The SprayList: A Scalable Relaxed Priority Queue

Anonymous

Abstract

High-performance concurrent priority queues are essential for applications such as task scheduling and discrete event simulation. Unfortunately, even the best performing implementations do not scale past a number of threads in the single digits. This is because of the sequential bottleneck in accessing the elements at the head of the queue in order to perform a DeleteMin operation.

In this paper, we present the `SprayList`, a scalable priority queue with relaxed ordering semantics. Starting from a non-blocking SkipList, the main innovation behind our design is that the `DeleteMin` operations avoid a sequential bottleneck by "spraying" themselves onto the head of the SkipList list in a coordinated fashion. The spraying is implemented using a carefully designed random walk, so that `DeleteMin` returns an element among the first $O(p \log^3 p)$ in the list, with high probability, where p is the number of threads. We prove that the running time of a `DeleteMin` operation is $O(\log^3 p)$, with high probability, independent of the size of the list.

Our experiments show that the relaxed semantics allow the data structure to scale for very high thread counts, comparable to a classic unordered SkipList. Furthermore, we observe that, for reasonably parallel workloads, the scalability benefits of relaxation considerably outweigh the additional work due to out-of-order execution.

1. Introduction

The necessity for increasingly efficient, scalable concurrent data structures is one of the main software trends of the past decade. Efficient concurrent implementations are known for several fundamental data structures, such as hash tables [17], linked lists [14], SkipLists [13], pools [4], and trees [5]. On the other hand, several impossibility results [3, 12] suggest that not all data structures can have efficient concurrent implementations, due to an inherent sequentiality which follows from their sequential specification.

A classic example of such a data structure is the *priority queue*, which is widely used in applications such as scheduling and event simulation, e.g. [20]. In its simplest form, a priority queue stores a set of key-value pairs, and supports two operations: `Insert`, which adds a given pair to the data structure and `DeleteMin`, which returns the key-value pair with the smallest key currently present. Sequential priority queues are well understood, and classically implemented using a heap [19]. Unfortunately, heap-based *concurrent* priority queues suffer from both memory contention and sequential bottlenecks, not only when attempting to delete the single minimal key element at the root of the heap, but also when percolating small inserted elements up the heap.

SkipList-based implementations were proposed [20, 21, 27] in order to reduce these overheads. SkipLists are randomized list-based data structures which classically support `Insert` and `Delete` operations [23]. A SkipList is composed of several linked lists organized in levels, each skipping over fewer elements. SkipLists are desirable because they allow priority queue insertions and removals without the costly percolation up a heap or the rebalancing of a search tree. Highly concurrent SkipList-based priority queues have been studied extensively and have relatively simple implementations [13, 16, 20, 24]. Unfortunately, an *exact* concurrent SkipList-based priority queue, that is, one that maintains a linearizable [16] (or even quiescently-consistent [16]) order on `DeleteMin` operations, must still remove the minimal element from the leftmost node in the SkipList. This means that all threads must repeatedly compete to decide who gets this minimal node, resulting in a bottleneck due to contention, and limited scalability [20].

An interesting alternative has been to *relax* the strong ordering constraints on the output for better scalability. An early instance of this direction is the seminal work by Karp and Zhang [18], followed up by several other interesting proposals, e.g. [6, 9, 25], designed for the (synchronous) PRAM model. Recently, there has been a surge of interest in relaxed concurrent data structures, both on the theoretical side, e.g. [15] and from practitioners, e.g. [22]. In particular, Wimmer et al. [28] explored trade-offs between ordering and scalability for asynchronous priority queues. However, despite all this effort, it is currently not clear whether it is possible to design a relaxed priority queue which provides both ordering guarantees under asynchrony, *and* scalability under high contention for realistic workloads.

In this paper, we take a step in this direction by introducing the `SprayList`, a *scalable* relaxed priority queue implementation based on a SkipList. The `SprayList` provides probabilistic guarantees on the relative priority of returned elements, and on the running time of operations. At the same time, it shows *fully scalable* throughput for up to 80 concurrent threads under high-contention workloads.

The main limitation of past SkipList-based designs was that all threads clash on the first element in the list. Instead, our idea will be to allow threads to "skip ahead" in the list, so that concurrent attempts try to remove distinct, uncontended elements. The obvious issue with this approach is that one cannot allow threads to skip ahead too far, or many high priority (minimal key) elements will not be removed.

Our solution is to have the `DeleteMin` operations traverse the SkipList, not along the list, but via a tightly controlled random walk from its root. We call this operation a *spray*. Roughly, at each SkipList level, a thread flips a random coin to decide how many nodes to skip ahead at that level. In essence, we use local randomness and the random structure of the SkipList to balance accesses to the head of the list. The lengths of jumps at each level are chosen such that the probabilities of hitting nodes among the first $O(p \log^3 p)$ are close to uniform. (See Figure 1 for the intuition behind sprays.)

While a `DeleteMin` in an exact priority queue returns the element with the smallest key—practically one of the p smallest keys if p threads are calling `DeleteMin` concurrently— the `SprayList` ensures that the returned key is among the $O(p \log^3 p)$ smallest keys (for some linearization of operations), and that each operation completes within $\log^3 p$ steps, both with high probability. Our design also provides anti-starvation guarantees, in particular, that elements with small keys will not remain in the queue for too long.

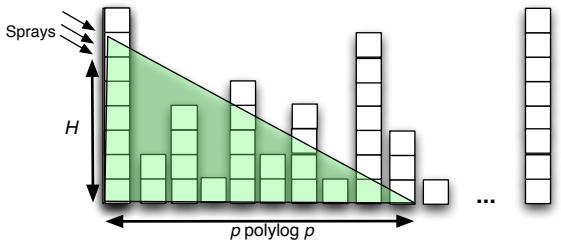

Figure 1: The intuition behind the `SprayList`. Threads start at height H and perform a random walk on nodes at the start of the list, attempting to acquire the node they land on.

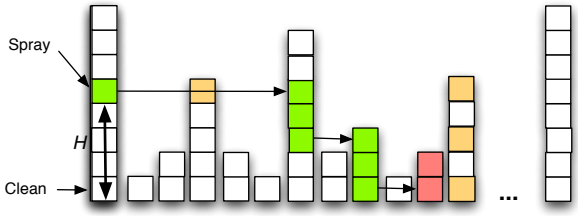

Figure 2: A simple example of a spray, with no padding. Green nodes are touched by the `Spray`, and the thread stops at the red node. Orange nodes could have been chosen for jumps, but were not.

The formal proof of these guarantees is the main technical contribution of our paper.

Specifically, our proofs are inspired by an elegant argument proving that sprays are near-uniform on an *ideal* (uniformly-spaced) SkipList, given in Section 3.2. However, this argument breaks on a realistic SkipList, whose structure is random. Precisely bounding the node hit distribution on a realistic SkipList is significantly more involved[1] Given the node hit distribution, we can upper bound the probability that two sprays collide. In turn, this upper bounds the expected number of operation retries, and the amount of time until a specific key is removed, given that a nearby key has already been removed. The uniformity of the spray distribution also allows us to implement an optimization whereby large contiguous groups of claimed nodes are physically removed by a randomly chosen *cleaner* thread.

In sum, our analysis gives strong probabilistic guarantees on the rank of a removed key, and on the running time of a `Spray` operation. Our algorithm is designed to be lock-free, but the same spraying technique would work just as well for a lock-based SkipList.

A key question is whether a priority queue with such relaxed guarantees can be useful in practice. We answer this question in the affirmative, by examining the practical performance of the SprayList through a wide series of benchmarks, including synthetic high-contention tests, discrete-event simulation, and running single-source shortest paths on grid, road network, and social graphs.

We compare our algorithm's performance to that of the quiescently-consistent priority queue of Lotan and Shavit [21], the state-of-the-art SkipList-based priority queue implementation of Lindén and Jonsson [20] and the recent k-priority queue of Wimmer et al. [28].[2]

Our first finding is that our data structure shows *fully scalable* throughput for up to 80 concurrent threads under high-contention workloads. We then focus on the trade-off between the strength of the ordering semantics and performance. We show that, for discrete-event simulation and a subset of graph workloads, the amount of additional work due to out-of-order execution is amply compensated by the increase in scalability.

Related Work. The first concurrent SkipList was proposed by Pugh [24], while Lotan and Shavit [21] were first to employ this data structure as a concurrent priority queue. They also noticed that the original implementation is not linearizable, and added a time-stamping mechanism for linearizability. Herlihy and Shavit [16] give a lock-free version of this algorithm.

Sundell and Tsigas [27] proposed a lock-free SkipList-based implementation which ensures linearizability by preventing threads from moving past a list element that has not been fully removed. Instead, concurrent threads help with the cleanup process. Unfortunately, all the above implementations suffer from very high contention under a standard workload, since threads are still all continuously competing for a handful of locations.

Recently, Lindén and Jonsson[20] presented an elegant design with the aim of reducing the bottleneck of deleting the minimal element. Their algorithm achieves a $30-80\%$ improvement over previous SkipList-based proposals; however, due to high contention compare-and-swap operations, its throughput does not scale past 8 concurrent threads. To the best of our knowledge, this is a limitation of all known exact priority queue implementations.

Other recent work by Mendes et al. [7] employed *elimination* techniques to adapt to contention in an effort to extend scalability. Even still, their experiments do not show throughput scaling beyond 20 threads.

Another direction by Wimmer et al. [28] presents lock-free priority queues which allow the user to dynamically decrease the strength of the ordering for improved performance. In essence, the data structure is distributed over a set of *places*, which behave as exact priority queues. Threads are free to perform operations on a place as long as the ordering guarantees are not violated. Otherwise, the thread merges the state of the place to a global task list, ensuring that the relaxation semantics hold deterministically. The paper provides analytical bounds on the work wasted by their algorithm when executing a parallel instance of Dijkstra's algorithm, and benchmark the execution time and wasted work for running parallel Dijkstra on a set of random graphs. Intuitively, the above approach provides a tighter handle on the ordering semantics than ours, at the cost of higher synchronization cost. The relative performance of the two data structures depends on the specific application scenario, and on the workload.

Our work can be seen as part of a broader research direction on high-throughput concurrent data structures with relaxed semantics [15, 26]. Examples include container data structures which (partially or entirely) forgo ordering semantics such as the rendezvous mechanism [2] or the CAFE task pool [4]. Recently, Dice et al. [10] considered randomized data structures for highly scalable exact and approximate counting.

2. The SprayList Algorithm

In this section, we describe the SprayList algorithm. The `Search` and `Insert` operations are identical to the standard implementations of lock-free SkipLists [13, 16], for which several freely available implementations exist, e.g. [8, 13]. In the following, we assume the reader is familiar with the structure of a SkipList, and give an overview of standard lock-free SkipList operations, then focus on our `Spray` (`DeleteMin`) procedure.

[1] We perform the analysis in a restricted asynchronous model, defined in Section 3.1.

[2] Due to the complexity of the framework of [28], we only provide a partial comparison with our algorithm in terms of performance.

2.1 The Classic Lock-Free SkipList

Our presentation follows that of Fraser [13, 16], and we direct the reader to these references for a detailed presentation and pseudocode. A useful general observation is that the the the data structure maintains an implementation of a set, defined by the bottom-level lock-free list. Throughout this paper we will use the convention that the lowest level of the SkipList is level 0.

Pointer Marking. A critical issue when implementing lock-free lists is that nodes might "vanish" (i.e., be removed concurrently) while some thread is trying access them. Fraser [13] solves this problem by reserving a *marked* bit in each pointer field of the SkipList. A node with a marked bit is itself *marked*. The bit is always checked and masked off before accessing the node.

Search. As in the sequential implementation, the SkipList search procedure looks for a *left* and *right* node at each level in the list. These are adjacent nodes, with key values less-than and greater-than-equal-to the search key, respectively.

The search loop checks whether nodes are marked and skips over them, since they have been logically removed from the list. The search procedure also helps clean up marked nodes from the list: if the thread encounters a sequence of marked nodes, these are removed by updating the unmarked successor to point to the unmarked predecessor in the list at this level. If the node accessed by the thread becomes marked during the list traversal, the entire search is re-started from the SkipList root. The operation returns the node with the required key, if found at some level of the list, as well as the list of successors of the node.

Delete. Deletion of a node with key k begins by first searching for the node. If the node is found, then it is *logically deleted* by updating its value field to `NULL`. The next stage is to mark each link pointer in the node. This will prevent an new nodes from being inserted after the deleted node. Finally, all references to the deleted node are removed. Interestingly, this can be done by simply performing a *search* for the key: recall that the search procedure swings list pointers over marked nodes.

Cleaners / Lotan-Shavit DeleteMin. In this context, the Lotan-Shavit [21] `DeleteMin` operation traverses the bottom list attempting to acquire a node via compare-and-swap. (Their original implementation is lock-based.) Once acquired, the node is logically deleted and then removed via a search operation. We note that this is exactly the same procedure as the periodic *cleaner* operations we implement.

Insert. A new node is created with a randomly chosen height. The node's pointers are unmarked, and the set of successors is set to the successors returned by the *search* method on the node's key. Next, the node is inserted into the lists by linking it between the successors and the predecessors obtained by searching. The updates are performed using compare-and-swap. If a compare-and-swap fails, the list must have changed, and the call is restarted. The insert then progressively links the node up to higher levels. Once all levels are linked, the method returns.

2.2 Spraying and Deletion

The goal of the `Spray` operation is to emulate a uniform choice among the $O(p \log^3 p)$ highest-priority items.[3] To perform a `Spray`, a process starts at the front of the SkipList, and at some initial height h. (See Figure 2 for an illustration.)

At each horizontal level ℓ of the list, the process first jumps forward for some small, randomly chosen number of steps $j_\ell \geq 0$. After traversing those nodes, the process descends some number of levels d_ℓ, then resumes the horizontal jumps. We iterate this procedure until the process reaches a node at the bottom of the SkipList.

Once on the bottom list, the process attempts to acquire the current node. If the node is successfully acquired, the thread starts the standard SkipList removal procedure, marking the node as logically deleted. (As in the SkipList algorithm, logically deleted nodes are ignored by future traversals.) Otherwise, if the process fails to acquire the node, it either re-tries a `Spray`, or, with low probability, becomes a *cleaner* thread, searching linearly through the bottom list for an available node.

Spray Parameters. An efficient `Spray` needs the right combination of parameters. In particular, notice that we can vary the starting height, the distribution for jump lengths at each level, and how many levels to descend between jumps. The constraints are poly-logarithmic time for a `Spray`, and a roughly uniform distribution over the head of the list. At the same time, we need to balance the average length of a `Spray` with the expected number of thread collisions on elements in the bottom list.

We now give an overview of the parameter choices for our implementation. For simplicity, consider a SkipList on which no removes have yet occurred due to `Spray` operations. We assume that the data structure contains n elements, where $n \gg p$.

Starting Height. Each `Spray` starts at list level $H = \log p + K$, for some constant K.[4] (Intuitively, starting the `Spray` from a height less than $\log p$ leads to a high number of collisions, while starting from a height of $C \log p$ for $C > 1$ leads to `Spray`s which traverse beyond the first $O(p \log^3 p)$ elements.)

Jump Length Distribution. We choose the maximum number of forward steps L that a `Spray` may take at a level to be $L = M \log^3 p$, where $M \geq 1$ is a constant. Thus, the number of forward steps at level ℓ, is uniformly distributed in the interval $[0, L]$.

The intuitive reason for this choice is that a randomly built SkipList is likely to have chains of $\log p$ consecutive elements of height one, which can only be accessed through the bottom list. We wish to be able to choose uniformly among such elements, and we therefore need L to be at least $\log p$. (While the same argument does not apply at higher levels, our analysis shows that choosing this jump length j_ℓ yields good uniformity properties.)

Levels to Descend. The final parameter is the choice of how many levels to descend after a jump. A natural choice, used in our implementation, is to descend one level at a time, i.e., perform horizontal jumps at each SkipList level.

In the analysis, we consider a slightly more involved random walk, which descends $D = \max(1, \lfloor \log \log p \rfloor)$ consecutive levels after a jump at level ℓ. We must always traverse the bottom level of the SkipList (or we will never hit SkipList nodes of height 1) so we round H down to the nearest multiple of D. We note that we found empirically that setting $D = 1$ yields similar performance.

In the following, we parametrize the implementation by H, L and D such that D evenly divides H. The pseudocode for `Spray`(H, L, D) is given below.

Node Removal. Once it has successfully acquired a node, the thread proceeds to remove it as in a standard lock-free SkipList [13, 16]. More precisely, the node is logically deleted, and its references are marked as invalid.

In a standard implementation, the final step would be to swing the pointers from its predecessor nodes to its successors. However, a spraying thread skips this step and returns the node. Instead, the pointers will be corrected by *cleaner* threads: these are randomly chosen `DeleteMin` operations which linearly traverse the bottom of the list in order to find a free node, whose operation is described in Section 2.3.

[3] The natural idea of picking such an element at random runs into the issue of how to access a random element efficiently under contention.

[4] Thoughout this paper, unless otherwise stated, we consider all logarithms to be integer, and omit the floor $\lfloor \cdot \rfloor$ notation.

```
x ← head          /* x = pointer to current location */
                                  /* Assume D divides H */
ℓ ← H                          /* ℓ is the current level */
while ℓ ≥ 0 do
    Choose j_ℓ ← Uniform[0, L]              /* random jump */
    Walk x forward j_ℓ steps on list at height ℓ
                   /* traverse the list at this level */
    ℓ ← ℓ − D                        /* descend D levels */
Return x
```

Algorithm 1: Pseudocode for Spray(H, L, D). Recall that the bottom level of the SkipList has height 0.

2.3 Optimizations

Padding. A first practical observation is that the Spray procedure above is biased against elements at the front of the list. For example, it would be extremely unlikely that the second element in the list is hit. To circumvent this bias, we simply "pad" the SkipList: we add $K(p)$ dummy entries in the front of the SkipList. If a Spray would return one of the first $K(p)$ dummy entries, it instead restarts. We choose $K(p)$ such that the restart probability is low, while, at the same time, the probability that a node in the interval $[K(p)+1, p\log^3 p]$ is hit is close to $1/p\log^3 p$ (uniform).

Cleaners. Before each new Spray, each thread flips a low-probability coin to decide whether it will become a *cleaner* thread. A cleaner thread simply traverses the bottom-level list of the SkipList linearly (skipping the padding nodes), searching for a key to acquire. In other words, a cleaner simply executes a lock-free version of the Lotan-Shavit [21] DeleteMin operation. At the same time, notice that cleaner threads adjust pointers for nodes previously acquired by other Spray operations, reducing contention and wasted work. Interestingly, we notice that a cleaner thread can swing pointers across a whole group of nodes that have been marked as logically deleted, effectively batching this part of the remove process.

The existence of cleaners is not needed in the analysis, but is a useful optimization. In the implementation, the probability of an operation becoming a cleaner is $1/p$, i.e., roughly one in p Sprays becomes a cleaner.

Adapting to Contention. We also note that the SprayList allows threads to adjust the spray parameters based on the level of contention. In particular, a thread can estimate p, increasing its estimate if it detects higher than expected contention (in the form of collisions) and decreasing its estimate if it detects low contention. Each thread parametrizes its Spray parameters the same way as in the static case, but using its estimate of p rather than a known value. Note that with this optimization enabled, if only a single thread access the SprayList, it will always dequeue the the element with the smallest key.

3. Spray Analysis

In this section, we analyze the behavior of Spray operations. We describe our analytical model in Section 3.1. We then give a first motivating result in Section 3.2, bounding the probability that two Spray operations collide for an ideal SkipList.

We state our main technical result, Theorem 3, and provide a proof overview in Section 3.3. The full proof of Theorem 3 is rather technical, and can be found in the non-anonymous supplemental material submitted with this paper. In essence, given our model, our results show that SprayLists do not return low priority elements except with extremely small probability (Theorem 2) and that there is very low contention on individual elements, which in turn implies the bound on the running time of Spray (Corollary 1).

3.1 Analytical Model

As with other complex concurrent data structures, a complete analysis of spraying in a fully asynchronous setting is extremely challenging. Instead, we restrict our attention to showing that, under reasonable assumptions, spraying approximates uniform choice amongst roughly the first $O(p\log^3 p)$ elements. We will then use this fact to bound the contention between Spray operations. We therefore assume that there are $n \gg p\log^3 p$ elements in the SkipList.

We consider a set of at most p concurrent, asynchronous processes trying to perform Spray operations, traversing a *clean* SkipList, i.e. a SkipList whose height distribution is the same as one that has just been built. In particular, a node has height $\geq i$ with probability $1/2^{i-1}$, independent of all other nodes. When two or more Spray operations end at the same node, all but one of them must retry, until all Sprays land at unique nodes (because at most one thread can obtain a node). Also, we do not consider concurrent inserts or removes among the first $O(p\log^3 p)$ items in the list.

On the one hand, this setup is clearly only an approximation of a real execution, since concurrent inserts and removes may occur in the prefix and change the SkipList structure. Also, the structure of the list may have been biased by previous Spray operations. (For example, previous sprays might have been biased to land on nodes of large height, and therefore such elements may be less probable in a dynamic execution.)

On the other hand, we believe this to be a reasonable approximation for our purposes. We are interested mainly in spray distribution; concurrent deletions should not have a high impact, since, by the structure of the algorithm, logically deleted nodes are skipped by the spray. Also, in many scenarios, a majority of the concurrent inserts are performed towards the back of the list (corresponding to elements of lower priority than those at the front). Finally, the effect of the spray distribution on the height should be limited, since removing an element uniformly at random from the list does not change its expected structure, and we closely approximate uniform removal. Also, notice that cleaner threads (linearly traversing the bottom list) periodically "refresh" the SkipList back to a clean state.

3.2 Motivating Result: Analysis on a Perfect SkipList

In this section, we illustrate some of the main ideas behind our runtime argument by first proving a simpler claim, Theorem 1, which holds for an idealized SkipList. Basically, Theorem 1 says that, on SkipList where nodes of the same height are evenly spaced, the Spray procedure ensures low contention on individual list nodes.

More precisely, we say a SkipList is *perfect* if the distance between any two elements of height $\geq j$ is 2^j, and the first element has height 0. On a perfect SkipList, we do not have to worry about probability concentration bounds when considering SkipList structure, which simplifies the argument. (We shall take these technicalities into account in the complete argument in the next section.)

We consider the Spray(H, L, D) procedure with parameters $H = \log p$, $L = \log p$, and $D = 1$, the same as our implementation version. Practically, the walk starts at level $\log p - 1$ of the SkipList, and, at each level, uniformly chooses a number of forward steps between $[1, \log p]$ before descending. We prove the following upper bound on the collision probability, assuming that $\log p$ is even:

Theorem 1. *For any position x in a perfect SkipList, let $F_p(x)$ denote the probability that a* Spray$(\log p, \log p, 1)$ *lands at x. Then $F_p(x) \leq 1/p$.*

Proof. Fix in the following parameters $H = \log p$, $L = \log p$, $D = 1$ for the Spray, and consider an arbitrary such operation. Let a_i be the number of forward steps taken by the Spray at level i, for all $0 \leq i \leq \log p - 1$.

We start from the observation that, on a *perfect* SkipList, the operation lands at the element of index $\sum_{i=0}^{\log p-1} a_i 2^i$ in the bottom list. Thus, for any element index x, to count the probability that a `Spray` which lands at x, it suffices to compute the probability that a $(\log p)$-tuple $(a_0, \ldots, a_{\log p-1})$ whose elements are chosen independently and uniformly from the interval $\{1, \ldots, \log p\}$ has the property that the jumps sum up to x, that is,

$$\sum_{i=0}^{\log p-1} a_i 2^i = x. \tag{1}$$

For each i, let $a_i(j)$ denote the jth least significant bit of a_i in the binary expansion of a_i, and let $x(j)$ denote the jth least significant bit of x in its binary expansion.

Choosing an arbitrary `Spray` is equivalent to choosing a random $(\log p)$-tuple $(a_1, \ldots, a_{\log p})$ as specified above. We wish to compute the probability that the random tuple satisfies Equation 1. Notice that, for $\sum_{i=0}^{\log p-1} a_i 2^i = x$, we must have that $a_0(1) = x(1)$, since the other a_i are all multiplied by some nontrivial power of 2 in the sum and thus their contribution to the ones digit (in binary) of the sum is always 0. Similarly, since all the a_i except a_0 and a_1 are bit-shifted left at least twice, this implies that if Equation 1 is satisfied, then we must have $a_1(1) + a_0(2) = x(2)$. In general, for all $1 \leq k \leq \log p - 1$, we see that to satisfy Equation 1, we must have that $a_k(1) + a_{k-1}(2) + \ldots + a_0(k) + c = x(k)$, where c is a carry bit determined completely by the choice of $a_0, \ldots, a_{k-1}$.

Consider the following random process: in the 0th round, generate a_0 uniformly at random from the interval $\{1, \ldots, \log p\}$, and test if $a_0(1) = x(1)$. If it satisfies this condition, continue and we say it passes the first round, otherwise, we say we fail this round. Iteratively, in the kth round, for all $1 \leq k \leq \log p - 1$, randomly generate an a_k uniformly from the interval $\{1, \ldots, \log p\}$, and check that $a_k(1) + a_{k-1}(2) + \ldots + a_0(k) + c = x(k) \mod 2$, where c is the carry bit determined completely by the choice of $a_0, \ldots, a_{k-1}$ as described above. If it passes this test, we continue and say that it passes the kth round; otherwise, we fail this round. If we have yet to fail after the $(\log p - 1)$st round, then we output `PASS`, otherwise, we output `FAIL`. By the argument above, the probability that we output `PASS` with this process is an upper bound on the probability that a `Spray` lands at x.

The probability we output `PASS` is then

$$\Pr[\text{pass 0th round}] \prod_{i=0}^{\log p-2} \Pr[\text{pass } (i+1)\text{th round} | A_i]$$

where A_i is the event that we pass all rounds $k \leq i$. Since a_0 is generated uniformly from the interval $\{1, 2, \ldots, \log p\}$, and since $\log p$ is even by assumption, the probability that the least significant bit of a_0 is $x(1)$ is exactly $1/2$, so

$$\Pr[\text{pass 0th round}] = 1/2. \tag{2}$$

Moreover, for any $1 \leq i \leq \log p - 2$, notice that conditioned on the choice of $a_1, \ldots, a_i$, the probability that we pass the $(i+1)$th round is exactly the probability that the least significant bit of a_{i+1} is equal to $x(i+1) - (a_i(2) + \ldots + a_0(i+1) + c) \mod 2$, where c is some carry bit as we described above which only depends on $a_1, \ldots, a_i$. But this is just some value $v \in \{0, 1\}$ wholly determined by the choice of $a_0, \ldots, a_i$, and thus, conditioned on any choice of $a_0, \ldots, a_i$, the probability that we pass the $(i+1)$th round is exactly $1/2$ just as above. Since the condition that we pass the kth round for all $k \leq i$ only depends on the choice of$a_0, \ldots, a_i$, we conclude that

$$\Pr[\text{pass } (i+1)\text{th round} | A_i] = 1/2. \tag{3}$$

Therefore, we have $\Pr[\text{output } \texttt{PASS}] = (1/2)^{\log p} = 1/p$, which completes the proof.

□

3.3 Complete Runtime Analysis for `DeleteMin`

In this section, we show that, given an arbitrary SkipList, each `Spray` operation completes in $O(\log^3 p)$ steps, with high probability. The crux of this result (stated in Corollary 1) is a characterization of the probability distribution induced by `Spray` operations on an arbitrary SkipList, which we obtain in Theorem 3.

Our results require some mathematical preliminaries. For simplicity of exposition, throughout this section and in the full analysis (given in the non-anonymous supplemental material submitted with this paper) we assume p which is a power of 2. (If p is not a power of two we can instead run `Spray` with the p set to the smallest power of two larger than the true p, and incur a constant factor loss in the strength of our results.)

We consider `Spray`s with the parameters $H = \log p$, $L = M \log^3 p$, and $D = \max(1, \lfloor \log \log p \rfloor)$. Let ℓ_p be the number of levels at which jumps are performed; in particular $\ell_p = \lfloor \log p / \lfloor \log \log p \rfloor \rfloor - 1$.

Since we only care about the relative ordering of the elements in the SkipList with each other and not their real priorities, we will call the element with the ith lowest priority in the SkipList the ith element in the SkipList. We will also need the following definition.

Definition 1. *Fix two positive functions $f(p), g(p)$.*

- *We say that f and g are* asymptotically equal*, $f \simeq g$, if $\lim_{p\to\infty} f(p)/g(p) = 1$.*
- *We say that $f \lesssim g$, or that g* asymptotically bounds *f, if there exists a function $h \simeq 1$ so that $f(p) \leq h(p)g(p)$ for all p.*

Note that saying that $f \simeq g$ is stronger than saying that $f = \Theta(g)$, as it insists that the constant that the big-Theta would hide is in fact 1, i.e. that asymptotically, the two functions behave exactly alike even up to constant factors.

There are two sources of randomness in the `Spray` algorithm and thus in the statement of our theorem. First, there is the randomness over the choice of the SkipList. Given the elements in the SkipList, the randomness in the SkipList is over the heights of the nodes in the SkipList. To model this rigorously, for any such SkipList S, we identify it with the n-length vectors $(h_1, \ldots, h_n)$ of natural numbers (recall there are n elements in the SkipList), where h_i denotes the height of the ith node in the SkipList. Given this representation, the probability that S occurs is $\prod_{i=1}^{n} 2^{-(h_i)}$.

Second, there is the randomness of the `Spray` algorithm itself. Formally, we identify each `Spray` with the ℓ_p-length vector $(a_0, \ldots, a_{\ell_p - 1})$ where $1 \leq a_i \leq M \log^3 p$ denotes how far we walk at height $i\lfloor \log \log p \rfloor$, and a_0 denotes how far we walk at the bottom height. Our `Spray` algorithm uniformly chooses a combination from the space of all possible `Spray`s. For a fixed SkipList S, and given a choice for the steps at each level in the `Spray`, we say that the `Spray` *returns* element i if, after doing the walk prescribed by the lengths chosen and the procedure described in Algorithm 1, we end at element i. For a fixed SkipList $S \in \mathcal{S}$ and some element i in the SkipList, we let $F_p(i, S)$ denote the probability that a `Spray` returns element i.

Definition 2. *We say an event happens* with high probability *or* w.h.p. *for short if it occurs with probability at least $1 - p^{-\Omega(M)}$, where M is the constant defined in Algorithm 1.*

With this definitions, the main theorems we prove are the following.

Theorem 2. *In the model described above, no `Spray` will return an element beyond the first $M(1 + \frac{1}{\log p})\sigma(p)p \log^3 p \simeq Mp \log^3 p$, with probability at least $1 - p^{-\Omega(M)}$.*

This theorem states simply that sprays do not go too far past the first $O(p \log^3 p)$ elements in the SkipList, which demonstrates that our SprayList does return elements with relatively small priority.

The proof of Theorem 2 is fairly straightforward and uses standard concentration bounds and is available in the non-anonymous supplemental material submitted with this paper. However, the tools we use there will be crucial to later proofs. The other main technical contribution of this paper is the following theorem.

Theorem 3. *For $p \geq 2$ and under the stated assumptions, there exists an interval of elements $I(p) = [a(p), b(p)]$ of length $b(p) - a(p) \simeq Mp\log^3 p$ and endpoint $b(p) \lesssim Mp\log^3 p$, such that for all elements in the SkipList in the interval $I(p)$, we have that*

$$\frac{1}{Mp\log^3 p} \lesssim F_i(p, S) \lesssim \frac{2}{Mp\log^3 p},$$

w.h.p. over the choice of S.

In plain words, this theorem states that there exists a range of elements $I(p)$, whose length converges to $Mp\log^3 p$, such that if you take a random SkipList, then with high probability over the choice of that SkipList, the random process of performing `Spray` approximates uniformly random selection of elements in the range $I(p)$, up to a factor of two. The condition $b(p) \lesssim Mp\log^3 p$ simply means that the right endpoint of the interval is not very far to the right. In particular, if we pad the start of the SkipList with $K(p) = a(p)$ dummy elements, the `Spray` procedure will approximate uniform selection from roughly the first $Mp\log^3 p$ elements, w.h.p. over the random choice of the SkipList. The proof of Theorem 3 is fairly involved; we provide an overview here, and the full argument is available in the non-anonymous supplemental material submitted with this paper.

Proof Overview. First we prove that the fraction of `Spray`s which hit any $i \in I(p)$ is asymptotically bounded above by $1/(Mp\log^3 p)$, then we prove the other direction of the inequality, which suffices to prove the theorem. The proof of the upper bound proceeds, at a high level, as follows. First, for any $i \in I(p)$, we filter the `Spray`s removing those `Spray`s which are not in the "valid" interval in the SkipList after each level of the spray, and argue that unless the `Spray` is in these intervals at each level, it cannot reach i w.h.p. over the choice of the SkipList. Thus it suffices to count the number of `Spray`s which are in these valid intervals at each step. By using concentration bounds we can argue that w.h.p., up to a small multiplicative error which vanishes in the limit as $p \to \infty$, the number of elements in the SkipList at any given height in any given interval behaves exactly like its expectation. Thus, we can count the number of `Spray`s which are in valid intervals at every height, and we derive that this number of asymptotically bounded by $1/(Mp\log^3 p)$.

The proof of the lower bound is in the same vein and makes the additional observation that if these valid intervals we described above are not too large and are close together, then every `Spray` which is in a valid interval at any height has a good shot of being valid at the next height, and any `Spray` which is in the bottommost valid interval can reach i. This, when done carefully, yields that the probability that a `Spray` stays valid and then hits i is lower bounded by $1/(2Mp\log^3 p)$ asymptotically.

Runtime Bound. Given this theorem, we then use it to bound the probability of collision for two Sprays, which in turn bounds the running time for a `DeleteMin` operation, which yields the following Corollary. Given Theorem 3, its proof is fairly straightforward, and can be found in the non-anonymous supplemental material submitted with this paper.

Corollary 1. *In the model described above,* `DeleteMin` *takes $O(\log^3 p)$ time in expectation, and probability at least $1 - p^{-\Omega(M)}$.*

4. Implementation Results

Methodology. Experiments were performed on a Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors. Each processor has 10 2.40 GHz cores, each of which multiplexes two hardware threads, so in total our system supports 80 hardware threads. Each core has private write-back L1 and L2 caches; an inclusive L3 cache is shared by all cores.

We examine the performance of our algorithm on a suite of benchmarks, designed to test its various features. Where applicable, we compare several competing implementations, described below.

Lotan and Shavit Priority Queue. The SkipList based priority queue implementation of Lotan and Shavit on top of Keir Fraser's SkipList [13] which simply traverses the bottom level of the SkipList and removes the first node which is not already logically deleted. The logical deletion is performed using a `Fetch-and-Increment` operation on a 'deleted' bit. Physical deletion is performed immediately by the deleting thread. Note that this algorithm is *not* linearizable, but quiescently consistent. This implementation uses much of the same code as the `SprayList`, but does not provide state of the art optimizations.

Lindén and Jonsson Priority Queue. The priority queue implementation provided by Lindén et. al. is representative of state of the art of linearizable priority queues [20]. This algorithm has been shown to outperform other linearizable priority queue algorithms under benchmarks similar to our own. This algorithm is optimized to minimize compare-and-swap (CAS) operations performed by `DeleteMin`. Physical deletion is batched and performed by a deleting thread only when the number of logically deleted threads exceeds a threshold.

Fraser Random Remove. An implementation using Fraser's SkipList which, whenever `DeleteMin` would be called, instead deletes a random element by finding and deleting the successor of a random value. Physical deletion is performed immediately by the deleting thread. Although this algorithm has no ordering semantics whatsoever, we consider it to be the performance ideal in terms of throughput scalability as it incurs almost no contention from deletion operations.

Wimmer et. al. k-Priority Queue. The relaxed k-Priority Queue given by Wimmer et. al. [28]. This implementation provides a linearizable priority queue, except that it is relaxed in the sense that each thread might skip up to k of the highest priority tasks; however, no task will be skipped by every thread. We test the hybrid version of their implementation as given in [28]. We note that this implementation does not offer scalability past 8 threads (nor does it claim to). Due to compatibility issues, we were unable to run this algorithm on the same framework as the others (i.e. Synchrobench). Instead, we show its performance on the original framework provided by the authors. Naturally, we cannot make direct comparisons in this manner, but the scalability trends are evident.

SprayList. The algorithm described in Section 2, which chooses an element to delete by performing a `Spray` with height $\lfloor \log p \rfloor + 1$, jump length uniformly distributed in $[1, \lfloor \log p \rfloor + 1]$ and padding length $p \log p/2$. Each thread becomes a *cleaner* (as described in Section 2.3) instead of `Spray` with probability $1/p$. Note that in these experiments, p is known to threads. Through testing, we found these parameters to yield good results compared to other choices. Physical deletion is performed only by cleaner threads. Our implementation is built on Keir Fraser's SkipList algorithm [13], described in the Appendix, using the benchmarking framework of Synchrobench[8].

4.1 Throughput

We measured throughput of each algorithm using a simple benchmark in which each thread alternates insertions and deletions, thereby preserving the size of the underlying data structure. We initialized each priority queue to contain 1 million elements, after which we ran the experiment for 1 second.

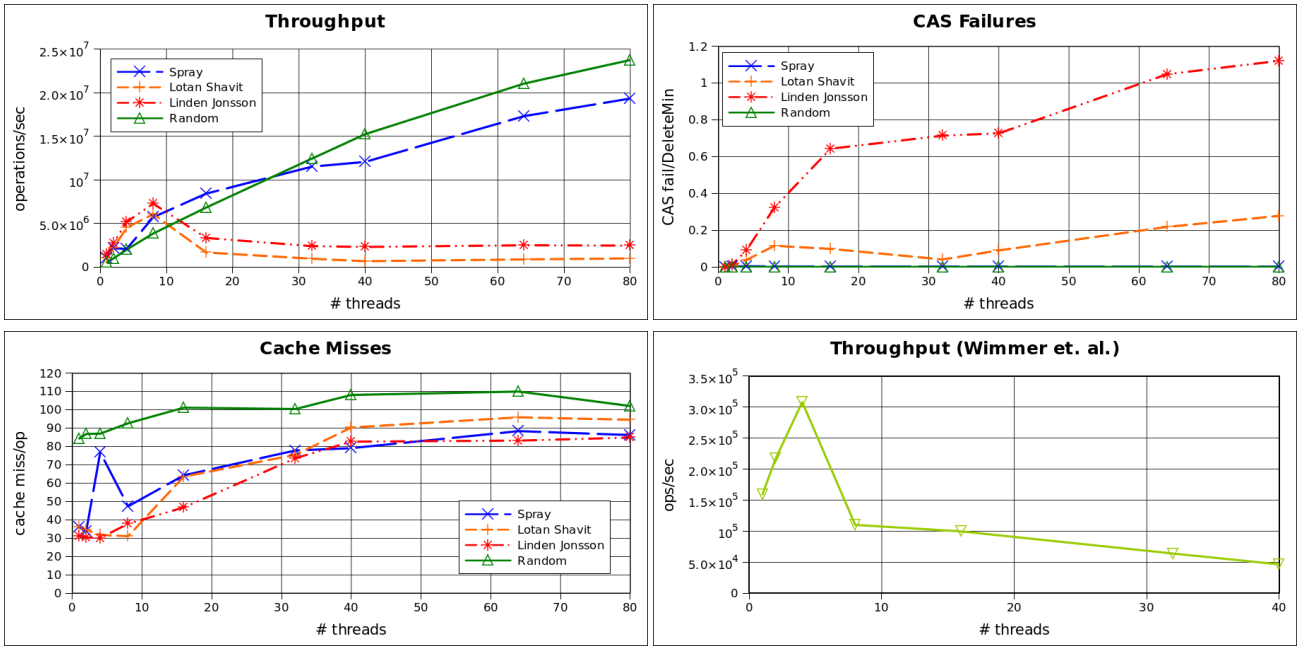

Figure 3: Priority Queue implementation performance on a 50% insert, 50% delete workload: throughput (operations completed), average CAS failures per DeleteMin, and average L1 cache misses per operation.

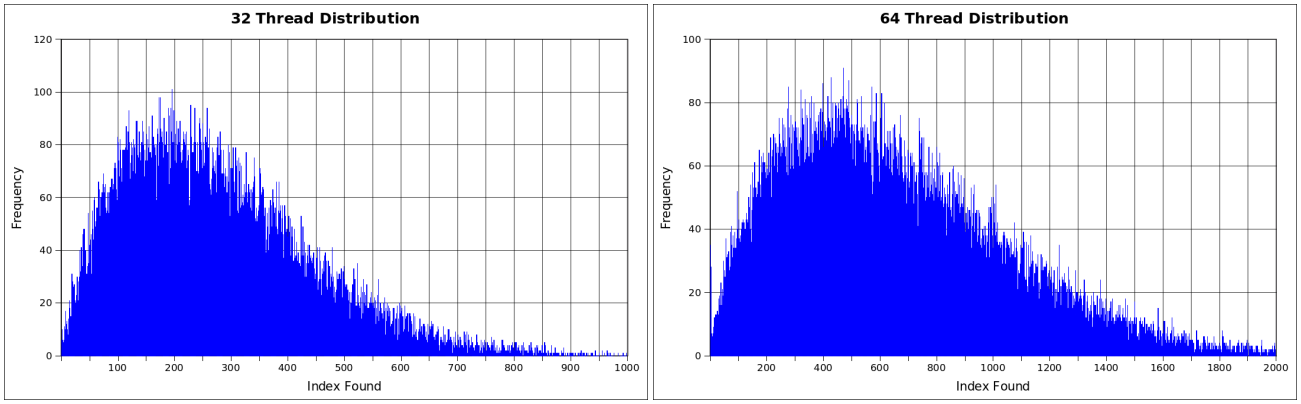

Figure 4: The frequency distribution of `Spray` operations when each thread performs a single `Spray` on a clean `SprayList` over 1000 trials. Note that the x-axis for the 64 thread distribution is twice as wide as for 32 threads.

Figure 3 shows the data collected from this experiment. At low thread counts ($\leq$ 8), the priority queue of Lindén et. al. outperforms the other algorithms by up to 50% due to its optimizations. However, like Lotan and Shavit's priority queue, Lindén's priority queue fails to scale beyond 8 threads due to increased contention on the smallest element. In particular, the linearizable algorithms perform well when all threads are present on the same socket, but begin performing poorly as soon as a second socket is introduced above 10 threads. On the other hand, the low contention random remover performs poorly at low thread counts due to longer list traversals and poor cache performance, but it scales almost linearly up to 64 threads. Asymptotically, the `SprayList` algorithm performs worse than the random remover by a constant factor due to collisions, but still remains competitive.

To better understand these results, we measured the average number of failed synchronization primitives per `DeleteMin` operation for each algorithm. Each implementation logically deletes a node by applying a (`CAS`) operation to the deleted marker of a node (though the actual implementations use `Fetch-and-Increment` for performance reasons). Only the thread whose `CAS` successfully sets the deleted marker may finish deleting the node and return it as the minimum. Any other thread which attempts a `CAS` on that node will count as a failed synchronization primitive. Note that threads check if a node has already been logically deleted (i.e. the deleted marker is not 0) before attempting a `CAS`.

The number of `CAS` failures incurred by each algorithm gives insight into why the exact queues are not scalable. The linearizable queue of Lindén et. al. induces a large number of failed operations (up to 2.5 per `DeleteMin`) due to strict safety requirements. Similarly, the quiescently consistent priority queue of Lotan and Shavit sees numerous `CAS` failures, particularly at higher thread counts. We observe a dip in the number of `CAS` failures when ad-

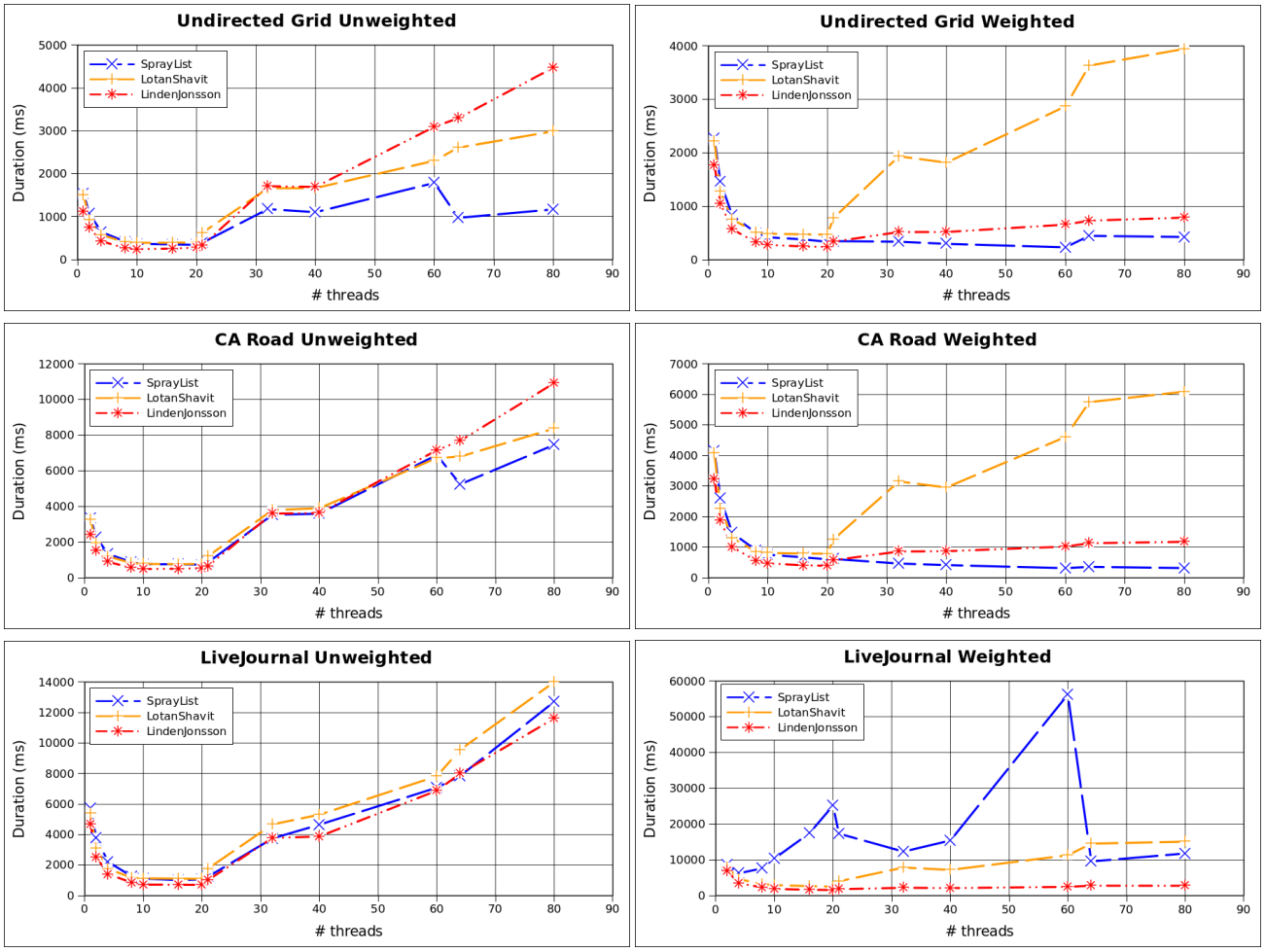

Figure 5: Runtimes for SSSP using each PriorityQueue implementation on each network (lower is better).

ditional sockets are introduced (i.e. above 10 threads) which we conjecture is due to the increased latency of communication, giving threads more time to successfully complete a `CAS` operation before a competing thread is able to read the old value. In contrast, the `SprayList` induces almost no `CAS` failures due to its collision avoiding design. The maximum average number of failed primitives incurred by the `SprayList` in our experiment was .0090 per `DeleteMin` which occurred with 4 threads. Naturally, the random remover experienced a negligible number of collisions due to its lack of ordering semantics.

Due to technical constraints, we were unable to produce a framework compatible with both the key-value-based implementations presented in Figure 3 and the task-based implementation of Wimmer et. al. However, we emulated our throughput benchmark within the framework of [28].

We implement tasks whose only functionality is to spawn a new task. Thus, each thread removes a task from the queue and processes that task by adding a new task to the queue. In this way, we measure the same pattern of alternating inserts and deletes in a task-based framework. As in the previous experiment, we initially populate the queue with 1 million tasks before measuring performance.

Figure 3 shows the total number of tasks processed by the k-priority queue of Wimmer et. al.[5] with $k = 1024$ over a 1 second duration. Similarly to the priority queue of Lindén et. al., the k-priority queue scales at low thread counts (again ≤ 8), but quickly drops off due to contention caused by synchronization needed to maintain the k-linearizability guarantees. Other reasonable values of k were also tested and showed identical results.

In sum, these results demonstrate that the relaxed semantics of `Spray` achieve throughput scalability, in particular when compared to techniques ensuring exact guarantees.

4.2 Spray Distribution

We ran a simple benchmark to demonstrate the distribution generated by the `Spray` algorithm. Each thread performs one `DeleteMin` and reports the position of the element it found. (For simplicity, we initialized the queue with keys $1, 2, \ldots$ so that the position of an element is equal to its key. Elements are not deleted from the `SprayList` so multiple threads may find the same element within a trial.) Figure 4 shows the distribution of elements found after 1000 trials of this experiment with 32 and 64 threads.

We make two key observations: 1) most `Spray` operations fall within the first roughly 400 elements when $p = 32$ and 1000 ele-

[5] We used the hybrid k-priority queue which was shown to have the best performance of the various implementations described [28].

ments when $p = 64$ and 2) the modal frequency occurred roughly at index 200 for 32 threads and 500 for 64 threads. These statistics demonstrate our analytic claims, i.e., that `Spray` operations hit elements only near the front of the list. The width of the distribution is only slightly superlinear, with reasonable constants. Furthermore, with a modal frequency of under 100 over 1000 trials (64000 separate `Spray` operations), we find that the probability of hitting a specific element when $p = 64$ is empirically at most about .0015, leading to few collisions, as evidenced by the low `CAS` failure count. These distributions suggest that `Spray` operations balance the trade-off between width (fewer collisions) and narrowness (better ordering semantics).

4.3 Single-Source Shortest Paths.

One important application of concurrent priority queues is for use in Single Source Shortest Path (SSSP) algorithms. The SSSP problem is specified by a (possibly weighted) graph with a given "source" node. We are tasked with computing the shortest path from the source node to every other node, and outputting those distances. One well known algorithm for sequential SSSP is Dijkstra's algorithm, which uses a priority queue to repeatedly find the node which is closest to the source node out of all unprocessed nodes. A natural parallelization of Dijkstra's algorithm simply uses a parallel priority queue and updates nodes concurrently, though some extra care must be taken to ensure correctness.

Note that skiplist-based priority queues do not support the decrease-key operation which is needed to implement Dijkstra's algorithm, so instead duplicate nodes are added to the priority queue and stale nodes (identified by stale distance estimates) are ignored when dequeued.

We ran the single-source shortest path algorithm on three types of networks: an undirected grid (1000×1000), the California road network, and a social media network (from LiveJournal) [1]. Since the data did not contain edge weights, we ran experiments with unit weights (resembling breadth-first search) and uniform random weights. Figure 5 shows the running time of the shortest paths algorithms with different thread counts and priority queue implementations.

We see that for many of the test cases, the `SprayList` significantly outperforms competing implementations at high thread counts. There are of course networks for which the penalty for relaxation is too high to be offset by the increased concurrency (e.g. weighted social media) but this is to be expected. The LiveJournal Weighted graph shows a surprisingly high spike for 60 cores using the `SprayList` which is an artifact of the parameter discretization. In particular, because we use $\rfloor \log p \lfloor$ for the `Spray` height, the `Spray` height for 60 cores rounds down to 5. The performance of the `SprayList` improves significantly at 64 cores when the `Spray` height increases to 6, noting that nothing about the machine architecture suggests a significant change from 60 to 64 cores.

4.4 Discrete Event Simulation

Another use case for concurrent priority queues is in the context of Discrete Event Simulation (DES). In such applications, there are a set of events to be processed which are represented as tasks in a queue. Furthermore, there are dependencies between events, such that some events cannot be processed before their dependencies. Thus, the events are given priorities which impose a total order on the events which subsumes the partial order imposed by the dependency graph. As an example, consider n-body simulation, in which events represent motions of each object at each time step, each event depends on all events from the preceding time step. Here, a total order is given by the time step of each event, along with an arbitrary ordering on the objects.

We emulate such a DES system with the following methodology: we initially insert 1 million events (labelled by an ID) into the queue, and generate a list of dependencies. The number of dependencies for each event i, is geometrically distributed with mean δ. Each event dependent on i is chosen uniformly from a range with mean $i + K$ and radius $\sqrt{K}$. This benchmark is a more complex version of the DES-based benchmark of [20], which in turn is based on the URDME stochastic simulation framework [11].

Once this initialization is complete, we perform the following experiment for 500 milliseconds: Each thread deletes an item from the queue and checks its dependants. For each dependant, if it is not present in the queue, then some other thread must have already deleted it. This phenomenon models an inversion in the event queue in which an event is processed with incomplete information, and must be reprocessed. Thus, we add it back into the queue. We call this early deletion and reinsertion *wasted work*. This can be caused by the relaxed semantics, although we note that even linearizable queues may waste work if a process stalls between claiming an event and actually processing it.

This benchmark allows us to examine the trade-off between the relaxed semantics and the increased concurrency of `SprayLists`. Figure 6 reports the actual work performed by each of the competing algorithms, where actual work is calculated by simply measuring the reduction in the size of the list over the course of the experiment, as this value represents the number of nodes which were deleted without being reinserted later and can thus be considered fully processed. For each trial, we set $\delta = 2$ and tested $K = 100, 1000, 10000$.

As expected, the linearizable priority queue implementation does not scale for any value of K. As in the pure throughput experiment, this experiment also presents high levels of contention, so implementations without scaling throughput cannot hope to scale here despite wasting very little work.

On the other hand, the `SprayList` also fails to scale for small values of K. For $K = 100$, there is almost no scaling due to large amounts of wasted work generated by the loose semantics. However, as K increases, we do start to see increased scalability, with $K = 1000$ scaling up to 16 threads and $K = 10000$ scaling up to 80 threads.

To demonstrate the dependence of scalability on the distribution of dependencies, we measured the minimum value of K needed to obtain maximum performance from a `SprayList` at each thread count. In particular, for each fixed value of n, we increased K until performance plateaued and recorded the value of K at which the plateau began.

Figure 7 reports the results of this experiment. We notice that the minimum K required increases near linearly with the number of threads. Note that the "bumps" at 40 and 80 threads due to the dependence of `Spray` width only on $\lfloor \log_2 n \rfloor$ (so the minimum K required will generally only increase at powers of 2). This plot suggests the required dependency sparsity in order for the `SprayList` to be a good choice of data structure for a particular application.

5. Discussion and Future Work

We presented a new design for a relaxed priority queue, which allows throughput scaling for large number of threads. The implementation weakens the strict ordering guarantees of the sequential specification, and instead provides probabilistic guarantees on running time and number of inversions. Our evaluation suggests that the main advantage of our scheme is the drastic reduction in contention, and that, in some workloads, the gain in scalability can fully compensate for the additional work due to inversions. We develop our technique on a lock-free SkipList, however a similar construct works for a lock-based implementation. Also, the relaxation parameters of our algorithm (spray height, step length) can be tuned depending on the workload.

An immediate direction for future work would be to tune the data structure for specific workloads, such as efficient traversals of

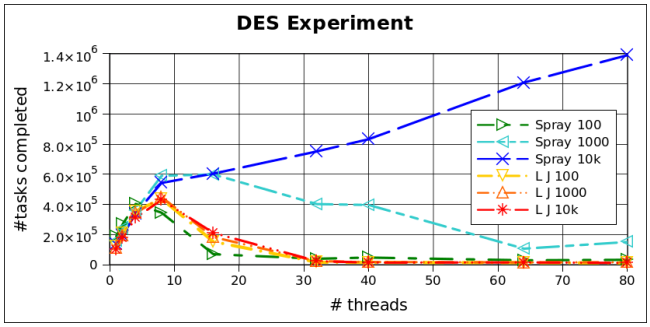

Figure 6: Work performed for varying dependencies (higher is better). The mean number of dependants is 2 and the mean distance between an item and its dependants varies between 100, 1000, 10000.

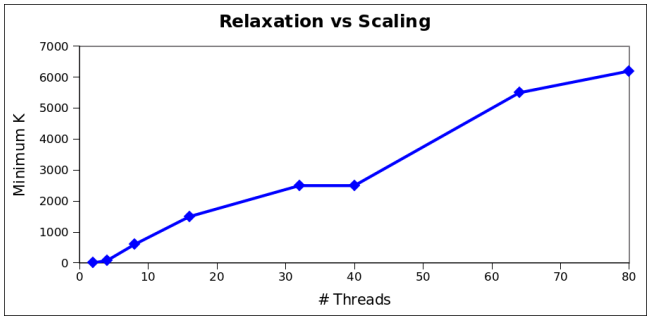

Figure 7: Minimum value of K which maximizes the performance of the `SprayList` for each fixed number of threads.

large-scale graphs. A second direction would be to adapt the spraying technique to obtain relaxed versions of other data structures, such as double-ended queues [16].

References

[1] Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`. Accessed: Sept. 2014.

[2] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. *Distributed Computing*, 26(4):243–269, 2013.

[3] D. Alistarh, J. Aspnes, S. Gilbert, and R. Guerraoui. The complexity of renaming. In *52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 718–727, Oct. 2011.

[4] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. Cafe: Scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 475–488, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] A. Braginsky and E. Petrank. A lock-free b+tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA*, pages 58–67, 2012.

[6] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1):4–21, 1998.

[7] I. Calciu, H. Mendes, and M. Herlihy. The Adaptive Priority Queue with Elimination and Combining. *ArXiv e-prints*, Aug. 2014.

[8] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. *ACM SIGPLAN Notices*, 47(8):161–170, 2012.

[9] N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, Mar. 1992.

[10] D. Dice, Y. Lev, and M. Moir. Scalable statistics counters. In *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada*, pages 43–52, 2013.

[11] B. Drawert, S. Engblom, and A. Hellander. Urdme: a modular framework for stochastic simulation of reaction-transport processes in complex geometries. *BMC Systems Biology*, 6(76), 2012.

[12] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[13] K. Fraser. *Practical lock-freedom*. PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[14] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[15] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, New York, NY, USA, 2013. ACM.

[16] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[17] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing, DISC 2008, Arcachon, France*, pages 350–364, 2008.

[18] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *J. ACM*, 40(3):765–789, 1993.

[19] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[20] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems*, pages 206–220. Springer, 2013.

[21] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.

[22] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[23] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[24] W. Pugh. Concurrent maintenance of skip lists. 1998.

[25] P. Sanders. Randomized priority queues for fast parallel access. *Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures*, 49:86–97, 1998.

[26] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[27] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[28] M. Wimmer, D. Cederman, F. Versaci, J. L. Träff, and P. Tsigas. Data structures for task-based priority scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2014)*, 2014. To appear.