sort implementation. It is simple in that it does not sort in-place. Instead, when splitting an array, each new child `Node` receives a full copy of the array to split. When a lower threshold for array length is reached, in-place insertion sort is used. Once an array has been sorted using insertion sort, its parent `Node` is notified. If both the parent `Node`'s children have been sorted, the parent `Node` is added to the queue. When the parent `Node` is examined by a `Worker`, the two sorted subarrays are combined, and the next parent `Node` in the hierarchy is notified. When both children of the root `Node` are combined, the entire array has been sorted.

In Section 3 we discussed different queue implementations, some of which only work with idempotent tasks. At first glance, quick sort should be an idempotent problem as it is fine to sort a list multiple times. The problem becomes apparent when we consider the actual implementation. Since the sorting in the leafs is done by an in-place insertion sort, inconsistencies can occur if two workers sort the same list at the same time. This could be solved by adding synchronization to the insertion sort, but that somewhat defeats the purpose of work-stealing queues.

Spanning Tree For our third case, we implemented the parallel spanning tree algorithm described by Bader and Cong [BC04], with some small differences. Their algorithm functions by first letting a single thread build a "stub tree", a small portion of the full tree built by randomly walking the graph. This tree's vertices are then evenly distributed into each queue. After this, each thread starts consuming work from their queues. Our implementation is slightly different, in that we do not build the initial stub tree. Instead, the first `Worker` is given a node in the graph to work on, and as it generates more work, the other `Worker`s can steal from it. This change was purely to make the implementation simpler, and does not affect the correctness of the algorithm, although it might make the initial steps slightly slower.

XML Serialization Our final case was inspired by Lu and Gannon's Parallel XML Processing algorithm [LG07]. By using work-stealing queues, the processing of an XML document can be easily load-balanced without knowing its structure beforehand. For this test case, we serialize a model of an XML document to a string. This is is done in a manner reminiscent of the quick sort case. When a `Node` is examined by a `Worker`, all its children are added to the queue. If the `Node` is a leaf, it is serialized, and its parent `Node` is notified. Once all the children of a `Node` have been serialized, the `Node` itself is serialized. The work is done when the root `Node` has been serialized.

As with the Quick Sort case, our implementation of XML Serialization is not idempotent. This is caused by the mechanism through which a parent `Node` keeps track of its completed children: An atomic counter is incremented whenever a child `Node` has been processed. Therefore, if a child `Node` was processed more than once, the counter in the parent `Node` would not accurately represent the number of completed children. As a consequence, the parent `Node` would never be able to know whether *all* of its children had been completed, or if the same children had simply been processed multiple times.

4.2 Results

Each of the four test cases were executed 50 times with the same random data. They were tested with 8, 16, 32, and 48 threads. Due to technical difficulties, they were not tested with more than 48 threads. See Section 6 for an explanation. All benchmarks were performed on a machine running Ubuntu 12.04.4, with two AMD Opteron 8386 SE 16 core processors, and 128GB of memory.

An overview of our results with 16 threads can be seen in Figure 3. The rest of our results can be found in Appendix A.

5 Performance Analysis

This section analyses the results of our benchmarks. We focus on some interesting observations, and discuss the cause behind the results we are seeing.

5.1 Chase-Lev vs. ABP

When comparing the performance of the Chase-Lev and ABP implementations in Figure 4, we see that Chase-Lev is faster at quick sort and XML serialization. On the surface it might seem odd that Chase-Lev should outperform ABP as Chase-Lev also has to maintain the circular array. However, as Chase-Lev does not have to deal with the ABA problem this better performance could be due to a lot of failed CAS operations in ABP. In general, Chase-Lev therefore cannot be expected to perform better than ABP. We can also see that Chase-Lev tends to be slower than ABP on cases with low cost tasks (Spanning tree and Raw) and speeds up as tasks become more work intensive. If more time is needed for each

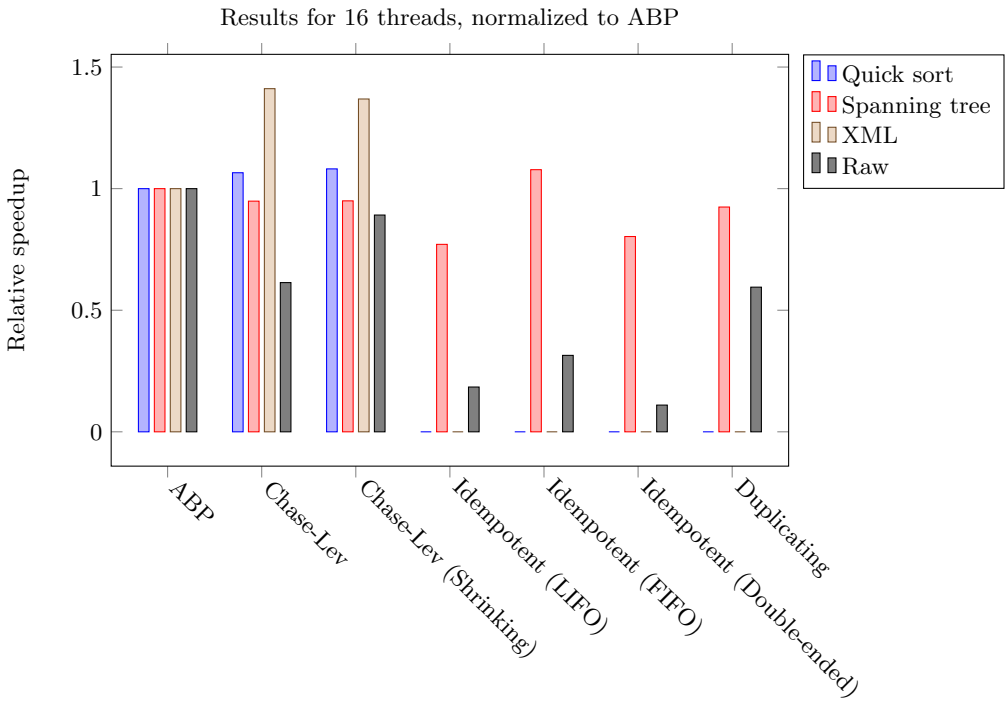


Fig. 3. Average of all results with 16 threads.

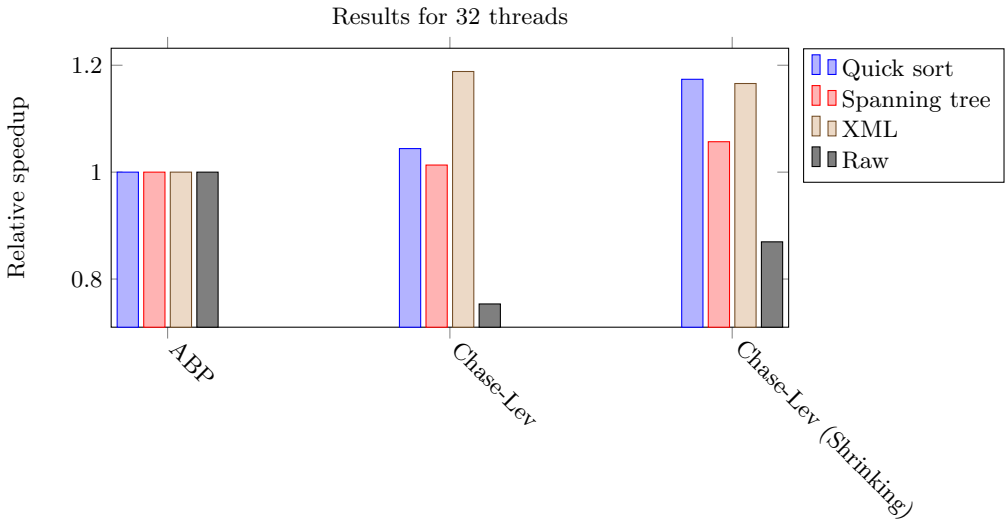


Fig. 4. Results for 32 threads with non-idempotent queues.

task, the relative time spend on queue operations lowers and thus the increased overhead of Chase-Lev has less of an effect on the overall performance.

Another advantage of Chase-Lev queue is that it does not need to know the size of the queue beforehand. This allows for more memory efficiency and makes it easier for the user. This means that Chase-Lev in general is a good choice for cases where the number of subtasks created by each task is hard to predict precisely, and especially for cases with more intensive individual tasks.

We have also made a shrinking version of Chase-Lev which has yielded some interesting results. Since this queue requires more overhead to not only expand the queue, but to also shrink it, we expected it to perform worse than both ABP and the standard Chase-Lev queue. However, it has performed just as well as the two others and even outperformed them in certain cases. The reason for this could be garbage collection. Since ABP has to have a queue at the size of the worst case scenario, most of the time only very little of that will actually be used. In contrast, the Chase-Lev shrinking queue is only as big as needed, and the leftovers are garbage collected during execution. This means that there are many more items for the garbage collector to keep track of in the ABP queue which might affect the performance when compared to the Chase-Lev queues.

5.2 Quick Sort and XML Serialization Scaling

As can be seen from Figure 5, the Quick Sort case does not scale well past 16 threads. The XML Serialization case also scales poorly past 16 cores (see Appendix A). While we have not been able to pinpoint the exact cause of this, we believe it to be caused by the two CPUs of the test system. CAS operations can incur significant overhead when dealing with multiple cores not on the same die [DS06, p. 19–20], but this should affect all our use cases equally. One of the things that sets the Quick Sort and XML Serialization apart from the other cases is that they are much more memory intensive. Keeping all this data synchronized between the two CPUs might be the cause of the slowdown past 16 cores.

5.3 Performance of Idempotent Queues

Looking at Figure 6, the idempotent queues are very slow in the Raw case. At first this does not seem right, as one of the advantages of the idempotent queues should be faster queue operations, which is exactly what Raw tests. The reason for this bad performance can be that there is no synchronization of information on what work is done and what needs to be done. If two threads receive the same task from a queue, they both spawn the children of that task, which will both be added to their respective queues. This means that if two workers perform the same task, not only the task itself has to be executed twice, but also all the children of that task.

The above potential explanation for the idempotent queue performance is backed by the fact that the duplicating queue is a lot faster at Raw than the other idempotent queues. This is because the duplicating queue only duplicates work when exactly one thief steals a task at the same time as the owner. In the cases

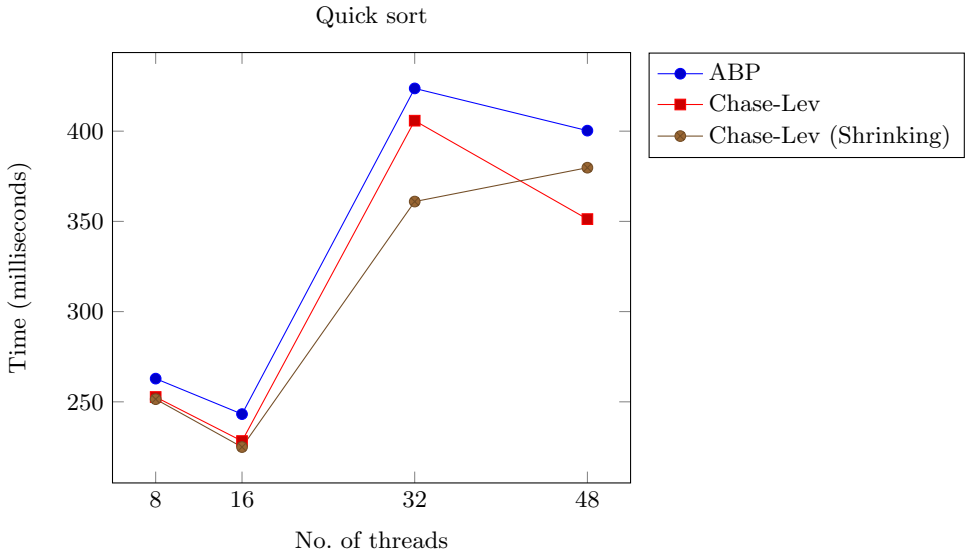


Fig. 5. Quick sort scaling.

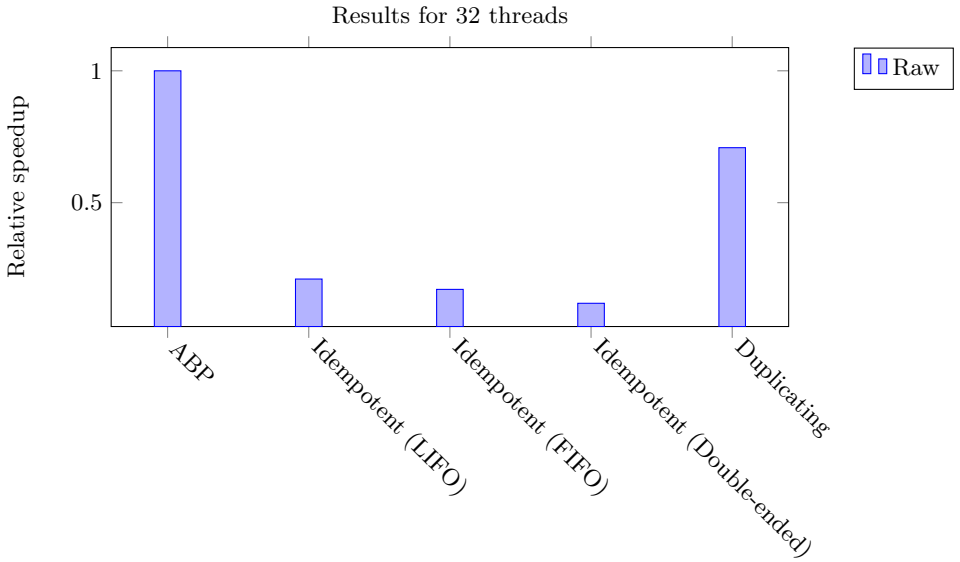


Fig. 6. Idempotent performance on Raw compared to ABP.

of the other idempotent queues it can happen in many other cases, and therefore occur much more frequently. With less work duplication, the duplicating queue performs a lot better in the Raw case.

6 Testing

Testing parallel algorithms with shared mutable state can be challenging. By using unit testing, we have been able to find many bugs when running our cases with a single or few threads. The problems arise due to the many possible interleavings when running with many threads. During our final benchmarks, we encountered bugs when running with more than 48 threads. Due to the nature of the bugs, we did not have time to find a solution.

This problem might have been found earlier, and fixed, if we had used an automatic testing solution that can test all possible interleavings. Microsoft CHESS can be used in .NET to test many possible interleavings. For the JVM, NASA has developed Java Path Finder. Both these tools can force certain interleavings, making it much easier to find the kind of bug we encountered. Unfortunately, Java Path Finder is very difficult to set up, and we found it too late in our development for us to use it.

7 Reflection

Besides the trouble with testing described in Section 6, we encountered several problems when trying to perform benchmarks on our various queue implementations. The most important of these issues are described below.

7.1 Cases For Idempotent Queues

While performing the benchmarks, we realized we had made a mistake when choosing our cases. First of all, when choosing which cases to implement, we did not consider which could easily be run with the idempotent queues. This meant we only had two cases to test the idempotent queues with. Further, the two cases that happen to work with the idempotent queues are the two cases that involve the least work for each thread. This makes it hard for us to tell to which degree the performance of the idempotent queues is affected by queue operations versus the overhead of possibly performing a piece of work multiple times.

7.2 Statistics For Queue Operations

Perhaps the greatest problem we found when analyzing our benchmarks, was the lack of detailed metrics for the queues. For instance, it would be very useful to be able to measure how many unsuccessful steals occur, how much work is duplicated, and how long each of these operations take. This degree of detailed information would let us reason more accurately about the performance characteristics of our different queues and cases. Unfortunately, it is difficult to make

these types of measurements without affecting the execution of the program. If we were only working with single threaded workloads, the additional overhead from the logging would be easy to account for. But in our program, where specific interleavings of multiple threads can produce reasonably large changes in execution time, it is hard to account for the effects of logging. Regrettably, we did not find an acceptable solution within the scope of this project.

7.3 Hardware

One unexpected area where we encountered difficulties was with the processors on the machine used for benchmarks. For most of the project we were under the impression that it was a 16 core Intel Xeon processor with hyper-threading. Later, we found out hhat it had two AMD Opteron processors with 16 cores each. Strangely, the information in `/proc/cpuinfo` and the output from `lscpu` both implied only 8 physical cores were available on each processor, with each core exposing two logical cores. To our knowledge, AMD does not support a technology equivalent to hyper-threading. Because of this, some uncertainty exists as to how the machine is actually configured.

7.4 Implementations with Software Transactional Memory

In addition to the presented implementations, we also implemented all of the queues using ScalaSTM [SSDT14], a Scala library for implementing algorithms with software transactional memory. When we tried to benchmark these implementations, however, they all ended up in an infinite loop. We do not know whether our implementations were faulty or if the library simply did not work.

As mentioned in Section 2, the philosophy behind STM is that the burden of thinking about concurrency should be shifted from the developer to the runtime system. Nevertheless, we found that the code for the queues became somewhat harder to read when implemented with STM, since we had to unfold all compare-and-swap operations into conditionals. We believe that the reason for this mostly lies in the fact that the queue implementations were not presented with STM in mind by their respective authors.

8 Conclusion

We set out to examine different work-stealing queue implementations and compare their performance. To do so, we discussed and analyzed several approaches, as well as implemented them for the Java Virtual Machine. These implementation were benchmarked in order to compare their performance. Based on the literature and our intuition we expected the Chase-Lev queues to be about as fast as the ABP queue, and the idempotent queues to be even faster. Our results show that while the ABP queue performs well, it was out-performed by the Chase-Lev implementations, possibly due to the heavier burden placed on the garbage-collector. Interestingly, the idempotent queues did not perform as

well as expected. The reason for this might be found in cases we have used for testing them. This could indicate that the performance of idempotent queues is quite case dependent.

Bibliography

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[BC04] David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps) (extended abstract), 2004.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008.

[CL05] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[DS06] Dave Dice and Nir Shavit. What really makes transactions faster? In *TRANSACT Workshop*, page 2006, 2006.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java concurrency in practice*. Addison-Wesley, Upper Saddle River, NJ, London, 2006.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[LG07] Wei Lu and Dennis Gannon. Parallel xml processing by work stealing. In *Proceedings of the 2007 Workshop on Service-oriented Computing Performance: Aspects, Issues, and Approaches*, SOCP '07, pages 31–38, New York, NY, USA, 2007. ACM.

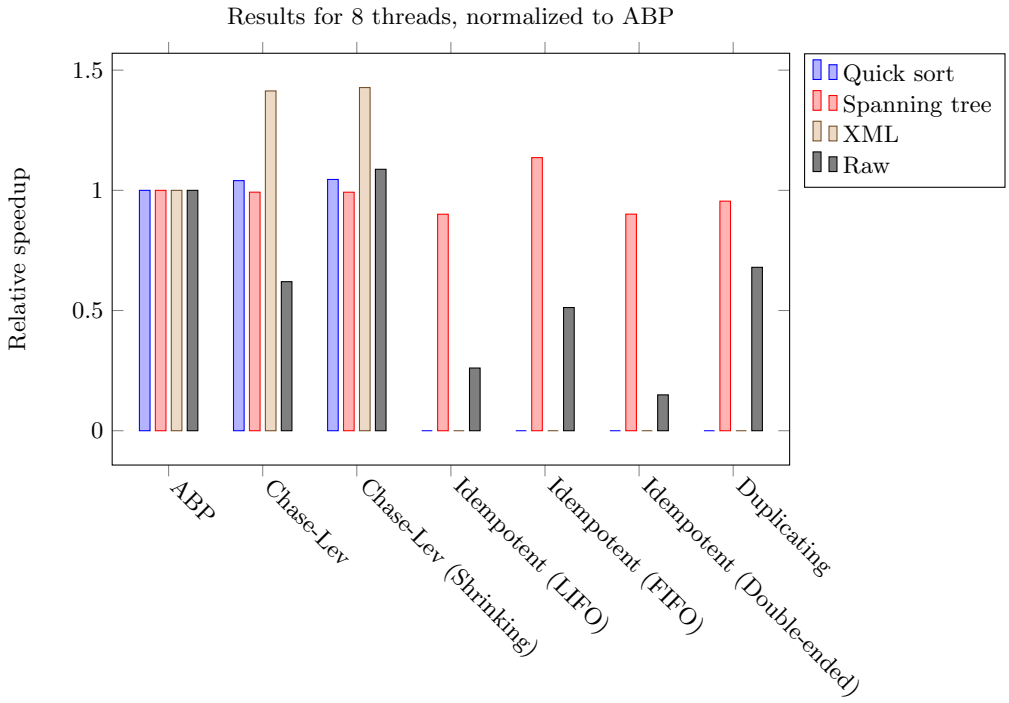
[LSB09] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, 44(10):227–242, October 2009.

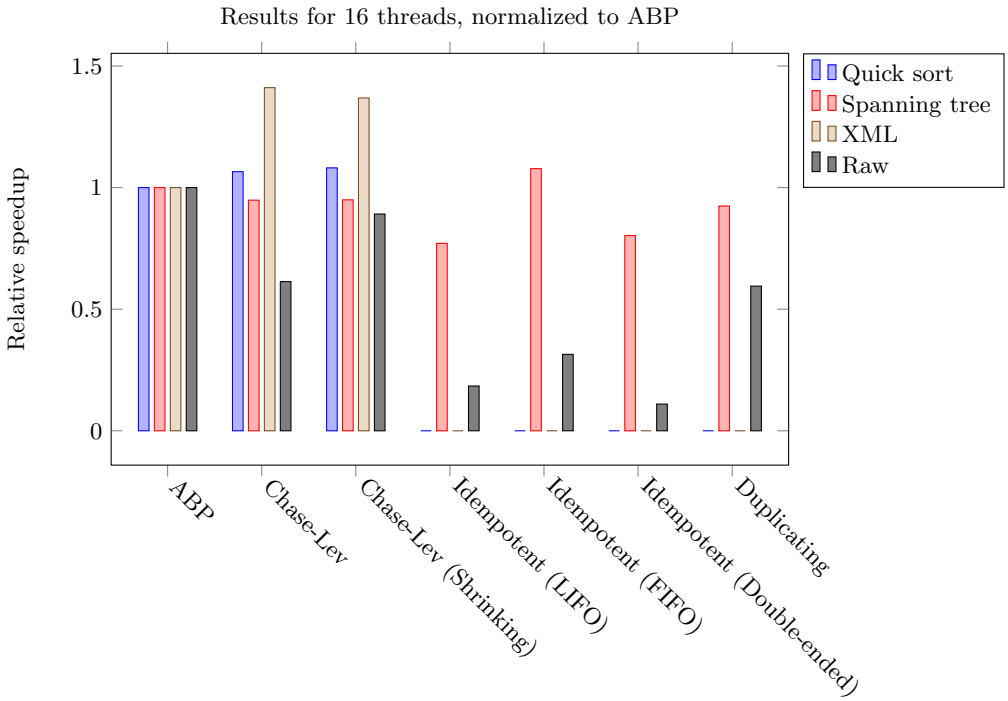
[MVS09] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. *SIGPLAN Not.*, 44(4):45–54, February 2009.

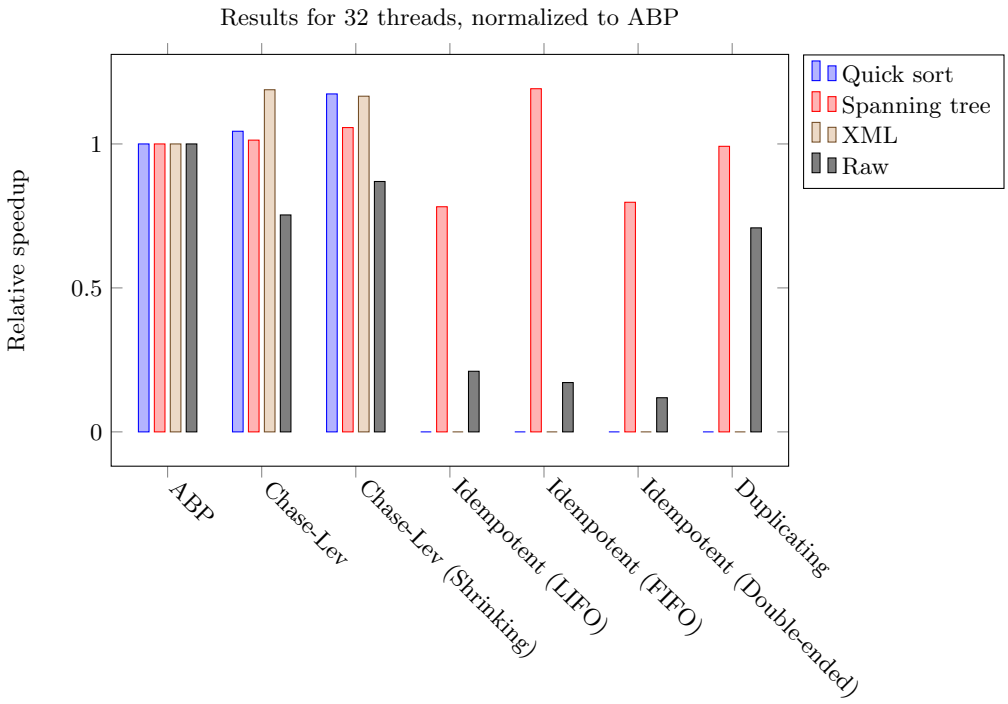
[SSDT14] The Scala STM Development Team. http://nbronson.github.io/scala-stm/, 2014. Accessed: 19-05-2014.

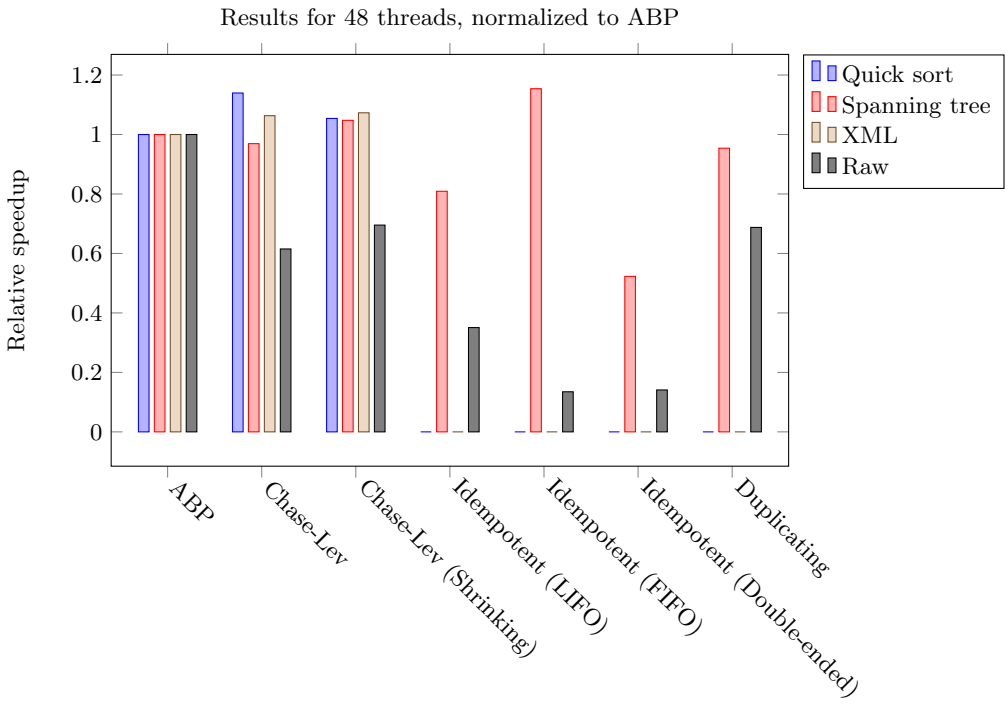
[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, 1995.

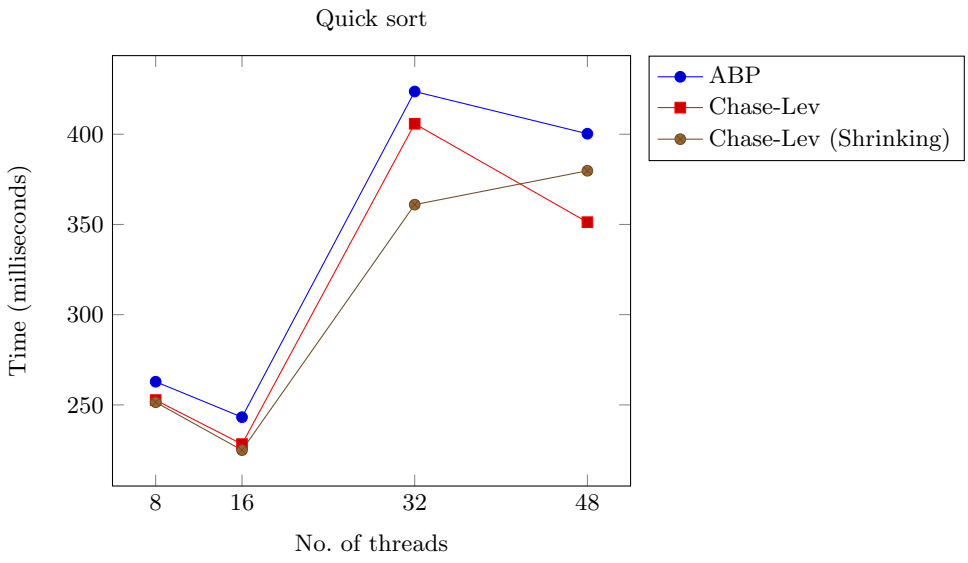
A Results

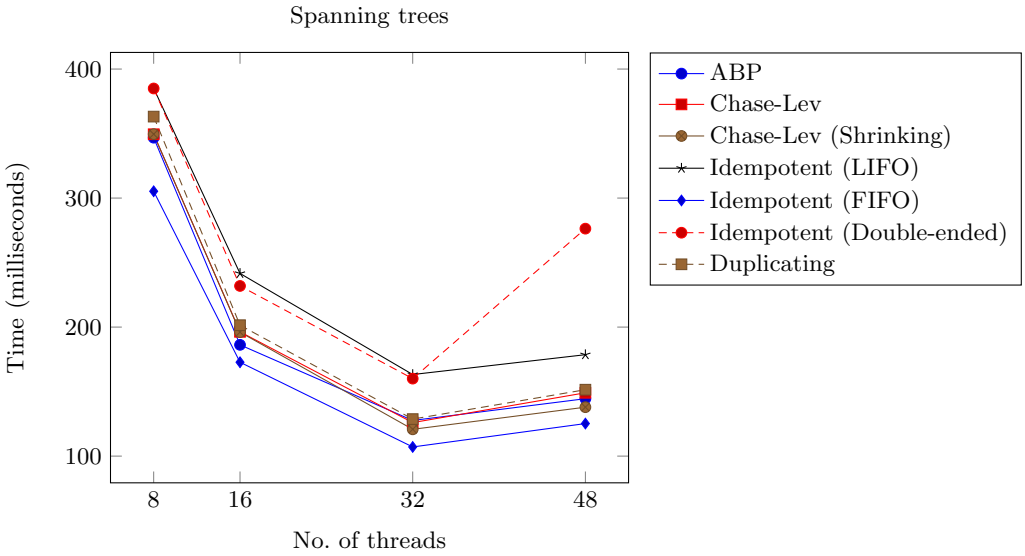


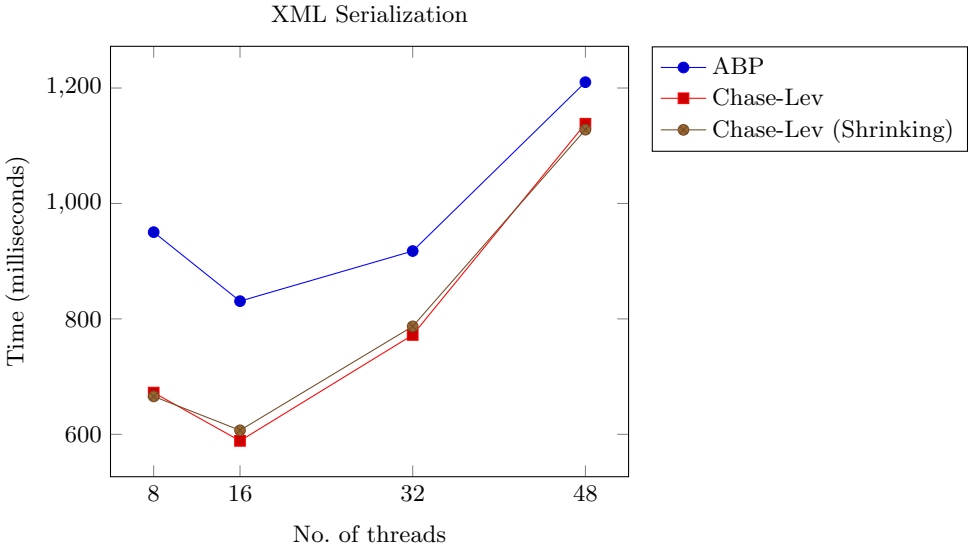

Results for 16 threads, normalized to ABP
Relative speedup
1.5
1
0.5
0
Quick sort
Spanning tree
XML
Raw
ABP
Chase-Lev
Chase-Lev (Shrinking)
Idempotent (LIFO)
Idempotent (FIFO)
Idempotent (Double-ended)
Duplicating

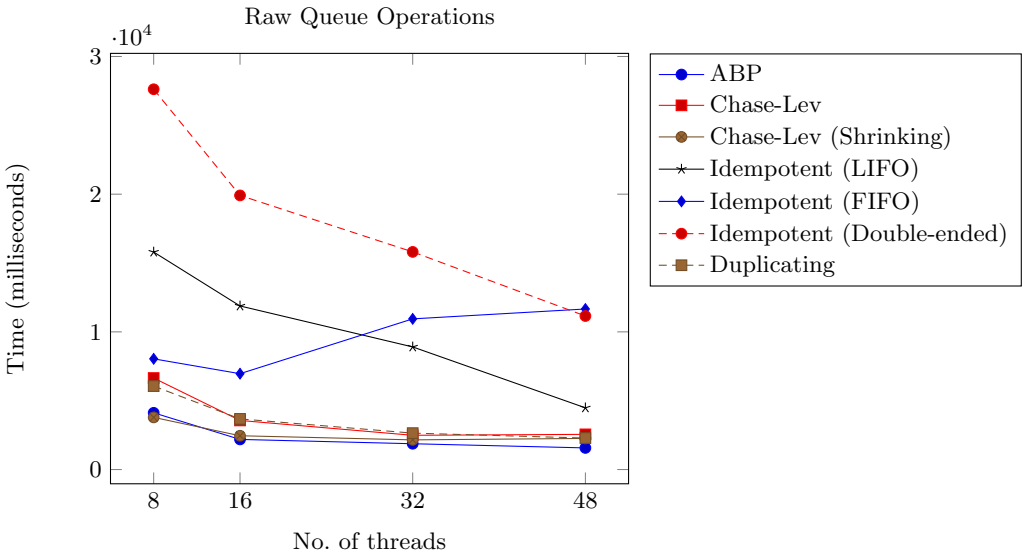

Results for 32 threads, normalized to ABP
Relative speedup
1
0.5
0
Quick sort
Spanning tree
XML
Raw
ABP
Chase-Lev
Chase-Lev (Shrinking)
Idempotent (LIFO)
Idempotent (FIFO)
Idempotent (Double-ended)
Duplicating


Results for 48 threads, normalized to ABP
Relative speedup
1.2
1
0.8
0.6
0.4
0.2
0
Quick sort
Spanning tree
XML
Raw
ABP
Chase-Lev
Chase-Lev (Shrinking)
Idempotent (LIFO)
Idempotent (FIFO)
Idempotent (Double-ended)
Duplicating


Quick sort
Time (milliseconds)
400
350
300
250
8
16
32
48
No. of threads
ABP
Chase-Lev
Chase-Lev (Shrinking)


Spanning trees
Time (milliseconds)
400
300
200
100
8
16
32
48
No. of threads
ABP
Chase-Lev
Chase-Lev (Shrinking)
Idempotent (LIFO)
Idempotent (FIFO)
Idempotent (Double-ended)
Duplicating


XML Serialization
Time (milliseconds)
1,200
1,000
800
600
8
16
32
48
No. of threads
ABP
Chase-Lev
Chase-Lev (Shrinking)


Raw Queue Operations
$\cdot 10^4$
3
2
1
0
Time (milliseconds)
8
16
32
48
No. of threads
ABP
Chase-Lev
Chase-Lev (Shrinking)
Idempotent (LIFO)
Idempotent (FIFO)
Idempotent (Double-ended)
Duplicating

8 Threads

Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	262,86	252,67	251,47				
Spanning Tree	346,81	349,47	349,53	385,08	305,27	384,94	363,16
XML	950,29	672,29	665,72				
Raw	4120,79	6646,03	3788,95	15797,32	8041,34	27620,75	6061,63
Standard deviation:							
Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	16,89	10,58	9,02				
Spanning Tree	35,24	31,44	29,20	37,96	35,62	33,75	30,62
XML	1006,75	32,09	35,59				
Raw	955,72	2130,07	1136,50	5023,73	2084,07	15317,67	1706,88

16 Threads

Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	243,20	228,27	224,93				
Spanning Tree	186,23	196,33	196,08	241,58	172,76	231,89	201,47
XML	830,74	588,71	606,98				
Raw	2188,83	3566,86	2455,44	11881,41	6962,21	19909,84	3679,41
Standard deviation:							
Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	26,49	26,05	21,12				
Spanning Tree	20,45	20,71	19,16	20,33	18,86	26,71	21,69
XML	808,90	69,14	82,53				
Raw	295,43	857,05	574,69	4254,40	1993,84	7394,86	814,02

32 Threads

Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	423,68	405,80	360,98				
Spanning Tree	127,63	125,96	120,76	163,23	107,09	160,05	128,69
XML	917,55	772,17	787,06	[redacted]			
Raw	1874,62	2488,21	2155,72	8905,86	10943,90	15807,66	2645,78
Standard deviation:							
Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	47,11	42,93	38,68				
Spanning Tree	21,31	19,77	17,88	22,89	18,20	24,53	22,13
XML	473,67	40,08	33,35				
Raw	227,27	401,09	313,83	2370,95	2422,73	5827,88	311,26

48 Threads

Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	400,28	351,26	379,75				
Spanning Tree	144,5	149,09	137,92	178,60	125,22	276,30	151,47
XML	1210,11	1138,10	1127,79				
Raw	1572,27	2556,13	2261,03	4482,83	11661,40	11152,57	2286,77
Standard deviation:							
Benchmark	ABP	Chase-Lev	Chase-Lev Shrinking	Idempotent (LIFO)	Idempotent (FIFO)	Idempotent (Double-Ended)	Duplicating
Quick sort	128,43	88,11	94,57				
Spanning Tree	23,07	22,64	22,13	26,13	24,08	55,81	26,09
XML	521,08	246,34	190,88				
Raw	165,14	462,28	349,88	1317,14	1882,95	2253,74	310,65