Relaxed Concurrent Ordering Structures

by

Justin Kopinsky

B.S., University of Illinois at Urbana-Champaign (2012)
S.M., Massachusetts Institute of Technology (2014)

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

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .... Signature redacted
Department of Electrical Engineering and Computer Science
May 23, 2018

Certified by.............. Signature redacted ....
Nir Shavit
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... Signature redacted
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Relaxed Concurrent Ordering Structures

by

Justin Kopinsky

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Efficient implementations of concurrent *ordering* structures, including stacks, queues, and priority queues, have long been elusive due to an inherent bottleneck on the 'head' element. We argue that classical semantics which are easy to support in sequential settings are stronger than necessary for concurrent applications, and instead define new semantics for implementing *relaxed* ordering structures: relaxed structures need only return elements which are probabilistically *near* the head element.

This thesis demonstrates the effectiveness of relaxed semantics by formally defining a notion of k-relaxation which imposes behavior 'similar' to that of a structure which returns one of the k elements nearest the head uniformly at random. This behavior is encapsulated by two probabilistic criteria: *error boundedness*—a bound on the distance of a returned element from the head—and *fairness*—a bound on the number of operations an element has to wait before being returned by some thread.

We design, analyze, and implement k-relaxed algorithms in this model, showing both that they achieve good values of k in theory and that they exhibit empirically good performance on applications such as Single-Source Shortest Paths.

Finally, we introduce a general framework for using relaxed structures to schedule and execute a wide class of problems which can be formulated as a series of task executions with dependencies between tasks. Our framework provides a case study demonstrating that applications can use our model of relaxed data structures to prove that the extra work induced by reordering tasks is low in the settings that we consider. Empirically, our benchmarks show that the low overhead is more than offset by increased throughput, resulting in improved performance on tasks such as Maximal Independent Set compared to an exact scheduler.

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

Acknowledgments

I would like to thank Professor Nir Shavit for six years of enthusiastic guidance, always remaining optimistic in all endeavors.

Many thanks to Professor Dan Alistarh, who from the very beginning of the pursuit of my Ph.D. has provided an incredible amount of support, feedback, and guidance, practically acting as a second adviser.

Thanks also Professor Erik Demaine for support and guidance over many years of great courses and research endeavors.

Further thanks to Erik Demaine and also to Julian Shun for serving on my thesis committee and providing much appreciated feedback.

This thesis is based in part on joint works with Dan Alistarh, Jerry Li, Giorgi Nadiradze, and Trevor Brown, and thanks to all of them for great work and enjoyable collaboration.

Thanks of course to my loving and supportive parents, Mervyn and Ilana Kopinsky.

Finally, many thanks to MIT and the Department of Electrical Engineering and Computer Science for institutional funding and exceptional support.

Contents

List of Figures

Chapter 1

Introduction

High performance, scalable, concurrent data structures have been in increasingly high demand in the last several decades. In that time, efficient implementations of many data structures have been developed. On the other hand, there is a large class of commonly used data structures which can be classified as serving *producer-consumer* access patterns that suffer from sequential bottlenecks on the "consumer" threads. The canonical example of such a structure is the *priority queue.* Priority queues store key, value pairs, or *elements*, and support at least the following operations: INSERT(), which inserts arbitrary pairs into the queue, and GETMIN(), which outputs the pair with the *smallest key* (or equivalently, *highest priority*) and removes it from the queue. Any concurrent data structure supporting asynchronous INSERT() and GETMIN() according to priority queue semantics with a standard asynchronous correctness criterion (typically linearizability [41]) suffers an inherent bottleneck on the highest priority element. Indeed, if several threads concurrently execute a GETMIN() operation, they necessarily experience a race condition on retrieving the highest priority element in the queue. Such behavior imposes a seemingly impassable barrier to scalability.

However, we observe that even if queue operations are *linearizable*, perhaps the strongest correctness condition one can ask for, threads might still get reordered after completing their respective GETMIN()s, so that the thread which retrieved the true smallest key element in the queue stalls, thus appearing to the client application as

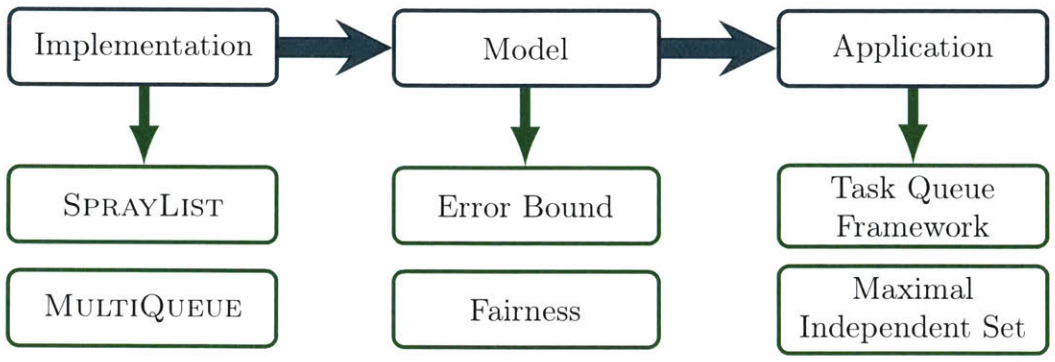

Figure 1-1: Our full stack approach to relaxed ordering structures. We give a model which real implementations can support and which practical applications can use.

though the GetMin() has not yet happened, even while threads which retrieved elements with larger keys continue running. Thus, we ask whether linearizability or similarly strong correctness conditions are useful in this setting.

A large body of past work has proposed *relaxed* semantics for concurrent data structures, wherein operations are allowed to return items which could not be returned by a corresponding sequential structure supporting *exact* correctness, but which are somehow 'close' to correct. For example, a simple criterion could be that the returned element must have at most the k^{th} smallest key in the system. Perhaps the first example of this idea was the relaxed PRAM priority queue of Karp and Zhang [45], but there have been many proposals since to produce scalable relaxed data structures on modern architectures, e.g. [1, 6, 9, 36, 39, 46, 49, 58, 64, 65, 67, 68, 74].

However, this line of study has lacked a foundation which is simultaneously *practical* and *theoretically rigorous.* On the one hand, although some formal models have been proposed, e.g. *Quasi-linearizability* [1] and the quantitative framework of Henzinger et al. [39], it is not yet clear whether performant structures can be built to support the semantics of these models, nor whether *applications* can make use of these guarantees to bound the overhead they incur when using relaxed structures. On the other hand, a number of relaxed structures have been proposed which are empirically scalable [6, 64, 65] but these practical structures did not initially adhere to any unified formal model of relaxation.

This thesis will build a case for the use of *randomized relaxation* for priority queues by a three-pronged approach: (1) we propose a *model* for probabilistically quantifying the relaxation of ordering structures which is both (2) demonstrably applicable to real, performant, relaxed ordering structures, allowing for theoretically grounded bounds on the effect of relaxation, and (3) *usable* by applications to prove at the top level that the cost incurred by weakening semantics incurs a bounded overhead in the form of dependency checking and possibly wasted work. We stress that our quantification criteria provide an interface between implementation and application, allowing application designers to abstract the minutae of the distribution of outputs particular relaxed implementations might provide, instead relying on algorithm designers to provide simple bounds that applications can make use of.

Figure 1-1 gives a visual representation of our approach. First, we build a model characterized by two parameterized criteria imposed on relaxed priority queues: *ϕ-error-boundedness* and *ψ-fairness*. We say that a queue which is both k-error bounded and k-fair is k-relaxed (equivalently, has relaxation factor k). Intuitively, a k-relaxed queue should exhibit behavior which is distributionally 'similar to' a queue which returns an item uniformly at random from the k highest priority items (and indeed, such a queue is itself k-relaxed); see below for more discussion and see Chapter 2 for a formal definition. Using this model, we give two implementations of relaxed queues which satisfy these criteria with good parameters while providing high throughput in contended executions (in particular, $k = O(n \log n)$ when there are n participating threads): first, the SPRAYLIST, a centralized queue based on asynchronous SkipLists (Chapter 3), and second, MULTIQUEUEs, a fully distributed algorithm based on classical 'balls into bins' systems (Chapter 4). Finally, we construct a general framework to solve problems which can be formulated as an execution of a series of tasks with known dependencies between tasks (Chapter 5). We give analytical and empirical results showing that relaxed queues are a good choice of scheduler for such problems when tasks can be permuted randomly and the dependency graph is *sparse*. More interestingly, we show that in the case of Maximal Independent Set and Maximal Matching, the overhead incurred by correcting for relaxation compared to using an exact (non-relaxed) scheduler is, surprisingly, *negligible*: there is *no* dependence on the size or structure of the input graph, rather only on the relaxation factor k of the scheduler. It follows that these applications are examples of an ideal use case for relaxed queues, as they benefit significantly from the increased throughput offered by relaxed implications while suffering negligibly from the weaker semantics.

In addition to being themselves widely used in applications such as scheduling and event simulation [51], priority queues can be thought of as generalizing a large class of structures which we will collectively refer to as *ordering structures*. Commonly used ordering structures include FIFO queues, priority queues, stacks, and in an abstract sense, counters. There has been a large amount of work on building performant concurrent implementations of these data structures with exact semantics [18, 28, 29, 31, 32, 40, 47, 51, 52, 53, 56, 57, 61, 62, 70, 72]. Both the SprayList and MultiQueue algorithms can be trivially adapted to support Queue or Stack operations and we correspondingly propose that using relaxed implementations of queues and stacks which are *scalable* can be similarly beneficial for applications which are robust to weaker semantics. Indeed, although the Task Queue Framework described in Chapter 5 is formulated using priority queues for generality, we only make use of FIFO queue semantics for the cases we analyze, particularly Maximal Independent Set.

The problem with exact semantics. In the classic, sequential setting, priority queues are very well understood. Highly efficient, *heap*-based implementations are highly optimized and used pervasively in the field [48]. Unfortunately, heap-based *concurrent* priority queues suffer from both memory contention and sequential bottlenecks. Heaps are implemented by perfectly balanced binary trees, and every heap operation must touch and usually modify the root. Because the heap element with the smallest key is stored in the root, every thread attempting to perform a GetMin() operation must modify it. Furthermore, all modifying heap operations including insertion are implemented by 'percolating' elements up the tree, swapping values at all levels, inducing more communication even below the root. Modifying threads inevitably clash with every other thread in the system, creating a seemingly insurmountable barrier to scalability.

Early attempts to improve on heaps in a concurrent setting looked to *Skiplists*, an idea which has maintained momentum for many years [18, 51, 53, 70]. SkipLists are randomized list-based data structures which classically support INSERT and DELETE operations [61]. A SkipLists is composed of several linked lists organized in levels, each containing a random subset of the elements in the list below it. 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 [32, 41, 51, 62]. Unfortunately, concurrent SkipList-based priority queues which maintains a linearizable [41] (or even quiescently-consistent [41]) order on GETMIN() operations, must still remove the minimal element from the leftmost node in the SkipList. Thus, as with heaps, all threads must repeatedly compete to get this minimal node, resulting in a contention bottleneck and limiting scalability [51].

Recently, there has been another attempt to circumvent thread clashing and contention issues with a technique called *flat combining*, or just *combining* [20, 24, 28, 37, 38]. The idea of combining is to have a single thread batch the requests of all the other threads and execute them itself. Priority Queues implemented using heaps with combining were shown to significantly outperform state-of-the-art implementations [37] by effectively immunizing the combining thread from cache invalidations and synchronization failures induced by other threads. However, due to the inherently sequential nature of the technique, combining-based data structures *still don't scale.*

In fact, the situtation is about as bad as it could be: ordering structures are *fundamentally limited* by impossibility results [2, 27] showing that the strong semantics required by these data structures comes with an *inherent* lack of scalability. These theoretical results suggest that in order to achieve truly scalable implementations of producer-consumer data structures, we need to consider alternative semantics. To that end, recent work has begun considering *relaxed* data structure specifications [39, 68], for which weaker semantics are accepted in a controlled manner in order to remove the theoretical bottleneck and substantially improve performance in contended workloads. Importantly, allowing relaxed semantics allows data structure implementations to make effective use of *randomization* to break symmetry between threads and thereby reduce contention.

Model. We begin by presenting a model of randomized, relaxed, concurrent ordering structures. In particular, relaxed ordering structures implement ApproxGetMin() in place of GetMin(), which might return elements other than the highest priority element. We give two probabilistic criteria which we use to *quantify* the strength of the relaxed semantics of a given implementation. Our criteria are parameterized by a *relaxation factor* k so that if a particular implementation satisfies them for a given k, we say that the implementation is k-relaxed. The model is designed to allow complex implementations to provide similar guarantees to the 'simplest' specification of a relaxed queue for which ApproxGetMin() simply returns an element uniformly at random from among the k highest priority elements. Such a uniformly random implementation is $\Theta(k)$-relaxed in our model.

The first criterion asks how large the *error* of a relaxed operation might be; that is, how large the *rank* of the element returned by ApproxGetMin() might be (a key has rank i if there are exactly $i-1$ smaller keys in the structure at a given time). The second criterion asks how long a particular high priority element can remain in the queue. Even if we can bound the error of an implementation, it might still not be very useful if the highest priority element is 'starved' while potentially having many dependents waiting for it to be processed. We will say that an implementation is *k-relaxed* if it is both *k-error bounded* and *k-fair*, which are respectively defined to mean that the implementation satisfies a simple tail bound, defined in Section 2.3.

Building off of the model, we will present and analyze two relaxed strategies for concurrent queues, priority queues, and counters which yield state of the art performance. We will focus on showing analytically that the relaxation factor as described by our model is low, while also demonstrating state-of-the-art performance on practical workloads.

The SprayList. Our first implementation is the SprayList, which implements a relaxed concurrent priority (or FIFO) queue by building off of existing efficient imple-

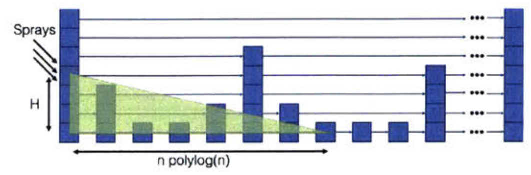

Figure 1-2: The intuition behind the SPRAYLIST. Threads start at height H and perform a random walk on nodes at the start of the list, attempting to acquire the node they land on.

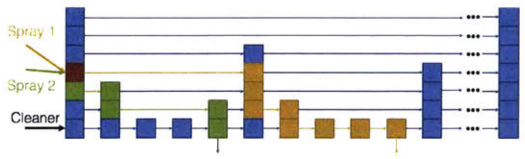

Figure 1-3: A simple example of a spray. Green nodes are touched by the SPRAY, and the thread stops at the red node. Orange nodes could have been chosen for jumps, but were not.

mentations of concurrent SkipLists [32, 34, 61, 62]. The main idea of the SPRAYLIST is to have threads take a controlled random walk down a SkipList, staying near the front, in order to do a relaxed GETMIN() operation. Instead of threads clashing on the first element, we allow threads to "skip ahead" in the list, so that concurrent operations attempt 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 GETMIN() operations traverse the SkipList, not along the list, but via a tightly controlled random walk from its head. 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 when there are n threads, the probabilities of hitting nodes among the first $O(n \log^3 n)$ are close to uniform. (See Figure 1-2 for the intuition behind sprays.)

While a GETMIN() in an exact priority queue returns the element with the smallest key—practically one of the n smallest keys if n threads are calling GETMIN() concurrently— the SPRAYLIST ensures that the returned key is among the $O(n \log^3 n)$ smallest keys, providing error-boundedness, and that each operation completes within $\log^3 n$ steps, both with high probability. We also provide fairness guarantees, i.e. that elements with small keys will not remain in the queue for too long. Chapter 3 gives a rigorous treatment of SPRAYLISTS, giving methodology which simultaneously achieves good runtime and relaxation factor, both in theory and on practical workloads.

Our proofs are inspired by an elegant argument proving that sprays are near-uniform on an *ideal* (uniformly-spaced) SkipList, given in Section 3.2.2. However, this argument breaks on a realistic SkipList, whose structure can be quite irregular. Precisely bounding the node hit distribution on a realistic SkipList turns out to be significantly more involved.[1] In particular, we prove that this distribution is close to uniform on the first $O(n \log^3 n)$ elements. We can then upper bound the probability that two sprays collide, and the expected number of operation retries. In turn, this upper bounds the running time of an operation and the relative rank of returned keys.

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. The trade-off between relaxed semantics and thread contention can be controlled via the SPRAY parameters (starting height, and jump length). We also give a simple back-off scheme which allows threads to "tighten" the semantics under low contention.

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.

MULTIQUEUES. In a seminal paper [8], Azar, Broder, Karlin and Upfal analyzed the following elegant load balancing process: we sequentially place b balls into m initially empty bins, where each ball has two destinations, chosen independently and uniformly at random. Each ball is placed into the less loaded of its chosen bins. Surprisingly, the difference between the most loaded bin and the average is $O(\log \log m)$ with high probability, and this difference remains stable as the process executes for increasingly many steps. The strong probabilistic guarantees provided by this process sparked an impressive amount of follow-up research, combining deep and elegant theoretical results on extensions of the process and their analyses, e.g. [55, 60] with non-trivial practical applications [63]. In particular, it is known [10, 60] that the *gap* between the most loaded bin and average load is $O(\log \log m)$, while the gap between the

[1]We perform the analysis in a restricted asynchronous model, defined in Section 3.2.1.

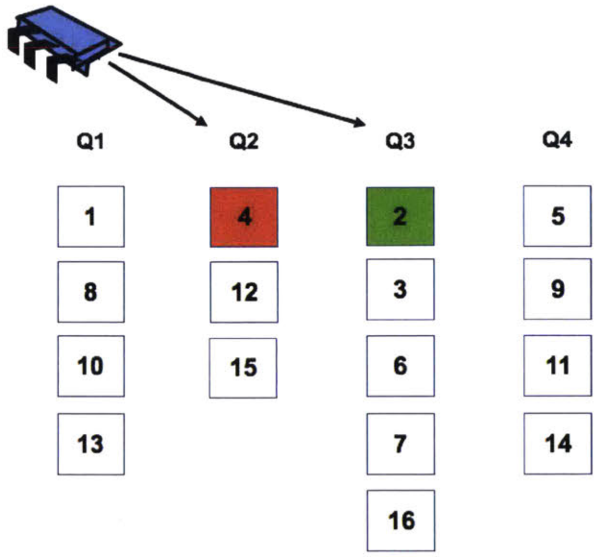

Figure 1-4: An illustration of the MULTIQUEUE process. On a GETMIN(), the thread looks at two queues and selects the one with a lower label top element. In this case, the thread looks at queues 2 and 3 and selects queue 3 (green), rejecting queue 2 (red).

least loaded bin and the most loaded one is bounded by $O(\log m)$ in expectation, independently of how long the process runs.

These results suggest a simple alternative strategy for implementing relaxed concurrent priority queues, obtained by the following MULTIQUEUE strategy [35, 64]. We start from n queues, each protected by a lock. To *insert* an element, a thread picks one of the n queues uniformly at random, locks it, and inserts into it. To *remove* an element, the thread picks *two queues* uniformly at random, locks them, and removes the element of *higher priority* among their two top elements. Note that this strategy can be easily adapted to implement FIFO queues or approximate counters as well. This strategy is illustrated in Figure 1-4.

Extensive testing [35, 64] has shown that this natural strategy can provide state-of-the-art throughput, and that the average number of priority inversions is relatively low. Further variants of this strategy are used to implement general priority schedulers [58] and relaxed concurrent queues [36].

Despite its apparent efficacy, the original MULTIQUEUE proposal demonstrated promising empirical results but lacked a theoretical grounding for the robustness of the technique. In fact, it seemed doubtful that MULTIQUEUEs provided bounded relaxation at all. Even with the insight of deleting from the higher priority of two queues, one still might expect the state of the MultiQueue system to deteriorate over time, eventually ending up with a potentially unbounded gap between the highest priority element in the system, and the top element of the 'worst' queue. We rectify this concern by showing that even in the MultiQueue setting in which elements are inserted into queues randomly, analogs of the balls into bins results still hold and MultiQueue ApproxGetMin() operations exhibit at most $O(n \log n)$ rank error with high probability. Chapter 4 covers an analysis of MultiQueues in a sequential setting, where queue operations take place atomically. Upcoming work extends the analysis to a fully asynchronous setting [3], but is beyond the scope of this thesis.

The main technical contribution of the chapter is showing that MultiQueues provide surprisingly strong rank guarantees: for any time $t \geq 0$ in the execution of the MultiQueue algorithm, the expected rank of an element removed at time t is $O(n)$, while the expected worst-case rank removed at a step is $O(n \log n)$. These bounds are asymptotically tight, and hold for arbitrarily large t. Our analysis generalizes to a $(1 + \beta)$ extension of the process, where the algorithm deletes from the higher priority among two random queues with probability $0 < \beta < 1$, and from a single randomly chosen queue with probability $(1 - \beta)$. It also extends to show that the process is *robust to bias* in the insertion distribution towards some bins by a constant factor $\gamma \in (0, 1)$. By contrast, we show that the strategy which always removes from *a single* randomly chosen queue *diverges*, in the sense that its average rank guarantee evolves as $\Omega\left(\sqrt{tn \log n}\right)$, for time $t \geq n \log n$.

Application: Task-Queue Framework. Given the now-pervasive nature of parallelism in computation, there has been a tremendous amount of research into efficient parallel algorithms for a wide range of tasks. A popular approach has been to map existing sequential algorithms to parallel architectures, by exploiting their *inherent parallelism.* The *deterministic* approach, e.g. [12, 13, 14, 15, 44, 69] has been to study the directed-acyclic graph (DAG) dependency structure in classic, widely-employed sequential algorithms, showing that, perhaps surprisingly, this dependence structure usually *has low depth.* One can then design schedulers which exploit this

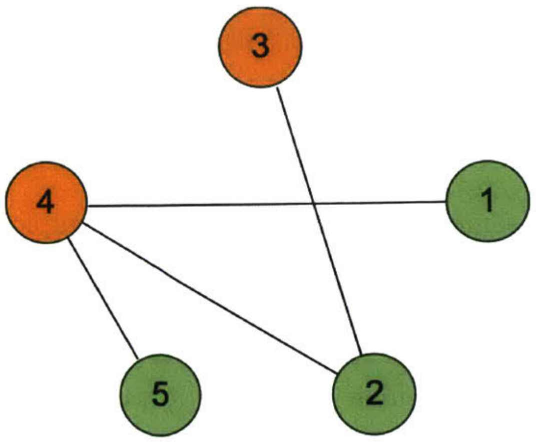

Figure 1-5: A simple example of Maximal Independent Set. Vertex labels were assigned randomly and the lexicographically first MIS is shown; vertices in the MIS are green and vertices not in the MIS are orange.

dependence structure for efficient execution on parallel architectures. As the name suggests, this approach ensures deterministic outputs, and can yield good practical performance [14], but requires a non-trivial amount of knowledge about the problem at hand, and the use of carefully-constructed parallel schedulers [14].

To illustrate, let us consider the classic sequential greedy strategy for solving the maximal independent set (MIS) problem on arbitrary graphs: the algorithm examines the set of vertices in the graph following a fixed, random sequential priority order, adding a vertex to the independent set if and only if no neighbor of higher priority has already been added. This effectively computes the *lexicographically first* MIS with respect to the random priority order. Figure 1-5 gives a small example. The basic insight for parallelization is that the outcome at each node may only depend on a small subset of other nodes, namely its neighbors which are higher priority in the random order. Blelloch, Fineman and Shun [14] performed an in-depth study of the asymptotic properties of this dependence structure, proving that, for any graph, the maximal depth of a chain of dependences is in fact $O(\log^2 n)$ with high probability, where n is the number of nodes in the graph. Recently, an impressive analytic result by Fischer and Noever [30] provided tight $\Theta(\log n)$ bounds on the maximal dependency depth for greedy sequential MIS, effectively closing this problem for MIS. Beyond greedy MIS, there has been significant progress in analyzing the dependency structure of other fundamental sequential algorithms, such as algorithms for matching [14], list contraction [69], Knuth shuffle [69], linear programming [12], and graph connectivity [12].

On the other hand, some work has already begun employing relaxed schedulers to do the same job. Starting with Karp and Zhang [45], the general idea is that, in some applications, the scheduler can relax the strict order induced by following the sequential algorithm, and allow tasks to be processed speculatively ahead of their dependencies, without loss of correctness. A standard example is parallelizing Dijkstra's single-source shortest paths (SSSP) algorithm, e.g. [49, 58, 65]: the scheduler can retrieve vertices in relaxed order without breaking correctness, as the distance at each vertex is guaranteed to eventually converge to the minimum. The trade-off is between the performance gains arising from using simpler, more scalable schedulers, and the loss of determinism and the wasted work due to relaxed priority order. Lenharth, Nguyen, and Pingali [49] further propose to use relaxed schedulers for Minimum Spanning Tree, Betweenness Centrality, Maximum Bipartite Matching, and Belief Propagation, bringing relaxed scheduling to an impressive repertoire of empirically efficacious use cases. This approach is quite popular in practice, as several high-performance relaxed schedulers have been proposed, which can attain state-of-the-art results in settings such as graph processing and machine learning [33, 58]. At the same time, despite good empirical performance, relaxed schedulers still lack analytical bounds, and outputs are not always deterministic. This leads to the natural question: is it possible to achieve both the simplicity and good performance of relaxed schedulers *as well as* the predictable outputs and runtime upper bounds of the "deterministic" approach?

We answer in the affirmative: we demonstrate that relaxed ordering structures in fact can execute a range of iterative sequential algorithms *deterministically*, i.e. with output uniquely determined by the input, and *provably efficiently*, providing analytic upper bounds on the total work performed. Our results cover the classic greedy sequential graph algorithms for *maximal independent set (MIS)*, *matching*, and *coloring*, as well as algorithms for *list contraction* and *generating permutations* via Knuth shuffle. We call this class *iterative algorithms with explicit dependencies.* Our key technical result is that, for MIS and matching in particular, we upper bound the overhead of using a relaxed scheduler by a polynomial term depending only on the relaxation factor k of the scheduler and which is *independent of the input graph size or structure.* Thus, computing MIS on large graphs using a relaxed scheduler incurs *negligible* overhead. This analytical result suggests that relaxed schedulers should be a viable alternative, a finding which is also supported by our concurrent implementation.

Specifically, we consider the following framework. Given an input, e.g., a graph, the sequential algorithm defines a set of *tasks*, e.g. one per graph vertex, which should be processed in order, respecting some fixed, arbitrary data dependencies, which can be specified as a DAG. Tasks will be accessible via a *relaxed scheduler.* This induces a sequential model,[2] where at each step, the scheduler returns a new task.

Assume a thread receives a task from the scheduler. Crucially, the thread *cannot process the task* if it has data dependencies *on higher-priority tasks*: this way, determinism is enforced. (We call such a failed removal attempt by the thread a *wasted* step.) However, threads are free to process tasks which do not have such outstanding dependencies, potentially out-of-order (we call these *successful* steps.) We measure *work* in terms of the total number of scheduler queries needed to process the entire input, including both successful and unsuccessful removal steps.

We provide a simple yet general approach to analyze this relaxed scheduling process, by characterizing the interaction between the dependency structure induced by a given problem on an arbitrary input, and the relaxation factor k in the scheduling mechanism, which yields bounds on expected work when executing such algorithms via relaxed schedulers. Our approach extends to general iterative algorithms, as long as task dependencies are *explicit*, i.e., can be statically expressed given the input, and tasks can be randomly permuted initially.

The work efficiency of this framework will critically depend on the rate at which

[2]We consider this sequential model, similar to [14], since there currently are no precise ways to model the contention experienced by concurrent threads on the scheduler. Instead, we validate our findings via a fully concurrent implementation.

threads are able to successfully remove dependency-free tasks. Intuitively, this rate appears to be highly dependent on (1) the problem definition, (2) the scheduler relaxation factor k, but also on (3) the structure of the input. Indeed, we show that in the most general case, a k-relaxed scheduler can process an input described by a dependency graph G on n nodes and m edges and incur $O(\frac{m}{n}\text{poly}(k))$ wasted steps, i.e. $n + O(\frac{m}{n}\text{poly}(k))$ total steps. This result immediately implies a low "cost of relaxation" for problems whose dependency graph is inherently *sparse*, such as Greedy Coloring on sparse graphs, Knuth Shuffle and List Contraction, which are characterized by a dependency structure with only $m = O(n)$ edges. Hence, in general, such sparse problems incur negligible relaxation cost when $k \ll n$.

Our main technical result is a counter-intuitive bound for greedy MIS: our framework equipped with a k-relaxed scheduler can execute greedy MIS on *any* graph G and experience only poly(k) wasted steps (i.e. $n + \text{poly}(k)$ total steps), *regardless of the size or structure of* G. This result is surprising as well as technically non-trivial, and demonstrates that for MIS on large graphs, operation-level speedups provided by relaxation come with a negligible global trade-off. A similar result holds for maximal matching. Our results suggest that *task priorities* can be supported in a scalable manner, through relaxation, without loss of determinism or work efficiency.

We validate our results empirically, by implementing our scheduling framework in C++, based on a lock-free extension implementation of the MultiQueue algorithm [64]. Our broad finding is that this relaxed scheduling framework can ensure scalable execution, with small overheads due to contention and verifying task dependencies. For instance, when solving MIS on large graphs, we obtain a scalable solution, which has 5.7x speedup at 24 threads.

Summary. We give a full stack argument for the effectiveness of relaxed ordering structures, summarized in Figure 1-1. Our model characterizes ordering structures by an *error bound* and a *fairness* parameter. We show that real implementations can satisfy these criteria with good relaxation factors by building the SprayList and analyzing MultiQueues. Finally, we demonstrate that this model is *useful* to applications by designing the generic Task Queue Framework and instantiating it for

Maximal Independent Set, then using our model to get tight bounds on the number of parallel iterations required for each.

Chapter 2

Model

In this chapter we will build a model of the *relaxed concurrent ordering structures*, defining each term. Section 2.1 covers concurrency, Section 2.2 covers ordering structures, and Section 2.3 covers relaxation.

2.1 Concurrency

Throughout this thesis, we will consider a standard asynchronous shared-memory model such as in [7, 41], in which n *threads* which we will label $T_1, \ldots, T_n$, communicate through *registers*. Threads may perform atomic operations such as READ, WRITE, COMPARE-AND-SWAP and FETCH-AND-INCREMENT. Operations are applied according to a discrete *schedule*: at each time step, some thread is chosen to perform an operation, and it does so atomically. In general, schedules can be arbitrary and opaque to participating threads, beyond what can be inferred from the return values of operations. This chapter will formally define all of the above concepts.

To begin the formalization, the fundamental unit of shared memory is a register:

Definition 2.1 (Registers and values). *A* register *is a memory location which contains a* value. *Registers may have an initial value specified. Threads can perform specified operations on registers, possibly changing the stored value.*

Throughout this thesis, we will primarily concern ourselves with the following operations on a register R:

1. READ(R): return the value stored in R without changing it.

2. WRITE(R, x): change the value stored in R to x (without reading it).

3. FETCH-AND-INCREMENT(R): increment the (integer) value stored in R and return its old value. We will often abbreviate FETCH-AND-INCREMENT to FAI.

4. COMPARE-AND-SWAP(R, x, y): If the value of R is equal to x, change it to y and return true. Otherwise, return false. See Algorithm 1 for pseudocode. We will often abbreviate COMPARE-AND-SWAP(x, y) to CAS.

All modern multi-core processors support these instructions atomically.

Algorithm 1 Pseudocode for the COMPARE-AND-SWAP operation.

```
Register $R$ with value $v_R$.
function CAS(R,x,y) atomic:
    if $v_R = x$ then
        $v_R \leftarrow y$
        return TRUE
    else
        return FALSE
```

Processes run *algorithms* to interact with registers and thereby communicate with each other:

Definition 2.2 (Shared memory algorithms). *A shared memory* algorithm *is a sequence of register operations executed by a single thread, possibly depending on the outputs of previous operations and possibly depending on the results of random coin flips.*

However, threads cannot depend on being allowed to take interleaving steps in any sort of predictable fashion. This is true even in practice due to phenomena which range from microsecond race conditions in the hardware, to varying levels of cache misses or even page faults, and all the way to system intervention. It is impossible to model such a chaotic system precisely, so we instead measure asymptotic performance against an *adversary*:

Definition 2.3 (Adversarial schedulers). *As threads execute their algorithms, the order of thread steps is controlled by an adversarial entity we call the* scheduler.

The time t is measured in terms of the number of shared-memory steps scheduled by the adversary. The adversary may choose to crash a set of at most $n-1$ threads by not scheduling them for the rest of the execution. A thread that is not crashed at a certain step is *correct*, and if it never crashes then it takes an infinite number of steps in the execution. For this thesis, we will assume an *oblivious* adversarial scheduler, which decides on the interleaving of thread steps independently of the coin flips they produce during the execution.[1]

The algorithms we consider are implementations of shared objects:

Definition 2.4 (Shared objects). *A* shared object *O is an abstraction providing a set of* methods, *each given by a sequential specification.*

In particular, an implementation of a method M for object O is a set of n algorithms, one for each executing thread. When thread P_i invokes method M of object O, it follows the corresponding algorithm until it receives a response from the algorithm. Upon receiving the response, the thread is immediately assigned another method invocation. We will not distinguish between a method M and its implementation. A method invocation is *pending* at some point in the execution if has been initiated but has not yet received a response. A pending method invocation is *active* if it is made by a *correct* thread (note that the thread may still crash in the future). For example, a concurrent counter could implement READ and INCREMENT methods, with the same semantics as those of the sequential data structure.

The standard correctness condition for concurrent implementations is *linearizability* [42]:

Definition 2.5 (Linearization). *Consider an execution, E, consisting of invocations and responses of the methods of a concurrent object O. A* linearization *of E is an*

[1]By contrast, one might consider a *strong* adversary which can adapt its scheduling decisions based on the results of coin flips threads have made in the past, but we will not be working in such a model.

ordering $\prec_E$ of the invocations and responses such that, (1) every invocation is immediately succeeded by its response under $\prec_E$, (2) the responses are consistent with the sequential semantics of O under $\prec_E$, and (3) for every response R preceding some invocation I in E, $R \prec_E I$.

Definition 2.6 (Linearizability). *An object, O is* linearizable *if any concurrent execution of the methods of O has a* linearization.

In short, linearizability induces a global order on the method calls, which is guaranteed to be consistent to a sequential execution in terms of the method outputs. We will sometimes refer to the *linearization point* of a method, M, which is the step in the execution of M in which M is appended to the linearization in question (i.e. the step at which M is *linearized*). Notably, each linearization point must occur between the start and end time of the corresponding method, and in fact we can often point to a specific line of code (or perhaps a small set of possible such lines) which will always act as the linearization point of M.

2.2 Ordering Structures

An *ordering* structure is any object, O, which maintains a set, S, of ⟨key, value⟩ pairs and provides at least the following GetMin() method. For simplicity, we will assume throughout that values are unique. We will sometimes say that elements with *smaller keys* have *higher priority* and elements with *larger keys* have *lower priority*.

GetMin(): If S is empty, return $\perp$. Otherwise, output $\arg\min_{\langle k,v\rangle \in S} k$ and (possibly) remove it from S.

With just this method, the semantics of S are characterized by how the set of ⟨key, value⟩ pairs is generated, typically with the help of additional supported operations. Most ordering structures provide an Insert() method, whose arguments depend on the semantics of O, which is used to populate S:

INSERT$(\cdots)$**:** Insert a pair $\langle k, v\rangle$ into S.

For example, a *queue* maintains user-input values paired with keys which given by insertion time while a *stack* also maintains user-input values, but its keys are *negative* insertion time.

A slightly more abstract application of this model allows us to express *counters* as an ordering structure. We can think of a counter as initially populated with $S = \{\langle i, i\rangle | i \in \mathbb{N}\}$. *Incrementing* the counter consists of removing the minimum value $\langle i, i\rangle$ from S, so that *reading* a counter can be implemented by GETMIN(), i.e. returning the value of the counter as the smallest value still in S. Note that in this case, GETMIN() does not remove the value read from S. While this may seem like a rather roundabout formulation of counters, especially since they would never be implemented this way, we will see why this is a useful model in Section 2.3 where we develop a unified set of criteria for *relaxed* ordering structures.

Importantly, observe that almost all ordering structures can be implemented by a *priority queue*. A priority queue is itself an ordering structure which maintains user-input values paired with user-input keys. Therefore, for any structure O for which $\langle$key, value$\rangle$ pairs can be efficiently computed at insertion time, a black box priority queue is sufficient to implement O. This is particularly relevant for our purposes as state-of-the-art relaxed concurrent priority queues such as those discussed in Chapters 3 and 4 also represent state-of-the-art relaxed, concurrent FIFO queues. For this reason, we will use the terms "priority queue" and "ordering structure" interchangeably.

2.3 Relaxation

Recent work, e.g. Henzinger et al. [39], considers relaxed variants of linearizability, in which methods are allowed to deviate from the sequential specification by a *relaxation* factor. Such relaxations appear to be necessary in the case of data structures such as exact counters or priority queues in order to circumvent strong linear lower bounds on their concurrent complexity [2]. For example, one implementation of a relaxed concurrent counter might provide read methods which may return a value that is a constant additive factor ahead or behind the number of update operations which have been applied (linearized) up to some point in time.

Henzinger et al. [39] proposed a formal model of relaxation for *general deterministic* data structures. While their model is very powerful, it precludes the use of *randomization.* Randomized concurrent data structures have become increasingly popular [68], due largely to the ability to use randomization to break symmetry between threads. In fact, randomized concurrent data structures have been shown to be able to circumvent some deterministic lower bounds, e.g. [26, 59]. As relaxed data structures utilizing randomization will be a focal point of this thesis, we will choose to instead use a simpler model tailored toward ordering structures which has the desirable properties of both being achievable by scalable and performant implementations of ordering structures and being sufficiently strong that we can get good bounds on the overhead incurred by algorithms which use structures in our model, as we will show in Chapter 5.

We say that O is a *relaxed* ordering structure if it implements ApproxGetMin():

ApproxGetMin(): If S is empty, return $\perp$. Otherwise, output some $\langle k, v\rangle \in S$ and (possibly) remove it from S.

On its own, this operation is not particularly useful because we have not quantified the distribution of keys returned. To do that, we will first need some terminology.

For a relaxed ordering structure O maintaining a ⟨key, value⟩ set S, we say that a value v has *rank* i at time t in S if there are exactly $i-1$ values with smaller keys in S, denoted by $\text{rank}_t(v) = i$. If ApproxGetMin() outputs v at time step t, then we also say that $\text{rank}(t) = \text{rank}_t(v)$. If, at some time step, $\langle u_1, v_1\rangle, \langle u_2, v_2\rangle \in S$ with $u_1 < u_2$ and an ApproxGetMin() operation outputs $\langle u_2, v_2\rangle$, we say that v_1 experiences an *inversion.* For any fixed value, v, let $\text{inv}(v)$ be the total number of inversions v experiences throughout E (before itself being removed by an ApproxGetMin() operation).

We quantify a relaxed ordering structure implementing some ApproxGetMin() operation with the following two criteria. We say that a relaxed ordering structure is (ρ, ϕ)-*relaxed* if it satisfies:

ρ-Error Boundedness. We say that a relaxed ordering structure O is ρ-*error bounded* if, for any invocation of ApproxGetMin() at time t, we have that

$$\Pr\left[\mathrm{rank}(t) > r\right] \leq \exp\left(-r/\rho\right).$$

ϕ-Fairness. We say that a relaxed ordering structure O is ϕ-*fair* if, for any value v,

$$\Pr\left[\mathrm{inv}(v) > r\right] \leq \exp\left(-r/\phi\right).$$

Intuitively, ρ-Error Boundedness enforces that ApproxGetMin() returns elements with rank $O(\rho)$ with high probability, while ϕ-fairness enforces that every element u sees at most $O(\phi)$ higher rank elements returned by ApproxGetMin() operations before u itself is returned.

In general, it is more convenient to assume a single quantification parameter k, and so we say that a (k, k)-relaxed ordering structure is simply k-relaxed. The remainder of this thesis will focus on demonstrating that this proposal is both *achievable* by real, performant implementations and *useful* for realistic applications.

Chapter 3

The SprayList

In this chapter, we present and analyze the SPRAYLIST, a SkipList-based relaxed priority queue implementation. We present the SPRAYLIST algorithm, and show that it is $O(n \log^3 n)$ relaxed for a set of parameters which minimizes contention by enforcing that the probability that a thread attempts to remove any given element is bounded by $O(1/n)$.

We compare our performance to that of the quiescently-consistent priority queue of Lotan and Shavit [53], the state-of-the-art SkipList-based priority queue implementation of Lindén and Jonsson [51] and the recent k-priority queue of Wimmer et al. [73].[1] 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 [62], while Lotan and Shavit [53] 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 [41] give a

[1]Due to the complexity of the framework of [73], we only provide a partial comparison with our algorithm in terms of performance.

lock-free version of this algorithm.

Sundell and Tsigas [70] 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 [51] 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.

Recent work by Mendes et al. [20] employed *elimination* techniques, speculatively matching GETMIN() and INSERT() operations, to adapt to contention in an effort to extend scalability. Still, their experiments do not show throughput scaling beyond 20 threads.

Another direction by Wimmer et al. [73] 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 will depend on the specific application scenario and on the workload.

An interesting vein of research investigates parallel data structures with priority-queue semantics in the PRAM model, e.g. [19, 23, 45, 67]. We note that, as opposed to our design, many of these proposals rely on the relative synchrony of threads to provide ordering semantics. Therefore, a precise comparison with this line of work is outside the scope of this thesis.

3.1 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 [32, 41], for which several freely available implementations exist, e.g. [32, 34]. In the following, we assume the reader is familiar with the structure of a SkipList (refer to [61]), and give an overview of standard lock-free SkipList operations.

3.1.1 The Classic Lock-Free SkipList

Our presentation follows that of Fraser [32, 41], and we direct the reader to these references for a detailed presentation.

General Structure. 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.) The SkipList is comprised of multiple levels, each of which is a linked list. Every node is inserted deterministically at the lowest level, and probabilistically at higher levels. It is common for the probability that a given node is present at level ℓ to be $2^{-\ell}$. (Please see Figure 1-2 for an illustration.) A key idea in this design is that a node can be independently inserted at each level. A node is *present* if it has been inserted into the bottom list; insertion at higher levels is useful to maintain logarithmic average search time.

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 and Harris [32] solve 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 nodes are adjacent om the list, with key values less-than and greater-than-equal-to the search key, respectively.

The search loop skips over marked nodes, 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 currently accessed node becomes marked during the traversal, the entire search is re-started from the SkipList head. 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, Fraser showed that this can be done by performing a *search* for the key: recall that the search procedure swings list pointers over marked nodes.

Cleaners / Lotan-Shavit GetMin. In this context, the Lotan-Shavit [53] GET-MIN() operation traverses the bottom list attempting to acquire a node via a locking operation. 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 in our design, described below.

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.

3.1.2 Spraying and Deletion

The goal of the Spray operation is to allow n threads to each emulate a uniform choice among the $k = O(n \log^3 n)$ highest-priority items. In doing so, we will show in Section 3.2 that Spray operations can implement ApproxGetMin() of a k-relaxed priority queue. To perform a Spray, a thread starts at the front of the SkipList, and at some initial height h. (See Figure 1-3 above for an illustration.)

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

Once on the bottom list, the thread 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 thread 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.

We note that, as with other SkipList based Priority Queue algorithms, the runtime of a Spray operation is independent of the size of the SkipList. This is because, with high probability, the Spray operation only accesses pointers belonging to the $k = O(n \log^3 n)$ items at the head of the list.

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 m elements, where $m \gg k$.

Starting Height. Each SPRAY starts at list level $H = \log n + H_0$, for some constant H_0.[2] (Intuitively, starting the SPRAY from a height less than $\log n$ leads to a high number of collisions, while starting from a height of $C \log n$ for $C > 1$ leads to SPRAYs which traverse beyond the first $O(n \log^3 n)$ elements.)

Jump Length Distribution. We choose the maximum number of forward steps L that a SPRAY may take at a level to be $L = M \log^3 n$, 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 $\log n$ 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 $\log n$. (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 $D = \max(1, \lfloor \log \log n \rfloor)$ 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

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

Algorithm 2 Pseudocode for SPRAY(H, L, D).

```
function SPRAY$(H, L, D)$
    $x \leftarrow head$        ▷ $x$ = pointer to current location        ▷ Assume $D$ divides $H$
    $\ell \leftarrow H$                                                   ▷ $\ell$ is the current level
    while $\ell \geq 0$ do
        Choose $j_\ell \leftarrow Uniform[0, L]$                          ▷ random jump
        Walk $x$ forward $j_\ell$ steps on list at height $\ell$    ▷ traverse the list at this level
        $\ell \leftarrow \ell - D$                                        ▷ descend $D$ levels
    return $x$
```

remove it as in a standard lock-free SkipList [32, 41]. 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, as described in Section 3.1.3.

3.1.3 Optimizations

Padding. A first practical observation is that, for small (constant) values of D, the SPRAY procedure above is biased against elements at the front of the list. For example, it would be extremely unlikely for the first element in the list to be hit by a walk with $D = 1$. To circumvent this bias, in such cases, we simply "pad" the SkipList: we add $R(n)$ dummy entries in the front of the SkipList. If a SPRAY lands on one of the first $K(p)$ dummy entries, it restarts. We choose $R(n)$ such that the restart probability is low, while, at the same time, the probability that any given node in the interval $[R(n) + 1, n \log^3 n]$ is hit is close to $1/n \log^3 n$. We note that padding is not necessary for higher values of D, e.g., $D = \Theta(\log \log n)$.

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 [53] 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 removal 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 $1/n$, i.e., roughly one in n 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 n, 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 n rather than a known value. Note that with this optimization enabled, if only a single thread access the SPRAYLIST, it will always dequeue the element with the smallest key.

3.2 Spray Analysis

In this section, we analyze the behavior of SPRAY operations. Our main goal is to prove that implementing APPROXGETMIN() with SPRAY operations yields a k-relaxed priority queue for $k = O(n \log^3 n)$, while at the same time maximizing scalability by showing that it is unlikely for concurrent SPRAY operations to collide. Note that there is a trade-off here: one could decrease the relaxation factor k but suffer more collisions and thus higher contention in exchange (ending up with an exact queue at the extreme), or one could further reduce contention at the cost of weaker semantic guarantees. We describe our analytical model in Section 3.2.1. We then give a first motivating result in Section 3.2.2, bounding the probability that two SPRAY operations collide for an ideal SkipList.

We state and prove our main technical result, Theorem 3.3, which will establish both *fairness* and *collision probability*. In essence, given our model, our results show that SprayLists do not return low priority elements except with extremely small probability (Theorem 3.2) and that there is very low contention on individual elements, which in turn implies the bound on the running time of SPRAY (Corollary 3.2).

3.2.1 Analytic 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 among roughly the first $O(n \log^3 n)$ elements. We will then use this fact to bound the contention between SPRAY operations. We therefore assume that there are $m \gg n \log^3 n$ elements in the SkipList.

We consider a set of at most n concurrent, asynchronous threads trying to perform DELETEMIN 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 $1/2^i$, independent of all other nodes. They do so by each performing SPRAY operations. When two or more SPRAY operations end at the same node, all but one of them must retry. if a SPRAY lands in the padded region of the SkipList, it must also retry. We repeat this until all SPRAYs land at unique nodes (because at most one thread can obtain a node). Our goal is to show that for all n threads, this process will terminate in $O(\log^3 n)$ time in expectation and with high probability. Note that since each SPRAY operation takes $O(\log^3 n)$ time, this is equivalent to saying that each thread must restart their SPRAY operations at most a constant number of times, in expectation and with high probability. We guarantee this by showing that SPRAY operations have low contention.

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.

3.2.2 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 3.1, which holds for an idealized SkipList. Basically, Theorem 3.1 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 consecutive 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 $H = \log n - 1$, $L = \log n$, and $D = 1$, the same as our implementation version. Practically, the walk starts at level $\log n - 1$ of the SkipList, and, at each level, uniformly chooses a number of forward steps between $[1, \log n]$ before descending. We prove the following upper bound on the collision probability, assuming that $\log n$ is even:

Theorem 3.1. *For any position x in a perfect SkipList, let $F_n(x)$ denote the proba-*

bility that a SPRAY$(\log n - 1, \log n, 1)$ *lands at* x. *Then*

$$F_n(x) \leq 1/(2n).$$

Proof. In the following, fix parameters $H = \log n - 1, L = \log n, D = 1$ 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 $0 \leq i \leq \log n - 1$.

We start from the observation that, on a *perfect* SkipList, the operation lands at the element of index $\sum_{i=0}^{\log n-1} a_i 2^i$ 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 $(\log n + 1)$-tuple $(a_0, \ldots, a_{\log n})$ whose elements are chosen independently and uniformly from the interval $\{1, \ldots, \log n\}$ has the property that the jumps sum up to x, that is,

$$\sum_{i=0}^{\log n-1} a_i 2^i = x. \tag{3.1}$$

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 $(\log n)$-tuple $(a_1, \ldots, a_{\log n})$ as specified above. We wish to compute the probability that the random tuple satisfies Equation 3.1. Notice that, for $\sum_{i=0}^{\log n-1} a_i 2^i = x$, 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 3.1 is satisfied, then we must have $a_1(1) + a_0(2) = x(2)$. In general, for all $1 \leq k \leq \log n - 1$, we see that to satisfy Equation 3.1, 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 $\{1, \ldots, \log n\}$, 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 $1 \leq k \leq \log n - 1$, randomly generate an a_k uniformly from the interval $\{1, \ldots, \log n\}$, 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 $(\log n - 1)$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

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

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

$$\Pr[\text{pass 0th round}] = 1/2. \tag{3.2}$$

Moreover, for any $1 \leq i \leq \log n - 2$, 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

$$\Pr[\text{pass } (i+1)\text{th round} | A_i] = 1/2. \tag{3.3}$$

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

□

3.2.3 Complete Runtime Analysis for ApproxGetMin()

In this section, we show that, given a randomly chosen SkipList, each ApproxGetMin() operation completes in $O(\log^3 n)$ steps, in expectation and with high probability. As mentioned previously, this is equivalent to saying that the Spray operations for each thread restart at most a constant number of times, in expectation and with high probability. The crux of this result (stated in Corollary 3.2) is a characterization of the probability distribution induced by Spray operations on an arbitrary SkipList, which we obtain in Theorem 3.3. Our results require some mathematical preliminaries. For simplicity of exposition, throughout this section and in the full analysis we assume n which is a power of 2. (If n is not a power of two we can instead run Spray with the n set to the smallest power of two larger than the true n, and incur a constant factor loss in the strength of our results.)

We consider Sprays with the parameters $H = \log n - 1$, $L = M \log^3 n$, and $D = \max(1, \log\log n)$. We will assume that all jump parameters are integers, and that D divides H. The claim is true even when these assumptions do not hold, but we only present the analysis in this special case because the presentation otherwise becomes too messy. Let ℓ_n be the number of levels at which traversals are performed, except the bottom level; in particular $\ell_n = H/D$.

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 3.1. *Fix two positive functions $f(n), g(n)$.*

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

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 $(\ell_n + 1)$-length vector $(a_0, \ldots, a_{\ell_n})$ where $1 \leq a_i \leq M \log^3 n$ denotes how far we walk at height iD, 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 Sprays. 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 $F_n(i, S)$ denote the probability that a Spray returns element i. We will write this often as $F_n(i)$ when it is clear which S we are working with.

Definition 3.2. *We say an event happens* with high probability *or* w.h.p. *for short if it occurs with probability at least* $1 - n^{-\Omega(M)}$*, where* M *is the constant defined in Algorithm 2.*

Top Level Theorems

With these definitions we are equipped to state our main theorems about SprayLists.

Theorem 3.2. *In the model described above, no* Spray *will return an element beyond the first* $M(1 + \frac{1}{\log n})\sigma(n) n \log^3 n \simeq M n \log^3 n$*, with probability at least* $1 - n^{-\Omega(M)}$.

This theorem establishes $k = O(n \log^3 n)$-error boundedness, stating that sprays do not go too far past the first $O(n \log^3 n)$ elements in the SkipList. The proof of Theorem 3.2 is fairly straightforward and uses standard concentration bounds. 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 3.3. *For $n \geq 2$ and under the stated assumptions, there exists an interval of elements $I(n) = [a(n), b(n)]$ of length $b(n) - a(n) \simeq Mn \log^3 n$ and endpoint $b(n) \lesssim Mn \log^3 n$, such that for all elements in the SkipList in the interval $I(n)$, we have that*

$$F_n(i, S) \simeq \frac{1}{Mn \log^3 n},$$

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

In plain words, this theorem states that there exists a range of elements $I(n)$, whose length is asymptotically equal to $Mn \log^3 n$, 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 $I(n)$, up to a factor of two. The condition $b(n) \lesssim Mn \log^3 n$ 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 $R(n) = a(n)$ dummy elements, the SPRAY procedure will approximate uniform selection from roughly the first $Mn \log^3 n$ elements, w.h.p. over the random choice of the SkipList.

Fairness and Runtime Bounds. Given this theorem, we can both establish $k = O(n \log^3 n)$-fairness and also bound the probability of collision for two Sprays, which in turn bounds the running time for a APPROXGETMIN() operation,yielding the following corollaries.

Corollary 3.1. *The* SPRAYLIST *is $k = O(n \log^3 n)$-fair.*

Corollary 3.2. *In the model described above,* APPROXGETMIN() *takes $O(\log^3 n)$ time in expectation. Moreover, for any $\epsilon > 0$,* APPROXGETMIN() *will run in time $O(\log^3 n \log \frac{1}{\epsilon})$ with probability at least $1 - \epsilon$.*

Corollary 3.1 is immediate. The proof of Corollary 3.2 is given in Section 3.2.3. Combining Theorem 3.2 with Corollaries 3.1 and 3.2 gives the complete SPRAYLIST picture:

Theorem 3.4. *The* SPRAYLIST *is k-relaxed for $k = O(n \log^3 n)$ supporting* APPROX-GETMIN() *in time $O(\log^3 n)$.*

Proof of Theorem 3.2

Throughout the rest of the section, we will need a way to talk about partial SPRAYs, those which have only completely some number of levels.

Definition 3.3. *Fix a* SPRAY *S, $(a_0, \ldots, a_{\ell_n})$ where $1 \le a_i \le M \log^3 n$.*

- *To any k-tuple $(b_k, \ldots, b_\ell)$ for $k \ge 0$, associate to it the walk which occurs if, descending from level $\ell_n D$, we take b_r steps at each height rD, as specified in* SPRAY. *We define the k-prefix of S to be the walk associated with $(a_k, \ldots, a_\ell)$. We say the k-prefix of S* returns *the element that the walk described ends at.*

- *To any $(k+1)$-tuple $(b_0, \ldots, b_k)$ for $k \le \ell_n$ and any starting element i, associate to it the walk which occurs if, descending from level kD, we take b_r steps at each height rD, as specified in* SPRAY. *We define the k-suffix of S to be the walk associated with $(a_0, \ldots, a_k)$, starting at the node the $(\ell - k - 1)$-prefix of S returns. We say the k-prefix of S* returns *the element that the walk described ends at.*

- *The kth* part *of S is the walk at level kD of length a_k starting at the element that the $(\ell_n - k + 1)$-prefix of S returns.*

Intuitively, the k-prefix of a spray is simply the walk performed at the k top levels of the spray, and the k-suffix of a spray is simply the walk at the bottom k levels of the spray.

For $k \ge 0$, let E_k denote the expected distance the SPRAY travels at the kDth level if it jumps exactly $M \log^3 n$ steps. In particular,

$$E_k = M2^{kD} \log^3 n.$$

We in fact prove the following, stronger version of Theorem 3.2.

Lemma 3.1. *Let $\sigma(n) = \log n/(\log n - 1)$. For any fixed α, the k-suffix of any* SPRAY *will go a distance of at most $(1+\alpha)\sigma(n)E_{kD+1}$, with probability at least $1 - n^{-\Omega(M\alpha^2 \log^2 n)}$ over the choice of the SkipList. To prove this we first need the following proposition.*

Notice that setting $\alpha = 1/\log n$ and $k = \ell_n$ then gives us the Theorem 3.2. Thus it suffices to prove Lemma 3.1. First, we require a technical proposition.

Proposition 3.1. *For $k \leq \log n$ and $\alpha > 0$, the probability that the kth part of a* SPRAY *travels more than $(1+\alpha)M2^k \log^3 n$ distance is at most $(1/n)^{\Omega(M\alpha^2 \log^2 n)}$.*

Proof. Fix some node x in the list. Let X_T be the number of elements with height at least k that we encounter in a random walk of T steps starting at x. We know that $\mathbb{E}(X_T) = T/2^k$. Choose $T = (1+\alpha)M2^k \log^3 n$. Then by a Chernoff bound, $\Pr(X_T \leq (1+\alpha)M \log^3 n) \leq n^{-\Omega(M\alpha^2 \log^2 n)}$.

Therefore, if we take T steps at the bottom level we will with high probability hit enough elements of the necessary height, which implies that a SPRAY at that height will not go more than that distance. □

Proof of Lemma 3.1. Without loss of generality, suppose we start start at the head of the list, and j is the element with distance $(1+\alpha)\sigma(n)E_{kD+1}$ from the head. Consider the hypothetical SPRAY which takes the maximal number of allowed steps at each level rD for $r \leq k$. Clearly this SPRAY goes the farthest of any SPRAY walking at levels kD and below, so if this SPRAY cannot reach j starting at the front of the list and walking only on levels kD and below, then no SPRAY can. Let x_r denote the element at which the SPRAY ends up after it finishes its rDth level for $0 \leq r \leq k$ and $x_{kD+1} = 0$, and let d_r be the distance that the SPRAY travels at level rD. For any $r \geq 0$, by Proposition 3.1 and the union bound, $\Pr(\exists k : d_r > (1+\alpha)E_{rD}) \leq n^{-\Omega(M\alpha^2 \log^2 n)}$.

Therefore, w.h.p., the distance that this worst-case spray will travel is upper bounded by

$$\begin{aligned}\sum_{r=0}^{k} d_r &\leq (1+\alpha)\sum_{r=0}^{k} E_r \\ &\leq (1+\alpha)\sigma(n)E_{kD+1}.\end{aligned}$$

□

Outline of Proof of Theorem 3.3

We prove Theorem 3.3 by proving the following two results:

Lemma 3.2. *For all elements i, we have*

$$F_i(n,S) \lesssim \frac{1}{nM\log^3 n}$$

with high probability over the choice of S.

Lemma 3.3. *There is some constant $A > 1$ which for n sufficiently large can be chosen arbitrarily close to 1 so that for all*

$$i \in [An\log^2 n, \frac{1}{1+1/\log n}Mn\log^3 n],$$

we have

$$F_i(n) \gtrsim \frac{1}{Mn\log^3 n}$$

with high probability over the choice of S.

Given these two lemmas, if we then let $I(n)$ be the interval defined in Theorem 3.3, it is straightforward to argue that this interval satisfies the desired properties for Theorem 3.3 w.h.p. over the choice of the SkipList S. Thus the rest of this section is dedicated to the proofs of these two lemmas.

Fix any interval $I = [a, b]$ for $a, b \in \mathbb{N}$ and $a \leq b$. In expectation, there are $(b - a + 1)2^{k-1}$ elements in I with height at least k in the SkipList; the following Lemma bounds the deviation from the expectation.

Proposition 3.2. *For any b, and any height h, let $D_{b,h}$ be the number of items between the $(b - k)$th item and the bth item in the SkipList with height at least h, and let $E_{b,h} = (k + 1)2^{1-h}$ be the expected value of $D_{b,h}$. Then for any $\alpha > 0$,*

$$\Pr\left[|D_{b,h} - E_{b,h}| > (1 + \alpha)E_{b,h}\right] < e^{-\Omega(E_{b,h}\alpha^2)}$$

Proof. Let X_i be the random variable which is 1 if the $(b-k+i)$th item has a bucket of height at least i, and 0 otherwise, and let $X = \sum_{i=0}^{k} X_i$. The result then follows immediately by applying Chernoff bounds to X. $\square$

Proof of Lemma 3.2

With the above proposition in place, we can now prove Lemma 3.2.

Proof of Lemma 3.2. Let $I_0 = [i - M \log^3 n + 1, i]$ and for $k \geq 1$ let

$$I_k = [\lceil i - (1 + \alpha)\sigma(n)E_{(k-1)D}\rceil + 1, i],$$

and let t_k denote the number of elements in the SkipList lying in I_k with height at least kD. Define a Spray to be *viable at level* k if its $(\ell - k)$-prefix returns some element in I_k, and say that a Spray is *viable* if it is viable at every level. Intuitively, a Spray is viable at level k if, after having finished walking at level kD, it ends up at a node in I_k. By Lemma 3.1, if a Spray is not viable at level k for any $1 \leq k \leq \ell_n$, it will not return x except with probability $n^{-\Omega(M\alpha^2 \log^2 n)}$ over the choice of the SkipList, for all k. Thus, by a union bound, we conclude that if a Spray is not viable, it will not return x except with probability $n^{-\Omega(M\alpha^2 \log^2 n)}$ over the choice of the SkipList. It thus suffices to bound the probability that a Spray is viable.

Let t_k be the number of elements in I_k with height at least kD. The probability that the kDth level of any Spray lands in I_k is at most $t_k/(M \log^3 n)$, since we choose how far to spray at level kD uniformly at random. By Proposition 3.2 we know that except with probability $e^{-\Omega(\alpha^2(E_k+1))} = n^{-\Omega(M\alpha^2 \log^2 n)}$, I_k contains at most

$$(1+\alpha)^2\sigma(n)E_{(k-1)D}2^{-kD}$$
$$= (1+\alpha)^2 M\sigma(n)\log^2 n$$

elements with height at least kD. Hence,

$$\frac{t_k}{(M\log^3 n)} \leq (1+\alpha)^2\sigma(n)\frac{1}{\log n}$$

except with probability $n^{-\Omega(M\alpha^2 \log^2 n)}$, for any fixed k. By a union bound over all $\log n/\log\log n$ levels, this holds for all levels except with probability $n^{-\Omega(M\alpha^2 \log^2 n)}$. Thus, the probability that a SPRAY lands in I_0 after it completes but the traversal at the bottom of the list is

$$\left((1+\alpha)^2\sigma(n)\frac{1}{\log n}\right)^{\log n/\log\log n}$$

except with probability $n^{-\Omega(M\alpha^2 \log^2 n)}$. If we choose $(1+\alpha)^2 = (1+\frac{1}{\log n})$ so that $\alpha = \sigma(n)^{1/2} - 1$, we obtain that since $(\log n)^{-\frac{\log n}{\log\log n}} = \frac{1}{n}$. Since $\sigma(n)^{\log n/\log\log n} \simeq 1$, and

$$\alpha^2\log^2 n = \left(\sqrt{\frac{\log n}{\log n - 1}} - 1\right)^2 \log^2 n \simeq \frac{1}{4},$$

it must be that with high probability, the fraction of SPRAYs that land in I_0 is asymptotically bounded by n^{-1}. Conditioned its ℓ-prefix returning something in I_0, for the SPRAY to return i, it must further take the correct number of steps at the bottom level, which happens with at most a $\frac{1}{M\log^3 n}$ fraction of these SPRAYs. Moreover, if the ℓ-prefix of the SPRAY does not return an element in I_0, then the SPRAY will not hit i, since it simply too far away. Thus $F_n(i,S) \lesssim \frac{1}{nM\log^3 p}$, as claimed. as claimed.

□

Proof of Lemma 3.3

Proof Strategy. We wish to lower bound the probability of hitting the i^{th} smallest item, for i in some reasonable interval which will be precisely defined below. For simplicity of exposition in this section, we will assume that all the endpoints of the intervals we define here are integers are necessary. While this is not strictly true, the proof is almost identical conceptually (just by taking floors and ceilings whenever appropriate) when the values are not integers and much more cumbersome.

Fix some index i. As in the proof of Lemma 3.2, we will again filter SPRAY by where they land at each level. By defining slightly smaller and non-overlapping regions than in the proof of Lemma 3.2, instead of obtaining upper bounds on the probabilities that a SPRAY lands at each level, we are instead able to lower bound the probability that a SPRAY successfully lands in the "good" region at each level, conditioned on the event that they landed in the "good" region in the previous levels.

Formally, let $I_0 = [i - \log^3 n, i - 1]$. Let S be a spray, chosen randomly. Then if $i - \log^3 n \geq 0$, we know that if the ℓ-prefix of S returns an element in I_0, then S has a $1/\log^3 n$ probability of stepping to i. Inductively, for all $k \leq \ell_n - 1$, we are given an interval $I_{k-1} = [a_{k-1}, b_{k-1}]$ so that $a_{k-1} \geq 0$. Notice that there are, except with probability $n^{-\Omega(M\alpha^2 \log^2 n)}$, at most $M \log^3 n$ elements in $[b_{k-1} - \frac{1}{1+\alpha} E_{kD}, b_{k-1}]$ with height kD, by Proposition 3.2.

Then, let $a_k = b_{k-1} - \frac{1}{1+\alpha} E_{(k-1)D}$ and $b_k = a_{k-1} - 1$, and let $I_k = [a_k, b_k]$. For all $0 \leq k \leq \ell_n - 1$, let t_k be the number of elements in I_k with height $(k+1)D$. Assume for now that $a_k \geq 0$. Then, if the $(k-1)$-prefix of S returns an element i in I_k, then every element of I_{k-1} of height kD by some walk of length at most $M \log^3 n$ at level kD, since there are at most $M \log^3 n$ elements of height $k \log \log n$ in the interval $[a_k, b_{k-1}]$ and $b_k < a_{k-1}$. Thus, of the SPRAYs whose $(k+1)$ prefixes return an element in I_k, a $t_k/(M \log^3 n)$ fraction will land in I_{k-1} after walking at height kD. The following proposition provides a size bound on the I_k.

Proposition 3.3. *Let $s_k = b_k - a_k + 1$. For all $k \geq 2$, we have*

$$\left(\gamma_0 - \gamma_1 \frac{1}{\log n}\right) E_{kD} \leq s_k \leq \left(\gamma_0 + \gamma_1 \frac{1}{\log^2 n}\right) E_k$$

with $\gamma_0 = \frac{\log n}{(\alpha+1)(\log n+1)}$ and $\gamma_1 = \frac{\alpha \log n+\alpha+1}{(\alpha+1)(\log n+1)}$.

Proof. Define ξ_k to be the quantity so that $s_k = \xi_k E_k$. Clearly $\xi_0 = 1$, and inductively,

$$\begin{aligned} s_k &= \frac{1}{1+\alpha} E_k - s_{k-1} \\ &= \left(\frac{1}{1+\alpha} - \frac{\xi_{k-1}}{\log n}\right) E_k \end{aligned}$$

so

$$\xi_k = \frac{1}{1+\alpha} - \frac{1}{\log n} \xi_{k-1}.$$

Homogenizing gives us a second order homogeneous recurrence relation

$$\xi_k = \left(1 - \frac{1}{\log n}\right) \xi_{k-1} + \frac{1}{\log n} \xi_{k-2}$$

with initial conditions $\xi_0 = 1$ and $\xi_1 = \frac{1}{1+\alpha} - \frac{1}{\log n}$. Solving gives us that

$$\xi_k = \gamma_0 + \gamma_1 \left(-\frac{1}{\log n}\right)^k.$$

Notice that $\xi_{2k+2} \leq \xi_{2k}$ and $\xi_{2k+3} \geq \xi_{2k+1}$ and moreover, $\xi_{2k+1} \leq \xi_{2k'}$ for any k, k'. Thus for $k \geq 2$ the maximum of ξ_k occurs at $k = 2$, and the minimum occurs at $k = 1$. Substituting these quantities in gives us the desired results. □

Once this result is in place, we use it to obtain a lower bound on the hit probability.

Lemma 3.4. *There is some constant $A > 1$ which for n sufficiently large can be chosen arbitrarily close to 1 so that for all $i \in [An \log^2 n, \frac{1}{1+1/\log n} Mn \log^3 n]$, we have $F_i(n) \gtrsim \frac{1}{Mn \log^3 n}$ with high probability.*

This statement is equivalent to the statement in Theorem 3.3.

Proof. The arguments made in Section A.3 are precise, as long as (1) every element of I_{k-1} can be reached by a walk from anywhere in I_k at level k of length at most $M\log^3 n$, and (2) each $a_k \geq 0$. By Proposition 3.2, condition (2) holds except with probability $n^{-O(\alpha^2 M \log^2 n)}$. Moreover, each $a_k \geq 0$ is equivalent to the condition that $i \geq \log^3 n + \sum_{k=0}^{\ell_n - 1} s_k$, but by Proposition 3.3, we have that (except with probability $n^{-O(M\alpha^2 \log^2 n)}$) that

$$\sum_{k=0}^{\ell_n-1} s_k \leq \left(\gamma_0 + \gamma_1 \frac{1}{\log^2 n}\right)\left(\sum_{k=0}^{\ell_n-1} E_k\right).$$

For the choice of $\alpha = \frac{1}{\log n}$, the first term in this product can be made arbitrarily close to one for n sufficiently large, and thus we have that except with probability $n^{-O(\alpha^2 M \log^2 n)}$,

$$\sum_{k=1}^{\ell_n-1} s_k \quad \leq \quad AMn\log^2 n,$$

for some A which can be made arbitrarily close to one for n sufficiently large.

By Propositions 3.2 and 3.3, by a union bound, we have that except with probability $n^{-O(M\alpha^2 \log^2 n)}$,

$$t_k \geq 2^{-D}\left(\gamma_0 - \gamma_1 \frac{1}{\log n}\right) M \log^3 n,$$

for all k. Thus by the logic above, if we let H_k denote the event that the $(k+1)$-prefix of the spray is in I_k, we have that the probability that the spray hits i is

$$\begin{aligned}
&\geq \Pr(\text{spray hits } i | H_0)\left(\prod_{k=1}^{\ell_n-1} \Pr(H_{k-1}|I_k)\right)\Pr(H_{\ell_n-1}) \\
&\geq \frac{1}{\log^3 n}\prod_{k=0}^{\ell_n-1}\frac{t_k}{M\log^3 n} \\
&\geq \frac{1}{\log^3 n}\left(2^{-\lfloor \log\log n \rfloor}\left(\gamma_0 - \gamma_1\frac{1}{\log n}\right)\right)^{\ell_n}.
\end{aligned}$$

If we choose $\alpha = \frac{1}{\log n}$, then one can show that

$$\left(\gamma_0 - \gamma_1 \frac{1}{\log n}\right)^{\ell_n} \simeq 1,$$

so we conclude that $F_i(n) \gtrsim \frac{1}{n \log^3 n}$ with high probability. □

Proof of Corollary 3.2

We have shown so far that on a clean skip list, SPRAY operations act like uniformly random selection on a predictable interval I near the front of the list of size tending to $Mn \log^3 n$. We justify here why this property is sufficient to guarantee that SPRAY operations execute in polylogarithmic time. A single SPRAY operation always takes polylogarithmic time, however, a thread may have to repeat the SPRAY operation many times. We show here that this happens with very small probability.

Corollary 3.2. *In the model described above,* DELETEMIN *takes* $O(\log^3 n)$ *time in expectation. Moreover, for any* $\epsilon > 0$, DELETEMIN *will run in time* $O(\log^3 n \log \frac{1}{\epsilon})$ *with probability at least* $1 - \epsilon$.

Proof. Recall that a thread has to retry if either (1) its SPRAY lands outside of I, or (2) the SPRAY collides with another SPRAY operation which has already removed that object. We know by Theorem 3.3 and more specifically the form of $I(n)$ given in Lemma 3.4 that (1) happens with probability bounded by $O(1/\log n)$ as $n \to \infty$ for each attempted SPRAY operation since Lemma 3.4 says that there are $O(n \log^2 n)$ elements before the start of $I(n)$, and Theorem 3.3 says that each is returned with probability at most $O(1/n \log^3 n)$, and (2) happens with probability upper bounded by the probability that we fall into set of size $n-1$ in I, which is bounded by $O(1/\log^3 n)$ for n sufficiently large by Lemma 3.2. Thus, by a union bound, we know that the probability that SPRAY operations must restart is bounded by $O(1/\log n) \leq 1/2$ for n sufficiently large. Each spray operation takes $\log^3 n$ time, and thus the expected time it takes for a SPRAY operation to complete is bounded by

$$\log^3 n \sum_{i=0}^{\infty} 2^{-1} = O(\log^3 n)$$

and thus we know that in expectation, the operation will run in polylogarithmic time, as claimed. Moreover, for any fixed $\epsilon > 0$, the probability that we will restart more than $O(\log(1/\epsilon)/\log\log n)$ times is at most ϵ, and thus with probability at least $1-\epsilon$, we will run in time at most $O(\log^3 n \log(1/\epsilon)/\log\log n)$. □

3.3 Implementation Results

Methodology. Experiments were performed on a Fujitsu 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 [32] 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 [51]. This algorithm has been shown to outperform other linearizable priority queue algorithms under benchmarks similar to our own and is optimized to minimize compare-and-swap (CAS) operations performed by DELETEMIN. Physical deletion is batched and performed by a deleting thread 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.

Wimmer et. al. k-Priority Queue. The relaxed k-Priority Queue given by Wimmer et. al. [73]. This implementation provides a linearizable priority queue, except that it is relaxed in 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 [73]. 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 3.1, 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 3.1.3) 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 [32], using the benchmarking framework of Synchrobench [34]. The code has been made publicly available [5].

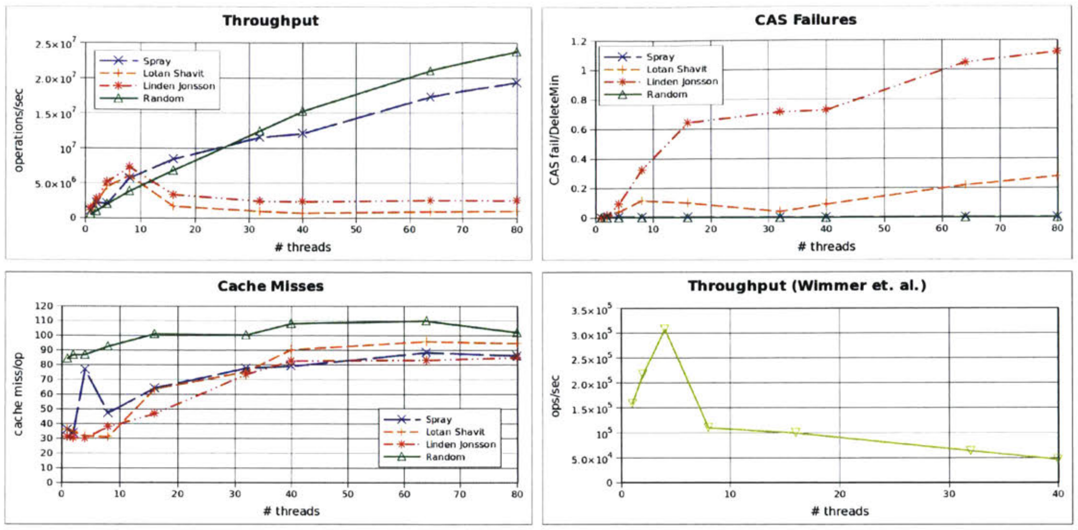

Figure 3-1: Priority Queue implementation performance on a 50% insert, 50% delete workload: throughput (operations completed), average CAS failures per DeleteMin, and average L1 cache misses per operation.

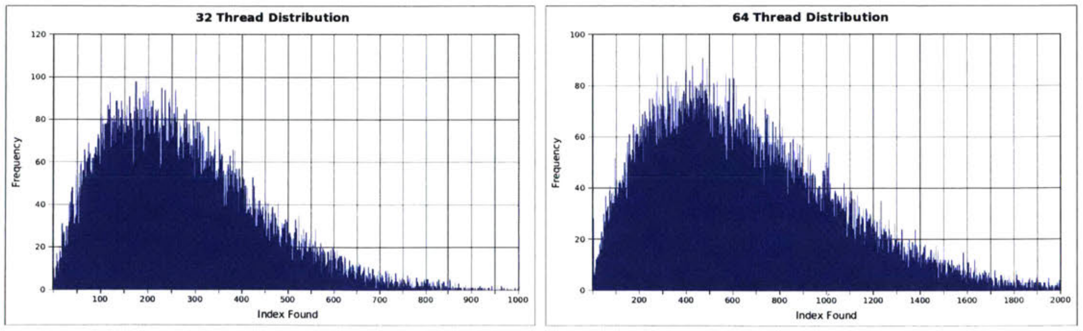

Figure 3-2: The frequency distribution of SPRAY operations when each thread performs a single SPRAY on a clean SPRAYLIST over 1000 trials. Note that the x-axis for the 64 thread distribution is twice as wide as for 32 threads.

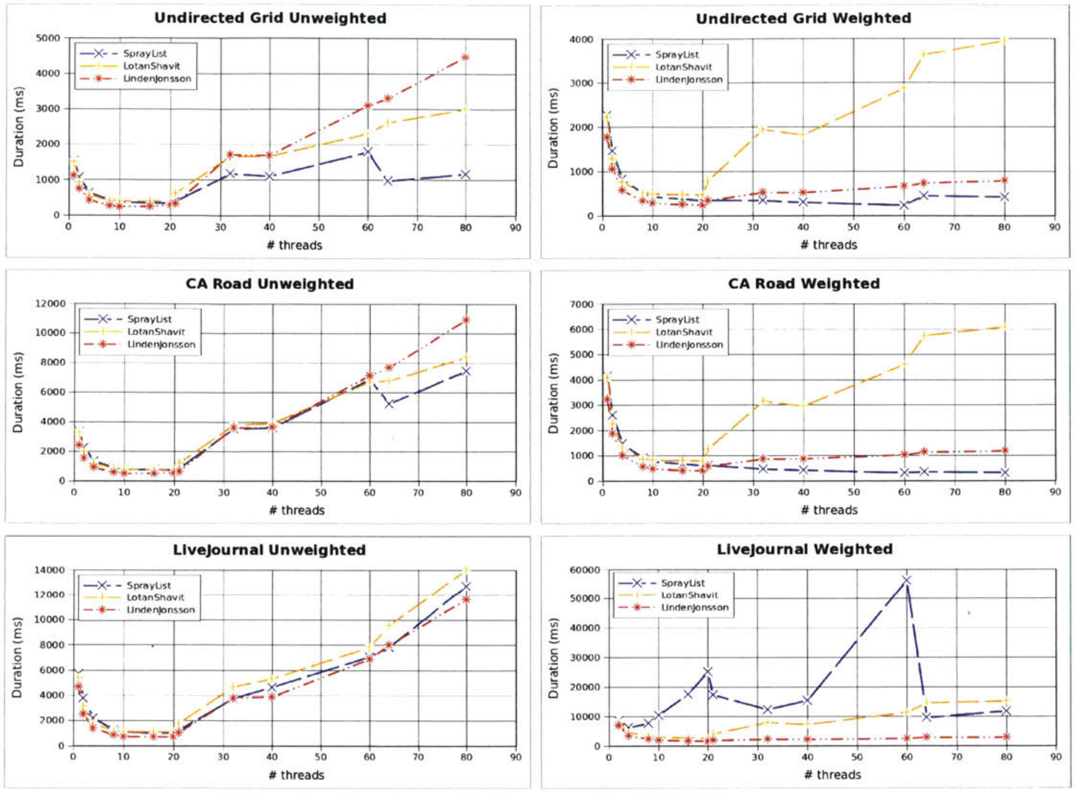

Figure 3-3: Runtimes for SSSP using each PriorityQueue implementation on each network (lower is better).

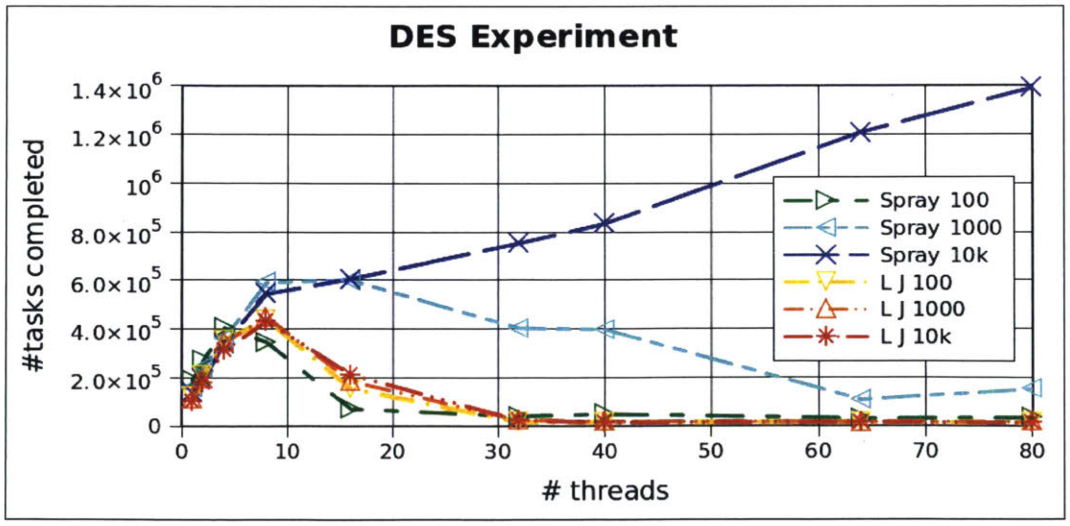

Figure 3-4: Work performed for varying dependencies (higher is better). The mean number of dependants is 2 and the mean distance between an item and its dependants varies between 100, 1000, 10000.

3.3.1 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 3-1 shows the data collected from this experiment. At low thread counts (≤ 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 additional 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 3-1 and the task-based implementation of Wimmer et. al. However, we emulated our throughput benchmark within the framework of [73] by implementing tasks whose only functionality is to spawn a new task and re-add it to the system.

Figure 3-1 shows the total number of tasks processed by the k-priority queue of Wimmer et. al.[3] 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.

3.3.2 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

[3]We used the hybrid k-priority queue which was shown to have the best performance of the various implementations described [73].

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 3-2 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 elements 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 super-linear, 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).

3.3.3 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 DecreaseKey() 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) [50]. Since the data did not contain edge weights, we ran experiments with unit weights (resembling breadth-first search) and uniform random weights. Figure 3-3 shows the running time of the SSSP 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 $\lfloor \log p \rfloor$ 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.

3.3.4 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 respect 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, the priority of an event corresponds to its time step.

We emulate such a DES system with the following methodology: we initially insert 1 million events (labeled 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 [51], which in turn is based on the URDME stochastic simulation framework [25].

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

This benchmark allows us to examine the trade-off between the relaxed semantics and the increased concurrency of SprayLists. Figure 3-4 reports the actual work performed by each algorithm, where actual work is calculated by 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 throughput experiment, this experiment presents high levels of contention, so implementations without scaling throughput cannot hope to scale here despite little wasted work. 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 relaxation. However, as K increases, we start to see more scalability, with $K = 1000$ scaling up to 16 threads and $K = 10000$ scaling up to 80 threads.

To quantify the scalability relative to the distribution of dependencies, for each thread count, we increased K until performance plateaued and recorded the value of K at which the plateau began. Figure 3-5 reports the results of this experiment. Note that the minimum K required increases near linearly with the number of threads.

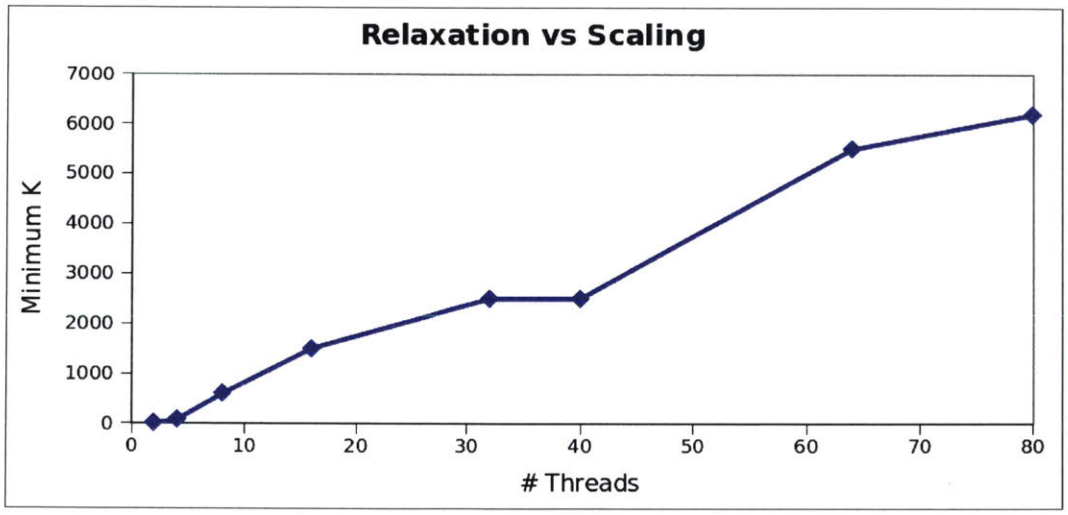

Figure 3-5: Minimum value of K which maximizes the performance of the SprayList for each fixed number of threads.

Chapter 4

MULTIQUEUEs: Power of Choice Allocations

In this chapter, we consider the MULTIQUEUE strategy based on the data structure of Rihani et al. [64]. We start by instantiating n sequential priority queues, each of which is protected by a lock. To *insert* an element, each thread repeatedly picks a queue at random, and tries to lock it. When successful, the thread inserts the element into the queue, and unlocks it. Otherwise, the thread either spins on the lock or re-tries with a new randomly chosen lock.

To *delete* an element, the algorithm samples two queues uniformly, and reads the top element from both. It then locks the queue whose top element had higher priority, removes that value, if it has not changed, and returns it. If the algorithm failed to obtain the lock, or if the top element has changed, then it releases all locks, and retries. Algorithms 3 and 4 give pseudocode for INSERT() and APPROXGETMIN() respectively.

Although MULTIQUEUEs were shown to have fantastic throughput and seemingly reasonable rank error when they were initially introduced [64], it was not clear whether they provided *provable* guarantees. While there is a simple argument showing that MULTIQUEUEs incur low rank error in the *initial state*, one might expect that even with the two-choice optimization applied to deletes, MULTIQUEUEs could still experience deteriorating state over long executions, possibly leading to rank error proportional to (some function of) execution length. This would be a very undesirable characteristic and a potential worry for application designers.

This chapter will address that issue, showing rigorously that MultiQueues exhibit $O(n \log n)$-error boundedness and fairness, independent of execution length. Although establishing $O(n \log n)$ fairness is relatively straightforward, the $O(n \log n)$-error boundedness result is quite technical and will be the main focus.

Algorithm 3 MultiQueue Insert()

```
repeat
    $i \leftarrow \mathsf{UniformRandom}(1, n)$
until try-lock($Q_i$) successful
$Q_i$.Insert()($e$)
$Q_i$.unlock()
```

Algorithm 4 MultiQueue GetMin()

```
repeat
    $i \leftarrow \mathsf{UniformRandom}(1, n)$
    $j \leftarrow \mathsf{UniformRandom}(1, n)$
    if $Q_i.\min > Q_j.\min$ then
        swap($i, j$)
until try-lock($Q_i$) successful
$Q_i$.GetMin()($e$)
$Q_i$.unlock()
```

4.1 Related Work

The first instance of a distributed data structure implementation using a similar approach to the one we consider is probably the parallel branch-and-bound framework by Karp and Zhang [45]. In their construction, each thread is assigned a queue, and elements are inserted randomly into one of the queues. Each thread removes from its own queue. This critically relies on the synchronicity of PRAM threads to achieve bounds on average rank and maximum rank difference. It is easy to see that thread delays can cause the rank difference to become unbounded.

Rihani et al. show via a basic balls-into-bins argument that, initially, before any elements are removed, the expected rank cost is $O(n)$, and the max cost is $O(n \log n)$ with high probability. However, this claim is not sufficient to establish a relaxation parameter k under which MultiQueues are k-relaxed, since the state of the MultiQueue system might deteriorate over time. Our argument is significantly more general, since it applies to any time in the execution of the process.

In addition to [64], variants of this approach have also been considered in other contexts, for instance for relaxed queues [36] and priority schedulers [58].

Technical Overview. A tempting first approach at analysis is to reduce to classic "power of two choices" processes, e.g. [8, 60], in which balls are always inserted into the least loaded of two randomly chosen bins. A simple reduction between these two processes exists for the case where labels are inserted into the queues in *round-robin* fashion (see Section 4.2 for details). However, this reduction breaks if elements are inserted uniformly at random, as is the case in real systems. Another important difference from the classic process is that elements are *labelled*, and so the state of the system at a given step is highly correlated with previous steps. For instance, given a queue Q whose top element has label ℓ, if we wish to characterize the probability that the next label on top of Q is a specific value $\ell' > \ell$, we need to know whether the history of elements examined by the algorithm (in other queues) contains element ℓ'. Such correlations make a standard step-by-step analysis extremely challenging.

We circumvent these issues by relating the original labelled process to a continuous *exponential process*, which reduces correlations by replacing integer labels with real-valued labels. In this process, on every "insertion", we insert a new label whose value equals that of the latest inserted label in that bin (or 0 for the first insertion), plus *an exponential random variable* of mean n. Intuitively, we wish to simulate the integer label insertion process with the same mean n, while removing correlations but using continuous label values to prevent label clashes and reduce correlations. Once this insertion process has ended, we replace each (real-valued) label with its *rank* among all (real-valued) labels.

We relate these two processes by the following key claim: *the rank distributions* *for the original process and for the exponential process are identical.* More precisely, for any queue i and rank j, the event that, after insertion, the label with rank j is located in queue i has the same probability in the original process and in the exponential process (and all such events are independent in both processes).

Given this fact, we can couple the two processes as follows. We first couple the insertion steps such that each bin contains the same sequence of ranks in both processes. We then couple the removal steps so that they make exactly the same sequence of random choices. Under this coupling, the two processes will pay the same rank cost. Thus, it will suffice to characterize the rank cost of the exponential process under our removal scheme.

This is achieved via two steps. The first is a potential argument which carefully characterizes the average value of labels on top of the queues, and the maximum deviation from the average by a queue, at any time $t > 0$. This part of the proof is very technical, and generalizes an analytic approach developed for weighted balls-into-bins processes by Peres, Talwar and Wieder [60]. Specifically, if we let $x_i(t)$ be the difference between the label on top of queue i at time t and the mean top label across all queues, then the *total potential* of the system at time t is defined as $\sum_{i=1}^{n} [\exp(\alpha x_i/n) + \exp(-\alpha x_i/n)]$.

The key claim in the argument is showing that the expected value of this potential is bounded by $O(n)$ for any step t. This is done by a careful technical analysis of the expected change in potential at a step, which proves that the potential has supermartingale-like behavior: that is, it tends to decrease in expectation once it surpasses the $O(n)$ threshold. It is interesting to note that neither of the two exponential factors in the definition of the potential satisfies this bounded increase property in isolation, yet their sum can be bounded in this way.

The $O(n)$ bound on the total potential provides a strong handle on the maximum deviation from the mean in the exponential process. The last step of our argument builds on this characterization to show that, since label values cannot stray too far either above or below the mean, the *ranks* of these values among all values on top of queues cannot be too large. In particular, the average rank of elements on top of queues must be $O(n)$, and the maximum rank is $O(n \log n)$. The rank equivalence theorem implies our main claim.

The analytic framework we sketched above is quite general. It can be extended to showing that this implementation strategy is robust to insertion bias, i.e. that it guarantees similar rank bounds even if the insertion process is biased (within some constant $\gamma \ll 1$) towards some of the bins. Further, it also applies to the $(1+\beta)$ variant of the process, where each removal considers two randomly chosen queues with probability $\beta < 1$, and a single queue otherwise. Formally, the expected cost at a step is $O(n/\beta^2)$, and the expected maximum cost at a step is $O(n \log n/\beta)$.

Balls-into-bins Processes. In the classic two-choice balanced allocation process, at each step, one new ball is to be inserted into one of n bins; the ball is inserted into the less loaded among two random choices [8, 54]. In this setting, it is known that the most loaded bin is at most $O(\log \log n)$ above the average. The literature studying extensions of this process is extremely vast; we direct the reader to [63] for a survey. Considerable effort has been dedicated to understanding guarantees in the "heavily-loaded" case, where the number of insertion steps is unbounded [10, 60], and in the "weighted" case, in which ball weights come from a probability distribution [11, 71]. In a tour-de-force, Peres, Talwar, and Wieder [60] gave a potential argument characterizing a general form of the heavily-loaded, weighted process on graphs. The second step in our proof, which characterizes the deviation of weights from the mean in the exponential process, builds on their approach. Specifically, we use a similar potential function, and the parts of the analysis are similar. However, we generalize their approach to the case of biased insert distributions, and our argument diverges in several technical points: in particular, the potential analysis under unbalanced conditions is different. The first and third steps of our argument are completely new, as we consider a more complex *labelled* allocation process.

4.2 Reduction to Two-Choices Process for Round-Robin Insertions

Process Description. As a warm-up result, we will consider the following simple process: we are given n queues, into which consecutively labeled elements are inserted in round-robin order: at the t^{th} insertion step, we insert the element with label t into bin $t \pmod{n}$. To remove an element, we pick two queues at random, and remove the element of lower label on top of either queue. We measure the cost of a removal as the rank of the removed label among labels still present in any of the queues.

Reduction. We will associate a "virtual" bin V_i with each queue i. Whenever an element is removed from the queue i, it gets immediately placed into the corresponding virtual bin. Notice that in every step we are removing the element of minimum label, which, by round-robin insertion, corresponds to the queue having been removed from less times than the other choice. Alternatively, we are inserting the removed element into the *less loaded* virtual bin, out of two random choices (implicitly breaking ties by bin ID). This is the classic definition of the two-choices process [8], which has been analyzed for the long-lived case in e.g. [10]. One can prove bounds on the rank of elements removed using the existing analyses of this process, e.g. [10, 60].

The Reduction Breaks for Random Insertions. Notice that the critical step in the above argument is the fact that, out of two random choices i and j, the less loaded virtual bin corresponds to the queue which has lower label on top. This does not hold in the random insertion case; for instance, it is possible for a bin to get two elements with consecutive labels, which breaks the above property. Upon closer examination, we can observe that in the random insertion case, the correlation between the event that queue i has lower label than queue j and the event that queue i has been removed from less times than queue j exists, but is too weak to allow a direct reduction.

4.3 Random Inserts Process Definition

We are given with n priority queues, labeled $1, \ldots, n$. To insert, we choose queue i with probability $\pi_i \in (0, 1)$, and insert into that queue. We assume that $\sum_{i=1}^{n} \pi_i = 1$, and that the bias is bounded, i.e. there exists a constant $\gamma \in (0, 1)$ such that, for any queue i, $1 - \gamma \leq \frac{1}{n\pi_i} \leq 1 + \gamma$. To remove, with probability $0 < \beta \leq 1$, we pick two queues u.a.r., and remove the element of minimum label among the two choices, and with probability $(1 - \beta)$ remove from a random queue. We call such a process a $(1 + \beta)$-sequential process. When omit β when unimportant or clear from context.

At any point in the execution, we define the *rank* of any element to be the number of elements currently in the system which have lower label than it (including itself), so the minimal rank is 1. Our goal is to show that the ranks of the elements returned by the random process are small, throughout the entire execution. We will restrict our attention to executions which never inspect empty queues, and no priority inversions on inserts are visible to the removal process. Formally:

Definition 4.1. *An execution of the sequential process is* prefixed *if except with negligible probability (i.e. probability less than $n^{-\omega(1)}$), the sequential process never inspects an empty queue in a remove, and no remove sees an insert with lower priority than one that is inserted later in the same queue.*

There are many natural types of executions which are prefixed. For instance, a common strategy in practice is to insert a large enough "buffer" of elements initially, so as to minimize the likelihood of examining empty queues. More formally, since in each iteration we touch a queue with probability at most $\frac{2}{n} + O(\frac{1}{n^2})$ and only remove one element at a time, by Bernstein's inequality, any execution of length T whose first $T/2 + \sqrt{n \cdot \omega(\log n)}$ operations are inserts, and which contains no inserts after this time, or which only inserts larger labels after this point, will be prefixed.

For the rest of the paper, we will tacitly assume that no remove inspects any empty queue in the sequential process, and that no priority inversions are visible to the removal process, and condition on the event that this does not happen. By Markov's inequality, as long as we only consider prefixed executions, then for any sequence of operations of length poly(n), this will change the max rank or average rank of the operations by at most a subconstant factor.

The process diverges when $\beta = 0$.

An important remark is that $\beta > 0$ is a necessary condition for good behavior. To see this, we will prove another warm-up theorem:

Theorem 4.1. *The expected maximum rank guarantee of the process which inserts and removes from uniform random queues in each time step t evolves as $\Omega(\sqrt{tn \log n})$, for $t = \Omega(n \log n)$.*

Proof. First, notice that we can apply the reduction in Section 4.2 to obtain that the maximum number of elements removed from a queue is the same as the maximum load of a bin in the classic long-lived random-removal process. (This holds irrespective of the insertion process, since we are performing uniform random removals.) The maximum load of a bin in the classic process is known to be $\Theta\left(t/n + \sqrt{\frac{t}{n} \log n}\right)$, with high probability, for $t > n \log n$ [60].

Let the queue with the most removals up to time t be Q. For any other queue Q_i with less removals, let $\ell(Q_i)$ be the difference between the number of elements removed from Q and the number of elements removed from Q_i. Notice that the expected number of elements removed from a queue up to t is t/n. Hence, it is easy to prove that, with probability at least $1/2$, there are at least $n/4$ queues which have had at most t/n elements removed. Hence, for any such queue Q_i, $\ell(Q_i) \geq \sqrt{\frac{t}{n} \log n}$. Hence, in expectation, $\sum_i \ell(Q_i) = \Omega(\sqrt{tn \log n})$.

We say that an element is *light* if its label is *smaller* than the label of the the top element of Q. By symmetry, we know that each element counted in $\sum_i \ell(Q_i)$ is light with probability $\geq 1/2$. Putting everything together, we get that, in expectation, we have $\Omega(\sqrt{tn \log n})$ elements which are *light*, i.e. of smaller label than the top label of Q. This implies that the expected rank of Q is $\Omega(\sqrt{tn \log n})$, as claimed.

□

4.4 Analysis of the Sequential Process

The goal of this section is to prove the following theorem:

Theorem 4.2 (see Corollary 4.1 and Corollary 4.2). *Fix a bias bound γ. In any prefixed $(1+\beta)$ sequential process, for $\beta = \Omega(\gamma)$, and for any time $t > 0$, the max rank of any element on top of a queue at time t is at most $O(\frac{1}{\beta}(n \log n + n \log \frac{1}{\beta}))$, and the average rank at time t is at most $O(n/\beta^2)$.*

The Exponential Process. For the purposes of analysis, it is useful to consider an alternative process, which we will call the *exponential process*, in which each bin independently generates real-valued labels by starting from 0, and adding an exponentially-distributed random variable with parameter $\lambda_i = 1/\pi_i$ to its previous label. More precisely, if $w_i(t)$ is the label present on top of bin i at time t, then the value of the next label is $w_i(t+1) = w_i(t) + \exp(1/\pi_i)$. Once we have enqueued all the items we wanted to enqueue, we proceed to remove items by the two-choice rule: we always pick the element of minimum rank (label value) among two random choices. At any step, and for any $v \in \mathbb{R}$, we define $\text{rank}(v)$ to be the number of elements currently in the system with label at most v. At each step, we pay a cost equal to the rank of the element we just removed.

Proof Strategy. The argument proceeds in three steps. First, we show that, perhaps surprisingly, after all insertion steps have been performed, the rank distribution of the exponential process is *the same* as the rank distribution of the original process (Section 4.4.1). With this in place, we will perform the following coupling between the two processes:

Given an arbitrary number M of balls to be inserted into the n bins, we first generate M total weights across the n queues in the exponential process, so that each bin has exactly the same number of elements in both processes. We then replace each (real-valued) generated weight with its *rank* among all weights in the system. As we will see, for any rank j and queue i, the probability that the ball with rank j is in bin i is exactly the same as in the simple process, where queue i has probability of insertion π_i. More precisely, if we fix an increasing sequence of ranks $r_1, r_2, \ldots$, the probability that bin i contains this exact sequence of ranks is the same across the two processes. Hence, we can couple the two processes to generate the same sequence of ranks in each bin.

For the removal step, we couple the two processes such that, in every step, they generate exactly the same β values and random choice indices. Since the ranks are the same, and the choices are the same, the two processes will remove from exactly the same bins at each step, and will pay the same rank cost in every step.

Hence, it will suffice to bound the expected rank cost paid at a step in the exponential process. Since this is difficult to do directly, we will first aim to characterize the *concentration* of the difference between the weight (value) on top of each bin and the average weight on top of bins, via a potential argument (Section 4.4.2). This will show that relatively few values can stray far from the mean. In turn, this will imply a bound on the average rank removed (Section 4.4.5).

In the following, we use the terms "queue" and "bin" interchangeably.

4.4.1 Equivalence between Rank Distributions

Let π_i be the probability that a ball is inserted into bin i. This section will be dedicated to showing that the distribution of ranks in the exponential process where bin i gets weights exponentially distributed with mean $1/\pi_i$ is identical to the distribution of labels in the original process, where the ith bin is chosen for insertion with probability π_i.

Theorem 4.3. *Let $I_{j\leftarrow i}$ be the event that the label with rank i is located in bin j, in either process. Let $\Pr_e[I_{j\leftarrow i}]$ be its probability in the exponential process, and $\Pr_o[I_{j\leftarrow i}]$ be its probability in the original process. Then for both processes $I_{j\leftarrow i}$ is independent from $I_{j'\leftarrow i'}$ for all $i \neq i'$ and*

$$\Pr_e[I_{j\leftarrow i}] = \Pr_o[I_{j\leftarrow i}] = \pi_j.$$

Proof. That $\Pr_o[I_{j\leftarrow i}] = \pi_j$ and that $I_{j\leftarrow i}, I_{j'\leftarrow i'}$ are independent for $i \neq i'$ in the original process both follow immediately by the construction of the sequential process.

We now require some notation. Let $\ell(i)$ be the label with rank i in the exponential process, and let $b(i)$ be the bin containing $\ell(i)$. We will employ the following *memoryless* property of exponential random variables:

Fact 4.1. *Let $X \sim \mathrm{Exp}(r)$. Then, for all $s, t \geq 0$, we have $\Pr[X > s + t] = \Pr[X > s]\Pr[X > t]$.*

Fix an arbitrary $i \geq 1$, and let $L = \ell(i-1)$ be the label of the $(i-1)$th element. For each bin i, we isolate the two consecutive weights between which L can be placed. More precisely, for each bin $1 \leq j \leq n$, denote the smallest label *larger* than L in bin j by $\ell_{j,>L}$ and let the largest label in B_j *smaller* than L be $\ell_{j,<L}$. By construction of the exponential process, $\ell_{j,>L} = \ell_{j,<L} + X_j$ where X_j is an exponentially distributed random variable with mean $1/\pi_j$. Furthermore, by assumption we have that $\ell_{j,>L} > L$, so $X_j > L - \ell_{j,<L}$. Then by the memoryless property above,

$$
\begin{aligned}
\Pr[\ell_{j,>L} > \ell_{j,<L} + s | \ell_{j,>L} > L] &= \Pr[X_j > s | X_j > (L - \ell_{j,<L})] \\
&= \Pr[X_j > (L - \ell_{j,<L}) + (s - L) | X_j > (L - \ell_{j,<L})] \\
&= \Pr[X_j > s - L]
\end{aligned}
$$

Let $\Delta_j = \ell_{j,>L} - L$, observing by the above that Δ_j is positive and distributed exponentially with mean π_i. Furthermore, since the X_j are independent, the Δ_j are as well. Consider the event that $\ell_{j,>L}$ is the smallest label larger than L in the system; that is $\ell(i) = \ell_{j,>L}$ and $b(i) = j$. This occurs if $\Delta_j < \Delta'_j$ for all $j' \neq j$, i.e. if Δ_j is the smallest such value among all bins. In turn, this occurs with probability

$$
\begin{aligned}
\int_0^\infty \Pr[\Delta_j = t] \Pi_{k \neq j} \Pr[\Delta_k > t] dt &= \int_0^\infty \pi_j \exp(-t\pi_j) \Pi_{k \neq j} \exp(-t\pi_k)\, dt \\
&= \pi_j \int_0^\infty \exp\left(-t \sum_{k=1}^{n} \pi_k\right) dt = \pi_j.
\end{aligned}
$$

□

4.4.2 Analysis of the Exponential Process

In the previous section, we have shown that the rank distributions are the same at the end of the insert process. The coupling described in Section 4.4, by which we inspect the same queues in each removal steps, implies that two processes will produce the same cost in terms of average rank removed. Hence, we focus on bounding the *rank* of the labels removed from the exponential process at each step t. Our strategy will be to first bound the deviation of the *weight* on top of each queue from the average, at every step t, and then re-interpret the deviation in terms of rank cost.

Notation. From this point on, we will assume for simplicity that, at the beginning of each step, queues are always ranked in *increasing order of their top label.* If p_i is the probability that we pick the ith ranked bin for a removal, and β is the two-choice probability, then it is easy to see that the $(1+\beta)$ process guarantees that

$$p_i = (1-\beta)\frac{1}{n} + \beta\left[\frac{2}{n}\left(1 - \frac{i-1}{n}\right) - \frac{1}{n^2}\right].$$

Further, notice that, for any $1 \le m \le n$, we have, ignoring the negligible $O(1/n^2)$ factor, that

$$\sum_{i=1}^{m} p_i \simeq \frac{m}{n}\left(1 + \beta - \frac{m}{n}\beta\right).$$

For any bin j and time t, let $w_j(t)$ be the label on top of bin j at time t, and let $x_j(t) = w_j(t)/n$ be the normalized label. Let $\mu(t) = \sum_{j=1}^{n} x_j(t)/n$ be the average normalized label at time t over the bins. Let $y_i(t) = x_i(t) - \mu(t)$, and let $\alpha < 1$ be a parameter we will fix later. Define

$$\Phi(t) = \sum_{j=1}^{n} \exp\left(\alpha y_i(t)\right), \text{ and } \Psi(t) = \sum_{j=1}^{n} \exp\left(-\alpha y_i(t)\right).$$

Finally, define the potential function

$$\Gamma(t) = \Phi(t) + \Psi(t).$$

Parameters and Constants. Define $\epsilon = \frac{\beta}{16}$. Recall that the parameter $\gamma > 0$ is such that, for every $1 \leq i \leq n$, $\frac{1}{n\pi_i} \in [1-\gamma, 1+\gamma]$. Next, let $c \geq 2$ be a constant, and δ be a parameter such that

$$1 + \delta = \frac{1 + \gamma + c\alpha\,(1+\gamma)^2}{1 - \gamma - c\alpha\,(1+\gamma)^2}. \tag{4.1}$$

In the following, we assume that the parameters α, β, and γ satisfy the inequality

$$\frac{\beta}{16} = \epsilon \geq \delta. \tag{4.2}$$

We assume that the insertion bias $\gamma \leq 1/2$ is small, and hence this is satisfied by setting $\beta = \Omega(\gamma)$ and $\alpha = \Theta(\beta)$.

The rest of this section is dedicated to the proof of the following claim:

Theorem 4.4. *Let $\vec{p} = (p_1, p_2, \ldots, p_n)$ be the vector of probabilities, sorted in increasing order. Let $\epsilon = \beta/16$, and $c \geq 2$ be a small constant. Let α, β, γ, δ be parameters as given above, such that $\epsilon \geq \delta$. Then there exists a constant $C(\epsilon) = \text{poly}(\frac{1}{\epsilon})$ such that, for any time $t \geq 0$, we have $\mathbb{E}[\Gamma(t)] \leq C(\epsilon)n$.*

Importantly, n-Error Boundedness is an immediate corollary of Theorem 4.4.

Theorem 4.5. *There exists a constant c such that $\Pr[y_n(t) \geq rcn\log n] < \exp(-r)$. Therefore, the* MultiQueue *process is $O(n\log n)$-Error Bounded.*

Proof. In order for ApproxGetMin() to return an item larger than $r(cn\log n)$ at time t, that item needs to be at the top of a queue at t, i.e. $y_n(t) \geq rcn\log n$. But

$$\begin{aligned}
\mathbb{E}[\Gamma(t)] &> \exp(\alpha rc\log n)\Pr[y_n(t) \geq rcn\log n] \\
C(\epsilon)n &> \exp(\alpha rc\log n)\Pr[y_n(t) \geq rcn\log n] \\
\exp(\log n(-\alpha rc + 1)) &> \Pr[y_n(t) \geq rcn\log n]
\end{aligned}$$

Picking $c > 2/\alpha$ gives that, for $r > 2$, $\Pr[y_n(t) \geq rcn\log n] < \exp(-r)$, and so MultiQueues are $cn\log n = O(n\log n)$-error bounded as claimed. □

Potential Argument Overview. The argument proceeds as follows. We will begin by bounding the change of each potential function Ψ and Φ, at each queue in a step (Lemmas 4.1 and 4.2). We then use these bounds to show that, if not too many queues have weights above or below the mean ($y_{n/4} \leq 0$ and $y_{3n/4} \geq 0$), then Φ and Ψ respectively decrease in expectation (Lemmas 4.3 and 4.4). Unfortunately, this does not necessarily hold in general configurations. However, we are able to show the following claim: if the configuration is unbalanced (e.g. $y_{n/4} > 0$) and Φ does not decrease in expectation at a step, then either the symmetric potential Ψ is large, and will decrease on average, or the global potential function Γ must be in $O(n)$ (Lemma 4.5). We will prove a similar claim for Ψ. Putting everything together, we get that the global potential function Γ always decreases in expectation once it exceeds the $O(n)$ threshold, which implies Theorem 4.4.

Potential Change at Each Step. We begin by analyzing the expected change in potential for each queue from step to step. We first look at the change in the weight vector $\vec{y} = (y_1, y_2, \dots, y_n)$. (Recall that we always re-order the queues in increasing order of weight at the beginning of each step.) Below, let κ_j be the cost increase if the bin of rank j is chosen, which is an exponential random variable with mean $1/\pi_j$. We have that, for every rank i,

$$y_i(t{+}1) = \begin{cases} y_i(t) + \kappa_i \left(1 - 1/n\right), \text{ with probability } p_i(\text{if the rank } i \text{ queue is picked}) \\ y_i(t) - \frac{\kappa_j}{n}, \text{if some other bin } j \neq i \text{ is chosen.} \end{cases}$$

The Change in Φ at a Step. We prove the following bound on the expected increase in Φ at a step.

Lemma 4.1. *For any bin rank i,*

$$\mathbb{E}\left[\Delta\Phi \,|\, y(t)\right] \leq \sum_{i=1}^{n} \hat{\alpha}\left((1+\delta)p_i - \frac{1}{n}\right) \exp\left(\alpha y_i(t)\right).$$

Proof. Let $\Phi_i(t) = \exp\left(\alpha y_i(t)\right)$. We have two cases. If the bin is chosen for removal, then the change is:

$$\begin{aligned}\Delta\Phi_i &:= \Phi_i(t+1) - \Phi_i(t)\\ &= \exp\left(\frac{\alpha}{n}(y_i(t) + \kappa_i\left(1 - \frac{1}{n}\right))\right) - \exp\left(\frac{\alpha}{n}y_i(t)\right)\\ &= \exp\left(\frac{\alpha}{n}y_i(t)\right)\left(\exp\left(\frac{\alpha}{n}\kappa_i\left(1-\frac{1}{n}\right)\right) - 1\right).\end{aligned}$$

Taking expectations with respect to the random choices made on insertion, that is, the value of κ_j, we have

$$\begin{aligned}\mathbb{E}_i\left[\exp\left(\kappa_i\frac{\alpha}{n}\left(1-\frac{1}{n}\right)\right)\right] &\overset{(a)}{=} \frac{\pi_i}{\pi_i - \left(\frac{\alpha}{n}(1-\frac{1}{n})\right)}\\ &= \frac{1}{1 - \frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right)}\\ &\overset{(b)}{\leq} 1 + \frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right) + c\left(\frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right)\right)^2,\end{aligned}$$

for some constant $c > 1$. Step (a) follows from the observation that the expectation we wish to compute is the moment-generating function of the exponential distribution at $\frac{\alpha}{n}(1-\frac{1}{n})$, while (b) follows from the Taylor expansion. (We slightly abused notation in step (a) by denoting with π_i the insert probability for the ith ranked queue, according to the ranking in this step.)
The second step if if some other bin $j \neq i$ is chosen for removal, then the change is:

$$\Delta\Phi_i = \exp\left(\tfrac{\alpha}{n}(y_i(t) - \kappa_j\tfrac{1}{n})\right) - \exp\left(\tfrac{\alpha}{n}y_i(t)\right) = \exp\left(\tfrac{\alpha}{n}y_i(t)\right)\left(\exp\left(-\tfrac{\alpha\kappa_j}{n^2}\right) - 1\right).$$

Again taking expectations with respect to the random choices made on insertion, i.e. the value of κ_j, we have

$$\begin{aligned}\mathbb{E}_i\left[\exp\left(-\kappa_j\frac{\alpha}{n^2}\right)\right] &= \frac{\pi_j}{\pi_j + \frac{\alpha}{n^2}}\\ &= \frac{1}{1+\frac{\alpha}{\pi_j n^2}}\\ &\leq 1 - \frac{\alpha}{\pi_j n^2} + \left(\frac{\alpha}{\pi_j n^2}\right)^2 - \ldots\end{aligned}$$

$$\leq 1 - \frac{\alpha}{\pi_j n^2} + \left(\frac{\alpha}{\pi_j n^2}\right)^2.$$

Therefore, we have that

$$\begin{aligned}
\mathbb{E}\left[\Delta\Phi_i\right]/\Phi_i(t) &\leq \left(1 + \frac{\alpha}{\pi_i n}\left(1 - \frac{1}{n}\right) + c\left(\frac{\alpha}{\pi_i n}\left(1 - \frac{1}{n}\right)\right)^2\right) p_i \\
&\quad + \sum_{j \neq i}\left(1 - \frac{\alpha}{\pi_j n^2} + \left(\frac{\alpha}{\pi_j n^2}\right)^2\right) p_j - 1 \\
&\leq \left(\frac{\alpha}{\pi_i n} + c\left(\frac{\alpha}{\pi_i n}\right)^2\right) p_i - \sum_{j=1}^{n}\left(\frac{\alpha}{\pi_j n^2} - c\left(\frac{\alpha}{\pi_j n^2}\right)^2\right) p_j \\
&\leq \left(1 + \gamma + c\alpha\left(1+\gamma\right)^2\right)\alpha p_i - \frac{\alpha}{n}\sum_{j=1}^{n}\left(1 - \gamma - c\alpha\left(1+\gamma\right)^2\right) p_j.
\end{aligned}$$

If we denote for convenience

$$\hat{\alpha} = \alpha\left(1 - \gamma - c\alpha\left(1+\gamma\right)^2\right), \text{ and recall that } \delta := \frac{1 + \gamma + c\alpha\left(1+\gamma\right)^2}{1 - \gamma - c\alpha\left(1+\gamma\right)^2} - 1,$$

then we can rewrite this as

$$\mathbb{E}\left[\Delta\Phi \mid y(t)\right] \leq \sum_{i=1}^{n}\hat{\alpha}\left(p_i(1+\delta) - \frac{1}{n}\right)\Phi_i(t).$$

□

The Change in Ψ. Using a symmetric argument, we can prove the following about the expected change in Ψ.

Lemma 4.2.

$$\mathbb{E}\left[\Delta\Psi \mid y(t)\right] \leq \sum_{i=1}^{n}\hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right)\exp\left(-\alpha y_i(t)\right).$$

Proof. **Case 1:** If the bin is chosen, then the change is:

$$\begin{aligned}
\Delta\Psi_i &= \Psi_i(t+1) - \Psi_i(t) \\
&= \exp\left(-\frac{\alpha}{n}(y_i(t) + \Delta_i\left(1-\frac{1}{n}\right))\right) - \exp\left(-\frac{\alpha}{n}y_i(t)\right) \\
&= \exp\left(-\frac{\alpha}{n}y_i(t)\right)\left(\exp\left(-\frac{\alpha}{n}\Delta_i\left(1-\frac{1}{n}\right)\right)-1\right)
\end{aligned}$$

Taking expectations with respect to the random choices made on insertion, we have

$$\begin{aligned}
\mathbb{E}_i\left[\exp\left(-\Delta_i\frac{\alpha}{n}\left(1-\frac{1}{n}\right)\right)\right] &= \frac{1}{1+\frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right)} \\
&= 1 - \frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right) + \left(\frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right)\right)^2 - \dots \\
&\leq 1 - \frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right) + \left(\frac{\alpha}{\pi_i n}\left(1-\frac{1}{n}\right)\right)^2 .
\end{aligned}$$

Case 2: If some other bin $j \neq i$ is chosen, then the change is:

$$\begin{aligned}
\Delta\Psi_i = {} & \exp\left(-\tfrac{\alpha}{n}(y_i(t) - \Delta_j\tfrac{1}{n})\right) - \exp\left(-\tfrac{\alpha}{n}y_i(t)\right) = \\
= {} & \exp\left(-\tfrac{\alpha}{n}y_i(t)\right)\left(\exp\left(\tfrac{\alpha\Delta_j}{n^2}\right)-1\right).
\end{aligned}$$

Again taking expectations with respect to the random choices made on insertion, we have

$$\begin{aligned}
\mathbb{E}_i\left[\exp\left(\Delta_i\frac{\alpha}{n^2}\right)\right] &= \frac{\pi_j}{\pi_j - \frac{\alpha}{n^2}} \\
&= \frac{1}{1-\frac{\alpha}{\pi_j n^2}} \\
&= 1 + \frac{\alpha}{\pi_j n^2} + \left(\frac{\alpha}{\pi_j n^2}\right)^2 + \dots \\
&\leq 1 + \frac{\alpha}{\pi_j n^2} + c\left(\frac{\alpha}{\pi_j n^2}\right)^2 .
\end{aligned}$$

Therefore, we have that

$$
\begin{aligned}
\mathbb{E}[\Delta\Psi_i] &\leq \left(1 - \frac{\alpha}{n\pi_i}\left(1 - \frac{1}{n}\right) + \left(\frac{\alpha}{n\pi_i}(1 - \frac{1}{n})\right)^2\right) p_i - 1 \\
&\quad + \sum_{j \neq i} \left(1 + \frac{\alpha}{n^2\pi_j} + c\left(\frac{\alpha}{n^2\pi_j}\right)^2\right) p_j \\
&\leq \left(-\frac{\alpha}{n\pi_i} + c\left(\frac{\alpha}{n\pi_i}\right)^2\right) p_i + \sum_{i=1}^{n} \left(\frac{\alpha}{n^2\pi_j} + c\left(\frac{\alpha}{n^2\pi_j}\right)^2\right) p_j \\
&\leq -\alpha p_i \left(1 - \gamma - c\alpha(1+\gamma)^2\right) p_i + \frac{\alpha}{n}\left(1 + \gamma + \alpha c(1+\gamma)^2\right) \\
&\leq \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right).
\end{aligned}
$$

□

Bounds under Balanced Conditions. Let us briefly stop to examine the bounds in the above Lemmas. The terms $\left(p_i(1+\delta) - \frac{1}{n}\right)$ are *decreasing* in i, and in fact become *negative* as i increases. (The exact index where this occurs is controlled by β and δ.) The $\exp(\alpha y_i(t))$ terms are *increasing* in i. Bins whose weight is *below* the mean (i.e., $y_i(t) \leq 0$) have a negligible effect on Φ, since each of their contributions is at most 1. At the same time, notice that the contribution of bins of large index i will be negative. Hence, we can show that, if at least $n/4$ bins have weights below average, then the value of Φ tends to *decrease* on average.

Lemma 4.3. *If $y_{n/4} \leq 0$, then we have that*

$$
\mathbb{E}[\Phi(t+1) \,|\, y(t)] \leq \left(1 - \frac{\hat{\alpha}\epsilon}{3n}\right)\Phi(t) + 1.
$$

Proof. We start from the inequality

$$
\begin{aligned}
\mathbb{E}[\Delta\Phi \,|\, y(t)] &\leq \sum_{i=1}^{n} \hat{\alpha}\left((1+\delta)p_i - \frac{1}{n}\right)\exp(\alpha y_i(t)) \\
&\leq \hat{\alpha}(1+\delta)\sum_{i=1}^{n} p_i \exp(\alpha y_i(t)) - \frac{\hat{\alpha}}{n}\Phi(t).
\end{aligned}
$$

We will now focus on bounding the first term (without the constants). We can rewrite it as:

$$\sum_{i=1}^{n} p_i \exp\left(\alpha y_i(t)\right) = \sum_{i=1}^{n/4-1} p_i \exp\left(\alpha y_i(t)\right) + \sum_{i=n/4}^{n} p_i \exp\left(\alpha y_i(t)\right). \tag{4.3}$$

Since $y_{n/4} \leq 0$, the first term is upper bounded by 1. For the second term, notice that

$$\sum_{i=n/4}^{n} p_i \exp\left(\alpha y_i(t)\right) = \sum_{j=1}^{3n/4} p_{n-j+1} \exp\left(\alpha y_{n-j+1}(t)\right). \tag{4.4}$$

The p terms are non-decreasing in j, while the y terms are non-increasing in j. Further, note that

$$\sum_{j=1}^{3n/4} \exp\left(\alpha y_{n-j+1}(t)\right) \leq \Phi. \tag{4.5}$$

The whole sum is maximized when these non-increasing terms are equal. We therefore are looking to bound

$$\sum_{i=n/4}^{n} p_i \exp\left(\alpha y_i(t)\right) \leq \frac{4\Phi}{3n} \sum_{i=n/4}^{n} p_i. \tag{4.6}$$

Notice that

$$\sum_{i=n/4}^{n} p_i = 1 - \sum_{i=1}^{n/4-1} p_i \leq 1 - \left(\frac{1}{4} + \epsilon\right) = \frac{3}{4} - \epsilon. \tag{4.7}$$

Hence we have that

$$\begin{aligned}
\mathbb{E}\left[\Delta\Phi \mid y(t)\right] &\leq \hat{\alpha}\left[(1+\delta)\sum_{i=1}^{n} p_i \exp\left(\alpha y_i(t)\right) - \frac{1}{n}\Phi(t)\right] \\
&\leq \hat{\alpha}\left[(1+\delta)\frac{4\Phi}{3n}\left(\frac{3}{4} - \epsilon\right) - \frac{1}{n}\Phi(t) + (1+\delta)\right] \\
&\leq \hat{\alpha}\left[\frac{1+\delta}{n}\left(1 - \frac{4\epsilon}{3}\right)\Phi(t) - \frac{1}{n}\Phi(t) + (1+\delta)\right] \\
&= \hat{\alpha}\left[\frac{\Phi(t)}{n}\left[\left(1 - \frac{4\epsilon}{3}\right)(1+\delta) - 1\right] + (1+\delta)\right] \\
&\leq -\hat{\alpha}\left[\frac{4\epsilon}{3}(1+\delta) - \delta\right]\frac{\Phi(t)}{n} + 1 \\
&\leq -\hat{\alpha}(1+\delta)\frac{\epsilon}{3}\frac{\Phi(t)}{n} + 1,
\end{aligned}$$

where in the last step we have used the fact that $\delta \le \epsilon$. Hence, we get that

$$\mathbb{E}\left[\Phi(t+1) \,|\, y(t)\right] \le \left(1 - \frac{\hat{\alpha}\epsilon}{3n}\right)\Phi(t) + 1.$$

□

A similar claim holds for Ψ, under the condition that there are at least $n/4$ bins with weight *larger* than average.

Lemma 4.4. *If $y_{3n/4} \ge 0$, then we have that*

$$\mathbb{E}\left[\Psi(t+1) \,|\, y(t)\right] \le \left(1 - \tfrac{\hat{\alpha}\epsilon}{3n}\right)\Psi + 1.$$

Proof. We start from the inequality

$$\begin{aligned}\mathbb{E}\left[\Delta\Psi \,|\, y(t)\right] &\le \sum_{i=1}^{n} \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right)\exp\left(-\alpha y_i(t)\right)\\ &\le \sum_{i=1}^{3n/4} \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right)\exp\left(-\alpha y_i(t)\right) +\\ &\quad \sum_{i=3n/4+1}^{n} \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right)\exp\left(-\alpha y_i(t)\right).\end{aligned}$$

The second term is upper bounded as

$$\sum_{i=3n/4+1}^{n} \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right) \le \hat{\alpha}(1+\delta)/4 - \hat{\alpha}\sum_{i=3n/4+1}^{n} p_i \le 1.$$

We therefore now want to bound

$$\sum_{i=1}^{3n/4} \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right)\exp\left(-\alpha y_i(t)\right).$$

We again notice that the first factor is non-decreasing, whereas the second one is non-increasing. Hence, the sum is maximized when the non-decreasing terms are equal.

At the same time, we have that

$$\sum_{i=1}^{3n/4} \exp\left(-\alpha y_i(t)\right) \leq \Psi.$$

Hence, it holds that

$$\begin{aligned}
\sum_{i=1}^{3n/4} \hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right)\exp\left(-\alpha y_i(t)\right) &\leq \frac{4\Psi}{3n}\sum_{i=1}^{3n/4}\hat{\alpha}\left((1+\delta)\frac{1}{n} - p_i\right) \\
&= \hat{\alpha}\frac{1+\delta}{n}\Psi - \hat{\alpha}\frac{4\Psi}{3n}\sum_{i=1}^{3n/4} p_i \\
&= \hat{\alpha}\frac{1+\delta}{n}\Psi - \hat{\alpha}\frac{1}{n}\Psi\left(1+\frac{4\epsilon}{3}\right) \\
&= \frac{\hat{\alpha}}{n}\Psi\left((1+\delta) - \left(1+\frac{4\epsilon}{3}\right)\right) \\
&\leq -\frac{\hat{\alpha}}{n}\frac{\epsilon}{3}\Psi,
\end{aligned}$$

where in the last step we have used the fact that $\delta \leq \epsilon$. Therefore, we get that

$$\mathbb{E}\left[\Psi(t+1)\,|\,y(t)\right] \leq \left(1 - \tfrac{\hat{\alpha}\epsilon}{3n}\right)\Psi + 1,$$

as claimed.

□

Bounds under Unbalanced Conditions. We now analyze unbalanced configurations, where there are either many bins whose weights are above average (e.g., $y_{n/4} > 0$), or below average ($y_{3n/4} < 0$). The rationale we used to bound each potential function independently no longer applies.

As an aside, we will demonstrate an *unbalanced* configuration where, for example, Φ does not decrease in expectation: Consider a configuration in which the first bin in the ordering has weight W, while all other bins have weight $kn+W$, where k is a large integer parameter. The average weight is $W + k(n-1)$, and hence $y_1 = -k(n-1)$, and $y_i = k$. For instance, for $k = \Theta(\log\log n)$, we get that $\Phi(t) = \Theta(n \log n)$.

Critically, in this configuration, the value of Φ will *increase* in expectation: with probability $2/n$, we hit queue 1 and increase its value by n, reducing the difference from weights on top of the other queues. Otherwise, with probability $1 - 2/n$, we hit one of the other queues, and increase its value by n. This *increases* the discrepancy between the queue costs. It follows by calculation that the expected value of Φ increases in expectation. Note that the process will eventually leave such local maxima, by taking out all the extra elements on top of queue 1, but this process takes $\Omega(n^2)$ steps.

Fortunately, we can show that, one of two things must hold: either the other potential Ψ is *larger* and *does decrease* in expectation, or the global potential Γ is bounded by $O(n)$.

Lemma 4.5. *Given ϵ as above, assume that $y_{n/4}(t) > 0$, and $\mathbb{E}[\Delta\Phi] \geq -\frac{\epsilon\hat{\alpha}}{3n}\Phi(t)$. Then either $\Phi < \frac{\epsilon}{4}\Psi$ or $\Gamma = O(n)$.*

Proof. Fix $\lambda = 2/3 - 1/54$ for the rest of the proof. We can split the inequality in Lemma 4.1 as follows:

$$\mathbb{E}\left[\Delta\Phi \mid y(t)\right] \leq \sum_{i=1}^{\lambda n} \hat{\alpha}\left((1+\delta)p_i - \frac{1}{n}\right)\exp\left(\alpha y_i(t)\right) + \tag{4.8}$$

$$\sum_{i=\lambda n+1}^{n} \hat{\alpha}\left((1+\delta)p_i - \frac{1}{n}\right)\exp\left(\alpha y_i(t)\right). \tag{4.9}$$

We bound each term separately. Since the probability terms are non-increasing and the exponential terms are non-decreasing, the first term is maximized when all p_i terms are equal. Since these probabilities are at most 1, we have

$$\sum_{i=1}^{\lambda n} \hat{\alpha}\left((1+\delta)p_i - \frac{1}{n}\right)\exp\left(\alpha y_i(t)\right) \leq \frac{\hat{\alpha}}{n}\left((1+\delta)\frac{1}{\lambda} - 1\right)\Phi_{\leq \lambda n}. \tag{4.10}$$

The second term is maximized by noticing that the p_i factors are non-increasing, and are thus dominated by their value at λn. Noticing that we carefully picked λ such that

$$p_{\lambda n} \leq \frac{1}{n} - \frac{4\epsilon}{n},$$

we obtain, using the assumed inequality $\delta \leq \epsilon$, that

$$\sum_{i=\lambda n+1}^{n} \hat{\alpha}\left((1+\delta)p_i - \frac{1}{n}\right)\exp\left(\alpha y_i(t)\right) \leq \hat{\alpha}\left(\delta - 4\epsilon\right)\frac{\Phi_{>\lambda n}}{n} \leq -\frac{3\epsilon\hat{\alpha}}{n}\Phi_{>\lambda n}. \quad (4.11)$$

By the case assumption, we know that $\mathbb{E}\left[\Delta\Phi \,|\, y(t)\right] \geq -\frac{\hat{\alpha}\epsilon}{3n}\Phi(t)$. Combining the bounds (4.9), (4.10), and (4.11), this yields:

$$\frac{\hat{\alpha}}{n}\left((1+\delta)\frac{1}{\lambda} - 1\right)\Phi_{\leq\lambda n} - \frac{3\epsilon\hat{\alpha}}{n}\Phi_{>\lambda n} \geq -\frac{\hat{\alpha}\epsilon}{3n}\Phi(t).$$

Substituting $\Phi_{>\lambda n} = \Phi - \Phi_{\leq\lambda n}$ yields:

$$(3\epsilon - \epsilon/3)\,\Phi \leq \left((1+\delta)\frac{1}{\lambda} - 1 + 3\epsilon\right)\Phi_{\leq\lambda n}.$$

For simplicity, we fix $C(\epsilon) = \frac{(1+\delta)\frac{1}{\lambda} - 1 + 3\epsilon}{3\epsilon - \epsilon/3} = O\left(\frac{1}{\epsilon}\right)$, to obtain

$$\Phi \leq C(\epsilon)\Phi_{\leq\lambda n}. \quad (4.12)$$

Let $B = \sum_{y_i>0} y_i$. Since we are normalizing by the mean, it also holds that $B = \sum_{y_i<0}(-y_i)$. Notice that

$$\Phi_{\leq\lambda n} \overset{y_i\text{incr.}}{\leq} \lambda n \exp\left(\alpha y_{\lambda n}\right) \overset{y_i\text{incr.}}{\leq} \lambda n \exp\left(\frac{\alpha B}{(1-\lambda)n}\right). \quad (4.13)$$

We put inequalities (4.12) and (4.13) together and get

$$\Phi(t) \leq \lambda n C(\epsilon)\exp\left(\tfrac{\alpha B}{(1-\lambda)n}\right), \quad (4.14)$$

Let us now lower bound the value of Ψ under these conditions. Since $y_{n/4} > 0$, all the costs below average must be in the first quarter of y. We can apply Jensen's inequality to the first $n/4$ terms of Ψ to get that

$$\Psi \geq \sum_{i=1}^{n/4} \exp\left(-\alpha y_i\right) \geq \frac{n}{4} \exp\left(-\alpha \frac{\sum_{i=1}^{n/4} y_i}{n/4}\right).$$

We now split the sum $\sum_{i=1}^{n/4} y_i$ into its positive part and its negative part. We know that the negative part is summing up to exactly $-B$, as it contains all the negative y_i's and the total sum is 0. The positive part can be of size at most $B/4$, since it is maximized when there are exactly $n-1$ positive costs and they are all equal. Hence the sum of the first $n/4$ elements is at least $-3B/4$, which implies that the following bound holds:

$$\Psi \geq \frac{n}{4} \exp\left(-\alpha \frac{-3B/4}{n/4}\right) \geq \frac{n}{4} \exp\left(\alpha \frac{3B}{n}\right). \tag{4.15}$$

If $\Phi < \frac{\epsilon}{4}\Psi$, then there is nothing to prove. Otherwise, if $\Phi \geq \frac{\epsilon}{4}\Psi$, we get from (4.15) and (4.14) that

$$\frac{\epsilon}{4}\frac{n}{4} \exp\left(\alpha \frac{3B}{n}\right) \leq \frac{\epsilon}{4}\Psi \leq \Phi(t) \leq \lambda n C(\epsilon) \exp\left(\tfrac{\alpha B}{(1-\lambda)n}\right),$$

Therefore, we get that:

$$\exp\left(\alpha \frac{B}{n}\left(3 - \frac{1}{1-\lambda}\right)\right) \leq \frac{4\lambda}{\epsilon} C(\epsilon) = O\left(\frac{1}{\epsilon^2}\right).$$

Using the mundane fact that $3 - \frac{1}{1-\lambda} = 3/19$, we get that

$$\exp\left(\frac{\alpha B}{n}\right) \leq O\left(\frac{1}{\epsilon^{14}}\right). \tag{4.16}$$

To conclude, notice that (4.16) implies we can upper bound Γ in this case as:

$$\begin{aligned}\Gamma &= \Phi + \Psi \\ &\leq \frac{4+\epsilon}{\epsilon}\Phi\end{aligned}$$

$$
\begin{aligned}
&\leq \frac{4+\epsilon}{\epsilon}\lambda n C(\epsilon)\exp\left(\frac{\alpha B}{(1-\lambda)n}\right)\\
&\leq O\left(\frac{1}{\epsilon^{14/(1-\lambda)}}\right)\frac{4+\epsilon}{\epsilon}C(\epsilon)\lambda n\\
&= O\left(\text{poly}\left(\frac{1}{\epsilon}\right)n\right).
\end{aligned}
$$

□

We can prove a symmetric claim for Ψ by a slightly different argument.

Lemma 4.6. *Given ϵ as above, assume that $y_{\frac{3n}{4}} < 0$, and that $\mathbb{E}[\Delta\Psi] \geq -\frac{\hat{\alpha}\epsilon}{3n}\Psi$. Then either $\Psi < \frac{\epsilon}{4}\Phi$ or $\Gamma = O(n)$.*

Proof. Fix $\lambda = \lambda_2 = 1/3 + 1/54$. We start from the general bound on $\Delta\Psi$ from Lemma 4.2. We had that

$$
\begin{aligned}
\mathbb{E}\left[\Delta\Psi \mid y(t)\right] &\leq \sum_{i=1}^{n}\hat{\alpha}\left((1+\delta)\frac{1}{n}-p_i\right)\exp\left(-\alpha y_i(t)\right)\\
&= \sum_{i=1}^{\lambda n-1}\hat{\alpha}\left((1+\delta)\frac{1}{n}-p_i\right)\exp\left(-\alpha y_i(t)\right)\\
&\quad+\sum_{i=\lambda n}^{n}\hat{\alpha}\left((1+\delta)\frac{1}{n}-p_i\right)\exp\left(-\alpha y_i(t)\right)\\
&\leq \frac{\hat{\alpha}}{n}\left((1+\delta)-(1+4\epsilon)\right)\Psi_{<\lambda n}+\frac{\hat{\alpha}}{n}\left(1+\delta\right)\Psi_{\geq\lambda n}\\
&\leq -\frac{3\hat{\alpha}\epsilon}{n}\Psi_{<\lambda n}+\frac{\hat{\alpha}}{n}\left(1+\delta\right)\Psi_{\geq\lambda n}.
\end{aligned}
$$

Using the assumption, we have that

$$
-\frac{\hat{\alpha}\epsilon}{3n}\Psi \leq \mathbb{E}\left[\Delta\Psi \mid y(t)\right] \leq -\frac{3\hat{\alpha}\epsilon}{n}\left(\Psi-\Psi_{\geq\lambda n}\right)+\frac{\hat{\alpha}}{n}\left(1+\delta\right)\Psi_{\geq\lambda n}.
$$

We can re-write this as

$$
\Psi \leq \Psi_{\geq\lambda n}\frac{1+\delta+3\epsilon}{3\epsilon-\epsilon/3} = C(\epsilon)\Psi_{\geq\lambda n}.
$$

However, we also have that

$$\Psi_{\geq \lambda n} \leq (1-\lambda) n \exp\left(-\alpha y_{\lambda n}\right) \leq (1-\lambda) n \exp\left(\frac{\alpha B}{n\lambda}\right).$$

At the same time, since $y_{3n/4} < 0$, we have that

$$\Phi \geq \frac{n}{4} \exp\left(\frac{3\alpha B}{n}\right).$$

If $\Psi < \frac{\epsilon}{4}\Phi$, we can conclude. Let us examine the case where $\Psi \geq \frac{\epsilon}{4}\Phi$. Putting everything together, we get

$$\frac{\epsilon}{4}\frac{n}{4} \exp\left(\frac{3\alpha B}{n}\right) \leq \frac{\epsilon}{4}\Phi \leq \Psi \leq C(\epsilon)\Psi_{\geq \lambda n} \leq (1-\lambda)\, nC(\epsilon) \exp\left(\frac{\alpha B}{n\lambda}\right).$$

Alternatively,

$$\exp\left(\frac{\alpha B}{n}\left(3 - \frac{1}{\lambda}\right)\right) \leq \frac{16}{\epsilon} C(\epsilon)(1-\lambda) = \frac{16}{\epsilon}(1-\lambda)\frac{1+\delta+3\epsilon}{3\epsilon - \epsilon/3} = O\left(\frac{1}{\epsilon^2}\right).$$

Therefore,

$$\exp\left(\frac{\alpha B}{n}\right) \leq O\left(\frac{1}{\epsilon^{13}}\right).$$

To complete the argument, we bound:

$$\Gamma = \Psi + \Phi \leq \left(1 + \frac{4}{\epsilon}\right)\Psi \leq (1-\lambda)\, C(\epsilon) \exp\left(\frac{\alpha B}{n\lambda}\right) n \leq O\left(\frac{n}{\epsilon^{22}}\right).$$

□

Endgame. We now finally have the required machinery to prove that Γ satisfies a supermartingale property:

Lemma 4.7. *There exists a constant ϵ such that*

$$\mathbb{E}[\Gamma(t+1)|y(t)] \leq \left(1 - \frac{\hat{\alpha}\epsilon}{4n}\right)\Gamma(t) + C, \text{ where } C \text{ is a constant in } O\left(\text{poly}\left(\frac{1}{\epsilon}\right)\right).$$

Proof. **Case 1:** If $y_{n/4} \leq 0$ and $y_{3n/4} \geq 0$, then the property follows by putting together Lemmas 4.3 and 4.4.

Case 2: If $y_{3n/4} \geq y_{n/4} \geq 0$. This means that the weight vector is unbalanced, in particular that there are few bins of low cost, and many bins of high cost. However, we can show that the expected decrease in Ψ can compensate this decrease, and the inequality still holds. Notice that Lemmas 4.1 and 4.2 imply that $\mathbb{E}[\Delta\Phi] \leq \frac{\hat{\alpha}\delta}{n}\Phi(t)$ and $\mathbb{E}[\Delta\Psi] \leq \frac{\hat{\alpha}\delta}{n}\Psi(t)$, respectively.

If $\mathbb{E}[\Delta\Phi] \leq -\frac{\epsilon\hat{\alpha}}{4n}\Phi(t)$, then the claim simply follows by putting this bound together with Lemma 4.4. Otherwise, if $\mathbb{E}[\Delta\Phi] \geq -\frac{\epsilon\hat{\alpha}}{4n}\Phi(t)$, we obtain from Lemma 4.5 that either $\Phi < \frac{\epsilon}{4}\Psi$ or $\Gamma = O(n)$. In the first sub-case, we have that

$$\begin{aligned}
\mathbb{E}[\Delta\Gamma] &= \mathbb{E}[\Delta\Phi] + \mathbb{E}[\Delta\Psi] \\
&\leq \frac{\hat{\alpha}\delta}{n}\Phi(t) + 1 - \frac{\hat{\alpha}\epsilon}{3n}\Psi(t) \\
&\leq -\frac{\hat{\alpha}\epsilon}{3n}\Psi(t) + 1 + \frac{\epsilon\delta\hat{\alpha}}{4n}\Psi(t) \\
&\leq -\frac{\hat{\alpha}\epsilon}{3n}\left(1 - \frac{3\delta}{4}\right)\Psi(t) + 1 \leq -\frac{\hat{\alpha}}{3n}\left(1 + \frac{\epsilon}{4}\right)^{-1}\left(1 - \frac{3\delta}{4}\right)\Gamma(t) \\
&\leq -\frac{\hat{\alpha}}{4n}\Gamma(t),
\end{aligned}$$

as claimed.

In the second sub-case, we know that $\Gamma \leq Cn$, for some constant C. Hence, we can get that

$$\mathbb{E}[\Delta\Gamma] = \mathbb{E}[\Delta\Phi] + \mathbb{E}[\Delta\Psi] \overset{(a)}{\leq} \frac{\hat{\alpha}\delta}{n}\Gamma \overset{\Gamma=O(n)}{\leq} 2C\hat{\alpha},$$

where in step (a) we used the upper bounds in Lemmas 4.1 and 4.2, respectively. On the other hand,

$$C - \frac{\hat{\alpha}\epsilon}{4n}\Gamma(t) \geq C - \frac{\hat{\alpha}\epsilon}{4n}Cn = C\left(1 - \frac{\hat{\alpha}\epsilon}{4}\right) \geq 3C\hat{\alpha} \geq \mathbb{E}[\Delta\Gamma].$$

Case 3: If $y_{n/4} \leq y_{3n/4} < 0$. This case is symmetric to the one above. □

The intuition behind the above bound is that Γ will always tend to decrease once it surpasses the $\Theta(n)$ threshold. This implies a bound on the expected value of Γ, which completes the proof of Theorem 4.4.

Lemma 4.8. *For any $t \geq 0$, $\mathbb{E}[\Gamma(t)] \leq \frac{4C}{\hat{\alpha}\epsilon}n$.*

Proof. By induction. This holds for $t = 0$ by a direct computation: see Lemma 4.7 for the argument. Then, we have

$$\begin{aligned}
\mathbb{E}[\Gamma(t+1)] &= \mathbb{E}[\mathbb{E}[\Gamma(t+1)|\Gamma(t)]] \\
&\leq \mathbb{E}\left[\left(1 - \frac{\hat{\alpha}\epsilon}{4n}\right)\Gamma(t) + C\right] \\
&\leq \frac{4C}{\hat{\alpha}\epsilon}n\left(1 - \frac{\hat{\alpha}\epsilon}{4n}\right) + C \\
&\leq \frac{4C}{\hat{\alpha}\epsilon}n.
\end{aligned}$$

□

4.4.3 Additional Guarantees on Max Rank

We can use the characterization of the exponential process to prove the following:

Lemma 4.9. *If $w_{\max}(t)$ is the maximum bin weight at time t and $w_{\min}(t)$ is the minimum, then*

$$\mathbb{E}\left[w_{\max}(t) - w_{\min}(t)\right] = O\left(\frac{1}{\alpha}n(\log n + \log C)\right). \tag{4.17}$$

Proof. Let $x_{\max}(t) = w_{\max}(t)/n$ and $x_{\min}(t) = w_{\min}(t)/n$. By definition, we have $\exp(\alpha(x_{\max}(t) - \mu(t))) \leq \Gamma(t)$ and $\exp(\alpha(\mu(t) - x_{\min}(t))) \leq \Gamma(t)$. Therefore $\alpha(x_{\max}(t) -$

$x_{\min}(t)) \leq 2\log\Gamma(t)$. Thus, we have

$$\mathbb{E}[\alpha(x_{\max}(t) - x_{\min}(t))] \leq 2\mathbb{E}[\log\Gamma(t)] \overset{(a)}{\leq} 2\log(\mathbb{E}[\Gamma(t)]) \overset{(b)}{=} O(\log n + \log C) ,$$

where (a) follows from Jensen's inequality and (b) follows from Theorem 4.4. Simplifying yields the desired claim. □

We will show that this implies the following theorem:

Theorem 4.6. *For all $t \geq 0$, we have $\mathbb{E}[\max\operatorname{rank}(t)] = O\left(\frac{1}{\alpha}n(\log n + \log C)\right)$.*

Proof. We will need the following fact about exponential disitributions:

Fact 4.2. *Let $X_1, X_2, \dots$ be independent and $X_i \sim \operatorname{Exp}(\lambda)$ for all i. Let $Y_i = \sum_{k \leq i} X_k$. Fix any interval $I \subseteq [0, \infty)$ of length m. Then, $\#\{i : Y_i \in I\} \sim \operatorname{Poi}(m\lambda)$.*

Let $I(t) = [w_{\min}(t), w_{\max}(t)]$, and let $L_j(t)$ be the number of elements in bin j in $I(t)$ at time t. Then, by memorylessness and Fact 4.2, for all bins j except the bin $j_{\max}$ containing $w_{\max}(t)$, the number of labels in $I(t)$ in bin j before any deletions occur is distributed as $\operatorname{Poi}(|I|/n)$. In particular, the expected number of elements after t rounds is bounded by $\mathbb{E}_{X \sim \operatorname{Poi}(|I|/n)}[X] = |I|/n$. Moreover, for the bin containing $w_{\max}(t)$, by definition, at time t it contains exactly one element in I, namely, $w_{\max}(t)$. Hence, we have

$$\begin{aligned}
\mathbb{E}[\max\operatorname{rank}(t)] &\leq \mathbb{E}_{I(t)}\left[\mathbb{E}\left[\sum_{j=1}^{n} L_j(t) \middle| I(t)\right]\right] \\
&\leq \mathbb{E}_{I(t)}\left[1 + (n-1)|I|/n\right] \\
&= O\left(\frac{1}{\alpha}n(\log n + \log C)\right) ,
\end{aligned}$$

by Lemma 4.9. □

By plugging in constants as in (4.1) and (4.2), we get:

Corollary 4.1. *For all $t \geq 0$, and any $\beta = \Omega(\gamma)$:*

$$\mathbb{E}[\max\operatorname{rank}(t)] = O\left(\frac{1}{\beta}n(\log n + \log 1/\beta)\right) .$$

4.4.4 Fairness

Having established $O(n \log n)$ error boundedness in Theorem 4.5, we can show that MultiQueues are $O(n \log n)$-fair relatively straightforwardly.

Theorem 4.7. *The* MultiQueue *algorithm is $O(n \log n)$ fair.*

Proof. By Theorem 4.5, MultiQueues are $\kappa = cn \log n$-error bounded, for some constant c. We will show that MultiQueues are 4κ-fair. We need to bound the probability that an element u suffers at least $4\kappa r$ inversions by $O(\exp(-r))$. We will proceed by taking a union bound over three possible bad events, at least one of which must occur for u to suffer $4\kappa r$ inversions. In particular, we will show that it is unlikely that any of the following occurs: (1) u suffers an inversion before u itself is rank κr, (2) if $u \in Q_i$, then at the time u becomes rank κr, there are more than $2\kappa r/n$ elements of smaller rank in Q_i, or (3) after u becomes rank κr, more than $4r\kappa$ queues with a higher rank top element than Q_i are deleted from before Q_i is deleted from $2\kappa r/n$ times.

Early inversions. Let E_1 be the event that u suffers any inversions before becoming rank κr. Notice that the largest rank element which could have been deleted so far is upper bounded by the largest element which is already at the top of some queue. Thus in order for E_1 to occur, there must exist a queue Q_j whose top element has rank greater than u, i.e. rank at least κr. By Theorem 4.5,

$$\Pr[E_1] \leq \Pr[y_n(t) \geq \kappa r] < \exp(-r).$$

Depth. Let E_2 be the event that there are more than $2\kappa r/n$ elements of smaller rank in Q_i when u has rank κr. Since elements of smaller rank than u are distributed among queues independently, the number of smaller rank elements in Q_i is binomially distributed with mean $\kappa r/n = \Theta(r \log n)$, so a standard Chernoff bound gives

$$\Pr[E_2] = \Pr[X > 2\mathbb{E}[X]] < \exp(-\Theta(\mathbb{E}[x])) = \exp(-\Theta(r \log n)) < \exp(-r).$$

Waiting time. Finally, let the random variable X be the number of times that queues with a higher rank top element than Q_i (which we call 'worse' queues than Q_i) are deleted from before Q_i is deleted from $2\kappa r/n$ times, starting from when u becomes rank κr. Let E_3 be the event that $X > 4\kappa r$. Note that it is impossible for u to suffer an inversion if a queue with a *lower* rank top element than Q_i is deleted from, so E_3 is a necessary event for u to suffer more than $4\kappa r$ inversions, given $\bar{E}_1, \bar{E}_2$. The probability of some queue worse than Q_i being deleted from is at most np_i, since the p_i are decreasing with rank of top element. Thus X is upper bounded by a random variable X' distributed according to a negative binomial parameterized by $2\kappa r/n$ successes and success probability $1/n$. Notably, $\mathbb{E}[X'] = 2\kappa r$. As above, a standard Chernoff bound on negative binomial distributions gives

$$\begin{aligned}
\Pr[X > 2\kappa r] &< \Pr[X' > 2\kappa r] \\
&= \Pr[X' > 2\mathbb{E}[X']] \\
&< \exp(-\Theta(\mathbb{E}[X'])) \\
&= \exp(-\Theta(rn\log n)) \\
&< \exp(-r).
\end{aligned}$$

Union bounding, the probability that u suffers more than $4\kappa r$ inversions is bounded by $\Pr[E_1] + \Pr[E_2] + \Pr[E_3] = O(\exp(-r))$, as required.

□

By definition, Theorem 4.5 and Theorem 4.7 imply $O(n \log n)$-relaxation:

Theorem 4.8. *The* MultiQueue *algorithm is $O(n \log n)$-relaxed.*

4.4.5 Guarantees on Average Rank

We now focus on at the rank cost paid in a step by the algorithm. Let $A = \log C/\alpha$. For real values $s \geq 0$, we "stripe" the bins according to their top value, denoting by $b_{>s}(t)$ the number of bins with $w_j(t) \geq (s + A)n + \mu$ at time t, and let $b_{<-s}(t)$ be the number of bins with $w_j(t) \leq \mu - (s + A)n$ at time t. For any bin j and interval I, we also let $\ell_{j,I}(t)$ be the number of elements in j at time t with label in I. Finally, let $p_{j,\mu}$ denote the PDF of $w_j(t)$ given μ_t.

First, we use the bounds on Γ to obtain the following bounds on the quantities defined previously:

Lemma 4.10. *For any time t, we have that $\mathbb{E}[b_{>s}] \leq n \exp(-\alpha s)$ and that $\mathbb{E}[b_{<-s}] \leq n \exp(-\alpha s)$.*

Proof. Recall that $\Phi(t) = \sum_{i=1}^{n} \exp(\alpha(x_i(t) - \mu))$ and that $\mathbb{E}[\Phi(t)] \leq Cn$. By linearity of expectation, we have

$$\mathbb{E}[\Phi(t)] \geq \mathbb{E}[b_{>s} \exp(\alpha(s + A))] = C \exp(\alpha s) \cdot \mathbb{E}[b_{>s}],$$

which implies the claim. The converse claim follows from the bound on $\Psi(t)$. □

This lemma gives us strong bounds on the tail behavior of the w_j. We show that this implies a bound on the average rank, in two steps. First, we show that this implies a bound on the rank of μ:

Lemma 4.11. *For all t, we have $E[\text{rank}(\mu(t))] \leq O((A + 1/\alpha^2)n)$.*

Proof. For any bin j, we have

$$\begin{aligned}
\mathbb{E}[\ell_{j,(-\infty,\mu]}(t)|\mu] &= \int_{-\infty}^{\mu} \mathbb{E}[\ell_{j,[x,\mu]}|\mu, w_j(t) = x] \cdot p_{j,\mu}(x)dx \\
&\overset{(a)}{\leq} \int_{-\infty}^{\mu} \mathbb{E}_{X \sim \text{Poi}(\mu - x)/n}[1 + X|\mu, w_j(t) = x] \cdot p_{j,\mu}(x)dx \\
&= \int_{-\infty}^{\mu} \frac{1}{n}(1 + \mu - x) \cdot p_{j,\mu}(x)dx = \frac{1}{n}\mathbb{E}[(1 + \mu - w_j(t))1_{w_j(t) \leq \mu}|\mu] \\
&\leq \frac{1}{n}\left(1 + \mathbb{E}[(\mu - w_j(t))1_{w_j(t) \leq \mu}|\mu]\right) .
\end{aligned}$$

where (a) follows because after we've conditioned on the value of $w_j(t)$, the values of the remaining labels in bin j is independent of μ and we can apply Fact 4.2. Therefore, we have

$$\begin{aligned}
\mathbb{E}[\mathrm{rank}(\mu)] &\overset{(a)}{\leq} \mathbb{E}[\mathrm{rank}(\mu - An)] + (A+1)n \\
&= \mathbb{E}\left[\sum_{j=1}^{n} \mathbb{E}[\ell_{j,(-\infty,\mu-An](t)}|\mu]\right] + (A+1)n \\
&= \mathbb{E}\left[\sum_{j=1}^{n} \mathbb{E}[\ell_{j,(w_j(t),\mu-An](t)}|\mu, w_j(t)]\right] + (A+1)n \\
&\overset{(b)}{\leq} 1 + \frac{1}{n}\mathbb{E}_\mu\left[\sum_{j=1}^{n}(\mu - w_j(t))1_{w_j(t)\leq\mu-An}\right] + (A+1)n \\
&\overset{(c)}{\leq} 1 + \frac{1}{n}\sum_{k=1}^{\infty}\mathbb{E}_\mu\left[\sum_{w_j(t)\in[\mu-(k+1+A)n,\mu-(k+A)n]}(\mu - w_j(t)+1)\right] + (A+1)n \\
&\leq 1 + \frac{1}{n}\sum_{k=1}^{\infty}(k+2)n\mathbb{E}_\mu\left[b_{<-i}\right] + (A+1)n \\
&\overset{(d)}{\leq} 1 + \sum_{k=1}^{\infty}(k+2)n\exp(-\alpha k) + (A+1)n \\
&\leq 1 + \frac{e^\alpha}{(e^\alpha-1)^2}n + (A+1)n\ ,
\end{aligned}$$

where (a) follows from Lemma 4.13, (b) follows from Fact 4.2, (c) follows from Lemma 4.13, and (d) follows from Lemma 4.10. Notice that for α small we have $\frac{e^\alpha}{(e^\alpha-1)^2} = O(1/\alpha^2)$. Simplifying the above then yields the claimed bound. □

The following simple lemma bounds the initial potential:

Lemma 4.12. *We have that* $\Gamma(0) = O(n)$.

Proof. In the initial state, the values $x_i(0)$ are independent exponentially distributed random variables with mean $1/(n\pi_i)$. Now,

$$\begin{aligned}
\mathbb{E}[\Phi(0)] &= \mathbb{E}\left[\sum \exp\left(\alpha\left(\frac{w_i(0)}{n} - \mu(0)\right)\right)\right] \\
&= \sum \mathbb{E}\left[\exp\left(\alpha\left(x_i(0) - \frac{1}{n}\sum x_j\right)\right)\right] \\
&= \sum \mathbb{E}\left[\exp\left(\alpha x_i(0)\left(1 - \frac{1}{n}\right)\right)\prod_{j\neq i}\exp\left(-\frac{\alpha}{n}x_j(0)\right)\right]
\end{aligned}$$

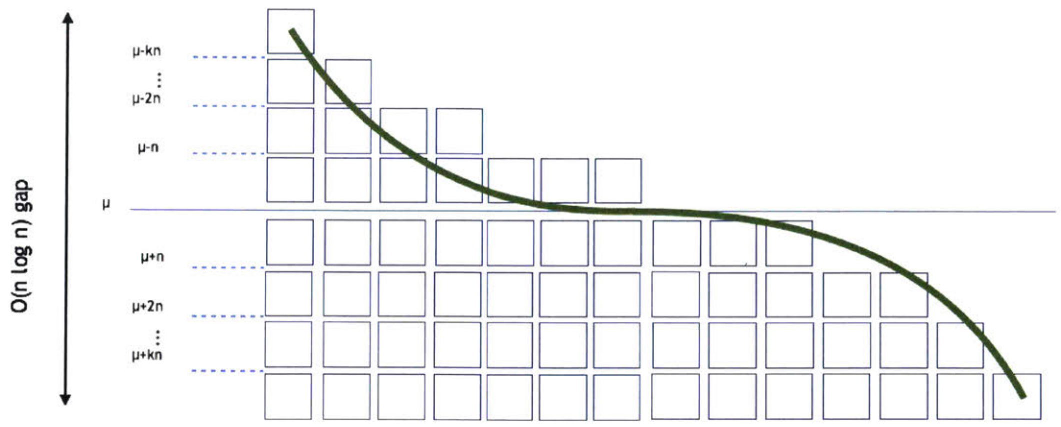

Figure 4-1: Intuition for the average rank bound. The number of bins per stripe decreases exponentially as the stripes get further from the mean.

$$= \sum \mathbb{E}\left[\exp\left(\alpha x_i(0)\left(1-\frac{1}{n}\right)\right)\right]\mathbb{E}\left[\prod_{j\neq i}\exp\left(-\frac{\alpha}{n}x_j(0)\right)\right],$$

where the last line follows from the (pairwise) independence of the x_i. All terms in the right hand side are now exactly moment generating functions of the w_i evaluated at some point. Since the moment generating function of an exponential with parameter λ evaluated at t is well known to be $\lambda/(\lambda - t)$, we can compute:

$$\begin{aligned}
\mathbb{E}[\Phi(0)] &= \sum \frac{\frac{1}{n\pi_i}}{\frac{1}{n\pi_i} - \alpha(1-\frac{1}{n})}\prod_{j\neq i}\frac{\frac{1}{n\pi_j}}{\frac{1}{n\pi_j}+\frac{\alpha}{n}} \\
&\le \sum \frac{1+\gamma}{1-\gamma-\alpha}\prod_{j\neq i}\frac{1}{1+\alpha\pi_j} \\
&\approx \sum \frac{1+\gamma}{1-\gamma-\alpha}\left(1-\sum_{j\neq i}\alpha\pi_j\right) \\
&= O(n).
\end{aligned}$$

The argument for $\Psi(0)$ is symmetric, replacing α with $-\alpha$.

□

Finally, we show that this implies:

Theorem 4.9. *For all t, we have $\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\text{rank}_j(t)\right] = O\left(A + \frac{1}{\alpha^2}\right)n$.*

Intuitively, the previous lemma controlled the ranks of all queues with $w_j(t) \le \mu$.

Figure 4-1 gives some visual intuition: the number of bins in each stripe should decrease exponentially as the stripes move away from the mean. By using a similar "striping" argument, except on the interval $[\mu, \infty)$, we can show the same for the other side, which gives the desired bound.

To prove Theorem 4.9, we will need the following technical lemma:

Lemma 4.13. *For any interval* $I = [a, b]$ *which may depend on* μ, *we have* $\mathbb{E}[\text{rank}(b) - \text{rank}(a)|\mu] \leq n(b - a + 1)$.

Proof. We first show that if I does not depend on μ, then $\mathbb{E}[\text{rank}(b) - \text{rank}(a)|\mu] \leq n(b - a)$. By Fact 4.2 we have

$$\begin{aligned}\mathbb{E}[\text{rank}(b) - \text{rank}(a)|\mu] &\leq \mathbb{E}\left[\sum_{j=1}^{n} \ell_{j,[a,b]} \middle| \mu\right] \\ &= \mathbb{E}\left[\sum_{j=1}^{n} \mathbb{E}_{X \sim \text{Poi}(b-a)}[X] \middle| \mu\right] \\ &= n(b-a) \ .\end{aligned}$$

To conclude the proof, we now observe that μ depends on at most n elements, namely, those on top of the queues, and that if we remove those elements, then the remaining elements behave just as above. Thus, by conditioning on μ, we increase the rank by at most an additional factor of n. □

We now prove Theorem 4.9:

Proof. By Lemma 4.11 and Lemma 4.13, we have that

$$\begin{aligned}\mathbb{E}[\text{rank}_j(t)|w_j(t) \leq \mu(t) + An] &\leq O\left((A + 1/\alpha^2)n\right) + (A+1)n \\ &= O\left((A + 1/\alpha^2)n\right) \ .\end{aligned}$$

Let $I_k = [\mu + (A+k)n, \mu + (A+k+1)n]$ be the k^{th} "stripe". We have

$$\mathbb{E}\left[\sum_{i=1}^{n} \text{rank}_j(t)\right] = \mathbb{E}\left[\sum_{w_j(t) \leq \mu + An} \mathbb{E}[\text{rank}_j(t)|\mu] + \sum_{w_j(t) > \mu + An} \mathbb{E}[\text{rank}_j(t)|\mu]\right]$$

$$
\begin{aligned}
&= \mathbb{E}\left[\sum_{w_j(t)\le \mu+An} \mathbb{E}[\mathrm{rank}_j(t)|\mu] \right.\\
&\quad \left. + \sum_{j=1}^{n} \sum_{w_j(t)>\mu+An} \mathbb{E}[\mathrm{rank}_j(\mu+An) + \ell_{j,(\mu,w_j(t))}|\mu, w_j(t)]\right]\\
&\overset{(a)}{\le} O\left(\left(A+\frac{1}{\alpha^2}\right)n^2\right) + \sum_{j=1}^{n}\mathbb{E}\left[\sum_{w_j(t)>\mu+An} \mathbb{E}[\ell_{j,(\mu,w_j(t))}|\mu, w_j(t)]\right]\\
&\overset{(b)}{\le} O\left(\left(A+\frac{1}{\alpha^2}\right)n^2\right) + \sum_{j=1}^{n}\mathbb{E}\left[\sum_{w_j(t)>\mu+An} (w_j(t)-\mu+1)\right]\\
&= O\left(\left(A+\frac{1}{\alpha^2}\right)n^2\right) + \sum_{j=1}^{n}\sum_{k=0}^{\infty}\mathbb{E}\left[\sum_{w_j(t)\in I_k} (w_j(t)-\mu+1)\right]\\
&\le O\left(\left(A+\frac{1}{\alpha^2}\right)n^2\right) + n\sum_{k=0}^{\infty}(A+k+2)n\mathbb{E}\left[b_{>k}\right]\\
&\overset{(c)}{\le} O\left(\left(A+\frac{1}{\alpha^2}\right)n^2\right) + O\left(\frac{1}{\alpha^2}n^2\right).
\end{aligned}
$$

where (a) follows by Lemma 4.11, (b) follows by Lemma 4.13, and (c) follows from Lemma 4.10. Thus

$$
\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\mathrm{rank}_j(t)\right] = O\left(A+\frac{1}{\alpha^2}\right)n
$$

as claimed. □

We now show how these imply bounds for our removal processes. The actual rank choice at time t is always better than a uniform choice in expectation, since it uses power of two choices. If we consider an $(1+\beta)$ process, where we only do two choices with probability $\beta = \Omega(\gamma)$, we obtain the following, by setting parameters as in (4.1) and (4.2) :

Corollary 4.2. *For all t, if we let $\beta = \Omega(\gamma)$, and we let $r(t)$ denote the rank of the removed element at time t, then*

$$
\mathbb{E}[r(t)] = O\left(\left(\frac{\log C}{\alpha} + \frac{1}{\alpha^2}\right)n\right) = O\left(\frac{n}{\beta^2}\right).
$$

4.5 Discussion and Future Work

We have provided tight rank guarantees for a practically-inspired randomized priority scheduling process. Moreover, we showed that this strategy is robust in terms of its bias and randomness requirements. Intuitively, our results show that, given biased random insertions into the queues, the preference towards lower ranks provided by the two-choice process is enough to give strong linear bounds on the average rank removed. We extend our analysis to a practical algorithm which improves on the state-of-the-art MULTIQUEUES in Section 4.6.

Tightness. The bounds we provide for the two-choice process are asymptotically tight. To see that the $O(n)$ bound is tight, it suffices to notice that, even in a best-case scenario, the rank of the kth least expensive queue is at least k. Hence, the process has to have expected rank cost $\Omega(n)$. The tightness argument for $\Theta(n \log n)$ expected worst-case cost is more complex. In particular, it is known [60, Example 2] that the gap between the most loaded and average bin load in a weighted balls-into-bins process with weights coming from an exponential distribution of mean 1 is $\Theta(\log n)$ in expectation. That is, there exists a queue j from which we have removed $\Theta(\log n)$ fewer times than the average. We can extend the argument in Section 4.4.5 to prove that there exist $\Theta(n)$ queues which have $\Theta(\log n)$ elements of higher label than the top element of queue j. This implies that the *rank cost* of queue j is $\Theta(n \log n)$, as claimed. We conjecture that the dependency in β for the expected rank bound on the $(1 + \beta)$ process can be improved to *linear*.

Relation to Concurrent Processes. On first glance, it might seem that a simple lock-based strategy should be linearizable to the sequential process we define in Section 4.3. For example, we could lock both queues that are to be examined (locking in order of their index to avoid deadlock and restarting the operation on failure) and declare the linearization point to be the point at which the second lock is grabbed. While this does linearize to *some* relaxed sequential process, it turns out that our upper bounds fail to hold for subtle reasons when concurrency is introduced.

Consider the extreme execution in which thread 0 grabs the locks on some two queues, say Q_i, Q_j, and then hangs for a long time. Meanwhile, all the other threads complete many operations while thread 0 holds these locks, and all operations will have to retry if they try to grab locks on Q_i or Q_j. In this case, many delete operations will be performed, none of which can delete from Q_i or Q_j. Such an execution could produce arbitrarily bad rank errors.

The simple locking strategy fails to be distributionally linearizable due to, e.g., the counter-example execution above. We would like to ask if there is a distributionally linearizable strategy, yet there appears to be an inherent limitation. Consider three threads i, j, k performing DELETE operations in lock-step. No matter what strategy is used, if some two threads, say i, j, try to delete from the same queue Q, at least one, say j, will necessarily be delayed. As a result i and k will finish their operations while j takes longer, causing i and k to be linearized before j (this can be forced by e.g. an insertion to Q while j is delayed). As a result, an additional constraint has been introduced which requires that the first two DELETE operations to complete in this execution cannot have deleted from the same queue, a constraint which is not present for the sequential process.

While beyond the scope of this thesis, upcoming work [3] closes this gap, showing that MULTIQUEUEs still retain $O(n \log n)$-relaxation, even in a fully asynchronous setting against an adversarial scheduler, albeit with a (large) constant factor blow up in the number of queues required by the analysis.

4.6 Experimental Results

Setup and Methodology. We implemented a $(1 + \beta)$ priority queue based on the MULTIQUEUE implementation from the priority queue benchmark framework of [74], and benchmarked it against the original MULTIQUEUE ($\beta = 1$), the Lindén-Jonsson [51] skiplist-based implementation, and the kLSM deterministic-relaxed data structure [74], with a relaxation factor of 256, which has been found to perform best. The MULTIQUEUE uses efficient sequential priority queues from the boost library. Tests were performed on a recent Intel(R) Xeon(R) CPU E7-8890 (Haswell architecture), with 18 hardware threads, each running at 2.5GHz. The tests for mean rank returned use coherent timestamps to record the times when elements are returned at each thread. We use these in a post-processing step to count rank inversions. This methodology might not be 100% accurate, since the use of timestamps might change the schedule; however, we believe results should be reasonably close to the true values.

For the throughput experiments, we consider executions consisting of alternating insert and GETMIN() or APPROXGETMIN() operations, for 10 seconds. Experiment outputs are averaged over 10 trials. Removals on empty queues do not count towards throughput. Since we are interested in the regime where queues are never empty, we insert 10 million elements initially. Threads are pinned to cores, and memory allocation is affinitized. The single-source shortest paths benchmark is a version of Dijkstra's algorithm, running on a weighted, directed California road network graph.

Results. Figure 4-2 illustrates the throughput differential between the various implementations, As previously stated, the MULTIQUEUE variants are superior to other implementations (except at very low thread counts). Of note, the variants with $\beta < 1$ improve on the standard implementation by up to 20%. Since throughput figures are not conclusive in isolation, we also benchmarked the average rank cost in Figure 4-3. (The y axis is logarithmic.) Note that the increase in average cost due to the further β relaxation is relatively limited. Results are conformant with our analysis for $\beta \geq 0.5$. The apparent inflection point at around $\beta = 0.5$ could be explained by the $\epsilon \geq \delta$ bias assumptions breaking down after this point, or by non-trivial correlations in the actual execution which mean that our analysis no longer applies.

Finally, Figure 4-4 gives a running times for a single-source shortest path benchmark, using a parallel version of Dijkstra's algorithm. We note that the relaxed versions with $\beta < 1$ can be superior in terms of running time to the version with $\beta = 1$, by up to 10%. The version with $\beta = 0$ (not shown) is the fastest at low thread counts, but then loses performance at thread counts ≥ 8, probably because of excessive relaxation.

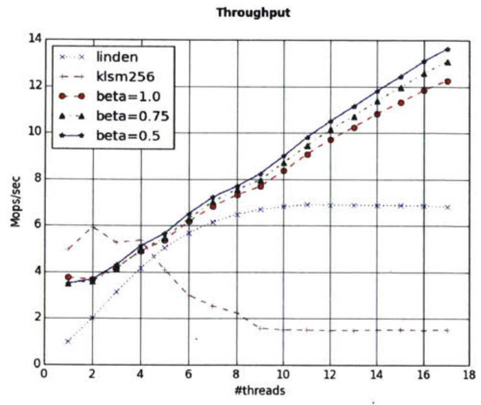

Figure 4-2: Throughput comparison for the $(1 + \beta)$ priority queue with $\beta = 0.5$ and 0.75, versus the original MULTIQUEUES, the Lindén-Jonsson implementation, and kLSM. Higher is better.

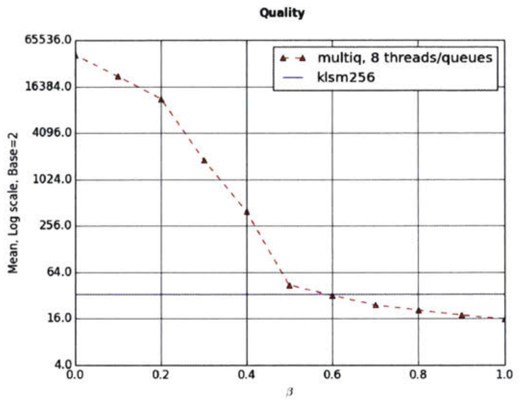

Figure 4-3: Mean rank returned (log scale) for the $(1 + \beta)$ priority queue, for various values of β on 8 queues and 8 threads. Lower is better.

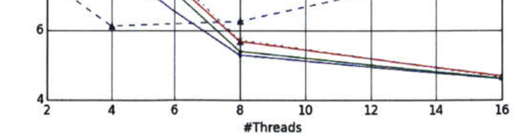

Figure 4-4: Running times for single-source shortest path benchmark, using various versions of the priority queue, and kLSM. Lower is better.

Chapter 5

Executing Iterative Algorithms Using Relaxed Priority Schedulers

In this chapter, we describe how to execute generic problems which can be formulated in terms of a 'Task-Queue'. In particular, we consider problems which can be described as a set, S, of tasks, a dependency graph $G = (V, E)$ with $V = S$ representing dependencies between tasks, and a permutation π on S which represents an *ordering* between tasks. Generally, we will think of G as being inherently undirected but edges have an *orientation* induced by π; that is, if $\pi(u) < \pi(v)$, then an edge $e = (u, v)$ is directed from u to v.

This formulation encompasses a large class of problems, including greedy graph algorithms (coloring, independent set, matching, etc.), Dijkstra's algorithm for shortest paths, List Contraction, and discrete time simulations (e.g. n-body simulation). See Section 5.1 for details.

We will argue that all Task-Queue based problems can be executed *correctly* by a relaxed scheduler, provided that G is explicit (or at least supports sufficient edge queries). Furthermore, we show that when π is uniformly random (as in many greedy algorithms) and G is *sparse*, i.e. $|E| = o(|V|^2)$, the overhead incurred by potential dependency violations due to the relaxed nature of the scheduler is *negligible*, in that it depends only on the relaxation factor k of the scheduler, and not the number of tasks.

Surprisingly, we also show the counter-intuitive result that greedy Maximal Independent Set and Maximal Matching have exploitable substructure which allows them to be executed by k-relaxed schedulers with negligible overhead (i.e. overhead proportional only to k), *irrespetive* of the size or structure of G. This result offers a compelling argument for the use of relaxed schedulers and their characteristic high throughput over exact schedulers in the context of Maximal Independent Set and Matching.

Related Work. The ideas in this chapter are in part inspired by the line of research by Blelloch et al. [12, 13, 14, 44, 69], as well as [21, 22, 30], whose broad goal has been to examine the dependency structure of a wide class of iterative algorithms, and to derive efficient scheduling mechanisms given such structure.

At the same time, there are several differences between these results and our work. First, at the conceptual level, [14, 69] start from analytical insights about the dependency structure of algorithms such as greedy MIS, and apply them to design scheduling mechanisms which can leverage this structure, which require problem-specific information. In some cases, e.g. [14], the scheduling mechanisms found to perform best in practice differ from the structure of the schedules analyzed. By contrast, we start from a realistic model of existing high-performance relaxed schedulers [64], and show that such schedulers can automatically and efficiently execute a broad set of iterative algorithms. Second, at the technical level, the methods we develop are *different*: for instance, the fact that the iterative algorithms we consider have low dependency depth [12, 14, 69] does not actually help our analysis, since a sequential algorithm could have low dependency depth and be inefficiently executable by a relaxed scheduler: the bad case here is when the dependency depth is low (logarithmic), but each "level" in a breadth-first traversal of the dependency graph has high fanout. Specifically, we emphasize that the notion of *prefix* defined in [14] to simplify analysis is *different* from the set of positions S which can be returned by the relaxed stochastic scheduler: for example, the parallel algorithm in [14] requires each prefix to be fully processed before being removed, whereas S acts like a sliding window of positions in our case. The third difference is in terms of analytic model: references such as [14] express work bounds in the CRCW PRAM model, whereas we count work in terms of number of tasks processing attempts. Our analysis is sequential, and we implement our algorithms on a shared memory architecture to demonstrate empirically good performance.

To our knowledge, the first instance of a relaxed scheduler is in work by Karp and Zhang [45], for parallelizing backtracking strategies in a (synchronous) PRAM model. This area has recently become extremely active, with several such schedulers (also called relaxed priority queues) being proposed over the past decade, see [4, 6, 9, 36, 53, 58, 64, 66, 74] for recent examples. In particular, we note that state-of-the-art packages for graph processing [58] and machine learning [33] implement such relaxed schedulers.

Parallel scheduling [16, 17] is an extremely vast area and a complete survey is beyond our scope. We do wish to emphasize that standard work-stealing schedulers *will not* provide this type of work bounds, since they do not provide any guarantees in terms of the *rank of elements removed*: the rank becomes unbounded over long executions, since a single random queue is sampled at every stealing step [4]. To our knowledge, there is only one previous attempt to add priorities to work-stealing schedulers [43], using a multi-level global queue of tasks, partitioned by priority. This technique is different, and provides no work guarantees.

5.1 A General Scheduling Framework

We now present our framework for executing task-queue based sequential programs, whose pseudocode is given in Algorithm 5. We assume a permutation π which dictates an execution order on tasks. If u is the i^{th} element in π, we will write $\pi(i) = u$ and $\ell(u) = i$ (ℓ for *label*). Algorithm 5 encapsulates a large number of common iterative algorithms on graphs, including Greedy Vertex Coloring, Greedy Matching, Greedy Maximal Independent Set, Dijkstra's SSSP algorithm, and even some algorithms which are not graph-based, such as List Contraction and Knuth Shuffle [12]. We show sample instantiations of the framework in Section 5.1.1.

Algorithm 5 Generic Task-Queue Framework

```
Require: Dependency Graph $G = (V, E)$
Require: Vertex permutation $\pi$
Require: Instantiated exact priority queue, $Q$
  $Q \leftarrow$ vertices in $V$ with priorities $\pi(V)$
  for each step $t$ do
      // Get new element from the buffer
      $v_t \leftarrow Q$.GetMin()
      Process($v$)
      if $Q$.empty() then
          break
```

Algorithm 6 *Relaxed* Scheduling Framework

```
Require: Dependency Graph $G = (V, E)$
Require: Vertex permutation $\pi$
Require: Instantiated k-relaxed priority queue, $Q$
  $Q \leftarrow$ vertices in $V$ with priorities $\pi(V)$
  for each step $t$ do
      // Get new element from the buffer
      $v_t \leftarrow Q$.ApproxGetMin()
      if $v_t$ has an unprocessed predecessor then
          $Q.insert(v_t, \pi(v_t))$ // Failed; reinsert
          continue
      else
          Process($v$)
      if $Q$.empty() then break
```

Algorithm 6 gives a method for adapting Algorithm 5 to use a *relaxed* queue, given an explicit dependency graph $G = (V, E)$ whose nodes are the tasks, and whose edges are dependencies between tasks. Importantly, given the dependency graph G, Algorithm 6 gives the same output as Algorithm 5, irrespective of the relaxation factor k. Unlike Algorithm 5, Algorithm 6 is *inherently parallel*, correctness is maintained even when iterations of the for loop are executed concurrently, regardless of the scheduling of iterations. As usual, we write $|V| = n$ and $|E| = m$. We assume that the permutation π represents a *priority order* so that an edge $e = (u, v) \in E$ means that v depends on u if $\ell(v) > \ell(u)$ and vice-versa. In the former case, we say that v is a *successor* of u and u is a *predecessor* of v.

Our main result regarding Algorithm 6, proven formally in Section 5.2.1, argues that if π is chosen uniformly at random from among all vertex permutations, then Algorithm 6 completes in at most $n + O(\frac{m}{n} poly(k))$ iterations (compared to exactly n for Algorithm 5). This result demonstrates that provided G is not too dense, the "cost of relaxation" is low for the class of problems which admit uniformly random task permutations. Notably, this class includes all of the problems mentioned above, except for Dijkstra's algorithm (since there, π needs to respect the ordering of nodes sorted by distance from the source).

5.1.1 Example Applications

Applying the sequential task-queue framework of Algorithm 5 only requires an implementation of Process(v). Implementing the relaxed framework in Algorithm 6 further requires G (either explicitly or via a predecessor query method). We now give examples for Greedy Vertex Coloring and List Contraction.

Greedy Vertex Coloring. Vertex Coloring is the problem of assigning a *color* (represented by a natural number) to each vertex of the input graph, G, such that no adjacent vertices share a color. The Greedy Vertex Coloring algorithm simply processes the vertices in some permutation order, π, and assigns each vertex in turn the smallest available color. The implementation of Process(v) for Greedy Vertex

Coloring needs to determine the color of v, which can be done as described below:

Algorithm 7 Vertex processing subroutine for greedy graph coloring

```
Require: Input Graph $G = (V, E)$
Require: Permutation $\pi$
Require: Partial coloring $c : V \to \mathbb{N}$
  function Process(v)
      $S \leftarrow \emptyset$
      for all $(u, v) \in G$, s.t. $\ell(u) < \ell(v)$ do
          $S \leftarrow S \cup \{c(u)\}$
      $c(v) \leftarrow \min_{i \in \mathbb{N}} i \notin S$
```

Since the underlying dependency graph is just the input graph with edge orientations given by π, this is all that needs to be provided.

List Contraction. List Contraction takes a doubly linked list, L, and iteratively *contracts* its nodes. Contracting a node v consists of setting v.next.previous $\leftarrow$ v.previous and v.previous.next $\leftarrow$ v.next, effectively removing v from the list. List Contraction is useful, e.g., for cycle counting. Although List Contraction is not inherently a graph problem, we can still construct a dependency graph G whose nodes are list elements and with an edge between elements which are adjacent in L. Then a predecessor query on v consists of checking whether either v.next or v.prev is an unprocessed predecessor. Process(v) can be implemented with just the two steps of contraction above (possibly along with the metrics the application is computing).

5.1.2 Greedy Maximal Independent Set

We give a variant of Algorithm 6 adapted for Greedy Maximal Independent Set (MIS), which makes use of some exploitable substructure. In particular, once some neighbor, u, of a vertex v is added to the MIS, then v can never be added to the MIS, at which point v's dependents no longer have to wait for v to be processed. Algorithm 8 implements MIS in the framework of Algorithm 6 while also making use of this observation. Interestingly, Algorithm 8 can also be used to find a maximal

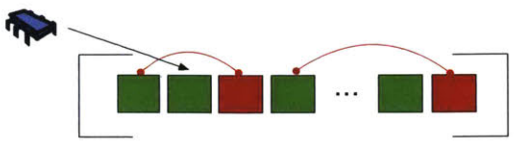

Figure 5-1: Simple illustration of the process. The blue thread queries the relaxed scheduler, which returns one of the top k tasks, on average (in brackets). Some of these tasks (green) can be processed immediately, as they have no dependencies. Tasks with dependencies (red) cannot be processed yet, and therefore result in failed removals.

matching by taking the input graph G of the matching instance and converting it to a graph G', where G' has a vertex for each edge in G and the edges are defined in the straightforward way. (One can view matching as an "independent set" of edges, no two of which are incident to the same vertex.) Like Algorithm 6, Algorithm 8 is inherently parallel; any concurrent scheduling of the iterations of the for loop will result in correct output.

Algorithm 8 Relaxed Queue MIS

```
Require: Graph $G = (V, E)$
Require: Vertex permutation $\pi$
Require: Instantiated k-relaxed priority queue, $Q$
  $Q \leftarrow$ vertices in random order, all marked live
  for each step $t$ do
      // Get new element from the buffer
      $v_t \leftarrow Q$.ApproxGetMin()
      if $v_t$ marked dead then
          continue
      else if $v_t$ has a live unprocessed predecessor then
          $Q.insert(v_t, \pi(v_t))$ // Failed; reinsert
          continue
      else
          Add $v_t$ to MIS
          Mark all of $v_t$'s neighbors dead
      if $Q$.empty() then
          break
```

5.2 Analysis

In this section, we will bound the relaxation cost for the general framework (Algorithm 6) and for Maximal Independent Set (Algorithm 8). Algorithm 6 is easier to analyze and will serve as a warmup. Note that in both cases, n iterations are required to process all nodes and are necessary even with no relaxation. Thus, we can think of the "cost" of relaxation as the number of further iterations beyond the first n, which can be equivalently counted as the number of *re-insertions* performed by the algorithm. We will sometimes refer to executing such a re-insertion as a "failed delete" by Q.

Our primary goal will be to bound the number of iterations of the for loops in Algorithm 6 and 8 when running them sequentially with a k-relaxed priority queue. Although the initial analysis is sequential, the algorithms are parallel: threads can each run their own for loops concurrently and correctness is maintained. The difficulty in extending the analysis to the asynchronous setting is that it is not clear how to model failed deletes of dependents of a node that is being processed. The likelihood of such deletes depend on particulars of both the problem (i.e. how long processing and dependency checking steps actually take) and the thread scheduler and so are hard to model in our generic framework. Instead, we show empirically that our bounds hold in practice on a realistic asynchronous machine where threads run the loops fully in parallel.

The theorems we will prove are the following. Given a dependency graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, we first bound the number of iterations of Algorithm 6:

Theorem 5.1. *Algorithm 6 runs for* $n + O\left(\frac{m}{n}\right) \operatorname{poly}(k)$ *iterations in expectation.*

By contrast to Algorithm 6, we show that using a relaxed queue for computing Maximal Independent Sets on large graphs has essentially no cost at all, even for dense graphs! In particular, Algorithm 8 incurs a relaxation cost with no dependence at all on the size or structure of G, only on the relaxation factor k:

Theorem 5.2. *Algorithm 8 runs for* $n + \operatorname{poly}(k)$ *iterations in expectation.*

Before delving into the individual analyses, we first consider some key characteristics of a particular relaxed queue which will be at play, and quantify them in terms of the *fairness* and *rank error* of Q. We will assume that Q is k-relaxed as defined in Section 2.3: i.e., Q provides exponential tail bounds on the rank error and on the number of inversions experienced by an element, in terms of the parameter k. In the context of this analysis, rather than trying to ground one's intuition in actual relaxed queue constructions such as the SprayList or MultiQueues, it may help to instead think of a queue which returns a uniformly random element of the top-k at each step as the "canonical" k-relaxed Q. See Figure 5-1 for an illustration. With this in mind, we first prove two technical lemmas parameterized by k.

First, we characterize the probability that, for some edge $e = (u, v)$ in the dependency graph where u is a predecessor of v, u experiences an inversion on or above v before being processed.

Lemma 5.1. *Consider running Algorithm 6 (or Algorithm 8) using a k-relaxed queue Q on input graph $G = (V, E)$ and random permutation π. For a fixed edge $e = (u, v)$, the probability that u experiences an inversion on or above v during the execution is bounded by $O(k^2 \log k/n)$.*

Proof. We begin by proving a few immediate claims.

Claim 5.1. *At any time t, the probability of removing the element of top rank from Q is at least $1/k$.*

Proof. By the rank bound, we have that $\Pr[\text{rank}(t) \geq 2] \leq (1/e)^{2/k} < 1 - 1/k$. It therefore follows that $\Pr[rank(t) = 1] \geq 1/k$. $\square$

Let t_u be the first time when u experiences an inversion, and let R_u be its rank at that time. Since an element of rank $\geq u$ must be chosen at t_u, we have that, for any $\ell \geq 1$,

$$\Pr[R_u \geq \ell] \leq \exp(-\ell/k).$$

In particular, $\Pr[R_u \geq ck \log k] \leq (1/k)^c$, for any constant $c \geq 1$. That is, u has rank $\leq ck \log k$ at the time where it experiences its first inversion, w.h.p. in k. We now wish to bound the number of removals between the point when u experiences its first inversion, and the point when u is removed. Let this random variable be Δ_k. By

Claim 5.1, the top element is always removed within $O(k)$ trials in expectation, and hence we can show that

$$\Delta_k \geq (c+2)k^2 \log k, \text{ with probability at most } 1/k^c,$$

for $c \geq 1$, by bounding the time until all elements with rank $\geq u$ get removed, and connecting with the negative binomial distribution.

We now wish to know the probability that one of these steps is an inversion experienced by u on or above v. Fix a step t, and pessimistically assume that u is at the top of the queue at this time. Node v has lesser priority than u, chosen uniformly at random. Let j be the position of v, noting that $\Pr[j = \pi(u) + \ell] = 1/n$, for any integer $\ell \geq 1$.

Fixing j, we have that the probability that u experiences an inversion on or above v is $\leq (1/e)^{j/k}$. Fixing Δ_k, and bounding over all choices of j, we have that the probability that v is chosen is $\leq \sum_{j=1}^{n} \frac{1}{n} \left(\frac{1}{e}\right)^{j/k} = O(1/n)$.

Finally, bounding over all possible values of Δ_k and their probabilities, we get that the probability that u experiences an inversion on v during the execution is at most $O(k^2 \log k/n)$. □

Observe that the calculations done above are robust to conditioning on $\ell(u) = t, \ell(v) > t$ everywhere except that the second case in the first line of computation becomes unnecessary and the value of $\Pr[\ell(v) < \ell(u) = r]$ is $O(r/(n-t))$ (rather than $O(r/n)$). This gives Corollary 5.1.

Corollary 5.1. *Consider running Algorithm 6 (or Algorithm 8) using a k-relaxed queue Q on input graph $G = (V, E)$ and random permutation π. For a fixed edge $e = (u, v)$, the probability that u experiences an inversion on or above v during the execution conditioned on $\ell(u) = t, \ell(v) > t$ is bounded by $O(k^2 \log k/(n-t))$.*

Secondly, we will quantify the expected number of *priority inversions* incurred by an element, u, of Q once u's dependencies have been processed—that is, the number of elements of Q with lower priority than u which are returned by GetApproxMin() before u is.

Lemma 5.2. *Consider running Algorithm 6 (or Algorithm 8) using a k-relaxed queue Q on input graph $G = (V, E)$. For a fixed node u, if u is a root at some time t, at most $O(k)$ other elements of Q with lower priority than u are deleted after t in expectation.*

Lemma 5.2 follows immediately from the fairness bound provided by Q.

We stress that these two lemmas quantify the entire contribution of (the randomness of) the relaxation of Q to the analysis. The major burden of the analysis, particularly for MIS, is instead to manage the interaction between the randomness of the *permutation* π (which is not inherently related to the relaxation of Q) and the structure of G. Equipped with these lemmas, we are ready to do just that.

5.2.1 Algorithm 6: The General Case

The following theorem shows that the relaxed queue in Algorithm 6 has essentially no cost for sparse dependency graphs with $m = O(n)$ and still completes in $O(nk)$ iterations even for dense dependency graphs when $m = O(n^2)$. For example, Theorem 5.1 demonstrates that task-queue based problems which are inherently sparse such as Knuth Shuffle and List Contraction [12] incur only negligible "wasted work" when utilizing a k-relaxed queue with $k \ll n$. Furthermore, graph problems with edge dependencies such as greedy vertex coloring incur a cost proportional to the sparsity of the underlying graph. Although the result is not technically challenging, it is tight up to factors of k.

Theorem 5.1. *For a dependency graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, Algorithm 6 runs for $n + O\left(\frac{m}{n}\right) \operatorname{poly}(k)$ iterations in expectation.*

Proof. If a vertex u has no predecessor in Q at some time t, we call u a *root*. We will compute the expected number of failed deletes directly as follows: Whenever a failed delete occurs on a node w, charge it to the lexicographically first edge, $e = (u, v)$, for which u and v are both unprocessed and $\ell(v) \leq \ell(w)$ (i.e., with possibly $v = w$). Note that (1) such an edge must exist or else a failed delete could not have occurred, (2) the failed delete must represent a priority inversion on u, and (3) u must be a root (because e is lexicographically first). The first time an edge e is charged, we call e

the *active* edge until u is processed. Since u is a root for the duration of e's status as active edge, by Lemma 5.2, u only experiences $O(k)$ priority inversions in expectation while e is active, which upper bounds the number of failed deletes charged to e.

Let A_e be the event that edge $e = (u, v)$ ever becomes active. A_e can only occur if u experiences an inversion on or above v during the execution, which is bounded by $O(k^2 \log k/n)$ by Lemma 5.1. Thus, the total expected cost of e is at most $\mathbb{E}[c(e)] = \Pr[A_e]\,\mathbb{E}[c(e) \big| A_e] = O(k^3 \log k)/n = \text{poly}(k)/n$. There are m edges so the total cost across all edges is $\Theta\left(\frac{m}{n}\right)\text{poly}(k)$ as claimed. □

Briefly, to see that Theorem 5.1 is tight (up to factors of k), consider executing a greedy graph coloring problem on a clique. In this case, at any step, only the highest priority node can ever be processed, and for each such node, u, it takes $O(k)$ delete attempts before u is processed. Thus in total, the algorithm runs for $O(nk)$ iterations.

5.2.2 Algorithm 8: Maximal Independent Set

Theorem 5.2. *Algorithm 8 runs for $n + \text{poly}(k)$ iterations in expectation.*

Proof. Denote the lexicographically first MIS of G with respect to π as MIS_π. We first identify the key edges in the execution of Algorithm 8. We will say an edge $e = (u, v)$ is a *hot edge* w.r.t. π if u is the smallest labeled neighbor of v in MIS_π. Note that if (u, v) is a hot edge, v is not in MIS_π and u has a smaller label than v. Let H_e be the event that e is a hot edge w.r.t. π. Importantly, H_e depends only on the randomness of π and not on the randomness of the relaxation of Q. We make two key observations about hot edges that will allow us to prove the theorem:

Claim 5.2. *There is exactly one hot edge incident to each vertex $v \in V \setminus MIS_\pi$, and therefore the total number of hot edges is strictly less than n.*

This is clear from the condition that u is the smallest labeled neighbor of v in MIS_π and the fact that if v is not in the MIS_π, v must have at least one neighbor in MIS_π, or else MIS_π isn't maximal.

Claim 5.3. *A node w is only re-inserted by Algorithm 8 if there is at least one hot edge $e = (u, v)$ with u a root and $\ell(w) \geq \ell(v)$ (with possibly $v = w$). If e is such an edge, we say e is* active. *Furthermore, at least one active hot edge satisfies $\ell(u) < \ell(w)$.*

If w is re-inserted, then w must be live and adjacent to some smaller labeled live vertex u. Either u is a root, in which case (u, w) is the claimed hot edge, or else u must be adjacent to an even smaller labeled live vertex. In the latter case, we can recurse the argument down to u and eventually find a hot edge. In either case, both nodes incident to the discovered active hot edge will have a label no greater than w's.

Proof Outline. The strategy from here is a follows: whenever a failed delete occurs on a node w, we will charge it to an arbitrary hot edge $e = (u, v)$ with u a root and $\ell(w) \leq \ell(v)$ (of which there must be at least one by Claim 5.3). Similar to Theorem 5.1, we will say that e is active during the interval between the first time u experiences an inversion on or above v and the time u is processed. We say that the *cost*, $c(e)$, of an edge, e, is the number of failed deletes charged to it (which is notably 0 unless e is both hot and, at some point, active). We then separately bound (1) the expected number of active hot edges which ever exist over the execution of Algorithm 8 and (2) the expected number of failed deletes charged to an edge, given that it is an active hot edge. Combining these will give the result.

In order to quantify the distribution of hot edges, we will need one more definition. Fix $e = (u, v)$ and let G_e be the subgraph of G induced by $V' = V \setminus \{u, v\}$ and let π_e be π restricted to V'. Let $L_{e,t}$ be the event that neither u nor v has a neighbor $w \in MIS_{\pi_e}$ with $\ell_{\pi_e}(w) < t$. Informally, $L_{e,t}$ is the event that both u and v are still *live* in G after running Algorithm 8 with an exact queue ($k = 1$) for $t - 1$ iterations but with u, v excluded from Q. Like H_e, $L_{e,t}$ depends only on π and not on the randomness of the relaxation of Q; furthermore, $L_{e,t}$ is independent from $\ell(u)$ and $\ell(v)$. Using this definition, we can compute:

$$\Pr[H_e] = \sum_t \Pr[L_{e,t}] \Pr[\ell(u) = t] \Pr\left[\ell(v) > \ell(u) \big| \ell(u) = t\right]$$

$$= \sum_t \Pr\left[L_{e,t}\right] \frac{1}{n} \frac{n-t}{n-1}.$$

$$\begin{aligned}
\Pr\left[\ell(u) = t \middle| H_e\right] &= \frac{\Pr\left[H_e \middle| \ell(u) = t\right] \Pr\left[\ell(u) = t\right]}{\Pr\left[H_e\right]} \\
&= \frac{\Pr\left[L_{e,t}\right] \Pr\left[\ell(v) > t \middle| \ell(u) = t\right] \Pr\left[\ell(u) = t\right]}{\Pr\left[H_e\right]} \\
&= \frac{\Pr\left[L_{e,t}\right] \frac{n-t}{n-1}\frac{1}{n}}{\sum_{t'} \Pr\left[L_{e,t'}\right] \frac{1}{n}\frac{n-t'}{n-1}} \\
&= \frac{\Pr\left[L_{e,t}\right](n-t)}{\sum_{t'} \Pr\left[L_{e,t'}\right](n-t')}.
\end{aligned}$$

Next, we use the above formulations to bound the probability that a hot edge e is ever *active*. Suppose we are given that e is a hot edge and $\ell(u) = t$. Then e becomes active if and only if u suffers an inversion on or above v before u is processed by the algorithm. Let A_e be the event that e becomes active. At this point, we might wish to apply Lemma 5.1 directly, but unfortunately it is not clear that $\Pr[A_e]$ is independent from H_e, which we will need. However, note that H_e entails $\ell(v) > \ell(u)$ but given only that, $\ell(v)$ is otherwise independent from H_e. Thus, if we condition on $\ell(u) = t$ and $\ell(v) > \ell(u)$, then $\ell(u)$ is fixed and $\ell(v)$ is (conditionally) independent from H_e, and therefore A_e also becomes (conditionally) independent from H_e. Now we can apply Corollary 5.1, giving

$$\Pr\left[A_e \middle| \ell(u) = t, H_e\right] = \Pr\left[A_e \middle| \ell(u) = t, \ell(v) > \ell(u)\right] = O\left(\frac{k^2 \log k}{n-t}\right).$$

Then:

$$\begin{aligned}
\Pr\left[A_e \middle| H_e\right] &= \sum_t \Pr\left[\ell(u) = t \middle| H_e\right] \Pr\left[A_e \middle| \ell(u) = t, H_e\right] \\
&= \sum_t \frac{\Pr\left[L_{e,t}\right](n-t)}{\sum_{t'} \Pr\left[L_{e,t'}\right](n-t')} \frac{O(k^2 \log k)}{n-t} = O(k^2 \log k) \frac{\sum_t \Pr\left[L_{e,t}\right]}{\sum_{t'} \Pr\left[L_{e,t'}\right](n-t')}.
\end{aligned}$$

Observe that for fixed $e = (u, v)$, $\Pr[L_{e,t}]$ is decreasing in t. In particular, for any permutation π in which the event $L_{e,t}$ occurs, $L_{e,t-1}$ occurs also, but the reverse is not true. Let $\mu = \frac{1}{n}\sum_t \Pr\left[L_{e,t}\right]$. Using *Chebyshev's sum inequality*, we obtain:

$$\Pr\left[A_e | H_e\right] \leq O(k^2)\frac{n\mu}{\sum_{t'} \mu(n-t')} = O(k^2 \log k)\frac{n}{\sum_{t'}(n-t')} = O\left(\frac{k^2 \log k}{n}\right).$$

Finally, since u is a root and we only charge e for failed deletes on nodes with a larger label than v, and therefore a larger label than u as well, the number of times we charge e is upper bounded by the total number of priority inversions suffered by u while a root, which, by Lemma 5.2, is given by $O(k)$ in expectation. Thus $\mathbb{E}[c(e)|A_e, H_e] = O(k)$.

Combining all the parts, we have a final bound on the total cost:

$$\begin{aligned}
\mathbb{E}\left[\sum_e c(e)\right] &= \sum_e \Pr\left[H_e\right]\Pr\left[A_e|H_e\right]\Pr\left[c(e)|A_e, H_e\right] \\
&= \sum_e \Pr\left[H_e\right] \cdot O\left(\frac{k^2 \log k}{n}\right) \cdot O(k)) \\
&= O\left(\frac{k^3 \log k}{n}\right)\mathbb{E}\left[\#\{H_e\}\right] \\
&\overset{\textit{Claim } 5.2}{<} O\left(\frac{k^3 \log k}{n}\right) \cdot n \\
&= O(k^3 \log k) \\
&= \text{poly}(k)
\end{aligned}$$

□

5.3 Experimental Results

Synthetic Tests. To validate our analysis, we implemented the sequential relaxed framework described in Algorithm 6, and used it to solve instances of MIS, matching, Knuth Shuffle, and List Contraction using a relaxed scheduler which uses the MultiQueue algorithm [64], for various relaxation factors. We record the average number of extra relaxations, that is, the number of failed removals during the entire execution, across five runs. Results are presented in Table 5.1. We considered graphs of various densities with 10^3 and 10^4 vertices. The results appear to confirm our analysis: the number of extra iterations required for MIS is low, and scales only in K and not $|V| + |E|$. There is some variation for fixed K and varying $|V| + |E|$, but it is always within a factor of 2 for our trials and does not appear to be obviously correlated with $|V| + |E|$.

		k				
$\|V\|$	$\|E\|$	4	8	16	32	64
1000	10000	14.5	57.0	122.5	272.0	557.0
	30000	3.0	31.0	94.5	264.0	488.0
	100000	6.0	36.5	115.5	265.5	517.5
10000	10000	12.0	44.5	180.5	415.5	862.0
	30000	14.5	59.0	171.5	387.0	825.5
	100000	13.5	56.0	140.0	295.0	517.5

Table 5.1: Simulation results for varying parameters of Maximal Independent Set. k is the relaxation factor, n is the number of nodes and m is the number of edges. The number of extra iterations is averaged over 2 runs.

Concurrent Experiments. In addition to our sequential simulation, we implemented a concurrent instance of our scheduling framework, using the MultiQueue [64] relaxed priority queue data structure. We assume a setting where the input, that is, the set of tasks, is loaded initially into the scheduler, and is removed by concurrent threads. We use lock-free lists to maintain the individual priority queues and we hold pointers to the adjacency lists of each node within the queue elements, in order to be able to efficiently verify whether a task still has outstanding dependencies.

We compared to the exact scheduling framework using the "Wait Free Queue as Fast as Fetch-and-Add" [75]. Since there could still be some reordering of tasks due to concurrency, we elect to use a backoff scheme wherein if an unprocessed predecessor is encountered, we use the x86 built-in pause instruction to back off and wait for the predecessor to process. In practice this rarely occurs.

Setup. Our experiments were run on an Intel *Haswell* machine with 4 sockets, 18 cores per socket and 2 hyperthreads per core, for a total of 36 threads per socket, and 144 threads total. The machine has 512GB of RAM. We pinned threads to avoid unnecessary context switches and to fill up sockets one at a time. Hyperthreading is used for threads beyond the first 18 per socket. The machine runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 6.3.0 with optimization level `-O3`. Experiments are performed on graphs with 10^6 and 10^8 nodes and 10^9 edges. Our experiments were bottlenecked by graph generation and loading time so as a matter of practicality we were limited to 10^9 edges. This is sufficient for long runs on a sparse graphs, but unfortunately our dense graph trials were short ($< .1$ seconds for max thread counts). The number of queues is $4\times$ the number of threads. We tested thread counts which are a multiple of 6 up to 36 (saturating one socket), and then multiples of 36 beyond that. We ran four trials for each data point and took the average run time. In all cases, the standard deviation was less than 5% of the mean.

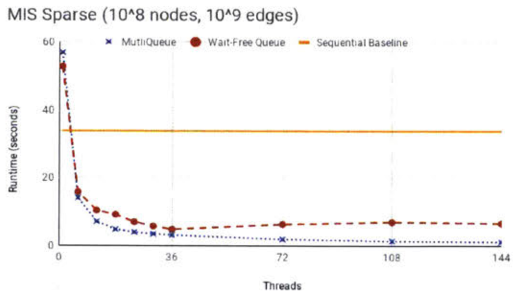

Figure 5-2: MIS algorithm run times on a sparse graph.

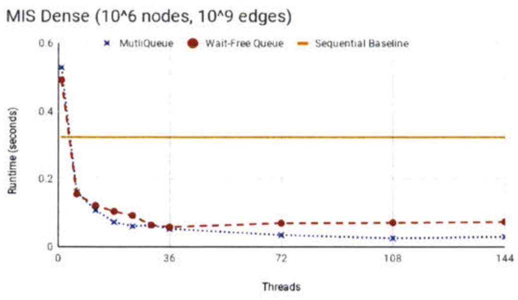

Figure 5-3: MIS algorithm run times on a dense graph.

Discussion. Figures 5-2 and 5-3 show that our framework using a relaxed scheduler scales with respect to the time to compute MIS over the target graph all the way up to max thread count. The exact framework using the fast wait-free queue also scales on one socket, but plateaus as soon as we add additional sockets (beyond 36 threads on our machine). At max thread count of 144 threads/72 cores, the relaxed scheduler is $\sim 29\times$ faster than optimized sequential code, $\sim 5.7\times$ faster than the exact scheduler using 144 threads, and $\sim 4.2\times$ faster than the peak performance of the exact scheduler, occurring at 36 threads (i.e. saturating one socket). An investigation of the overheads shows that the number of failed removals is extremely small (less than 0.01%

of all removal attempts); this suggest that most of the overheads versus sequential are in the scheduler structure, and in the slower per-thread queue performance, since we optimize for concurrency. The sequential algorithm is significantly faster at 1 thread, likely because it is highly optimized for the sequential setting, whereas the concurrent algorithms have several consistency checks that are superfluous in sequential runs.

5.4 Future Work

From a theoretical perspective, the natural next step would be to tighten the poly(k) bound on failed deletes, both for the generic algorithm and for MIS; in fact, we conjecture that the poly(k) bounds in both Theorems 5.1 and 5.2 can be replaced with $\Theta(k)$. However, proving such a bound seems to require a deep understanding of the interplay between the structure of G and the effects of the randomness of a k-relaxed queue, which we had to take care in our analysis to keep *separate.* Also of interest is to discover more applications, and perhaps more instances like MIS in which the bound in Theorem 5.1 can be improved on.

Chapter 6

Conclusion

We have argued for randomized relaxed concurrent data structures, showing that they are able to bypass inherent sequential bottlenecks in exchange for rank error with provable bounds. We have demonstrated their effectiveness in the setting of concurrent ordering structures, providing a definition which *quantifies* relaxation in a way which is both achievable by concrete algorithm and useful for application design.

Implementations: The SprayList provides *flexible* relaxation in the form of a centralized, versatile data structure with provable guarantees. In addition to its ordering properties, the SprayList can easily be augmented to support any standard Skiplist operations as needed by the application (Search(), GetMax(), etc.). We have demonstrated that the SprayList significantly outperforms exact concurrent queues both in raw throughput, and on realistic benchmarks such as Shortest Paths and Maximal Independent Set. On the flip side, there are still significant communication costs incurred when GetMin() is highly contended and batch deletes are frequent.

On the other hand, the MultiQueue algorithm offers an even more light weight, low overhead, distributed data structure while still providing guarantees similar to the SprayList. Although MultiQueues are not as versatile as the SprayList (for example, it is very difficult to search for a specified key) and can't adapt their semantics to contention, these shortcomings are made up for by increased throughput. Threads executing the MultiQueue algorithm incur no communication costs once the initial TRYLOCK() is successful, and can thereafter use blazingly fast, state-of-the-art sequential algorithms for the underlying queue operations.

The key open question in this sphere is whether the benefits of the SPRAYLIST and MULTIQUEUEs can be achieved simultaneously: is there a relaxed, concurrent ordering structure which is *adaptive*: providing strong semantics when contention is low, *versatile*: able to support all operations that applications could reasonably require, and *low overhead*: minimizing memory accesses and communication costs. Such a structure represents the holy grail of the field.

On a local level, there are still a few gaps in the analysis for the SPRAYLIST and MULTIQUEUEs. First, it would be valuable to verify that the given bounds are robust to fully arbitrary schedules with arbitrary interleavings of inserts and deletes. For MULTIQUEUEs in particular, it would be good to reduce the large constant number of queues currently required for the analysis to work. Another important step would be to improve the bounds on rank error or provide matching lower bounds showing that no randomized relaxed concurrent data structure can do better than $\Omega(n \log n)$ rank error without incurring significant slowdown on contended workloads (i.e., with runtime $o(n^\epsilon)$ per operation in an n-thread execution). In addition to these sorts of improvements, one might also hope to find a simpler and perhaps more elegant argument for why the bounds that we've shown hold.

Applications: Given our proposed definition of k-relaxation, we have shown that k-relaxed queues can correctly execute any problem that can be formulated as a *task queue*, and that these executions are guaranteed to be *efficient* as long the underlying dependency graph is not dense. This property makes relaxed queues *ideal* for computing problems with inherently sparse dependency structures, such as List Contraction, since the overhead incurred due to relaxation is negligible. Moreover, we gave the surprising result that Maximal Independent Set and Maximal Matching can also be computed with negligible overhead irrespective of the size or structure of the input graph. These results collectively demonstrate that the increased throughput of relaxed ordering structures more than makes up for their weaker semantics in realistic settings.

The obvious question to ask is whether there are further problems which are well-suited for using relaxed structures, either inherently or by exploiting some substructure to eliminate dependencies, as we did with Maximal Independent Set. For example, while we were able to achieve good performance executing Dijkstra's Algorithm on realistic graphs, we currently lack any theoretical grounding to say when Dijkstra's algorithm can be executed efficiently using relaxed structures, and indeed we do not expect relaxed queues to perform well if the input graph is selected adversarially, e.g. in the case of graph which is just a single long path. Finding conditions under which the amount of wasted work incurred due to relaxation while running Dijkstra's algorithm is low would certainly be of interest.

With respect to our analysis of task-queue problems, there is still the open question of tightening the polynomial factors of k that we use to bound the number of dependency violations. In particular, our current analysis only manages to bound the number of failed deletes by k^3, but we conjecture that a linear bound, that is, $O(k)$, is possible.

On the experimental side, we believe that relaxed data structures are reaching a point of sufficient maturity to be used more prominently in commercial production systems. Accordingly, a thorough comparison of relaxed and exact data structures in the context of a large scale application would be of great value.

Bibliography

[1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[2] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18:1–18:51, June 2014.

[3] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. Distributionally linearlizable data structures. *Under Submission to SPAA 2018*, 2018.

[4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. *arXiv preprint arXiv:1706.04178*, 2017. To appear in PODC 2017.

[5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. Spraylist. `https://github.com/jkopinsky/SprayList`.

[6] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, USA, 2015. ACM.

[7] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[8] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[9] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. Café: 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.

[10] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 745–754, New York, NY, USA, 2000. ACM.

[11] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, December 2008.

[12] Guy E Blelloch. Some sequential algorithms are almost always parallel. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 24–26, 2017.

[13] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *ACM SIGPLAN Notices*, volume 47, pages 181–192. ACM, 2012.

[14] Guy E Blelloch, Jeremy T Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 308–317. ACM, 2012.

[15] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 467–478. ACM, 2016.

[16] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[17] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[18] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. Cbpq: High performance lock-free priority queue. In *European Conference on Parallel Processing*, pages 460–474. Springer, 2016.

[19] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1):4–21, 1998.

[20] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In *International Symposium on Distributed Computing*, pages 406–420. Springer, 2014.

[21] Neil Calkin and Alan Frieze. Probabilistic analysis of a parallel algorithm for finding maximal independent sets. *Random Structures & Algorithms*, 1(1):39–50, 1990.

[22] Don Coppersmith, Prabhakar Raghavan, and Martin Tompa. Parallel graph algorithms that are efficient on average. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 260–269. IEEE, 1987.

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

[24] Dave Dice, Virendra J Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 65–74. ACM, 2011.

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

[26] Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *International Symposium on Distributed Computing*, pages 149–160. Springer, 1998.

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

[28] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.

[29] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory Comput. Syst.*, 55(3):475–520, 2014.

[30] Manuela Fischer and Andreas Noever. Tight analysis of parallel randomized greedy mis. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2152–2160. SIAM, 2018.

[31] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd annual ACM symposium on Principles of Distributed Computing (PODC' 04)*, pages 50–59, New York, NY, USA, 2004. ACM Press.

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

[33] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[34] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench. In *Proceedings of the 20th Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[35] Jakob Gruber, Jesper Larsson Träff, and Martin Wimmer. Brief announcement: Benchmarking concurrent priority queues. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 361–362, New York, NY, USA, 2016. ACM.

[36] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 17:1–17:9, 2013.

[37] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[38] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *International Symposium on Distributed Computing*, pages 79–93. Springer, 2010.

[39] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana 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.

[40] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=1760631.1760646`.

[41] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[42] 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.

[43] Shams Imam and Vivek Sarkar. Load balancing prioritized tasks via work-stealing. In *European Conference on Parallel Processing*, pages 222–234. Springer, 2015.

[44] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro*, 36(3):105–117, 2016.

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

[46] Christoph M Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. Performance, scalability, and semantics of concurrent fifo queues. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 273–287. Springer, 2012.

[47] Doug Lea, 2007. `http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`.

[48] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. The MIT press, 2001.

[49] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. In *European Conference on Parallel Processing*, pages 209–221. Springer, 2015.

[50] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

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

[52] Yujie Liu and Michael Spear. Mounds: Array-based concurrent priority queues. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 1–10. IEEE, 2012.

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

[54] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[55] Michael David Mitzenmacher. *The Power of Two Random Choices in Randomized Load Balancing*. PhD thesis, PhD thesis, Graduate Division of the University of California at Berkley, 1996.

[56] Adam Morrison. Scaling synchronization in multicore programs. *Commun. ACM*, 59(11):44–51, October 2016.

[57] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. *SIGPLAN Not.*, 48(8):103–112, February 2013.

[58] Donald Nguyen, Andrew Lenharth, and Keshav 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.

[59] Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.

[60] Yuval Peres, Kunal Talwar, and Udi Wieder. Graphical balanced allocations and the 1 + beta-choice process. *Random Struct. Algorithms*, 47(4):760–775, December 2015.

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

[62] William Pugh. Concurrent maintenance of skip lists. 1998.

[63] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[64] Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.

[65] Konstantinos Sagonas and Kjell Winblad. The contention avoiding concurrent priority queue. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 314–330. Springer, 2016.

[66] Konstantinos Sagonas and Kjell Winblad. A contention adapting approach to concurrent ordered sets. *Journal of Parallel and Distributed Computing*, 2017.

[67] 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.

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

[69] Julian Shun, Yan Gu, Guy E Blelloch, Jeremy T Fineman, and Phillip B Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 431–448. SIAM, 2014.

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

[71] Kunal Talwar and Udi Wieder. Balanced allocations: The weighted case. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 256–265, New York, NY, USA, 2007. ACM.

[72] Orr Tamir, Adam Morrison, and Noam Rinetzky. A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[73] 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.

[74] M. Wimmer, Gruber J., J. L. Träff, and P. Tsigas. The lock-free k-lsm relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*, 2015.

[75] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *SIGPLAN Not.*, 51(8):16:1–16:13, February 2016.