Thesis for The Degree of Licentiate of Engineering

Semantic Relaxation of Concurrent Data Structures: Efficient and Elastic Designs

Kåre von Geijer

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2025

Semantic Relaxation of Concurrent Data Structures: Efficient and Elastic Designs

Kåre von Geijer

ISSN 1652-876X

Department of Computer Science and Engineering
Division of Computer and Network Systems
Distributed Computing and Systems
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg,
Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2025.

"The only thing that makes life possible is permanent, intolerable uncertainty: not knowing what comes next."

- Ursula K. Le Guin, The Left Hand of Darkness

Semantic Relaxation of Concurrent Data Structures: Efficient and Elastic Designs

Kåre von Geijer

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

The growing availability of hardware parallelism continues to challenge developers to design programs that efficiently utilize it. Many widely used concurrent data structures — such as FIFO queues and priority queues — enforce strict ordering semantics that inevitably create memory contention, limiting scalability. Semantic relaxation has emerged as a powerful technique for increasing parallelism at the expense of a controlled weakening of the ordering semantics. However, many open questions remain in the field of relaxed data structures, both in the terms of efficient, flexible designs, and in their practical applicability.

This thesis advances the theory and practice of relaxed concurrent data structures. First, we revisit the classic *balanced allocations* paradigm in the context of queues, introducing the d-CBO relaxed FIFO queue. The d-CBO queue utilizes the classical d-choice in a new way to evenly balance operation counts across sub-queues. Our analysis shows provably low, stable relaxation errors, and experiments demonstrate a better relaxation-performance trade-off than previous relaxed FIFO designs.

Second, we explore the applicability of relaxation in graph analytics. Using a relaxed priority queue in a parallel Single-Source Shortest Path (SSSP) implementation, we achieve state-of-the-art performance on sparse graphs and remain competitive across other graph types. Our findings show that relaxed designs can be used within parallel algorithms to outperform state-of-the-art without extensive parameter tuning or problem-specific tailoring.

Finally, we introduce the concept of elastic relaxation, which enables relaxed implementations to adjust their semantics dynamically during run time. We extend a state-of-the-art framework for relaxed data structures to support elastic variants of queues, stacks, deques, and counters, with correctness guarantees and deterministic relaxation bounds. Experiments show that these elastic capabilities incur minimal overhead. When combined with a lightweight controller for relaxation, they demonstrated an improved trade-off between throughput and work-efficiency compared to static designs.

Keywords

Semantic Relaxation, Concurrency, Data Structures, Multicore, Shared-Memory, FIFO Queue, Single-Source Shortest Path, Elastic Relaxation

List of Publications

Appended publications

This thesis is based on the following publications:

[**Paper A**] **K. von Geijer**, P. Tsigas, E. Johansson, S. Hermansson, *Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue*
Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), 2025, pp. 382–395.

[**Paper B**] M. D'Antonio*, **K. von Geijer***, T. Son Mai, P. Tsigas, H. Vandierendonck, *Relax and don't Stop: Graph-Aware Asynchronous SSSP*
Proceedings of the 1st FastCode Programming Challenge (FCPC), 2025, pp. 43–47.

[**Paper C**] **K. von Geijer**, P. Tsigas, *Elastic Relaxation of Concurrent Data Structures*
IEEE Transactions on Parallel and Distributed Systems (TPDS), 2025, vol. 36, no. 12, pp. 2578–2595.

*Authors contributed equally

Other publications

The following publications were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[**Paper a**] **K. von Geijer**, P. Tsigas, *How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures*
Euro-Par 2024: Parallel Processing, 2024, pp. 119-133.

Research Contribution

I was the lead author, designer, and implementor in **Paper A**, **Paper C**, and **Paper a**. Me and Marco D'Antonio shared the lead equally in **Paper B**.

Acknowledgment

First of all, I am grateful for my supervisor Philippas Tsigas for his steady guidance and trust in me. Thank you for always listening to my ideas, for our many discussions, and for taking a chance on me. I would also like to thank my co-supervisors Marina Papatriantafilou, for always checking in with me and for providing many teaching opportunities, and Vincenzo Massimiliano Gulisano, for your feedback and discussions. Thank you Marvin Williams for discussions regarding the analysis of the d-choice and multi-queue designs. Furthermore, many thanks to Marco D'Antonio and Hans Vandierendonck for our fruitful collaboration and interesting discussions.

I am honored and thankful to have professor Michael Spear serve as the discussion leader for my Licentiate seminar. I am also grateful for my Ph.D. examiner Aarne Ranta for his support.

I want to thank all the teachers and mentors I have had over the years, for believing in me and pushing me to go further. Thank you, Jonas Johnsson, for getting me to participate in Skolornas matematiktävling and for your warm teaching style. Thank you Jonas Skeppstedt for your interesting courses, contagious enthusiasm, and for helping me take my first steps toward pursuing a Ph.D.

A special thank you to Martin, Yixing, Vinh, Jingyu, and Jacob for making the journey so enjoyable. I want to thank my dear colleagues for all the corridor chats and for making a nice work environment (including, but not limited to): Huaifeng, Yenan, Atmane, Hashim, Wanya, Umer, Magnus, Romaric, Moui, Elad, and Thomas. Thank you to Katya, Lorenzo, and Krishna, whom I enjoyed working with in the Ph.D. council. I also want to thank those who came before, for warmly welcoming me into the group during my master thesis visit and during the start of my Ph.D. Thank you Dimitrios, Christos, Georgia, Bastian, and Kalle.

I want to express my gratitude to the Swedish Research Council for providing funding for my Ph.D. under grant No. 2021-05443.

Finally, I want to thank my family and friends. Thank you to my parents, for your unwavering love and support. I would never have gotten here without you. Thank you, Hanna, for always being there for me and brightening my days. At last, thank you to my lovely friends. I feel so lucky to have you all, and I am looking forward to continuing to enjoy your friendship throughout life.

Contents

Part I

Thesis Overview

1 Introduction

The exponential growth predicted by Moore's Law [1], [2] has defined the evolution of computer hardware for over half a century. This so-called law is a prediction from the sixties [1], refined in the seventies [2], by Intel's co-founder Gordon E. Moore who predicted that the number of transistors on a chip would double every two years. This prediction has been incredibly accurate until very recently [3]. For decades, this transistor scaling also translated into rapidly increasing CPU clock frequencies. Around 2005, however, this frequency scaling largely came to an end due to power and thermal limits [4], and architects started turning to parallelism. Since then, computers have shifted from one or a few cores to dozens or hundreds, ushering in the multicore age [5].

While this hardware parallelism allows for a high level of computation, it also places the onus on programmers to efficiently utilize the available parallelism. One of the central challenges in writing parallel programs is how to synchronize threads accessing the same resources [6]. This challenge is not unique to multicore programming, as the idiom *"Too many cooks spoil the broth"* suggests: coordination is hard. Programmers must ensure threads work effectively toward the same goal, avoid destroying each other's work, and preferably not spending all their time communicating with other threads.

Concurrent data structures are at the heart of many parallel applications. A naïve approach for making a data structure concurrent to completely encapsulate it within a mutual exclusion lock. However, this serializes all access and eliminates most potential parallelism, as only one thread can interact with the data structure at a time. To further exploit multicore hardware, developers use fine-grained synchronization schemes where multiple threads can work concurrently on disjoint parts of the data structure. Among the most prominent techniques enabling high-performance synchronization are lock-free and wait-free data structures [7]. These designs ensure that system progress is never blocked by individual thread failures or delays, yielding both robustness and practical efficiency.

Over the last few decades, the design of concurrent data structures has advanced dramatically. Fundamental structures such as stacks, queues, and lists have been refined to achieve ever-higher performance, leveraging efficient atomic primitives, sophisticated memory models, and careful cache-aware layouts. Despite these advances, many data structures still face scalability limits, as bottlenecks inevitably emerge when hundreds of threads contend for the same shared access point, such as a head or tail pointer. One promising solution to circumventing these issues is relaxed data structures [5], [8], which trade strict semantics for greater scalability. For example, in a relaxed FIFO queue, dequeues need not return the oldest item, and may instead return *one of* the oldest items. This relaxation preserves the spirit of the queue while enabling designs with far higher parallelism [9]–[12]. Different forms of relaxation have been proposed—ranging from bounded worst-case relaxations [8], [10]–[12] to probabilistic guarantees [9], [11], [13]–[15]—each balancing correctness and performance in distinct ways.

This thesis studies the use of relaxation in concurrent data structures

for shared-memory multicore systems. We focus mostly on queues–such as FIFO, LIFO, deques, and priority queues–whose strict ordering semantics make parallelization hard. Relaxed semantics alleviate this and make higher parallelism possible. Relaxed queues have for example proved efficient schedulers for parallel algorithms [9], [11], [13], [15]–[17]. Furthermore, we also focus on lock-free algorithms, as they provide desirable liveness guarantees and excellent performance in practice. In broad strokes, this thesis spans the area of relaxation by (i) designing a new relaxed FIFO queue in Paper A, (ii) efficiently utilizing relaxed data structures for graph traversals in Paper B, and (iii) extends pre-existing designs with elastic relaxation in Paper C.

The thesis is structured in two parts: Thesis Overview and Appended Papers. The thesis overview starts with this introduction. Section 2 then covers some basics of modern shared-memory multicore systems. Section 3 describes different shared-memory synchronization strategies, like non-blocking algorithms. Section 4 covers concurrent data structures, including correctness conditions, memory reclamation, and some foundational designs and design patterns. Section 5 similarly covers relaxed concurrent data structures, and also highlights research questions for the thesis. Section 6 then summarizes the the contributions of the thesis, including how they connect to the research questions. Section 7 concludes the thesis overview part with conclusions and future work directions. The Appended Papers part includes the three papers making up the contributions of this thesis. The appended papers were originally published in peer-reviewed venues and the appended versions differ mainly in presentation, with the following deviations. The appended Paper A includes a revised analysis, establishing the relaxation bound as $\mathcal{O}(n \log n)$ with high probability, in place of the previously stated $\mathcal{O}(n \log \log n / \log d)$. The appended Paper B adds four missing subplots to its Figure 2, which were previously cut due to space constraints.

2 Shared-Memory Multicore Systems

Designing practically efficient algorithms requires accounting for the underlying system. For example, state-of-the-art BFS differs greatly on CPUs [18] versus TCUs [19]. This section highlights the hardware aspects of shared-memory multicore systems most relevant to efficient parallel algorithms.

In multicore systems, a *software thread* is an execution context scheduled to run on hardware. We distinguish software threads, which are created by the program, from *hardware threads*, which are execution slots provided by the CPU capable of executing software threads. A multicore CPU has multiple cores, each of which may expose multiple hardware threads, with modern CPUs typically providing 2 hardware threads per core via simultaneous multithreading (SMT). A *process* is an operating system abstraction that contains one or more software threads and its own address space. We use *processor* to mean the physical multicore chip. Theoretical works sometimes use processor to mean an abstract unit of computation (roughly a software thread).

2.1 System and Memory Model

The *parallel random-access machine* (PRAM) [20], [21] models a shared-memory system with a single memory and many processors (often denoted by P), see Figure 1. Each instruction, including accessing memory, takes one unit of time. The simplicify of PRAM enables clean analyses, but algorithms which are theoretically optimal in PRAM are rarely optimal on real hardware.

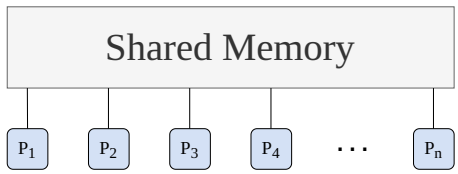


Figure 1: PRAM system model.

Modern systems have a hierarchical memory architecture [22]. As in Figure 2, several cache levels sit between each core and main memory. The L1 cache is small and fast, holding recently accessed data. Farther levels are larger but slower, with main memory the slowest relevant level for our purposes. Accessed data is brought into the L1 cache and less-used data is evicted to farther (slower) levels to make room for hot data. Caches typically work on a cache line granularily (txypically 64 bytes), instead of tracking and moving individual bytes. A read that finds the requested line in a given cache is a cache hit; otherwise it is a miss. Maximizing cache hits is critical for performance (see, e.g., cache-friendly queues [23]–[26]).

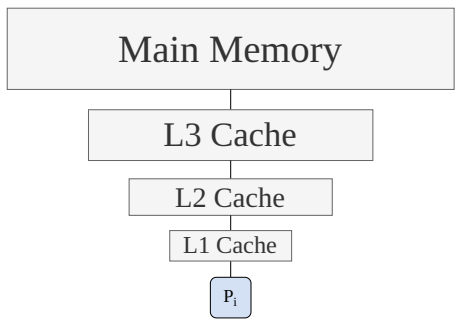


Figure 2: Hierarchical system memory model from the perspective of one core.

Many systems exhibit non-uniform memory access (NUMA). Figure 3 shows cores grouped into 8-core complexes sharing an L3 cache. If core P_1 reads data used by P_2 in the same complex, the request can be satisfied from the shared L3, avoiding main memory. Accessing data owned by P_{16} in another complex typically traverses main memory and is slower. Each core commonly exposes two hardware threads (sibling threads) via SMT. Siblings share private resources such as L1. The evaluation platform used in the appended papers (AMD EPYC 9754) has 2 hardware threads per core, 8 cores per complex,

16 complexes per socket (128 cores), and 2 sockets. Algorithm design should minimize inter-complex traffic and prefer locality within NUMA regions when feasible.

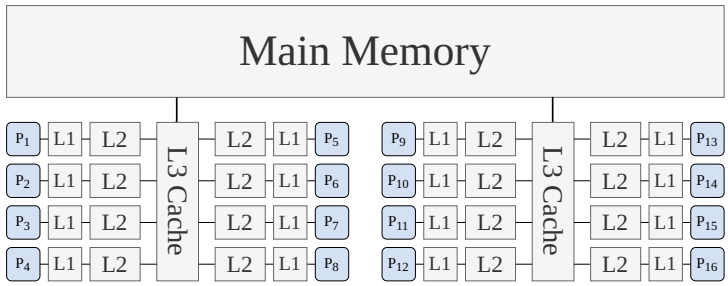


Figure 3: Example NUMA topology with two 8-core complexes with one shared L3 cache each.

Hardware provides *cache coherence*, ensuring that threads observe a consistent value for each memory location at all times. Most systems use invalidation-based protocols (e.g., MESI/MOESI), where multiple caches may hold a shared copy of a value, but writes require first invalidating other copies to obtain exclusivity. Therefore, read-mostly shared data is cheap, but frequently written shared data incurs invalidations and misses, degrading performance. Coherence operates at a cache line granularity. Writing to x invalidates the entire line, potentially evicting nearby y (*false sharing*). Padding or aligning per-thread data are common methods to avoid false sharing.

Modern CPUs include hardware *prefetchers* that speculatively fetch cache lines likely to be accessed soon. Common mechanisms include next-line and streaming prefetchers –capturing sequential scans– and simple stride detectors –capturing regular step patterns–. These work well for contiguous data such as arrays, but are far less effective for pointer-chasing structures like linked lists with their irregular access. Effective algorithms therefore structure hot paths to expose simple access patterns the prefetchers can learn, even if this leads to a slightly higher number of memory accesses.

Minimizing cache misses is paramount when designing efficient algorithms. On an AMD EPYC Rome system, typical memory latencies are: L1 $\approx$ 2 ns, L2 $\approx$ 6 ns, L3 $\approx$ 19.5 ns, DRAM $\approx$ 110 ns [27]. Thus, these three principles are useful to keep in mind when designing concurrent algorithms (1: thread locality) minimize writes to data contended by other threads, (2: temporal locality) attempt to return to the same set of data repeatedly, instead of excessively reading new cold memory, (3: spatial locality) prefer contiguous memory layouts such as arrays over pointer-chasing when possible.

2.2 Atomic Primitives

Plain reads and writes are insufficient for implementing many non-blocking synchronization tasks [7]. Consequently, shared-memory CPUs provide a set of atomic read-modify-write (RMW) primitives to coordinate data access

between threads. Atomic means that the operation appears indivisible and from software's perspective takes effect in a single step.

Algorithm 1 gives the semantics of *compare-and-swap* (CAS) and *fetch-and-add* (FAA) – the two mainly utilized primitives in this thesis. CAS atomically replaces a memory location with `desired` iff its current value equals `expected`; otherwise it fails. Many APIs (e.g., C/C++ atomic.h) overwrite the caller's `expected` argument on CAS failure with the observed value, for easier implementations of eg. CAS retry loops. CAS can suffer from the ABA problem on pointer-like values, and common mitigations include version tagging or hazard pointers (see §4.3). Some platforms provide *load-linked/store-conditional* (LL/SC) instead of CAS, which in turn can implement CAS. In practice, most portable libraries expose CAS-like operations, so algorithm descriptions usually assume CAS. On x86 processors, CAS is supported for up to 128-bit wide variables [28], but other architectures such as ARM only support up to 64-bit wide CAS [29]. x86-64 supports 16-byte CAS (e.g., `CMPXCHG16B`) [28]. On ARM, availability depends on the extension set and many systems only guarantee CAS up to 64 bits [29].

FAA atomically adds an integer to a location and returns the old value. Unlike CAS, FAA can not fail due to interference, so it often scales better for counters, indices, and ticket-style allocation. In highly contended hot-spots, replacing CAS with FAA (when semantics allow) can markedly improve throughput. This is for example seen in the LCRQ [23] FIFO queue, which shifts a hot-spot from CAS to FAA and significantly outscales earlier works. However, FAA is still a RMW operation and thus generates coherence traffic, meaning excessive use can remain a bottleneck.

Algorithm 1: Semantics of compare-and-swap (CAS) and fetch-and-add (FAA).

```
1.1  function CAS(address: pointer, expected: type, desired: type) → bool
1.2      atomically do
1.3          if *address = expected then
1.4              *address ← desired;
1.5              return true;
1.6          else return false;

1.7  function FAA(address: pointer, increment: integer) → integer
1.8      atomically do
1.9          oldvalue ← *address;
1.10         *address ← oldvalue + increment;
1.11         return oldvalue;
```

3 Synchronization

When several threads cooperate on a task, they must at some point synchronize to reach a consistent shared state. On shared-memory multicore systems, synchronization relies on atomic primitives (§2.2) combined into different strategies. The two main approaches are blocking (§3.1), which is simpler to use, and non-blocking (§3.2), which is more intricate but can be more efficient.

3.1 Blocking Methods

The traditional way of synchronizing access to some shared state is through mutual exclusion. Here we define certain blocks of code as *critical sections*, and ensures that only one thread can be in the critical section at a time. The common way to achieve this is through a *lock* object, which can be seen as a lock around the critical section with a single key. Such lock objects have two main methods: *acquire* and *release*. When a thread calls acquire, it waits its turn until it can get the shared key, after which it takes the key and enters the critical section. During the time the thread is in the critical section, we usually say it holds the lock, or is the lock owner. When the thread is done with the critical section, it calls release to give up the key, either to the next waiting thread or just leaving it for the next thread to arrive. Locks commonly also support the *try-lock* method, which tries to acquire the lock immediately, but if it is already held by another thread, the calling thread aborts to maybe do something else.

The traditional mechanism is *mutual exclusion*. Here, *critical sections* are created – often with *lock* objects – ensuring only one thread enters at a time. A lock typically exposes *acquire*, *release*, and optionally *try_lock*. A successful acquire operation grants the caller ownership of the lock. A release operation either transfers ownership to the next waiting thread or leaves the lock free. If the lock is already held, try_lock returns immediately so the caller can perform other work. We typically call the thread with lock ownership the *lock holder*.

Locks are widely used because they are simple and well supported by operating system and language runtimes. Blocking designs are often easier to implement and reason about than non-blocking ones (§3.2). However, they have some drawbacks, as we will soon describe.

There are many flavors of locks, each with different performance and fairness properties. A minimal test-and-set/CAS spinlock (Algorithm 2) flips a shared boolean to to true when aquiring the lock, and flips it back to false on release. Under contention this design causes heavy coherence traffic, which reduces throughput. A ticket lock [30], [31] instead assigns each acquirer a ticket via FAA and grants entry when the serving counter equals its ticket (Algorithm 3). Ticket locks provide FIFO fairness and reduce some contention by separating arrival (`ticket`) from waiting (`serving`). However, the single shared serving counter still invalidates globally on each release, causing unnecessary coherence traffic.

Even though there are many nice lock implementations [30]–[36], algorithms using them all suffer from common drawbacks. A major problem is vulnerability

Algorithm 2: A minimal CAS spinlock.

```
2.1 struct BasicLock
2.2     taken: bool;
2.3 function acquire(lock: BasicLock)
2.4     while !CAS(&lock.taken, false, true) do /* wait */ ;
2.5 function release(lock: BasicLock)
2.6     lock.taken ← false;
2.7 function try_lock(lock: BasicLock) → bool
2.8     return CAS(&lock.taken, false, true);
```

Algorithm 3: The Ticket Lock.

```
3.1 struct TicketLock
3.2     ticket_machine: int;
3.3     serving_counter: int;
3.4 function acquire(lock: TicketLock)
3.5     ticket ← FAA(&lock.ticket_machine, 1);
3.6     while lock.serving_counter ≠ ticket do /* wait */;
3.7 function release(lock: TicketLock)
3.8     FAA(&lock.serving_counter, 1);
```

to failures, as a stalled or crashed lock holder will prevent others from making progress. Additionally, composing multiple locks can create dependency cycles, where threads wait for each other indefinitely in a deadlock. Finally, time spent spinning or blocking reduces throughput, often leading to sub-optimal parallelism. These issues can be mitigated with scalable locks and careful design, but many high-performance data structures achieve better scalability with non-blocking methods.

3.2 Non-blocking Methods

The alternative to blocking is non-blocking synchronization, which avoids critical sections and mutual exclusion. Herlihy and Shavit [37] define non-blocking as a progress condition, where an arbitrary delay of one thread does not prevent other threads from making progress. The main progress conditions are *wait-freedom* and *lock-freedom* [7]. The weaker *obstruction-freedom* [38], also exists but is less commonly used in practice.

Obstruction-freedom [38] guarantees a thread makes progress if it runs in isolation for a finite number of steps. Obstruction-free operations often access multiple shared locations. Under contention, a write by one thread can invalidate another thread's tentative state and force a restart. Because the progress guarantee is weaker, these algorithms are often simpler to implement than lock-free or wait-free ones. Practical implementations often add a con-

tention manager that backs off colliding threads to create isolation windows. With effective contention management, obstruction-free designs can approach wait-free behavior in practice [39], although lacking its worst-case guarantees.

Lock-freedom [7] garantees that *one* thread makes progress in a finite number of system-wide steps. This is the most common non-blocking guarantee in practice, and many lock-free algorithms tend to behave as if they were wait-free in practice [40]. A classic lock-free pattern is to design data structures around a CAS retry loop [41]–[44], where a thread tries to complete its operation with a single CAS. If the CAS fails, another thread must have succeeded, ensuring the system-wide progress guarantee. Recent designs replace contended CAS loops with FAA where semantics permit [23]–[25]. As FAA cannot fail, updates serialize without retries, reducing wasted work on hot spots.

Wait-freedom [7] is the strongest progress condition, and guarantees that each thread makes progress in a finite number of steps. Unlike lock-freedom, it prevents a few fast threads from monopolizing progress. Universal constructions [7] and general techniques [45], [46] provide general recipes for creating wait-free implementations, but specialized designs often perform better [24]. A common technique is *helping*, where threads assist each other's in-flight operations to guarantee per-thread completion.

Compared to blocking, non-blocking methods provide stronger progress guarantees and often deliver better performance. The trade-off is algorithmic complexity and a higher risk of subtle bugs, which hinders more wide-spread adaption. Nonetheless, for data structures such as queues, stacks, and priority queues, which can often become hot-spots in larger applications, efficient non-blocking designs are well established [23]–[25], [42], [47]–[50]. That said, lock-based approaches can become competitive, for example by utilizing techniques such as batching or combining [51].

4 Concurrent Data Structures

Data structures efficiently organize and utilize data, and sit at the core of many algorithms. Concurrent data structures are ones designed to be access concurrently by multiple threads. Concurrency adds a new dimension to the design space, as efficient sequential designs often face difficulties scaling under parallel access. It also introduces correctness concerns since threads must synchronize in ways so that the data structure remains in a consistent state.

When implementing a concurrent data structure, an important decision is whether to design it with blocking (§3.1) or non-blocking (§3.2) synchronization. Blocking is generally simpler, especially with coarse-grained locks, while non-blocking often can reach higher scalability. Designers should also try to optimize for memory locality (§2.1), and avoid designs where many threads mutate the same cache cache line. Furthermore, one should prefer more scalable primitives when semantics allow, for example using FAA instead of CAS loops at hot-spots (§2.2).

4.1 Correctness

The most common correctness condition for concurrent data structures is *linearizability* [52]. Informally, a concurrent data structure is linearizable if all operations on it appears to take effect instantaneously, while upholding the sequential semantics of the data structure. This is powesful, as it unables us to reason about concurrent executions as if operations occurred in a total sequential order.

Linearizability is most easily proven by identifying so called *linearization points* for each operation. This is the instant at which the operation appears to take effect, or *linearize*. With coarse-grained locking, any point between the lock acquisition and release can serve as linearization point. Linearization points can become complex in non-blocking designs. In simple cases like the Michael and Scott queue [41] or Treiber stack [42], the operations repeatedly try to linearize with a single CAS, after which they they get help from other threads with potential finishing writes. Other designs such as LCRQ [23] use a sequence of writes to complete an operation and the linearization point may only be identifiable in retrospect.

For a slightly more complete definition of linearizability, consider a data structure *history* as a set of operation calls with arguments and returns with values. In a sequential history, each call is immediately followed by its return. In a concurrent history, operations from different threads can overlap, so calls and returns form a partial order that respects real-time orderings. A data structure is linearizable if for every partially ordered concurrent history, there exists an extension to a total order that yields a valid sequential history under the data structure specification [52].

Strong linearizability [53] strengthens linearizability by requiring a prefix-preserving mapping from concurrent histories to sequential histories. Intuitively, this means that the choice of linearization point cannot depend on future events. It is surprisingly hard to design strongly linearizable non-blocking data structures, and even very simple classical designs don't satisfy the condition [54]. The advantage of strong linearizability is that it – unlike linearizability – does not allow adversarials schedulers to control the probability distribution of the program result.

Quiescent consistency [6] requires that operations separated by a quiescent period (a period when no operation is executing) appear in real-time order. A data structure satisfies quiescent consistency if (1) all operations appear to take effect once at a time, and (2) all operations separated by a period of quiescence appear to take effect in real-time order. It is weaker than linearizability because it does not preserve real-time order for operations that are not separated by quiescence.

4.2 Memory Reclamation

Data structures often work with dynamic memory, which has to be allocated and deallocated (freed) dynamically during an execution. In sequential programs, or when using coarse-grained locking, one can directly free pieces of data when detached from the data structure. In non-blocking or fine-grained locking

designs, immediate deallocation becomes unsafe, as a concurrent traversal may still hold a reference to some node after it has been unlinked. If that node is freed, a reader can dereference an invalid pointer, which results in undefined behavior.

Innumerable memory reclamation schemes have been proposed. As an example, reference counting [55], [56] struggles with cycles and has performance overheads, but are simple to use and for example used by the CPython interpreter [57]. In the literature, two families dominate for non-blocking structures. Hazard pointers (HP) [58] and epoch-based reclamation (EBR) [59].

Hazard pointers [58] provide robust reclamation, even in the presence of thread failures. Each thread publishes the addresses it may dereference in a small array of hazard slots that is visible to others. When a node is unlinked it is retired to a per-thread list. Periodically, each thread scans the published hazard slots across threads and frees nodes in its own list that are not currently protected. If another thread still protects a node, reclamation is deferred. Hazard Pointers adds per-access overhead to protect pointers and incur scans whose complexity grow with the number of threads and hazard slots. They are also not generally applicable in all data structures. Recent works such as Hazard Eras [60] and HP++ [61] work toward improving performance and increasing applicability, respectively.

Epoch-based reclamation [59] trades some robustness for speed and easy of use. The core idea is that a global epoch counter advances over time and that each thread announces the epoch it is operating in. When an object is unlinked it is retired with the current epoch. It can later be freed once every active thread has advanced past the corresponding epoch, ensuring they don't retain any references to deallocated data. The speed of EBR comes from it avoiding per-pointer announcements, only occasionally updating its epoch. Although being fast and easy to applicate, it lacks robustness, as one stalled thread block all further data from being deallocated. Recent work, like Debra+ [62], NBR [63], and PEBR [64], use similar methods to EBR and resolve the robustness issue, but re-introduce issues like applicability.

4.3 Non-blocking Concurrent Data Structures

This section surveys classical non-blocking designs and recurring design patterns. It starts with the classical Treiber stack [42] and Michael & Scott queue [41], and then follows up with more efficient state-of-the-art queues such as LCRQ [23]. These designs are fundamental prerequisites for the appended Paper A and Paper C.

A common problem in non-blocking designs is the ABA problem. Consider a data structure which is implemented as a single mutable pointer to an otherwise immutable data structure. To make an operation on the data structure, you copy the whole current state, make a local modification, and then update the shared state by using CAS to change the shared pointer to your modified copy. If another thread has changed the pointer, you have to read the new shared pointer and recompute the new state before trying to linearize with CAS again. Now, the ABA problem is when you read A as the shared pointer and compute

your update, but before your CAS, some other thread swaps the shared pointer to B, deallocates A, allocates a new pointer which becomes A, and then uses CAS to change B to A. In this case, your CAS will succeed, as the shared pointer was A, as you expected, but what the pointer actually pointed to had changed, making your update incorrect. This is an important problem to keep in mind when using CAS. It can often be solved by a good memory reclamation scheme (§4.2), or by embedding a strict monotonic counter into the CAS object which makes the CAS distinguish identical states based on the counter.

A recurring pitfall is the *ABA problem*. The key issue is that CAS compares only value equality, not distinguishing different writes of one value. Suppose a shared pointer has value A. A thread reads A and prepares an update. Meanwhile another thread changes the pointer to B, frees the node at A, then allocates a new node that reuses the same address A, which is finally written back to the shared pointer. The first thread now sees the expected value A and its CAS succeeds, even though the underlying object is not the one it observed. Mitigations include robust memory reclamation that prevents reuse while references may exist (§4.2), and *tagged pointers* that pair the pointer with a version counter. Tags can overflow on small counters, so tagged pointers reduce rather than eliminate risk in some cases.

4.3.1 A Simple Lock-free Stack

The Treiber stack [42] is a beautifully simple lock-free stack, shown in Algorithm 4. The shared state is only the pointer to its top node. Each stack node stores a value and a link to the next node. A push reads the current top, sets the new node's next pointer to that top, and then uses CAS to swing the top pointer to the new node, restarting if failing the CAS. Similarly, popping a node from the stack also uses a CAS retry loop, trying to swing the top pointer to the second node in the stack.

This CAS-retry pattern appears in many lock-free designs. In this simple design, nodes are immutable after insertion and the only mutable shared state is the top pointer. Thus, operations repeatedly try to linearize with a successful CAS, and after a failed CAS do some small recomputations before trying to linearize again.

Algorithm 4 omits memory reclamation by design. The Treiber stack is commonly paired with hazard pointers [58] or EBR [59] as discussed in Section 4.2.

Finally, the Treiber stack is a great example of the ABA problem in action. With nodes A and B on the stack, thread 1 initiates a pop, reads A as the top, its next node as B, and then stalls. Thread 2 pops twice and later pushes a node that reuses A's address. Thread 1 resumes and its CAS on the top pointer from A to B now succeeds, even though the node at A is different. Robust reclamation (§4.2) or a tagged top pointer, avoids the bug. Tags must be sized to avoid wraparound in the expected lifetime of the program, with implementations often relying on the 16-byte CAS of x86-64 [28].

Algorithm 4: The Treiber Stack.

```
4.1  struct Stack
4.2      top: *StackNode;
4.3  struct StackNode
4.4      data: Any;
4.5      next: *StackNode;
4.6  function push(stack: *Stack, value: Any)
4.7      new_top ← StackNode{.data = value};
4.8      do
4.9          new_top.next ← stack.top;
4.10     while !CAS(&stack.top, new_top.next, new_top);
4.11 function pop(stack: *Stack) → Any|Null
4.12     do
4.13         top ← stack.top;
4.14         if top = Null then return Null;
4.15         next ← top.next;
4.16     while !CAS(&stack.top, top, next);
4.17     return top.data;
```

4.3.2 A Simple Lock-free Queue

The Michael and Scott (MS) FIFO queue [41] is the most foundational lock-free FIFO queue, and influences many later designs [10], [11], [23]–[25]. Algorithm 5 shows the simplified pseudocode, again omitting memory reclamation for simplicity.

Like the Treiber stack [42], the MS queue uses CAS loops to link and unlink nodes. A successful dequeue linearizes when CAS advances the head to the successive node at line 5.31. Enqueues linearize at line 5.17 by making the previous tail point to the newly enqueued node. Updating the shared tail pointer is a best-effort update for efficient tail access and any thread may help advance it at lines 5.18, 5.15, 5.28.

The drawback shared by both the MS queue and the Treiber stack is that they suffer poor scalability under high contention. When many threads want to dequeue an item at once, they all enter a CAS retry loop trying to update the same head pointer. This causes cache-line bouncing and a high rate of failed CAS attempts.

4.3.3 Faster Lock-free Queues

There have been many lock-free FIFO queue designs which improve on the MS queue [26], [44], [65], [66]. The LCRQ [23] was the first design to efficiently replace the CAS retry loop with FAA, achieving significantly improved scalability. It organizes the queue as a linked list – as the MS queue – of cyclic ring buffers. Each buffer maintains head and tail indices, which are updated with FAA. An enqueue (dequeue) reserves a slot by using FAA on the tail

Algorithm 5: The Michael & Scott FIFO Queue.

```
5.1  struct Queue
5.2      head: *QueueNode;
5.3      tail: *QueueNode;
5.4  struct QueueNode
5.5      data: Any;
5.6      next: *QueueNode;
5.7  function create_queue() → Queue
5.8      dummy ← QueueNode{.next = Null};
5.9      return Queue{.head = dummy, .tail = dummy};
5.10 function enqueue(queue: *Queue, value: Any)
5.11     new_tail ← QueueNode{.data = value, .next = Null};
5.12     do
5.13         old_tail ← queue.tail;
5.14         if old_tail.next ≠ Null then
5.15             CAS(&queue.tail, old_tail, old_tail.next);
5.16             continue;
5.17     while !CAS(&old_tail.next, Null, new_tail) ;
5.18     CAS(&queue.tail, old_tail, new_tail);
5.19 function dequeue(queue: *Queue) → Any|Null
5.20     while true do
5.21         head ← queue.head;
5.22         tail ← queue.tail;
5.23         next ← head.next;
5.24         if head = queue.head then
5.25             if head = tail then
5.26                 if next = Null then ;
5.27                 return Null;
5.28                 CAS(&queue.tail, tail, tail.next);
5.29             else
5.30                 data ← next.data;
5.31                 if CAS(&queue.head, head, next) then return data;
```

(head) buffer's tail (head) counter and writes (reads) its value to (from) the reserved index. The design includes mechanisms to handle wraparound, close and allocate new buffers, reclaim empty buffers, and preserve lock-freedom under stalled producers and consumers. The core idea is that hot spots are changed from CAS loops to FAA increments. This allows the LCRQ to scale significantly better than earlier works, and it is still among state-of-the-art.

One problem with the LCRQ is that it requires CAS2 (128-bit wide CAS). This is due to it having to write both the enqueued value itself into the buffer, as well as some metadata for the cyclic nature of the queue. A practical issue for LCRQ is the need for a wide (16-byte) CAS to update a value together with its metadata atomically. This is included for the cyclic buffers to function correctly. As CAS2 is not generally accessible in hardware, a recent work introduces the LPRQ [67] as a portable modification that only uses regularly sized CAS, while retaining similar performance.

The FAAArrayQueue [25] adopts the FAA reservation idea of the LCRQ, but removes its cyclic buffers. This trades some space efficiency at small sizes for a simpler algorithm that is easier to understand and extend. It performs very similarly to the LCRQ in many benchmarks.

The YMC queue [24] also builds on the LCRQ but removes its cyclic buffers to be able to achieve wait-freedom. It uses a fast-path that mirrors the FAA-reservation pattern and a slow-path that guarantees per-thread completion, following the fast-path slow-path approach [45]. Most operations take the fast-path while the slow-path enforces the wait-free bound with helping.

5 Relaxed Concurrent Data Structures and Research Questions

Although there are many smart data structure designs optimized for scalability, many of them still suffer from limited parallelism. For example, in the LCRQ [23] lock-free queue, all enqueues and dequeues contend for calling FAA on the tail or head counter, respectively. FAA does not fail like CAS, but the frequency at which the counter's cache line bounces between cores becomes a significant bottleneck. As long as operations modify a few shared variables, such as head or tail pointers or counters, these access points will incur high synchronization costs and become scalability bottlenecks.

Relaxed concurrent data structures alleviate the bottleneck of a few shared access points [5]. The mos common form is *out-of-order* relaxation [8], where the order of linearized operations may deviate up to some distance from the sequential specification. This deviation can be bounded deterministically [8], [10]–[12], [68] or probabilistically [9], [11], [13], [14], [69]. By changing the specification, operations no longer have to access the same few memory locations, allowing higher parallelism at the cost of some controlled disorder.

As an example, out-of-order relaxed FIFO queues [8] allow the dequeue operation to return out of order with respect to the linearization order of the corresponding enqueue operations. If using a bounded relaxation bound of k, such as the 2D queue [10], the dequeue may return any of the $k + 1$ oldest

items in the queue. If there are fewer than $k + 1$ dequeueing threads, then implementations could in theory direct each thread to dequeue a different item in parallel, alleviating the load on a single memory location.

A common design pattern for out-of-order designs is to split the data structure into n disjoint *sub-structures*. For instance, a relaxed FIFO queue can be built from n MS queues [41] as sub-queues. Then, each operation is directed to linearize on one of the n sub-queues. Different ways of selecting sub-queues to operate on results in different trade-offs between performance and relaxation [9]–[11], [13].

5.1 Correctness

One advantage of this type of relaxed data structures is that only the data structure semantics are relaxed, meaning linearizability [52] can still be used as a correctness condition. The most common semantic relaxation is out-of-order relaxation. Here, the order of operations is relaxed, but all other properties, like only deueueing each enqueued once, are retained. The effect of the relaxation is measured by the *relaxation error* of each operation, and the two main such errors are *rank errors* and *delays*. The rank error is the number of previous operations we would ignore to make the relaxed operation correct, such as the number of items with higher rank than the dequeued item in a priority queue. Delay is used mainly for queues and, for each dequeued item, counts how many other dequeues utilize relaxation to skip the specified item, effectively postponing its removal.

Quantitative relaxation is a theoretical framework for defining semantic relaxations. The idea is to define a cost function over the transitions between each two states in the linearized history. The paper defines k-relaxation, where each such transition cost is bounded by k. The framework formalizes and introduces k-out-of-order relaxation, and its *rank errors* (although with another term). It is a useful foundation for defining relaxations, but it only targets relaxations with worst-case bounds.Thes becomes limiting for later designs such as the MultiQueue [70], which only provide probabilistic error guarantees rather than worst-case bounds.

Following quantitative relaxation [8], *distributional linearizability* [14] was proposed as a correctness condition for relaxed concurrent data structures with probabilistic bounds. The paper [14] also gavn an early analysis of the popular MultiQueue [70], under simplifying assumptions. Adaption has been limited, likely because the model includes the scheduler of the concurrent algorithm. Later work often simplifies the analysis to the sequential setting, where it derives probabilistic bounds on the relaxation errors [9], [13], [71].

In summary, relaxation errors are measured in linearized histories, meaning linearizability first has to be used to linearize the concurrent history. The main error is the out-of-order rank error, formalized by quantitative relaxation [8]. A second common error is the delay, mainly used for queues [9], [72]. Although quantitative relaxation [8] and distributional linearizability [14] cover many designs, they assume static relaxation and do not caputer designs where the relaxation can be reconfigured on the fly. This leads to our first research

question:

Research Question 1

What is a good correctness condition for relaxed data structures where the relaxation can be re-configured during runtime?

5.2 Relaxed Concurrent FIFO Queues

FIFO queues, together with priority queues, are among the most studied relaxed data structures. Here we highlight a few of the prominent relaxed FIFO queues.

The k-segment queue [12] is perhaps the simplest relaxed FIFO queue. It can be seen as a MS queue [41] whose nodes are segments of size k. Operations respect the order between segments, while the order within a segment is unspecified. For example, when enqueueing an item, the thread searches for an empty slot in the tail segment, trying to linearize by writing its item into that slot with a CAS. If no slot is found, a new segment is enqueued in an MS queue fashion. The design is simple and yields rank error and delay bounds of $k - 1$. However, it suffers from scalability issues as the search for a valid spot becomes a bottleneck at larger values of k.

The 2D framework [10] defines relaxed counters, a FIFO queue, LIFO queues, and a deque, with worst-case rank errors. The idea is use n sub-structures and superimpose a *window* of a given *depth* over the sub-structures. The window specifies what sub-structures are eligible to linearize on, for example being allowed to do *depth* linearizations on each sub-structure each window. As in the k-queue [12], the order within a window is undefined, but the order between windows is preserved. For a 2D queue with window depth of 4, threads may enqueue up to 4 items per sub-queue each window, with each sub-queue becoming saturated after the fourth enqueue. When all sub-queues are saturated, the window shifts upward, enabling 4 more enqueues per sub-queue. The queue maintains two windows, one at the tail for enqueues and one at the head for dequeues. The 2D queue uses MS queues [41] as sub-queues. The 2D queue resembles the k-segment queue but improves scalability via the window depth, reducing search time and improving memory locality. To our knowledge, it is the fastest published relaxed FIFO queue with bounded rank and delay errors, especially at higher allowed relaxation levles.

The d-RA queue [11] takes a different approach than k-segment [8], [12] and 2D [10] queues. It maintains n sub-queues, implemented as MS queues [41]. To enqueue an item, a thread samples d sub-queues ($d \geq 2$) at random, estimating their sizes from head and tail ABA counters, and linearizes on the smallest sampled sub-queue. Similarly, dequeues linearize on the largest of the d sampled sub-queues. This yields good scalability as the sampling cost is independent of thread count and number of sub-queues. However, the paper does not give theoretical guarantees on relaxation errors, which leads to our second research question:

Research Question 2

Does the d-RA load-balancing scheme provide probabilistic guarantees on rank or delay errors? If not, can it be adapted so that errors solely depend on the number of sub-queues?

There are other relaxed FIFO queues, such as the family of distributed queues [11], but the above implementations cover state-of-the-art. Currently, all relaxed queues using the sub-queue design pattern, such as the d-RA [11], distributed queues [11], and 2D queue [10], use the MS queue [41] as their sub-queue. However, as noted in Section 4.3, the MS queue slower compared than more recent FIFO designs. This leads us to our next research question:

Research Question 3

Can state-of-the-art FIFO queues be integrated as sub-queues in relaxed FIFO queues to improve performance?

5.3 Relaxed Concurrent Priority Queues

Relaxed priority queues have received significant attention recent years, in part due to their potential to parallelize algorithms like single-source shortest-path (SSSP) [13], [15]–[17], [68]. The SprayList [69] keeps a concurrent skiplist [73] with non-relaxed insertions, while deletions perform a random walk to remove an item near the head, spreading out operations and reducing contention. The k-LSM [68] keeps a small per-thread LSM [74], only synchronizing with an unbounded shared LSM when the local becomes full. Deletions remove the highest priority item from the local and one of the highest priority ones in the shared LSM. The CA-PQ [75] follows a similar idea with an unbounded shared priority queue, and threads insert and delete items in large batches under high contention. These designs all have their own trade-offs, and are able to significantly outperform strict concurrent priority queues.

The most popular relaxed priority queue design in recent years is the MultiQueue [70], similar in spirit to d-RA (§5.2). It maintains n concurrent priority queues as sub-queues. Each insertion linearizes at one sub-queue uniformly at random. Each deletion samples d sub-queues ($d \geq 2$) and deletes from the sub-queue whose top has the highest priority. The beauty of the MultiQueue is its simplicity and extensibility. Furthermore, there has been significant efforts in theoretically analyzing the MultiQueue, showng that its relaxation errors can be probabilistically bounded in terms of the number of sub-queues [14], [71], [76], under some simplifying assumptions.

Although the base MultiQueue [70] scales well and exhibit low relaxation error, it suffers from poor cache locality, as each operation tends to sample and linearize on different sub-queues. Recent extensions [13], [72] mitigate this with *buffering* and *stickiness*. Buffering adds flat insertion and deletion buffers in front of each sub-queue so that most operations only hit a flat buffer, improving spatial locality. Stickiness alters the load balancing such that each thread stays

with a small set of sub-queues for several successive operations, improving temporal locality at the cost of higher relaxation errors. These optimizations lead to a very fast implementation with good rank-error guarantees that outperforms earlier relaxed priority queues on most benchmarks.

The base MultiQueue idea [70] also gave rise to variants such as the Stealing MultiQueue [17] and MultiBucketQueue [15]. These show strong results on SSSP and related benchmarks, but often require careful tuning across inputs to reach peak performance. It would be preferable if one could use a less specialized data structure that required less tuning to be used efficiently for general algorithms such as SSSP, which leads into our final research question:

Research Question 4

Can general relaxed data structures be competitive as parallel schedulers against tailored state-of-the-art algorithms in parallel graph processing problems?

6 Thesis Contributions

This section briefly describes the main contributions of each appended paper and how they relate to the research questions in Section 5.

6.1 Efficient Relaxed Data Structure Designs

Relaxed data structures should ideally have high performance, for example measured in throughput or latency, while keeping relaxation errors close to zero. In practice this is rarely achievable, so designs must carefully balance performance and relaxation.

As described in Section 5, the sub-structure design pattern is popular for creating relaxed designs. As noted in Research Question 2, the d-RA queue [11] achieves great scalability with threads, but lacks analysis of its relaxation errors. In Paper A, we show that the d-RA rank errors can scale with queue size, which is neither expected nor prefered.

Paper A introduces the d-CBO queue, which similarly to d-RA uses the d-choice [77] to select sub-queue for operations, although in a different manner. It retains good scalability, while simultaneously probabilistically bounding relaxation errors with the number of sub-queues. The d-CBO selects a sub-queue for its enqueue (dequeue) by sampling d sub-queues and linearizing on the sampled one with fewest earlier enqueues (dequeues). Our analysis shows that, with high probability in n (the number of sub-queues), the d-CBO queue has delay and rank errors of $\mathcal{O}(n \log n)$ under some simplifying assumptions following the MultiQueue analysis [14], [71], [72], [76]. Experimentally, the d-CBO has stable, low rank errors, averaging around n for $d = 2$.

While the d-CBO queue has low relaxation errors, good designs must also be fast. As highlighted by Research Question 3, one method to increase performance is to use faster sub-structures. State-of-the-art FIFO queues are

harder to use as sub-queues than the MS queue [41], but the d-CBO manages to only rely on a generic sub-queue interface that can be implemented by all of the state-of-the-art queues, significantly improving performance.

Thus, the d-CBO design from Paper A both reduces relaxation errors – via a new load balancing scheme – and improves performance – via a flexible, simple interface that suits state-of-the-art concurrent FIFO queues. Compared to d-RA, its most similar competitor, it simultaneously achieves higher throughput and lower relaxation errors. It offers a better trade-off between relaxation and performance than other state-of-the-art relaxed FIFO queues.

6.2 Relaxation in Efficient Graph Processing

One of the exciting applications of relaxed data structures is within graph algorithms such as SSSP. While previous work has applied relaxed data structures to these problems, papers benchmarking against state-of-the-art within these problems [15], [17] often require careful tuning and deviations from simple base relaxed designs like the MultiQueue [70]. Research Question 4 therefore asks whether less custom-tailored relaxed implementations can compete with state-of-the-art within these problems.

Paper B presents the parallel SSSP solver AdaMW, with which we won the FastCode Programming Challenge at PPoPP 2025, challenging researchers to create the fastest possible parallel SSSP solvers. AdaMW is graph-aware, configuring itself based on sampling the input graph, and selecting either Wasp [78] or the MultiQueue [13] to solve the SSSP. Wasp can be seen as a relaxed sheduler based on bucket queues and work-stealing, but it is not a general relaxed data structure. By contrast, the MultiQueue is an general relaxed priority queue, and with just a few SSSP-specific optimizations, it performed extremely well. On low-diameter graphs, Wasp generally performed better, but the MultiQueue was often very competetive. On high-diameter graphs, the MultiQueue dominated the competition. The paper also showed that the MultiQueue was very forgiving to configuring, while other methods only have a narrow band of optimal settings.

Paper A and Paper C both benchmark their queues on parallel BFS. Paper C shows that elastic relaxation can reduce work and execution time over static relaxation. Paper A shows that the d-CBO queue is faster for BFS than other relaxed and non-relaxed concurrent FIFO queues. However, neither paper compares against state-of-the-art BFS algorithms and further work is needed to match direction-optimizing approaches [18], especially on low-diameter graphs.

6.3 Elastic Relaxation of Concurrent Data Structures

As shown in Paper B, optimally configuring relaxed data structures is hard and depends on the characteristics of the targeted workload. Thus, workloads with temporal variation would benefit from relaxed data structures that can be elastically reconfigured at runtime. Research Question 1 thus addresses how such elastic relaxation should work.

Paper C introduces the term *elastic relaxation* for relaxed data structures that can be reconfigured during runtime. The elastic correctness condition builds on the relaxation errors defined in the quantitative relaxation framework [8], but instead of a static bound for the whole execution, the bound is a function of the relaxation history up to that point.

Paper C goes on to extend the 2D framework for relaxed data structures [10] to encompass elastic relaxation. It identifies two patterns for elastically extending 2D data structures, yielding two FIFO queues, two stacks, one counter, and one deque. Experiments show that the elastic capabilities incur negligible overheads when unused. Furthermore, the paper designs a lightweight controller for dynamically adapting relaxation based on contention. Utilizing the dynamic controller demonstrated improved trade-offs between throughput and relaxation errors than static implementations in a parallel BFS macro-benchmark.

7 Conclusions and Future Work

This thesis introduces, extends, and applies, out-of-order relaxed concurrent data structures. These data structures achieve far better performance at high thread counts than classical designs – more efficiently utilizing hardware parallelism – and can serve as building blocks in fast parallel algorithms. Our d-CBO queue attains low relaxation errors via a new sub-queue selection scheme and high throughput in part through its use of efficient sub-queues. Our elastic designs can trade relaxation for performance at runtime across a range of data structures. We also apply relaxed data structures to outperform state-of-the-art on SSSP for sparse graphs, indicating the broader value of relaxation in parallel algorithms.

There are several promising directions for future work within relaxation. One direction is to extend the d-CBO queue. Practically, performance could be improved by improving cache locality – for example adding MultiQueue-like stickiness, a key factor in the MultiQueue's competitive SSSP performance. Analytically, it would be useful to analyze rank and delay errors in expectation and extend the theory toward histories better reflect arbitrary concurrent executions.

Another direction is to apply elastic relaxation to relaxed priority queues such as the MultiQueue, given their demonstrated usefulness. Adjusting d in the d-choice, or the stickiness at runtime is straightforward and a natural first step. Efficiently changing the number of sub-queues at runtime is harder and an orthogonal configuration, making it an interesting problem.

Finally, the field of relaxed data structures would benefit from increased cross-pollination with parallel algorithms. We took a step by using the MultiQueue for parallel SSSP in the FCPC competition, and we hope it will be one of many. Since the MultiQueue does not target SSSP specifically, it would be interesting to test competitiveness in parallel algorithms for other problems. Another open question is whether analysis of relaxation errors can be used to bound application properties such as work efficiency.

Bibliography

[1] G. E. Moore, 'Cramming more components onto integrated circuits,' *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[2] G. E. Moore, 'Progress in digital integrated electronics [technical literaiture, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.],' *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 36–37, 2006. DOI: 10.1109/N-SSC.2006.4804410.

[3] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez and T. B. Schardl, 'There's plenty of room at the top: What will drive computer performance after moore's law?' *Science*, vol. 368, no. 6495, eaam9744, 2020. DOI: 10.1126/science.aam9744. eprint: https://www.science.org/doi/pdf/10.1126/science.aam9744.

[4] P. E. Ross, 'Why cpu frequency stalled,' *IEEE Spectrum*, vol. 45, no. 4, pp. 72–72, 2008. DOI: 10.1109/MSPEC.2008.4476447.

[5] N. Shavit, 'Data structures in the multicore age,' *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011. DOI: 10.1145/1897852.1897873.

[6] M. Herlihy, N. Shavit, V. Luchangco and M. Spear, *The Art of Multiprocessor Programming*. Elsevier Science, 2020.

[7] M. Herlihy, 'Wait-free synchronization,' *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991. DOI: 10.1145/114005.102808.

[8] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin and A. Sokolova, 'Quantitative relaxation of concurrent data structures,' *SIGPLAN Not.*, vol. 48, no. 1, pp. 317–328, 2013. DOI: 10.1145/2480359.2429109.

[9] K. von Geijer, P. Tsigas, E. Johansson and S. Hermansson, 'Balanced allocations over efficient queues: A fast relaxed fifo queue,' in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 382–395. DOI: 10.1145/3710848.3710892.

[10] A. Rukundo, A. Atalar and P. Tsigas, 'Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework,' in *33rd International Symposium on Distributed Computing (DISC 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 146, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 31:1–31:15. DOI: 10.4230/LIPIcs.DISC.2019.31.

[11] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch and A. Sezgin, 'Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation,' in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy: Association for Computing Machinery, 2013. DOI: 10.1145/2482767.2482789.

[12] C. M. Kirsch, H. Payer, H. Röck and A. Sokolova, 'Performance, scalability, and semantics of concurrent fifo queues,' in *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'12, Fukuoka, Japan: Springer-Verlag, 2012, pp. 273–287. DOI: 10.1007/978-3-642-33078-0_20.

[13] M. Williams and P. Sanders, 'The multiqueue: A simple and fast relaxed concurrent priority queue,' *ACM Trans. Parallel Comput.*, Oct. 2025, Just Accepted. DOI: 10.1145/3771738.

[14] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li and G. Nadiradze, 'Distributionally linearizable data structures,' in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, vol. test, Vienna, Austria: Association for Computing Machinery, 2018, pp. 133–142. DOI: 10.1145/3210377.3210411.

[15] G. Zhang, G. Posluns and M. C. Jeffrey, 'Multi bucket queues: Efficient concurrent priority scheduling,' in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24, Nantes, France: Association for Computing Machinery, 2024, pp. 113–124.

[16] M. D'Antonio, K. von Geijer, T. S. Mai, P. Tsigas and H. Vandierendonck, 'Relax and don't stop: Graph-aware asynchronous sssp,' in *Proceedings of the 1st FastCode Programming Challenge*, ser. FCPC '25, The Westin Las Vegas Hotel & Spa, Las Vegas, NV, USA: ACM, 2025, pp. 43–47.

[17] A. Postnikova, N. Koval, G. Nadiradze and D. Alistarh, 'Multi-queues can be state-of-the-art priority schedulers,' in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 353–367. DOI: 10.1145/3503221.3508432.

[18] S. Beamer, K. Asanović and D. Patterson, 'Direction-optimizing breadth-first search,' in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012.

[19] Y. Niu and M. Casas, 'Berrybees: Breadth first search by bit-tensor-cores,' in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 339–354. DOI: 10.1145/3710848.3710859.

[20] S. Fortune and J. Wyllie, 'Parallelism in random access machines,' in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ser. STOC '78, San Diego, California, USA: Association for Computing Machinery, 1978, pp. 114–118. DOI: 10.1145/800133.804339.

[21] J. JaJa, *An Introduction to Parallel Algorithms*. Reading, MA: Addison-Wesley, 1992.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[23] A. Morrison and Y. Afek, 'Fast concurrent queues for x86 processors,' in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Shenzhen, China: Association for Computing Machinery, 2013, pp. 103–112. DOI: 10.1145/2442516.2442527.

[24] C. Yang and J. Mellor-Crummey, 'A wait-free queue as fast as fetch-and-add,' in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, Spain: Association for Computing Machinery, 2016. DOI: 10.1145/2851141.2851168.

[25] P. Ramalhete, *FAAArrayQueue - MPMC Lock-Free Queue*, Accessed: 2024-07-29, 2016. [Online]. Available: http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html.

[26] A. Gidenstam, H. Sundell and P. Tsigas, 'Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency,' in *Principles of Distributed Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 302–317. DOI: https://doi.org/10.1007/978-3-642-17653-1_23.

[27] M. Velten, R. Schöne, T. Ilsche and D. Hackenberg, 'Memory performance of amd epyc rome and intel cascade lake sp server processors,' in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22, Beijing, China: Association for Computing Machinery, 2022, pp. 165–175. DOI: 10.1145/3489525.3511689.

[28] Intel Corporation, *Intel® 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4*, Version 080, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[29] ARM Ltd., *Arm architecture reference manual armv8, for armv8-a architecture profile*, ARM DDI 0487F.b, ARM Ltd., 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest.

[30] M. J. Fischer, N. A. Lynch, J. E. Burns and A. Borodin, 'Resource allocation with immunity to limited process failure,' in *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, ser. SFCS '79, USA: IEEE Computer Society, 1979, pp. 234–254. DOI: 10.1109/SFCS.1979.37.

[31] J. M. Mellor-Crummey and M. L. Scott, 'Algorithms for scalable synchronization on shared-memory multiprocessors,' *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991. DOI: 10.1145/103727.103729.

[32] T. S. Craig, 'Building FIFO and priority-queueing spin locks from atomic swap,' Department of Computer Science and Engineering, University of Washington, Seattle, WA, Tech. Rep. UW-CSE-93-02-02, Feb. 1993. [Online]. Available: https://dada.cs.washington.edu/research/tr/1993/02/UW-CSE-93-02-02.pdf.

[33] P. S. Magnusson, A. Landin and E. Hagersten, 'Queue locks on cache coherent multiprocessors,' in *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, Cancún, Mexico: IEEE, Apr. 1994, pp. 165–171.

[34] D. Dice and A. Kogan, 'Hemlock: Compact and scalable mutual exclusion,' in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 173–183. DOI: 10.1145/3409964.3461805.

[35] D. Dice, V. J. Marathe and N. Shavit, 'Lock cohorting: A general technique for designing numa locks,' in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, New Orleans, Louisiana, USA: Association for Computing Machinery, 2012, pp. 247–256. DOI: 10.1145/2145816.2145848.

[36] A. Agarwal and M. Cherian, 'Adaptive backoff synchronization techniques,' in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ser. ISCA '89, Jerusalem, Israel: Association for Computing Machinery, 1989, pp. 396–406. DOI: 10.1145/74925.74970.

[37] M. Herlihy and N. Shavit, 'On the nature of progress,' in *Principles of Distributed Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 313–328.

[38] M. Herlihy, V. Luchangco and M. Moir, 'Obstruction-free synchronization: Double-ended queues as an example,' in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ser. ICDCS '03, USA: IEEE Computer Society, 2003, p. 522.

[39] F. E. Fich, V. Luchangco, M. Moir and N. Shavit, 'Obstruction-free algorithms can be practically wait-free,' in *Proceedings of the 19th International Conference on Distributed Computing*, ser. DISC'05, Cracow, Poland: Springer-Verlag, 2005, pp. 78–92. DOI: 10.1007/11561927_8.

[40] D. Alistarh, K. Censor-Hillel and N. Shavit, 'Are lock-free concurrent algorithms practically wait-free?' *J. ACM*, vol. 63, no. 4, Sep. 2016. DOI: 10.1145/2903136.

[41] M. M. Michael and M. L. Scott, 'Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,' in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275. DOI: 10.1145/248052.248106.

[42] R. Treiber, 'Systems programming: Coping with parallelism,' International Business Machines Incorporated, Thomas J. Watson Research Center, Tech. Rep., 1986.

[43] M. M. Michael, 'Cas-based lock-free algorithm for shared deques,' in *Euro-Par 2003 Parallel Processing*, Springer Berlin Heidelberg, 2003, pp. 651–660.

[44] P. Tsigas and Y. Zhang, 'A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems,' in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01, Crete Island, Greece: Association for Computing Machinery, 2001, pp. 134–143. DOI: 10.1145/378580.378611.

[45] A. Kogan and E. Petrank, 'A methodology for creating fast wait-free data structures,' in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, New Orleans, Louisiana, USA: Association for Computing Machinery, 2012, pp. 141–150. DOI: 10.1145/2145816.2145835.

[46] P. Fatourou and N. D. Kallimanis, 'A highly-efficient wait-free universal construction,' in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 325–334. DOI: 10.1145/1989493.1989549.

[47] D. Hendler, N. Shavit and L. Yerushalmi, 'A scalable lock-free stack algorithm,' *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 1–12, 2010. DOI: 10.1016/j.jpdc.2009.08.011.

[48] H. Sundell and P. Tsigas, 'Fast and lock-free concurrent priority queues for multi-thread systems,' *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005. DOI: https://doi.org/10.1016/j.jpdc.2004.12.005.

[49] A. Braginsky, N. Cohen and E. Petrank, 'Cbpq: High performance lock-free priority queue,' in *Euro-Par 2016: Parallel Processing*, Cham: Springer International Publishing, 2016, pp. 460–474.

[50] J. Lindén and B. Jonsson, 'A skiplist-based concurrent priority queue with minimal memory contention,' in *Principles of Distributed Systems*, Cham: Springer International Publishing, 2013, pp. 206–220.

[51] D. Hendler, I. Incze, N. Shavit and M. Tzafrir, 'Flat combining and the synchronization-parallelism tradeoff,' in *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10, Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 355–364. DOI: 10.1145/1810479.1810540.

[52] M. P. Herlihy and J. M. Wing, 'Linearizability: a correctness condition for concurrent objects,' *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990. DOI: 10.1145/78969.78972.

[53] W. Golab, L. Higham and P. Woelfel, 'Linearizable implementations do not suffice for randomized distributed computation,' in *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 373–382. DOI: 10.1145/1993636.1993687.

[54] S. M. Hwang and P. Woelfel, 'Strongly Linearizable Linked List and Queue,' in *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 217, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 28:1–28:20. DOI: 10.4230/LIPIcs.OPODIS.2021.28.

[55] J. D. Valois, 'Lock-free linked lists using compare-and-swap,' in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95, Ottowa, Ontario, Canada: Association for Computing Machinery, 1995, pp. 214–222. DOI: 10.1145/224964.224988.

[56] A. Gidenstam, M. Papatriantafilou, H. Sundell and P. Tsigas, 'Efficient and reliable lock-free memory reclamation based on reference counting,' in *Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, ser. ISPAN '05, USA: IEEE Computer Society, 2005, pp. 202–207. DOI: 10.1109/ISPAN.2005.42.

[57] Python Software Foundation, *Python/c api reference manual*, Accessed: 2025-10-24, 2025. [Online]. Available: https://docs.python.org/3/c-api/intro.html.

[58] M. M. Michael, 'Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects,' *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004. DOI: 10.1109/TPDS.2004.8.

[59] K. Fraser, 'Practical lock-freedom,' Ph.D. dissertation, University of Cambridge, 2003.

[60] P. Ramalhete and A. Correia, 'Brief announcement: Hazard eras - non-blocking memory reclamation,' in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 367–369. DOI: 10.1145/3087556.3087588.

[61] J. Jung, J. Lee, J. Kim and J. Kang, 'Applying hazard pointers to more concurrent data structures,' in *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '23, Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 213–226. DOI: 10.1145/3558481.3591102.

[62] T. A. Brown, 'Reclaiming memory for lock-free data structures: There has to be a better way,' in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, ser. PODC '15, Donostia-San Sebastián, Spain: Association for Computing Machinery, 2015, pp. 261–270. DOI: 10.1145/2767386.2767436.

[63] A. Singh, T. Brown and A. Mashtizadeh, 'Nbr: Neutralization based reclamation,' in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 175–190. DOI: 10.1145/3437801.3441625.

[64] J. Kang and J. Jung, 'A marriage of pointer- and epoch-based reclamation,' in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 314–328. DOI: 10.1145/3385412.3385978.

[65] M. Hoffman, O. Shalev and N. Shavit, 'The baskets queue,' in *Principles of Distributed Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 401–414. DOI: https://doi.org/10.1007/978-3-540-77096-1_29.

[66] M. Moir, D. Nussbaum, O. Shalev and N. Shavit, 'Using elimination to implement scalable and lock-free fifo queues,' in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05, Las Vegas, Nevada, USA: Association for Computing Machinery, 2005, pp. 253–262. DOI: 10.1145/1073970.1074013.

[67] R. Romanov and N. Koval, 'The state-of-the-art lcrq concurrent queue algorithm does not require cas2,' in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 14–26. DOI: 10.1145/3572848.3577485.

[68] M. Wimmer, J. Gruber, 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*, ser. PPoPP 2015, San Francisco, CA, USA: ACM, 2015, pp. 277–278. DOI: 10.1145/2688500.2688547.

[69] D. Alistarh, J. Kopinsky, J. Li and N. Shavit, 'The spraylist: A scalable relaxed priority queue,' *SIGPLAN Not.*, vol. 50, no. 8, pp. 11–20, 2015. DOI: 10.1145/2858788.2688523.

[70] H. Rihani, P. Sanders and R. Dementiev, 'Multiqueues: Simple relaxed concurrent priority queues,' in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 80–82. DOI: 10.1145/2755573.2755616.

[71] S. Walzer and M. Williams, 'A Simple yet Exact Analysis of the MultiQueue,' in *33rd Annual European Symposium on Algorithms (ESA 2025)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 351, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 85:1–85:14. DOI: 10.4230/LIPIcs.ESA.2025.85.

[72] M. Williams, P. Sanders and R. Dementiev, 'Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues,' in *29th Annual European Symposium on Algorithms (ESA 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17. DOI: 10.4230/LIPIcs.ESA.2021.81.

[73] W. Pugh, 'Skip lists: A probabilistic alternative to balanced trees,' *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990. DOI: 10.1145/78973.78977.

[74] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, 'The log-structured merge-tree (lsm-tree),' *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996. DOI: 10.1007/s002360050048.

[75] K. Sagonas and K. Winblad, 'The Contention Avoiding Concurrent Priority Queue,' in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 314–330.

[76] D. Alistarh, J. Kopinsky, J. Li and G. Nadiradze, 'The power of choice in priority scheduling,' in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 283–292. DOI: 10.1145/3087801.3087810.

[77] Y. Azar, A. Z. Broder, A. R. Karlin and E. Upfal, 'Balanced Alocations,' *SIAM Journal on Computing*, vol. 29, no. 1, pp. 180–200, 1999, A preliminary version appeared in *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 593–602, Montreal, Quebec, Canada, May 23–25, 1994. ACM Press, New York, NY.

[78] M. D'Antonio, S. T. Mai and H. Vandierendonck, ' Toward Efficient Asynchronous Single-Source Shortest Path,' in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2025.

Part II

Appended Papers

Paper A

Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue

Kåre von Geijer, Philippas Tsigas, Elias Johansson, Sebastian Hermansson

Revised version of the paper published in Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), 2025, pp. 382–395

doi: 10.1145/3710848.3710892

Abstract

Relaxed semantics have been introduced to increase the achievable parallelism of concurrent data structures in exchange for weakening their ordering semantics. In this paper, we revisit the balanced allocations d-choice load balancing scheme in the context of relaxed FIFO queues. Our novel load balancing approach distributes operations evenly across n sub-queues based on operation counts, achieving low relaxation errors independent on the queues size, as opposed to similar earlier designs. We prove its relaxation errors to be of $\mathcal{O}(n \log n)$ with high probability in n for a collection of possible executions. Furthermore, our scheme, contrary to previous ones, manages to interface and integrate the most performant linearizable queue designs from the literature as components. Our resulting relaxed FIFO queue is experimentally shown to outperform the previously best design using balanced allocations by more than four times in throughput, while simultaneously incurring less than a thousandth of its relaxation errors. In a concurrent breadth-first-search benchmark, our queue consistently outperforms both relaxed and strict state-of-the-art FIFO queues.

1 Introduction

The available hardware parallelism has seen tremendous growth the last two decades, which in turn has heightened the need for efficient concurrent data structures [1]. The design of efficient concurrent queues has proven to be a hot area, and there are now many efficient FIFO queues [2], [3], stacks [4], [5] and priority queues [6], [7]. However, the scalability of such sequential designs is limited, as all operations must form a total order and often contend for the same few memory locations [8].

A popular approach to further increase scalability of such inherently highly contented data structures has been to relax data structure semantics [1]. The most common type of relaxation is the *out-of-order* one, where a set of operations is allowed to deviate from the sequential linearization order. Relaxed queues often relax the ordering of their dequeue operations and use the terms *rank error* and *delay* to quantify the relaxation errors. Consider the dequeue of item x at time t in a FIFO queue. Then the *rank error* becomes the number of items older than x in the queue at t, and the *delay* the number of items enqueued after the enqueue of x that were dequeued before t.

Queues are commonly relaxed by being split up into disjoint sub-queues [9]–[11]. Here, operations linearize by operating on one of these sub-queues, and the main algorithmic difference between the queues is what load balancing scheme they use. A large part of the research is focused on load balancing that guarantees deterministic worst-case rank error and delay bounds [10], [12], [13], which leads to designs that are often quite easy to work with and extend [14]. On the other hand, several other designs leverage the power of randomization, often enabling them to achieve better trade-offs between efficiency and observed relaxation [11], [15], [16], at the cost of losing their deterministic worst-case relaxation bounds.

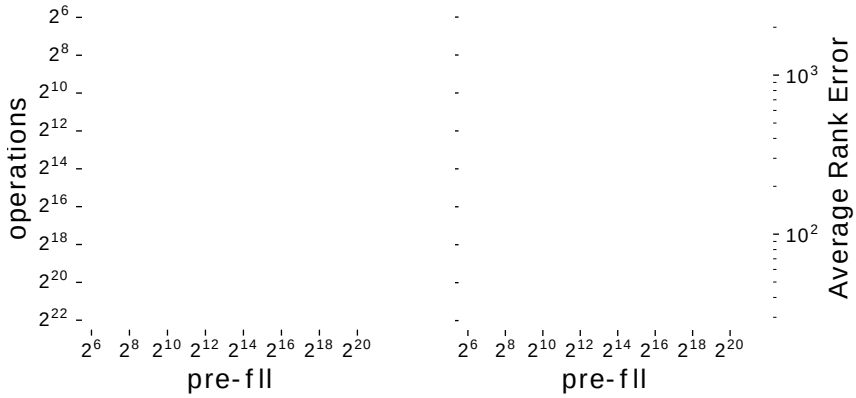
A prevalent technique in randomized relaxation [11], [15] has been the use of the d-choice load balancer, first analyzed by Azar et al. [17]. Consider trying to balance the load of m balls over a set of n bins. If one inserts each new ball into the least loaded bin out of a random choice of d (where $d \geq 2$), it is proven that the heaviest bin will diverge at most $\mathcal{O}(\log \log n / \log d)$ from the average load *with high probability* (w.h.p.) in n[1] [17], [18]. This has been used to relax queues by splitting them up into n sub-queues and operating on the preferred sub-queue out of the d randomly selected ones for each operation [9], [13].

Applying this d-choice load balancer to queues requires a preference between sampled sub-queues. The MultiQueue [9], [15], [16] relaxed priority queue has proven successful by choosing the sub-queue with the highest priority item for deletes. For relaxed FIFO queues, the d-RA queue [11], [13] demonstrated efficiency by enqueuing into the shortest sampled sub-queue and dequeueing from the longest. However, basing the d-choice on the length of the sub-queues comes at the expense of making the rank error dependent on the total queue size. This is demonstrated in Figure 1(a), which visualizes the average rank error of the 2-RA with 64 sub-queues over 100 runs in a sequential setting. It shows

[1]Throughout the paper, when using w.h.p., unless specifying otherwise, we mean with respect to n. This means that a probability is bounded above by $1 - n^{-c}$ for some $c > 0$.

empirically that the rank errors in the d-RA can degrade with increasing queue size. The prior analysis on the d-RA only bounds the difference in sub-queue size as a function of the number of sub-queues, missing the connection to the rank errors, and their experiments seem to only run on small queues where the relaxation looks good.

Another issue with contemporary relaxed FIFO queues is that essentially all of them [10], [11], [13] build upon suboptimal sub-queues such as the Michael-Scott (MS) queue [19]. The main reason for this is that the MS queue provides linearizable exact enqueue and dequeue counts, a crucial element of the respective relaxed designs. More scalable FIFO queue designs, such as the LCRQ [2] lack this property, making them more intricate to utilize. By using modern sub-queues based on arrays and fetch-and-add (FAA) instead of compare-and-swap (CAS), one can get better performance at similar relaxation errors.



(a) Length-based (b) Operation-based

Figure 1: FIFO rank errors when using 2-choice load balancing over length and operations respectively, using 64 sub-queues.

In this paper, we introduce a d-choice sub-queue load balacer that gives both good throughput and low relaxation. For enqueues (dequeues) it samples d sub-queues at random and operates on the one with fewest total enqueues (dequeues) performed on it. By using results from balanced allocations for heavily loaded balls-into-bins [18] we bound the rank errors and delay of our load balancing scheme to $\mathcal{O}(n \log n)$ w.h.p. in a subset of executions. Figure 1 demonstrates the difference in rank errors when balancing using sub-queue lengths (Figure 1(a)) and operation counts (Figure 1(b)). It shows that the earlier length-based scheme, as used in the d-RA queue [11], [13], has errors increasing with the size of the queue, while our operation-based balancing maintains low rank errors invariantly of the `pre-fill` and number of `operations`.

Furthermore, we show how to incorporate fast queue designs, such as the LCRQ by Morrison and Afek [2] and the wait-free queue by Yang and Mellor-Crummey [3], as sub-queues for such d-choice load balanced queues. We describe the interface a sub-queue needs to implement, which mainly entails

noticing if the queue has changed between two points in time, for ensuring empty-linearizability and lock-freedom. We then describe how to implement this for a set of such fast queues by using internal counters that are incremented and used every operation. We also show how any queue can be used as sub-queue when the requirement for empty-linearizability is removed, as it is not standard in relaxed queues [10], [15].

Contributions We introduce the randomly relaxed FIFO d-Choice Balanced Operations (d-CBO) queue that builds upon the theory of balanced allocations. The operation-based load balancer is shown analytically to give rank errors and delay of $\mathcal{O}(n \log n)$ w.h.p. in a collection of executions. Our experimental evaluation finds that it gives orders of magnitude smaller errors in concurrent executions than previous designs, and that they stay within our analytical high probability bound. The queue further improves on earlier designs by incorporating state-of-the-art sub-queues for increased performance. The sub-queue interface is generic and leaves room for future queue designs, and our experimental evaluation shows that the fast sub-queues can give more than 10 times better throughput compared to simpler designs, when using a low number of sub-queues. Our breadth-first-search (BFS) macro-benchmark demonstrates that the d-RA has consistent and significantly faster running times than earlier strict and relaxed concurrent FIFO queues.

Outline We present the novel algorithm in Section 3. Based on this algorithm description, we analyze the expected errors from its load balancing scheme in Section 4, where we also prove its linearizability and lock-freedom. We describe how to integrate our algorithm with state-of-the-art sub-queues in Section 5. Our experimental evaluation in Section 6 evaluates throughput and rank errors in micro-benchmarks, as well as execution times in a concurrent BFS benchmark. Finally, the paper concludes in Section 7.

2 Related Work

The Michael-Scott (MS) queue [19] was introduced in 1996 and has since become a foundational design in lock-free data structures due to its simplicity and extensibility, for example shown in its adaptation in the Harris linked list [20] and many relaxed queues [10], [11], [13]. However, the design faces scalability issues due to two highly contended CAS retry loops at the head and tail. In the following years, there were several lock-free queue designs [21]–[24] that improved on the MS queue in different scenarios.

Morrison and Afek proposed the LCRQ in 2013 [2] which since has remained among the most scalable and flexible lock-free queue designs. In utilizes a linked list of circular ring buffers with epoch counters inside cells for linearizability. The key is that they use FAA on head and tail counters in each buffer to assign operations to cells. This leads to dequeues occasionally reaching the cell before the corresponding enqueue, poisoning the cell and forcing the future enqueue to retry. However, the performance increase from using FAA over CAS far dwarfs these retry costs in most cases.

There has also been work on wait-free queues, but until recently, the fastest wait-free queue was orders of magnitude slower than that of lock-free queues [25]. However, Kogan and Petrank [26] presented the *fast-path-slow-path* methodology in 2012, that was later used by Yang and Mellor-Crumney in 2016 to create an efficient wait-free queue [3] (WFQ). Similarly to the LCRQ, the WFQ is a linked list of bounded buffers, where FAA is used to assign operations to cells. The difference is that after a certain set of failed attemps at linearization, a slow operation gets help from others until it succeeds. However, in the fast path, which is most cases, it performs very similarly to the LCRQ. A similar design to the WFQ, but without the wait-free helping, called the FAAArrayQueue [27] was described by Ramalhete in 2016.

Another area with a lot of interest over the years is balanced allocations. In the balls-into-bins problem, we are given m balls to insert into n bins with the goal of minimizing the maximal bin load, which is the same as minimizing the $gap = \underset{i=1...n}{max} \{bin_i - \frac{m}{n}\}$. The problem assumes that one cannot always inspect all bins, and is divided into the *lightly loaded* case where $n = m$ and the *heavily loaded* case where m is arbitrarily large. In the simplest *one-choice* process where balls are allocated uniformly at random, the gap w.h.p. is $\mathcal{O}(\frac{\log n}{\log \log n})$ [28] in the lightly loaded case ($n = m$), and w.h.p. $\mathcal{O}(\sqrt{\frac{m}{n} \log n})$ [29] in the heavily loaded case.

Simply extending the strategy to a *d-choice* process, where each ball is allocated to the least loaded of d ($d \geq 2$) random bins, proves a huge improvement over the one-choice process. This was first analyzed by Azar et al. [17] who showed that the process w.h.p. only has a gap of $\log_d \log n + \mathcal{O}(1)$ in the lightly loaded case. Berenbrink et al. [18] extended the analysis to the heavily loaded case, for which they found the same bound as in the light setting $\log_d \log n + \mathcal{O}(1)$. This heavily loaded bound was later also proved by Talwar and Wieder [30], but using a simpler proof technique that was generalizable for the case of weighted balls, and simpler to understand.

The d-choice process has proven useful within the design of relaxed priority queues. Perhaps the most successful design yet is the MultiQueue [9], [15] which is a relaxed priority queue which inserts items into random partial priority queues, and removes items from the the partial with the highest priority item out of a random choice of d (usually $d = 2$). It has proven to be effective in practice [15], for example as a priority scheduler [16]. Its relaxation has also been analyzed using techniques from balls-into-bins [16], [31], [32].

Furthermore, the d-choice has been applied to relaxed FIFO queues with the d-RA queue [13], where items are enqueued into the shortest (and dequeued from the longest) of d sampled queues. Although showing comparatively good performance to other designs, the relaxation error of the queue is not properly analyzed and it is based on the sub-optimal MS queue [19] for sub-queues, depending on the exact enqueue and dequeue counters this provides.

There are also relaxed FIFO queues that guarantee worst-case bounds on the rank error. The k-segment queue [13] is designed as a MS queue [19] where every node is an array with space for k items. The tail (head) is only advanced when completely filled (emptied), which gives a maximal rank error

of $k-1$. Similarly, the 2D queue [10] superimposes a window with a *width* and *depth* over its partial queues at its head and tail, that define operable areas for the dequeues and enqueues respectively. This leads to a rank error bound of $(width-1)\cdot depth$. Although the 2D queue improves on the k-segment queue, they both share the issue of threads searching for valid sub-queues the final operations before the segment/window advances.

The b-round-robin queue [11] is another relaxed FIFO queue. In it, every thread is attached to one of b counters for enqueues and dequeues respectively, and for every operation is uses fetch-and-add on the counter, modulo the number of partial queues, to get which queue to operate on. Letting the relaxation of a queue become unbounded, but requiring empty-linearizability, corresponds to concurrent pools. Sundell et al. [33] presented an efficient pool for workloads where each thread both insert and removes items, and Gidron et al. [34] presented an efficient design for producer-consumer workloads.

The potential of relaxing the sequential semantics of concurrent data structures was first highlighted by Shavit [1] and first formalized by the concept of *quantitative relaxation* by Henzinger et al. [12]. However, their definition only covers designs with worst-case bounds. Alistarh et al. [32] defined *distributional linearizability* as a correctness condition for concurrent designs with randomized relaxation, such as the MultiQueue [9]. The concept of *elastic relaxation* has also been recently introduced by von Geijer et al. [14] as a correctness condition for designs in which the relaxation bounds can be changed during run-time to match dynamic executions.

3 Algorithmic Description

The pseudocode of our d-Choice Balanced Operations (d-CBO) queue is shown in Algorithm 1. It starts by presenting the generic functions required for the sub-queue to implement (line 1.1). As for all FIFO queues, the sub-queues must implement *Enqueue* and *Dequeue*. Furhermore, the sub-queue must implement *EnqCount* and *DeqCount* which should return how many items have been enqueued and dequeued from the sub-queue. Finally, *EnqVersion* returns an integer that is unique for each current tail item.

For linearizability, the sub-queues must be linearizable, and the *EnqVersion* must change during the linearization of each enqueue, never returning the same count for two different tail items. The *EnqCount* and *DeqCount* should be exact during sequential executions, but are allowed to deviate in concurrent executions as long as the deviation is on the same order of magnitude for all sub-queues. To guarantee lock-freedom, the sub-queue must be lock-free, and *EnqVersion* is only allowed to change a finite number of times before an enqueue linearizes. Most lock-free queues can implement this interface with small additions, and we show how to implement it for a few modern sub-queues in Section 5.

Moving on to the d-CBO methods, its *Enqueue* (line 1.7) samples d sub-queues at random (with replacement), selects the optimal sampled sub-queue as the one with lowest *EnqCount*, and linearizes by enqueueing into it. As outlined

Algorithm 1: d-Choice Balanced Operations Queue

```
1.1  generic SubQueue
1.2      method SubQueue.Enqueue(item)
1.3      method SubQueue.Dequeue() → Item | NULL
1.4      method SubQueue.EnqCount() → uint
1.5      method SubQueue.DeqCount() → uint
1.6      method SubQueue.EnqVersion() → uint
1.7  function Enqueue(sub_queues, d, item)
1.8      samples ← randomly sample (q_1, ..., q_d) where q_i ∈ sub_queues
1.9      optimal ← argmin_{q∈samples} EnqCount(q)
1.10     optimal.Enqueue(item)
1.11 function Dequeue(sub_queues, d)
1.12     samples ← randomly sample (q_1, ..., q_d) where q_i ∈ sub_queues
1.13     optimal ← argmin_{q∈samples} DeqCount(q)
1.14     dequeued ← optimal.Dequeue()
1.15     if dequeued ≠ NULL then
1.16         return dequeued
1.17     else
1.18         return DoubleCollect(sub_queues)
1.19 function DoubleCollect(sub_queues)
1.20     versions ← [0, ..., 0]
1.21     repeat
1.22         for q_i ∈ sub_queues do
1.23             versions[i] ← EnqVersion(q_i)
1.24             dequeued ← q_i.Dequeue()
1.25             if dequeued ≠ NULL then
1.26                 return dequeued
1.27         if ∀q_i ∈ sub_queues, EnqVersion(q_i) = versions[i] then
1.28             return NULL
```

above, this will also change the *EnqVersion* and increment the *EnqCount* of the sub-queue.

The d-CBO *Dequeue* (line 1.11) similarly tries to dequeue from the sub-queue with the lowest *DeqCount* out of the d sampled ones. However, if the sub-queue *Dequeue* returns `NULL`, the d-CBO *Dequeue* cannot immediately return `NULL`. Instead it must enter the *DoubleCollect* function to guarantee empty-linearizability.

The *DoubleCollect* function (line 1.18) uses the double-collect [35] mechanism to try to get an atomic snapshot where all sub-queues are empty. It does this by first iterating over the sub-queues (line 1.22), recording their *EnqVersion* and trying to dequeue an item from them. To avoid sub-queue bias, this iteration should start at a random index. If it at any point succeeds in dequeuing an item, it successfully returns it. However, if it found all sub-queues empty in the first iteration, it then does a second iteration (line 1.27) where it checks if any *EnqVersion* has changed, signalling that an item might have been enqueued, in which case it restarts. If no *EnqVersion* has changed, no item can have been enqueued between the end of the first iteration and the start of the second, at which point all sub-queues were empty and it is safe to return *NULL*.

4 Analysis

4.1 Probabilistic Relaxation Guarantees

This section analyzes the relaxation errors in the d-CBO queue. We limit the analysis to the set of executions where operations are non-overlapping (or equivalently occur atomically), and where all enqueues are done before dequeues. These simplifications are the norm in related literature [15], [31], [36], and simplify the analysis while retaining the core behavior of the data structure.

On a high level, our analysis starts by formalizing the sub-queue selection as an enqueue and dequeue multivariate process over the *EnqCount* and *DeqCount*, similarly to the balls-into-bins process. The rank error and delay are then formulated in terms of those two processes. Then, Lemma 4.1 gives an intermediate result about the d-choice balls-into-bins, bounding the gap between the least and most loaded bin, utilizing earlier results by Peres et al. [37]. The balls-into-bins process is then connected to our enqueue and dequeue processes in Lemmas 4.2 and 4.3, bounding a key term for d-CBO relaxation errors. These two lemmas naturally prove Theorem 4.4 and Theorem 4.5, which bound the rank error and delay to $\mathcal{O}(n \log n)$ w.h.p.

Let $\mathcal{E}_i(t)$ denote the *EnqCount* of sub-queue i at time t. The enqueue operation, as described on line 1.7, inserts its item into the sub-queue with the smallest EnqCount among the d sampled queues, in the process incrementing that *EnqCount* by 1. In terms of $\mathcal{E}$, an enqueue samples d $\mathcal{E}_i(t)$ and increments the smallest of them.

Similarly, let $\mathcal{D}_i(t)$ denote the *DeqCount* of sub-queue i at time t. A dequeue operation as described on line 1.11 tries to dequeue an item from the sub-queue

with the smallest *DeqCount* out of the d sampled sub-queues, incrementing its *DeqCount*. However, if the sampled sub-queue with smallest *DeqCount* is empty, the *DoubleCollect* mechanism at line 1.18 is used to dequeue an item from another sub-queue or return *NULL* if the whole queue is empty. This can be seen as sampling d counts in $\mathcal{D}(t)$ and incrementing the smallest such $\mathcal{D}_j(t)$ if $\mathcal{D}_j(t) < \mathcal{E}_j(t)$. But if $\mathcal{D}_j(t) \geq \mathcal{E}_j(t)$, another i – where $\mathcal{D}_i(t) < \mathcal{E}_i(t)$ – is chosen (this analysis ignores which i is selected) and $\mathcal{D}_i(t)$ incremented. If no such $\mathcal{D}_i(t)$ exists, the dequeue has no effect and returns *NULL*.

Definition 1. *The rank error of the dequeue of item x, which was enqueued at time t_e and dequeued at t_d, is the number of other items that were enqueued before t_e and are still in the queue at time t_d. In terms of $\mathcal{D}$ and $\mathcal{E}$, this can be expressed as*

$$\begin{aligned} \textit{rank error} &= \sum_{i=1}^{n} \textit{max}(0, \mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)) \\ &\leq \sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|. \end{aligned} \tag{1.1}$$

Definition 2. *The delay of item x, which was enqueued at time t_e and dequeued at time t_d (where $t_d = \infty$ if x is never dequeued), is defined as the number of items enqueued after t_e but dequeued before t_d. This is formulized in terms of $\mathcal{E}$ and $\mathcal{D}$ as*

$$\begin{aligned} \textit{delay} &= \sum_{i=1}^{n} \textit{max}(0, \mathcal{D}_i(t_d) - \mathcal{E}_i(t_e)) \\ &\leq \sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|. \end{aligned} \tag{1.2}$$

Definition 3. *The gap of an n-dimensional vector $\mathcal{X}$, such as $\mathcal{E}$ or $\mathcal{D}$, is defined as the difference between its largest and smallest value.*

$$\textit{gap}(\mathcal{X}) = \textit{max}(\mathcal{X}) - \textit{min}(\mathcal{X})$$

It is worth noting that *gap* is normally defined as the difference between the max and the mean, $\max(\mathcal{X}) - \overline{\mathcal{X}}$, for which a bound of $\mathcal{O}(\frac{\log \log n}{\log d})$ has been proven [18], [30]. However, our coming analysis requires knowledge of the difference between the max and min, hence the slightly unusual *gap* definition. In the following lemma, we bound this gap w.h.p. when $\mathcal{X}$ is a d-choice process, before connecting it to $\mathcal{E}$ and $\mathcal{D}$ in Lemma 4.2.

Lemma 4.1. *If $\mathcal{X}(t)$ is an n-dimensional greedy d-choice load vector, with $\mathcal{X}(0) = \mathbf{0}$, where at each time t, the smallest of d uniformly at random sampled entries – with ties broken arbitrarily – is incremented by 1, it holds that*

$$\textit{gap}(\mathcal{X}(t)) = \mathcal{O}(\log n).$$

Proof. First, letting $\mathcal{X} = \mathcal{X}(t)$, $G = gap(\mathcal{X})$, and using the Markov inequality, the probability of G being at least k, for $k > 0, a > 0$, is bounded as follows,

$$P(G \geq k) = P(e^{aG} \geq e^{ak}) \leq \frac{\mathbb{E}[e^{aG}]}{e^{ak}}. \tag{1.3}$$

Then, we utilize the elegant potential analysis by Peres, Talwar, and Wieder [30], [37] that, for some $a' > 0, b > 0$, gives the following tail bound:

$$\mathbb{E}\left[\sum_{i=1}^{n} \exp(a'(\mathcal{X}_i - \overline{\mathcal{X}})) + \exp(-a'(\mathcal{X}_i - \overline{\mathcal{X}}))\right] \leq bn.$$

This in turn gives an upper bound on the probability from Eq. 1.3,

$$\begin{aligned} bn &\geq \mathbb{E}\left[\sum_{i=1}^{n} \exp(a'(\mathcal{X}_i - \overline{\mathcal{X}})) + \exp(-a'(\mathcal{X}_i - \overline{\mathcal{X}}))\right] \\ &\geq \mathbb{E}[\exp\left(a'(\max_i(\mathcal{X}_i) - \overline{\mathcal{X}})\right) + \exp\left(a'(\overline{\mathcal{X}} - \min_i(\mathcal{X}_i))\right)] \\ &\geq \mathbb{E}[\exp\left(a(\max_i(\mathcal{X}_i) - \min_i(\mathcal{X}_i))\right)] = \mathbb{E}[e^{aG}] \\ \iff bn &\geq \mathbb{E}[e^{aG}], \end{aligned} \tag{1.4}$$

where we have used that $a' = 2a$. Thus, utilizing Eq. 1.4 in Eq. 1.3 gives

$$P(G \geq k) \leq \frac{\mathbb{E}[e^{aG}]}{e^{ak}} \leq \frac{bn}{e^{ak}},$$

and using $k = \frac{c+1}{a} \ln n$ for some $c > 0$ concludes the proof.

$$P\left(G \geq \frac{c+1}{a} \ln n\right) \leq bn^{-c}$$

□

Now, to connect this result with the $\mathcal{E}$ and $\mathcal{D}$ processes, we first note the duality between rank error and delay in Equations 1.1 and 1.2, where they simply swap the sign inside the *max* at each sub-queue. Bounding the sum $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|$ thus becomes the key to bounding both the rank error and delay, which is done in the following two lemmas, utilizing the *gap* bound from Lemma 4.1.

Lemma 4.2. *At any time t, it holds that $gap(\mathcal{E}(t)) = \mathcal{O}(\log n)$ and $gap(\mathcal{D}(t)) = \mathcal{O}(\log n)$.*

Proof. As the updates of $\mathcal{E}$ matches the d-choice balanced allocation (Greedy[d]) process for balls-into-bins [17], we can directly use the result from Lemma 4.1, which bounds $gap(\mathcal{E}) = \mathcal{O}(\log n)$ w.h.p.

On the other hand, $\mathcal{D}$ does not evolve exactly as a Greedy[d] process and deviates slightly in the presence of empty sub-queues. In the case where the Greedy[d] process tries to increment $\mathcal{D}_i$ corresponding to an empty sub-queue ($\mathcal{D}_i(t) = \mathcal{E}_i(t)$), the double-collect fallback is invoked and the dequeue instead

linearizes at another non-empty sub-queue, corresponding to incrementing another $\mathcal{D}_j$ where $\mathcal{D}_j(t) \neq \mathcal{E}_j(t)$. However, as our analytical setting limits all enqueues to be done before dequeues, if sub-queue i is the first to become empty at time t_1 ($\mathcal{D}_i(t_1) = \mathcal{E}_i(t_1)$), then for any $t' \geq t_1$, as $\mathcal{D}$ and $\mathcal{E}$ have both evolved as strict Greedy[d] processes up to this point, Lemma 4.1 gives that $\max_i(\mathcal{E}_i(t')) - \min_i(\mathcal{D}_i(t')) = \mathcal{O}(\log n)$ w.h.p. Since $\mathcal{E}_i(t) \geq \mathcal{D}_i(t)$ for any i, t, this gives that $gap(\mathcal{D}(t)) = \mathcal{O}(\log n)$.

□

Lemma 4.3. *For any item enqueued at time t_e and dequeued at time t_d, it holds that $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)| = \mathcal{O}(n \log n)$.*

Proof. Assume the item was dequeued from queue i. Then $\mathcal{D}_i(t_d) = \mathcal{E}_i(t_e)$ due to the total order in the sub-queue. Thus, Lemma 4.2 implies that $|\overline{\mathcal{E}(t_e)} - \overline{\mathcal{D}(t_d)}| = \mathcal{O}(\log n)$ w.h.p., which when summed over all sub-queues gives $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)| = \mathcal{O}(n \log n)$ w.h.p. □

With Lemmas 4.2 and 4.3 we now have all we need to create simple proofs bounding the rank error and the delay of the d-CBO queue.

Theorem 4.4. *The d-CBO queue w.h.p. has rank errors of $\mathcal{O}(n \log n)$.*

Proof. The rank error of a dequeue is in equation 1.1 bounded by $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|$, where t_e and t_d are the timestamps when the item was enqueued and dequeued respectively. That sum in turn is found to be $\mathcal{O}(n \log n)$ in Lemma 4.3. □

Theorem 4.5. *Items in the d-CBO queue w.h.p. have delays of $\mathcal{O}(n \log n)$.*

Proof. The delay of an item is in equation 1.2 bounded by $\sum_{i=1}^{n} |\mathcal{E}_i(t_e) - \mathcal{D}_i(t_d)|$, where t_e is the time the item was enqueued and t_d the time it was dequeued (or ∞ if never dequeued). Lemma 4.3 in turn finds that sum to be of order $\mathcal{O}(n \log n)$. □

4.2 Correctness Guarantees

In this part we prove the linearizability and lock-freedom of the d-CBO queue. The analysis utilizes the sub-queue interface, which guarantees that they should be linearizable, lock-free, that *EnqVersion* must be uniquely changed for every tail, and that at least one enqueue must linearize after the *EnqVersion* has changed a certain finite number of times.

Theorem 4.6. *The d-CBO queue is linearizable with respect to the semantics of a concurrent pool.*

Proof. An enqueue operation always linearizes with a linearizable enqueue to a sub-queue at line 1.10. Similarly, a non-empty dequeue also linearizes on a sub-queue at either line 1.16 or line 1.26.

Finally, dequeues returning *NULL* linearize with the double-collect [35] mechanism by repeating two iterations over the sub-queues in the *DoubleCollect*

function (line 1.18). The first (line 1.22) collects *EnqVersion* counts for every sub-queue, after which it ensures that the queue is empty (as it otherwise returns a dequeued item at line 1.26). The *DoubleCollect* call only returns *NULL* if the sub-queues have unchanged *EnqVersion* counts during the second iteration (line 1.27). As the version counts are guaranteed to be incremented during the linearization of an enqueue, this means that no enqueue has linearized between the end of the first iteration and the start of the second. Finally, as all sub-queues were found empty in the first iteration, the operation can linearize by returning *NULL* with the linearization point between the two iterations. □

Theorem 4.7. *The d-CBO queue is lock-free.*

Proof. Assume a period of time, with pending operations, where no enqueue linearizes for the d-CBO. We will prove that this period must be finite in length, as an enqueue or dequeue otherwise must complete. Inspecting enqueuers, we note that each d-CBO enqueue call (line 1.7) only performs d calls to *EnqCount* and one sub-queue *Enqueue* call before returning.

The d-CBO dequeue can invoke more sub-queue calls, but before it enters the *DoubleCollect* call it will similarly have performed d calls to *DeqCount* and one *Dequeue* at a sub-queue. After it enters *DoubleCollect* (line 1.18), it repeatedly (line 1.21) performs two loops containing up to n *Dequeue* and $2n$ *EnqVersion* calls. However, the *DoubleCollect* function will return *NULL* on line 1.28 in one of these iterations if no sub-queue *EnqVersion* has changed. The correctness condition on *EnqVersion* guarantees that it can only change a finite number of times before an enqueue linearizes.

Therefore, in this setting without enqueue progress, each d-CBO operation can only invoke a finite number of instructions and sub-queue calls before returning. As the sub-queues are lock-free, this duration without enqueue progress must either be finite in time or include dequeue linearizations, implying that the d-CBO queue is lock-free.

□

5 Implementations

5.1 Integrating Fast Sub-Queues

Section 3 outlined the interface sub-queues must implement, and this section describes how to uphold those guarantees for four selected lock-free queues. We start with the simple MS queue [19] which is commonly used in earlier relaxed FIFO designs [10], [11], [13], and then move on to state-of-the-art lock-free queues designed around FAA.

An optimization used for all sub-queues is that their dequeues first check if $EnqCount \leq DeqCount$, in which case they linearize by returning *NULL*. This reduces the number of increments of *DeqCount* and *EnqCount* not resulting in linearizations, improving the accuracy of the counters for the d-choice.

MS queue Naturally, the MS queue [19] supports lock-free linearizable enqueue and dequeue methods. Furthermore, by using counted head and tail

pointers[2], as in the original design [19], we can implement exact *EnqCount* and *DeqCount* functions. Finally, as those counts are incremented during the linearization of operations, we can let *EnqVersion* = *EnqCount*. Note also that the *EnqCount* is *only* updated during the linearization of an enqueue.

FAAArrayQueue The FAAArrayQueue [27] is an efficient, yet very simple, lock-free queue designed around FAA. It uses a linked list of queue segments where each segment has enqueue and dequeue counts used to assign operations to cells within the segment. By instrumenting each segment with a `segment_number`, increasing by one for each new segment, we can approximate the *EnqCount* as `tail.segment_number * SEGMENT_SIZE + min(tail.enq_count, SEGMENT_SIZE)`, where `tail` is the newest queue segment. Similarly, the *DeqCount* can be approximated by `head.segment_number * SEGMENT_SIZE + min(head.deq_count, SEGMENT_SIZE)`, where `head` is the oldest nonempty queue segment.

By limiting `enq_count` and `deq_count` from exceeding `SEGMENT_SIZE`, we ensure that each *EnqCount* corresponds to a unique tail item. Therefore, we can use *EnqCount* as *EnqVersion*. Furthermore, the queue is operation-wise lock-free as an enqueue will succeed system-wide every `SEGMENT_SIZE` attempts when enqueueing a new segment. Therefore it fulfills the requirement of one enqueue linearizing for some finite number of changes to *EnqVersion*. The *EnqCount* and *DeqCount* are not exact, as cells can be poisoned when the queue is close to empty, but the balanced operations spread these errors indiscriminately across the sub-queues.

WFQ The wait-free queue (WFQ) [3] is similar to the FAAArrayQueue, with the addition that threads can help each other achieve wait-freedom. The WFQ has one central enqueue count and dequeue count for assigning operations to cells across segments. These can be used direcly as *EnqCount* and *DeqCount*, with very similar behavior as the FAAArrayQueue. By letting *EnqVersion* = *EnqCount*, the same argument as for the FAAArrayQueue guarantees a unique tail for every *EnqVersion*. The wait-freedom, together with the fact that the *EnqCount* is only incremented by enqueues, ensures that for some finite number of changes to *EnqVersion*, at least one enqueue must have linearized.

LCRQ The LCRQ [2] is similar to the FAAArrayQueue, with the non-trivial optimization that segments are cyclic. This cyclic nature can improve performance in some cases, but also makes it more difficult to integrate it in *d*-CBO. Firstly, as the maximum size of each segment is unknown – as opposed to the FAAArrayQueue – we instrument each segment with a `starting_count`, which initialized to the `enq_count` of the previous segment. The *EnqCount* is then estimated as `tail.starting_count + tail.enq_count`, and *DeqCount* as `head.starting_count + head.deq_count`, where `tail` and `head` are the

[2]This is a necessary slight overhead, as the counted pointers are not required inside the MS queue when using memory management such as hazard pointers [38] or epoch-based reclamation [39]. Such counts are not added for the other sub-queues, as one then can use internal pre-existing counters.

newest and oldest non-empty segments respectively.

A problem with the `starting_count` is that the `enq_count` of the previous segment can be incremented after the initialization of the new `starting_count`. This is a potential source of error when choosing the optimal sub-queue and also leads to the same *EnqCount* being reported for different tail items. Therefore, *EnqCount* cannot be used directly as *EnqVersion* and we instead let *EnqVersion* be the bitwise concatenation of `tail.enq_count` and `tail.starting_count`.

To fulfill the requirement that *EnqVersion* can only change a finite number of times before an enqueue linearizes, the optional *fixState* mechanism must be disabled. This is an optimization enabling dequeue operations to increase `enq_count`, which can cause dead-locks in the double-collect of an empty d-CBO. It is clear that each *EnqVersion* corresponds to a unique tail item, only being changed during the linearization of an enqueue. As the base LCRQ is lock-free, it now implements the sub-queue interface.

Memory Overhead The memory usage of the d-CBO depends on the sub-queues used. Array-based queues such as the LCRQ, WFQ, and FAAArrayQueue incur a memory overhead due to the arrays not always being filled. This is compounded when using them as sub-queues, as the overhead is multiplied by the number of sub-queues used. In the worst case, each sub-queue has empty *head* and *tail* array segments, translating to `NBR_SUBQUEUES * 2 * SEGMENT_SIZE` empty cells, which bounds the d-CBO's memory overhead (disregarding potential overhead in used cell), as there is no replication of items. In memory-constrained environments, one should consider using smaller array segments, when using several sub-queues, compared to the array segments used in the original strict queues.

5.2 Relaxing Empty-Linearizability

As shown above, especially for the LCRQ, integrating new sub-queues for the d-CBO requires some care and understanding of the sub-queue properties. Even then, the *EnqCount* and *DeqCount* are not guaranteed to be accurate. Here we introduce the *Simplified d-Choice Balanced Operations queue* (Simple d-CBO), which relaxes the empty-linearity guarantee of the d-CBO in favor of a simplified design that can be used for any sub-queue, inheriting its properties.

Replacing the double-collect with a single-collect, where a dequeue returns *NULL* iff it has dequeued *NULL* from all sub-queues, removes the need for *EnqVersion* by sacrificing the empty-linearizability. For problems such as graph-traversals, where threads iteratively both dequeue and enqueue from the queue and can suspend after dequeueing *NULL*, the queue must be empty when all threads have dequeued *NULL* and suspended. This leads to the same termination conditions as if the queue was empty-linearizable.

Once the *EnqVersion* is made redundant, the d-CBO can be made completely generic over its sub-queue by also removing the need for sub-queue *EnqCount* and *DeqCount* implementations. Instead, one can create a wrapper for each sub-queue, with additional *EnqCount* and *DeqCount* fields that are updated with FAA after enqueues and non-empty dequeues. This sacrifices a bit of

performance due to added contention at these counters, but can cause the counters to be more exact.

The relaxation analysis in Section 4.1 trivially still holds for this Simple d-CBO, and the only relaxation difference is that we allow relaxed empty returns. Relaxed empty returns are also part of the *quantitative relaxation* [12] and *distributed linearizability* specifications of out-of-order relaxations, when such empty returns are within the error bound. Furthermore, the Simple d-CBO inherits lock-free and linearizable properties directly from its sub-queue.

6 Experimental Results

We have implemented the d-CBO [40], using the sub-queues presented in the last section, in a C benchmark suite for relaxed data structures built on top of the ASCYLIB suite [41]. To evaluate our implementations, we first present a variety of synthetic benchmarks to evaluate the scalability of throughput and rank errors. Secondly, we have implemented a concurrent BFS algorithms that accommodates relaxed queues, which we use to compare the d-CBO against state-of-the-art relaxed and strict queues in a more realistic setting. All experiments use $d = 2$, as that has been shown to be a good choice in the literature [9], [11], [15]–[17], and was also verified in our experiments.

When comparing against state-of-the-art, we have selected the FAAArrayQueue[3] [27], WFQ[4] [3], and LCRQ[4] [2] as representatives of strict FIFO queues. For relaxed queues, we have selected the 2D queue[5] [10], [14], and the d-RA queue[3] [11], [13], as these are the best bounded and randomized designs we know of in the literature. The 2D queue always uses `depth=16` in our benchmarks, as it showed good trade-offs between performance and relaxation errors. All these implementations have been incorporated into our benchmarking suite [40], using SSMEM [41] for consistent and fast epoch-based memory management.

System Description All experiments run on an AMD EPYC 9754 running at 2.25 GHz with 128 cores using two-way SMT, 256 MB L3 cache, and 755 GB RAM. The machine runs OpenSUSE Tumbleweed with the Linux 6.9.9 kernel. All experiments are written in C, using pthreads for concurrency and compiling with gcc 13.3.0 at optimization level O3. Software threads are pinned to hardware threads in a round-robin fashion between core clusters, starting to use SMT after 128 threads.

6.1 Benchmarking Scalability

To evaluate the performance of the d-CBO, we here evaluate its throughput and average rank error. Each data point shows the average and standard deviation of ten runs, each running for half a second. We mainly show results

[3]Implemented ourselves.
[4]Code from https://github.com/chaoran/fast-wait-free-queue [3].
[5]Code from the relaxed 2D benchmarking suite [42].

of using 128 sub-queues, as that empirically gave good results in the BFS benchmark, but also show sub-queue scalability in Figure 4. Furthermore, most benchmarks pre-fill the queues with 10^6 items to avoid empty returns for dequeues, which can cause deceivingly low relaxation errors or high throughput. Each experiment is run in two settings:

- **Random Enqueue/Dequeue**: Here, each thread repeatedly flips a fair coin and enqueues or dequeues an item based on the outcome.
- **Producer-Consumer**: Here half the threads are producers, repeatedly enqueueing items, and the other half consumers, repeatedly dequeueing items.

We measure rank errors with similar methodology as previous works [11], [15] by timestamping each operation and afterwards using the timestamps to re-create a sequential history. The rank errors can then be determined from the sequential history. This timestamping incurs some overhead, and relaxation is therefore measured in separate executions from the throughput measurements.

Efficient Sub-Queue Designs We begin by evaluating the performance benefits of our design's ability to integrate the most effective queue designs. Figure 2 compares the scalability of our new d-CBO designs, including the simplified implementations. Unsurprisingly, the MS queue has the lowest throughput scalability, while the other sub-queues scale relatively similarly. The simplified versions are a bit slower than their standard versions, due to the extra work of updating the external counters. The simplified designs also demonstrate worse average rank error, especially at higher levels of concurrency. This is due to the external counters being updated *after* the sub-queue operations linearize, leading to slightly stale information for the d-choice sub-queue selection. The queues show very similar scalability in both the *Random Enqueue/Dequeue* and *Producer-Consumer* settings.

Load Balancing Schemes To evaluate our new load balancing scheme, we have implemented the d-Choice Balanced-Lenghts (d-CBL) queue, which uses the previous d-RA load balancer [13] to select sub-queues based on their length. As shown in Figure 3, d-RA often leads to slightly lower throughput, as sampling must read both the enqueue and dequeue counts for each sub-queue. However, the standout differentiator is rank errors where the operation-based balancing is an order of magnitude better.

Scalability with Sub-Queues Figure 4 shows how the d-CBO scales with increasing number of sub-queues. The MS queue is significantly slower – though its more accurate counters result in slightly better rank errors – at lower widths where sub-queue contention is higher. The figure includes a line of $width + a$, where a is a constant, which suggest that the rank errors scale as $\mathcal{O}(n)$ in expectation. This also correlates well with our analytical bound of $\mathcal{O}(n \log n)$ w.h.p., which essentially represents a worst-case scenario.

State-of-the-art Experiments comparing the d-CBO with state-of-the-art

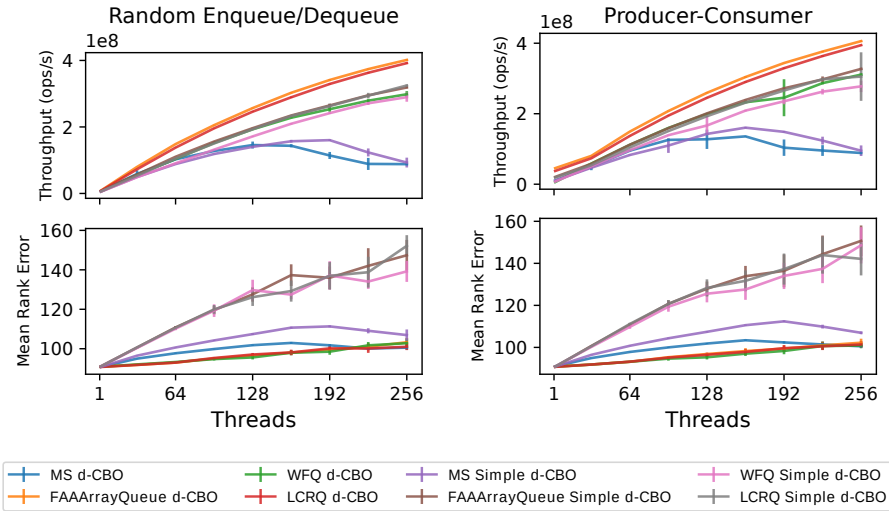


Figure 2: Evaluating the different d-CBO designs, using 128 sub-queues and 10^6 pre-fill.

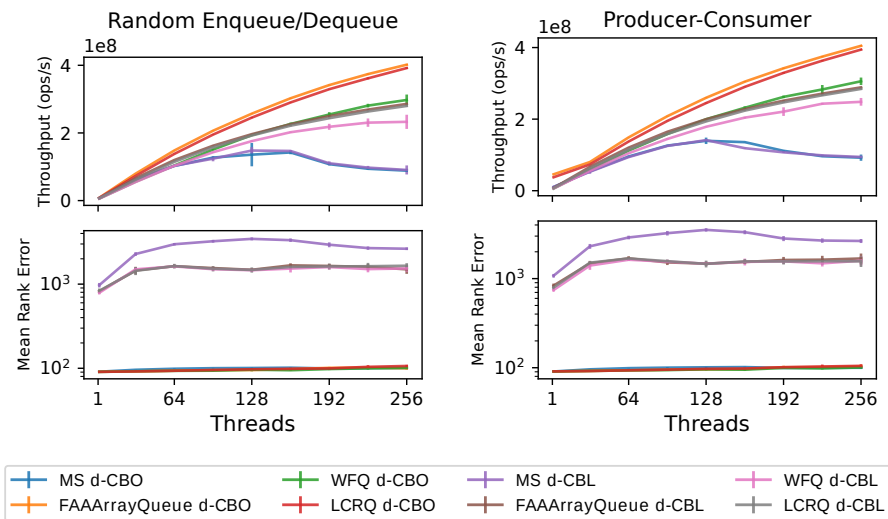


Figure 3: Comparing our operation-based load balancing with the earlier length-based balancing scheme, using 128 sub-queues and 10^6 pre-fill.

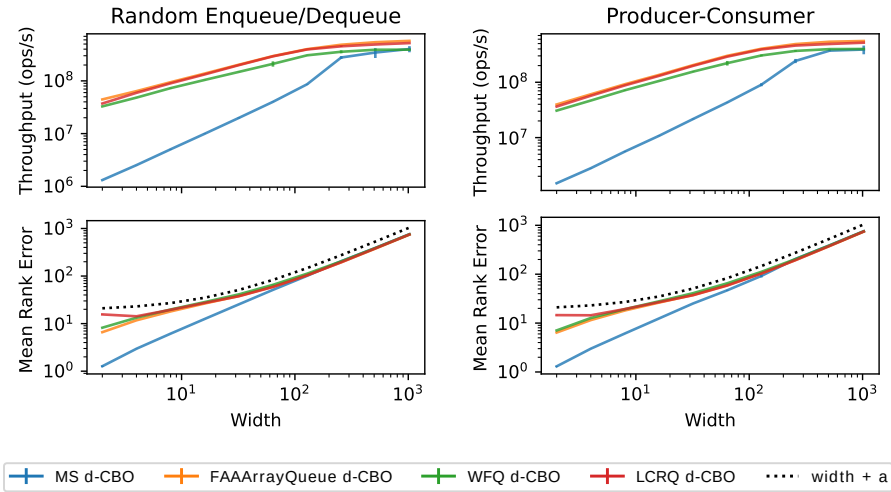
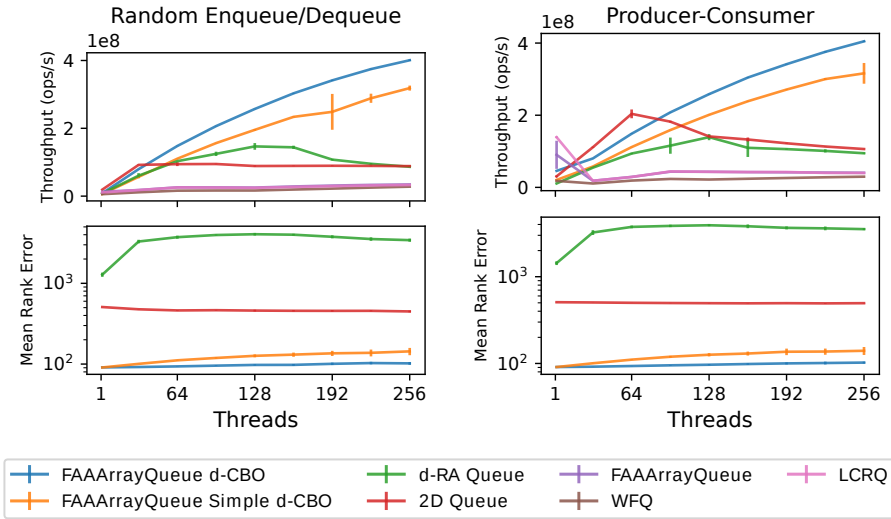


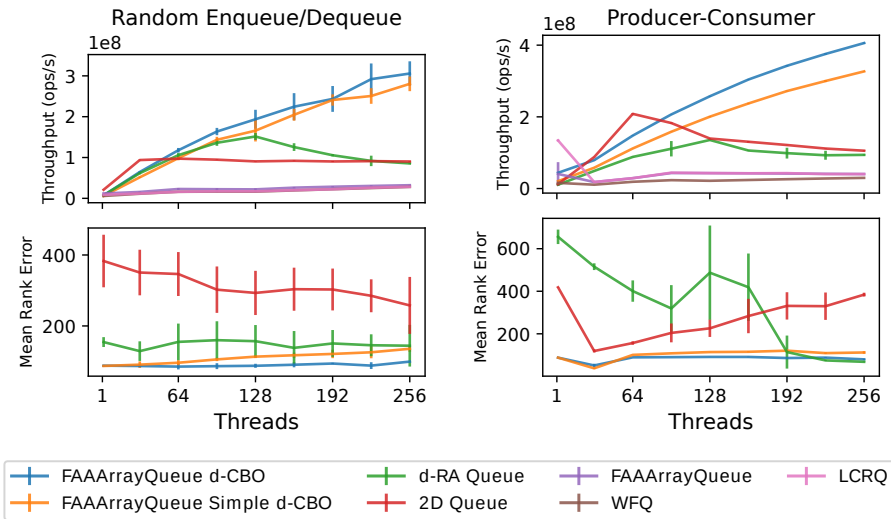
Figure 4: Evaluating how the d-CBO scales with the number of sub-queues, using 256 threads and 10^6 pre-fill. The line of $width + a$, where a is some constant, is included for comparing with rank error scalability.

designs [2], [3], [10], [13], [27] are shown in Figure 5, presenting results for both large and small pre-fill. For clarity, we only included the FAAArrayQueue d-CBO as it performed the best in the in previous comparisons. Both the simplified and standard d-CBO significantly outscale earlier designs in throughput while also maintaining a far lower rank error average than the 2D and d-RA designs. The d-RA manages to achieve relatively low rank errors in the presence of low pre-fill, but its heuristic is bad for large queues as also seen in Figure 3 and Figure 1. The reason for the high errors in larger queues is that the enqueue/dequeue counts for two sub-queues can drift relative to each other over time, as long as the difference between the counts for all queues stays the same.

Generally, the experiments with low pre-fill are more volatile, as *NULL* dequeue returns can significantly alter the performance. The number of d-CBO operations entering the double-collect in the benchmarks varied by sub-queue due to differences in dequeue and enqueue latencies. The MS queue was the only one with faster dequeues than enqueues, thus entering double-collect in around 20% of dequeues for the producer-consumer scenario. For the random enqueue/dequeue setting, we saw between four and ten percent of dequeues entering the double-collect with low pre-fill, and none at high pre-fill. In general, relaxed queues similar to the d-CBO are not as effective when the number of empty dequeues are significantly higher than this and you want low-latency empty dequeues, due to the double-collects. If one wants faster empty dequeues, the double-collect can be removed, or changed to a single-collect, at the cost of empty-linearizability.



(a) 10^6 pre-fill



(b) 10^3 pre-fill

Figure 5: Comparison against state-of-the-art, using 128 sub-queues for all relaxed queues, and `depth` = 16 for the 2D queue.

6.2 Concurrent BFS Benchmark

To evaluate the usability of the d-CBO, we utilize a concurrent BFS benchmark. In the BFS, as shown in Algorithm 2, threads iteratively, through a series of so-called relaxations (line 2.12), decrease the known fastest path to vertices. The algorithm continues until the queue is empty and all threads are idle, where the fastest path to each vertex is found, assuming unweighted edges.

Algorithm 2: Concurrent BFS Algorithm

```
2.1  graph ← ReadToCSR()
2.2  distances ← [∞, ..., ∞]
2.3  queue ← [source]
2.4  distances[source] ← 0
2.5  concurrently while One thread has work do
2.6      at ← queue.Dequeue()
2.7      if at ≠ NULL then
2.8          at_dist ← distances[at]
2.9          for neigh ∈ neighbors(at, graph) do
2.10             neigh_dist ← distances[neigh]
2.11             while neigh_dist < at_dist + 1 do
2.12                 if CAS(distances[neigh], neigh_dist, at_dist + 1) then
2.13                     queue.Enqueue(neigh)
2.14                     break
```

We define the *work* of an execution as the number of times the tentative distance to a node is changed (line 2.12). In the sequential setting, the work will equal the number of nodes (assuming the graph is strongly connected), as one always processes the wavefronts in order. However, interleavings and relaxation errors in concurrent executions can cause sub-optimal paths to be explored, leading to additional work.

For evaluation, we selected real-world graphs following previous research [16], [43] as shown in Table 1. We selected a set of road networks, one graph from numerical simulation, one from Delaunay triangulation, one from census data, and one research citation graph. They are all strongly connected and we arbitrarily select node 1 as the source. Table 1 also includes the average distance from the source, which is related to the diameter and heavily affects the BFS execution characteristic.

The results from running the BFS algorithm with our d-CBO and the selected state-of-the-art queues, is shown in Figure 6. Each data point shows the speedup and work increase compared to the sequential algorithm with a strict queue, averaged over 10 runs, including standard deviation.

The results vary significantly for different graphs, but the d-CBO consistently achives the highest speedup for all the graphs at high thread counts, with the FAAArrayQueue d-CBO often slightly outperforming the other FAA-based designs, as also shown in the synthetic benchmarks. The d-CBO often has to process more work than the sequential designs, but compensates with its

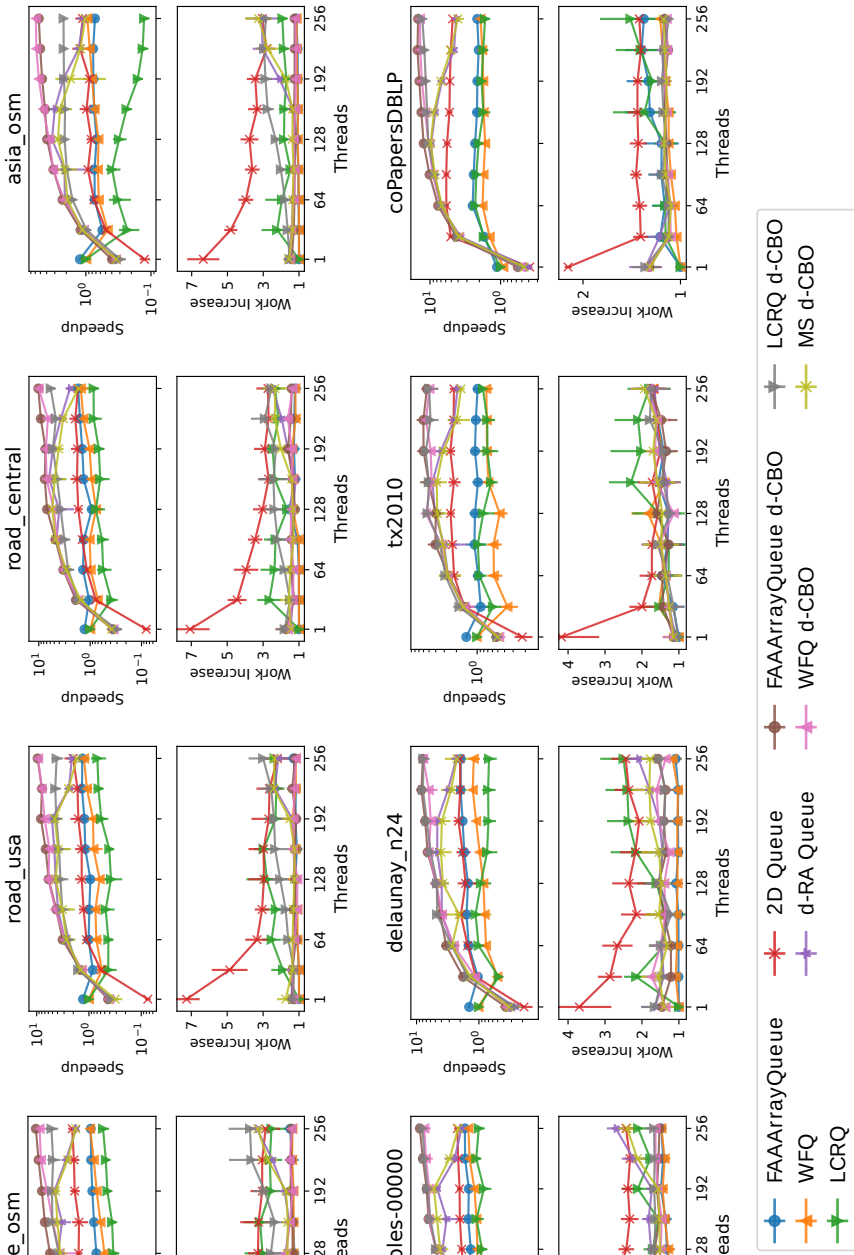


Figure 6: Comparing queues with a concurrent label-correcting BFS algorithm, using 128 sub-queues for all relaxed queues, and `depth=32` for the 2D queue.

Table 1: Graphs used in BFS benchmark. All available in the SuiteSparse Matrix Collection [44].

Graph	Nodes	Edges	Avg. Source Dist.
europe_osm	$51M$	$108M$	5028
road_usa	$24M$	$58M$	2762
road_central	$14M$	$34M$	1901
asia_osm	$11M$	$25M$	11619
hugebubbles-00000	$18M$	$55M$	3322
delaunay_n24	$17M$	$101M$	926
tx2010	$914T$	$4.4M$	123
coPapersDBLP	$540T$	$30M$	5.82

superior throughput. The 2D queue consistently has among the highest work increases, especially at low thread counts, but manages to scale rather well and for the most part outperforms the strict queues. The d-RA performs similar to the MS d-CBO, as the queue likely often is relatively small, leading to lesser rank errors as also shown in Figure 5.

Interestingly, the strict queues can have rather high work increases as well, purely due to concurrency interleavings. Especially in the citation graph, with its low diameter, several of the strict queues do more work than the relaxed queues. This shows that the drawback of relaxed semantics, namely the relaxed order, in some cases does not pose a significant difference to normal concurrent designs. This is especially noticeable for relaxed queues such as the d-CBO with relatively small relaxation errors.

7 Conclusion

We introduced the relaxed d-CBO queue, that significantly improves on state-of-the-art designs in throughput and relaxation errors simultaneously. This is achieved by providing a new algorithmic design that (i) is capable of interfacing against efficient sub-queues and (ii) introduces a new load balancing scheme for assigning operations to sub-queues based on operation counts. Besides the experimental results that depict its performance improvements, we prove that the rank errors and delay is $\mathcal{O}(n \log n)$ with high probablity in executions with non-overlapping operations where enqueues proceed dequeues. Our experimental evaluation supports this bound, but shows that the errors are closer to n in expectation. Finally, our concurrent BFS benchmark demonstrated the utility of relaxed queues, where d-CBO again consistently performed the best.

For future work, we would like to extend the theoretical analysis to encompass all concurrent executions. It would also be of great interest to give a theoretical bound on the expectation of the relaxation errors. Another interesting direction is to extend the algorithmic design to be able to elastically change its relaxation during runtime.

Bibliography

[1] N. Shavit, 'Data structures in the multicore age,' *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011. DOI: 10.1145/1897852.1897873.

[2] A. Morrison and Y. Afek, 'Fast concurrent queues for x86 processors,' in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Shenzhen, China: Association for Computing Machinery, 2013, pp. 103–112. DOI: 10.1145/2442516.2442527.

[3] C. Yang and J. Mellor-Crummey, 'A wait-free queue as fast as fetch-and-add,' in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, Spain: Association for Computing Machinery, 2016. DOI: 10.1145/2851141.2851168.

[4] D. Hendler, N. Shavit and L. Yerushalmi, 'A scalable lock-free stack algorithm,' *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 1–12, 2010. DOI: 10.1016/j.jpdc.2009.08.011.

[5] M. Dodds, A. Haas and C. M. Kirsch, 'A scalable, correct time-stamped stack,' in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, Mumbai, India: Association for Computing Machinery, 2015, pp. 233–246. DOI: 10.1145/2676726.2676963.

[6] J. Lindén and B. Jonsson, 'A skiplist-based concurrent priority queue with minimal memory contention,' in *Principles of Distributed Systems*, Cham: Springer International Publishing, 2013, pp. 206–220.

[7] A. Braginsky, N. Cohen and E. Petrank, 'Cbpq: High performance lock-free priority queue,' in *Euro-Par 2016: Parallel Processing*, Cham: Springer International Publishing, 2016, pp. 460–474.

[8] F. Ellen, D. Hendler and N. Shavit, 'On the inherent sequentiality of concurrent objects,' *SIAM Journal on Computing*, vol. 41, no. 3, pp. 519–536, 2012. DOI: 10.1137/08072646X.

[9] H. Rihani, P. Sanders and R. Dementiev, 'Multiqueues: Simple relaxed concurrent priority queues,' in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 80–82. DOI: 10.1145/2755573.2755616.

[10] A. Rukundo, A. Atalar and P. Tsigas, 'Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework,' in *33rd International Symposium on Distributed Computing (DISC 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 146, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 31:1–31:15. DOI: 10.4230/LIPIcs.DISC.2019.31.

[11] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch and A. Sezgin, 'Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation,' in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy: Association for Computing Machinery, 2013. DOI: 10.1145/2482767.2482789.

[12] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin and A. Sokolova, 'Quantitative relaxation of concurrent data structures,' *SIGPLAN Not.*, vol. 48, no. 1, pp. 317–328, 2013. DOI: 10.1145/2480359.2429109.

[13] C. M. Kirsch, H. Payer, H. Röck and A. Sokolova, 'Performance, scalability, and semantics of concurrent fifo queues,' in *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'12, Fukuoka, Japan: Springer-Verlag, 2012, pp. 273–287. DOI: 10.1007/978-3-642-33078-0_20.

[14] K. von Geijer and P. Tsigas, 'How to relax instantly: Elastic relaxation of concurrent data structures,' in *Euro-Par 2024: Parallel Processing*, Cham: Springer Nature Switzerland, 2024, pp. 119–133. DOI: https://doi.org/10.1007/978-3-031-69583-4_9.

[15] M. Williams, P. Sanders and R. Dementiev, 'Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues,' in *29th Annual European Symposium on Algorithms (ESA 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17. DOI: 10.4230/LIPIcs.ESA.2021.81.

[16] A. Postnikova, N. Koval, G. Nadiradze and D. Alistarh, 'Multi-queues can be state-of-the-art priority schedulers,' in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 353–367. DOI: 10.1145/3503221.3508432.

[17] Y. Azar, A. Z. Broder, A. R. Karlin and E. Upfal, 'Balanced Alocations,' *SIAM Journal on Computing*, vol. 29, no. 1, pp. 180–200, 1999, A preliminary version appeared in *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 593–602, Montreal, Quebec, Canada, May 23–25, 1994. ACM Press, New York, NY.

[18] P. Berenbrink, A. Czumaj, A. Steger and B. Vöcking, 'Balanced allocations: The heavily loaded case,' *SIAM Journal on Computing*, vol. 35, no. 6, pp. 1350–1385, 2006. DOI: 10.1137/S009753970444435X.

[19] M. M. Michael and M. L. Scott, 'Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,' in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275. DOI: 10.1145/248052.248106.

[20] T. L. Harris, 'A pragmatic implementation of non-blocking linked-lists,' in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01, Berlin, Heidelberg: Springer-Verlag, 2001, pp. 300–314.

[21] P. Tsigas and Y. Zhang, 'A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems,' in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01, Crete Island, Greece: Association for Computing Machinery, 2001, pp. 134–143. DOI: 10.1145/378580.378611.

[22] A. Gidenstam, H. Sundell and P. Tsigas, 'Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency,' in *Principles of Distributed Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 302–317. DOI: https://doi.org/10.1007/978-3-642-17653-1_23.

[23] M. Hoffman, O. Shalev and N. Shavit, 'The baskets queue,' in *Principles of Distributed Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 401–414. DOI: https://doi.org/10.1007/978-3-540-77096-1_29.

[24] M. Moir, D. Nussbaum, O. Shalev and N. Shavit, 'Using elimination to implement scalable and lock-free fifo queues,' in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05, Las Vegas, Nevada, USA: Association for Computing Machinery, 2005, pp. 253–262. DOI: 10.1145/1073970.1074013.

[25] P. Fatourou and N. D. Kallimanis, 'A highly-efficient wait-free universal construction,' in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 325–334. DOI: 10.1145/1989493.1989549.

[26] A. Kogan and E. Petrank, 'A methodology for creating fast wait-free data structures,' in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, New Orleans, Louisiana, USA: Association for Computing Machinery, 2012, pp. 141–150. DOI: 10.1145/2145816.2145835.

[27] P. Ramalhete, *FAAArrayQueue - MPMC Lock-Free Queue*, Accessed: 2024-07-29, 2016. [Online]. Available: http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html.

[28] G. H. Gonnet, 'Expected length of the longest probe sequence in hash code searching,' *J. ACM*, vol. 28, no. 2, pp. 289–304, Apr. 1981. DOI: 10.1145/322248.322254.

[29] M. Raab and A. Steger, '"Balls into bins" — a simple and tight analysis,' in *Randomization and Approximation Techniques in Computer Science*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 159–170.

[30] K. Talwar and U. Wieder, 'Balanced allocations: A simple proof for the heavily loaded case,' in *Automata, Languages, and Programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 979–990. DOI: 10.1007/978-3-662-43948-7_81.

[31] D. Alistarh, J. Kopinsky, J. Li and G. Nadiradze, 'The power of choice in priority scheduling,' in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 283–292. DOI: 10.1145/3087801.3087810.

[32] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li and G. Nadiradze, 'Distributionally linearizable data structures,' in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, vol. test, Vienna, Austria: Association for Computing Machinery, 2018, pp. 133–142. DOI: 10.1145/3210377.3210411.

[33] H. Sundell, A. Gidenstam, M. Papatriantafilou and P. Tsigas, 'A lock-free algorithm for concurrent bags,' in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 335–344. DOI: 10.1145/1989493.1989550.

[34] E. Gidron, I. Keidar, D. Perelman and Y. Perez, 'Salsa: Scalable and low synchronization numa-aware algorithm for producer-consumer pools,' in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 151–160. DOI: 10.1145/2312005.2312035.

[35] G. L. Peterson and J. E. Burns, 'Concurrent reading while writing ii: The multi-writer case,' in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, ser. SFCS '87, USA: IEEE Computer Society, 1987, pp. 383–392. DOI: 10.1109/SFCS.1987.15.

[36] S. Walzer and M. Williams, 'A Simple yet Exact Analysis of the MultiQueue,' in *33rd Annual European Symposium on Algorithms (ESA 2025)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 351, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 85:1–85:14. DOI: 10.4230/LIPIcs.ESA.2025.85.

[37] Y. Peres, K. Talwar and U. Wieder, 'The $(1 + \beta)$-choice process and weighted balls-into-bins,' in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '10, Austin, Texas: Society for Industrial and Applied Mathematics, 2010, pp. 1613–1619.

[38] M. M. Michael, 'Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects,' *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004. DOI: 10.1109/TPDS.2004.8.

[39] K. Fraser, 'Practical lock-freedom,' Ph.D. dissertation, University of Cambridge, 2003.

[40] K. von Geijer, *Artifact for: Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue*, version v1, Jan. 2025. DOI: 10.5281/zenodo.14223312.

[41] T. David, R. Guerraoui and V. Trigonakis, 'Asynchronized concurrency: The secret to scaling concurrent search data structures,' *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 631–644, 2015. DOI: 10.1145/2786763.2694359.

[42] K. von Geijer and P. Tsigas, *Artifact of the paper: How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures*, version v1, Jun. 2024. DOI: 10.5281/zenodo.11547063.

[43] M. Naumov, A. Vrielink and M. Garland, 'Parallel depth-first search for directed acyclic graphs,' in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3'17, Denver, CO, USA: Association for Computing Machinery, 2017. DOI: 10.1145/3149704.3149764.

[44] T. A. Davis and Y. Hu, 'The university of florida sparse matrix collection,' *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. DOI: 10.1145/2049662.2049663.

Paper B

Relax and Don't Stop: Graph-Aware Asynchronous SSSP

Marco D'Antonio*, **Kåre von Geijer***, Thai Son Mai, Philippas Tsigas, Hans Vandierendonck

Proceedings of the 1st FastCode Programming Challenge (FCPC), 2025, pp. 43–47

doi: 10.1145/3711708.3723446

*Authors contributed equally

Abstract

The Parallel Single-Source Shortest Path (SSSP) problem has been tackled through many implementations, yet no single approach consistently outperforms others across diverse graph structures. Moreover, most implementations require extensive parameter tuning to reach peak performance. In this paper, we introduce the AdaMW scheduler, which dynamically selects between the schedulers Wasp and MultiQueue. To achieve this, we use graph sampling and heuristics to select and configure the scheduler. In contrast to common state-of-the-art bulk-synchronous implementations, AdaMW is fully asynchronous, thus not needing to stop for global barriers. The resulting scheduler is highly competitive with the best manually-tuned, state-of-the-art implementations.

1 Introduction

Single-Source Shortest Path (SSSP) is a fundamental problem with plenty of applications [1]–[4] and well-known solutions in the sequential setting [5]–[7]. However, in a parallel context, achieving scalable performance across the large variety of existing graphs is still an open challenge with no clear winner [8]. For this reason, one of the FastCode Programming Challenge (FCPC) tracks at PPoPP'25 was to efficiently parallelize SSSP on a shared-memory machine across a variety of both small-diameter and large-diameter graphs [9]. Shared-memory implementations for SSSP have recently garnered increased interest, as modern machines can now completely fit large graphs in memory [10].

A common parallel implementation is Δ-stepping [11], which groups vertices in buckets based on their coarsened distance $dist[u]/\Delta$, and processes each bucket in parallel in a bulk-synchronous fashion. Compared to the traditional Dijkstra's algorithm [5], this approach increases parallelism but also introduces redundant work due to vertices being processed out of the work-efficient order.

Although Δ-stepping is an efficient parallel approach, its bulk-synchronous nature imposes significant synchronization overhead. This issue is particularly pronounced in large-diameter graphs, such as road networks, where buckets often contain few vertices, leading to frequent global barriers and performance degradation [12]. To address these limitations, alternative approaches relax the strict synchronization constraints while maintaining efficient parallel execution.

One such approach is Wasp [13], which builds upon coarsening of Δ-stepping but relaxes its synchronization by allowing threads to process buckets asynchronously. The asynchronous Δ-coarsening enables Wasp to perform well across most graphs, and especially ones with uniform edge weights, because the number of items is balanced across Δ-buckets. Another approach is the MultiQueue [14], [15], a relaxed priority queue that allows highly parallel operations at the expense of a degree of processing disorder. The MultiQueue exhibits lower peak throughput, but performs well on large-diameter graphs which typically expose less parallelism. Moreover, as the MultiQueue does not use Δ-coarsening, it is more resilient than Wasp to skewed edge weights, which generate more redundant work when using bucketing.

Combining the strengths of Wasp [13] and the MultiQueue [15], we introduce the fused ***Ada****ptive* ***M****ultiQueue-* ***W****asp* (AdaMW) scheduler, which dynamically selects one of the two schedulers based on the target graph. By analyzing the average degree and a sample of the edge weights, it both selects the appropriate scheduler and configures its parameters to enable suitable optimizations. AdaMW was the fastest SSSP solver at FCPC 2025 [9], reaching a geometric average performance of 150.9 $MEdges/s$ across their selected graphs, while in comparison, the second-best performing solution achieved a geometric average of 52.6 $MEdges/s$.

In this paper, we introduce AdaMW and demonstrate its great performance across a variety of graphs while requiring minimal preprocessing for selecting and configuring the scheduler. Preprocessing also enables us to selectively apply optimizations based on the graph structure. Finally, our experimental evaluation shows AdaMW's efficiency compared to Wasp [13], Δ^*-stepping [16],

Table 1: Graph datasets used in the experimental evaluation. $|V|$ is the number of vertices, $|E|$ is the number of undirected edges. SD and LD in Properties stand for "Small Diameter" and "Large Diameter". SR stands for "Skewed Random edge weights", UR for "Uniform Random edge weights", RE for "Real Edge weights".

Abbr.	**Graph**	$\|V\|$	$\|E\|$	**Properties**
LJ	LiveJournal	4.7 M	41.9 M	SD - UR
HW	Hollywood	1.0 M	55.1 M	SD - SR
ER	ER	9.89 M	490 M	SD - UR
EW	enwiki-2023	6.5 M	147.1 M	SD - SR
GR	Grid-1k-10k	9.89 M	19.6 M	LD - UR
GL	Geolife	24.6 M	77.1 M	LD - RE
NA	North America	86.0 M	107.9 M	LD - RE
SA	South America	21.8 M	29.1 M	LD - RE

and the MultiQueue [15], each configured with optimal Δ or *stickiness* per graph. The results show that AdaMW can achieve competitive or better performance than the state-of-the-art over different graph characteristics, without offline tuning of parameters.

Outline Section 2 gives a background to the problem and utilized schedulers. Section 3 highlights some related work for parallel SSSP. Section 4 describes AdaMW. Section 5 contains our evaluation, comparing AdaMW to Wasp, the MultiQueue, and Δ^*-stepping. Section 6 concludes the paper.

2 Background

Given an undirected weighted graph $G = (V, E, w)$, with vertices $v \in V$, edges $(u, v) \in E \subseteq V \times V$, and weights $w : E \to \mathbb{R}^+$, as well as a source $s \in V$, the SSSP problem is to find the shortest path from s to every other reachable $u \in V$. Our algorithm is allowed some time for graph initialization and minor preprocessing before it is given s and should minimize the time to compute a correct shortest-path distance mapping $d : V \to \mathbb{R}^+$ for all vertices. The graphs used in the evaluation are shown in Table 1 and comprise small and large diameter graphs, with varied edge weight distributions.

2.1 The Wasp Priority Scheduler

Wasp [13] is an asynchronous priority scheduler with a distributed bucketing structure and priority-aware work stealing. When a thread processes an item, it is pushed into a thread-local bucket. This alleviates the need for synchronization between threads to manage the buckets. Threads share work through a priority-aware work-stealing protocol. Stealing is performed only if the items to be stolen have a higher priority than locally available items. Otherwise, asynchrony

allows the thread to process local items with no barriers.

Wasp utilizes priority coarsening to discretize the number of priority levels, defining a parameter Δ. Therefore, for an efficient Δ-stepping implementation, Δ must be tuned to each graph.

2.2 The Relaxed MultiQueue

The base MultiQueue [14] by Rihani et al. is a concurrent relaxed priority queue, which keeps $c \cdot T$ lock-protected sequential priority queues (the sub-queues), where c is a constant tuning parameter and T is the number of threads. When an item is inserted in the MultiQueue, it is uniformly at random inserted into one of the sub-queues. To dequeue an item, the operation randomly samples the highest priority item from two sub-queues and then proceeds by dequeuing from the sampled sub-queue with the highest priority item. The *rank error* of a dequeue is defined as the number of items currently in the queue which has a higher priority than the dequeued item, and there are theoretical guarantees that it is rather low and stable [15], [17], [18].

3 Related Work

The Δ^*-stepping and ρ-stepping algorithms are built around a stepping, bulk-synchronous algorithm framework that abstracts Δ-stepping [16]. A Lazy-Batched Priority Queue supports the framework for managing vertex distances. The queue supports updates to record new distances and extraction of vertices with keys below a given threshold. Extracted vertices are processed in parallel, and relaxed neighbors have their distance updated in the queue. The framework supports both push and pull operations for SSSP, allowing the processing of dense frontiers efficiently [19].

The simple MultiQueue [14], [15] core has also led to the Stealing MultiQueue [20], which integrates thread-affine sub-queues and work stealing, as well as the Multi Bucket Queue [21], which uses Δ-coarsening to speed up its sub-queues. Additionally, a strong potential argument [17], [18] as well as a Markov chain analysis [22] give strong guarantees on the rank errors in the MultiQueue.

4 The AdaMW Scheduler

Building on top of Wasp [13] and the MultiQueue [15], we develop AdaMW to get the best of both worlds. By utilizing the allowed preprocessing period in FCPC, AdaMW analyses the graph structure, selecting and configuring the preferred scheduler for that graph type.

Our experiments show that Wasp is best for graphs with uniform weight distributions, while the MultiQueue is best for large-diameter graphs with skewed weight distributions. To characterize the graph structure, AdaMW samples a set of edge weights W. In the competition, we used $|W| = 100,000$. It then estimates whether the weights are skewed or not by looking at the ratio

$\frac{\overline{W}}{M(W)}$, where $\overline{W}$ is the average and $M(W)$ is the median of W. Furthermore, as small-diameter graphs usually have a large average degree $\overline{d}$, AdaMW selects the underlying scheduler as follows:

$$Sched = \begin{cases} Wasp, & \text{if } \overline{d} > 8 \text{ or } \frac{\overline{W}}{M(W)} < 2 \\ MQ, & \text{otherwise} \end{cases}$$

4.1 Estimating Δ in Wasp-mode

The Δ-coarsening used in Wasp requires the Δ parameter to be carefully tuned in order to balance the trade-off between parallelism and redundant work. Since increasing Δ allows for an increase in parallelism, a first-stage decision can be made based on the graph structure. Graphs with a large average degree $\overline{d}$ usually need smaller values of Δ, as vertices have more neighbors and, therefore, expose more parallelism. Low-degree graphs, instead, need a larger Δ to increase parallelism and keep all threads busy. Therefore, we initially set $\Delta = 1$ for small-diameter graphs, while for large-diameter graphs, we start with $\Delta = T$, the number of threads. Then, these values are scaled by the graph-specific ratio $\frac{M(W)}{\overline{d}}$ that accounts for the weights and the degrees of the graph, obtaining the final formula:

$$\Delta = \begin{cases} \frac{M(W)}{\overline{d}}, & \text{if } \overline{d} > 8 \\ \frac{M(W)}{\overline{d}} T, & \text{otherwise} \end{cases}$$

4.2 Configuring MultiQueue-mode

We use an optimized MultiQueue implementation by Williams et al. [15] that utilizes 8-ary heaps as sub-queues and insertion and deletion buffers to improve cache locality. Furthermore, they implement *stickiness*: threads return to the same sub-queues several operations in a row, improving cache performance at the expense of priority order [23]. We set the number of sub-queues to twice the number of threads used, as it has previously proven a good balance [14], [15]. Additionally, we set the buffer size to 16 after trying a few different options, seeing that it consistently performed slightly better than the others. Configuring stickiness is the hardest part, as we found that the optimal choice, much like Δ, varied for different graphs. However, it is a bit easier to configure than Δ, as it does not depend on the size of the weights, but rather the degree distribution. In the competition, AdaMW used $stickiness = 128$ for all sparse graphs with good results.

4.3 SSSP Optimizations

AdaMW uses two SSSP-specific optimizations. These are part of the base Wasp [13], but not of the MultiQueue [15]. The first optimization is to add bidirectional relaxation [16]. When processing a vertex, before relaxing outgoing edges, we first relax all incoming edges to possibly decrease the tentative distance of the processed vertex. This optimization is always enabled

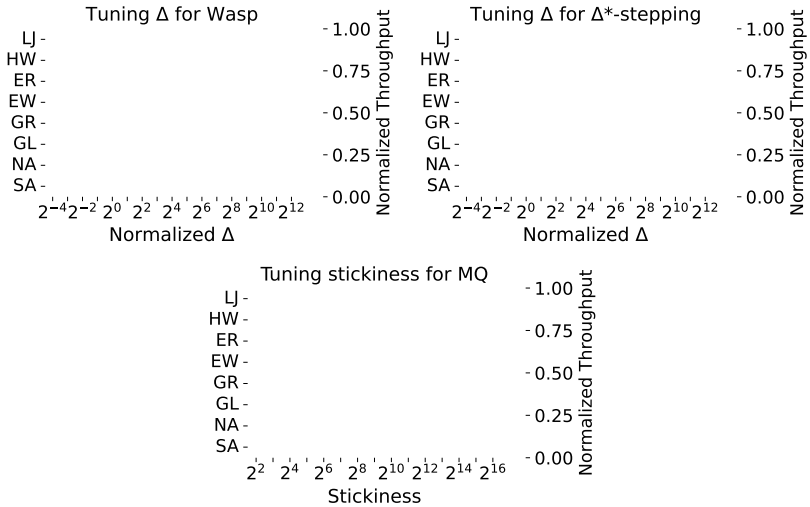


Figure 1: Performance of different Δ and *stickiness* for the schedulers. The throughput of each graph is normalized by the largest observed throughput. Δ is normalized by $\frac{M(W)}{\overline{d}}$, where $M(W)$ is the sampled median weight and $\overline{d}$ the average degree.

Table 2: Precomputation time for scheduler selection and parameter tuning per graph. **% SSSP** is the percentage relative to the average SSSP execution time. The last column shows the average and geometric mean across all graphs.

	LJ	HW	ER	EW	GR	GL	NA	SA	**mean**
Time [s]	0.014	0.012	0.012	0.012	0.013	0.016	0.067	0.025	0.021
% SSSP	12.441	10.833	3.043	7.903	12.092	6.363	12.024	11.740	8.776

for high-diameter graphs, while for small-diameter graphs, it is only used if the neighborhood fits into an L1 cache line.

Secondly, AdaMW does not add degree-1 vertices back into the scheduler when relaxing their incoming edge. These vertices are leaves of the SSSP tree; hence, relaxing their outgoing edge will not find a lower distance to any vertex. All leaf vertices are precomputed and tracked in a bit-set. AdaMW only uses this optimization when more than 10% of vertices are leaves, as constantly querying the bit-set becomes too costly otherwise.

5 Evaluation

We evaluate AdaMW, Wasp [13], the MultiQueue[1] [15], and Δ^*-stepping[2] [16] – a representative from state-of-the-art – on the graphs from FCPC, presented in

[1] https://github.com/marvinwilliams/multiqueue
[2] https://github.com/ucrparlay/Parallel-SSSP

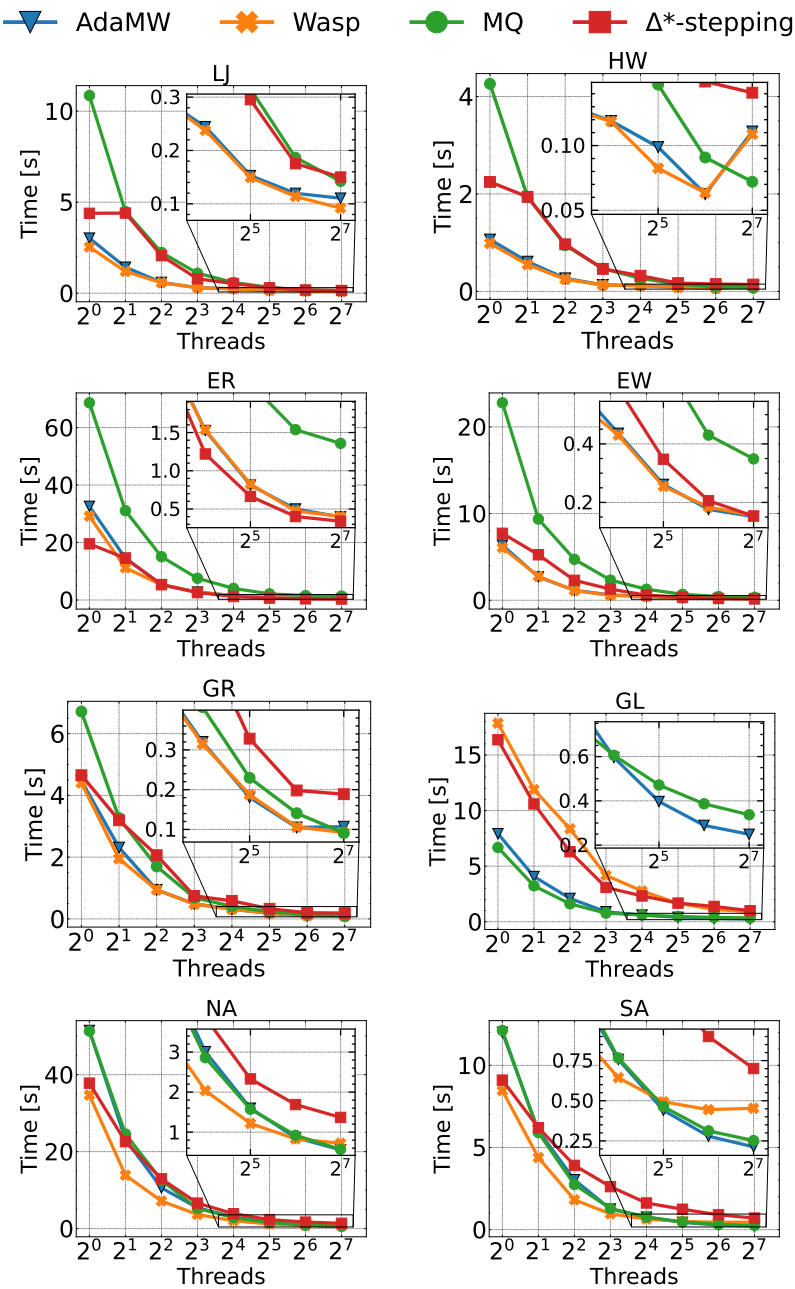


Figure 2: Scalability of the different schedulers.

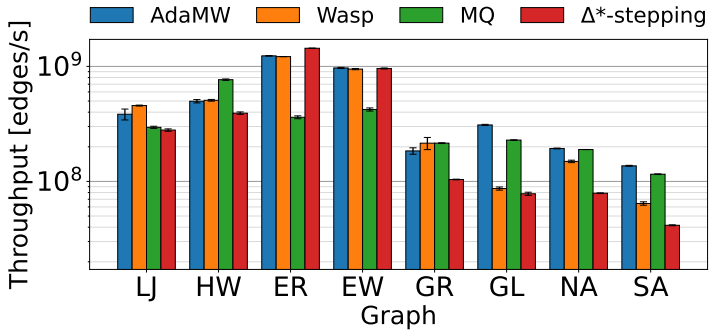


Figure 3: Scheduler throughput with 128 threads.

Table 1. All experiments are run on a dual-socket AMD EPYC 7713 processor with 64 cores per socket, simultaneous multi-threading disabled, 1TB DRAM, and 4 NUMA nodes per socket. The implementations are compiled using gcc-14.1.0 and C++20 with the `-O3` and `-march=native` compilation options. All experiments used `numactl -i all` to interleave the memory allocations across NUMA nodes.

Since Wasp and Δ*-stepping use Δ-coarsening, in our evaluation we tune Δ by sampling powers of 2, as common in literature [16], [24]. For the MultiQueue, we tune the stickiness value, while the other parameters are the same as used in AdaMW. We tune Δ and the stickiness for each thread configuration in the evaluation.

To highlight the difficulty of selecting the correct configuration parameters, Figure 1 shows their impact on the different schedulers at 128 threads. The preprocessing time for AdaMW is not considered in these plots. The optimal configuration choice varies greatly between graphs. Especially Δ is rather sensitive to bad configurations while configuring the stickiness is slightly more forgiving. Interestingly, the optimal choice of Δ is distinctly different between Wasp and Δ*-stepping, even though serving the same purpose in both schedulers. One can verify that AdaMW's choice of $\mathit{stickiness} = 2^7$ for the MultiQueue is suitable for the sparse graphs. Conversely, AdaMW's use of normalized $\Delta = 1$ for small-diameter graphs and $\Delta = T$ for large-diameter graphs can also be verified. An exception is the Geolife graph, which has skewed weights; AdaMW chooses the MultiQueue scheduler for this graph.

Table 2 reports the preprocessing time AdaMW requires to configure its schedulers. The weight sampling and leaves optimization bound this time to $O(|V|+|W|)$, and in FCPC, AdaMW currently utilizes the allowed preprocessing budget. However, this time can be further reduced, for example, by sampling a smaller W. This highlights that one can avoid tuning Δ for each graph with low impact on performance.

Figure 2 shows the strong scaling performance of the implementations. AdaMW performs well across all thread counts, despite only using heuristics for its configurations, unlike the others, which use the optimal choice. The throughput of Wasp is reduced while using both sockets for the Hollywood

graph, letting the MultiQueue overtake it at 128 threads. This behavior is reproducible, but we could not determine why it only happens for the Hollywood graph.

Finally, the performance of the schedulers at scale is shown in Figure 3. For all evaluated graphs, AdaMW is always close to the optimal. On the GeoLife, North America, and South America graphs, it is even better than all optimally tuned schedulers, thanks to the optimizations from Section 4.3.

6 Conclusion

AdaMW combines the strengths of the asynchronous Wasp and MultiQueue schedulers, dynamically adapting to different graph structures to maximize performance. Moreover, it demonstrates that a graph sampling heuristic can be used to tune critical parameters, achieving performance on par with the optimal scheduler configuration. This approach addresses a critical challenge in many schedulers — balancing adaptability and efficiency without requiring exhaustive parameter tuning.

Acknowledgments

This work was partially funded by the European Union, Horizon Europe 2021-2027 Framework Programme, Grant No. 101072456, the UK Research and Innovation, Engineering and Physical Sciences Research Council, Grant No. EP/X029174/1, and the Swedish Research Council, Grant No. 2021-05443. Views or opinions are those of the authors.

Bibliography

[1] T. Hoefler, T. Schneider and A. Lumsdaine, 'Optimized Routing for Large-Scale InfiniBand Networks,' in *2009 17th IEEE Symposium on High Performance Interconnects*, Aug. 2009, pp. 103–111. DOI: 10.1109/HOTI.2009.9.

[2] F. B. Zhan and C. E. Noon, 'Shortest Path Algorithms: An Evaluation Using Real Road Networks,' *Transportation Science*, vol. 32, no. 1, pp. 65–73, Feb. 1998. DOI: 10.1287/trsc.32.1.65.

[3] R. Guimerà, S. Mossa, A. Turtschi and L. A. N. Amaral, 'The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles,' *Proceedings of the National Academy of Sciences*, vol. 102, no. 22, pp. 7794–7799, May 2005. DOI: 10.1073/pnas.0407994102.

[4] U. Brandes, 'A faster algorithm for betweenness centrality*,' *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, Jun. 2001. DOI: 10.1080/0022250X.2001.9990249.

[5] E. W. Dijkstra, 'A note on two problems in connexion with graphs,' *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959. DOI: 10.1007/BF01386390.

[6] R. Bellman, 'On a routing problem,' *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958. DOI: 10.1090/qam/102435.

[7] L. R. Ford, 'Network Flow Theory.,' RAND Corporation, Tech. Rep., Jan. 1956. [Online]. Available: https://www.rand.org/pubs/papers/P923.html (visited on 23/09/2024).

[8] A. Azad, M. M. Aznaveh, S. Beamer, M. P. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz, H. A. Gabb, G. Gill, B. Hegyi, S. Kolodziej, T. M. Low, A. Lumsdaine, T. Manlaibaatar, T. G. Mattson, S. McMillan, R. Peri, K. Pingali, U. Sridhar, G. Szarnyas, Y. Zhang and Y. Zhang, 'Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite,' in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2020, pp. 216–227. DOI: 10.1109/IISWC50251.2020.00029.

[9] Yihan Sun and Tim Kaler, *FastCode Programming Challenge*, Accessed: 2025-01-30, 2025. [Online]. Available: https://fastcode.org/events/fastcode-challenge/.

[10] L. Dhulipala, G. E. Blelloch and J. Shun, 'Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable,' *ACM Transactions on Parallel Computing*, vol. 8, no. 1, 4:1–4:70, Apr. 2021. DOI: 10.1145/3434393.

[11] U. Meyer and P. Sanders, 'Δ-stepping: A parallelizable shortest path algorithm,' *Journal of Algorithms*, 1998 European Symposium on Algorithms, vol. 49, no. 1, pp. 114–152, Oct. 2003. DOI: 10.1016/S0196-6774(03)00076-2.

[12] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe and J. Shun, 'Optimizing ordered graph algorithms with GraphIt,' in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020, New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 158–170. DOI: 10.1145/3368826.3377909.

[13] M. D'Antonio, S. T. Mai and H. Vandierendonck, ' Toward Efficient Asynchronous Single-Source Shortest Path,' in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2025.

[14] H. Rihani, P. Sanders and R. Dementiev, 'MultiQueues: Simple Relaxed Concurrent Priority Queues,' in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 80–82. DOI: 10.1145/2755573.2755616.

[15] M. Williams, P. Sanders and R. Dementiev, 'Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues,' in *29th Annual European Symposium on Algorithms*, vol. 204, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17. DOI: 10.4230/LIPIcs.ESA.2021.81.

[16] X. Dong, Y. Gu, Y. Sun and Y. Zhang, 'Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths,' in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, Jul. 2021, pp. 184–197. DOI: 10.1145/3409964.3461782.

[17] D. Alistarh, J. Kopinsky, J. Li and G. Nadiradze, 'The power of choice in priority scheduling,' in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 283–292. DOI: 10.1145/3087801.3087810.

[18] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li and G. Nadiradze, 'Distributionally Linearizable Data Structures,' in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 133–142. DOI: 10.1145/3210377.3210411.

[19] S. Beamer, K. Asanovic and D. Patterson, 'Direction-optimizing Breadth-First Search,' in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–10. DOI: 10.1109/SC.2012.50.

[20] A. Postnikova, N. Koval, G. Nadiradze and D. Alistarh, 'Multi-queues can be state-of-the-art priority schedulers,' in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 353–367. DOI: 10.1145/3503221.3508432.

[21] G. Zhang, G. Posluns and M. C. Jeffrey, 'Multi Bucket Queues: Efficient Concurrent Priority Scheduling,' in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24, New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 113–124. DOI: 10.1145/3626183.3659962.

[22] S. Walzer and M. Williams, *A simple yet exact analysis of the multiqueue*, 2024. arXiv: 2410.08714.

[23] A. Rukundo, A. Atalar and P. Tsigas, 'Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework,' in *33rd International Symposium on Distributed Computing (DISC 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 146, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 31:1–31:15. DOI: 10.4230/LIPIcs.DISC.2019.31.

[24] L. Dhulipala, G. Blelloch and J. Shun, 'Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing,' in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17, New York, NY, USA: Association for Computing Machinery, Jul. 2017, pp. 293–304. DOI: 10.1145/3087556.3087580.

Paper C

Elastic Relaxation of Concurrent Data Structures

Kåre von Geijer, Philippas Tsigas

IEEE Transactions on Parallel and Distributed Systems (TPDS), 2025, vol. 36, no. 12, pp. 2578–2595

doi: 10.1109/TPDS.2025.3587888

Abstract

The sequential semantics of many concurrent data structures, such as stacks and queues, inevitably lead to memory contention in parallel environments, thus limiting scalability. Semantic relaxation has the potential to address this issue, increasing the parallelism at the expense of weakened semantics. Although prior research has shown that improved performance can be attained by relaxing concurrent data structure semantics, there is no one-size-fits-all relaxation that adequately addresses the varying needs of dynamic executions.

In this paper, we first introduce the concept of *elastic relaxation* and consequently present the *Lateral* structure, which is an algorithmic component capable of supporting the design of elastically relaxed concurrent data structures. Using the *Lateral*, we design novel elastically relaxed, lock-free queues, stacks, a counter, and a deque, capable of reconfiguring relaxation during run-time. We establish linearizability and define worst-case bounds for relaxation errors in our designs. Experimental evaluations show that our elastic designs match the performance of state-of-the-art statically relaxed structures when no elastic changes are utilized. We develop a lightweight, contention-aware controller for adjusting relaxation in real time, and demonstrate its benefits both in a dynamic producer-consumer micro-benchmark and in a parallel BFS traversal, where it improves throughput and work-efficiency compared to static designs.

1 Introduction

As hardware parallelism advances with the development of multicore and multiprocessor systems, developers face the challenge of designing data structures that efficiently utilize these resources. Numerous concurrent data structures exist [1], but theoretical results have demonstrated that many common data structures, such as queues, have inherent scalability limitations [2] as threads must contend for few access points. One of the most promising solutions to tackle this scalability issue is to relax the sequential specification of data structures [3], which permits designs that increase the number of memory access points, at the expense of weakened sequential semantics.

The k *out-of-order* relaxation formalized by Henzinger et al. [4] is a popular relaxation type [5]–[8] that allows relaxed operations to deviate from the sequential order by up to k; for example, for the dequeue operation on a FIFO queue, any of the first $k + 1$ items can be returned instead of just the head. This error, the distance from the head for a FIFO dequeue, is called the *rank error*.

While other relaxations, such as quiescent consistency [1] are incompatible with linearizability [9], k out-of-order relaxation can easily be combined with linearizability, as it modifies the semantics of the data structure rather than the consistency. Despite extensive work on out-of-order relaxation [4]–[6], [8], [10]–[14], almost all existing methods are static, requiring a fixed relaxation degree during the data structure's lifetime.

In applications with dynamic workloads, such as bursts of activity with latency constraints, it is essential to be able to temporarily sacrifice sequential semantics for improved performance. This paper addresses the problem of specifying and designing data structures whose degree of relaxation can be adjusted dynamically at run-time—a concept we term *elastic relaxation*. Elastically relaxed data structures enable the design of instance-optimizing systems, an area that is evolving rapidly across various communities [15]. The trade-off between rank error and throughput is well demonstrated by Williams et al. [11] and Postnikova et al. [12], whose shortest-path benchmarks show that increased relaxation improves throughput at the expense of work-efficiency. These relaxed designs have outperformed state-of-the-art static approaches in parallel SSSP on sparse high-diameter graphs [16], highlighting the potential of further exploring the field of relaxed data structures.

Several relaxed data structures are implemented by splitting the original concurrent data structure into disjoint *sub-structures*, and then using load-balancing algorithms to direct different operations to different sub-structures [5], [6], [11], [14], [17]. In this paper, we base our elastic designs on the relaxed 2D framework [5], which has excellent scaling with both threads and relaxation, as well as provable rank error bounds. The key idea of the 2D framework is to superimpose a *window* (Win) over the sub-structures, as seen in green in Figure 1 for the 2D queue, where operations inside the window can linearize in any order. The Win^{enq} shifts upward by *depth* when full, and Win^{deq} similarly shifts upward when emptied, to allow further operations. The size of the window dictates the rank error, as a larger Win allows for more reorderings.

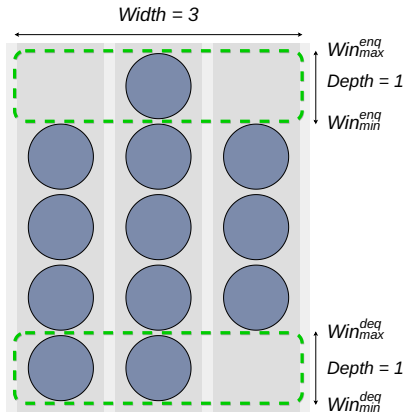


Figure 1: The 2D queue has two windows defining the operable area for the enqueue and dequeue operations.

The algorithmic design concept we propose in this paper is the *Lateral* structure, which we use to extend the 2D framework to encompass elastic relaxation. This *Lateral* is a strict concurrent version of the relaxed data structure—so for a relaxed queue, it is a strict queue—maintained alongside the substructures to track elastic changes. The *Lateral* is visualized for one of our elastic queues in Figure 2. We show two generic designs for incorporating the *Lateral* into the window mechanism of the 2D framework while achieving deterministic rank error bounds, one of which is simple, and one more complicated with better elastic capabilities. Although using the 2D framework as the base for our designs, the *Lateral* can also accommodate other designs, such as the distributed queues by Haas et al. [6], the k-queue [7], and the k-stack [4].

Contributions This work takes crucial steps toward designing reconfigurable relaxed concurrent data structures with deterministic error bounds, capable of adjusting relaxation levels during run-time.

- We introduce the concept of *elastic relaxation*, allowing rank errors to change over time. Furthermore, we introduce the *Lateral* component for efficiently enhancing relaxed data structures with elasticity.

- We elastically extend the 2D framework, deriving elastic queues, stacks, a counter, and a deque using the *Lateral*, and establish both correctness and rank error bounds. Our *Lateral*-based designs are grouped into two paradigms: Lateral-plus-Window (LpW), which can be applied to other relaxed structures with minimal modification, and Lateral-as-Window (LaW), which are simpler, but specially tailored for the 2D framework.

- We evaluate the scalability of our proposed data structures against both non-relaxed and relaxed data structures. These evaluations show that the elastic designs significantly outscale non-relaxed data structures, and perform

as well as the best statically relaxed ones, while simultaneously supporting elastic relaxation.

- Finally, we demonstrate the elastic capabilities of our designs by implementing a lightweight, contention-aware controller for adjusting relaxation. We evaluate it in two dynamic settings: a producer-consumer benchmark, where it adapts relaxation over time, and a BFS traversal, where it improves both runtime and work-efficiency compared to static configurations.

Outline Section 2 presents the preliminaries, focusing on the 2D framework. Section 3 introduces elastic relaxation and our novel data structures, which we then prove correct and provide worst-case bounds for in Section 4. Section 5 experimentally evaluates the new algorithms. Section 6 discusses related work. Section 7 concludes the paper.

2 Preliminaries

An *out-of-order* relaxed data structure relaxes the sequential specification of the underlying data structure to allow operations to linearize [9] out of order. The concept was formalized by Henzinger et al. [4] in their theoretical framework *quantitative relaxation*, which defines *relaxation errors* based on transition costs in the linearized history. In the case of a FIFO queue, if the third item is dequeued, the out-of-order error would be 2, as one would need to remove the enqueue operations of the two first items for the operation to be correct. Recent relaxed queue designs [11], [14] distinguish *rank errors* and *delay errors*, where rank errors are the errors described above, and the *delay* when dequeuing an item is the number of earlier dequeues that returned items of lower rank; items which were enqueued later in a FIFO queue.

The algorithm descriptions in this paper assume a sequentially consistent [18] programming model for simplicity. However, efficient implementations, such as ours for the evaluation [19], should use less restrictive memory orderings where possible. The algorithms also utilize the atomic *compare-and-swap* instruction, where `CAS(loc, exp, new)` atomically sets the memory at `loc` equal to `new` if the previous value at `loc` equals `exp`.

2.1 Static 2D Framework

The 2D framework for k out-of-order relaxed data structures [5] (hereby called the *static* 2D framework) unifies designs across FIFO queues, stacks, deques, and counters. Furthermore, its implementations outscale other relaxed implementations from the literature with deterministic error bounds [4], [6], [7] and its throughput scales monotonically with k. It achieves its good performance by distributing operations across disjoint sub-structures – reducing contention – as well as having threads return to the same sub-structure for several successive operations – increasing data locality.

The 2D framework can be seen as a two-dimensional grid, where the columns are concurrent sub-structures, and the rows are indexes within those

sub-structures. In this view, a normal (strict) queue would be a single column, where one inserts items at rows $0, 1, 2 \ldots$, and removes them in the same order. The core of the 2D framework is to superimpose a 2D *window* (Win) over the grid, and only allow operations at rows and columns within Win. An example 2D queue state is shown in Figure 1. If there is no valid operation in Win, a thread *shifts* Win up or down to allow further operations. For conciseness, we call the number of sub-structures or columns the *width* (Win_{width}), and the number of rows spanned by each window the *depth* (Win_{depth}).

Algorithm 1 shows the core code for the 2D queue, which is almost identical to the other data structures. For each operation, the thread iterates over the columns and tries to linearize on a row within Win, otherwise shifting Win before trying again. The dequeue deviates from this, in that it returns `EMPTY` if a linearizable double-collect scan [20] validates that all sub-queues are empty. The static Win mainly includes a *max* (Win_{max}) field ($Win_{min} := Win_{max} - depth$), and Win can thus be shifted with a single CAS that increases or decreases Win_{max}. An important detail is that the iterations at lines 1.5 and 1.13 should start at the column where the thread last linearized, for better cache performance, similarly to *stickiness* in later related work [11].

Algorithm 1: Core of the Static 2D Queue

```
1.1  global Window enqWin;
1.2  global Window deqWin;
1.3  function Enqueue(item)
1.4      win ← enqWin;
1.5      foreach sub ∈ sub_structures do                ▷ Start at last visited
1.6          while win.min() ≤ (row ← sub.enq_row) < win.max do
1.7              if win ≠ enqWin then goto line 1.4;
1.8              if sub.TryEnqueue(item, row) then return;
1.9      ShiftEnq(win);                                 ▷ Increment enqWin.max with CAS
1.10     goto line 1.4;
1.11 function Dequeue()
1.12     win ← deqWin;
1.13     foreach sub ∈ sub_structures do                ▷ Start at last visited
1.14         while win.min() ≤ (row ← sub.deq_row) < win.max do
1.15             if win ≠ deqWin then goto line 1.12;
1.16             ok, item ← sub.TryDequeue(row);
1.17             if item = EMPTY then break else if ok then return item;
1.18     if DoubleCollectAllEmpty(sub_queues) then return EMPTY;
1.19     if sub.deq_row = win.max ∀ sub ∈ sub_structure then ShiftDeq(win);
1.20     goto line 1.12;
```

Looking closer at the 2D queue, both its windows (Win^{enq} and Win^{deq}) only contain a Win_{max} field, and both *ShiftEnq* and *ShiftDeq* simply increment Win_{max} by *depth* with a CAS. As in all 2D designs, if another thread shifts Win before you, your shift aborts after the failed CAS to not shift Win twice. Using

the state in Figure 1 as an example, Win^{enq} spans row 4 where columns 0 and 2 are valid for enqueues, while column 1 is already enqueued into. If column 0 and 2 had been enqueued into, the enqueue operation would atomically shift Win^{enq}_{max} up by *depth*, before again trying to find a valid column to enqueue at. The sub-queues in the 2D queue are implemented as Michael-Scott (MS) queues [21] with counted head and tail pointers. This choice makes it easy to implement *TryEnqueue* (line 1.8) and *TryDequeue* (line 1.16), which only succeed if able to linearize at the correct row. Furthermore, the counted head and tail pointers are used to determine the row of the queue's head or tail (lines 1.6 and 1.14). The usage of such counted pointers has been used in similar relaxed queues [6], and they are simple to implement with the 16-byte CAS on x86 [22]. If not having access to a 16-byte CAS, one can instead include the row count within each node, or use an array-based sub-queue which maps row indexes to array slots [23]–[25].

The 2D stack is only slightly different from the queue in Algorithm 1. Firstly, only one *Win* is required, as both pushes and pops operate on the same side of the data structure. This leads to Win_{max} no longer increasing monotonically, but instead decreasing under pop-heavy workloads, and increasing under push-heavy ones. Furthermore, Win_{max} shifts by $Win_{depth}/2$ instead of Win_{depth}, which roughly leaves *Win* half-full after each shift. As Win_{max} does not strictly monotonically increase, *Win* also includes a version count to circumvent the ABA problem. The sub-stacks are implemented as Treiber stacks [26], again using counted pointers to the top item, to track its row and facilitate easy *TryPush* and *TryPop* methods.

The 2D deque can be derived as a combination of the queue and stack, where it has one *Win* at each end of the data structure, as the 2D queue, and these *Win* can shift both up and down (by *depth*/2), as the 2D stack. The simplest way to implement the sub-deques is using the Maged deque [27] with counted top and bottom pointers.

Finally, the 2D counter can be seen as a special case of the 2D stack which disregards the items, and only tracks the sub-structure sizes. Therefore, it only needs a single *Win*, and updates its sub-counters in the same way the 2D stack would increment or decrement the top row of each sub-stack. The size of the counter is approximated by the row of one sub-counter multiplied by Win_{width}.

In summary, the 2D designs can be modeled as two-dimensional grids, where items are inserted or deleted from valid rows within the current *Win*. The hard bound imposed by the *Win* makes it possible to give worst-case guarantees on the rank errors of the operations. For example, for the 2D queue, operations that linearize during different *Win* are totally ordered by proxy of the *Win* order, and each sub-queue is totally ordered as it is a strict MS queue. Therefore, it is simple to see that rank errors are bounded by $(Win_{width} - 1)\, Win_{depth}$ [5]. Similar bounds hold for the other designs [5], but are slightly harder to derive [28].

3 Design of Elastic Algorithms

This section introduces several algorithms with *elastic* out-of-order relaxation, by extending the static 2D framework [5]. The two dimensions of relaxation in the 2D framework are Win_{width} and Win_{depth}, whose optimal configurations depend on both the number of threads and the surrounding use-case. In general, Win_{width} is often set proportional to the number of threads [5], [11], while Win_{depth} is determined by algorithm-specific factors. As Win_{width} and Win_{depth} serve different uses, it is essential that both of them can be reconfigured during runtime for an elastic extension of the 2D framework to be practically useful.

Our new elastic designs therefore let Win_{width} and Win_{depth} change with every window shift. In the pseudocode, we use *width_desired()* and *depth_desired()* as hooks representing the currently desired values, which can be controlled either manually or by a dynamic controller. Changing Win_{depth} is straightforward in practice, although it affects the error bounds. Efficiently varying Win_{width} demands more attention, as it involves modifying the number of active sub-structures while maintaining error bounds. To keep this lightweight, our approach is to leave existing items in place and apply the new Win_{width} only to future insertions.

We track the elastic changes to Win_{width} and Win_{depth} using a *Lateral* structure. This *Lateral* consists of a set of nodes, each corresponding to a range of rows in the substructures, and provides a width bound for those rows. The *Lateral width bound* of a row is the largest width bound of any *Lateral* node spanning that row. This approach of using the *Lateral* is visualized in Figure 2 for the queue. There, items were initially enqueued with $Win^{enq}_{width} = 3$, but then it changed to $Win^{enq}_{width} = 4$ at the second row, and then again to $Win^{enq}_{width} = 2$ at the fourth row. These two changes can be seen in the *Lateral*, which has a *Lateral* node for each, storing the new width and row where the change took effect. Each *Lateral* node is enqueued at a row, and implicitly spans all rows until the next *Lateral* node. Notably, there are now empty slots in the sub-structures, such as the last column on the first row, where no item will be inserted. However, the rows are purely logical, in the implementations only corresponding to counters, so these empty rows do not consume memory. Importantly, Win^{enq} is allowed to adjust its dimensions freely, while Win^{deq} must adapt its *width* to these changes by inspecting in which columns items may have been enqueued, as indicated by the *Lateral*.

Our elastic queue designs, presented in Section 3.1 and 3.2, are relatively simple, as only a single Win^{enq} spans each row. In contrast, the remaining designs—such as the stacks in Section 3.3 and 3.4—are more complex. There, the width of a row can change multiple times during execution, requiring the *Lateral* to be updated continuously to stay correct. To denote this correctness, we call the *Lateral consistent* if the *width bound* for each row is greater than or equal to the maximum column index of all items currently in that row.

We identify two design paradigms for elastically extending the 2D data structures using the *Lateral*. The Lateral-plus-Window (LpW) designs add the *Lateral* alongside the existing static components and modify only the logic for shifting the windows. In contrast, the Lateral-as-Window (LaW) designs

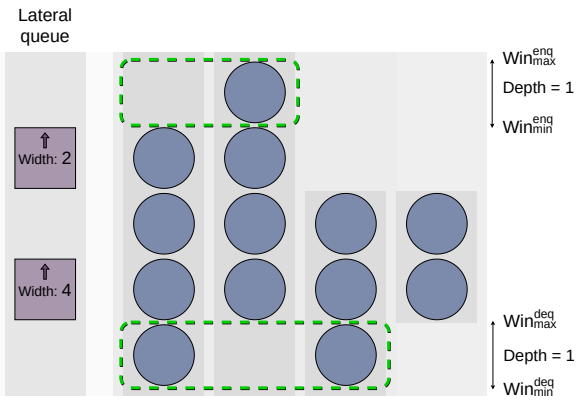


Figure 2: By adding a *Lateral* to the 2D queue, changes in width at Win^{enq} can be tracked and adjusted to by Win^{deq}. This depicts a possible state in the elastic LpW queue.

merge the *Lateral* and *Win* into a single linked-list structure of windows. LaW designs are simpler to implement and analyze than LpW designs, but they are more specific to the 2D framework and slightly less flexible in supporting certain elastic behaviors. The difference between the LpW and LaW designs can be seen when comparing the LpW queue in Figure 2 and LaW queue in Figure 3, where the LpW design only adds a *Lateral* node when Win^{enq}_{width} changes, and the LaW design lets each *Win* be a *Lateral* node – enqueueing one each window shift.

In the following sections, we present algorithms for both LpW and LaW queues and stacks, as well as a LaW deque and an LpW counter. These algorithms focus only on the novel modifications to the *Win* structure and its shifting logic; all other behavior follows the static procedures of Algorithm 1, except that operations now traverse only the substructures within the current Win_{width}.

3.1 Elastic Lateral-plus-Window 2D Queue

The elastic Lateral-plus-Window (LpW) queue builds on the static 2D queue from Algorithm 1, with the key change that operations iterate only over columns within the current Win_{width}. Algorithm 2 presents the extensions to the static version, including the integration of the *Lateral* structure and. Elasticity is achieved by allowing Win^{enq}_{depth}, Win^{enq}_{width}, and Win^{deq}_{depth} to change mostly freely at each *Win* shift. When Win^{enq}_{width} changes, a new node is added to the *Lateral*, recording the new width for future dequeuers. This ensures that Win^{deq}_{width} always covers at least the rightmost populated column between Win^{deq}_{min} and Win^{deq}_{max}.

A possible state of the LpW queue is shown in Figure 2. As there are two *Lateral* nodes, there has been two changes in Win^{enq}_{width}, going from the current

Algorithm 2: Elastic *Win* and *Lateral* in the LpW queue

```
2.1  struct EnqueueWindow
2.2      uint max;
2.3      uint width;
2.4      uint next_width;
2.5  struct DequeueWindow
2.6      uint max;
2.7      uint width;
2.8  struct LateralNode
2.9      LateralNode* next;
2.10     uint row;
2.11     uint width;
2.12 struct Lateral
2.13     LateralNode* head;
2.14     LateralNode* tail;
2.15 global EnqueueWindow* enqWin;
2.16 global DequeueWindow* deqWin;
2.17 global Lateral lat;

2.18 function ShiftEnq(old_win)
2.19     if old_win.width ≠
           old_win.next_width then
           lat.SyncTail(old_win) ;
2.20     new_win ← EnqueueWindow {
2.21       width: old_win.next_width,
2.22       next_width: width_desired(),
2.23       max: old_win.max +
           depth_desired()
2.24     };
2.25     CAS(&enqWin, old_win, new_win);

2.26 method Lateral.SyncTail(window)
2.27     tail ← lat.tail;
2.28     if tail.row ≤ window.max then
2.29         new_tail ← LateralNode {
2.30           row: window.max + 1,
2.31           width: window.next_width
2.32         };
2.33         lat.TryEnqueue(new_tail, tail);
2.34 method Lateral.TryEnqueue(new, old);
2.35 method Lateral.TryDequeue(old);
2.36 method Lateral.Peek();
2.37 function ShiftDeq(old_win)
2.38     while (head ← lat.Peek()).row ≤
           old_win.max do
2.39         lat.TryDequeue(head); ▷ Cleanup
2.40     if head.row > old_win.max + 1 then
2.41         width ← old_win.width;
2.42     else
2.43         width ← head.width;
2.44         head ← head.next;
2.45     max ← Min(enqWin.max, head.row - 1,
           old_win.max + depth_desired()) ;
2.46     new_win ← DequeueWindow {
2.47       width: width,
2.48       max: max
2.49     };
2.50     CAS(&deqWin, old_win, new_win);
```

$Win^{deq}_{width} = 3$ at the bottom, to $Win_{width} = 4$ at the second row, and finally to $Win_{width} = 2$ at the fourth row. Changes in Win_{depth} are not tracked in the *Lateral*. The core idea is that Win^{enq}_{width} changes freely, but must *first* enqueue a *Lateral* node with the width change, and that the Win^{deq}_{width} adapts based on the *Lateral*.

The LpW queue expands the fields within the windows compared to the static algorithm, which only tracks Win_{max}. Firstly, Win^{enq}_{width} and Win^{deq}_{width} are added so that operations know which sub-queues to iterate over. Additionally, $Win^{enq}_{next\_width}$ is added, which holds the upcoming Win^{enq}_{width}. The delay of one Win^{enq} shift before applying this new width is included to keep the *Lateral* consistent. Both the sub-queues and *Lateral* are implemented as MS queues [21], and provide *TryEnqueue* and *TryDequeue* methods, which succeed only if that end of the queue matches the argument.

Shifting Win^{enq} is done by creating a new Win^{enq} with a desired Win^{enq}_{depth}, Win^{enq}_{width} equal to the last $Win^{enq}_{next\_width}$, and $Win^{enq}_{next\_width}$ as desired, finally updating the global Win^{enq} with a CAS (line 2.25). However, before this is done, if $Win^{enq}_{width} \neq Win^{enq}_{next\_width}$ (line 2.19), the algorithm ensures a *Lateral* node with $width = Win^{enq}_{next\_width}$ is enqueued into the *Lateral* first. Enqueuing

this node into the *Lateral* before setting the new Win^{enq}_{width} into effect ensures that the *Lateral* is always consistent, so that Win^{deq}_{width} will always cover all populated columns between Win^{deq}_{max} and Win^{deq}_{min}.

Conversely, when shifting Win^{deq} (line 2.37), the algorithm first dequeues all *Lateral* nodes below Win^{deq}_{max} (line 2.38), as they represent changes in *width* that have already been logically used. If the bottom of the new Win^{deq} is the same row as the bottommost *Lateral* node, the new Win^{deq}_{width} is set to that node's *width* (line 2.43); otherwise, it remains the same as the previous Win^{deq}_{width} (line 2.41). The new Win^{deq}_{max} is then set by incrementing the old Win^{deq}_{max} by the desired *depth*, while ensuring it does not surpass Win^{enq}_{max} or the row of the *Lateral* tail (line 2.45). This restriction exists to ensure that Win^{deq} overlaps with at most one *Lateral* node, which occurs when that node has $row = Win^{deq}_{min} + 1$. This maintains the invariant that all rows in a Win^{deq} have equal widths.

Having covered how Win^{enq}, Win^{deq}, and the *Lateral* synchronize, the detail of which row to enqueue items on remains. In the static 2D queue, items are always enqueued immediately above the last item. But as seen at the bottom right in Figure 2, our elastic queues can have gaps in the sub-queues following increases in Win^{enq}_{width}. Therefore, items are instead enqueued at the larger of the row above the last item and $Win^{enq}_{min} + 1$. This can be implemented by storing row information within each node in the sub-queues, but this pointer-indirection causes a slight overhead, and one can instead either include Win^{enq}_{depth} in Win^{enq}, or use the row of the *Lateral*'s tail to calculate the row.

3.2 Elastic Lateral-as-Window 2D Queue

This elastic LaW queue merges the *Win* and *Lateral* structures into a Michael-Scott (MS) queue [21] of windows. Algorithm 3 presents the extensions relative to the static queue in Algorithm 1. It again uses conditional *TryEnqueue* and *TryDequeue* methods, which only linearize if the tail or head matches the expected value. Each *Win* in the *Lateral* contains *max*, *depth*, and *width*. The global Win^{enq} and Win^{deq} become the head and tail nodes in the *Lateral*. Every shift of Win^{enq} enqueues a new *Win* into the *Lateral* (line 3.12), and each shift of Win^{deq} dequeues a *Win* (line 3.20).

An example state of the LaW queue is shown in Figure 3, and can be compared to the LpW queue in Figure 2, which may result from the same sequence of operations. In the LpW queue, only Win^{enq} and Win^{deq} are stored, and the *Lateral* contains only *changes* in Win^{enq}_{width}. In contrast, the LaW queue stores information about *every Win* between Win^{enq} and Win^{deq}. This results in increased memory usage, but the overhead is typically modest, as each *Win* is just a node in a linked list that often spans many items. The benefit of keeping all *Win* instances in the *Lateral* is that it unifies the *Lateral* and *Win* designs, making it easier to change relaxation and track its changes over time.

As shown in *ShiftEnq* (line 3.12), Win_{depth} and Win_{width} can be updated from arbitrary variables at each shift, enabling elastic behavior. The main

Algorithm 3: The *Lateral* in the elastic LaW queue

```
3.1  struct Window
3.2      Window* next;
3.3      uint max;
3.4      uint depth;
3.5      uint width;
3.6  struct Lateral
3.7      Window* head;
3.8      Window* tail;
3.9  global Lateral lat;
3.10 method Lateral.TryEnqueue(old, new);
3.11 method Lateral.TryDequeue(old);
3.12 function ShiftEnq(old_win)
3.13     depth ← depth_desired();
3.14     new_win ← Window {
3.15       width: width_desired(),
3.16       depth: depth,
3.17       max: oldWin.max + depth
3.18     };
3.19     lat.TryEnqueue(old_win, new_win);
3.20 function ShiftDeq(old_win)
3.21     lat.TryDequeue(old_win);
```

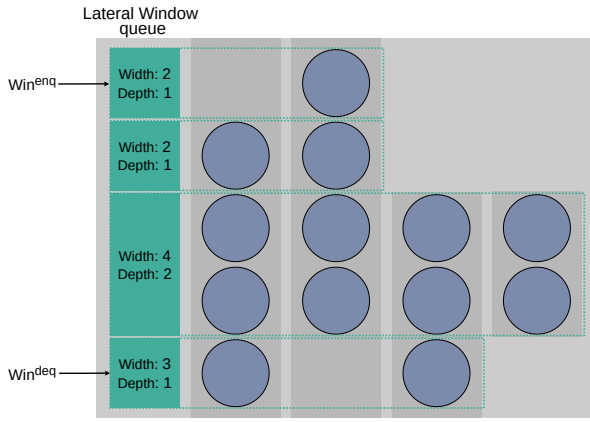


Figure 3: A possible state in the Elastic LaW queue, where windows are stored in the *Lateral*, such that Win^{enq} is the tail and Win^{deq} is the head. Looking at the *Lateral*, the queue must have initially been configured as $Win^{enq}_{width} = 3$, $Win^{enq}_{depth} = 1$, but then changed to $Win^{enq}_{width} = 4$, $Win^{enq}_{depth} = 2$, and finally $Win^{enq}_{width} = 2$, $Win^{enq}_{depth} = 1$.

drawback of this design is that the relaxation can only change at Win^{enq}, and must propagate through the queue to eventually reach Win^{deq}.

Finally, as in the LpW queue, the LaW queue only enqueues items within the current Win^{enq}. After selecting a sub-queue, each item is enqueued at row $max(Win^{enq}_{max} - Win^{enq}_{depth}, sub\_queue.tail\_row) + 1$, which may create gaps in the sub-queues, as shown in Figure 3.

3.3 Elastic Lateral-as-Window 2D Stack

The elastic LaW stack retains as much simplicity from the LaW queue as possible, while handling the bidirectional window shifts of the 2D stack. As the LaW queue in Figure 3, the LaW stack merges the *Win* with the *Lateral*, keeping a Treiber stack [26] of windows. Pushes and pops work similarly to the static 2D stack, where the thread finds a sub-stack to operate on within

the current *Win* (the top *Win* in the *Lateral*), and then linearizes with a CAS. Additionally, sub-stack nodes store the row of the node below, and items are always inserted above Win_{min} as in the elastic queues.

The main complication of elastically extending the static 2D stack is that its *Win* can shift downward at any point after all sub-queues inside Win_{width} have been seen at Win_{min}. This does not guarantee that all sub-stacks remain at Win_{min} at the linearization of the shift. Therefore, a naive elastic design could shift downward from Win^A to Win^B and leave items above Win^B_{max}, and outside Win^B_{width}, which would lead to correctness issues.

The LaW stack is designed around the simplifying constraint that items are always pushed within, and remain within, a *Win* on the *Lateral*, as formalized by Property 3.1. This property simplifies elastic extension and also leads to a tighter and simpler rank error bound than the Static 2D stack (as presented and discussed in [28]).

Property 3.1 (LaW Stack window cover)**.** *If item x is pushed to sub-stack col_x at row row_x, and has not been popped, then $\exists Win^x \in Lateral$ such that $col_x < Win^x_{width}$ and $Win^x_{min} < row_x \leq Win^x_{max}$.*

Algorithm 4: Elastic 2D LaW Stack

```
4.1  function ShiftUp(old_win)
4.2      if old_win.shrinking then
4.3          new_win ← old_win.Clone();
4.4          new_win.shrinking ← false;
4.5      else
4.6          depth ← depth_desired();
4.7          new_win ← {
4.8              max: old_win.max + depth/2,
4.9              depth: depth,
4.10             width: width_desired(),
4.11             shrinking: false,
4.12             next: old_win
4.13         };
4.14     CAS(&win, old_win, new_win);

4.22 function ShiftDown(old_win)
4.23     if !old_win.shrinking then
4.24         new_win ← old_win.Clone();
4.25         new_win.shrinking ← true;
4.26     else
4.27         assert(∀sub ∈ sub_stacks[old_win.next.width... old_win.width]:
               sub.row < old_win.max − old_win.depth and sub.frozen);
4.28         assert(∀sub ∈ sub_stacks[0... old_win.width]:
               sub.row ≤ old_win.next.max);
4.29         new_win ← old_win.next.Clone();
4.30     CAS(&win, old_win, new_win);

4.15 struct Window
4.16     Window* next;
4.17     uint max;
4.18     uint depth;
4.19     uint width;
4.20     bool shrinking;

     // Top of the Lateral stack
4.21 global Window win;
```

Algorithm 4 shows how to shift the *Win* for the LaW stack in a way to uphold Property 3.1. A downward *Win* shift is now split in two steps: shrinking *Win* (line 4.25), and in a later shift invocation, shifting *Win* down (line 4.29).

When $Win_{shrinking}$ is set, push operations are only allowed to push within the intersection of *Win* and Win^{next}. Similarly, upward shifts either unshrink the *Win*, or if if it is not shrinking, push a new *Win* to the *Lateral*.

Furthermore, a *frozen* bit is incorporated into the counted top pointer of each sub-stack. If this bit is set, a push is not allowed to linearize on the sub-stack. Pop operations must have seen this bit set on all sub-stacks outside Win^{next}_{width}, together with them being at or below Win_{min}, before shifting down (assert at line 4.27). As a sub-stack push can only linearize by first reading the descriptor, then validating it is valid under the global *Win*, and finally linearizing with a CAS, setting these *frozen* bits ensure that no item can be pushed outside the current *Win*, ensuring Property 3.1.

Shifting the *Win* upward is a similar process, but without the possibility of leaving items outside the *Lateral* (as it only expands it). Firstly, if $Win_{shrinking}$ is set, we clear the bit and undo the shrinking with a CAS (line 4.4). Otherwise, a new *Win* is simply created with the desired Win_{width} and Win_{depth}, and pushed to the *Lateral* (line 4.7). Finally, when push operations encounter a frozen sub-stack valid for a push, it first clears the *frozen* bit with a CAS, and then re-reads the sub-stack and *Win* before proceeding with the push.

3.4 Elastic Lateral-plus-Window 2D Stack

The elastic LpW stack is similar in design and motivation to the LpW queue. It can change relaxation behavior more flexibly than the LaW stack, and slightly improves performance by not having to shrink the *Win* at shifts. Instead of storing all windows in the *Lateral*, as the LaW stack, it only stores *Lateral* nodes at rows where the width changes, similarly to the LpW queue in Figure 2. As this algorithm can shift the *Win* more freely than the LaW stack, it requires more care to keep the *Lateral* consistent, which is its main drawback.

Algorithm 5 shows how to shift the *Win* and synchronize the *Lateral* in the LpW stack. It again implements the *Lateral* as a Treiber stack [26], but now only pushes nodes to it when the *width* changes. However, as the *width* of a row can change many times in a 2D stack, due to its non-monotonic Win_{max}, the *Lateral* nodes must be re-adjusted before every *Win* shift (line 5.12). There, it possibly replaces several *Lateral* nodes with a single CAS (line 5.73. Like the LaW stack, it always pushed items above Win_{min}, and stores the row count of items in their sub-stack nodes.

To increase the elasticity from the LaW stack, the LpW stack separates Win_{width} into Win_{push_width} and Win_{pop_width} for the push and pop operations. Then, Win_{push_width} can be updated at will (line 5.14) and Win_{pop_width} is set as the upper bound on the *width* of rows in the *Win*, which can be calculated using the *Lateral* (line 5.24).

Shifting the *Win* (line 5.11) is done by first ensuring the *Lateral* is stabilized (line 5.12), then creating a new *Win*, including Win_{push_width} and Win_{pop_width} as described above, finally linearizing with a CAS (line 5.25). Furthermore, the *Win* stores the direction of the shift (`UP` or `DOWN`), and the last Win_{push_width}, which are used to keep the *Lateral* consisent at the next shift.

The function *StabilizeLateral* is the core of the LpW stack (line 5.12), keeping

Algorithm 5: *Lateral* and *Win* logic for the elastic LpW stack

```
5.1  struct Window
5.2      uint max;
5.3      uint depth;
5.4      uint push_width;
5.5      uint pop_width;
5.6      uint last_push_width;
5.7      enum last_shift;                ▷UP or DOWN
5.8      uint version;
5.9  global Window* win;
5.10 global Lateral lat;
5.11 function Shift(old_win, dir)
5.12     StabilizeLateral(lat, old_win);
5.13     new_win ← Window {
5.14         push_width: width_desired(),
5.15         last_push_width:
               old_win.push_width,
5.16         depth: depth_desired(),
5.17         last_shift: dir,
5.18         version: old_win.version + 1
5.19     };
5.20     if dir = UP then
5.21         new_win.max ← old_win.max +
               new_win.depth/2;
5.22     else
5.23         new_win.max ← old_win.Min() +
               new_win.depth/2;
5.24     new_win.pop_width ←
           max(new_win.push_width,
           lat.Width(new_win.Min()));
5.25     CAS(&win, old_win, new_win);
5.26 method LateralNode.Update(self, old_win)
5.27     if self.row ≤ old_win.Min() then
5.28         return self;
5.29     new_node ← node.Clone();
5.30     if node.width ≤
           old_win.push_width then
5.31         new_node.row ← old_win.Min();
5.32     else if node.width >
           old_win.last_push_width and
           old_win.last_shift = DOWN then
5.33         last_min ← old_win.max −
               old_win.depth/2;
5.34         new_node.row ← min(node.row,
               last_min);
5.35     next ←
           new_node.next.Update(old_win);
5.36     if new_node.row ≤
           node.next.row then
5.37         return next;                ▷Remove node
5.38     else
5.39         new_node.next ← next;
5.40         return new_node;
5.41 struct Lateral
5.42     LateralNode* top;
5.43     uint version;
5.44 struct LateralNode
5.45     LateralNode* next;
5.46     uint row;
5.47     uint width;
5.48 method Lateral.Width(self, row)
5.49     return max(node.width for node in
           self.nodes where node.row > row);
5.50 method Window.Min(self)
5.51     return self.max - self.depth;
5.52 function StabilizeLateral(old_lat, old_win)
5.53     if win ≠ old_win or old_lat.version =
           old_win.version then
5.54         return ;                    ▷Already completed
         //Phase 1: Update local Lateral
5.55     new_lat.top ← old_lat.top.Update(old_win);
         //Phase 2: Push new top if changed
           width
5.56     upper_bound ← max(old_win.max,
           stack.row for stack in
           subStacks[old_win.push_width...
           old_win.last_push_width]);
5.57     if old_win.push_width >
           old_win.last_push_width and
           new_lat.top.row < old_win.Min() then
5.58         new_node ← LateralNode {   ▷New top
5.59             row: old_win.Min(),
5.60             width: old_win.last_push_width,
5.61             next: new_lat.top
5.62         };
5.63         new_lat.top ← new_node; ▷New width
5.64     else if old_win.push_width <
           old_win.last_push_width    and
           new_lat.top.row < upper_bound then
5.65         new_node ← LateralNode {   ▷New top
5.66             row: upper_bound,
5.67             width: old_win.last_push_width,
5.68             next: new_lat.top
5.69         };
5.70         new_lat.top ← new_node; ▷New width
5.71     if new_lat ≠ old_lat then
5.72         new_lat.version ← old_win.version;
5.73         CAS(&lat, old_lat, new_lat);
```

the *Lateral* consistent at every *Win* shift. The function maintains the stack invariant that for any *Lateral* node l, all sub-stack items between $l.row$ and $l.next.row$ will be at columns less than $l.width$. We divide the synchronization into two phases, which together create a local top candidate for the *Lateral* stack, and linearize with a CAS at line 5.73.

The first phase clones and tries to *lower Lateral* nodes above Win_{min} (line 5.54). By lowering a *Lateral* node l to row r, we set $l.row \leftarrow min(l.row, r)$, and then if $l.row \leq l.next.row$, l is removed from the stack (by re-linking its parent $l', l'.next \leftarrow l.next$). If a *Lateral* node $l, l.row > Win_{min}$ has $l.width \leq Win_{push\_width}$, then it is lowered to Win_{min} (line 5.31), as new nodes can have been pushed outside $l.width$ within the *Win*, invalidating its bound. Otherwise, if $l.width > Win_{last\_push\_width}$ and the last shift was downwards, all sub-stacks must have been seen at the previous Win_{min} before it shifted down, so l is lowered to a conservative estimate of the previous Win_{min} (line 5.34). This estimate only deviates from the actual previous Win_{min} when the last Win^{min} was smaller than $Win_{depth}/2$, in which case it overestimates, thus keeping a correct bound.

In the second phase (line 5.55), a new *Lateral* node with $width = Win_{last\_push\_width}$ is pushed if the *width* has changed.

- If $Win_{push\_width} > Win_{last\_push\_width}$, the width has increased, and a *Lateral* node is pushed at Win_{min} to signify that the *width* from there on is smaller. This is not needed for correctness, but is helpful in limiting the Win_{pop_width} if the *width* shrinks in the future.

- If $Win_{push\_width} < Win_{last\_push\_width}$, the *width* has decreased and a new *Lateral* node needs to be pushed at the highest row containing nodes between Win_{push_width} and $Win_{last\_push\_width}$. This can not reliably be calculated from the present *Win* variables, so we simply iterate over the sub-stacks (line 5.56).

In summary, the elastic LpW stack uses a similar idea as the elastic LpW queue, but needs to do extra work to maintain the *Lateral* invariant. However, unless the workload is very pop-heavy and there have been very many elastic *width* changes, the *Lateral* nodes should quickly stabilize and let the *push_width* and *pop_width* be equal.

3.5 Elastic Lateral-as-Window 2D Deque

The elastic LaW deque combines the ideas from the LaW queue and LaW stack, but is mainly similar to the LaW stack due to both having bidirectional *Win* shifts. However, as the deque can operate on both ends of the data structure, it requires one *Win* at the top and one at the bottom.

Algorithm 6 shows how the LaW deque shifts *Win*, but to avoid repetition, it only shows how to shift Win^{top}, as the methods for Win^{bot} are the mirrored equivalents. Enqueues and dequeues are done as in the LaW stack, finding a sub-deque within the current *Win* and updating the sub-deque with a linearizable enqueue or dequeue within the *Win*.

Algorithm 6: Elastic 2D LaW deque

```
6.1  function ShiftTopUp(old_win)
6.2      assert(old_win.shrinking ≠ UP);
6.3      if old_win.shrinking ≠ NO then
6.4          new_win ← old_win.Clone();
6.5          new_win.shrinking ← NO;
6.6          lat.TryReplaceTop(old_win,
               new_win);
6.7      else
6.8          new_depth ← depth_desired();
6.9          new_win ← Window {
6.10             below: old_win,
6.11             max: old_win.max +
                   new_depth/2,
6.12             depth: new_depth,
6.13             width: width_desired(),
6.14             shrinking: NO
6.15         };
6.16         lat.TryPushTop(old_win, new_win);

6.17 struct Window
6.18     Window* above;
6.19     Window* below;
6.20     int max;
6.21     uint depth;
6.22     uint width;
         // NO, UP, or DOWN
6.23     enum shrinking;

6.24 global Deque<Window> lat;
     // Conditional updates, linearizing
        only if the old top is unchanged.
6.25 method Deque.TryPushTop(old, new);
6.26 method Deque.TryPopTop(old);
6.27 method Deque.TryReplaceTop(old, new);

6.28 function ShiftTopDown(old_win)
6.29     assert(old_win.shrinking ≠ UP);
6.30     if old_win.shrinking = NO then
6.31         new_win ← old_win.Clone();
6.32         new_win.shrinking ← DOWN;
6.33         lat.TryReplaceTop(old_win, new_win);
6.34     else if old_win.below ≠ NULL then
6.35         assert(∀sub ∈ sub_deques[old_win.below.width... old_win.width]:
               sub.top_row < old_win.max − old_win.depth and sub.top_frozen);
6.36         assert(∀sub ∈ sub_deques[0...old_win.width]:
               sub.top_row ≤ old_win.below.max);
6.37         lat.TryPopTop(old_win);
```

As in the LaW stack, the LaW deque relies on all items in the deque always being within one *Win* in the *Lateral*, as described by Property 3.2. To achieve this, it employs the same method of shrinking the *Win* before dequeueing it (line 6.32), and freezing sub-deques outside Win_{width} during a shrinking *Win*, ensuring that no item is pushed outside the current *Win*. However, as a deque can push and pop on both ends, a *Win* can now shrink in either direction. However, due to the *Win* only starting to shrink when it is at an end of the *Lateral*, with another *Win* below or above, and stops shrinking when enqueueing an adjacent *Win* (line 6.6), it can only ever start shrinking from a non-shrinking state.

Property 3.2 (LaW Deque window cover)**.** *If item x is enqueued in sub-deque col_x at row row_x, and has not been dequeued, then $\exists Win^x \in Lateral$ such that $col_x < Win^x_{width}$ **and** $Win^x_{min} < row_x \leq Win^x_{max}$.*

Finally, to ensure correctness of the deque, some requirements are placed upon the *Lateral* deque, which can be achieved by modifying the Maged deque [27]. Firstly, it must support conditional *TryPush*, *TryPop*, and *TryReplace* methods for both deque ends, which take in the old expected end *Win*,

only linearizing if the end is unchanged. The *TryReplace* method is non-standard, and repalces the end *Win*, and is used for shrinking or un-shrinking the end *Win*. Secondly, the deque must be ABA aware when reading an end of the deque, so that it notices if another *Win* has been pushed and then popped between two reads. This also applies to the update methods *TryPush*, *TryPop*, *TryReplace*, which take in the current expected *Win*.

3.6 Elastic Lateral-plus-Window 2D Counter

The 2D counter stands out as the only non-queue 2D design, not allocating any nodes and instead only using the counters which in the queues associate items with rows. Beside incrementing and decrementing counts, much as the 2D stack, the counter also estimates the total counter size by multiplying a sampled sub-count by Win_{width}. Elastic extensions must ensure that they can estimate the total count efficiently, while allowing elastic changes in Win_{width}.

The elastic LpW counter adds a *Lateral* counter that tracks the offset of the true count to the sum of all counters within *Win*, as described in Property 3.3. When reducing Win_{width}, all counts outside Win_{width} are set to 0 and added to the *Lateral* count. Similarly, then increasing Win_{width}, all counts are set to the middle row in *Win*, and those counts are subtracted from the *Lateral*.

Property 3.3 (LpW Counter True Count)**.** *When* $Lateral.version = Win_{version}$*, all sub-counters outside* Win_{width} *have count* 0*, and the true count of the counter (number of linearized increments − decrements) is:* $Lateral.offset + \sum_i sub\_counter[i]$.

Algorithm 7 presents how to shift the window and synchronize the *Lateral* for the elastic LpW counter. In line with earlier LpW designs, shifting *Win* is done by creating a new *Win* with the desired dimensions, including saving the previous Win_{width} and incrementing a version count, finally changing the global *Win* with a CAS (line 7.13).

The *Lateral* is said to be synchronized with *Win* when $Lateral.version = Win_{version}$, and this is required when shifting *Win* or reading the counter. They become unsynchronized every *Win* shift (line 7.13), after which one must call *SyncLateral* to synchronize them. If $Win_{width} = Win_{last\_width}$, synchronizing the lateral only involves incrementing *Lateral.version*. Otherwise, synchronizing the *Lateral* is done in two parts that ensure the correctness of Property 3.3:

1. First, the *Lateral* offset change is calculated. If the Win_{width} has decreased, the offset increases by the sum of all counts outside Win_{width} (line 7.35), as these counts will be reset to 0. To avoid updates to the counters after these reads, the readers also increment the version count on the sub-counters. If Win_{width} has increased, the offset decreases by $Win_{mid} := Win_{max} - Win_{depth}/2$) multiplied by the change in Win_{width} (line 7.37), as those counters will be incremented to Win_{mid} before being operated on in the new window. This step linearizes by updating the *Lateral* on line 7.39, also setting *Lateral.syncing*, to flag that the *Lateral* is between the two synchronization steps.

Algorithm 7: Elastic 2D LpW counter

```
7.1  function Shift(old_win, dir)
7.2      new_win ← Window {
7.3          last_width: old_win.width,
7.4          width: width_desired(),
7.5          depth: depth_desired(),
7.6          version: old_win.version + 1,
7.7      };
7.8      new_max ← if dir = UP then
7.9          old_win.max + new_win.depth/2;
7.10     else
7.11         old_win.max − old_win.depth
              + new_win.depth/2;
7.12     new_win.max ← max(new_win.depth,
          new_max);
7.13     CAS(&win, old_win, new_win);
7.14     SyncLateral();

7.15 struct Window
7.16     uint max;
7.17     uint version;
7.18     uint depth;
7.19     uint width;
7.20     uint last_width;

7.21 struct Lateral
7.22     uint offset;
7.23     uint version;
7.24     bool syncing;

7.25 global Window* win;
7.26 global Lateral* lat;

7.27 function SyncLateral()
7.28     repeat
7.29         old_win ← win;
7.30         old_lat ← lat;
7.31     until old_win = win;
7.32     win_mid ← old_win.max − old_win.depth/2;
7.33     if old_lat.version < old_win.version & !old_lat.syncing then
             // Update Lateral offset count
7.34         offset ← if old_win.width < old_win.last_width then
7.35             sum of counters[old_win.widthwin_mid. old_win.last_width];
7.36         else
7.37             − win_mid ∗(old_win.width − old_win.last_width);
7.38         new_lat ← Lateral, offset: old_lat.offset − offset, version: old_lat.version,
              syncing: true};
7.39         CAS(&lat, old_lat, new_lat);
7.40     old_lat ← lat;
7.41     if old_lat.version < old_win.version then
7.42         for i ∈ {old_win.width... old_win.last_width} do
7.43             old_counter ← counters[i];
7.44             if old_win ≠ win return;
7.45             new_counter ← {count: 0, version: old_counter.version + 1};
7.46             CAS(&counters[i], old_counter, new_counter);
7.47         for i ∈ {old_win.last_width... old_win.width} do
7.48             old_counter ← counters[i];
7.49             if old_lat ≠ lat return;
7.50             new_counter ← {count: win_mid, version: old_counter.version + 1};
7.51             CAS(&counters[i], old_counter, new_counter);
7.52         new_lat ← Lateral {offset: old_lat.offset, version: old_win.version, syncing:
              false};
7.53         CAS(&lat, old_lat, new_lat);
```

2. Then, the potential sub-counters outside Win_{width} are set to 0 (line 7.46), or the new ones inside Win_{wdith} are set to Win_{mid} (line 7.51), counteracting the change in the *Lateral* above. This linearizes by updating the *Lateral* to a synchronized state at line 7.53.

Incrementing and decrementing a counter is done as in the static 2D counter, by reading a sub-counter, validating that it is within *Win*, additionally verifying that the *Lateral* is synchronized with *Win*, and then updating the sub-counter with a CAS. Reading the counter is done equivalently and returns $\textit{sub-count} \cdot \textit{Win}_{\textit{width}} + \textit{Lateral.offset}$.

4 Analysis

In this section, we prove the correctness and relaxation bounds for our elastic designs. Our elastic bounds give the worst-case rank errors, but if no elastic changes are made, our elastic designs have equal or lower error bounds compared to their static 2D counterparts.

Static k out-of-order relaxation is formalized by defining and bounding a *transition cost* (rank error) of the "get" methods within the linearized concurrent history [4]. Elastic relaxation allows the relaxation configuration to change over time, which will naturally change the bound k as well. Therefore, we define *elastic out-of-order* data structures similarly to static out-of-order, with the difference that the rank error bound is a function of the relaxation configuration history during the lifetime of the accessed item. In the simplest case, such as the elastic LaW queue from Section 3.2, the rank error bound for every dequeued item is a function of $\textit{Win}^{deq}_{width}$ and $\textit{Win}^{deq}_{depth}$ when the item is dequeued.

Lock-freedom To avoid repetitive arguments, we here collectively state that our designs are lock-free as (i) each sub-structure, the *Win*, and the *Lateral* are updated in a lock-free manner (by linearizing with a CAS), (ii) the *Lateral* is updated at most twice every window, and (iii) that there cannot be an infinite number of window shifts without progress on any of the sub-structures [5].

Memory usage The required memory of our designs is presented in Property 4.1. It is simple to generalize across designs, so we motivate it in the following paragraph. Importantly, as there is only an additive factor with the *width*, the memory overhead compared to a strict data structure is not very large.

Property 4.1 (Memory usage of Elastic designs)**.** *Our elastic queues, stacks, and deque, use $\mathcal{O}(n + w)$ memory, where n is the number of items in the data structure, and w is the largest width of all rows. The counter only uses $\mathcal{O}(w)$ memory.*

The memory usage of our designs is split into three parts: (1) sub-structures, (2) the *Lateral* (3) inserted items. Each sub-structure uses a small, constant amount of memory, when ignoring stored items. Thus, if keeping a dynamic vector of the number of sub-structures needed, they use $\mathcal{O}(w)$ memory. For the counter, as the sub-counters and *Lateral* are of $\mathcal{O}(1)$ size, its total size is $\mathcal{O}(w)$. Otherwise, the *Lateral* —in the worst case—has one *Lateral* node per occupied row in the data structure, which is $\mathcal{O}(n)$, where n is the current number of items. However, in most cases, as in the queues, the worst case is one *Lateral*

node per $Win_{width} \cdot Win_{depth}$ items, which is often quite large. Finally, each item is enqueued into a single sub-structure, using the same memory as a strict linked list of $\mathcal{O}(n)$, where n is the list size.

Notation When referencing item x, we denote $Win^{enq\ x}$ as the Win which was read at the enqueue of x, analogously defining $Win^{deq\ x}$. Furthermore, $Win_{field}^{max\ x}$ (or $Win_{field}^{min\ x}$) reference the maximum (or minimum) value of Win_{field} during the lifetime of x. Finally, row_x refers to the row where x was inserted and col_x to the sub-structure x was inserted.

4.1 Elastic LpW Queue

The monotonically increasing Win_{max} of the FIFO queues makes their analysis relatively straightforward. The main difficulty with the LpW queue is that Win_{depth}^{enq} does not necessarily equal Win_{depth}^{deq}, which means that $Win^{enq\ x}$ does not have to span the same rows as $Win^{deq\ x}$.

The proof idea of the rank error bound is that we in Lemma 4.2 prove that any changes in Win_{width}^{enq} are *first* inserted into the *Lateral*, and *then* appear in a new Win_{width}^{enq}. This essentially proves the *Lateral* is consistent. Using this, we essentially show that the worst rank error of dequeueing item x is when $Win_{max}^{enq\ x} = row_x = Win_{min}^{deq\ x} + 1$, and as $Win_{width}^{enq\ x} = Win_{width}^{deq\ x}$, this leads to a worst-case rank error of $(Win_{width}^{enq\ x} - 1)(Win_{depth}^{enq\ x} + Win_{depth}^{deq\ x} - 1)$.

To simplify notation, we introduce an ordering of the windows such that $Win^i < Win^j$ if $Win_{max}^i < Win_{max}^i$.

Lemma 4.2. *If a Lateral node l is enqueued at time t, then $Win_{max}^{deq} < row_l$ at t.*

Proof. We first note that $row_l \leftarrow Win_{max}^{enq} + 1$ when it is enqueued (line 2.30). Furthermore, the enqueue of l completes before the window shifts, and it cannot be enqueued again after the window has shifted as the comparison against the strictly increasing row count of the tail will fail (line 2.28). As $Win_{max}^{deq} \leq Win_{max}^{enq}$ (line 2.45), and $Win_{max}^{enq} < row_l$ at t, it holds that $Win_{max}^{deq} < row_l$ at t. □

Theorem 4.3. *The elastic LpW queue is linearizeable with respect to a k out-of-order elastically relaxed FIFO queue, where for every dequeue of x, $k = (Win_{width}^{enq\ x} - 1)(Win_{depth}^{enq\ x} + Win_{depth}^{deq\ x} - 1)$.*

Proof. Enqueues and non-empty dequeues linearize with a successful update on a MS sub-queue. An empty dequeue linearizes when $Win_{max}^{deq} = Win_{max}^{enq}$ after a double-collect where it sees all sub-queues within Win_{width}^{deq} empty. Lemma 4.2 together with $Win_{max}^{deq} = Win_{max}^{enq}$ gives that all nodes must be within Win_{width}^{deq}, meaning that empty returns can linearize at a point where the queue was completely empty.

The core observation for proving the out-of-order bound is to observe that the row counts for the sub-queues and the max of the windows strictly increase. For a node y to be enqueued before x, and not dequeued before x, it must hold

that $Win^{enq\ y} \leq Win^{enq\ x}$ and $Win^{deq\ y} \geq Win^{deq\ x}$. As $Win_{min}^{enq\ x} \leq Win_{max}^{deq\ x}$, we can bound the possible row of y by $Win_{min}^{deq\ x} \leq row_y \leq Win_{max}^{enq\ x}$. Using Lemma 4.2 together with the fact that Win^{deq} only spans rows with one *width* at a time (line 2.45), we get that these valid rows for y must have width $Win_{width}^{enq\ x} = Win_{width}^{deq\ x}$. As both $Win^{enq\ x}$ and $Win^{deq\ x}$ must share at least row_x, it holds that $Win_{min}^{enq\ x} \leq Win_{max}^{deq\ x}$. This is then used to limit the valid number of row_y by $Win_{max}^{enq\ x} - Win_{min}^{deq\ x} = Win_{min}^{enq\ x} + Win_{depth}^{enq\ x} - Win_{max}^{deq\ x} + Win_{depth}^{deq\ x} \leq Win_{depth}^{enq\ x} + Win_{depth}^{deq\ x} - 1$. As all operations within each sub-queue are ordered, we reach the final bound of $(Win_{width}^{enq\ x} - 1)(Win_{depth}^{enq\ x} + Win_{depth}^{deq\ x} - 1)$. □

Furthermore, one can follow the same arguments to prove that the *delay* (as introduced in Section 2) for the elastic LpW queue is bounded by the k in Theorem 4.3.

4.2 Elastic LaW Queue

Analyzing correctness for the elastic LaW queue becomes simple due to the strict order the *Lateral* windows define over all operations. Namely, for item x, as $Win^{enq\ x} = Win^{deq\ x}$, the enqueue and dequeue operations are totally ordered against all operations linearizing in other windows, and can only be out-of-order with respect to other operations in the same window. This leads to the simple bound in Theorem 4.4. As Win_{depth}^{deq} cannot change arbitrarily as in the LpW queue, the LaW queue gets a tighter worst-case bound than the LpW queue in Theorem 4.3.

Theorem 4.4. *The elastic LaW queue is linearizeable with respect to a k out-of-order elastically relaxed FIFO queue, where $k = (Win_{width}^{deq} - 1)\, Win_{depth}^{deq}$.*

Proof. The key observation is that every Win^{enq} must fill all Win_{depth}^{deq} rows of Win_{width}^{deq} items each before shifting to the next Win. These items can be enqueued in any order, except that each sub-queue is totally ordered. Enqueues to different Win are totally ordered due to the sequential semantics of the *Lateral*. The Win^{deq} uses the same Win structs as the Win^{enq}, by traversing the *Lateral* of past Win^{enq}, not shifting past such a window until it has also dequeued all its $Win_{width} \times Win_{depth}$ items.

Thus, as the oldest items in the queue always are in Win^{deq}, and dequeues only ever operate within Win^{deq}, together with the fact that the sub-queues are totally ordered, means that a dequeue can at most skip the first $(Win_{width}^{deq} - 1)\, Win_{depth}^{deq}$ items. □

As in the LpW queue, it is simple to see that the k in Theorem 4.4 also bounds the *delay* of the LaW queue.

4.3 Elastic LaW Stack

Bounding the rank error for the LaW stack becomes relatively simple thanks to Property 3.1, which guarantees that items are always within one of the *Win*

in the *Lateral*. Furthermore, bounding the rank error of an individual item x is helped by the key insight that any item pushed after x, but not popped before x, must have been pushed into a *Win* which contains x. Thus the error bound stretches Win_{depth} up and down for every sub-stack, potentially reordering x against the $Win_{width} - 1$ other sub-stacks.

Theorem 4.5. *The elastic LaW stack is linearizable with respect to a k out-of-order elastically relaxed stack, where k is bounded for every item x as $k = (Win_{width}^{max\ x} - 1)(2\,Win_{depth}^{max\ x} - 1)$.*

Proof. As the algorithm always tries to pop from all sub-stacks within Win_{width} before returning empty, that it linearizes with a CAS like the Treiber stack [26], and that Property 3.1 gives that there are no items not contained in a *Win* inside *Lateral*, it is obvious that the design is correct with respect to pool specifications.

To bound the number of items pushed after x, but not popped before x, we will bound on what rows they can be pushed. If an item y is pushed during the lifetime of x, then Property 3.1 gives that it must be pushed in a window $Win^{push\ y}$ where $Win_{max}^{push\ y} \geq row_x$. Furthermore, when popping x, Property 3.1 gives that no items are above $Win_{max}^{pop\ x}$. Thus, the possible range of allowed rows for items pushed after x, but not popped before x, spans $Win_{depth}^{pop\ x} + Win_{depth}^{push\ y} - 1$ rows where the -1 comes from the two windows overlapping with at least row_x. As there are no out-of-order items in col_x, the maximum error is bounded by $(Win_{width}^{max\ x} - 1)(2\,Win_{depth}^{max\ x} - 1)$. □

Another way to view the relaxation errors in the LaW stack it to logically relax both pushes and pops, instead of just pops, which is the norm [4]. Using similar arguments as above, one would then see that the worst-case rank errors would be $(Win_{width}^{push\ x} - 1)\,Win_{depth}^{push\ x}$ for the push and $(Win_{width}^{pop\ x} - 1)\,Win_{depth}^{pop\ x}$ for the pop. This alternate formulation might be more useful for describing the actual relaxation in the stack.

4.4 Elastic LpW 2D Stack

Analyzing the LpW stack is more involved than analyzing the simpler LaW stack. The idea is to first prove that the *Lateral* correctly bounds row widths in Lemma 4.6, then bound the size of sub-stacks in Lemma 4.7 and 4.8, and finally derive the rank error bound in Theorem 4.9.

For brevity, we denote $Win_{shift} \coloneqq \lfloor Win_{depth}/2 \rfloor$ and the top row (size) of sub-stack j as N_j. Furthermore, we introduce a *width bound* ($width_r$) for each row r as $l.width$ if there exists a *Lateral* node $l, l.next.row < r \leq l.row$, or Win_{push_width} if $r > l.row\ \forall\ l$ (if this properly bounds the row widths, the *Lateral* is *consistent*). Due to *Lateral* nodes being removed from the stack if their row overlaps the next node's row, this width bound is uniquely defined.

Lemma 4.6. *At the moment preceding the linearization of each window shift, it holds for each row r and every item x where$row_x = r$, that $width_r \geq index_x$.*

Proof. Informally, this lemma states that the *Lateral* stack properly bounds all rows with widths greater than Win_{push_width} after the call to *Stabilize* at line 5.52. We prove this with induction over the sequence of window shifts, and it is easy to see that it holds at the first window shift, as all nodes will be pushed within Win_{put_width}. Now, if the lemma held at the previous window shift, we check if it continues to hold for the next shift where we shift from Win^i.

First, we inspect rows at and below Win^i_{min} and note that during the lifetime of Win^i, all nodes are pushed above Win^i_{min}, and $width_r$ will not be changed at rows at or below Win^i_{min} (lines 5.27, 5.31, 5.34) from lowering a *Lateral* node. Thus, if $Win^i_{push\_width} = Win^i_{last\_push\_width}$, the lemma continues to hold for all rows lower or equal to Win^i_{min}. If the $Win^i_{push\_width} \neq Win^i_{last\_push\_width}$ a new *Lateral* node can be inserted with width $Win^i_{last\_push\_width}$. This node changes row bounds for rows at or below Win^i_{min} if there was no other *Lateral* node above Win^i_{min} at the shift to Win^i, and in that case those rows would have before Win^i been bounded by $Win^i_{last\_push\_width}$, which is the same as the width bound this node enforces. Therefore, the induction invariant continues to hold for rows at or below Win^i_{min}.

Now we inspect rows above Win^i_{min} to see if the invariant also holds there. Firstly, no row will have a width bound smaller than Win_{push_width}, as smaller widths will be lowered or inserted to or below Win^i_{min} (lines 5.31, 5.58). Therefore, only items above Win^i_{min} outside $Win^i_{push\_width}$ can break the invariant. In the lowering phase, *Lateral* nodes $l, l.width > Win^i_{last\_push\_width} \wedge l.width > Win^{push\_width}$ are lowered iff the shift to Win^i was downwards, as then all sub-stacks outside $Win^i_{last\_push\_width}$ were seen at the bottom of the last window. Thus, if $Win^i_{push\_width} \geq Win^i_{last\_push\_width}$ no nodes could have been pushed outside $Win^i_{push\_width}$ since the shift to Win^i, and as all widths bound held then, they will hold at the shift from Win^i. Otherwise, if $Win^i_{push\_width} < Win^i_{last\_push\_width}$, every row $r, r \leq l.row$ for the topmost *Lateral* node l must have a valid bound, and the rows above $l.row$ were before Win^i bounded by $Win^i_{last\_push\_width}$ which is smaller than the new bound $Win^i_{push\_width}$, so the width bounds must hold for all rows in this case as well. □

Lemma 4.7. *If x lives on the stack during $'x$, then for any item y pushed during $'x$, $row_y \geq Win^{push\ x}_{min} - Win^{max\ x}_{shift}$.*

Proof. This is proved by contradiction. Assume x was pushed at time t_x and there exists an item $y : row_y \leq Win^x_{min} - Win^{max\ x}_{shift}$, pushed at time $t_y > t_x$. Call the point in time where a thread shifted the window to $Win^{push\ y}$ as t_s.

- If $t_s < t_x$, then as $t_y > t_x$, x must be pushed during $Win^{push\ y}$. However, t_x can't be during $Win^{push\ y}$, as an item is pushed at or below Win_{max}.
- If instead $t_s > t_x$, we call the window before $Win^{push\ y}$ as Win^z. For a thread to shift from Win^z, it must have seen $\forall j, N_j = Win^z_{min}$ (as Lemma 4.6 shows iterating over Win_{pop_width} is the same as iterating over all j) at some point t_z (set t_z as the time it started this iteration) during Win^z.

As $Win^z_{min} < row_x$, $t_x > t_z$, which means that t_x must have been during Win^z, as we above showed $t_x < t_s$. This is impossible as during Win^z, items are pushed at or below Win^z_{max}.

□

Lemma 4.8. *If x lives on the stack during $'x$, then for any item y pushed, but not popped, during $'x$, $row_y \leq Win^{pop\ x}_{max} + Win^{max\ x}_{shift}$.*

Proof. Similarly to Lemma 4.7 this is proved by contradiction. Assume that x was pushed at t_x, popped at time t'_x and that there exists an item $y, row_y > Win^{pop\ x}_{max} + Win^{max\ x}_{shift}$ pushed at t_y and not popped at t'_x, where $t_x < t_y < t'_x$. Call the point in time where a thread shifted to $Win^{pop\ x}$ as t_s.

- If $t_s < t_y$, then y must have been pushed in the same window as x was popped. But items are only pushed at or below Win_{max}, which contradicts the assumption, as y is pushed too low.
- If $t_y < t_s$, we call the window proceeding $Win^{pop\ x}$ as Win^z. For a thread to shift from Win^z, it must have seen $\forall j, N_j = Win^z_{min}$ at some point t_z (set t_z as the time it started the iteration, seeing the first $N_j = Win^z_{min}$) during Win^z. Therefore, $t_z < t'_y$, which is impossible as $Win^z_{max} < row_y$ and $t_y < t_s$.

□

Theorem 4.9. *The elastic LpW stack is linearizeable with respect to a k out-of-order elastically relaxed stack, where k is bounded for every item x as $k = (Win^{max\ x}_{width} - 1)(3\,Win^{max\ x}_{depth} - 1)$.*

Proof. Assume that y is pushed after x and not popped before x. Then Lemma 4.7 and 4.8 gives $Win^{push\ x}_{min} - Win^{max\ x}_{shift} \leq row_y \leq Win^{pop\ x}_{max} + Win^{max\ x}_{shift}$. As each sub-stack is internally ordered and the maximum number of items pushed after x and not popped before x becomes $(Win^{max\ x}_{push\_width} - 1)(Win^{pop\ x}_{max} - Win^{push\ x}_{min} + 2\,Win^{max\ x}_{shift}) \leq (Win^{max\ x}_{push\_width} - 1)(3\,Win^{max\ x}_{depth} - 1)$. □

As for the LaW stack, one can reformulate this rank error bound such that both push and pop operations are relaxed, with worst-case bounds of $(Win^{push\ x}_{last\ width} - 1)\,Win^{push\ x}_{last\ depth} + (Win^{push\ x}_{width} - 1)\,Win^{push\ x}_{depth}/2$ for the push and $(Win^{pop\ x}_{last\ width} - 1)\,Win^{pop\ x}_{last\ depth} + (Win^{pop\ x}_{width} - 1)\,Win^{pop\ x}_{depth}/2$ for the pop, where $Win_{last\ width}$ and $Win_{last\ depth}$ is the value of the respective dimension in the previous Win. The reason the bounds for the LpW stack are looser than the LaW stack is that it does not shrink Win before shifting, potentially linearizing in the previous Win at times.

4.5 Elastic LaW 2D Deque

The rank error of when dequeueing x from one end of a deque becomes the number of other items pushed to the same end after x, but not popped before x. Even though we can now push items from both ends of the deque, using

Property 3.2 instead of Property 3.1, proving the following correctness theorem of the LaW deque follows the same path as the LaW stack. We therefore omit the proof, and refer the reader to the proof of Theorem 4.5.

Theorem 4.10. *The elastic LaW deque is linearizable with respect to a k out-of-order elastically relaxed deque, where k is bounded for every item x as $k = (Win_{width}^{max\ x} - 1)(2\,Win_{depth}^{max\ x} - 1)$.*

4.6 Elastic LpW 2D Counter

The relaxation bound for the LpW counter is slightly different from the queues, as we the out-of-order error for the counter is defined as the absolute difference between the returned count and the true count, contrary to the ordering of items. The idea is to use Property 3.3 to restrict the area of concern to columns within Win_{width}, and then see that sub-counts must be within the current, or previous Win, bounding the maximum difference between any two sub-counters.

Theorem 4.11. *The elastic LpW counter is linearizable with respect to a k out-of-order elastically relaxed counter, where $k = (Win_{width} - 1)(Win_{depth}/2 + Win_{depth}^{last} - 1)$, where Win^{last} is the previous Win.*

Proof. Using Property 3.3, and assuming we sample sub-counter i for the read, the relaxation error of the read operation becomes $|\sum_j^{Win_{width}}(count_i - count_j)| \leq \sum_j^{Win_{width}} |count_i - count_j| \leq (Win_{width} - 1) max_j |count_i - count_j|$.

As we only consider sub-counts within Win_{width}, the maximum difference between any two sub-counters can be proven in the same was as in the static 2D case, but also considering elastic Win_{depth}. Lemma 4 in the static 2D paper [5] proves that all counts are always in two adjacent windows, and with elastic Win_{depth}, this can easily be adapted to state that all sub-counts are within Win^{last} or Win. As the shift from Win^{last} to Win is $Win_{depth}/2$, the maximum difference between any two sub-counts becomes $Win_{depth}/2 + Win_{depth}^{last} - 1$. □

5 Experimental Evaluation

In this section, we first evaluate the overhead of our elastic queues, stacks, and counters, when not using their elastic capabilities, after which we examine how these elastic capabilities can be utilized in dynamic executions. All experiments run on a 128-core 2.25GHz AMD EPYC 9754 with two-way SMT, 256 MB L3 cache, and 755 GB RAM. The machine runs openSUSE Tumbleweed with the 6.13.1 Linux kernel. All experiments are written in C and compiled with gcc 13.2.1 at its highest optimization level, using pthreads for parallelization. Threads are pinned in a round-robin fashion between core clusters, starting to use SMT after 128 threads.

Our elastic 2D implementations build on an optimized version of the static 2D framework [5]. We use SSMEM [29] for efficient epoch-based memory management [30] of our dynamic memory. Furthermore, to facilitate fair

comparison against the static 2D designs, we don't change the used sub-structures, and thus use the 16-byte CAS for the counted head and tail pointers in the queues and stacks. Our implementations, including scripts to re-run all experiments, are available in the Zenodo repository [19].

5.1 Static Relaxation

To understand the potential overhead of our elastic extensions, we compare their scalability, during constant relaxation, against that of state-of-the-art k out-of-order and strict concurrent designs. For the queues, we select the static 2D queue [5] and the k-segment queue [7] as k out-of-order designs. Furthermore, we select the LCRQ [24] and the YMC queue [25] as the state-of-the-art strict FIFO queues. For the stacks, we select the 2D stack [5] and the k-segment stack [4] as k out-of-order designs, the lock-free elimination-backoff stack [31] as an efficient strict design, and the Treiber stack [26] as a baseline. For the counters, we select the 2D counter [5] as the relaxed design, and two simple counters implemented on top of fetch-and-add (FAA) and CAS for our strict designs. The counters are strictly non-negative, but the simple FAA counter violates this due to the nature of FAA, giving it an advantage when the other counters have to do emptiness checks. All data structures were implemented in our framework using SSMEM [29] for memory management, with external non-trivial designs based on their respective paper's implementation.

We use a benchmark where threads repeatedly perform insert or remove operations at random, each with equal probability, for a duration of one second. Test results are averaged over 10 runs, with standard deviation included in the plots. The benchmark requires specifying the maximum rank error of each relaxed data structure, and as the optimal choice of Win_{width} and Win_{depth} is application-specific [5], we set $Win_{width} = 2 \times$ `nbr_threads`, and use the maximum Win_{depth} which does not violate the bound. This choice of Win dimensions gives acceptable scalability, and we saw the same trends when using other combinations.

Accurately measuring rank errors without altering their distribution is an open problem, and we adapt a common method in the literature [5], [17], [32] of encapsulating the linearization points of all methods in a global lock, giving a total order of operations. This order can be used to re-run the execution with a strict data structure, where the rank error of each operation easily can be calculated. For queues and stacks, the rank error of dequeuing x is the distance between x and the tail or top item, and the rank error for counters becomes the absolute difference between the true count at each operation and the returned count.

Figure 4 presents how the elastic designs scale with threads, including both settings with pre-filled and empty data structures. Pre-filling drastically changes the dynamics of dequeues, as they don't have to do potentially costly emptiness checks. However, the throughput trends are mostly identical irrespective of prefill. The largest difference when not using pre-fill is that the rank errors become smaller, as they naturally cannot exceed the queue size at any time. All experiments show that the elastic designs have close to zero overhead when

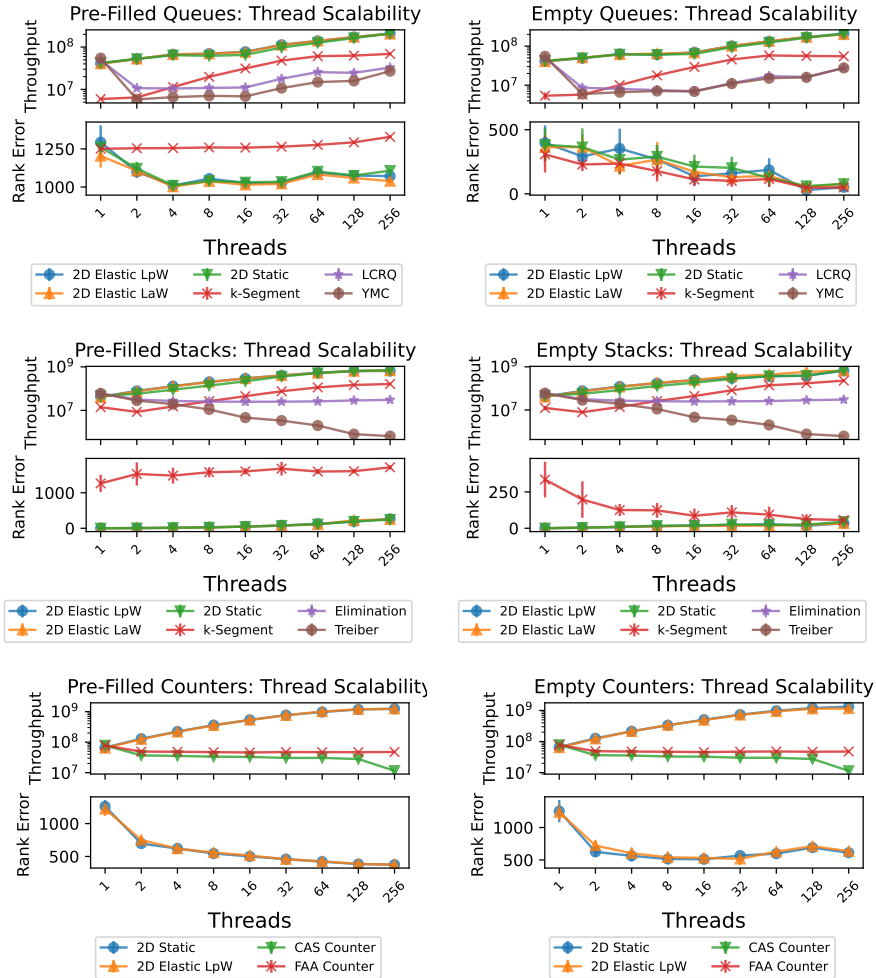


Figure 4: Scalability of throughput and mean rank error with increasing number of threads, using a rank error bound of $k = 5 \times 10^3$. Plots in the left column have a pre-fill of 10^6, and plots in the right column do not use any pre-fill.

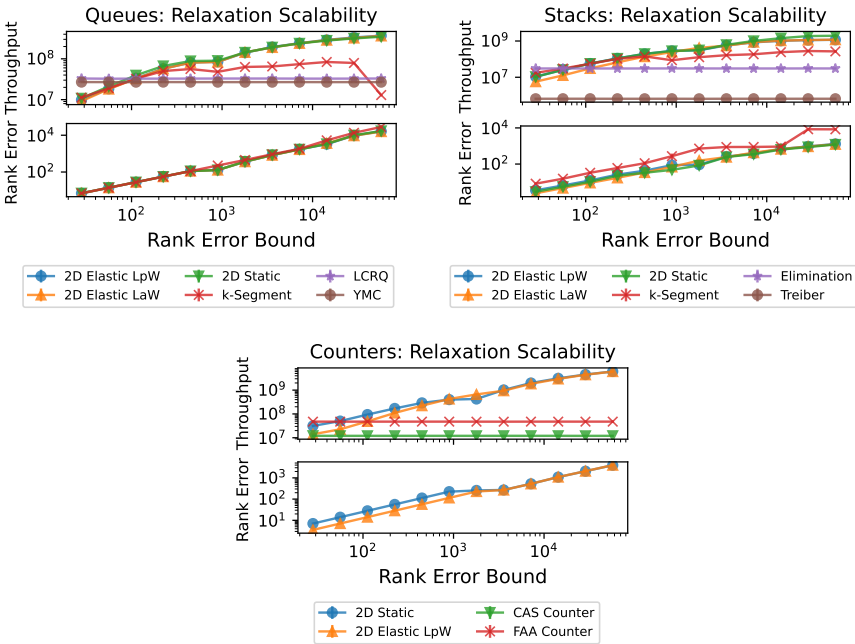


Figure 5: Scalability of throughput and mean rank error as rank error bound increases, using 256 threads and 10^6 pre-fill.

compared to the static 2D designs. Similarly, the elastic designs don't show any increase in rank errors when compared to the static designs. Overall, the elastic designs scale as well as the static 2D framework, and out-scale the other data structures, highlighting the light weight of the *Lateral*.

Figure 5 further shows how the designs scale with increasing rank error bounds. Here we pre-fill the data structures with 10^6 items, as the throughput is not that affected as seen in Figure 4, and the designs otherwise just become pools. The elastic LaW stack and elastic LpW counter have slight overheads in throughput at lower relaxation levels. The reason for this is that the *Lateral* incurs some overhead when shifting *Win* in these designs, and such shifts happen more often at lower relaxation levels. However, overall, the elastic 2D designs scale monotonically with relaxation, mostly performs identically to the static 2D designs, and outperforms other designs as relaxation increases.

5.2 Elastic Relaxation - Dynamic Controller

In this seciton, we evaluate the usefulness of our elastic extensions in dynamic workloads with varying degrees of parallelism. We focus on the LaW queue, as relaxed FIFO queues are more popular in the literature than stacks or counters, and as we think the simplicity of the LaW design makes it preferable over the LpW queue. As the optimal Win_{width} is tightly coupled to the degree of parallelism, we design a lightweight controller to dynamically adjust Win_{width}

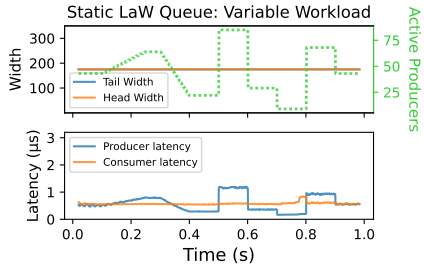
based on measured contention. We first examine how the controller reacts over time in a dynamic producer-consumer micro-benchmark, and then how it can provide end-to-end benefits in a parallel BFS algorithm with varying number of threads.

The dynamic controller should balance relaxation errors with latency, and a lightweight control signal for this is the failure rate of sub-structure CAS linearizations [32]–[34]. So the controller should strive to keep the failure rate constant, while being stable, responding quickly, and imposing low latency overhead. The conference version of this paper [33] used simple threshold-based thread-local controllers, similar to CA-PQ [34], but such simple and local controllers have a hard time reacting fast without significantly overshooting and being unstable. Instead, we here use a shared wait-free controller inspired by leaky proportional-integral controllers from control theory [35]. The controller keeps an exponential moving average (EMA) of the total sub-queue CAS failure rate, which provides an accurate an reasonably up-to-date estimate of the true failure rate. To reduce overhead, the threads only update this EMA every 200 operations, and if an update fails due to contention, it is aborted. The controller can be seen as a leaky integrator with a small proportional factor.

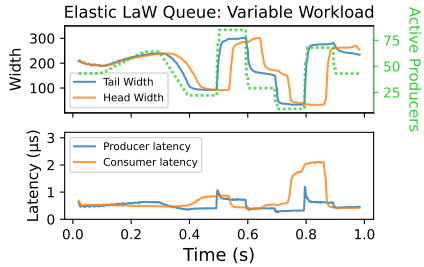
The controller has a target T, which is the desired fraction of failing sub-queue CAS. When a thread synchronizes its state with the controller, it computes its local failure rate F and logarithmic error $E_t = \ln(F/T)$ since the last synchronization, updating the global error E_g as $E_g = \alpha E_t + (1-\alpha)E_g$, which is the EMA of failures. When shifting Win^{enq}, its width w_i is updated from the previous width w_{i-1} based on the global error: $\ln(w_i) = \ln(w_{i-1}) + K_p E_g \Leftrightarrow w_i = w_{i-1}\exp(K_p E_g)$, where K_p is a gain constant. The EMA lives in the log domain as its updates then become symmetric w.r.t. percentage deviations from T. Finally, to reduce windup, E_g is reduced by 25% after each change to Win^{enq}_{width}. In our experiments, we use $\alpha = 0.01, K_p = 0.01, T = 0.012$. Configuring T can easily be done by finding the failure rate of good configurations in static executions. Varying α between 0.1 and 0.005 did not cause a significant difference in our experiments. Finally, K_p is the most important to tune, as a too large value can cause instability, and we therefore used this relatively low value.

5.2.1 Producer-Consumer Micro-Benchmark

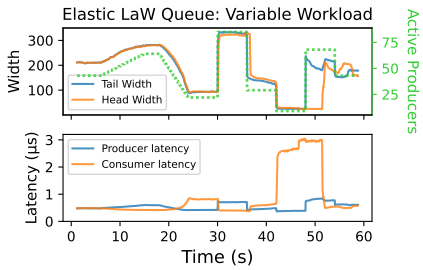
To evaluate the controller's ability to adapt to workload changes, we design a micro-benchmark that emulates a dynamic producer-consumer scenario. It consists of a shared relaxed FIFO queue accessed by 128 threads. One third of the threads (42) act as consumers, continuously attempting to dequeue tasks. The remaining 86 threads act as producers, repeatedly enqueuing tasks. Furthermore, the number of active producers varies over time, as shown in Figure 6. The queues are initially empty. This setup models a high-contention server-side task queue, where a fixed pool of worker threads (consumers) handles a fluctuating stream of external requests (producers). The benchmark tests whether the controller can dynamically adjust Win^{enq}_{width} to maintain a suitable balance between relaxation and latency under shifting load.



(a) Static Relaxation, 1 second



(b) Dynamic Relaxation, 1 second



(c) Dynamic Relaxation, 1 minute

Figure 6: Latency, width, and active producers over time, in a producer-consumer benchmark where the number of producers vary over time. All plots use the LaW queue, but (a) uses static relaxation, while the others use the dynamic controller to keep the producers' contention stable.

Figure 6 shows the average operational latency for consumers and producers, as well as the Win^{enq}_{width} and Win^{deq}_{width}, averaged over 10 runs, both for static Win_{width} and when the producers use the dynamic controller. It uses $Win_{depth} = 8$ and the static Win_{width} was set to 180, the stable width of the controller when half the producers are active. When using static width, the producers' latency clearly scales with the number of active producers, while the consumer latency is mostly constant. When using the controller, the Win^{enq}_{width} quickly adapts to the number of active producers, which in turn makes its producers' latencies more stable than the static queue. Notably, as we only directly change Win^{enq}_{width}, the change must propagate through the queue for it to affect Win^{deq}_{width}, delaying the Win^{deq}_{width} change. Furthermore, the latencies of the producers and consumers are not completely independent, as for example seen around 53 seconds in Figure 6(c) where the increase in Win^{deq}_{width} causes the consumers' latencies to decrease, which in turn increases the producers' latencies and contention, in turn also increasing Win^{enq}_{width}. This dependence of different aspects of the system highlights the difficulty of choosing a good control signal for the controller.

The results in Figure 6 shows that the controller is quick, stable, has reasonably low overhead, and mostly adjusts the Win^{enq}_{width} to be roughly

Table 1: Thread workloads for the BFS benchmark. They describe the fraction of active threads over time (t), where t is normalized to $[0, 1)$, after which the pattern repeats.

Name	Definition	Description
Const.	1	Constantly max threads
MJ	See Figure 7	Google queries, 1 day
Wikipedia	See Figure 7	Wikipedia queries, 1 week
Lin. Inc.	t	Linear increase
Lin. Dec.	$1 - t$	Linear decrease
Exp. Inc.	$\frac{1-e^{5t}}{1-e^5}$	Exponential increase
Exp. Dec.	$1 - \frac{1-e^{5t}}{1-e^5}$	Exponential decrease

proportional to the number of active producers. One disadvantage is how the size of Win^{deq}_{width} seems to affect the contention at the enqueues, for example leading to Win^{enq}_{width} not dropping as fast or much as we would expect at 0.9 seconds in Figure 6(b), or suddenly increasing without the nubmer of threads changing after 52 seconds in Figure 6(c). However, this is hard to avoid when basing the controller on CAS contention, and we anticipate it will not pose large issues in practice.

The most important consideration when using such a controller is defining the desired behavior. In this synthetic scenario, we aimed at stabilizing producer latency effectively minimizing the relaxation of incoming user requests, while still keeping user latency acceptable under high load. Since the number of consumers remained constant and they performed no elastic changes, their latency was not prioritized and occasionally rose to as much as 3 μs. If consumer latency is also a concern, they could be included in the controller's feedback loop, or a lower bound could be placed on Win^{enq}_{width} to prevent excessive spikes in their latency.

Table 2: Graphs used in the BFS benchmarks.

Graph	Nodes	Edges	Graph Type
USA (road_usa [36])	$24M$	$58M$	road
EU (europe_osm [36])	$51M$	$108M$	road
GL5 (geolife5_sym [37], [38])	$25M$	$309M$	kNN

5.2.2 Breadth-First Search Macro-Benchmark

To evaluate the effectiveness of our elastic queue designs and dynamic controller in a realistic algorithmic setting, we implement a concurrent breadth-first search (BFS) benchmark in a setting with dynamically accesible parallelism. BFS is a representative example of a parallel algorithm that uses relaxed queues as a shared work-list, where balancing throughput and ordering accuracy is critical. In such algorithms, increasing relaxation may improve scalability but

		Const.		MJ		Wikipedia		Lin. Inc.		Lin. Dec.		Exp. Inc.		Exp. Dec.		GMean	
		Time	Work	Time	Work	Time	Work	Time	Work	Time	Work	Time	Work	Time	Work	Time	Work
USA	Static	664	2.48	1501	2.96	815	2.60	979	3.01	735	2.44	1864	2.94	667	2.46	956	2.69
	Dynamic	**549**	**2.11**	**1012**	**1.63**	**621**	**1.84**	**673**	**1.77**	**659**	**1.96**	**1300**	**1.54**	**582**	**2.06**	**735**	**1.83**
EU	Static	1171	**1.86**	**2006**	1.92	1220	**1.89**	1333	1.97	**1207**	**1.88**	2484	1.94	1104	**1.86**	1437	1.90
	Dynamic	**919**	2.12	2076	**1.68**	**1172**	1.97	**1073**	**1.73**	1226	2.04	**1964**	**1.44**	**967**	2.03	**1279**	**1.85**
GL5	Static	**463**	3.07	1197	4.38	**635**	3.64	789	4.25	**550**	3.27	1528	4.32	479	3.10	731	3.68
	Dynamic	499	**2.74**	**1027**	**2.82**	655	**3.08**	**581**	**2.46**	661	**2.96**	**1147**	**2.30**	**472**	**2.46**	**684**	**2.67**
AVG	Static	711	2.42	1533	2.92	858	2.62	1010	2.93	**787**	2.47	1920	2.91	707	2.42	1001	2.66
	Dynamic	**631**	**2.31**	**1292**	**1.98**	**781**	**2.24**	**749**	**1.96**	811	**2.28**	**1431**	**1.72**	**643**	**2.17**	**863**	**2.08**

Table 3: Runtime (in ms) and work, lower is better, for a concurrent BFS algorithm using the 2D LaW queue with either statically $Win_{width} = 1024$, or using the dynamic controller. Thread workloads (columns) defined in Table 1 and the graphs (rows) in Table 2. The last column is the geometric mean of all workloads, and similarly the last row is the geometric mean over all graphs.

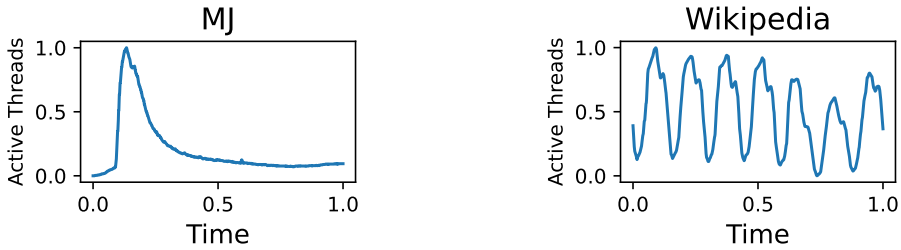


Figure 7: Realistic workloads used in BFS benchmarks. The MJ workload (left) is the number of Google searches of Michael Jackson the day of his death, from 13pm PDT [39]. The Wikipedia workload (right) is the number of search queries on the English Wikipedia over one week in 2009 [40].

can reduce work efficiency due to redundant or out-of-order processing [11], [14]. Our goal is to assess whether dynamic control over relaxation can yield end-to-end benefits in this more realistic scenario.

The benchmark uses a relaxed FIFO queue as a work-list [14], and is analogous to how Dijkstra's algorithm is parallelized with relaxed priority queues [11], [17]. We compare the performance of the LaW queue with the dynamic controller to the static 2D queue across a range of workloads. These workloads, shown in Table 1, include constant, increasing, and decreasing thread counts, as well as real-world load traces, shown in Figure 7. Each workload runs for half a second before repeating. We measure both total runtime and total *work*, defined as the number of successful enqueues divided by the number of uniquely enqueued items (vertices).

Relaxed priority queues have been shown to perform very well for parallel SSSP over high-diameter sparse graphs [11]–[13], [16], so we focus on such graphs. The graphs are summarized in table 2. Firstly, we use the EU and USA road graphs [36], which are commonly used road graphs [11], [13], [14]. Second, we use the GL5 kNN graph [37], [38], whose irregular nature made it a difficult graph for many participants in the 2025 FastCode Pogramming Challenge on parallel SSSP [16].

Table 3 shows the runtime and work across the selected graphs and workloads, where all results are averaged over 100 runs. We searched power of 2 combinations for the optimal *width* and *depth* configuartion for the static queue, and found that $Win_{width} = 1024$ and $Win_{depth} = 8$ had the best geometric mean (gmean) runtime across all graphs and workloads. The dynamic queue therefore also used $Win_{depth} = 8$, and its contention target T was chosen so that the average *width* during the constant workload had a gmean of 1024 across the three graphs, which was 0.012.

The results show that the dynamic controller in the majority of these benchmarks reduce runtime, with the gmean across all graphs and tests being 13% faster and 28% more work-efficient. Furthermore, the dynamic LaW is faster in 16, and more work-efficient in 17, out of the 21 benchmarks. The dynamic relaxation is the most beneficial in the *Lin. Inc.* and *Exp. Inc.*

workloads, with a gmean speedup of 35% and 34% respectively. On the contrary, its worst workload is the *Lin. Dec.*, where it has a 3% gmean slowdown, the only workload with a gmean slowdown.

The impact of relaxation on work efficiency in concurrent BFS remains poorly understood. Our results indicate that the early phase of execution plays a critical role where large relaxation errors at the start can compound over time, leading to significant work inefficiencies. Unlike relaxed priority queues, relaxed FIFO queues lack self-stabilizing ordering mechanisms [11]–[13], making them especially sensitive to early misordering. This may explain why the dynamic queue outperforms the static design in the *Const.* workload, with a lower gmean of work, and why increasing workloads yield better results than decreasing ones.

In summary, using our elastic LaW queue with the dynamic controller outperforms a statically configured 2D queue in the BFS benchmark with varying levels of parallelism. A key advantage of this approach is that it eliminates the need to know the number of accessing threads in advance, instead only a contention target must be specified. This target is a single tunable parameter that can also be adjusted at runtime. As a result, we believe our elastic designs, together with a dynamic controller, are better suited for integration into long-running systems—where workload characteristics may change over time—than static designs.

6 Related Work

One of the earliest uses of relaxed data structures is from 1993 by Karp and Zhang [41]. However, it was not until more recently that relaxed data structures garnered stand-alone interest as a promising technique to boost parallelism [3]. Relaxed designs have shown exceptional throughput on highly parallel benchmarks [5], [6], [11], [42], have proven suitable in heuristics for graph algorithms [11], [12], [43], and been theoretically analyzed in e.g. [4], [44]–[46].

Henzinger et al. formalized *quantitative relaxation* [4] to define relaxed data structures with a rank error bound. They also introduce the relaxed k-segment stack, which builds upon the earlier relaxed k-FIFO queue [7]. Their theoretical framework is easy to extend to encompass elastic relaxation by allowing the rank error bound to vary over time, and it is simple to elastically extend their designs using the *Lateral*, as their k-designs can be modeled within the 2D framework.

Rather than deterministically bounding rank error, some designs achieve better performance by only giving probabilistic error guarantees [6], [11], [14], [44]. The MultiQueue is a relaxed priority queue that has proven effective in for example shortest-path graph algorithms [11]–[13], [16]. It enqueues items into random sub-queues and dequeues the highest-priority item from a random choice of two sub-queues. Similar strategies have been applied to FIFO queues [6], [14] and stacks. The probabilistic rank error guarantees of the MultiQueue have been analyzed both with a strong potential argument [44], [46] and with Markov chains [47], but its errors are still not completely understood.

The SprayList [32] is another probabilistically relaxed priority queue. Experimentally, the SprayList is outperformed by the MultiQueue [11] but it is, to the best of our knowledge, the only design in the literature that can reconfigure relaxation well during run-time, which it does by adjusting thread-local parameters based on contention.

The CA-PQ [34] is another relaxed priority queue, again outperformed by the MultiQueue [11], which can be seen as elastic, as threads toggle between synchronizing with the global state and working locally, depending on contention. They use a simple thread-local controller, similar to the one in the conference version of this paper [33], which works well because (1) synchronization is toggled per thread, and (2) only on/off decisions are made. In contrast, our shared controller from Section 5.2 better supports global configurations such as *Win* dimensions, and can prevent divergence in per-thread settings when the configuration space is larger.

Relaxed priority queues have been widely used to parallelize single-source shortest path (SSSP) graph traversals [11]–[13], [16], [34]. In the recent Fast-Code Programming Challenge at PPoPP 2025 [48], the winning contribution in the parallel SSSP track used the MultiQueue [11] to achieve strong results on sparse input graphs [16]. Strict and relaxed FIFO queues have also been proposed for parallel BFS [14], [49], though their performance has yet to surpass level-synchronous algorithms [50]. The work-efficiency of relaxed queues in such graph traversals remains largely unstudied and is an interesting open question.

7 Conclusion

We have presented the concept of elastic relaxation for concurrent data structures, and extended the 2D framework to incorporate elasticity. Using the *Lateral* component, we proposed the simple LaW designs and more involved LpW designs. If incorporating the *Lateral* into designs outside the 2D framework, we recommend the LaW paradigm for its simplicity and efficiency, but recognize that the LpW design might be more suitable for some data structures, such as the 2D counter. Our implementations offer worst-case bounds and match state-of-the-art performance during periods of constant relaxation, while also supporting dynamic reconfiguration of relaxation at runtime. We introduced a lightweight dynamic controller for relaxation, based on the exponential moving average of per-thread CAS contention, which efficiently trades relaxation for latency in real time. Its performance was evaluated both over time in a producer-consumer benchmark and in a BFS macro-benchmark, where it demonstrated improved throughput and work efficiency in several dynamic workloads compared to a statically relaxed queue. We believe elasticity to be essential for relaxed data structures to become practically viable, and see this paper as a first step in that direction.

As future work, we find it of interest to incorporate elastic relaxation into other data structures, such as relaxed priority queues. As relaxed priority queues have already proven state-of-the-art when it comes to parallel SSSP on

some graphs, we find it of interest to see if elastic relaxation could help further increase their performance in similar graph traversal benchmarks.

Bibliography

[1] M. Herlihy, N. Shavit, V. Luchangco and M. Spear, *The Art of Multiprocessor Programming*. Elsevier Science, 2020.

[2] F. Ellen, D. Hendler and N. Shavit, 'On the inherent sequentiality of concurrent objects,' *SIAM Journal on Computing*, vol. 41, no. 3, pp. 519–536, 2012. DOI: 10.1137/08072646X.

[3] N. Shavit, 'Data structures in the multicore age,' *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011. DOI: 10.1145/1897852.1897873.

[4] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin and A. Sokolova, 'Quantitative relaxation of concurrent data structures,' *SIGPLAN Not.*, vol. 48, no. 1, pp. 317–328, 2013. DOI: 10.1145/2480359.2429109.

[5] A. Rukundo, A. Atalar and P. Tsigas, 'Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework,' in *33rd International Symposium on Distributed Computing (DISC 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 146, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 31:1–31:15. DOI: 10.4230/LIPIcs.DISC.2019.31.

[6] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch and A. Sezgin, 'Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation,' in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy: Association for Computing Machinery, 2013. DOI: 10.1145/2482767.2482789.

[7] C. M. Kirsch, H. Payer, H. Röck and A. Sokolova, 'Performance, scalability, and semantics of concurrent fifo queues,' in *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'12, Fukuoka, Japan: Springer-Verlag, 2012, pp. 273–287. DOI: 10.1007/978-3-642-33078-0_20.

[8] M. Wimmer, J. Gruber, 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*, ser. PPoPP 2015, San Francisco, CA, USA: ACM, 2015, pp. 277–278. DOI: 10.1145/2688500.2688547.

[9] M. P. Herlihy and J. M. Wing, 'Linearizability: a correctness condition for concurrent objects,' *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990. DOI: 10.1145/78969.78972.

[10] G. Kappes and S. V. Anastasiadis, 'A family of relaxed concurrent queues for low-latency operations and item transfers,' *ACM Trans. Parallel Comput.*, vol. 9, no. 4, 2022. DOI: 10.1145/3565514.

[11] M. Williams, P. Sanders and R. Dementiev, 'Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues,' in *29th Annual European Symposium on Algorithms (ESA 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17. DOI: 10.4230/LIPIcs.ESA.2021.81.

[12] A. Postnikova, N. Koval, G. Nadiradze and D. Alistarh, 'Multi-queues can be state-of-the-art priority schedulers,' in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 353–367. DOI: 10.1145/3503221.3508432.

[13] G. Zhang, G. Posluns and M. C. Jeffrey, 'Multi bucket queues: Efficient concurrent priority scheduling,' in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24, Nantes, France: Association for Computing Machinery, 2024, pp. 113–124.

[14] K. von Geijer, P. Tsigas, E. Johansson and S. Hermansson, 'Balanced allocations over efficient queues: A fast relaxed fifo queue,' in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 382–395. DOI: 10.1145/3710848.3710892.

[15] T. Kraska, 'Towards instance-optimized data systems, keynote,' *2021 International Conference on Very Large Data Bases*, 2021.

[16] M. D'Antonio, K. von Geijer, T. S. Mai, P. Tsigas and H. Vandierendonck, 'Relax and don't stop: Graph-aware asynchronous sssp,' in *Proceedings of the 1st FastCode Programming Challenge*, ser. FCPC '25, The Westin Las Vegas Hotel & Spa, Las Vegas, NV, USA: ACM, 2025, pp. 43–47.

[17] H. Rihani, P. Sanders and R. Dementiev, 'Multiqueues: Simple relaxed concurrent priority queues,' in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 80–82. DOI: 10.1145/2755573.2755616.

[18] L. Lamport, 'How to make a multiprocessor computer that correctly executes multiprocess programs,' *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[19] K. von Geijer and P. Tsigas, *Artifact of the paper: Elastic Relaxation of Concurrent Data Structures*, version v2. DOI: 10.5281/zenodo.11547062.

[20] G. L. Peterson and J. E. Burns, 'Concurrent reading while writing ii: The multi-writer case,' in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, ser. SFCS '87, USA: IEEE Computer Society, 1987, pp. 383–392. DOI: 10.1109/SFCS.1987.15.

[21] M. M. Michael and M. L. Scott, 'Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,' in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275. DOI: 10.1145/248052.248106.

[22] Intel Corporation, *Intel® 64 and ia-32 architectures software developer's manual*, Combined Volumes 1-5, Intel Corporation, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[23] A. Rudén and L. Andersson, 'Relaxed priority queue & evaluation of locks,' Master's thesis, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, Jun. 2023. [Online]. Available: https://gupea.ub.gu.se/handle/2077/78919.

[24] A. Morrison and Y. Afek, 'Fast concurrent queues for x86 processors,' in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Shenzhen, China: Association for Computing Machinery, 2013, pp. 103–112. DOI: 10.1145/2442516.2442527.

[25] C. Yang and J. Mellor-Crummey, 'A wait-free queue as fast as fetch-and-add,' in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, Spain: Association for Computing Machinery, 2016. DOI: 10.1145/2851141.2851168.

[26] R. Treiber, 'Systems programming: Coping with parallelism,' International Business Machines Incorporated, Thomas J. Watson Research Center, Tech. Rep., 1986.

[27] M. M. Michael, 'Cas-based lock-free algorithm for shared deques,' in *Euro-Par 2003 Parallel Processing*, Springer Berlin Heidelberg, 2003, pp. 651–660.

[28] K. von Geijer and P. Tsigas, *How to relax instantly: Elastic relaxation of concurrent data structures*, arXiv:2403.13644, 2024. arXiv: 2403.13644 [cs.DS].

[29] T. David, R. Guerraoui and V. Trigonakis, 'Asynchronized concurrency: The secret to scaling concurrent search data structures,' *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 631–644, 2015. DOI: 10.1145/2786763.2694359.

[30] K. Fraser, 'Practical lock-freedom,' Ph.D. dissertation, University of Cambridge, 2003.

[31] D. Hendler, N. Shavit and L. Yerushalmi, 'A scalable lock-free stack algorithm,' *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 1–12, 2010. DOI: 10.1016/j.jpdc.2009.08.011.

[32] D. Alistarh, J. Kopinsky, J. Li and N. Shavit, 'The spraylist: A scalable relaxed priority queue,' *SIGPLAN Not.*, vol. 50, no. 8, pp. 11–20, 2015. DOI: 10.1145/2858788.2688523.

[33] K. von Geijer and P. Tsigas, 'How to relax instantly: Elastic relaxation of concurrent data structures,' in *Euro-Par 2024: Parallel Processing*, Cham: Springer Nature Switzerland, 2024, pp. 119–133.

[34] K. Sagonas and K. Winblad, 'The Contention Avoiding Concurrent Priority Queue,' in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 314–330.

[35] K. Åström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning*, English. ISA - The Instrumentation, Systems and Automation Society, 1995.

[36] T. A. Davis and Y. Hu, 'The university of florida sparse matrix collection,' *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. DOI: 10.1145/2049662.2049663.

[37] Y. Wang, S. Yu, L. Dhulipala, Y. Gu and J. Shun, 'Geograph: A framework for graph processing on geometric data,' *SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, pp. 38–46, Jun. 2021.

[38] Y. Zheng, L. Liu, L. Wang and X. Xie, 'Learning transportation mode from raw gps data for geographic applications on the web,' in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08, Beijing, China: ACM, 2008, pp. 247–256.

[39] R. J. Pittman. 'Outpouring of searches for the late Michael Jackson.' Official Google Blog. (Jun. 2009), [Online]. Available: https://googleblog.blogspot.com/2009/06/outpouring-of-searches-for-late-michael.html (visited on 09/05/2025).

[40] G. Urdaneta, G. Pierre and M. van Steen, 'Wikipedia workload analysis for decentralized hosting,' *Comput. Netw.*, vol. 53, no. 11, pp. 1830–1845, Jul. 2009.

[41] R. M. Karp and Y. Zhang, 'Randomized parallel algorithms for backtrack search and branch-and-bound computation,' *J. ACM*, vol. 40, no. 3, pp. 765–789, 1993. DOI: 10.1145/174130.174145.

[42] A. Haas, T. Hütter, C. M. Kirsch, M. Lippautz, M. Preishuber and A. Sokolova, 'Scal: A benchmarking suite for concurrent data structures,' in *Networked Systems*, Springer, 2015, pp. 1–14. DOI: 10.1007/978-3-319-26850-7_1.

[43] D. Nguyen, A. Lenharth and K. Pingali, 'A lightweight infrastructure for graph analytics,' in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 456–471. DOI: 10.1145/2517349.2522739.

[44] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li and G. Nadiradze, 'Distributionally linearizable data structures,' in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, vol. test, Vienna, Austria: Association for Computing Machinery, 2018, pp. 133–142. DOI: 10.1145/3210377.3210411.

[45] D. Hendler, A. Khattabi, A. Milani and C. Travers, 'Upper and Lower Bounds for Deterministic Approximate Objects,' *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, vol. 00, pp. 438–448, 2021. DOI: 10.1109/icdcs51616.2021.00049.

[46] D. Alistarh, J. Kopinsky, J. Li and G. Nadiradze, 'The power of choice in priority scheduling,' in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 283–292. DOI: 10.1145/3087801.3087810.

[47] S. Walzer and M. Williams, 'A Simple yet Exact Analysis of the MultiQueue,' in *33rd Annual European Symposium on Algorithms (ESA 2025)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 351, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 85:1–85:14. DOI: 10.4230/LIPIcs.ESA.2025.85.

[48] *FCPC '25: Proceedings of the 1st FastCode Programming Challenge*, The Westin Las Vegas Hotel & Spa, Las Vegas, NV, USA: ACM, 2025.

[49] M. A. Hassaan, M. Burtscher and K. Pingali, 'Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms,' in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, San Antonio, TX, USA: ACM, 2011, pp. 3–12.

[50] S. Beamer, K. Asanović and D. Patterson, 'Direction-optimizing breadth-first search,' in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012.