

# Distributionally Linearizable Data Structures

Dan Alistarh  
IST Austria

Trevor Brown  
IST Austria

Justin Kopinsky  
MIT

Jerry Z. Li  
MIT

Giorgi Nadiradze  
ETH Zurich

## ABSTRACT

Relaxed concurrent data structures have become increasingly popular, due to their scalability in graph processing and machine learning applications ([14, 24]).

Despite considerable interest, there exist families of natural, high performing *randomized* relaxed concurrent data structures, such as the popular MultiQueue [27] pattern for implementing relaxed priority queue data structures, for which no guarantees are known in the concurrent setting [4].

Our main contribution is in showing for the first time that, under a set of analytic assumptions, a family of relaxed concurrent data structures, including variants of MultiQueues, but also a new approximate counting algorithm we call the MultiCounter, provides strong probabilistic guarantees on the degree of relaxation with respect to the sequential specification, *in arbitrary concurrent executions*. We formalize these guarantees via a new correctness condition called *distributional linearizability*, tailored to concurrent implementations with randomized relaxations. Our result is based on a new analysis of an *asynchronous* variant of the classic power-of-two-choices load balancing algorithm, in which placement choices can be based on inconsistent, outdated information (this result may be of independent interest). We validate our results empirically, showing that the MultiCounter algorithm can implement scalable relaxed timestamps.

### ACM Reference Format:

Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3210377.3210411>

## 1 INTRODUCTION

Consider a system of  $n$  threads, which share a set of  $m$  distinct atomic counters. We wish to implement a *scalable approximate counter*, which we will call a *MultiCounter*, by distributing the contention among these  $m$  distinct instances: to *increment* the global counter, a thread selects two atomic counters  $i$  and  $j$  uniformly at random, reads their values, and (atomically) increments by 1 the value of the one which has *lower value* according to the values it

read. To *read* the global counter, the thread returns the value of a randomly chosen counter  $i$ , multiplied by  $m$ .<sup>1</sup>

The astute reader will have noticed that this process is similar to the classic two-choice load balancing process [7], in which a sequence of balls are placed into  $m$  initially empty bins, and, in each step, a new ball is placed into the less loaded of two randomly chosen bins. Here, the individual atomic counters are the *bins*, and each increment corresponds to a new *ball* being added. This sequential load balancing process is extremely well studied! [23, 26]: a series of deep technical results established that the difference between the most loaded bin and the average is  $O(\log \log m)$  in expectation [7, 23], and that this difference remains stable as the process executes for increasingly many steps [10, 25]. We would therefore expect the above relaxed concurrent counter to have relatively low and stable skew among the outputs at consecutive operations, and to scale well, as contention is distributed among the  $m$  counters.

However, there are several technical issues when attempting to analyze this natural process in a concurrent setting.

- First, concurrency interacts with classic two-choice load balancing process in non-trivial ways. The key property of the two-choice process which ensures good load balancing is that trials are *biased towards less loaded bins*—equivalently, operations are biased towards incrementing counters of lesser value. However, this property may break due to concurrency: *at the time of the update*, a thread may end up updating the counter of *higher* value among its two choices if the counter of smaller value is updated concurrently since it was read by the thread, thus surpassing the other counter.
- Second, perhaps surprisingly, it is currently unclear how to even *specify* such a concurrent data structure. Despite a significant amount of work on specifying *deterministic relaxed* data structures [1, 15, 17], none of the existing frameworks cover relaxed *randomized* data structures.
- Finally, assuming such a data structure can be analyzed and specified, it is not clear whether it would be in any way *useful*: many existing applications are built around data structures with deterministic guarantees, and it is not obvious how scalable, relaxed data structures can be leveraged in standard concurrent settings.

One may find it surprising that analysing such a relatively simple concurrent process is so challenging. Beyond this specific instance, these difficulties reflect wider issues in this area: although these constructs are reasonably popular in practice due to their good scalability, e.g. [8, 24, 27, 31], their properties are non-trivial to pin

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
SPAA '18, July 16–18, 2018, Vienna, Austria

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-5799-9/18/07...\$15.00  
<https://doi.org/10.1145/3210377.3210411>

<sup>1</sup>This multiplication serves to maintain the same magnitude as the total number of updates to the distributed counter up to a point in time.

down [4], and it is as of yet unclear how they interact with the higher-order algorithmic applications they are part of [20].

**Contribution.** In this work, we take a step towards addressing these challenges. Specifically:

- We provide the first analysis of a two-choice load balancing process in an asynchronous setting, where operations may be interleaved, and the interleaving is decided by an adversary. We show that the resulting process is robust to concurrency, and continues to provide strong balancing guarantees in potentially infinite executions, as long as the ratio between the number of bins and the number of threads is above a large constant threshold.
- We introduce a new correctness condition for *randomized relaxed* data structures, called *distributional linearizability*. Intuitively, a concurrent data structure  $D$  is distributionally linearizable to a *sequential random process*  $R$ , defined in terms of a sequential specification  $S$ , a cost function *cost* measuring the deviation from the sequential specification, and a distribution  $\mathcal{P}$  on the values of the cost function, if every execution of  $D$  can be mapped onto an execution of the relaxed sequential process  $R$ , respecting the outputs and the costs incurred, as well as the order of non-overlapping operations.
- We prove that the randomized *MultiCounter* data structure introduced above is distributionally linearizable to a (sequential) variant of the classic two-choice load balancing process. This allows us to formally define the properties of MultiCounters. Moreover, we show that this analytic framework also covers variants of *MultiQueues* [27], a popular family of concurrent data structures implementing relaxed concurrent priority queues. This yields the first analytical guarantees for MultiQueues in concurrent executions.
- We implement the MultiCounters, and show that they can provide a highly scalable approximate timestamping mechanism, with relatively low skew.

**Techniques.** Our main technical contribution is the concurrent analysis of the classic two-choice load balancing process, in an asynchronous setting, where the interleaving of low-level steps is decided by an oblivious adversary. The core of our analysis builds on the elegant potential method of Peres, Talwar and Wieder [25], which we render robust to asynchronous updates based on potentially stale information. To achieve this, we overcome two key technical challenges. The first is that, given an operation  $op$ , as more and more other operations execute between the point where it reads and the point where it updates, the more stale its information becomes, and so the probability that  $op$  makes the “right” choice at the time of update, inserting into the less loaded of its two random choices, *decreases*. Moreover, operations updating with stale information will “stampede” towards lower-weight bins, effectively skewing the distribution. The second technical issue we overcome is that long-running operations, which experience a lot of concurrency, may in fact be adversarially biased towards the wrong choice, inserting into the more loaded of its two choices with non-trivial probability.

In brief, our analysis circumvents this issue by showing that a variant of the two-choice process where up to a constant fraction of updates are corrupted, in the sense that they perform the “wrong”

update, will still have similar balance properties as the original process. It is interesting to note that even the *order* in which corrupted updates occur can be controlled by the adversary through increased concurrency, which is not the case in standard analyses [25]. The critical property which we leverage in our analysis is that, while individual operations can be arbitrarily contended (and therefore biased), there is a bound of  $n$  on the *average* contention per operation, which in turn bounds the average amount of bias the adversary can induce over a period of time. Our argument formalizes this intuition, and phrases it in terms of the evolution of the potential function.

We show that this result has implications beyond “parallelizing” the classic two-choice process, as we can leverage it to obtain probabilistic bounds on the skew of the MultiCounter. Using the framework of [4], which connected two-choice load balancing with MultiQueue data structures in the sequential case, we can obtain guarantees for this popular data structure pattern in concurrent executions.

## 2 RELATED WORK

**Randomized Load Balancing.** The classic two-choice balanced allocation process was introduced in [7], where the authors show that, under two-choice insertion, the most loaded among  $m$  bins is at most  $O(\log \log m)$  above the average, both in expectation and with high probability. The literature studying analyses and extensions of this process is extremely vast, hence we direct the reader to [23, 26] for in-depth surveys of these techniques. Considerable effort has been dedicated to understanding guarantees in the “heavily-loaded” case, where the number of insertion steps is unbounded [10, 25], and in the “weighted” case, in which ball weights come from a probability distribution [11, 30]. A tour-de-force by Peres, Talwar, and Wieder [25] gave a potential argument characterizing a general form of the heavily-loaded, weighted process on *graphs*. Our analysis starts from their framework, and modifies it to analyze a concurrent, adversarial process. One significant change from their analysis is that, due to the adversary, changes in the potential are only partly stochastic: most steps might be slightly biased away from the better of the two choices, while a subset of choices might be almost deterministically biased towards the *wrong* choice. Further, the adversary can decide the *order* in which these different steps, with different biases, occur.

Lenzen and Wattenhofer [21] analyzed *parallel* balls-into-bins processes, in which  $m$  balls need to be distributed among  $m$  bins, under a communication model between the balls and the bins, showing that almost-perfect allocation can be achieved in  $O(\log^* m)$  rounds of communication. This setting is quite different from the one we consider here. Similar delayed information models, where outdated information is given to the insertion process were considered by Mitzenmacher [22] and by Berenbrink, Czumaj, Englert, Fridetzky, and Nagel [9]. The former reference proposes a bulletin board model with periodic updates, in which information about the load of the model is updated only periodically (every  $T$  seconds), and various allocation mechanisms. The author provides an analysis of this process in the asymptotic case (as  $m \rightarrow \infty$ ), supported by simulations. The latter reference [9] considers a similar model where balls arrive in *batches*, and must perform allocations collectively

based solely on the information available at the beginning of the batch, without additional communication. The authors prove that the greedy multiple-choice process preserves its strong load balancing properties in this setting: in particular, the gap between min and max remains  $O(\log m)$ . The key difference between these models and the one we consider is that our model is completely asynchronous, and in fact the interleavings are chosen adversarially. The technique we employ is fundamentally different from those of [9, 22]. In particular, we believe our techniques could be adapted to re-derive the main result of [9], albeit with worse constants.

Recent work by a subset of the authors [4] analyzed the following producer-consumer process: a set of balls labelled  $1, 2, \dots, b$  are inserted sequentially at random into  $m$  bins; in parallel, balls are removed from the bins by always picking the lower labelled (higher priority) of two uniform random choices.<sup>2</sup> This process sequentially models a series of popular implementations of concurrent priority queue data structures, e.g. [16, 27]. This process provides the following guarantees: in each step  $t$ , the expected *rank* of the label removed among labels still present in the system is  $O(m)$ , and  $O(m \log m)$  with high probability in  $m$ . That is, this sequential process provides a structured probabilistic relaxation of a standard priority queue.

**Relaxed Data Structures.** The process considered in [4] is sequential, whereas the data structures implemented are concurrent. Thus, there was a significant gap between the theoretical guarantees and the practical implementation. Our current work extends to concurrent data structures, closing this gap. Under the oblivious adversary assumption and given our parametrization, we show for the first time that practical data structures such as [4, 16, 27] provide guarantees in real executions.

Designing efficient concurrent/parallel data structures with relaxed semantics was initiated by Karp and Zhang [19], with other significant early work by Deo and Prasad [12] and Sanders [28]. It has recently become an extremely active research area, see e.g. [4, 5, 8, 16, 24, 27, 29, 31] for recent examples. To the best of our knowledge, ours is the first analysis of randomized relaxed concurrent data structures which works under arbitrary oblivious schedulers: previous analyses such as [4, 5, 27] required strong assumptions on the set of allowable interleavings. Dice et al. [13] considered randomized data structures for scalable exact and approximate counting. They consider the efficient parallelization of sequential approximate counting methods, and therefore have a significantly different focus than our work.

### 3 SYSTEM MODEL

**Asynchronous Shared Memory.** We consider a standard asynchronous shared-memory model, e.g. [6], in which  $n$  threads (or processes)  $P_1, \dots, P_n$ , communicate through shared memory, on which they perform atomic operations such as read, write, compare-and-swap and fetch-and-increment. The fetch-and-increment operation takes no arguments, and returns the value of the register before the increment was performed, incrementing its value by 1.

**The Oblivious Adversarial Scheduler.** Threads follow an algorithm, composed of shared-memory steps and local computation,

including random coinflips. The order of process steps is controlled by an adversarial entity we call the *scheduler*. Time  $t$  is measured in terms of the number of shared-memory steps scheduled by the adversary. The adversary may choose to crash a set of at most  $n - 1$  processes by not scheduling them for the rest of the execution. A process that is not crashed at a certain step is *correct*, and if it never crashes then it takes an infinite number of steps in the execution. In the following, we assume a standard *oblivious* adversarial scheduler, which decides on the interleaving of thread steps independently of the coinflips they produce during the execution.

**Shared Objects.** The algorithms we consider are implementations of shared objects. A shared object  $O$  is an abstraction providing a set of *methods*, each given by a sequential specification. In particular, an implementation of a method  $M$  for an object  $O$  is a set of  $n$  algorithms, one for each executing process. When thread  $P_i$  invokes method  $M$  of object  $O$ , it follows the corresponding algorithm until it receives a response from the algorithm. Upon receiving the response, the process is immediately assigned another method invocation. In the following, we do not distinguish between a method  $M$  and its implementation. A method invocation is *pending* at some point in the execution if it has been initiated but has not yet received a response. A pending method invocation is *active* if it is made by a *correct* process (note that the process may still crash in the future). For example, a concurrent counter could implement read and increment methods, with the same semantics as those of the sequential data structure.

**Linearizability.** The standard correctness condition for concurrent implementations is *linearizability* [18]: roughly, a linearizable implementation ensures that each concurrent operation can be seen as executing at a single instant in time, called its linearization point. The mapping from method calls to linearization points induces a global order on the method calls, which is guaranteed to be consistent to a sequential execution in terms of the method outputs; moreover, each linearization point must occur between the start and end time of the corresponding method.

Recent work, e.g. [17], considers deterministic relaxed variants of linearizability, in which operations are allowed to deviate from the sequential specification by a *relaxation* factor. Such relaxations appear to be necessary in the case of data structures such as exact counters or priority queues in order to circumvent strong linear lower bounds on their concurrent complexity [2]. While specifying such data structures in the concurrent case is well-studied [1, 15, 17], less is known about how to specify structured randomized relaxations.

**With High Probability.** We say that an event occurs *with high probability* in a parameter, e.g.  $m$ , if it occurs with probability at least  $1 - 1/m^c$ , for some constant  $c \geq 1$ .

### 4 THE MULTICOUNTER ALGORITHM

**Description.** The algorithm implements an approximate counter by distributing updates among  $m$  distinct counters, each of which supports atomic read and increment operations. Please see Algorithm 1 for pseudocode. To read the counter value, a thread simply picks one of the  $m$  counters uniformly at random, and returns its value multiplied by  $m$ . To increment the counter value, the thread picks two counter indices  $i$  and  $j$  uniformly at random, and reads

<sup>2</sup>Balls in each bin are sorted in increasing order of label, i.e. each bin corresponds to a sequential priority queue.

their current values sequentially. It then proceeds to update (increment) the value of the counter which appeared to have a lower value given its two reads. (In case of a tie, or when the two choices are identical, the tie is broken arbitrarily.)

---

**Algorithm 1** Pseudocode for the MultiCounter Algorithm.
 

---

**Shared:**  $Counters[m]$  // Array of integers representing set of  $m$  distinct counters

**function** Read()

$i \leftarrow \text{random}(1, m)$

**return**  $m \cdot Counters[i].\text{read}()$

**function** Increment()

$i \leftarrow \text{random}(1, m)$

$j \leftarrow \text{random}(1, m)$

$v_i \leftarrow Counters[i].\text{read}()$

$v_j \leftarrow Counters[j].\text{read}()$

$Counters[\arg \min(v_i, v_j)].\text{increment}()$

---

**Relation to Load Balancing.** A *sequential* version of the above process, in which the counter is read or incremented *atomically*, is identical to the classic two-choice balanced allocation process [7], where each counter corresponds to a bin, and each increment corresponds to a new ball being inserted into the less loaded of two randomly chosen bins.

In a concurrent setting, the critical departure from the sequential model is that the values read can be *inconsistent* with respect to a sequential execution: there may be no single point in time when the two counters had the values  $v_i$  and  $v_j$  observed by the thread; moreover, these values may change between the point where they are read, and the point where the update is performed.

More technically, the sequential variant of the two-choice process has the crucial property that, at each increment step, it is “biased” towards incrementing the counter of lower value. This does not necessarily hold for the concurrent approximate counter: for an operation where a large number of updates occur between the read and the update points, the read information is stale, and therefore the thread’s increment choice may be no better than a perfectly random one; in fact, as we shall see in the analysis, it is actually possible for an adversary to engineer cases where the algorithm’s choice is biased towards incrementing the counter of *higher value*.

## 5 DISTRIBUTIONAL LINEARIZABILITY

We generalize the classic linearizability correctness condition to cover *randomized relaxed* concurrent data structures, such as the MultiCounter. Intuitively, we will say that a concurrent data structure  $D$  is distributionally linearizable to a corresponding *relaxed sequential process*  $R$ , defined in terms of a sequential specification  $S$ , a cost function  $cost$  measuring the deviation from the sequential specification, and a distribution  $\mathcal{P}$  on the  $cost$  function values, such that every execution of  $D$  can be mapped onto an execution of the relaxed sequential process  $R$ , respecting the outputs and the incurred costs, as well as the order of non-overlapping operations. To formalize this definition, we introduce the following machinery, part of which is adopted from [17].

**Data Structures and Labeled Transition Systems.** Let  $\Sigma$  be a set of methods including input and output values. A sequential history  $s$  is a sequence over  $\Sigma$ , i.e. an element in  $\Sigma^*$ . A (sequential) data structure is a sequential specification  $S$  which is a prefix-closed set of sequential histories. For example, the sequential specification of a stack consists of all valid sequences for a stack, i.e. in which every push places elements on top of the stack, and every pop removes elements from the top of the stack.

Given a sequential specification  $S$ , two sequential histories  $s, t \in S$  are equivalent, written  $s \approx t$ , if they correspond to the same “state:” formally, for any sequence  $u \in \Sigma^*$ ,  $su \in S$  iff  $tu \in S$ . Let  $[s]_S$  be the equivalence class of  $s \in S$ .

**DEFINITION 5.1.** Let  $S$  be a sequential specification. Its corresponding labeled transition sequence (LTS) is an object  $LTS(S) = (Q, \Sigma, \rightarrow, q_0)$ , with states  $Q = \{[s]_S | s \in S\}$ , set of labels  $\Sigma$ , transition relation  $\rightarrow \subseteq Q \times \Sigma \times Q$  given by  $[s]_S \xrightarrow{m} [sm]_S$  iff  $sm \in S$ , and initial state  $q_0 = [\epsilon]_S$ .

Notice that the sequential specification  $S$  can be alternatively defined as the set of all traces of the initial state of  $LTS(S)$ : formally, for any  $u \in \Sigma^*$ , we have  $u \in S$  iff  $q_0 \xrightarrow{u}$ .

**Randomized Quantitative Relaxations.** Let  $S \in \Sigma^*$  be a data structure with  $LTS(S) = (Q, \Sigma, \rightarrow, q_0)$ . To obtain a randomized quantitative relaxation of  $S$ , we apply the following four steps. The first three steps are identical to deterministic quantitative relaxations [17], whereas the fourth defines the probability distribution on costs:

- (1) **Completion:** We start from  $LTS(S)$ , and construct a completed labeled transition system, with transitions from any state to any other state by any method:

$$LTS_c(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0).$$

- (2) **Cost function:** We add a cost function  $cost : Q \times \Sigma \times Q \rightarrow \mathbb{R}$  to the LTS. The transition cost will satisfy

$$cost(q, m, q') = 0 \text{ if and only if } q \xrightarrow{m} q' \text{ in } LTS(S).$$

A quantitative path is a sequence

$$\kappa = q_1 \xrightarrow{m_1, k_1} q_2 \xrightarrow{m_2, k_2} \dots \xrightarrow{m_n, k_n} q_{n+1}.$$

We call the sequence  $\tau = (m_1, k_1), \dots, (m_n, k_n)$  of transitions and costs the quantitative trace of  $\kappa$ , denoted by  $qtr(\kappa)$ .

- (3) **Path cost function:** Given a quantitative path  $\kappa$ , its path cost is defined as  $pcost : qtr(S) \rightarrow C$ . Path costs are monotone with respect to prefix order: if  $\tau$  is a prefix of  $\tau'$ , then  $pcost(\tau) \leq pcost(\tau')$ .

- (4) **Probability distribution:** Given an arbitrary state  $[s]$  in  $LTS(S)$ , we define a probability space  $(\Omega, \mathcal{F}, \mathcal{P})$  on the set of possible transitions and their corresponding costs from this state, where the sample space  $\Omega$  is the set of all transitions in  $Q \times \Sigma \times Q$ , the  $\sigma$ -algebra  $\mathcal{F}$  is defined in the straightforward way based on the set of elementary events  $\Omega$ , and  $\mathcal{P}$  is a probability measure  $\mathcal{P} : \mathcal{F} \rightarrow [0, 1]$ .

Importantly, this allows us to define, for any path, the notion of probability for costs incurred at each step. This probability space is readily extended for arbitrary finite paths, where we assume that the cost probabilities at each step are independent of previous steps, i.e., historyless. This process induces

a Markov chain, whose state at each step is given by the state  $[s]$  of the corresponding LTS, and whose transitions are LTS transitions, with costs and probabilities as above.

**Distributional Linearizability.** With this in place, we now define distributionally linearizable data structures:

**DEFINITION 5.2.** *Let  $D$  be a randomized concurrent data structure, and let  $R$  be a randomized quantitative relaxation  $R$  of a sequential specification  $S$  with respect to a cost function  $\text{cost}$ , and a probability distribution  $\mathcal{P}$  on costs. We say that  $D$  is distributionally linearizable to  $R$  iff for every concurrent schedule  $\sigma$ , there exists a mapping of completed operations in  $D$  under  $\sigma$  to transitions in the quantitative path of  $R$ , preserving outputs, and respecting the order of non-overlapping operations. This mapping can be used to associate any schedule  $\sigma$  to a distribution of costs for  $D$  under the schedule  $\sigma$ .*

We now make a few important remarks on this definition.

- (1) The main difficulty when formally defining the “costs” incurred by  $D$  in a concurrent execution is in dealing with the execution history, and with the impact of pending operations on these costs. The above definition allows us to define costs, given a schedule, only in terms of the sequential process  $R$ , and bounds the incurred costs in terms of the probability distribution defined in  $R$ . This definition ensures that the probability distribution on costs incurred at each step only depends on the current state of the sequential process.
- (2) The second key question is how to use this definition. One subtle aspect of this definition is that the mapping to the sequential randomized quantitative relaxation is done *per schedule*: intuitively, this is because an adversary might change the schedule, and cause the distribution of costs of the data structure to *change*. Thus, it is often difficult to specify a precise cost distribution, which covers all possible schedules. However, for the data structures we analyze, we will be able to provide *tail bounds* on the cost distributions induced by *all possible schedules*.

The natural next question, which we answer in the following section, is whether non-trivial such data structures exist and can be analyzed.

## 6 ANALYSIS OF THE MULTICOUNTER

We will focus on proving the following result.

**THEOREM 6.1.** *Fix a large constant  $C$ . Given an oblivious adversary,  $m$  distributed counters and  $n$  threads with  $m \geq 8Cn$ , for any fixed schedule, the MultiCounter algorithm is distributionally linearizable to a randomized relaxed sequential counter process, which, at any step  $t$ , returns a value that is at most  $O(m \log m)$  away from the number of increments applied up to  $t$ , both in expectation and with high probability in  $m$ .*

We emphasize that the relaxation guarantees are independent of the time  $t$  at which the guarantee is examined, and that they would thus hold in infinite executions.

### 6.1 Modeling the Concurrent Process

In the following, we will focus on analyzing executions formed exclusively of increment operations, whose lower-level steps may

be interleaved. (Adding read operations at any point during the execution will be immediate.) We model the process as follows. First, we assume a schedule that is fixed by the adversary. For each thread  $P_j$ , and non-negative integers  $j$ , we consider a sequence of increment operations  $(op_i^{(j)})$ , each of which is defined by its starting time  $s_i^{(j)}$ , corresponding to the time when its first read step was scheduled, and completion time  $f_i^{(j)}$ , corresponding to the time when its update time is scheduled, such that  $s_{i+1}^{(j)} > f_i^{(j)}$  for all  $i, j$ . (Recall that the scheduler defines a global order on individual steps.) At most  $n$  operations may be active at a given time, corresponding to the fact that we only have  $n$  parallel threads.

For each operation  $op_i$ , we record its *contention*  $\ell_i$  as the number of distinct increment operations scheduled between its start and end time. (Alternatively, we could define this quantity as the number of operations which complete in the time interval  $(s_i, f_i)$ .) Note that at most  $n - 1$  distinct operations can be concurrent with  $op_i$  at any given time, but the contention for a specific operation is potentially unbounded.

We can rephrase the original process as follows. For each operation  $op_i$ , the adversary sets the time when it performs the first and its second read of counter values / bin weights, as well as its contention  $\ell_i$ , by scheduling other operations concurrently. The only constraint on the adversary is that not more than  $n$  operations can be active at the same time.

Since the adversary is oblivious, we notice that the update process is equivalent to the following: at the time when the update is scheduled, the thread executing the operation generates two uniform random indices  $i$  and  $j$ , and is given values  $v_i$  and  $v_j$  for the two corresponding counters / bin weights, read at previous (possibly different) points in time. We will stick to the bin weight formulation from now on, with the understanding that the two are equivalent.

The thread will then increment the weight of the bin with the smaller value read (among  $v_i$  and  $v_j$ ) by 1. This formulation has the slight advantage that it makes the update process sequential, by moving the random choices to the time when the update is made, using the principle of deferred decisions. Critically, the bin weights on which the update decision is based are potentially stale. We will focus on this simplified variant of the process in the following.

**Discussion.** The key difference between the above process and the classic power-of-two-choices process is the fact that the choice of bin / counter which the thread updates is based on stale, potentially invalid information. Recall that key to the strong balancing properties of the classic process is the fact that it is biased towards inserting in *less loaded* bins; the process which inserts into randomly chosen bins is known to diverge [25]. In particular, notice it is possible that, by the time when the thread performs the update, the order of the bins' load may have changed, i.e. the thread in fact inserts into the *more loaded* bin among its two choices at the time of the update.

Since the oblivious adversary decides its schedule independently of the threads' random choices, it cannot *deterministically* cause a specific update to insert into the more loaded bin. However, it can *significantly bias* an update towards inserting into the more loaded bin:

Assume for example an execution suffix where all  $n$  threads read concurrently at some time  $t_R$ <sup>3</sup> and then proceed to perform updates, one after another. Pick an operation  $op$  for which the gap between the two values read  $v_i$  and  $v_j$  (at the time of the read) is 1, say  $v_i = v_j + 1$ . So  $op$  will increment  $v_j$ . At the same time, notice that all the other operations which read concurrently with  $op$  are biased towards inserting in  $v_j$  rather than  $v_i$ , since its rank (in increasing order of weight) is lower than that of bin  $i$ . Hence, as the adversary schedules more and more operations between  $t_R$  at  $op$ 's update time, it is increasingly likely to *invert* the relation between  $i$  and  $j$  by the time of  $op$ 's update, causing it to insert into the “wrong” bin.

The previous example suggests that the adversary is able to bias some subset of the operations towards picking the wrong bin at the time of the update. Another issue is that operations which experience high contention, for which there are many updates between the read point and the update point, the read values  $v_i$  and  $v_j$  become meaningless: for example, if the weights of bin  $i$  and  $j$  become equal at some time  $t_0$  between  $t_R$  and  $op$ 's update, then from this point in time these two bins appear completely symmetrical to the algorithm, and  $op$ 's choice given the information that  $v_i > v_j$  at  $t_R$  may be no better than uniform random.

One issue which further complicates this last example is that, at  $t_0$ , there may be a non-zero number of other operations which already made their reads (for instance, at  $t_R$ ), but have not updated yet. Since these operations read at a point where  $v_i > v_j$ , they are in fact biased towards inserting in  $v_j$ . So, looking at the event that  $op$  updates the *less loaded* of its two random choices at update time, we notice that its probability in this example is *strictly worse* than uniform random choice.

We summarize this somewhat lengthy discussion with two points, which will be useful in our analysis:

- (1) As they experience concurrent updates, operations may accrue bias towards inserting into the *more loaded* of their two random choices.
- (2) Long-running operations may in fact have a higher probability of inserting into the more loaded bin than into the less loaded one, i.e. may become biased towards making the “wrong” choice at the time of the update.

## 6.2 Notation and Background

**The  $(1 + \beta)$  Process.** In the following, it will be useful to consider the following *sequential* relaxation of the two-choice process, introduced by [25], called the  $(1 + \beta)$ -choice process: We are given  $m$  bins, initially empty. In each step  $t$ , we flip a biased coin: with probability  $\beta > 0$  we will insert a ball into the less loaded of two randomly chosen bins; otherwise, we insert the ball into a randomly chosen bin. This process is analyzed in [25], which shows that, at any time  $t$  in its execution, the gap between the maximum and minimum value of a bin is  $O(\log m/\beta)$ , with high probability in  $m$ , irrespective of  $t$ .

We now introduce some notation, which will be common between our analysis and that of [25]. For simplicity, we will assume that, at the beginning of each step in this sequential process, bins

<sup>3</sup>Technically, since we count time in terms of shared-memory operations, these reads occur at consecutive times after  $t_R$ . However, all their read values are identical to the read value at  $t_R$ , and hence we choose to simplify notation in this way.

are always ranked in *increasing order of their weight*. If  $p_i$  is the probability that we pick the  $i$ th ranked bin for insertion, and  $\beta$  is the two-choice probability, then it is easy to see that the  $(1 + \beta)$  process guarantees

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

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

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

For any bin  $j$  and time  $t$ , let  $x_j(t)$  be the weight of bin  $j$  at time  $t$ . Let  $\mu(t) = \sum_{j=1}^m x_j(t)/m$  be the average weight at time  $t$  over the bins. Let  $y_i(t) = x_i(t) - \mu(t)$ , and let  $\alpha < 1$  be a parameter to be fixed later. Define

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

Finally, define the potential function

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

The main technical result of [25] can be phrased as:

**THEOREM 6.2.** *Let  $\alpha, \beta$  be parameters as given above, and let  $\epsilon = \frac{\beta}{16}$ . Then there exists a constant  $C(\epsilon) = \text{poly}(\frac{1}{\epsilon})$  such that, for any time  $t \geq 0$ , we have  $\mathbb{E}[\Gamma(t)] \leq C(\epsilon)m$ .*

In turn, this implies that the maximum gap between the most loaded and the least loaded bin at a step is  $O(\log m)$  in expectation and with high probability in  $m$ .

## 6.3 Main Argument

**Analysis Overview.** Throughout the analysis, we will fix a large constant  $C$  such that  $m \geq 8Cn$ . The analysis proceeds in the following technical steps.

- We define an operation  $op_t$  as *good*( $\gamma$ ) for  $\gamma > 0$  if, with probability at least  $1/2 + \gamma$ , the bin  $op_t$  adds to is not accessed by another operation at any point during the execution of  $op_t$ . We will identify a constant  $\gamma > 0$  such that all operations with contention  $\leq Cn$  are *good*( $\gamma$ ).
- We lower bound the expected decrease in potential caused by a step that is *good*( $\gamma$ ).
- We upper bound the expected increase in potential caused by a step that is not *good*( $\gamma$ ).
- We argue that, for any adversarial strategy, out of any group of  $Cn$  consecutive operations, at least  $(C - 1)n$  have to be *good*( $\gamma$ ). We then upper bound the change in potential over any stretch of  $Cn$  operations, showing that it has to stay in  $O(m)$ .

We will prove the following simple claim as starting point for the analysis:

**LEMMA 6.3.** *If for  $op_t$  we have that its contention  $\ell_t \leq Cn$ , then the step (operation)  $t$  is *good*( $\frac{1}{5}$ ).*

PROOF. Let  $i$  and  $j$  be two random bins chosen by  $op_t$ , w.l.o.g we assume that  $i$  is chosen to add to by  $op_t$ . Considering that  $\ell_t \leq Cn$  and for any operation, the probability of accessing bin  $i$  is at most  $\frac{2}{m}$ , we get:

$$Pr[\text{bin } i \text{ untouched}] \geq \left(1 - \frac{2}{m}\right)^{Cn} \geq 2^{-\frac{1}{2}} \geq \frac{7}{10}.$$

where we used inequalities  $1 - \frac{x}{2} \geq 2^{-x}$  and  $m \geq 8Cn$ .  $\square$

Next we try to bound the expected decrease in potential if operation  $t$  is *good*( $\gamma$ ).

LEMMA6.4. *If  $op_t$  is good( $\gamma$ ), then :*

$$\mathbb{E}[\Gamma(t+1)|y(t)] \leq \Gamma(t) \left(1 - \frac{\alpha\epsilon}{4m}\right) + c, \quad (1)$$

for  $\epsilon = \gamma/6$  and  $c = c(\epsilon) = \text{poly}(1/\epsilon)$ .

PROOF. First notice that if  $op_t$  chooses to delete from bin  $i$  after looking at bins  $i, j$  and bin  $i$  is not accessed by any other operation during execution of  $op_t$ , then bin  $i$  must have been less loaded than bin  $j$  for the entire interval between the *second* read of  $op_t$  and the write of  $op_t$ .

Now, we will use Theorem 3.1 from [25], stated below. Assume that we are given a weight vector  $y(t)$ , in increasing order of weight, and two probability vectors  $p = (p_1, p_2, \dots, p_m)$  and  $q = (q_1, q_2, \dots, q_m)$ , where we assume that probability vectors are sorted in decreasing order. We say that  $p$  majorizes  $q$  if for any  $1 \leq k \leq m$ :

$$\sum_{i=1}^k p_i \geq \sum_{i=1}^k q_i$$

Let  $\mathbb{E}[\Gamma_p(t+1)|y(t)]$  be expected potential function if we choose bin according to probability vector  $p$  (that is,  $i$ -th less loaded bin is chosen with probability  $p_i$ ) and let  $\mathbb{E}[\Gamma_q(t+1)|y(t)]$  be expected potential function if we choose bin according to probability vector  $q$  (that is, the  $i$ -th least loaded bin is chosen with probability  $q_i$ ). Then Theorem 3.1 from [25] implies that  $\mathbb{E}[\Gamma_p(t+1)|y(t)] \leq \mathbb{E}[\Gamma_q(t+1)|y(t)]$ , because probability vector  $p$  is more biased towards lesser loaded bins than probability vector  $q$ .

What we need to show is that probability vector of  $op_t$  which is *good*( $\gamma$ ) majorizes the probability vector of  $1 + \beta$  choice process for some  $\beta = 2\gamma$ . That is, for any  $1 \leq k$ :

$$\sum_{i=1}^k p_i \geq \sum_{i=1}^k q_i.$$

We do exactly that in the following. Let  $q$  be the probability vector for the bin choice of the fully sequential process. As we know, for any  $1 \leq i \leq m$ ,  $q_i = \beta \frac{2(m-i)+1}{m^2} + \frac{1-\beta}{m}$ . Recall that if bin  $i$  is not accessed by another operation during the execution of  $op_t$ , then  $op_t$  must add to the bin which is the lesser loaded at the time of writing. Thus, if  $\rho \geq 1/2 + \gamma$  is the probability that  $op_t$  adds to the lesser loaded bin, we have that  $p_i = \rho \frac{2(m-i)}{m^2} + \frac{1}{m^2} + (1-\rho) \frac{2(i-1)}{m^2}$ .

Then:

$$\begin{aligned} \sum_{i=1}^k p_i &= \sum_{i=1}^k \left( \rho \frac{2(m-i)}{m^2} + \frac{1}{m^2} + (1-\rho) \frac{2(i-1)}{m^2} \right) \\ &= \rho \frac{2mk - k(k+1)}{m^2} + \frac{k}{m^2} + (1-\rho) \frac{k(k-1)}{m^2} \\ &= \rho \frac{2mk - 2k^2}{m^2} + \frac{k^2}{m^2} = 2\rho \frac{mk - k^2}{m^2} + \frac{k^2}{m^2}. \end{aligned}$$

On the other hand:

$$\begin{aligned} \sum_{i=1}^k q_i &= \sum_{i=1}^k \left( \frac{1-\beta}{m} + \beta \frac{2(m-i)+1}{m^2} \right) \\ &= \frac{k}{m} + \beta \frac{mk - k^2}{m^2} = \beta \frac{mk - k^2}{m^2} + \frac{mk - k^2}{m^2} + \frac{k^2}{m^2} \\ &= (1+\beta) \frac{mk - k^2}{m^2} + \frac{k^2}{m^2}. \end{aligned}$$

From the equations above, it is easy to see that for any  $k$  and  $\beta = 2\gamma$ :

$$\sum_{i=1}^k p_i \geq \sum_{i=1}^k q_i.$$

Theorem 2.9 from [25] gives us that :

$$\mathbb{E}[\Gamma_q(t+1)|y(t)] \leq \Gamma(t) \left(1 - \frac{\alpha\epsilon}{4m}\right) + c, \quad (2)$$

for  $\epsilon = \beta/12 = \gamma/6$  and  $c = c(\epsilon) = \text{poly}(1/\epsilon)$ .

The fact that  $\mathbb{E}[\Gamma(t+1)|y_t] = \mathbb{E}[\Gamma_p(t+1)|y(t)] \leq \mathbb{E}[\Gamma_q(t+1)|y(t)]$  gives us the lemma.  $\square$

If  $op_t$  inserts into the lesser loaded bin with probability at most  $\frac{1}{2}$ , we assume the worst scenario. That is, we assume that  $op_t$  always inserts into the more loaded bin and we try to bound the expected potential increase for that case. For this, again let us assume that the weight vector  $y(t)$  is ordered such that  $y_1(t) \leq y_2(t) \leq \dots \leq y_m(t)$  and let  $p_i$ ,  $1 \leq i \leq m$  be a probability that bin  $i$  is chosen. In this case,  $p_i = \frac{2i-1}{m^2}$ .

We now fix some constants: let  $\lambda = 1$  and  $S = 1$ , so that for every  $z < \lambda/2$  we have  $e^z < 2S$ . Also, to be consistent with the above lemma we fix  $\epsilon = \beta/12 = \gamma/6$ . At this point, we also fix  $\alpha = \min(\frac{\lambda}{2}, \frac{\epsilon}{6S})$ . This allows us to prove the following lemma:

LEMMA6.5. *If  $op_t$  is a bad operation, then:*

$$\mathbb{E}[\Gamma(t+1)|y(t)] \leq \left(1 + \frac{2}{m} \left(\alpha + \frac{\epsilon}{6}\right)\right) \Gamma(t). \quad (3)$$

PROOF. First, we consider what is expected change in  $\Phi$ . Let  $\Phi_i(t) = \exp(\alpha y_i(t))$ . We have two cases here. If bin  $i$  is chosen,

then the change is:

$$\begin{aligned}
\Delta\Phi_i &= \Phi_i(t+1) - \Phi_i(t) = \\
&= \exp\left(\alpha\left(y_i(t) + \left(1 - \frac{1}{m}\right)\right)\right) - \exp\left(\alpha y_i(t)\right) \\
&= \exp\left(\alpha y_i(t)\right) \left( \exp\left(\alpha\left(1 - \frac{1}{m}\right)\right) - 1 \right) \\
&\stackrel{*}{=} \exp\left(\alpha y_i(t)\right) \left( 1 + \alpha\left(1 - \frac{1}{m}\right) + e^\zeta \left(\alpha\left(1 - \frac{1}{m}\right)\right)^2 / 2 - 1 \right) \\
&\leq \exp\left(\alpha y_i(t)\right) \left( \alpha\left(1 - \frac{1}{m}\right) + S\alpha^2 \right).
\end{aligned} \tag{4}$$

where in (\*) we used the Taylor expansion of the exponential around 0 and the fact that since  $\zeta \in [0, \alpha(1 - \frac{1}{m})]$ , we have that  $e^\zeta < 2$ . Using similar arguments we can prove that, when some other bin  $j \neq i$  is chosen:

$$\Delta\Phi_i \leq e^{\alpha y_i(t)} \left( -\frac{\alpha}{m} + S\frac{\alpha^2}{m^2} \right). \tag{5}$$

Therefore, we get that:

$$\begin{aligned}
\mathbb{E} \left[ \Delta\Phi_i | y(t) \right] &= p_i \left( \alpha\left(1 - \frac{1}{m}\right) + S\alpha^2 \right) e^{\alpha y_i(t)} \\
&\quad + (1 - p_i) \left( -\frac{\alpha}{m} + S\frac{\alpha^2}{m^2} \right) e^{\alpha y_i(t)} \\
&\leq e^{\alpha y_i(t)} \left( p_i \left( \alpha + S\alpha^2 \right) - \frac{\alpha}{m} + S\left(\frac{\alpha}{m}\right)^2 \right) \\
&\leq e^{\alpha y_i(t)} \left( p_i \left( \alpha + S\alpha^2 \right) \right) \leq e^{\alpha y_i} \frac{2}{m} \left( \alpha + \frac{\epsilon}{6} \right),
\end{aligned} \tag{6}$$

where we used that  $p_i \leq \frac{2}{m}$  and  $S\alpha \leq \frac{\epsilon}{6}$ . This gives us that:

$$\mathbb{E}[\Delta\Phi | y(t)] = \sum_{i=1}^m \mathbb{E}[\Delta\Phi_i | y(t)] \leq \frac{2}{m} \left( \alpha + \frac{\epsilon}{6} \right) \Phi(t). \tag{7}$$

In a similar way, we can prove that:

$$\mathbb{E}[\Delta\Psi | y(t)] \leq \sum_{i=1}^m \left( p_i \left( -\alpha + S\alpha^2 \right) + \left( \frac{\alpha}{m} + S\frac{\alpha^2}{m^2} \right) \right) e^{-\alpha y_i}. \tag{8}$$

Since  $\alpha S \leq 1$ , we get that  $\mathbb{E}[\Delta\Psi | y(t)] \leq \frac{2}{m} \alpha \Psi(t)$ . Combining this with inequality 7 gives us the Lemma.  $\square$

Now we consider  $Cn$  consecutive operations and prove that at most  $n$  of them can be bad:

LEMMA6.6. *For any  $t$ , we have that:  $|t' : t \leq t' \leq t + Cn - 1, \ell_{t'} > Cn| < n$ .*

PROOF. We argue by contradiction. Let us assume that the number of bad operations is at least  $n$ . By the pigeonhole principle, there exist bad operations  $op_i$  and  $op_j$ ,  $t \leq i < j \leq t + Cn - 1$ , which are performed by the same thread. This means that since these operations are not concurrent, we have that  $s_j > f_i = i$ . Thus, we get a contradiction:  $Cn \leq \ell_j = |t' : s_j \leq t' < f_j = j| \leq j - i < Cn$ .  $\square$

**Endgame.** With all this machinery in place, we proceed to prove the following.

LEMMA6.7. *Given any time  $t$ , we have*

$$\mathbb{E}[\Gamma(t)] \leq e^2 \frac{8c}{\alpha\epsilon} m.$$

PROOF. We will proceed by induction on  $t$ . We will first prove that, if  $\Gamma(t) \leq e \frac{8c}{\alpha\epsilon}$ , then  $\mathbb{E}[\Gamma(t+Cn) | \Gamma(t)] \leq e \frac{8c}{\alpha\epsilon}$ .

We have two cases. The first is if there exists a time  $\tau \in [t, t+Cn]$  such that  $\Gamma(\tau) \leq \frac{7c}{\alpha\epsilon} m$ . Let us now focus on bounding the maximum expected value of  $\Gamma(t+3n)$  in this case. First notice that the maximum expected increase of  $\Gamma$  because of a good step is an additive  $c$  factor. The expected value of  $\Gamma$  after a bad step is upper bounded a multiplicative  $(1 + \frac{2}{m}(\alpha + \frac{\epsilon}{6}))$  factor. Hence, by Lemma 6.6 and using the fact that  $C \geq 3$ , the expected maximum value of  $\Gamma$  at  $t+Cn$  is at most

$$\left( \frac{7c}{\alpha\epsilon} m + c(C-1)n \right) \left( 1 + \frac{2}{m} \left( \alpha + \frac{\epsilon}{6} \right) \right)^n \leq e \frac{8c}{\alpha\epsilon} m.$$

The second case is if there exists no such time in  $[t, t+Cn]$ , meaning that  $\Gamma(\tau) > \frac{7c}{\alpha\epsilon} m, \forall \tau \in [t, t+Cn]$ . Then, by Lemma 6.4, we have that, at each good step,

$$\mathbb{E}[\Gamma(t+1) | y(t)] \leq \Gamma(t) \left( 1 - \frac{\alpha\epsilon}{12m} \right). \tag{9}$$

Hence, we can expand the recursion to upper bound the change in  $\Gamma$  between  $t$  and  $t+Cn$  as

$$\mathbb{E}[\Gamma(t+Cn) | y(t)] \leq \Gamma(t) \left( 1 - \frac{\alpha\epsilon}{12m} \right)^{(C-1)n} \left( 1 + \frac{2}{m} \left( \alpha + \frac{\epsilon}{6} \right) \right)^n. \tag{10}$$

This last expression is upper bounded by  $\Gamma(t)$  if  $C \geq 1 + 36/\epsilon$ , which concludes the proof of our first claim above. Hence, at the end of each interval of  $Cn$  additional operations, the expected potential cannot exceed  $e \frac{8c}{\alpha\epsilon}$ . To complete the proof, we notice that the value that  $\Gamma$  attains *inside* the interval of size  $Cn$  occurs if  $n$  bad steps occur in succession immediately after its start. However, the maximum value that  $\Gamma$  can attain is upper bounded as

$$\mathbb{E}[\Gamma(t+n) | y(t)] \leq \Gamma(t) \left( 1 + \frac{2}{m} \left( \alpha + \frac{\epsilon}{6} \right) \right)^n \leq e^2 \frac{8c}{\alpha\epsilon} m. \tag{11}$$

This concludes the proof of the Lemma.  $\square$

**The Constant  $C$ .** A sufficient setting for  $C$  for the analysis to hold is  $C \geq 1024$ , yielding  $m \geq 8192n$ .

The following claim completes the proof of Theorem 6.1.

LEMMA6.8. *Fix a large constant  $C$ . Given an oblivious adversary,  $m$  distributed counters and  $n$  threads with  $m \geq 8Cn$ , for any time  $t$  in the execution of the approximate counter algorithm the counter returns a value that is at most  $O(m \log m)$  away from the number of increment operations which completed up to time  $t$ , in expectation. Moreover, for any  $t$  and all  $R$  sufficiently large, we have*

$$\Pr \left[ \exists j : \left| m \cdot x_j(t) - m \cdot \mu_j(t) \right| > Rm \log m \right] \leq m^{-\Omega(R)}.$$

PROOF. The proof is similar to [25] (the main difficulty was to reach asymptotically the same potential upper bound). We aim to bound  $\text{Gap}(t)$ , the maximum gap between the weight of two bins at a step.

By choosing  $C$  sufficiently large, we have that  $\alpha^{-1}, \epsilon^{-1} = \Theta(1)$  in Lemma 6.7. We first prove the bound in expectation. Note that Lemma 6.7 implies that  $\mathbb{E}[\Phi(t)] = O(m)$  and  $\mathbb{E}[\Psi(t)] = O(m)$  for

all  $t$ . Let  $\max(t)$  denote the maximum weight of any bin at time  $t$ , and let  $\min(t)$  be the minimum weight of any bin. Then, we have

$$\begin{aligned} \alpha \mathbb{E}[\max(t) - \mu(t)] &= \log \exp(\mathbb{E}[\alpha(\max(t) - \mu(t))]) \\ &\stackrel{(a)}{\leq} \log \mathbb{E}[\exp(\alpha(\max(t) - \mu(t)))] \\ &\stackrel{(b)}{\leq} \log \mathbb{E}[\Phi(t)] \leq O(\log m), \end{aligned}$$

where (a) follows from Jensen's inequality, and (b) follows from the definition of  $\Phi$ . Similarly, we have  $\mathbb{E}[\mu(t) - \min(t)] \leq O(\log m)$ . Since the true value of the counter at time  $t$  is  $m \cdot \mu(t)$ , these two statements imply that for all  $j$ , we have  $\mathbb{E}[|m \cdot x_j(t) - m \cdot \mu(t)|] \leq O(m \log m)$ , as desired.

We now prove the high probability bound. Observe that if  $\max(t) - \mu(t) > R \log m$ , then we have  $\Gamma(t) \geq \Phi(t) \geq e^{\alpha R \log m}$ . Hence,

$$\begin{aligned} \Pr[\max(t) - \mu(t) > R \log m] &\leq \Pr[\Phi(t) \geq e^{\alpha R \log m}] \\ &\leq \frac{O(m)}{e^{\alpha R \log m}} \\ &\leq m^{-O(R)}. \end{aligned}$$

Similarly,  $\Pr[\mu(t) - \min(t) > R \log m] \leq m^{-O(R)}$ .

Combining these two guarantees with a union bound immediately yields the desired guarantee.  $\square$

## 6.4 Application to Concurrent Relaxed Queues

We now provide an overview of how to extend the analysis to imply distributional linearizability guarantees in concurrent executions for a variant of the MultiQueue process analyzed by [4]. This process performs concurrent *Enqueue()* and *Dequeue()* operations on the relaxed concurrent queue. We assume that we are given a set of  $m$  linearizable priority queues such that each supports *Add(e, p)*, where  $p$  is the priority of the element  $e$ , and *DeleteMin*, which deletes the element with the highest priority. We also assume that each processor  $i$  has access to the clock which gives an absolute time, and which is consistent amongst all the processors, that is, if processor  $i$  reads the clock in the linearization before processor  $j$ , then processor  $i$ 's value is smaller. Such an assumption is realistic; recent Intel processors support the RDTSC hardware operation, which provides this functionality for cores on the same socket.

To enqueue, a thread reads the wall clock, chooses a random priority queue, and adds the element to that priority queue with priority given by the time. To dequeue, we choose two random priority queues, find the one having a higher priority element on top, and delete from that priority queue. In case two processes enqueue to the same priority queue concurrently, their clock values will ensure a consistent ordering, handled by the internal implementation of the priority queues.

Following [4], we make the additional assumption that every execution is *prefixed*, intuitively implying that no *Dequeue()* operation ever observes an empty priority queue.

We sketch the analysis of this process for  $m = 8Cn$ . Initially, as in [4], we couple this process with an exponential process, where if *Dequeue()* removes element with timestamp  $t$  from priority queue  $i$ , the new head element of  $i$  has timestamp  $t + \text{Exp}(1/m)$ . Here,  $\text{Exp}(1/m)$  is an exponentially distributed random variable with mean  $m$ . Notice that if we divide timestamps by  $m$ , we have

that timestamps increase by 1 in expectation. Next, we can show that Lemmas 6.4 and 6.5, hold even if instead of incrementing counter(timestamp) by 1, we increment it by a random variable with mean 1 (the proof is given in the full version of the paper [3]). Lemma 6.7 yields that the potential  $\Gamma$  defined for the exponential process above (after we divide timestamps by  $m$ ) is linear in expectation. This allows us to use Theorem 4 in [4], which states that in this case expected *rank gap* between the smallest timestamp head element of any queue and the largest timestamp head element of any queue in the original process is at most  $O(m \log m)$ . A complete proof of this claim is given in the full version of this paper [3].

## 7 EXPERIMENTAL RESULTS

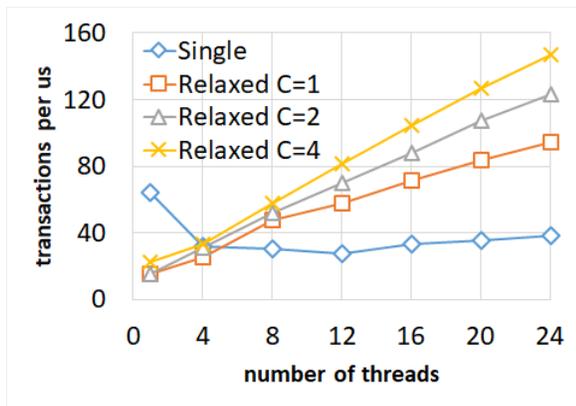
**Setup.** Our experiments were run on an Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 24 threads, and 128GB of RAM. In all of our experiments, we pinned threads to avoid unnecessary context switches. Hyperthreading is only used with more than 12 threads. The machine runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 6.3.0 with compilation options `-std=c++11 -mcx16 -O3`.

**Synthetic Benchmarks.** We implemented and benchmarked the MultiCounter algorithm on a multicore machine. To test the behavior under contention, threads continually increment the counter value using the two-choice process. We use no synchronization other than the atomic fetch and increment instruction for the update. Figure 1(a) shows the scalability results, while Figure 1(b) shows the "quality" guarantees of the implementation in terms of values returned by the counter over time, as well as maximum gap between bins over time. Quality is measured in a single-threaded execution, for 64 counters. (Recording quality accurately in a concurrent execution appears complicated, as it is not clear how to order the concurrent read steps.)

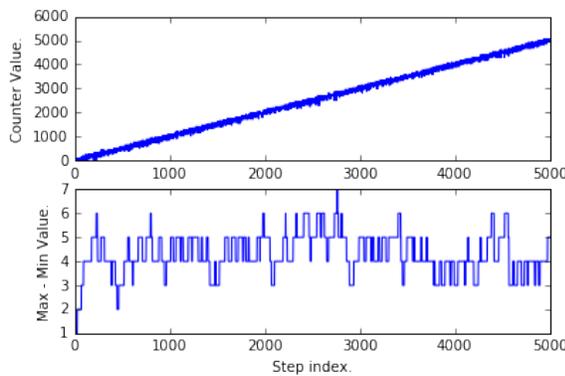
## 8 CONCLUSIONS AND FUTURE WORK

We have presented the first concurrent analysis of the two-choice load-balancing process, showing that this classic randomized algorithm is in fact robust to asynchrony under an oblivious adversary. Our analysis extends existing tools, namely [25], in non-trivial ways, in particular by showing that the potential analysis can withstand adversarially corrupted updates. Our results have non-trivial practical applications, as they show that a popular set of randomized concurrent data structures in fact provide strong probabilistic guarantees in arbitrary executions, which we express via a new correctness condition called *distributional linearizability*. This inspires a scalable approximate counting mechanism, trading off contention and exactness guarantees.

An immediate direction of future work is to reduce the large constant gap between the number of bins  $m$  and the number of threads  $n$ . We did not specifically optimize for this gap in the current version. It is interesting to also ask whether the process will preserve its properties even under high contention, e.g.  $m \leq n$ . The main reason for which we assume that  $m \geq 8Cn$  is to withstand adversarial executions in which the adversary schedules whole "blocks" of updates, which effectively reset the distribution of bin loads. Threads acting with stale information after such a block perform choices which are effectively random (or worse than random).



(a) Scalability of the concurrent counter for different values of the ratio  $C$  between counters and # threads.



(b) Quality results for the concurrent counter in a single-threaded execution. The  $x$  axis is # increments.

## Figure 1: Experimental Results for the Concurrent Counter

## 9 ACKNOWLEDGMENTS

Giorgi Nadiradze was supported by the Swiss National Fund Ambizione Project PZ00P2 161375. Justin Kopinsky was supported by the NSF under grants IIS-1447786 and CCF-1563880. Jerry Li was supported by NSF CAREER Award CCF-1453261, a Google Faculty Research Award, and an NSF Fellowship. Trevor Brown was funded in part by a post-doctoral fellowship from the Natural Sciences and Engineering Research Council of Canada.

## REFERENCES

- [1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.
- [2] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18:1–18:51, June 2014.
- [3] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. Distributionally linearizable data structures, 2018. arXiv:1804.01018.
- [4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. *arXiv preprint arXiv:1706.04178*, 2017. To appear in PODC 2017.
- [5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, USA, 2015. ACM.
- [6] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.
- [7] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.
- [8] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. Café: Scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC '11, pages 475–488, Berlin, Heidelberg, 2011. Springer-Verlag.
- [9] Petra Berenbrink, Artur Czumaj, Matthias Englert, Tom Friedetzky, and Lars Nagel. Multiple-choice balanced allocation in (almost) parallel. In *APPROX-RANDOM*, pages 411–422. Springer, 2012.
- [10] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 745–754, New York, NY, USA, 2000. ACM.
- [11] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, December 2008.
- [12] N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, March 1992.
- [13] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, Montreal, QC, Canada, pages 43–52, 2013.
- [14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.
- [15] Andreas Haas, Thomas A Henzinger, Andreas Holzer, Christoph M Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability. *arXiv preprint arXiv:1502.07118*, 2015.
- [16] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 17:1–17:9, 2013.
- [17] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. *SIGPLAN Not.*, 48(1):317–328, January 2013.
- [18] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [19] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.
- [20] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. In *European Conference on Parallel Processing*, pages 209–221. Springer, 2015.
- [21] Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing. *Distrib. Comput.*, 29(2):127–142, April 2016.
- [22] Michael Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.
- [23] Michael David Mitzenmacher. *The Power of Two Random Choices in Randomized Load Balancing*. PhD thesis, PhD thesis, Graduate Division of the University of California at Berkeley, 1996.
- [24] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.
- [25] Yuval Peres, Kunal Talwar, and Udi Wieder. Graphical balanced allocations and the  $1 + \beta$ -choice process. *Random Struct. Algorithms*, 47(4):760–775, December 2015.
- [26] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.
- [27] Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.
- [28] P. Sanders. Randomized priority queues for fast parallel access. *Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures*, 49:86–97, 1998.
- [29] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.
- [30] Kunal Talwar and Udi Wieder. Balanced allocations: The weighted case. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 256–265, New York, NY, USA, 2007. ACM.
- [31] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *ACM SIGPLAN Notices*, volume 50, pages 277–278. ACM, 2015.