Relaxed Schedulers Can Efficiently Parallelize Iterative Algorithms

Dan Alistarh
IST Austria
dan.alistarh@ist.ac.at

Trevor Brown
IST Austria
me@tbrown.pro

Justin Kopinsky*
Massachusetts Institute of Technology
jkopin@mit.edu

Giorgi Nadiradze†
ETH Zurich
giorgi.nadiradze@inf.ethz.ch

ABSTRACT

There has been significant progress in understanding the parallelism inherent to iterative sequential algorithms: for many classic algorithms, the depth of the dependence structure is now well understood, and scheduling techniques have been developed to exploit this shallow dependence structure for efficient parallel implementations. A related, applied research strand has studied methods by which certain iterative task-based algorithms can be efficiently parallelized via relaxed concurrent priority schedulers. These allow for high concurrency when inserting and removing tasks, at the cost of executing superfluous work due to the relaxed semantics of the scheduler.

In this work, we take a step towards unifying these two research directions, by showing that there exists a family of relaxed priority schedulers that can efficiently and deterministically execute classic iterative algorithms such as greedy maximal independent set (MIS) and matching. Our primary result shows that, given a randomized scheduler with an expected relaxation factor of k in terms of the maximum allowed priority inversions on a task, and any graph on n vertices, the scheduler is able to execute greedy MIS with only an additive factor of poly(k) expected additional iterations compared to an exact (but not scalable) scheduler. This counter-intuitive result demonstrates that the overhead of relaxation when computing MIS is *not* dependent on the input size or structure of the input graph. Experimental results show that this overhead can be clearly offset by the gain in performance due to the highly scalable scheduler. In sum, we present an efficient method to deterministically parallelize iterative sequential algorithms, with provable runtime guarantees in terms of the number of executed tasks to completion.

*Justin's work was funded by the National Science Foundation, grant CCF-1563880 along with a generous grant from the Intel Corporation.
†Giorgi's work was funded by the Swiss National Fund Ambizione Project PZ00P2 161375.

PODC'18, July 23–27, 2018, Egham, United Kingdom

ACM ISBN 978-1-4503-5795-1/18/07…$15.00
https://doi.org/10.1145/3212734.3212756

ACM Reference Format:
Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. 2018. Relaxed Schedulers Can Efficiently Parallelize Iterative Algorithms. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3212734.3212756

1 INTRODUCTION

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*. In this paper, we will focus on two specific variants of this strategy.

The *deterministic* approach, e.g. [5–8, 18, 25] has been to study the directed-acyclic graph (DAG) step dependence 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 dependence structure for efficient execution on parallel architectures. As the name suggests, this approach ensures deterministic outputs (i.e. outputs uniquely determined by the input), and can yield good practical performance [7], but requires a non-trivial amount of knowledge about the problem at hand, and the use of carefully-constructed parallel schedulers [7].

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. 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 [7] 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 [13] 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 [7], list contraction [25], Knuth shuffle [25], linear programming [5], and graph connectivity [5].

An alternative approach has been to employ *relaxed data structures* to schedule task-based programs. Starting with Karp and Zhang [19], 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. [1, 20, 22]: 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. 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 [14, 20]. At the same time, despite good empirical performance, this approach still lacks analytical bounds, and results are no longer deterministic.

In this paper, we ask: 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?

Contribution. In a nutshell, this work shows that a natural family of *fair relaxed schedulers*—providing upper bounds on the degree of relaxation, and on the number of inversions that a task can experience—can execute a range of iterative sequential algorithms *deterministically*, preserving the dependence structure, 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*, but also algorithms for *list contraction* and *generating permutations* via the Knuth shuffle. We call this class *iterative algorithms with explicit dependencies*. Our main technical result is that, for MIS and matching in particular, the overhead of relaxed scheduling is *independent of the graph size or structure*. This analytical result suggests that relaxed schedulers should be a viable alternative, a finding which is also supported by our preliminary 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 *scheduler*, which is *relaxed*, in the sense that it could return tasks *out of order*. This induces a sequential model,[1] where at each step, the scheduler returns a new task: for simplicity, assume for now that the scheduler returns at each step a task chosen *uniformly at random* among the top-k available tasks, in descending priority order. (We will model realistic relaxed schedulers [2, 21] precisely in the following section.)

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 delete 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 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 $\text{poly}(k)$ 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 nontrivial, 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.

In the broader context of the parallel scheduling literature, our results suggest that *task priorities* can be supported in a scalable manner, through relaxation, without loss of determinism or work efficiency. We believe this is the first time this observation is made. We validate our results empirically, via a preliminary implementation of the scheduling framework in C++, based on a lock-free extension of the MultiQueue relaxed schedulers [21]. Our broad finding is that this relaxed scheduling framework can ensure scalable execution, with minimal overheads due to contention and verifying task dependencies. For MIS on large graphs, we obtain a solution, with 6x speedup at 24 threads versus an optimized sequential baseline.

Related Work. Our work is inspired by the line of research by Blelloch et al. [5–7, 18, 25], as well as [11–13], 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, [7, 25] start from

[1] We consider this sequential model, similar to [7], 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.

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. [7], 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 [21], 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 [5, 7, 25] 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 [7] 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 [7] 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 [7] 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 [19], 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 [1, 2, 4, 15, 20, 21, 23, 24, 26] for recent examples. In particular, we note that state-of-the-art packages for graph processing [20] and machine learning [14] implement such relaxed schedulers.

Recent work by a subset of the authors [2] showed that a simple and popular priority scheduling mechanism called the MultiQueue [14, 15, 21] enforces strong probabilistic guarantees on the rank of elements returned, in an idealized model. Concurrent work [3] proves that these guarantees in fact hold in asynchronous concurrent executions, under some analytic assumptions. Based on this result, our work bounds should hold when using MultiQueues as relaxed schedulers, in concurrent executions.

Parallel scheduling [9, 10] 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 [2]. To our knowledge, there is only one previous attempt to add priorities to work-stealing schedulers [17], using a multi-level global queue of tasks, partitioned by priority. This technique is different, and provides no work guarantees.

2 EXECUTING ITERATIVE ALGORITHMS VIA PRIORITY SCHEDULERS

2.1 Modeling Relaxed Priority Schedulers

In the following, we will provide the sequential specification of a generic relaxed priority scheduler Q, which contains a set of $\langle task, priority \rangle$ pairs. A relaxed priority scheduler will provide the following methods:

- ApproxGetMin(), which returns a $\langle task, priority \rangle$ pair and deletes it from the structure, if a task is available, or $\perp$, otherwise. The relaxation guarantees of this operation are precisely defined below;
- Empty(), which returns whether the scheduler still has tasks or not;
- Insert($\langle task, priority \rangle$), which inserts a new task into Q.

Let $\mathsf{rank}(t)$ be the rank of the task which is returned by the t^{th} ApproxGetMin operation, among all tasks present in Q. We say that task $u \in Q$ experiences a *priority inversion* at an ApproxGetMin step if a task v of *lower* priority than u is retrieved at that step. For any task u, let $\mathsf{inv}(u)$ be the number of inversions which the task u experiences before being removed.

Definition 2.1. Fix a relaxed priority scheduler Q, with parameters $k \geq 1$, the *rank bound*, and ϕ, the *fairness bound*. We say that Q is an (k, ϕ)-relaxed priority scheduler if it ensures the following:

(1) **Rank Bound.** For any time t, and any integer $\ell > 1$, $\Pr[\mathsf{rank}(t) \geq \ell] \leq \exp(-\ell/k)$.
(2) **Fairness Bound.** For any task u, and any integer $\ell \geq 1$, $\Pr[\mathsf{inv}(u) \geq \ell] \leq \exp(-\ell/\phi)$.

Relation to Practical Schedulers. Upon inspection, both the SprayList [1] and the MultiQueue [21] relaxed priority schedulers ensure these exponential tail bounds on both rank and fairness, under some analytic assumptions. These conditions are trivially ensured by deterministic implementations such as [26]. In particular, the SprayList ensures these bounds with parameters k and ϕ in $O(p \operatorname{polylog} p)$, where p is the number of processors [1]. MultiQueues ensure these bounds with parameters $k = O(m)$, and $\phi = O(m \log m)$, where m is the number of distinct priority queues [2]. This holds even in concurrent executions [3].

In the following, it will be convenient to assume a single parameter k, which upper bounds both the rank and the relaxation parameters. We call the (k, k)-relaxed scheduler simply a k-relaxed scheduler.

2.2 A General Scheduling Framework

We now present our framework for executing task-based sequential programs, whose pseudocode is given in Algorithm 1. 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 1 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 [5]. We show sample instantiations of the framework in Section 2.3.

Algorithm 1: Generic Task-based Framework

```
Data: Dependency Graph G = (V, E)
Data: Vertex permutation π
// Q is an exact priority queue
Q ← vertices in V with priorities π(V)
for each step t do
    // Get new element from the buffer
    v_t ← Q.GetMin()
    Process(v)
    Remove v_t from Q
    if Q.empty() then
        break
```

Algorithm 2: Relaxed Scheduling Framework

```
Data: Dependency Graph G = (V, E)
Data: Vertex permutation π
Data: Parameter k
// Q is a k-relaxed scheduler
Q ← vertices in random order
for each step t do
    // Get new element from the buffer
    v_t ← Q.ApproxGetMin()
    if v_t has unprocessed predecessor then
        Q.insert(v_t, π(v_t)) // Failed; reinsert
        continue
    else Process(v)
    if Q.empty() then break
```

Algorithm 2 gives a method for adapting Algorithm 1 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 2 gives the same output as Algorithm 1, irrespective of the relaxation factor k. 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 2, proven formally in Section 3.1, argues that if π is chosen uniformly at random from among all vertex permutations, then Algorithm 2 completes in at most $n + O(\frac{m}{n} poly(k))$ iterations (compared to exactly n for Algorithm 1). 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).

2.3 Example Applications

Applying the sequential task-based framework of Algorithm 1 only requires an implementation of Process(v). Implementing the relaxed framework in Algorithm 2 further requires G (either explicitly or via a predecessor query method). We now give examples for Greedy Vertex Coloring and List Contraction, whose dependency graph is implicit.

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 3: Greedy Vertex Coloring Process(v)

```
Data: Input Graph G = (V, E)
Data: Permutation π
Data: Partial coloring c : V → ℕ
Function Process(v) :
    S ← ∅
    foreach (u, v) ∈ G, s.t. ℓ(u) < ℓ(v) do
        S ← S ∪ {c(u)}
    c(v) ← min_{i∈ℕ} i ∉ 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 swinging two pointers: 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. If we induce a priority order on list elements (e.g. uniformly at random), then there is an induced orientation of the edges of the dependency graph which forms a DAG. 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).

2.4 Greedy Maximal Independent Set

We give a variant of Algorithm 2 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 4 implements MIS in the framework of Algorithm 2 while also making use of this observation. Interestingly, Algorithm 4 can also be used to find a maximal 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 there is an edge between vertices of G' if the corresponding edges of G share an incident vertex. (One can view matching as an "independent set" of edges, no two of which are incident to the same vertex.)

Algorithm 4: Relaxed Queue MIS

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

As we will show in Section 3, the simple improvement Algorithm 4 makes over Algorithm 2 results in only a negligible number of extra iterations due to relaxation.

3 ANALYSIS

In this section, we will bound the relaxation cost for the general framework (Algorithm 2) and for Maximal Independent Set (Algorithm 4). Algorithm 2 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 2 and 4 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 2:

Theorem 3.1. *Algorithm 2 runs for $n + O\left(\frac{m}{n}\right)\text{poly}(k)$ iterations in expectation.*

By contrast to Algorithm 2, 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 4 incurs a relaxation cost with no dependence at all on the size or structure of G, only on the relaxation factor k:

Theorem 3.2. *Algorithm 4 runs for $n + \text{poly}(k)$ iterations in expectation.*

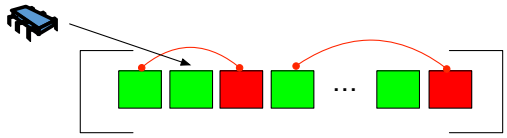

Figure 1: Simple illustration of the process. The blue thread queries the relaxed scheduler, which returns one of the top k tasks, on average (in brackets). Some of these tasks (green) can be processed immediately, as they have no dependencies. Tasks with dependencies (red) cannot be processed yet, and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

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. As discussed in Section 2.1, we will assume that Q is k-relaxed: that is, 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. Intuitively, it may help to 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 1 for an illustration. (As discussed in Section 2.1, real schedulers have slightly different properties, which are captured in our framework.) We state and 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, v experiences an inversion before u is processed. We say that vertex u experiences an inversion on or above node v at some point during the execution if $\ell(u) < \ell(v)$, but some node with label *at least* $\ell(v)$ is returned by Q before u is processed during the execution.

Lemma 3.3. *Consider running Algorithm 2 (or Algorithm 4) using a k-relaxed queue Q on input graph $G = (V, E)$. 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 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 $\Pr[rank(t) = 1] \geq 1/k$. □

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 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 $\Pr[j = \pi(u) + \ell] \leq 1/n$, 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 $O(k^2 \log k/n)$. □

Furthermore, the above proof directly implies the following corollary:

Corollary 3.4. *Consider running Algorithm 2 (or Algorithm 4) using a k-relaxed queue Q on input graph $G = (V, E)$. For a fixed edge $e = (u, v)$, the probability that u experiences an inversion on or above v during an execution on a random permutation π conditioned on $\ell(u) = t, \ell(v) > t$ is bounded by $O(k^2 \log k/(n - t))$.*

In Appendix A, we also prove a slightly tighter version of Lemma 3.3 for the case where the implementation of Q provides the further guarantee of *only* returning elements from the top-k. Note that such a queue is always k-rank bounded, but is not necessarily k-fair.

Our second technical lemma quantifies 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 times an element of Q with lower priority than u is returned by GetApproxMin() before u is. If a vertex u has no predecessor in Q at some time t, we call u a *root*.

Lemma 3.5. *Consider running Algorithm 2 (or Algorithm 4) 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.*

Proof. Follows immediately from the k-fairness 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.

3.1 Algorithm 2: The General Case

The following theorem shows that the relaxed queue in Algorithm 2 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 3.1 demonstrates that task-based problems which are inherently sparse such as Knuth Shuffle and List Contraction [5] 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 3.1. *For a dependency graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, Algorithm 2 runs for $n + O\left(\frac{m}{n}\right)\text{poly}(k)$ iterations.*

Proof. 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 3.5, 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 3.3. Thus, the total expected cost of e is at most $\mathbb{E}[c(e)] = \Pr[A_e]\,\mathbb{E}[c(e)|A_e] = O(k^3 \log k)/n = \text{poly}(k)/n$. There are m edges so the total cost is $\Theta\left(\frac{m}{n}\right)\text{poly}(k)$ as claimed. □

Briefly, to see that Theorem 3.1 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.

3.2 Algorithm 4: Maximal Independent Set

The following theorem bounds the number of iterations of Algorithm 4. By contrast to Algorithm 2, 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 4 incurs a relaxation cost with no dependence on the size or structure of G, only on the relaxation factor k.

Theorem 3.2. *Algorithm 4 runs for $n + \text{poly}(k)$ iterations.*

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 4. 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 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 3. *A node w is only re-inserted by Algorithm 4 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 $\ell(w) \geq \ell(v)$ (of which there must be at least one by Claim 3). Similar to Theorem 3.1, 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 4 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 4 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:

$$\begin{aligned}\Pr[H_e] &= \sum_t \Pr[L_{e,t}] \Pr[\ell(u) = t] \Pr\left[\ell(v) > \ell(u) \middle| \ell(u) = t\right] \\ &= \sum_t \Pr[L_{e,t}] \frac{1}{n} \frac{n-t}{n-1}.\end{aligned}$$

and

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

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 3.3 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 3.4, giving

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

Then:

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

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 $\mu = \frac{1}{n} \sum_t \Pr[L_{e,t}]$. Using *Chebyshev's sum inequality*, we obtain:

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

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 3.5, is given by $O(k)$ in expectation. Thus $\mathbb{E}[c(e) \middle| A_e, H_e] = O(k)$.

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

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

□

		k				
$\|V\|$	$\|E\|$	4	8	16	32	64
1000	10000	12.8	56.8	148.8	308.6	583.0
	30000	7.0	40.8	108.6	264.2	478.6
	100000	12.4	40.0	100.6	225.8	427.2
10000	10000	11.0	43.2	145.4	336.4	738.6
	30000	16.6	71.4	196.0	437.6	890.2
	100000	13.0	56.2	144.4	290.6	529.6

Table 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.

4 EXPERIMENTAL RESULTS

Synthetic Tests. To validate our analysis, we implemented the sequential relaxed framework described in Algorithm 2, and used it to solve instances of MIS, matching, Knuth Shuffle, and List Contraction using a relaxed scheduler which uses the MultiQueue algorithm [21], for various relaxation factors. We record the average number of extra relaxations, that is, the number of failed deletes during the entire execution, across five runs. Results are presented in Table 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 in $|V| + |E|$. 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 $|V| + |E|$.

Concurrent Experiments. We implemented a simple version of our scheduling framework, using a variant of the MultiQueue [21] 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 check 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 [27]. 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 wait for the predecessor to process. In practice this rarely occurs.

Setup. Our experiments were run on an Intel Xeon Gold 6150 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 memory, with 128GB *local* to each socket. Accesses to local memory are cheaper than accesses to remote memory. We pinned threads to avoid unnecessary context switches and to fill up sockets one at a time. The machine runs Ubuntu 14.04 LTS. We used the GNU C++ compiler (G++) 6.3.0 with optimization level `-O3`.

Experiments are performed on $G(n, p)$ random graphs in three classes; sparse graphs with 10^8 nodes and 10^9 edges, *small dense* graphs with 10^6 nodes and 10^9 edges, and *large dense* graphs with 10^7 nodes and 10^{10} edges. Our experiments were bottlenecked by graph generation and loading time so we were limited to these graph sizes.

For each data point, we run five trials. In each trial, a graph is generated in parallel by 144 threads, and then we measure the time for n threads to compute an MIS. Note that, even when $n = 1$, we generate the graph using 144 threads. To ensure that this does not change memory locality in a way that would invalidate our results, we used the `numactl` utility to cause memory to be allocated locally on *only* the sockets where n threads will compute the MIS. We verified that this yields the same behavior as experiments where the graph is generated with only n threads. (Without using *numactl*, we observed significant slowdowns in the sequential algorithm.)

The number of queues in the MultiQueue is 4× the number of threads. In our graphs, we plot the average run time on a logarithmic y-axis versus the number of concurrent threads. Error bars show *minimum* and *maximum* run times.

Discussion. Figure 2 shows 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, but not as well. In the sparse graphs, the relaxed scheduler is up to $\sim$ 18.2× faster than optimized sequential code, compared to the exact scheduler, which peaks at $\sim$ 5.0× faster. In the small dense graphs, where the time spent traversing edges in the MIS algorithm dominates the minor cost of dequeuing nodes, the exact scheduler achieves a peak speedup of $\sim$ 17.8× over the sequential algorithm, which is approaching the relaxed scheduler's peak speedup of $\sim$ 24.6×. However, in the large dense graphs, even though many edges are still traversed by the MIS algorithm, there are sufficiently many nodes to be dequeued that the performance advantage of the relaxed scheduler shows through: it achieves speedup of up to $\sim$ 16.3× compared to the exact scheduler, which manages only $\sim$ 6.9×. Note that the single threaded performance of the relaxed scheduler is also quite close to the sequential algorithm. In contrast, the exact scheduler is orders of magnitude slower with a single thread.

5 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 3.1 and 3.2 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 3.1 can be improved on.

One shortcoming of our approach is the fact that our cost measure is the number of *vertex* accesses in the priority queue. Notice that in theory our bounds may be substantially different when expressed in other metrics, such as the number of edge accesses for the algorithm to terminate, which are closer to standard *work* bounds. We plan to investigate such cost measures in future work.

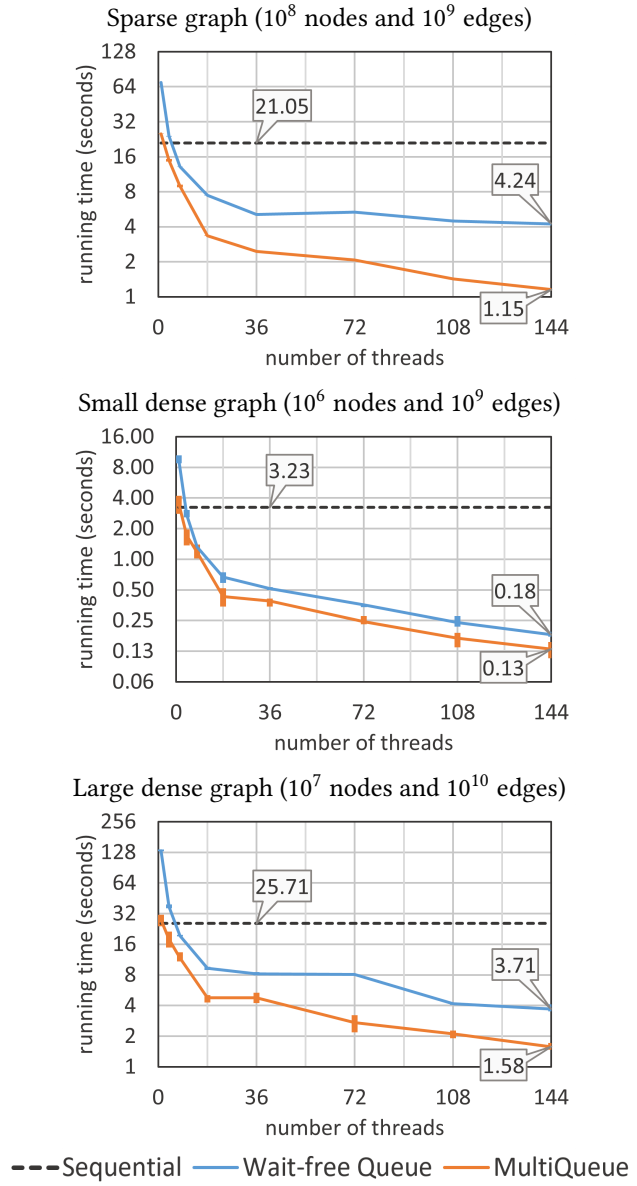

Figure 2: Results for concurrent MIS experiments.

From the practical perspective, the immediate step would to improve upon our preliminary results, and implement a high-performance variant of this scheduler, and use this framework in the context of more general graph processing packages.

REFERENCES

[1] 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.

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 283–292, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4992-5. doi: 10.1145/3087801.3087810. URL http://doi.acm.org/10.1145/3087801.3087810.

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

[4] 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. ISBN 978-3-642-24099-7. URL http://dl.acm.org/citation.cfm?id=2075029.2075087.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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. doi: 10.1145/2482767.2482789. URL http://doi.acm.org/10.1145/2482767.2482789.

[16] Nick Harvey. Lecture notes in randomized algorithms. http://www.cs.ubc.ca/~nickhar/W12/Lecture3Notes.pdf, 2012.

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

[18] 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.

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

[20] 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. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522739. URL http://doi.acm.org/10.1145/2517349.2522739.

[21] 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. ISBN 978-1-4503-3588-1. doi: 10.1145/2755573.2755616. URL http://doi.acm.org/10.1145/2755573.2755616.

[22] 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.

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

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

[25] 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.

[26] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *ACM SIGPLAN Notices*, volume 50, pages 277–278. ACM, 2015.

[27] 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. ISSN 0362-1340. doi: 10.1145/3016078.2851168. URL http://doi.acm.org/10.1145/3016078.2851168.

A A TIGHTER VERSION OF LEMMA 3.3.

We now prove a slightly tighter version for the case where only elements from the top-k may be chosen by the scheduler Q. Note that the condition in the following Lemma (that u and v are simultaneously in the top-k) is a pre-requisite for u to experience an inversion on or above v, and thus the Lemma is slightly stronger than necessary.

Lemma A.1. *Consider running Algorithm 2 (or Algorithm 4) using a k-relaxed queue Q on input graph $G = (V, E)$. For a fixed edge $e = (u, v)$, the probability that both u and v are simultaneously in the top-k of Q during any execution on a random permutation π is bounded by $O(k^2/n)$.*

Proof. We will write $e \in$ top-k as shorthand for the event that u and v are simultaneously in the top-k of Q at some time. Note that no matter what dependencies exist in the top-k of Q, the entire top-k is flushed after the rank 1 element gets deleted k times. The number of iterations it takes to delete the rank 1 element k times after u enters the top-k (thereby flushing u with certainty) is a negative binomially distributed random variable X_u with mean k^2 and success probability $1/k$ (due to the fairness of Q), and similarly for X_v. Since S_e entails that either $\ell(u) < \ell(v) < \ell(u) + X_u$ or $\ell(v) < \ell(u) < \ell(v) + X_v$, we note that the two cases are symmetric and compute:

$$\begin{aligned}
\Pr[e \in \text{top-k}] &\le \Pr[\ell(u) < \ell(v) < \ell(u) + X_u] \\
&\quad + \Pr[\ell(v) < \ell(u) < \ell(v) + X_v] \\
&= 2\Pr[\ell(u) < \ell(v) < \ell(u) + X_u] \\
&= 2\sum_r \Pr[X_u = r]\Pr[\ell(u) < \ell(v) < \ell(u) + r] \\
&= 2\sum_r \Pr[X_u = r]\frac{r}{n} \\
&\le \frac{2ck^2}{n}\Pr\left[X_u \le ck^2\right] + 2\sum_{r > ck^2}\Pr[X_u = r]\frac{r}{n} \\
&\le O\left(\frac{k^2}{n}\right) + 2\sum_{r'}\Pr\left[X_u > r'k^2\right]\frac{(r'+1)k^2}{n} \\
&\le O\left(\frac{k^2}{n}\right)\left(1 + \sum_{r'} e^{O(-r')}(r'+1)\right) \qquad (*) \\
&= O\left(\frac{k^2}{n}\right),
\end{aligned}$$

where (*) uses a standard tail bound on the Negative Binomial Distribution[2]. □

[2] See [16] for a derivation.