MergeArray and Scalable, Relaxed, Concurrent, Mergeable Priority Queues

by
Michael Joseph Coulombe

B.S., Computer Science
University of California at Davis, 2013

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Signature of Author: ________________________

Department of Electrical Engineering and Computer Science
May 18, 2015

Certified by: ________________________

Nir Shavit
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ________________________

Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

MergeArray and Scalable, Relaxed, Concurrent, Mergeable Priority Queues

by
Michael Joseph Coulombe

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2015, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science and Engineering

Abstract

The priority queue is a well-studied data structure which has prospered in the ever-growing field of distributed computing. However, in the asynchronous shared-memory model, one operation was left behind: merge. I present the MergeArray, a framework for implementing scalable, relaxed, concurrent, and mergeable objects, which exploits disjoint access parallelism by using an array of sequential objects and performs merges lazily, index-by-index.

I use MergeArray to build a linearizable and scalable priority queue with lock-free merge and insert and a relaxed, deadlock-free remove-min with expected worst-case rank-error of $O(p \log p)$ for p threads under common assumptions. I show experimental evidence that supports this rank-error estimate in practice as well as increased performance and scalability on a relaxed Minimum Spanning Tree benchmark compared to SprayList, a cutting-edge relaxed priority queue.

Thesis Supervisor: Nir Shavit
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to start by thanking my advisor, Nir Shavit, for providing me the amazing opportunity to do research under your guidance and surrounded by all the great people in the theory group at MIT. Most recently, thank you for inviting me to visit you and your colleagues at Tel Aviv University; I had a fun and productive time escaping the chill of Cambridge.

I must also thank my own colleagues for the wonderful intellectual and social atmosphere in the theory group, as well as in CSAIL and MIT in general. I particularly wish to thank William Leiserson, Charith Mendis, and Adam Sealfon for being great friends since day 1, and Rati Gelashvili for joining me in Tel Aviv and helping me through the thesis process.

I would like to thank Dan Gusfield for introducing me to research as an undergraduate at UC Davis, and Kristian Stevens for his collaboration with me in studying the perfect phylogeny problem. Without the research experiences in your bioinformatics group, I may never have applied to graduate school. I would also like to thank my teachers from San Leandro High School for laying strong foundations for my academic career, especially Dan Del Rosario, Victor Doan, and Rick Styner for engaging me in math, biology, and computer science.

Of course, I also thank my friends from San Leandro and UC Davis for all the fun times together, from playing made-up games as kids to playing Smash Bros, D&D, Mario Kart, and more. Notably, I'd like to thank Brian Davidson for sticking with me since second grade; I always look forward to catching up with you and everyone else when I visit home.

Last and far from least, I give enormous thanks to my parents and the rest of my family. I owe everything to the loving, supportive, and cultivating environment they provided for me and continue to.

Contents

List of Figures

List of Algorithms

1 Introduction

The *priority queue* is a well-studied data structure, consisting of a totally-ordered collection P supporting INSERT(P, e), which puts a given value e into P, and REMOVE-MIN(P), which takes the smallest contained value out of P and returns it. The *mergeable priority queue* is a common variant which supports an additional operation: MERGE(P,Q) (also called MELD or UNION), which creates a new priority queue containing $P \cup Q$.

1.1 Sequential Mergeable Priority Queues

Many theoretically- and practically-efficient sequential implementations exist for mergeable priority queues. *Leftist heaps* were among the first to support all operations in worst case $O(\log n)$ time by using a balanced binary tree structure of size n [Cra72]. *Binomial heaps* improved on this to support INSERT in amortized $O(1)$ time with a forest of heaps where a MERGE simulates binary addition [Vui78]. While array-based binary heaps cannot do better than linear time copying, a MERGE algorithm for binary heaps implemented as trees was invented which takes $O(\log n * \log k)$ time for heaps of size n and k [SS85]. *Fibonacci heaps* extended binomial heaps to support REMOVE-MIN in amortized $O(\log n)$ time and all other operations in amortized $O(1)$ time, which led to improved bounds on many graph optimization algorithms such as shortest paths and minimum spanning tree [FT87].

Because the benefits of Fibonacci heaps are primarily asymptotic, the *pairing heap* was developed as a simple-to-implement and performant alternative which has $\Theta(1)$ INSERT and MERGE and amortized $O(\log n)$ REMOVE-MIN [FSST86]. *Skew heaps* are a similarly-beneficial alternative to leftist heaps which support all operations in only amortized $O(\log n)$ time but merge faster in practice [ST86]. *Brodal queues* were the first implementation to achieve the same operation bounds as the Fibonacci heap in non-amortized worst case time in a RAM model [Bro96], and strict Fibonacci heaps achieved it in a pointer-based model using simpler means [BLT12], but neither is known to be practically efficient. Constructions have also been derived in the RAM model that grant non-mergeable priority queues over integers in

[C] the ability to perform MERGE and other operations in amortized $O(\alpha(n, n) \log \log C)$ time, where α is the inverse Ackermann function [MTZ04].

1.2 Applications

The most common application of mergeable priority queues is solving the minimum spanning tree problem, where the goal is to find a set of edges in a graph with minimum total cost which forms a tree connecting all vertices in the graph.

In the undirected case, the classic algorithms are Kruskal's, Prim's, and Sollin's algorithms, but out of the three only Sollin's takes advantage of mergeable priority queues, which has made it the starting point of parallel approaches to the problem [Zar97]. The algorithm is simple: the graph is partitioned into supernodes with associated priority queues of unprocessed edges incident to the nodes inside. Starting from singleton supernodes, each iteration removes the smallest-weight edges incident to each supernode and merges the priority queues that each edge connects. Every iteration shrinks the number of supernodes by at least half (in each connected component of the graph), thus at most logarithmically-many are needed. The set of removed edges which resulted in a merge of two supernodes is guaranteed to be a minimum spanning tree.

The minimum spanning tree problem on directed graphs has also been tackled using mergeable priority queues with Edmonds's algorithm [Edm67], which uses a similar contraction process as Sollin's algorithm but is more complex: non-standard operations are required to implement it efficiently, such as ADD(P,Δ), which adds Δ to every value in P, or MOVE(P,e,Q), which moves $e \in P$ into Q [MTZ04]. Additionally, multiple tree optimization problems have been found to be efficiently solved using mergeable priority queues [Gal80].

1.3 Asynchronous Shared Memory Model

With the widespread proliferation of multicore processors and other distributed systems, there is an increasing demand for implementing and understanding concurrent data structures. In the asynchronous shared memory model, threads perform concurrent operations by taking atomic steps which are interleaved by an adversarial scheduler, and data structure designers must balance the trade-offs of providing guarantees such as linearizability, progress, and scalability.

Linearizable operations appear to take effect at a single point during its execution, which allows for sequential reasoning about the ordering of operations when using the data structure [HW90]. Progress conditions describe the requirements under which operations complete, from deadlock-freedom (if the scheduler is fair, then someone makes progress) to lock-freedom (someone always makes progress) to wait-freedom (everyone scheduled makes progress) [HS11]. Scalability measures the efficiency of operations (ex. throughput or latency) as a function of the number of processors using the data structure. While there are many concurrent priority queue implementations, to the best of our knowledge none support linearizable, lock-free MERGE in a scalable manner.

1.4 Related Work

1.4.1 EREW PRAM Priority Queue Merge Algorithms

Efficient parallel algorithms for merging priority queues are known in the EREW PRAM model. In this model, processors cooperate synchronously to perform a single high-level operation on priority queues implemented from read-write registers that are never accessed by more than one processor at once [Zar97]. Chen designed algorithms to merge two heaps of sizes n and k in $O(\log n)$ time with $\log k$ processors, as well as merging other types of mergeable priority queues with similar bounds [Che92]. Later, Das et al. built a parallel data structure based on binomial heaps which perform all of the described operations in doubly logarithmic time and are work-optimal, as well as DELETE (which removes a given value) and thus DECREASE-KEY (which changes the ordering of a given value) with amortized guarantees, by employing $p \in \Theta(\log n / \log \log n)$ processors [CDP96].

In contrast to the EREW PRAM algorithms, scalable data structures built in the asynchronous shared memory model must support multiple concurrent high-level operations. The mutually-exclusive nature of these algorithms do not achieve the goal of a scalable, linearizable, lock-free MERGE operation.

1.4.2 Semantic Relaxation

One approach to tackling the trade-off between the strong guarantees is to relax the semantics of the object in question. Due to the inherent nature of asynchrony, this can be acceptable in applications where strict semantics cannot be taken advantage of. For example, a linearizable task queue cannot guarantee that processors complete their tasks in the same order as the corresponding DEQUEUE operations, so using a faster k-FIFO queue (where a DEQUEUE removes some value from rank 1 to k) can give better performance in practice [KLP13], as well as provide a better correspondence between operation invocation and value ordering [HKLP12], without modifying the algorithm which must already deal with unmet dependencies between tasks.

Relaxed concurrent priority queue implementations, such as the Spraylist [AKLS14] and MultiQueue [RSD14], weaken the guarantee of how small the rank is of an value REMOVE-MIN can return and subsequently demonstrate increased performance over strict implementations when measuring time per operation and time of practical algorithms using them.

1.5 Results

In this thesis, I present MergeArray, a framework on top of which I implement a concurrent, relaxed, mergeable priority queue. To meet the goals of linearizability, lock-freedom, and scalability, a MergeArray priority queue is composed of an array of sequential mergeable priority queues, such as pairing heaps. INSERT and REMOVE-MIN operations are applied by randomly choosing an open index in the array and attempting to perform the operation on the object at that index under a lock. In this way, threads do the expensive work at separate indices to exploit disjoint access parallelism.

A MERGE operation on two MergeArrays lazily combines the two arrays into one array, cell-by-cell, by having a list of pending merge operations associated with each index. In a lock-free manner, a thread which calls MERGE will, for each index, add the sequential priority queue at that index in one array to the pending merge list at the same index in the other array. To guarantee linearizability, before performing a INSERT or REMOVE-MIN operation on a sequential mergeable priority queue, a thread must fulfill each pending merge in the list at the index it chose.

To maximize the ease of use and applicability of MergeArray MERGE operations, I decided to implement aliasing semantics, where a call MERGE(P, Q) causes all references to P and Q to alias to the same merged object. Compared to alternative semantics where P takes all of the values and Q is left empty or destroyed, aliasing allows for concise implementations of algorithms such as Sollin's algorithm by encapsulating the complexity required to correctly linearize arbitrary interactions between concurrent MERGE operations.

The MergeArray mergeable priority queue implementation is fully linearizable. MERGE is always lock-free, INSERT is lock-free if the length w of the array is at least the number of threads, and REMOVE-MIN is deadlock-free. The degree of relaxation of the REMOVE-MIN operation is estimated using the rank error metric (measuring how many values in the priority queue were smaller than the value actually returned) to be expected worst-case $O(w \log w)$ and expected average-case w, which both closely models experimental results and is competitive with the aformentioned relaxed concurrent priority queues.

2 The Asynchronous Shared Memory Model

2.1 Model Definition

The asynchronous memory model consists of a set of threads (also called processes or processors), a set of shared objects, a set of "atomic" operations of those objects, and an adversarial scheduler. The computation starts after each thread has performed its initial local computation and then submits its first shared operation to the scheduler. At each step of the computation, the scheduler chooses one thread's shared operation to perform, after which that thread continues its local computations until it submits its next shared operation.

We will consider addressable shared objects with the following three atomic operations:

1. Read: "$x \leftarrow s$" takes the value of shared object s and assigns it to local variable x. When a shared object is used in an expression where a local variable would be read (ex. "**if** $s == 3$ **then** ..."), an atomic read into a temporary variable is implied. Multiple mutable shared objects will never appear in the same expression unless the ordering is defined (ex. by short-circuiting).

2. Write: "$s \leftarrow x$" unconditionally overwrites the value of the shared object s with the current value of the local variable x.

3. Compare and Swap: CAS($loc, expected, new$) writes the value of local variable new to the shared object at address loc only if the current value of $*loc$ is the same as the value of local variable $expected$, otherwise no write occurs. CAS returns **true** if the write occurred, otherwise it returns **false**.

2.2 Linearizability

When designing concurrent data structures, it is often desirable to imagine that every high-level operation, a method which may be composed of many atomic operations, is itself an atomic operation. This facilitates the use of mature sequential reasoning techniques for creating and satisfying specifications of high-level concurrent objects. This fantasy becomes reality when operations are designed to be linearizable, a correctness condition invented by Herlihy and Wing [HW90].

Definition 1. A *history* is a finite sequence of method invocations and responses, each of which is associated with a calling thread and parameter list or return value, respectively.

A history is *complete* if it only consists of invocations and their matching responses. $complete(H)$ is the maximal complete subsequence of a history H.

A history is *sequential* if every invocation (unless it is the last) is immediately followed by its matching response, and every response is immediately preceded by its matching invocation

A *sequential specification* of an object is a prefix-closed set of *legal* sequential histories which involve only methods acting on the object.

A *process subhistory* $H|P$ is the subsequence of H of the invocations and responses only made by thread P.

Histories H and H' are *equivalent* if for all threads P, $H|P = H'|P$.

A history H induces an irreflexive partial ordering where $H_r <_H H_i$ if response H_r precedes invocation H_i in H.

Definition 2. A object is *linearizable* if, for every history H containing only methods acting on the object, responses may be appended to H to make H' such that $complete(H')$ is equivalent to a legal sequential history S where $<_H \subseteq <_S$.

A powerful tool for proving that an implementation of an object is linearizable is the concept of a linearization point.

Definition 3. An *execution* is a sequence of atomic operations which can be scheduled in some computation. A history H is *compatible* with an execution E if there exists a full computation which contains H and E as subsequences.

Given a history H and compatible execution E, the *linearization point* of a method call is an atomic operation E_ℓ after the invocation H_i and before the response H_r in the full computation such that the ordering of all method calls in H by their linearization points in E induces an legal sequential history.

An object is *linearizable* if every method call acting on the object has a linearization point in all histories.

2.3 Progress Conditions

While linearizability is a useful safety property, meaning it prohibits histories which do not meet a sequential specification, it would be a trivial property to satisfy if every method could simply run without terminating. Progress conditions characterize method implementations by the situations in which invocations guarantee a response, and in the asynchronous shared memory model, the classic categories can be precisely-defined by assumptions on the scheduler [HS11]. For this discussion to make sense, we will consider infinite histories.

Definition 4. A method on an object guarantees *minimal progress* if, given a history H, every suffix of H which contains an invocation of the method but no response must contain a response to another invocation of some method acting on the same object. A method on an object guarantees *maximal progress* if, given a history H, every suffix of H which contains an invocation of the method contains a matching response. A history is *fair* if every thread makes infinitely-many method invocations with matching responses.

The difference between minimal and maximal progress is the assumption of a benevolent vs malevolent scheduler. Algorithms which guarantee maximal progress typically do so by using helping mechanisms to defend against schedules which starve some threads of progress, which can increase the complexity and run time of implementations. However, in models of real systems, operating system schedulers are sufficiently fair and benevolent for algorithms to achieve maximal progress [ACS13].

Definition 5. A method is *non-blocking* if, at any point during a history with a pending invocation, the calling thread would eventually return a matching response if it is the only thread scheduled, otherwise the method is *blocking*. A method is *dependent* if it does not guarantee minimal progress in every history, while a method is *independent* if it guarantees minimal progress in every history.

The independence property, which implies non-blocking, is important from both a theoretical and practical viewpoint. Not needing assumptions about the scheduler to guarantee some thread can make progress allows for more fundamental reasoning about computability, more general applicability of specific implementations of methods, and ideally more performance because every thread has the ability to make progress [HS11].

Definition 6. A method is *deadlock-free* if it guarantees minimal progress in every fair history and maximal progress in some fair history. A method is *lock-free* if it is independent and guarantees maximal progress in some history, and additionally *wait-free* if maximal progress is guaranteed in every history.

The goal of MergeArray is to provide or allow lock-free method implementations when possible to maximize progress, but maximizing the overall scalability comes at the concession of only guaranteeing deadlock-freedom for some operations.

2.4 Scalability and Trade-offs

Scalability describes how well a data structure performs as a function of the number of threads using it, and can be measured in various ways, such as worst-case latency per operation or total throughput. While linearizability and lock-freedom are powerful guarantees on concurrent data structure method implementations, there is a fundamental trade-off between linearizability, progress, and scalability. For example, it has been shown that linearizable implementations of certain data structures (such as counters, stacks, and queues) have inherent sequential bottlenecks [EHS12], which poses both theoretic and practical concerns about potential speed gains due to parallelism.

When analyzing mergeable data structures such as priority queues, the trade-offs illustrate why supporting a concurrent MERGE with these properties is not straightforward. As in the sequential case, performing MERGE using REMOVE-MIN and INSERT operations to move values one-by-one may be both lock-free and scalable but is far from linearizable because other processors can operate on the priority queues in illegal, intermediate states.

To guarantee linearizability, sacrifices can be made to either progress or scalability. On one extreme, wait-free universal constructions exist for single- and even multi-object systems [AM95] which could be used to provide a linearizable MERGE, but both suffer in scalability due to complex implementations and strict sequential bottlenecks that serialize every operation. More realistically, a MERGE may be performed by locking (ex. with a RW lock) the two priority queues and moving items. Not only is this far from lock-free because it blocks all other processors' concurrent reads and updates until the MERGE is done, but the blocking and the sequential bottleneck of the lock will limit the scalability, especially if it occurs frequently or is expensive to perform.

Each of these possible implementations suffer from the harsh need for MERGE to both atomically make global changes to a large, high-level object and to update two objects simultaneously. A scalable solution must allow MERGE operations to run concurrently with INSERT and REMOVE-MIN operations without interfering with their progress or correctness.

3 MergeArray

3.1 High-level Description

In light of these problems and approaches, my solution is MergeArray, a framework for building concurrent, mergeable objects. At a high-level, MergeArray consists of an array of sequential, mergeable objects that supports a linearizable, lock-free MERGE operation between two MergeArrays as well as a linearizable, conditionally lock-free APPLY-UNTIL operation to access the individual objects in a MergeArray.

The MERGE operation on two MergeArrays uses lazyness to merge the sequential objects in each array. Each index in a MergeArray has a list of pending merges into the sequential object at that index, and a MERGE operation consists of adding each object in the "source" MergeArray to the corresponding list in the "destination" MergeArray, then raising a flag to announce that every index has been added.

The APPLY-UNTIL operation applies a user-defined function F to sequential objects in the MergeArray in a fair but randomized order, where F must signal either that it was able to apply itself (thus is done) or that it needs to try another object. Because MERGE is lazy, the thread must fulfill the pending merges in the list at a chosen index before applying F on the sequential object. For each object in the list, the fulfillment process may involve helping finish the MERGE operation which put it in the list if other indices have not been added yet as well as recursively fulfilling the object's own pending merges.

3.2 Sequential Specification

A MergeArray consists of three components:

1. A Handle is a user-facing reference, which manages aliasing of Bags.
2. A Bag is an array of Nodes with a fixed size (width) and identifying number.
3. A Node holds an Element and a list of pending merges.
4. An Element is a user-defined sequential object. An Element must behave like a value,

meaning no external references are held to an Element in a bag and an Element holds no references to external mutable memory.

The interface of a MergeArray is as follows:

- Handle MAKE-HANDLE(int id, uint width, Element init)

  Returns a Handle to a new bag with identifier id holding width copies of init.

- R APPLY-UNTIL(Handle* h, Maybe(Element × R) **function**(Element) F)

  Calls F on an unspecified sequence of Elements in the Bag which h references as long as F returns NONE. Once F returns SOME(e,r), then e replaces the Element passed to F in Bag's array and r is returned. F should behave like a pure function.

- MergeResult MERGE(Handle* a, Handle* b)

  If a and b already reference the same Bag, then returns **were-already-equal** and nothing happens. If a and b reference unequal Bags with the same identifier, returns **id-clash** and nothing happens. If a and b reference Bags with different widths, then returns **incompatible** and nothing happens.

  Otherwise, after returning **success**, a and b will reference the same Bag. Other handles which previously referenced either Bag will now reference this bag as well. The new Bag has the smaller identifier of the Bags and elements which are the zip of the smaller and larger identified Bags' Elements mapped with MERGE-ELEM:

  – Element MERGE-ELEM(Element dest, Element src)

    User-defined sequential merging function. MERGE-ELEM should behave as an associative and commutative operator w.r.t. calls to APPLY-UNTIL. The lifetime of dest and src end after this call.

- void UPDATE(Handle** a)

  Attempts to optimize future accesses through *a after MERGE operations have occurred, by mutating *a to point to another Handle which references the same Bag.

3.3 Low-level Overview

There are two fundamental layers to the MergeArray between the external interface and the sequential objects. The outer layer is an inter-object disjoint set structure, made up of Handles, which manages aliasing caused by MERGE operations. By defining MERGE(P, Q) to identify P and Q as the same merged object, MergeArray is more generally useful in the concurrent asynchronous setting than other semantics of different sequential implementations. Destroying both P and Q and giving exclusive access to the result to the caller limits scalability, and it is not clear how concurrent operations should be affected. Moving all values into P and destroying only Q is unnecessarilly asymmetric and still ill-defined, and emptying Q is also asymmetric and limits sharing of internal structures. From a practical perspective, aliasing semantics allows the simple use of a MergeArray priority queue in relevant algorithms such as Sollin's for minimum spanning trees.

The inner and more complex layer of the MergeArray is a Bag's lists of pending merges into the sequential objects. The list intrusively threads through the same-index Nodes of other Bags, and supports a Node being appended once to exactly one list even when multiple processors are concurrently attempting to append it to the same list or to lists of other Bags. A MERGE operation essentially ensures that each Node of one Bag is appended to some list such that subsequently-linearized APPLY-UNTIL operations to either Bag are guaranteed to find and complete every pending merge of individual sequential objects before allowing access.

By carefully controlling and defending against the possible states and state transitions of each involved Node at each step of the algorithm, I show that a MERGE operation is able to ensure every cell is appended to some list with lock-free progress guarantees. This allows for APPLY-UNTIL operations to consume these lists by ensuring that pending merges have been linearized (by helping append the other Nodes) and then performing the pending merges between the sequential objects. As such, an APPLY-UNTIL operation requires the processor to have exclusive access to a cell's sequential object and list consumption, but when the array contains at least one cell per processor, there is always an index such that no cell at that index in any Bag of the same size is locked by any other processor, thus if the processor requests a single access to any sequential object then its APPLY-UNTIL will be lock-free.

The guarantees of these two operations, MERGE and APPLY-UNTIL, allow for MergeArray to implement a priority queue with lock-free MERGE and INSERT, whereas its deadlock-free REMOVE-MIN can block if every cell with values is locked by another processor and the rest are empty. MergeArray uses randomness to determine the order Nodes are chosen by APPLY-UNTIL operations, therefore under the assumption of a uniform allocation of values to w Elements, an uncontended REMOVE-MIN can be expected to return values up to rank $O(w \log w)$ with an expected rank of w. This is derived from the solution to the Coupon Collector's Problem [MR95], and is the same bound with similar analysis as the MultiQueue [RSD14]. w must be at least the number of processors p for INSERT to be lock-free, giving a practical bound on the expected worst-case rank-error of $O(p \log p)$ and expected average-case rank-error p.

4 Pseudocode

Algorithm 1 Struct Definitions

```
struct Handle
    Bag* bag
    Handle* next
    int id
end struct
struct Bag
    Node[ ] nodes
end struct
struct Node
    Element elem
    Handle* handle
    Node* parent
    Node* head
    Node* upper-next
    Node* upper-skip
end struct
```

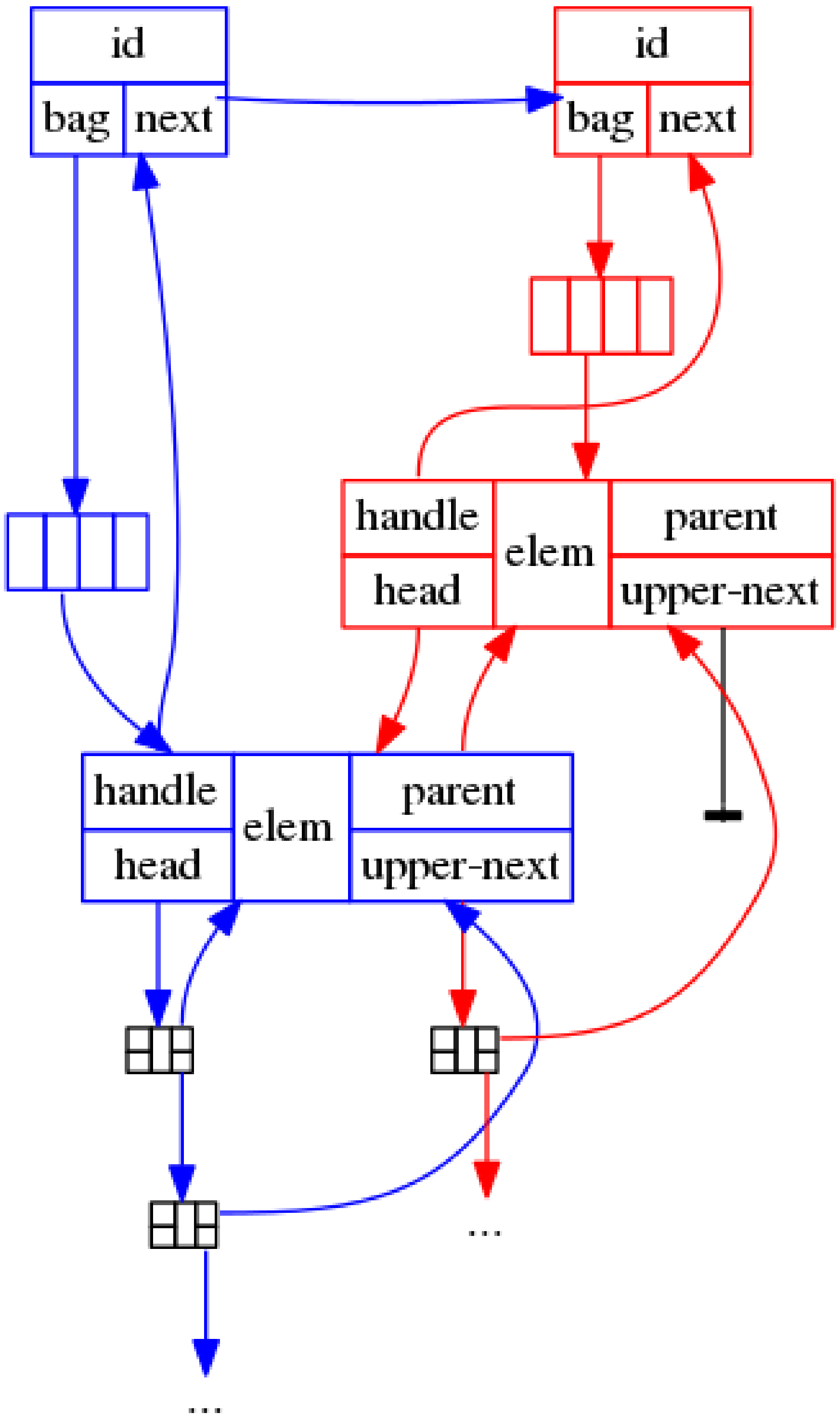


Figure 1: Two MergeArray objects (red and blue) in the process of a MERGE.

Algorithm 2 Public: Merge a and b

```
MergeResult function MERGE(Handle* a, Handle* b)
    if WIDTH(a) $\neq$ WIDTH(b) then
        return incompatible
    end if
    loop
        DESCEND-MERGE(&a)
        DESCEND-MERGE(&b)
        if a = b then
            return were-already-equal
        else if id(a) = id(b) $\wedge$ bag(a) $\neq$ null then
            return id-clash
        else if id(a) > id(b) $\wedge$ CAS(&next(a), null, b) then
            ENSURE-MERGED-INTO(a, b)
            return success
        else if id(a) < id(b) $\wedge$ CAS(&next(b), null, a) then
            ENSURE-MERGED-INTO(b, a)
            return success
        end if
    end loop
end function
void function DESCEND-MERGE(Handle** a)
    loop
        FIND-CLOSEST-BAG(a)
        next $\leftarrow$ next(a)
        if next = null then
            break
        end if
        ENSURE-MERGED-INTO(*a, next)
    end loop
end function
```

Algorithm 3 Returns nearest unmerged bag to a and compresses the path taken

```
Bag* function FIND-CLOSEST-BAG(Handle** a)
    start $\leftarrow$ *a
    bag $\leftarrow$ FIND-CLOSEST-BAG-NO-COMPRESS(a)
    while id(start) > id(a) do
        next $\leftarrow$ next(start)
        CAS-WHILE-LOWERS-ID(&next(start), a)
        start $\leftarrow$ next
    end while
    return bag
end function
```

Algorithm 4 Moves a down to the closest node with a non-**null** bag and returns the bag

```
Bag* function FIND-CLOSEST-BAG-NO-COMPRESS(Handle** a)
    loop
        bag ← bag(a)
        if bag ≠ null then
            return bag
        end if
        *a ← next(a)
    end loop
end function
```

Algorithm 5 Assigns dest to *ptr only if id(*ptr) would decrease as a result

```
void function CAS-WHILE-LOWERS-ID(Handle** ptr, Handle* dest)
    loop
        cur ← *ptr
        if id(cur) ≤ id(dest) then
            break
        else if CAS(ptr, cur, dest) then
            break
        end if
    end loop
end function
```

Algorithm 6 If a's merge into next is not linearized, then help finish the merge and commit it

```
void function ENSURE-MERGED-INTO(Handle* a, Handle* next)
    bag ← bag(a)
    if bag ≠ null then
        MERGE-PER-ELEMENT-INTO(bag, next)
        bag(a) ← null
    end if
end function
void function ENSURE-MERGED(Handle* a)
    if a ≠ null then
        next ← next(a)
        if next ≠ null then
            ENSURE-MERGED-INTO(a, next)
        end if
    end if
end function
```

Algorithm 7 Simple accessors

```
int function WIDTH(Handle* a)
    return WIDTH(FIND-CLOSEST-BAG(&a))
end function
int function WIDTH(Bag* bag)
    return length(nodes(bag))
end function
```

Algorithm 8 Helper Functions

```
bool function IS-APPENDABLE-TAIL(Node* owner, Node* cur)
    return cur $=$ nil $\vee \neg$ OWNED-BY(cur, owner)
end function
Maybe Node* function LIST-CONTINUES-AFTER(Node* owner, Node* cur)
    if $\neg$ IS-APPENDABLE-TAIL(owner, cur) then
        next $\leftarrow$ upper-next(cur)
        if next $\neq$ dummy then
            return SOME(next)
        end if
    end if
    return NONE
end function
Node* $\times$ Node* function LIST-FIND-END(Node* owner, Node* prev, Node* cur)
    skip $\leftarrow$ upper-skip(prev)
    while skip $\neq$ nil do
        further $\leftarrow$ upper-skip(skip)
        if further $=$ nil then
            cur $\leftarrow$ upper-next(prev)
            break
        else
            CAS(&upper-skip(prev), skip, further)
            prev $\leftarrow$ skip
            skip $\leftarrow$ further
        end if
    end while
    while SOME(next) $\leftarrow$ LIST-CONTINUES-AFTER(owner, cur) do
        CAS(&upper-skip(prev), nil, cur)
        (prev, cur) $\leftarrow$ (cur, next)
    end while
    return (prev, cur)
end function
```

Algorithm 9 Insert elements of src-bag into the merge lists of dest's array nodes

```
void function MERGE-PER-ELEMENT-INTO(Bag* src-bag, Handle* dest)
    indices ← TAKE(RANDOM-INDEX-RANGE(src-bag), WIDTH(src-bag))
    label indices-loop:
    while ¬ empty(indices) do
        i ← front(indices)
        dest-bag ← FIND-CLOSEST-BAG(&dest)
        dest-node ← &nodes(dest-bag)[i]
        src-node ← &nodes(src-bag)[i]
        void function TRY-INSERT(Node** loc, Node* expected)
            if IS-OWNED(src-node) then
                POP-FRONT(&indices)
                goto indices-loop
            else if CAS(loc, expected, src-node) then
                if ¬ OWNED-BY(src-node, dest-node) then
                    CAS(loc, src-node, nil)
                end if
                POP-FRONT(&indices)
                goto indices-loop
            end if
        end function
        first ← head(dest-node)
        if first ≠ null then
            if SOME(second) ← LIST-CONTINUES-AFTER(dest-node, first) then
                (last, next) ← LIST-FIND-END(dest-node, first, second)
                if IS-APPENDABLE-TAIL(dest-node, next) then
                    TRY-INSERT(&upper-next(last), next)
                    new-next ← upper-next(last)
                    if new-next ≠ dummy then continue
                end if
                if head(dest-node) ≠ first then continue
            end if
            TRY-INSERT(&head(dest-node), first)
        end if
    end while
end function
```

Algorithm 10 Merge List Ownership Management

```
bool **function** IS-OWNED(Node* src)
    p ← parent(src)
    **return** p ≠ **null**
**end function**
bool **function** OWNED-BY(Node* src, Node* dest)
    CAS(&parent(src), **null**, dest)
    p ← parent(src)
    **return** p = dest
**end function**
```

Algorithm 11 An infinite range of indices for randomly (but fairly) covering bag's node array

```
auto **function** RANDOM-INDEX-RANGE(Bag* bag)
    start ← UNIFORM-RANDOM-NAT(0, WIDTH(bag))
    **return** DROP(CYCLE(IOTA(0, WIDTH(bag))), start)
**end function**
```

Algorithm 12 Public: Moves a shared pointer down the chain to the nearest bag, bails out on concurrent update

```
void **function** UPDATE(Handle** ptr)
    a ← *ptr
    **if** a ≠ **null** **then**
        **while** bag(a) = **null** **do**
            next ← next(a)
            **if** ¬ CAS(ptr, a, next) **then**
                **break**
            **end if**
            a ← next
        **end while**
    **end if**
**end function**
```

Algorithm 13 Public: Applies a function to elements of a's closest bag until it returns **true**

```
R **function** APPLY-UNTIL(Handle* a, Maybe (Element × R) **function**(Element) F)
    **loop**
        **if** SOME(r) ← TRY-APPLY-UNTIL(FIND-CLOSEST-BAG(&a), F) **then**
            **return** r
        **end if**
    **end loop**
**end function**
```

Algorithm 14 Applies a function to bag's elements until it returns **true** or bag has been merged

```
Maybe R function TRY-APPLY-UNTIL(Bag* bag, Maybe (Element × R) function(Element) F)
    for all i ∈ RANDOM-INDEX-RANGE(src-bag) do
        node ← &nodes(bag)[i]
        try synchronized node do
            if WAS-MERGED(node) then
                return NONE
            end if
            try
                EVAL-MERGES-LOCKED(node, nil)
            catch BailoutException
                continue
            end try
            if WAS-MERGED(node) then
                return NONE
            else if SOME(e,r) ← F(elem(node)) then
                elem(node) ← e
                return SOME(r)
            end if
        end try synchronized
    end for
end function
```

Algorithm 15 Recursively evaluates the lazy, linearized merges in node's merge list. Returns if the merge list was emptied (set to new-head $\in$ {**nil**, **null**}) or throws BailoutException if a lock was contended.

```
void function EVAL-MERGES-LOCKED(Node* node, Node* new-head)
    repeat
        cur ← head(node)
        stop ← cur = new-head
        if stop then
            return
        else if LIST-CONTINUES-AFTER(node, cur) = NONE ∨ head(cur) = null then
            stop ← CAS(&head(node), cur, new-head)
            continue
        end if
        while cur ≠ nil do
            if head(cur) ≠ null then
                ENSURE-MERGED(handle(cur))
                try synchronized cur do
                    EVAL-MERGES-LOCKED(cur, null)
                    elem(node) ← MERGE-ELEM(elem(node), elem(cur))
                else
                    throw BailoutException
                end try synchronized
            end if
            label remove-cur:
            next ← upper-next(cur)
            if ¬ IS-APPENDABLE-TAIL(node, next) then
                head(node) ← next
                cur ← next
            else if CAS(&upper-next(cur), next, dummy) then
                stop ← CAS(&head(node), cur, new-head)
                break
            else
                goto remove-cur
            end if
        end while
    until ¬ stop
end function
```

Algorithm 16 Returns whether or not node's bag has been merged (linearized)

```
bool function WAS-MERGED(Node* node)
    if ¬ IS-OWNED(node) then
        return false
    else if head(node) = null then
        return true
    else
        return IS-FULLY-MERGED-LOCKED(&handle(node))
    end if
end function
```

Algorithm 17 Returns whether a is **null** or points to a DS node which has been merged (linearized) into another node

```
1:  bool function IS-FULLY-MERGED-LOCKED(Handle** ptr)
2:      a ← *ptr
3:      if a = null then
4:          return true
5:      else if bag(a) = null then
6:          *ptr ← null
7:          return true
8:      else
9:          return false
10:     end if
11: end function
```

Algorithm 18 Public: Constructor for a DS node handle

```
1: Handle* function MAKE-HANDLE(int i, uint width, Element init)
2:     a ← new Handle{ bag: null, next: null, id: i }
3:     new-node ← $\lambda j$. Node{ elem: init, handle: h, head: nil, upper-next: nil, parent: null }
4:     bag(a) ← new Bag{ nodes: [ new-node(j) | j ∈ IOTA(0, width) ] }
5:     return a
6: end function
```

Algorithm 19 Relaxed Mergeable Priority Queue Implementation, using Skew Heaps as the Elements

```
 1: PQ function EMPTY-PRIORITY-QUEUE(int id, uint width)
 2:     return PQ{ handle: MAKE-HANDLE(id, width, SKEW-HEAP-EMPTY()) }
 3: end function
 4: void function INSERT(PQ* pq, T t)
 5:     Maybe (SkewHeap × unit) function DG(SkewHeap* skew-heap)
 6:         return SOME(SKEW-HEAP-INSERT(skew-heap, t), ())
 7:     end function
 8:     a ← handle(shared-pq)
 9:     APPLY-UNTIL(a, DG)
10:     UPDATE(&handle(pq))
11: end function
12: function TRY-REMOVE-MIN(PQ* pq, int max-tries)
13:     t ← NONE
14:     tries ← 0
15:     Maybe (SkewHeap × Maybe T) function DG(SkewHeap* skew-heap)
16:         if EMPTY(skew-heap) then
17:             tries ← tries + 1
18:             return if tries = max-tries then SOME(skew-heap, NONE) else NONE
19:         else
20:             (new-heap, t) ← SKEW-HEAP-REMOVE-MIN(skew-heap)
21:             return SOME(new-heap, SOME(t))
22:         end if
23:     end function
24:     a ← handle(pq)
25:     maybe-t ← APPLY-UNTIL(a, DG)
26:     UPDATE(&handle(pq))
27:     return maybe-t
28: end function
29: MergeResult function MERGE(PQ* pq1, PQ* pq2)
30:     a ← handle(pq1)
31:     b ← handle(pq2)
32:     result ← MERGE(a, b)
33:     UPDATE(&handle(pq1))
34:     UPDATE(&handle(pq2))
35:     return result
36: end function
```

5 Proofs

5.1 States and Transitions of Struct Members

For ease of notation and discussion, I may conflate a pointer to an object with the object itself via implicit dereferencing.

Definition 7. Given Handle h, h *references* Bag b if bag(h) = b else if next(h) *references* b.

Definition 8. Given Node* ℓ and Node n, ℓ is an *empty suffix* of n's merge list if $\ell =$ **nil**, upper-next(ℓ) = **dummy**, or $\neg$ OWNED-BY(ℓ, n).

Lemma 1. *Given Node* ℓ and Node n, if* IS-APPENDABLE-TAIL*(n,ℓ) or if* LIST-CONTINUES-AFTER*(n,ℓ) returns* NONE*, then ℓ was linearized to be an empty suffix of n's merge list.*

Proof. IS-APPENDABLE-TAIL(n,ℓ) is equivalent to checking $\ell =$ **nil** $\vee$ $\neg$ OWNED-BY(ℓ,n).

If LIST-CONTINUES-AFTER(n,ℓ) returned NONE, then either IS-APPENDABLE-TAIL(n,ℓ) or upper-next(ℓ) = **dummy**, thus it was observed that either $\ell =$ **nil**, upper-next(ℓ) = **dummy**, or $\neg$ OWNED-BY(ℓ, n).

Therefore, by definition 8, both conditions imply that ℓ was linearized to be an empty suffix of n's merge list. □

Definition 9. Given Node n, n's *merge list* is the abstract sequence L where head(n) $= L_0$ if it is not an empty suffix, and inductively upper-next(L_i) $= L_{i+1}$ unless it is an empty suffix. n's *pending merges* are elem(L_i) where head(L_i) $\neq$ **null**.

5.1.1 Handle

Lemma 2. *Given Handle h, bag(h) is initialized non-**null**, is only ever assigned **null**, and is not assigned to until next(h) $\neq$ **null** and for every n $\in$ nodes(bag(h)), upper-next(n) $\neq$ **null**.*

Proof. bag(h) is initialized in MAKE-HANDLE on line 4 (due to a circular reference) to a non-**null** new Bag. bag(h) is only written to in ENSURE-MERGED-INTO on line 5, where it is assigned **null** after a call to MERGE-PER-ELEMENT-INTO. By Lemma 11, at that point every n $\in$ nodes(bag(h)) has upper-next(n) $\neq$ **null**. Additionally, ENSURE-MERGED-INTO is only called when next $\neq$ **null**, which was read from next(h). □

Lemma 3. *Given Handle h, next(h) is initially **null** and is only ever assigned handles with strictly decreasing identifiers smaller than id(h). The first non-**null** assignment persists as long as bag(h) $\neq$ **null**.*

Proof. next(h) is initialized in MAKE-HANDLE to **null**. next(h) is assigned in 2 places:

1. In MERGE on line 12, WLOG, a CAS replaces next(a) with b only if next(a) = **null** and if id(a) > id(b).

2. In CAS-WHILE-LOWERS-ID called from FIND-CLOSEST-BAG, a CAS replaces next(start) with a only if id(next(start)) > id(a). Note that this only happens if FIND-CLOSEST-BAG-NO-COMPRESS observed that bag(start) = **null**, which by lemma 2 is permanent. a was found by following next members from start by the definition of FIND-CLOSEST-BAG-NO-COMPRESS. By induction, the first assignment to a next member of any Handle is in MERGE from **null**, and CAS-WHILE-LOWERS-ID can only decrease id, thus following next must always result in observing a smaller id. Therefore, for CAS-WHILE-LOWERS-ID to modify next(start), it must be that id(start) > id(next(start)) > id(a). □

5.1.2 Bag

Lemma 4. *Given Bag b, if some n $\in$ nodes(b) has head(n) = **null**, then no handle references b.*

Proof. head(n) is only ever assigned to in MERGE-PER-ELEMENT-INTO and EVAL-MERGES-LOCKED. It can only be assigned **null** in the latter function when new-head is **null** on line 8 or line 27 because line 24 assigns the non-**null** value of next to head(n) and MERGE-PER-ELEMENT-INTO only ever assigns to it either src-node or **nil**. The possible assignments occur when the merge list is found to be logically empty in a recursive call to EVAL-MERGES-LOCKED, thus after ENSURE-MERGED(&handle(n)) has been called, so the unique handle a which held the pointer to b has had bag(a) assigned **null**, thus by definition no handle referenced b at the point of assignment to head(n). □

5.1.3 Node

Lemma 5. *Given Bag b and Node n ∈ nodes(b), elem(n) is only modified during calls to* APPLY-UNTIL *by either applying the given function or resolving lazy merges.*

Proof. elem(n) is only modified in TRY-APPLY-UNTIL after a call to F on line 16, and in EVAL-MERGES-LOCKED on line 16 to the result of MERGE-ELEM. □

Lemma 6. *Given Bag b and Node n ∈ nodes(b), head(n) is initially* ***nil****. Let head(n) be* replaceable *if it is an empty suffix or the start of a* ***dummy****-terminated merge list of n.*

If head(n) is replaceable, then any thread may assign to it such that it stays replaceable.

If head(n) is replaceable, then a merging thread may assign it a Node n′ where parent(n′) = ***null****, but head(n) will not be modified until parent(n′) is assigned.*

If head(n) is replaceable, a thread holding n's lock can assign it ***null****, and it will never change again.*

If head(n) is not replaceable, then only a thread holding n's lock can modify the state of head(n). It will not change as long as head(head(n)) ≠ ***null****, and will only become replaceable by assigning* ***dummy*** *to upper-next(head(n)).*

Proof. head(n) is initialized in MAKE-HANDLE to **nil**, an empty suffix by definition 8. In MERGE-PER-ELEMENT-INTO, head(dest-node) may be assigned to in TRY-INSERT when called on line 32. There are three ways in which flow reaches line 32 from where head(dest-node) is read into first on line 21:

1. If LIST-CONTINUES-AFTER(dest-node, first) returns NONE, then by lemma 1, this means head(dest-node) was an empty suffix of dest-node's merge list, thus it was replaceable.

2. If the insertion of src-node fails on line 26 because upper-next(last) became **dummy** (observed on line 28) even though head(dest-node) is again equal to first (observed on

line 30), then dest-node's merge list ends in **dummy** so head(dest-node) was replaceable. By lemma 7, the permanency of upper-next members implies that first will forever lead to **dummy**, so an ABA scenario does not cause a problem.

3. If IS-APPENDABLE-TAIL(dest-node, next) succeeded then head(dest-node) is again equal to first (observed on line 30), then because it was observed that LIST-CONTINUES-AFTER(dest-node, next) = NONE in the call to LIST-FIND-END, it must be that either IS-APPENDABLE-TAIL(dest-node, next) failed or upper-next(next) = **dummy**. The two conditions checked by IS-APPENDABLE-TAIL(dest-node, next) are permanent by lemma 8 thus by contradiction it must be that upper-next(next) = **dummy**, so dest-node's merge list ends in **dummy** meaning head(dest-node) was replacable.

In all cases, head(n) was observed to be replaceable and is assigned using CAS on line 13 to src-node if it has not changed. If n does not then take ownership of src-node, then head(n) is still replaceable by definition 8, so line 15 attempts to undo the assignment via CAS to **nil**, which is still an empty suffix thus still replaceable. If n does take ownership of src-node, then the current thread does nothing more to dest-node or src-node.

In EVAL-MERGES-LOCKED, while n is locked, head(n) may be assigned to in three places.

1. On line 8, it is observed that either head(n) was equal to **nil** but must become **null** or equal to some unowned node (checked in LIST-CONTINUES-AFTER) or it was a previously-removed node (which has a **null** head and thus leads to a **dummy**), so head(n) was replaceable. It cannot be **null** because it was checked to be non-**null** in the calling function: either in EVAL-MERGES-LOCKED on line 12 (while n is being removed from a merge list locked by this thread) or in TRY-APPLY-UNTIL on line 5 (while n is locked to make sure n was not already merged before applying the function). By assigning **nil** with a

2. On line 24, it was observed that head(n) = cur and upper-next(cur) was an owned node continuing the merge list. This guarantees that this value of upper-next(cur) is permanent and that no other thread has modified head(n) since n took ownership, thus

since head(cur) = **null**, cur's merge is no longer pending, so head(n) can be assigned next to safely remove cur from the merge list.

3. On line 27, it is observed that upper-next(cur) = **dummy**. Because cur was head(n), it was replaceable. If the CAS succeeds and new-head was **null**, then this thread has succeeded at clearing the list and marking n for removal from its owner's merge list. □

Lemma 7. *Given Node n, upper-next(n) is initially* ***nil****.*

If upper-next(n) is an empty suffix, then a merging thread may assign to it ***nil*** *or a Node n′ where parent(n′) =* ***null****, but upper-next(n) will not be modified until parent(n′) is assigned.*

If n is the only member of parent(n)'s merge list, then a thread locking parent(n) may assign ***null*** *to head(n) then assign* ***dummy*** *permanently to upper-next(n).*

If upper-next(n) is not an empty suffix, then it will never change again.

Proof. Given Node n, upper-next(n) is initialized in MAKE-HANDLE to **nil**. It may be assigned to in MERGE-PER-ELEMENT-INTO in TRY-INSERT on lines 13 and 15 if loc is &upper-next(n) (where, like head(n), it may be assigned src-node and reassigned if the node is not owned). In EVAL-MERGES-LOCKED, upper-next(n) may be assigned **dummy** on line 26 when it is observed that n = cur = head(node), parent(n) = node, and upper-next(n) was an empty suffix. □

Lemma 8. *Given Node n, parent(n) is initially* ***null****, and is modified exactly once such that n instantly joins the new parent(n)'s merge list.*

Proof. parent(n) is initialized in MAKE-HANDLE to **null**. The use of parent(n) is encapsulated in IS-OWNED and OWNED-BY, where it can only ever be assigned a new value exactly once, via CAS from **null** to a non-**null** node. In MERGE-PER-ELEMENT-INTO, IS-OWNED is called on line 10 to guard against double insertion, and OWNED-BY is called on line 14 (owned by dest-node after src-node is inserted into its merge list) and within lines 23, 24, and 25 (node found in a merge list before another inserting thread gave it a parent). In EVAL-MERGES-LOCKED, OWNED-BY is called within lines 7 and 23 (set to node if cur or next is found in the list without a parent). In WAS-MERGED, IS-OWNED is called because by lemma 11, if it returns **false** then there was no merge linearized before the call. □

Lemma 9. *Given Bag b and Node n ∈ nodes(b), handle(n) is a back-pointer to Handle h where bag(h) was initially b. As an optimization to limit indirections, handle(n) may be permanently assigned* ***null*** *after bag(h) becomes* ***null****.*

Proof. handle(n) is initialized in MAKE-HANDLE to a, which gets bag(a) assigned b. handle(n) is used in EVAL-MERGES-LOCKED on line 13 (where n is cur) as an argument to ENSURE-MERGED because elem(cur) can only be merged with elem(node) if the merge of cur's bag into node's bag has been linearized, thus bag(handle(n)) must have been **null** before the end of the call to ENSURE-MERGED. In IS-FULLY-MERGED-LOCKED, handle(n) is set to **null** only after bag(handle(n)) is read to be **null**. By lemma 2, the latter was a permanent state. IS-FULLY-MERGED-LOCKED is only used in WAS-MERGED which is only used in TRY-APPLY-UNTIL such that if handle(n) was assigned **null** then WAS-MERGED returns **true**, guarding node from being used. □

5.2 Linearizability

Lemma 10. FIND-CLOSEST-BAG *is linearizable to finding the bag which the given handle references.*

Proof. FIND-CLOSEST-BAG-NO-COMPRESS(&h) is a direct algorithmic translation of definition 7. If next(h) was followed, then it must be that bag(h) = **null**, thus h and next(h) reference the same bag. Inductively, the first handle must have referenced the returned bag belonging to the final handle at the linearization point of call when the bag was read non-**null**.

The rest of the work in FIND-CLOSEST-BAG also preserves the reference because in each iteration where id(start)) > id(a), it must have been that bag(start) = **null**, so start and next(start) both refer to the same bag. □

Lemma 11. *After a call* MERGE-PER-ELEMENT-INTO*(src-bag, dest) returns, then every src-node* $\in$ *nodes(src-bag) has a non-**null** parent(src-node) such that handle(parent(src-node)) references the same bag as dest.*

Proof. MERGE-PER-ELEMENT-INTO only returns when it iterates over every member of indices, which covers the indexes of every src-node $\in$ nodes(src-bag) exactly once. Calling POP-FRONT(&indices) advances the thread to the next src-node, and is only done in TRY-INSERT after either IS-OWNED(src-node) returns **true** or OWNED-BY(src-node, dest-node) is called, thus after src-node is owned by some dest-node.

MERGE-PER-ELEMENT-INTO is only called in ENSURE-MERGED-INTO(a, next), where next was equal to next(a) and was passed as parameter dest. Because bag(a) $\neq$ **null** was checked after next was read, by lemma 3, every thread which called ENSURE-MERGED-INTO(a, next) and entered MERGE-PER-ELEMENT-INTO must have had the same value of next thus the same initial value of dest. Because dest is only modified in MERGE-PER-ELEMENT-INTO on line 6 by calls to FIND-CLOSEST-BAG(&dest), dest always referenced the same bag as next.

For every src-node $\in$ nodes(src-bag), after a src-node is appended, parent(src-node) becomes equal to the dest-node of some merging thread because OWNED-BY(src-node, n) is only called when n is the owner of the merge list which src-node was appended to and is always called by every merging thread unless it otherwise determines that parent(src-node) $\neq$ **null** already. This means that handle(parent(src-node)) held the initial value of dest of that merging thread, which references the same value as next, thus references the same bag as the dest of all merging threads. □

Lemma 12. *During* TRY-APPLY-UNTIL*, if* EVAL-MERGES-LOCKED*(node, **nil**) returns without throwing and then* WAS-MERGED*(node) returns **false**, then there is a linearization point such that elem(node) holds the correct value with respect to all previously-linearized calls to* APPLY-UNTIL *and* MERGE*.*

Proof. elem(node) held the correct linearized value when it was initialized and node is locked so there are no concurrent APPLY-UNTIL operations affecting node, so consider inductively the previous linearization point of elem(node) and every correspondingly-indexed elem(child) which must be merged with elem(node).

If parent(child) = node, then child was appended to node's merge list by a merging thread. Because EVAL-MERGES-LOCKED(node, **nil**) returned without throwing, node's merge list is empty at the end of the call, thus either this call or a partial, failed previous call must have removed child after fulfilling its pending merge on line 16 after a call to EVAL-MERGES-LOCKED(child, **null**).

If parent(child) $\neq$ node, then child was appended to another merge list. By lemma 11, handle(parent(child)) references the same bag as handle(node) and id(handle(child)) $>$ id(handle(parent(child))) $>$ id(handle(node)), so elem(parent(child)) also needs to be merged with elem(node). By induction, the pending merge of elem(parent(child)) must have been fulfilled, so a previous call to EVAL-MERGES-LOCKED(parent(child), new-head) must have removed child that merge list, thus transitively fulfilling the pending merge of elem(child). □

Theorem 1. MERGE *is linearizable to its sequential specification.*

Proof.

- Suppose MERGE(a,b) returns **incompatible**. Every call to MERGE is guarded by the equal-width check, and the width of a bag is a constant, thus the width of a handle is a constant and it is not necessary to syncronize the two WIDTH calls to meet the specification.

- Suppose MERGE(a,b) returns **were-already-equal**. This means that after calling DESCEND-MERGE on a then b, both of them descended via FIND-CLOSEST-BAG to the same Handle, so the arguments both reference the same bag on return by lemma 10.

- Suppose MERGE(a,b) returns **id-clash**. This means that after calling DESCEND-MERGE on a then b, both of them descended to different Handles with the same identifier. Because these operations were not synchronized, a may reference a different bag by the time the id is checked, thus if bag(a) is not **null** afterwards then by lemma 2 it was not

null during the linearization point of the second call, thus a and b did both reference the bags which they directly point to.

- Suppose MERGE(a,b) returns **success**. WLOG assume the id(a) > id(b) branch was taken. Because the CAS must have been successful, this means that between the linearization point of the last DESCEND-MERGE(a) and the CAS, no other concurrent MERGE was performed on a, therefore this MERGE is the next to be linearized.

  ENSURE-MERGED-INTO will call MERGE-PER-ELEMENT-INTO on bag(a) given that another thread hasn't already linearized the merge by assigning it **null**. By Lemma 11 and Theorem 2, once the call returns and MERGE can be linearized (if it hasn't already been) by assigning bag(a) the value **null**, every bag node will have been inserted into a merge list which is available to be merged in a call to EVAL-MERGES-LOCKED linearized afterwards. □

Theorem 2. APPLY-UNTIL*(a,*F*) is linearizable to its sequential specification if* F *is a pure function.*

Proof. In a loop, APPLY-UNTIL(a,F) gets bag from FIND-CLOSEST-BAG(a) then calls TRY-APPLY-UNTIL(bag,F) until it returns **true**. By Lemma 10, bag is the Bag that a referenced at the linearization point of the call.

TRY-APPLY-UNTIL iterates over every node ∈ nodes(bag) infinitely often by the specification of RANDOM-INDEX-RANGE until the body returns. For each chosen Node, TRY-APPLY-UNTIL will apply F to elem(node) only if, while its lock is taken, EVAL-MERGES-LOCKED(node, **nil**) is successful (returns without throwing a BailoutException) then WAS-MERGED(node) returns **false**.

By Lemma 12, if EVAL-MERGES-LOCKED throws BailoutException, then elem(node) may not be in a linearizable state, but because all accesses to elem(node) must come after a successful call to EVAL-MERGES-LOCKED, bailing out causes no damage. However, if the call is successful, then elem(node) reflects all operations linearized before the call.

WAS-MERGED(node) is used to check that a still references bag. This can only fail if bag(a) has been set to **null** due to a concurrent MERGE, which monotonically changes the return value from **false** to **true**. If the check after the call to EVAL-MERGES-LOCKED returns **false**, then it would have returned **false** at its linearization point, otherwise the operation must retry because bag is no longer referenced by a.

Lastly, if F(elem(node)) returns NONE, then aborting to find another element meets the specification and does not break linearization because F is pure so it did not modify elem(node) or cause side-effects. If F returns SOME(e,r), then the linearization point of APPLY-UNTIL is the linearization point of the last call to EVAL-MERGES-LOCKED. This meets the specification because F was applied to elem(node) in the state it was during the linearization point of EVAL-MERGES-LOCKED when a still referenced bag and all linearized operations updating elem(node) have done so. □

Theorem 3. UPDATE *is linearizable to its sequential specification.*

Proof. UPDATE takes a pointer to a handle a which is assumed to be modifiable by other threads in all possible ways. The handle is only modified by UPDATE when it is observed that a $\neq$ **null** and bag(a) $=$ **null**, thus that a does not reference a bag it is holding. By definition 7, next(a) references the same bag as a does, so replacing a with next(a) preserves the semantics of the other public functions. □

5.3 Progress

Lemma 13. FIND-CLOSEST-BAG *is lock-free.*

Proof. First, FIND-CLOSEST-BAG calls FIND-CLOSEST-BAG-NO-COMPRESS, which is lock-free because it will return unless every handle it traverses has a **null** bag and a non-**null** next, but by Lemma 3 there cannot be cycles, so other threads would have to be infinitely often creating and merging new bags, thus making progress.

Second, there is a traversal loop of start's descendants from id(start) down to id(a) where each iteration calls CAS-WHILE-LOWERS-ID which is lock-free also by Lemma 3. □

Theorem 4. APPLY-UNTIL *is lock-free if*

1. *width is at least the number of threads,*

2. F *is a pure function,*

3. F *will return* SOME*(e,r) after a finite number of applications,*

4. *and the locks of Node objects have lock-free try-lock and unlock methods.*

Proof. Inside APPLY-UNTIL, TRY-APPLY-UNTIL only runs forever if each node $\in$ nodes(bag) is observed to be locked infinitely often, if EVAL-MERGES-LOCKED throws BailoutException infinitely often, or if EVAL-MERGES-LOCKED runs forever.

EVAL-MERGES-LOCKED loops until stop = **true**, which happens when head(node) is observed to be or is assigned new-head. It will only run forever in the following cases, which all require that other threads make infinite progress, thus it is lock-free:

- Other threads in MERGE-PER-ELEMENT-INTO infinitely often succeed at appending owned nodes to the merge list, so this thread is never able to exhaust the list in the while loop.

- Other threads in MERGE-PER-ELEMENT-INTO infinitely often succeed at appending unowned nodes then undoing their action, so this thread reads the end of the list as the unowned node between the append and undo but executes the CAS operation on lines 8 or 27 afterwards thus infinitely often fails.

- ENSURE-MERGED(handle(cur)) runs forever. By Lemma 14, it is lock-free.

- EVAL-MERGES-LOCKED(cur, **null**) runs forever. If the recursion runs infinitely deep then infinitely-many MERGE calls must be succeeding, otherwise one of these cases must be occurring in a child node, so by induction this is lock-free.

Because width is at least the number of threads, there always exists a node where it and all of the child nodes below it are not locked or otherwise occupied by other threads. The thread moves to a new node exactly when it completes an iteration of the for loop in TRY-APPLY-UNTIL, thus when a lock (belonging to node or any child below it) is found to be held by another thread.

Therefore, for a thread to infinitely fail to grab locks over every node, another threads must be infinitely often succeeding to get those locks over every node. While recursing in EVAL-MERGES-LOCKED, the locks are only ever taken in increasing handle-identifier order, so there can be no dependency cycles within or between threads, thus some thread must succeed at every try-lock it attempts and return without throwing a BailoutException.

Because TRY-APPLY-UNTIL is thus lock-free, and by Lemma 13 FIND-CLOSE-BAG is lock-free, APPLY-UNTIL can only run forever if TRY-APPLY-UNTIL only ever returns **false**. This only happens if WAS-MERGED(node) or F returns **false** infinitely often, but the latter will not happen by assumption and the former requires infinitely-many concurrent MERGE operations, therefore APPLY-UNTIL is lock-free. □

Lemma 14. ENSURE-MERGED-INTO*(a, next) is lock-free, given that next was next(a).*

Proof. If bag(a) is already **null**, then ENSURE-MERGED-INTO returns immediately, otherwise it only runs forever if MERGE-PER-ELEMENT-INTO(bag, next) does.

MERGE-PER-ELEMENT-INTO iterates over every src-node $\in$ nodes(bag) once, only progressing when POP-FRONT(&indices) is called in TRY-INSERT. Thus, it can only run forever if some src-node never joins any merge list because even if only the current thread always fails to append on line 13, by lemma 8, IS-OWNED(src-node) returns **true** after any thread has succeeded to append it on line 13 and then take ownership of it.

The CAS on line 13 fails when another thread succeeds at modifying loc first. If TRY-INSERT was called on line 26, then loc points to upper-next(prev) and was observed to be **nil** or an unowned node. If TRY-INSERT was called on line 32, then loc points to head(dest-node) and was observed to be an empty suffix. If other threads infinitely often appended a src-node in MERGE-PER-ELEMENT-INTO, then all threads trying to insert those src-nodes will infinitely often POP-FRONT and advance. If other threads infinitely often replace an unowned or dummy node with **nil**, **null**, or **dummy** in EVAL-MERGES-LOCKED, then there must be infinitely-many nodes being made dummy nodes, being successfully inserted without getting ownership, or being doubly-inserted after a remove (this ABA scenario can only happen once per node per thread so it can only finitely delay progress).

If a thread fails to append because it never calls TRY-INSERT, then it must either be stuck in an infinite recursion in LIST-FIND-END (thus the list is appended to infinitely many times so this thread never reaches the end), stuck in the lock-free FIND-CLOSEST-BAG call on line 6, or be executing infinite iterations of indices-loop, the while loop of MERGE-PER-ELEMENT-INTO.

indices-loop only repeats infinitely without calling TRY-INSERT if head(dest-node) is infinitely-often observed to be **null** (thus infinitely-many MERGE operations are succeeding because it was not **null** before the call to FIND-CLOSEST-BAG) or if IS-APPENDABLE-TAIL always fails and head(dest-node) $\neq$ first (thus infinitely-many nodes are being successfully appended to the merge list by other merging threads and then made dummy nodes in EVAL-MERGES-LOCKED). □

Theorem 5. MERGE *is lock-free.*

Proof. MERGE(a,b) can only run forever if WIDTH runs forever on line 2, if DESCEND-MERGE on line 6 (WLOG) runs forever, if ENSURE-MERGED-INTO on line 13 (WLOG) runs forever, if infinitely-often id(a) = id(b) when the condition on line 10 fails, or if the CAS on line 12 (WLOG) always fails.

By 13, WIDTH is lock-free.

By lemma 13 and 14, DESCEND-MERGE is lock-free because it can only take infinitely-many iterations if infinitely-many other MERGE operations succeed.

By lemma 14, ENSURE-MERGED-INTO(a, b) is lock-free because it is always called when b was equal to next(a).

If it is observed that a $\neq$ b and id(a) = id(b) but bag(a) = **null**, then because the call to DESCEND-MERGE(&a) linearized a to be a Handle with bag(a) $\neq$ **null**, some merging thread must be terminating a successful MERGE call because by lemma 2 bag(a) is only assigned **null** in ENSURE-MERGED-INTO after MERGE-PER-ELEMENT-INTO succeeds. Therefore, if MERGE runs forever due to infinitely-often observing this, then infinitely-many other MERGE operations are succeeding.

If the CAS fails infinitely-many times, then infinitely-many other MERGE operations must be succeeding at the CAS and terminating. □

Theorem 6. UPDATE *is lock-free.*

Proof. UPDATE can only run forever if bag(a) = **null** is observed in every iteration of the loop and the CAS(ptr, a, next) always succeeds. By lemma 2, this implies that infinitely-many MERGE operations must be completing. □

5.4 Priority Queue

Theorem 7. MERGE*(pq1, pq2) is linearizable and lock-free.*

Proof. The PQ MERGE is a thin wrapper around the MergeArray MERGE, which by theorems 1 and 5 is linearizable and lock-free. The calls to UPDATE do not cause any semantic effect because it is also linearizable and lock-free by theorems 3 and 6. □

Theorem 8. INSERT*(pq, t) is linearizable and lock-free, given that pq has sufficient width and lock-free lock methods.*

Proof. INSERT calls APPLY-UNTIL on a local closure DG which just inserts the value t into the SkewHeap. Because DG is pure and never returns NONE, it is linearizable by theorem 2 and lock-free by theorem 4. □

Theorem 9. TRY-REMOVE-MIN*(pq, max-tries) is lock-free, and if max-tries is finite and positive then it is linearizable to one successful attempt at removing a value if it returns* SOME*(e) and linearizable to one unsuccessful attempt if it returns* NONE*, given that pq has sufficient width and lock-free lock methods.*

Proof. TRY-REMOVE-MIN calls APPLY-UNTIL on a local closure DG which attempts to remove a value from the SkewHeap.

DG is not pure, but only because it mutates tries, a local counter. Given positive max-tries, The only impure information which escapes TRY-REMOVE-MIN is whether or not the counter hit max-tries, which is exactly whether or not the return value is SOME(e) or NONE.

If it returns SOME(e), then e was removed from some SkewHeap. No information about other indices escapes TRY-REMOVE-MIN, thus it is linearizable to the final, successful application of DG being the only one.

If it returns NONE, then no modifications to any SkewHeaps were performed, and the only information that is leaked is that at least one index has an empty SkewHeap, so it is linearizable to the to the final, unsuccessful application of DG being the only one.

Thus, while DG is not pure, it is safely encapsulated within TRY-REMOVE-MIN and DG returns SOME(e) after finitely-many applications, so it is linearizable by a similar argument to theorem 2 and lock-free by a similar argument to theorem 4. □

Definition 10. The i^{th} harmonic number is $H(i) = \sum\limits_{j=1}^{i} \frac{1}{j} = O(\log i)$.

Lemma 15. *For any positive integer n, $\sum\limits_{i=1}^{n-1} H(i) = nH(n) - n$.*

Proof. Base Case:

$$\sum_{i=1}^{0} H(i) = 1 \times H(1) - 1 = 1 \times 1 - 1 = 0 \checkmark$$

Inductive Hypothesis:

$$\begin{aligned}
\sum_{i=1}^{k-1} H(i) &= kH(k) - k \\
\sum_{i=1}^{k} H(i) &= H(k) + \sum_{i=1}^{k-1} H(i) \\
&= H(k) + kH(k) - k \\
&= (k+1)H(k) - k \\
&= (k+1)H(k) + 1 - (k+1) \\
&= (k+1)(H(k) + \frac{1}{k+1}) - (k+1) \\
&= (k+1)H(k+1) - (k+1) \checkmark
\end{aligned}$$

□

Definition 11. Given a set P, the *rank error* of a value $v \in P$ is its index in the ordered sequence of values in P.

Theorem 10. *After x solo-executions of* INSERT*(pq,t), starting from an empty PQ with width w and as x approaches infinity, a solo-execution of* TRY-REMOVE-MIN*(pq,w) approaches expected worst-case $O(w \log w)$ and average-case w rank error.*

Proof. RANDOM-INDEX-RANGE returns a "fair" infinite range r of indices over the interval $[0, w)$, where FRONT(r) $\sim U[0, w-1]$ and for every index $i \in [0, w)$ and $n \in \mathbb{N}$, COUNT(TAKE(r, $w \times n$), i) = n.

This implies that an execution of INSERT(pq,t) without contention will insert t at a uniformly random index of *pq*. As x approaches infinity, the probability that some index is never chosen approaches zero, so the result of an execution of TRY-REMOVE-MIN(pq,w) without contention approaches a uniformly random choice of the minimum element at each index.

The rank error of the returned value is its position in the sequence of all values at every index in sorted order. Because each INSERT samples independently and the minimum value of a SkewHeap is independent of insertion ordering, WLOG assume the values are globally inserted in order. The expected worst-case rank error is thus the expected number of INSERT operations until every index has been chosen. Similarly, the expected average-case rank error becomes the expected number of operations before any particular index was chosen for the first time.

This is a direct reduction to the Coupon Collectors Problem [MR95]. Let random variable R be a sorted array of the minimum rank at each index. Selection is uniform, so if i indices have been chosen at least once, the probability of choosing a previously-unchosen index is $\frac{w-i}{w}$, thus the waiting time follows a geometric distribution with mean $\mathbb{E}(R[i]-R[i-1]) = \frac{w}{w-i}$.

$$\begin{aligned}
\mathbb{E}R[0] &= 1 \\
\mathbb{E}R[i] &= \sum_{j=0}^{i} \frac{w}{w-j} \\
&= w \sum_{j=0}^{i} \frac{1}{w-j} \\
&= w \sum_{j=w-i}^{w} \frac{1}{j} \\
&= w \sum_{j=1}^{w} \frac{1}{j} - w \sum_{j=1}^{w-i-1} \frac{1}{j} \\
&= wH(w) - wH(w-i-1)
\end{aligned}$$

Using this equality, the expected worst-case rank error becomes:

$$\mathbb{E}R[w-1] = wH(w) - wH(w-(w-1)-1) = wH(w) = O(w \log w)$$

The expected average rank error is:

$$\begin{aligned}
\mathbb{E}\left(\frac{1}{w}\sum_{i=0}^{w-1} R[i]\right) &= \frac{1}{w}\sum_{i=0}^{w-1} \mathbb{E}R[i] \\
&= \frac{1}{w}\sum_{i=0}^{w-1}(wH(w) - wH(w-i-1)) \\
&= \sum_{i=0}^{w-1}(H(w) - H(w-i-1)) \\
&= \sum_{i=0}^{w-1} H(w) - \sum_{i=0}^{w-1} H(w-i-1) \\
&= wH(w) - \sum_{i=1}^{w-1} H(i) \\
&= wH(w) - (wH(w) - w) \\
&= w
\end{aligned}$$

□

6 Experiments

6.1 Implementation Details

I wrote MergeArray and all benchmarks in the D programming language, and compiled with the gcc-backend compiler, gdc (D 2.066.1 and gcc version 4.9.2). Spraylist [AKLS14] was written in C, also compiled with gcc version 4.9.2, and natively linked to the D benchmarks. Allocations by the data structures were done in large thread-local buffers to avoid contention within the allocator and to disable garbage collection during benchmarks.

All code was compiled for x86_64 Linux and run on a Fujitsu PRIMERGY® RX600 S6 server. The server has four Intel® Xeon® E7-4870 processors each with ten cores running at 2.40 GHz and with two-way hyperthreading.

6.2 Relaxed Minimum Spanning Tree

As described earlier in this thesis, the Minimum Spanning Tree problem is a promenant application of mergeable priority queues. This benchmark measures a parallel Sollin's Algorithm implemention using MergeArray at various width settings (Algorithm 20) and compares it to a parallel Kruskal's Algorithm implemented using SprayList (Algorithm 21) as well as a non-parallel Kruskal's Algorithm implemented using an array-backed binary heap.

Figure 2 shows the runtime comparison on a randomly-generated dense graph with 20,000 nodes and 79,986,146 edges (uniform 20% chance of inclusion). The parallelism of Sollin's Algorithm, made efficiently possible by MergeArray's MERGE, allows it to greatly out-perform both Kruskal's Algorithm implementations which use a centralized priority queue. While MergeArray only guarantees lock-free INSERT when the width w is at least the number of threads p, the experiments using smaller widths still show comparable runtime to the $w = p$ case, except for $w = 1$ where each MergeArray permits no concurrent INSERT or REMOVE-MIN operations.

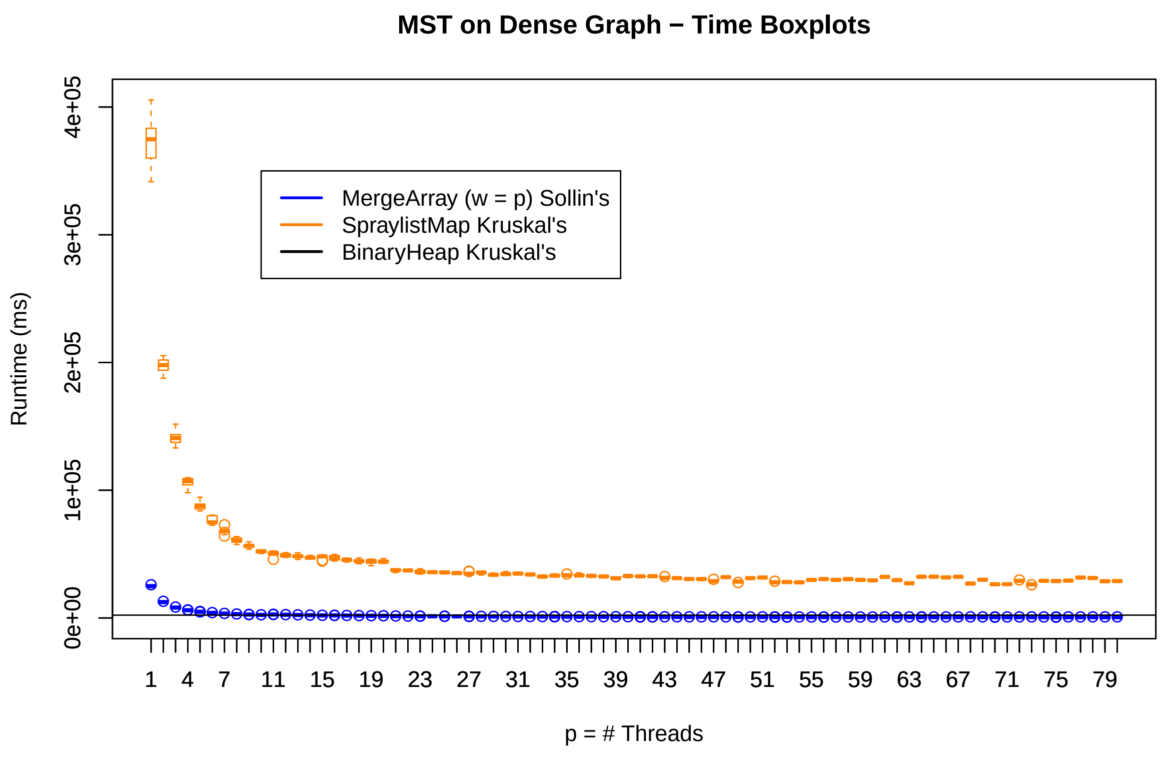
The scalability of the implementations is demonstrated by their speedup, shown in Figure 3. Comparing the ratio of the single thread runtime of an algorithm to the runtimes when

Algorithm 20 Sollin's Algorithm for Undirected Relaxed MST using MergeArray

```
Edge[] function SOLLIN-MERGEARRAY(Edge[][] adj, int num-threads)
    S ← new PQ[](length(adj))
    for (u, edges) ∈ adj in parallel do
        S[u] ← EMPTY-PRIORITY-QUEUE(u, num-threads / 4 + 1)
        for all e ∈ edges do
            INSERT(&S[u], e)
        end for
    end for
    T ← Ø
    for tid ∈ IOTA(0, num-threads) in parallel do
        while length(T) < length(adj)−1 do
            u ← UNIFORM-RANDOM-NAT(0, length(adj))
            if SOME(e) ← TRY-REMOVE-MIN(&S[u]) then
                (weight, u, v) ← e
                if MERGE(&S[u], &S[v]) ≠ were-already-equal then
                    T ← T ∪ {e}
                end if
            end if
        end while
    end for
    return T
end function
```

Algorithm 21 Kruskal's Algorithm for Undirected Relaxed MST using Spraylist

```
Edge[] function KRUSKAL-SPRAYLIST(Edge[] edges, int num-nodes, int num-threads)
    queue ← MAKE-SPRAYLIST($2 + \log_2$(length(edges)))
    for (i,e) ∈ edges in parallel do
        (weight, u, v) ← e
        if u ≤ v then
            INSERT(&queue, weight, i)
        end if
    end for
    set ← [ i : SDSNode{null, i} | i ∈ IOTA(0, num-nodes) ]
    T ← Ø
    for tid ∈ IOTA(0, num-threads) in parallel do
        while length(T) < length(adj)−1 do
            if SOME(weight, i) ← REMOVE-SPRAY(&queue) then
                (weight, u, v) ← edges[i]
                if SDS-MERGE(set[u], set[v]) then
                    T ← T ∪ {e}
                end if
            end if
        end while
    end for
    return T
end function
```



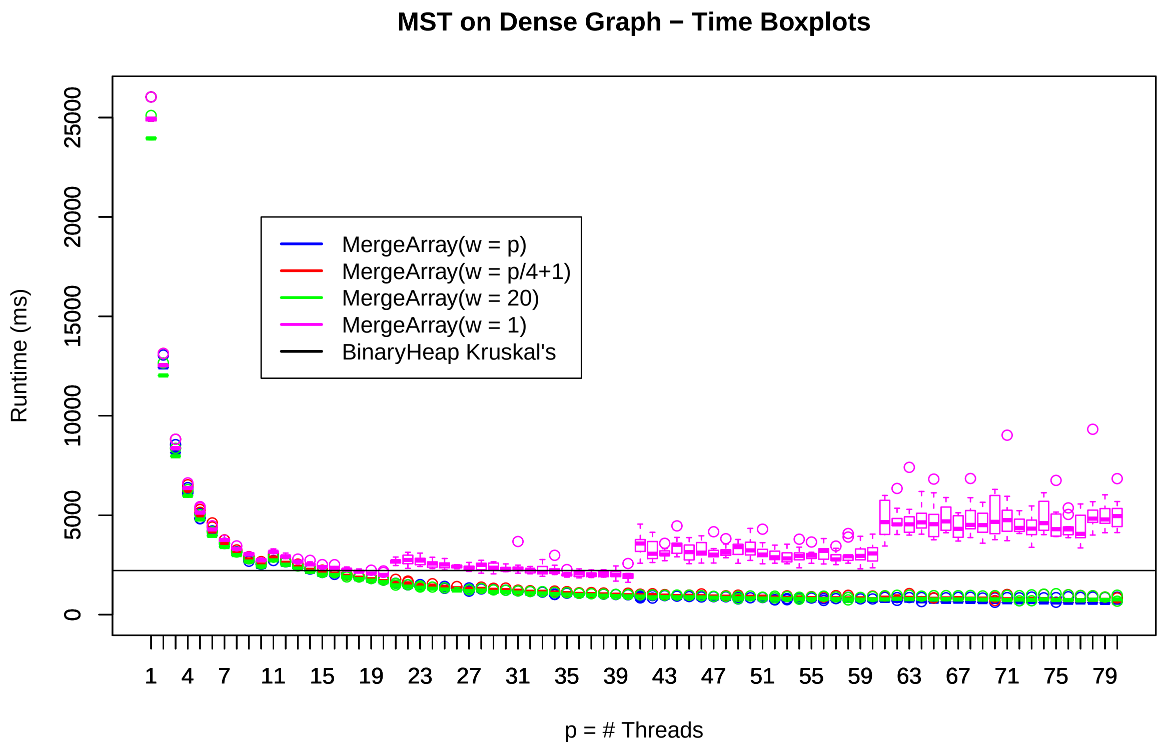


Figure 2: MST Benchmark Runtimes on dense random graph

Algorithm 22 Concurrent Disjoint Set [SB08]

```
struct SDSNode
    SDSNode* next
    int id
end struct
SDSNode* function FIND(SDSNode* a)
    old-next ← next(a)
    if old-next = null then return a
    rep ← FIND(old-next)
    if old-next ≠ rep then CAS(&next, oldnext, rep)
    return rep
end function
bool function LINK-WITH(SDSNode* a, SDSNode* b)
    if id(a) > id(b) ∧ CAS(&next(a), null, b) then return true
    if id(a) < id(b) ∧ CAS(&next(b), null, a) then return true
    return false
end function
bool function SDS-MERGE(SDSNode* a, SDSNode* b)
    loop
        a-rep ← FIND(a)
        b-rep ← FIND(b)
        if id(a-rep) = id(b-rep) then return false
        if LINK-WITH(a-rep, b-rep) then return true
    end loop
end function
```

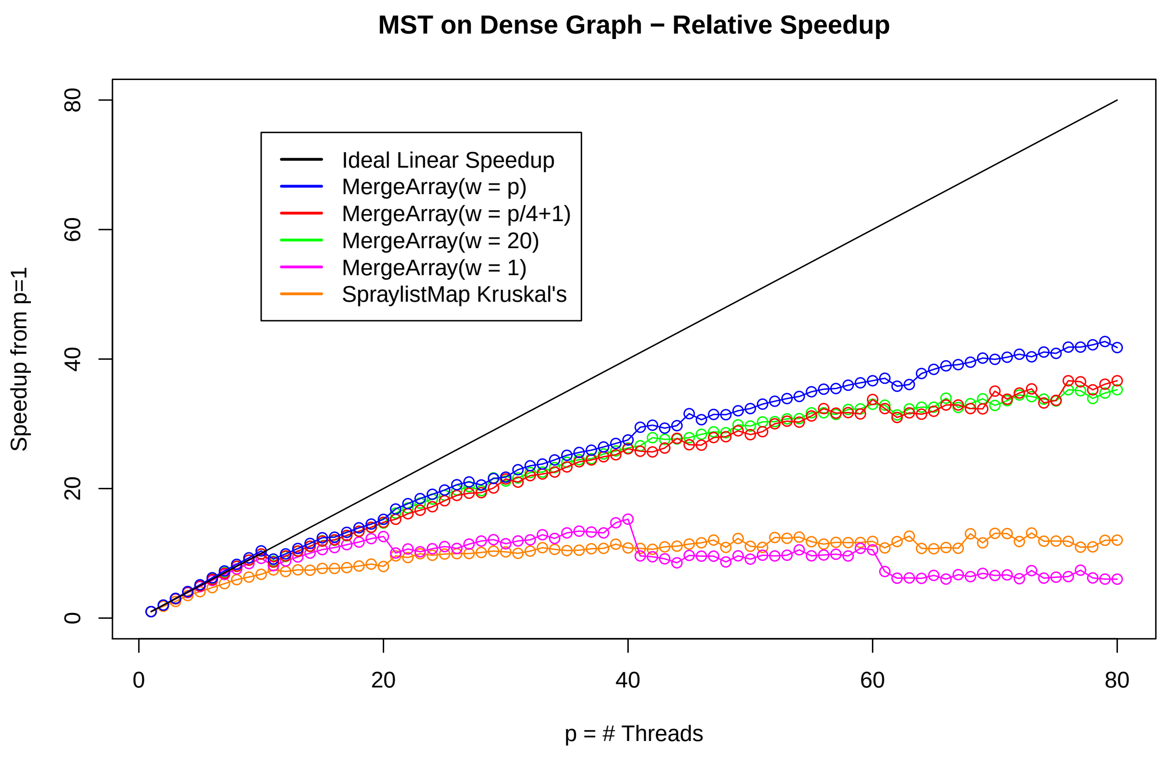


Figure 3: MST Benchmark Relative Speedups on dense random graph

using higher numbers of threads, MergeArray with $w = p$ shows the highest scalability, and all MergeArray experiments (except $w = 1$) scale up to 80 threads. The Spraylist flatlines after about $p = 20$, but because most of the runtime is spent doing the $O(|E|)$ INSERT operations (implemented as a normal skiplist INSERT) rather than the $O(|V|)$ REMOVE-MIN operations (implemented as a relaxed spray), this is not the fault of the innovation of its creators.

It should be noted that the architecture of the server has left an imprint in the runtime patterns. While each MergeArray experiment scales near-perfectly up to $p = 10$, there is a drop afterwards when hyperthreading kicks in because for $p \leq 20$ each thread is run on a single processor. At $p = 21$, $p = 41$, and $p = 61$, a additional processor is used, which causes some performance loss due to NUMA (non-uniform memory access) between sockets; this is especially evident in the MergeArray experiement when $w = 1$.

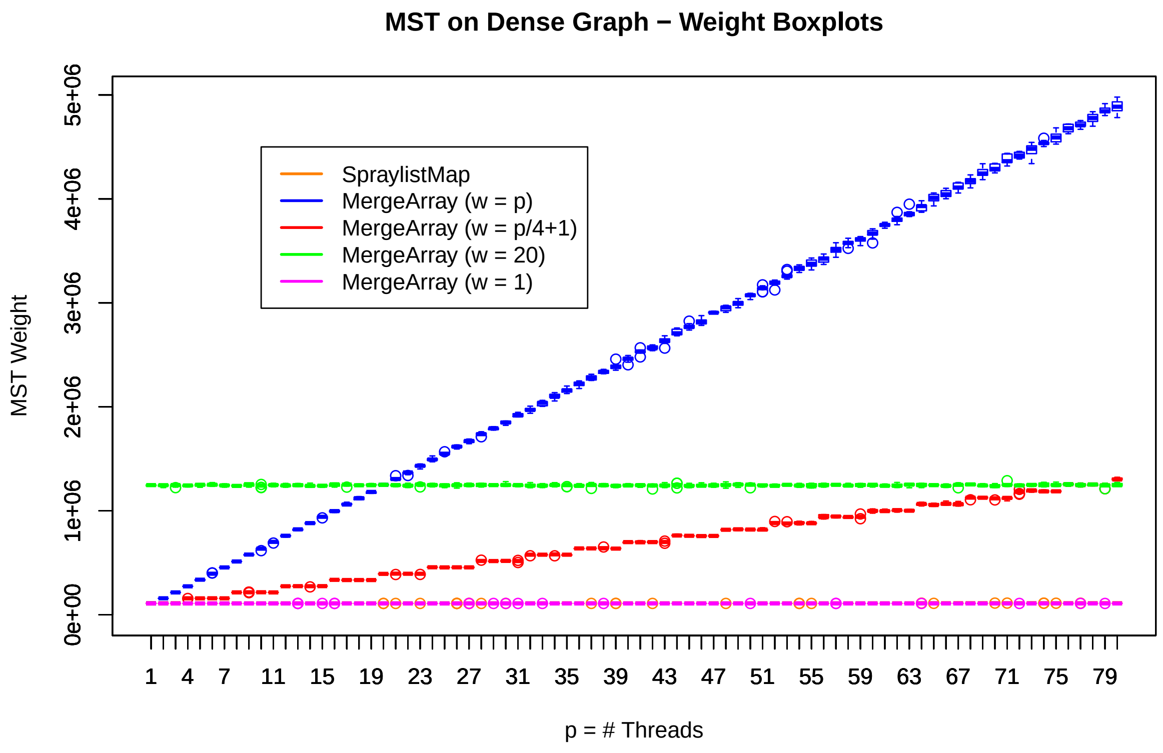


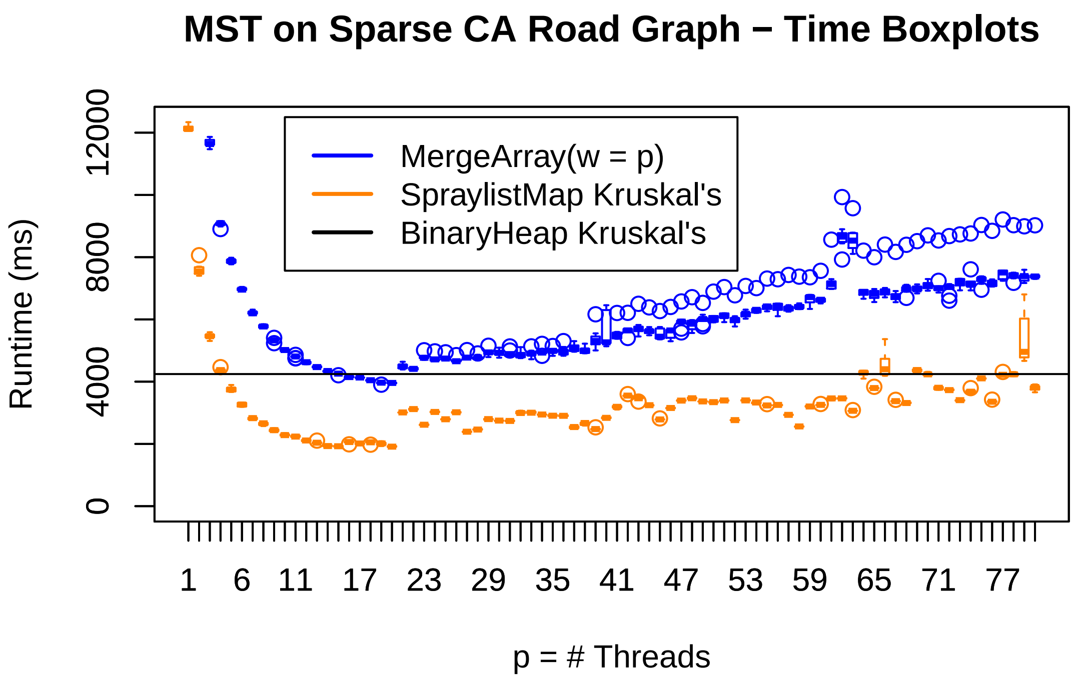
Figure 4: MST Benchmark Output Weights on dense random graph

These algorithms used relaxed data structures, and as such are not guaranteed to produce minimum spanning trees. As a baseline, in the graph, the weight of each edge was chosen uniformly from $[0, |V|)$, giving an optimal MST weight of 109,122 units, which is about 5.4561 per edge. Figure 4 shows the total weight of the MSTs output by the algorithms.

In each algorithm, the variation caused by in number of threads is small compared to the uncertainty between runs, thus most variation in the weight is due to the relaxation of the data structure. MergeArray with $w = 1$ is the top performer with an average weight of 109,169.3 (5.458737 per edge) and maximum weight of 109,273 (5.463923 per edge), but the Spraylist comes in a close second, outputting an average weight of 109,639.3 (5.482239 per edge) and maximum 112,940 (5.647282 per edge) over all experiments. Considering every MergeArray experiment, it is clear that the output weight is approximately a linear function of the width. For $w = 20$, the average weight is 1,246,434 units, or 62.32482 per edge, and for $w = 80$ the average weight is 4,887,520 (244.3882 per edge).

This benchmark illustrates the tradeoff between performance and accuracy when using relaxed data structures which must scale to many threads and large data sets, as well as the benefits of parallel algorithms which use MERGE to efficiently distribute and collect work rather than ones which use a centralized data structure.

I should note that looking at different classes of graphs can result in different performance characteristics. For example, the California road network used in Spraylist benchmarks, made fully-connected, is a sparse 1,965,206 node and 11,048,470 edge graph ($\frac{|E|}{|V|} = 5.622042$) which neither MergeArray nor Spraylist MST algorithms scale well on beyond 20 threads, as shown in Figure 5.



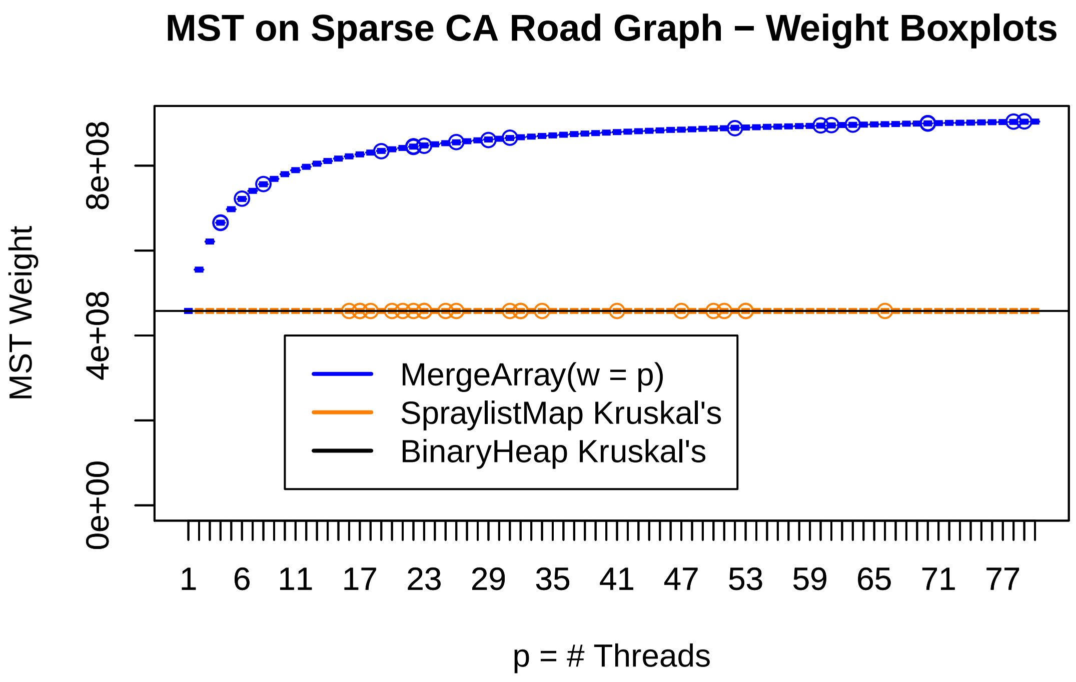


Figure 5: MST Benchmark Runtimes and Weights on sparse California road network

6.3 Priority Queue Operation Benchmarks

To measure the run time of each operation of the MergeArray priority queue, the following three-step benchmark was performed:

1. In parallel, an array of 2^{16} MergeArrays are created (with id = index in array) and filled with 1525 integers each. No two threads insert into the same one.
2. In parallel, each MergeArray is merged with the index zero MergeArray.
3. In parallel, each thread calls REMOVE-MIN $2^{16} \times 1525/p$ times to empty the MergeArray.

Figure 6 is the results of this experiment. While the contention of MERGE operations into the same "destination" MergeArray increases the time to perform step 2, its lazy nature makes MERGE a small fraction of the overall runtime which persistently decreases up to 80 threads.

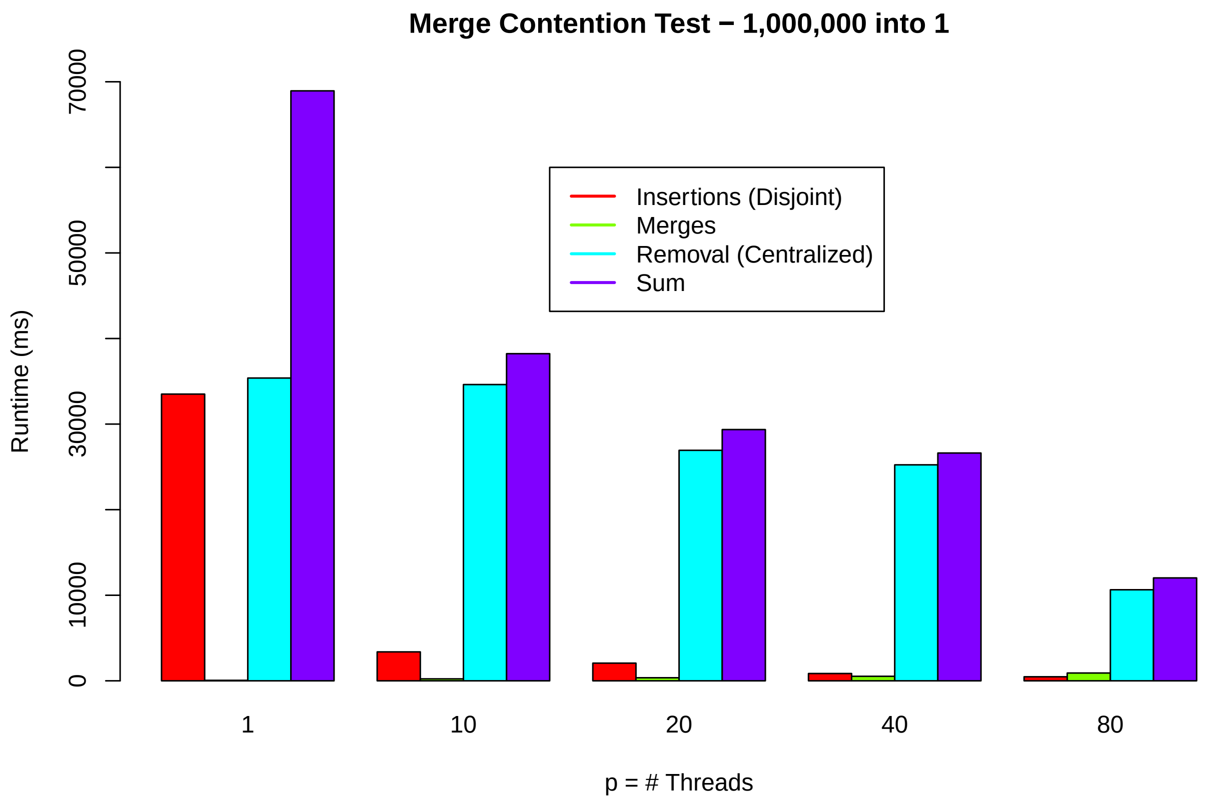


Figure 6: Results of Merge Contention Test and Operation Benchmark

6.4 Rank Error

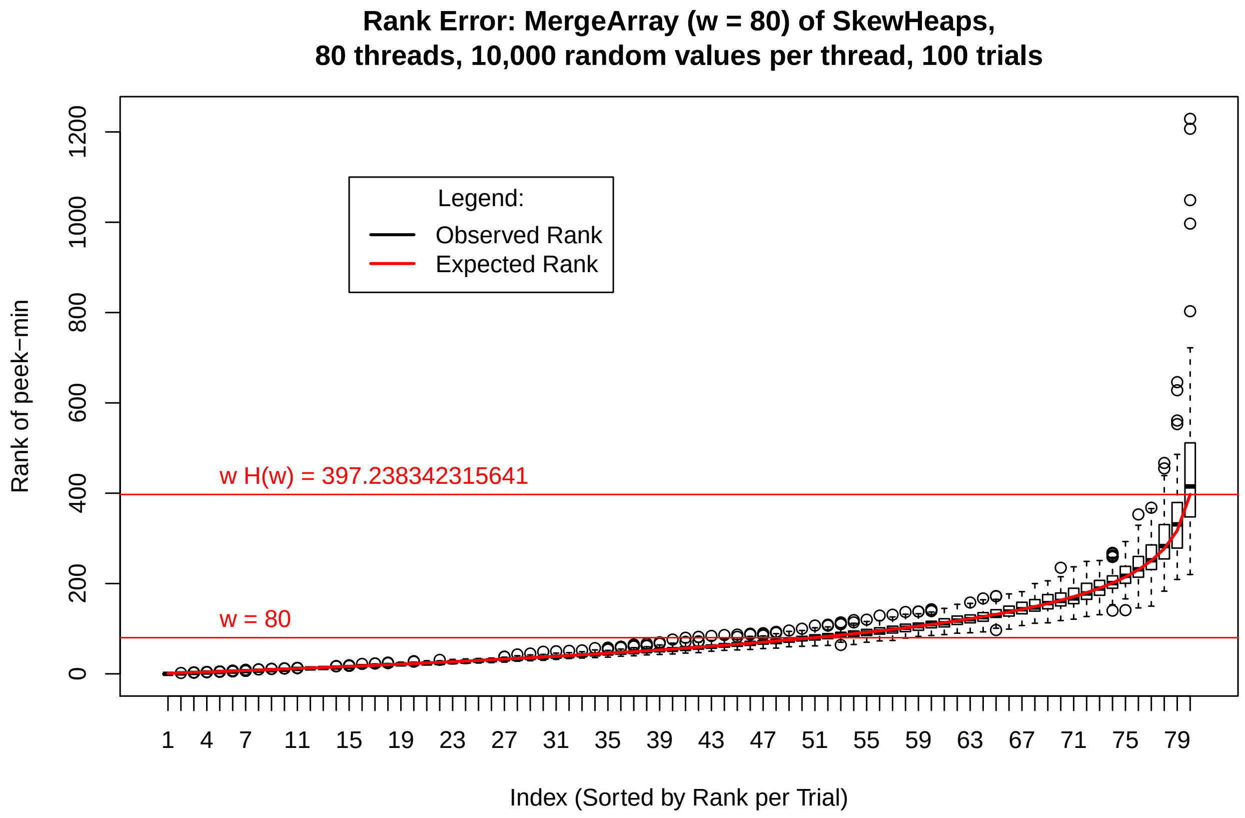
Theorem 10 predicts a expected worst-case rank-error of $wH(w) = O(w \log w)$ and average-case w for the results of a REMOVE-MIN operation on a MergeArray with width w after sufficiently-many non-concurrent INSERT operations are performed. In practice, INSERT operations may be highly concurrent, and the locking of indices in the MergeArray breaks the theoretical guarantee of uniform value distribution. The benchmark described in Algorithm 23 measures the distribution of minimum values over different indices in a more realistic setting.

Algorithm 23 MergeArray Rank Error Benchmark

```
int[] function MERGEARRAY-RANK-ERROR(int chunk-size, int num-threads)
    bound ← chunk-size × num-threads
    items ← RANDOM-SHUFFLE(ARRAY(IOTA(0, bound)))
    M ← EMPTY-PRIORITY-QUEUE(0, num-threads)
    for chunk ∈ CHUNKS(items, chunk-size) in parallel do
        for all i ∈ chunk do
            INSERT(&M, i)
        end for
    end for
    R ← [ PEEK-MIN(elem(n)) | n ∈ nodes(bag(handle(M))) ]
    return SORTED(R)
end function
```

Figure 7 (top) shows the results of running Algorithm 23 on a MergeArray of width $w = 80$ with chunk-size = 10,000 and num-threads = 80, resulting in 800,000 total insertions. While INSERT operations are concurrent rather than sequential, the observed rank error of the minimum value at each index of the MergeArray closely matches the theorem's prediction.

To further support that the distribution in practice is close to the prediction, the benchmark was also run where the random shuffle was done within each thread's chunk rather than over the entire items array. This means that each thread inserts a biased sample of the values, and in particular that exactly one thread inserts all items below 10,000. Figure 7 (bottom) shows the results, which still closely matches the predicted ranks. The outcome of this benchmark is strong evidence that APPLY-UNTIL makes an approximately uniform choice of indices in practice, and that Theorem 10 is a good model of the relaxation of the MergeArray priority queue.



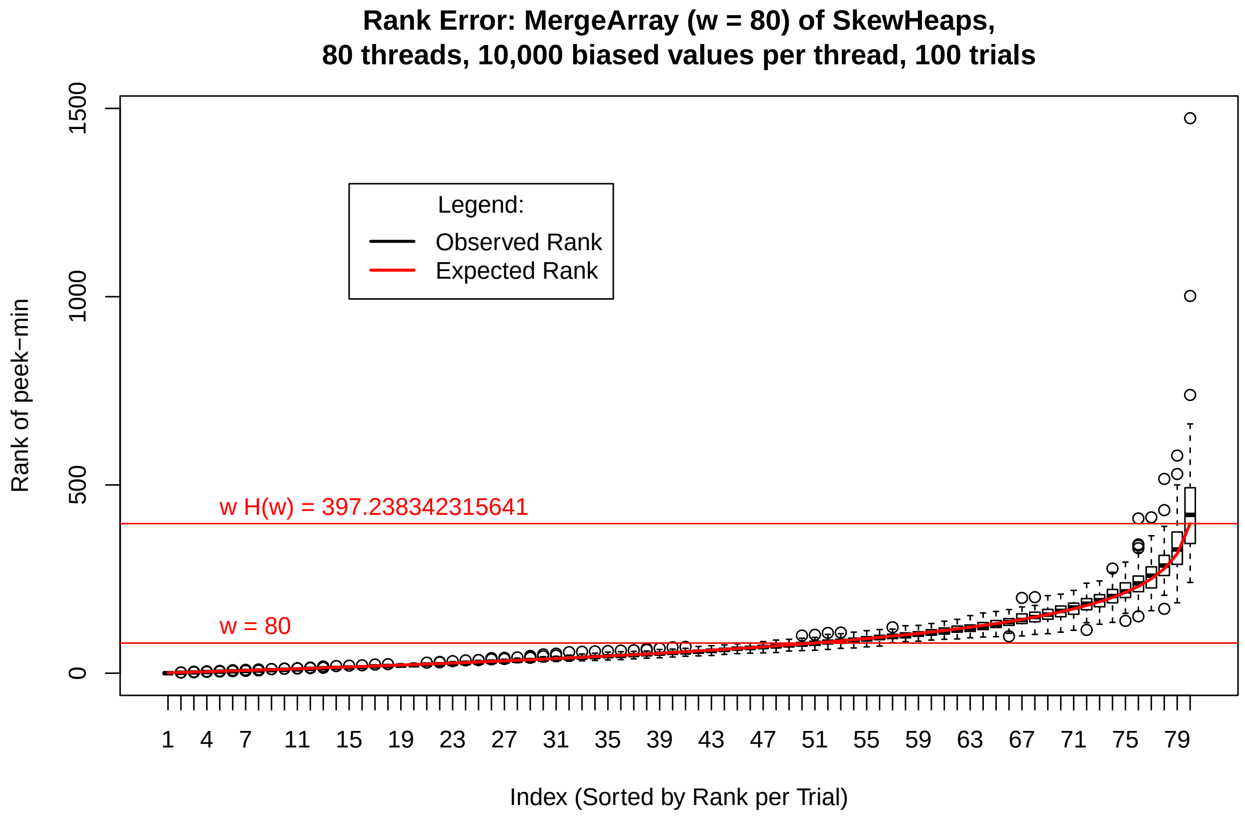


Figure 7: Results of Rank Error Experiment

7 Discussion and Future Work

MergeArray is a framework for building scalable, relaxed, linearizable concurrent data structures which supports a lock-free MERGE and conditionally-lock-free APPLY-UNTIL. While MergeArray is a small step towards an entirely-lock-free mergeable priority queue, it has met the goal of bringing the MERGE operation into the world of concurrent lock-free data structures.

A primary direction for future research is to study the essential parts of MergeArray which allow for lock-free MERGE, such as the relaxed semantics, lazy bulk movement, and coordination using monotonicity, and try to build a new mergeable priority queue which permits fine-grained access to every value without the semi-coarse-grained locking which makes supporting a fully-lock-free REMOVE-MIN difficult.

References

[ACS13] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? *CoRR*, abs/1311.3200, 2013.

[AKLS14] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. Technical Report MSR-TR-2014-16, September 2014. Under submission.

[AM95] James H. Anderson and Mark Moir. Universal constructions for multi-object operations (extended abstract). In *PROCEEDINGS OF THE 14TH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING*, pages 184–193, 1995.

[BLT12] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1177–1184, New York, NY, USA, 2012. ACM.

[Bro96] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, 1996.

[CDP96] V.A. Crupi, S.K. Das, and M.C. Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, volume 1, pages 255–262 vol.1, Aug 1996.

[Che92] Jingsen Chen. Merging and splitting priority queues and deques in parallel. In *Symposium Proceedings on Theory of Computing and Systems*, ISTCS'92, pages 1–11, London, UK, UK, 1992. Springer-Verlag.

[Cra72] Clark Allan Crane. *Linear Lists and Priority Queues As Balanced Binary Trees.* PhD thesis, Stanford, CA, USA, 1972. AAI7220697.

[Edm67] J Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, 71B:233–240, 1967.

[EHS12] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

[FSST86] Michael L. Fredman, Robert Sedgewick, DanielD. Sleator, and RobertE. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[Gal80] Zvi Galil. Applications of efficient mergeable heaps for optimization problems on trees. *Acta Informatica*, 13(1):53–58, 1980.

[HKLP12] Andreas Haas, Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. How fifo is your concurrent fifo queue? In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 1–8, New York, NY, USA, 2012. ACM.

[HS11] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Principles of Distributed Systems*, pages 313–328. Springer, 2011.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[KLP13] ChristophM. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg, 2013.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995. Cambridge Books Online.

[MTZ04] Ran Mendelson, Mikkel Thorup, and Uri Zwick. Meldable ram priority queues and minimum directed spanning trees. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 40–48, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[RSD14] Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *CoRR*, abs/1411.1209, 2014.

[SB08] Morten Stöckel and Søren Bøg. Concurrent data structures. 2008.

[SS85] Jörg-Rüdiger Sack and Thomas Strothotte. An algorithm for merging heaps. *Acta Inf.*, 22(2):171–186, 1985.

[ST86] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978.

[Zar97] Christos D. Zaroliagis. Simple and work-efficient parallel algorithms for the minimum spanning tree problem. *Parallel Processing Letters*, 07(01):25–37, 1997.