Atomic snapshots in $O(\log^3 n)$ steps using randomized helping*

James Aspnes† Keren Censor-Hillel‡

September 7, 2018

Abstract

A randomized construction of single-writer snapshot objects from atomic registers is given. The cost of each snapshot operation is $O(\log^3 n)$ atomic register steps with high probability, where n is the number of processes, even against an adaptive adversary. This is an exponential improvement on the linear cost of the previous best known snapshot construction [12, 17] and on the linear lower bound for deterministic constructions [18], and does not require limiting the number of updates as in previous sublinear constructions [5]. One of the main ingredients in the construction is a novel *randomized helping* technique that allows out-of-date processes to obtain up-to-date information.

Our construction can be adapted to implement snapshots in a message-passing system. While a direct adaptation using the Attiya-Bar-Noy-Dolev construction gives a cost of $O(\log^3 n)$ time and $O(n \log^3 n)$ messages per operation with high probability, we show that exploiting the inherent parallelism of a message-passing system can eliminate the need for randomized helping and reduce the complexity to $O(\log^2 n)$ time and $O(n \log^2 n)$ messages per operation in the worst case. This implementation includes an $O(1)$-time, $O(n)$-message construction of an unbounded-value max register that may be of independent interest.

*A preliminary version of this work appeared in the proceedings of the 27th International Symposium on Distributed Computing (DISC), pages 254–268, 2013. We mention that the implementation of the unbounded max array given in the conference version does not give the claimed step complexity, and is replaced here by a different construction.

†Yale University, Department of Computer Science. Supported in part by NSF grants CCF-0916389, CCF-1637385, and CCF-1650596.

‡Technion, Department of Computer Science.

1 Introduction

An **atomic snapshot** object allows processes to obtain the entire contents of a shared array as an atomic operation. The first known wait-free implementations of snapshot from atomic registers [2, 3, 9] required $\Theta(n^2)$ steps to carry out a snapshot with n processes; subsequent work [12, 17] reduced this cost to $O(n)$, which was shown to be optimal in the worst case for non-blocking deterministic algorithms by Jayanti *et al.* [18].

The Jayanti *et al.* lower bound applies to any object that is **perturbable**, which roughly means that the output of a future read operation can be affected by inserting new updates into an execution, no matter how these updates are linearized with respect to incomplete operations already present in the execution.[1] In addition to snapshots, other examples of perturbable objects are **max registers** [4] (which support `ReadMax` operations that return the largest value written by earlier `WriteMax` operations) and counters.

Limitations of the lower bound technique of [18] became apparent with the development of wait-free sublinear-complexity **limited-use** and **bounded-value** variants of these objects. These included deterministic implementations of max registers and counters [4], and later of snapshot objects [5]. In each case, these implementations had step complexity polylogarithmic in the number of operations applied to them.[2] These objects still have linear cost in the worst case, but the worst case is reached only after exponentially many operations.

The dependence on the range of possible values was shown to be necessary initially for max registers [4], where a lower bound of $\min(\lceil \log m \rceil, n-1)$ was shown[3] for an m-bounded-value max register, and later for a variety of objects [7], which satisfy a perturbability condition similar to that used in the Jayanti *et al.* lower bound. For randomized implementations, a lower bound of $\Omega(\log n / \log(w \log n))$ for `ReadMax` operations was shown [4], where w is an upper bound for the complexity of `WriteMax` operations. That is, for every correct implementation, there is a strategy for an adaptive adversary that yields, with high probability, a schedule that contains either a `WriteMax` operation that takes w steps or a `ReadMax` operation that takes $\Omega(\log n / \log(w \log n))$ steps.

[1] The full definition is a bit more technical; see [18] for details.

[2] In the case of snapshot, this requires both registers large enough to hold a complete snapshot and the cooperation of updaters. The assumption of large registers may be avoidable for some applications of snapshot where only summary information is needed.

[3] All of the logarithms taken in this paper are with base 2.

We show that significantly larger lower bounds are impossible. Using a new randomized helping procedure along with a simple unbounded max register implementation, it is possible to adapt the max register implementation of [4] so that every operation finishes in $O(\log n)$ steps with high probability[4], regardless of the number of previous operations, provided the max register value does not change too quickly.

The significance of this improvement goes beyond the implementation of max registers. Max registers were previously used in [5] to implement a bounded-value **2-component max array**, a kind of specialized snapshot object that contains two fields, each of which is a multiwriter max register, and that provides the ability to take a snapshot of these two max registers atomically. This was a key component in implementing a limited-use snapshot, which the authors of [5] constructed using a tree of 2-component max arrays to consolidate partial snapshots into a single snapshot readable at the root (this algorithm is described in more detail Section 5). A central result of the current work is removing the limited-use requirement for this algorithm. This requires building an unlimited-use 2-component max array.

Unfortunately, applying our techniques directly to the 2-component max array of [5], as was done in the conference version of this paper, does not give $O(\log^2 n)$ step complexity. Instead, we provide a different construction of a 2-component max array, which uses the same randomized helping technique as in our max register construction to get $O(\log^2 n)$ expected step complexity for all operation,s provided that its component values do not change too quickly. We use this new implementation of a 2-component max array in the snapshot algorithm of [5] to get a snapshot object whose operations require $O(\log^3 n)$ steps with high probability, even when an unbounded number of operations are applied to it.

We further adapt our construction to a message-passing system, showing that we can obtain an improvement in performance, compared to a naïve conversion using the atomic register simulation of Attiya, Bar-Noy, and Dolev [11] (commonly known as *ABD*), while also removing the need for randomization. This adaptation includes a deterministic construction of an unbounded-value max register for a message-passing system that runs in $O(1)$ time and $O(n)$ messages per operation, assuming $t < n/2$ crash failures.

Most of the results in this paper previously appeared in DISC 2013 [8]. New results include a different 2-component max array construction, the

[4]Throughout the paper, we use the term *with high probability* to refer to an event that happens with probability at least $1 - 1/n^c$, for any fixed constant $c \geq 1$.

extension to message-passing, and the round-robin technique used in the helping mechanism.

1.1 Model

We consider two models: a standard wait-free shared-memory model in which processes $\{p_0, \ldots, p_{n-1}\}$ communicate by reading and writing multi-writer multi-reader atomic registers, and a standard asynchronous message-passing model tolerating $t < n/2$ crash failures. In both models, we represent concurrency by interleaving.

In the shared-memory model, each non-faulty process is always carrying out some high-level operation, and has a pending step on some register in each state. In each state, an adversary chooses which of these pending steps is applied next. When a non-faulty process completes its operation, the adversary chooses which operation it performs next, including the values of any arguments to that operation.

In the message-passing model, processes send and receive messages, with the adversary determining the timing of message delivery subject to the requirements that any message sent by a non-faulty process is eventually delivered. To ensure interleaving, we also require that no two messages are received at the same time. The adversary also determines when a process takes a step, which means that it sends a message.

Processes are probabilistic, and may use **local coins** to decide what to do next. A local coin operation provides a random value to the process that performs it, and is not visible to other processes.

1.1.1 The adversary

We assume an **adaptive adversary** that can observe the internal states of the processes and thus effectively observes the outcome of each local coin as soon as it is generated, although it cannot predict the outcome of future coins. Formally, the adversary corresponds to a scheduler that chooses the next pending event based on the local states of all processes and the content of the shared memory. Moreover, the adversary chooses the sequence of operations on the implementation that each process performs. In contrast, an **oblivious adversary** must choose which process carries out the next operation at each step without reference to the state of the system. That is, an oblivious adversary is a scheduler that chooses the sequence of steps in advance before the execution starts. We do not consider an oblivious adversary for our algorithms, although an oblivious adversary plays a role

in some of the lower bounds referenced in the paper. Naturally, such lower bounds apply *a fortiori* to the stronger adaptive adversary.

1.1.2 Implementations and linearizability

A **sequential object** provides one or more **operations** that may update the state of the object and return a value. An **implementation** of an object provides, for each operation, an algorithm that allows it to be carried out using atomic register operations (in a shared-memory system) or messages (in a message-passing system). Each instance of an operation is **invoked** through an explicit step internal to some process, and completes by some explicit return step. The **execution interval** of an operation is the interval between the invocation and return steps of the operation.

An implementation is **linearizable** [16] if, for any execution H of the implementation, it is possible to order all operations that have returned and some subset of the operations that have been invoked but not returned to obtain an execution S of the sequential object, such that each operation that returns in H returns the same value in S, and whenever π_1 and π_2 are operations in H such that π_1 finishes before π_2 starts, π_1 appears before π_2 in S. If these properties hold, S is called a **linearization** of H. A common method of constructing a linearization of H is to assign each operation π to be included in S a **linearization point** somewhere within the execution interval of π, and order the operations by their linearization points.

1.1.3 Complexity measures

Fixing a particular adversary removes all nondeterminism from the system other than the outcomes of the local coin-flips. This means that we can assign a probability to each execution equal to the probability of the sequence of coin-flips that occurs in this execution. When measuring the complexity of an algorithm, we need to take this distribution over executions into account.

For example, to compute the **individual step complexity** of an operation in a single execution, we look at the number of steps carried out by the process performing the operation in that execution. The **worst-case expected individual step complexity** is then the maximum over all adversaries of the expected number of such operations. If, for all adversary strategies, the probability of the operation taking more than $O(f(n))$ steps is bounded by n^{-c} for any constant $c > 0$, then we write that the operation has individual step complexity $O(f(n))$ **with high probability**. We will

generally be interested in high-probability bounds.

In the message-passing model, we consider both message complexity (the number of messages sent and received by all processes while carrying out some operation) and time complexity (the time from the start to finish of an operation assuming that all messages are delivered after at most one time unit and that each message is sent as soon as possible). As in the shared-memory case, we may consider either bounds on the expected values of these measures for a worst-case adversary, or bounds that hold with high probability for all adversaries.

A complication in both models is that it is impossible to measure the complexity of an operation that does not occur, and the choice of which operations to invoke and when is also under the control of the adversary. To avoid this complication, when discussing the complexity of a particular operation, we condition on the partial execution up until the operation starts.

1.2 Some standard objects

A **max register** [4] supports two operations: A `WriteMax` operation, which has an input v, and a `ReadMax` operation, which returns the value of the largest input to any `WriteMax` operation that is linearized before it. Equivalently, a max register is a register where `ReadMax` simply returns the state of the register while `WriteMax` will only overwrite a value with a larger value.

A **2-component max array** [5] supports a `MaxUpdate` operation, which specifies a value and a component, and a `MaxScan` operation, which returns the maximum values written to each of the two components in all `MaxUpdate` operations linearized before it.

A **single-writer snapshot** [2] object consists of n locations and supports two operations. One operation is an `Update` operation with argument v by process i, which does not have a return value. The other operation is a `Scan` operation, which takes no input and returns n values, such that the value corresponding to each location i is the value of the last `Update` operation by process i that has been linearized before the `Scan` operation. A **multi-writer snapshot** object is similar to the above, with the difference that the number of locations can be arbitrary, and each location can be updated by every process.

1.2.1 Read and write operations

Randomized helping as defined in Section 2 works by having operations that change the state of the object also read the state of the object and store the result for use by operations that read the object. In defining this mechanism, it is helpful to adopt an object-oriented approach to object operations. Given an object O, we will let $O.\texttt{write}$ and $O.\texttt{read}$ be the appropriate write and read operations for this object. Applying this to the objects described above:

1. If M is a max register, then $M.\texttt{write}$ is `WriteMax` and $M.\texttt{read}$ is `ReadMax`.
2. If A is a max array, then $A.\texttt{write}$ is `MaxUpdate` and $A.\texttt{read}$ is `MaxScan`.
3. If S is a snapshot object, then $S.\texttt{write}$ is `Update` and $S.\texttt{read}$ is `Scan`.

Note that in some cases, the `write` operation may take additional parameters beyond the value written.

When convenient, we will use the assignment operator $\leftarrow$ as syntactic sugar for `write` or `read` operations. An expression of the form

$$x \leftarrow O$$

is interpreted as

$$x \leftarrow O.\texttt{read}()$$

while an expression of the form

$$O \leftarrow x$$

is interpreted as

$$O.\texttt{write}(x).$$

1.3 Previous constructions

Before giving more detail on our construction, we give a quick review of the previous work on which it is based. The basic building block of the limited-use snapshot construction in [5] is a bounded-value 2-component max array, which is in turn built from bounded-value max registers. To directly build an unlimited-use snapshot object, we need an unbounded-value 2-component max array, which in turn requires an unbounded-value max register.

The max register construction of [4] consists of a tree of *switches*, which are one-bit registers that initially hold the value 0 and can only be set to 1. Each leaf represents a value for the register. A `WriteMax`(v) operation follows the path towards the leaf representing value v and sets the switches along it for which the path descends to the right, from bottom to top. A `ReadMax` operation follows the rightmost path of set switches, descending to the left child if a switch is unset, and returns the value corresponding to the leaf it reached, which is the largest value written. The problem with an unbounded-value max register according to this construction is that the length of an operation reading the rightmost path in the infinite tree construction is unbounded. This is because this operation is searching for the first node on the rightmost path whose switch is 0, and the depth of this node depends on the values that have been written, which are now unbounded. Even worse, such an operation is not guaranteed to be wait-free, as it might not terminate if new `WriteMax` operations keep coming in with greater values, forcing it to continue moving down the tree to the right. To handle this, the tree in [4] is truncated and combined with an unlimited-use single-writer snapshot object with $O(n)$ step complexity per operation. The latter is used for larger values in order to bound the number of steps. Formally, this means that at some threshold level, the node on the rightmost path of switches no longer points to an infinite subtree of switches but rather to this unlimited-use single-writer snapshot object. All `WriteMax` operations writing values that are at least the threshold set the switch at this node after writing their value to the snapshot object. All `ReadMax` operations accessing this node continue by performing a `Scan` operation on the snapshot object. If the threshold is set to $\Theta(2^n)$, this gives a step complexity of $O(\min(\log v, n))$ for a `ReadMax` or `WriteMax` operation with the value v.

The bounded-value 2-component max-array construction of [5] builds upon the above max register construction by combining the trees of the two components in a subtle manner. The data structure consists of a main tree, corresponding to the tree of the first component. The tree of the second component is embedded in the main tree at *every* node. That is, each switch of the main tree is associated with a separate copy of the tree of the second component. Writing to the first component is done by writing to the main tree, ignoring the copies of the second component at the switches. Writing to the second component is done by writing to the copy associated with the root of the main tree. The coordination between the pairs of values is left for the `MaxScan` operations. Such an operation travels down the main tree in order to read the value of the first component, while dragging down the maximal value it reads for the second component along its path. It is

proven in [5] that this implementation gives a linearizable bounded-value 2-component max array.

In the snapshot algorithm of [5], processes are organized into a tree, where each internal node in the tree holds a 2-component max array and each leaf represents the updates provided by a single process. This 2-component max array is used to track the number of updates in the node's left and right subtrees. Scans of the max array provide a sequence of pairs of counts that are non-decreasing in both sides, which gives a consistent interleaving of update events in the two subtrees. This interleaving is used by updaters to propagate partial scans to the root of each subtree representing all updates carried out by processes in that subtree. At the root of the whole tree stands a single max register that indexes a table containing complete snapshots.

1.4 Our contributions

Our first contribution is a construction of an unbounded-value max register in which the step complexity of each operation is $O(\log n + \log k)$ in expectation and with high probability, under the assumption that the maximum value written to the register increases by at most k per write operation (this is formalized more carefully in Section 3.1).

In effect, this assumption of **bounded increments** substitutes for the assumption of bounded values required by the previous max register construction of [4]. Because that construction encodes each value of a max register as a leaf in a binary tree of one-bit registers, either the tree must be made finite, giving a bounded-value max register; or the tree must have unbounded depth, giving an unbounded-value max register whose step complexity increases indefinitely as larger values are written.

Our construction, described in Section 3, avoids this difficulty by combining a simple unbounded-tree max register consisting of a single long path of bits, with an unbounded sequence of bounded max registers that we can think of as hanging off of this path. The long path encodes the high-order bits of the max register, while the low-order bits are stored in one of the bounded max registers. Under the assumption of bounded increments, the high-order bits do not change often. So a process performing a read operation can find the last bit set quickly as long as not too many writes have occurred since the last read by the same process. The low-order bits can then be read from the appropriate bounded max register.

But if many writes have occurred, the reader will need to get help from the writers. We implement this help using a novel technique of **randomized helping**, described in Section 2. The key idea is to keep the cost of obtaining

help down by having the writers supply help and then leave their identities in an n^3-element array of recently active processes, so that, once enough writes have occurred, choosing a random location in the array is likely to identify a writer that has supplied a help value recently. This reduces the cost of finding a good value from the $\Theta(n)$ cost of a collect to an expected $O(1)$ cost of picking a good random location. The price is that it may take a polynomial number of helping operations to occur before the active array fills up with pointers to good values.

It is here that the division between high-order bits and low-order bits comes into play. Because helping is used only for the reader to catch up on the high-order bits, under the assumption of bounded increments we can ensure that many writers will have attempted to help the reader if the high-order bits change significantly. This gives us a construction of a max register where, with high probability, the step complexity is logarithmic in the number of processes n and the increment bound k.

We extend this approach to get a similar construction of an unbounded max array that gets polylogarithmic step complexity with high probability, again under the assumption of bounded increments (Section 4). Here we implement a snapshot of the high-order bits of the array components using a simple double collect, with the low-order bits represented by a bounded max array as implemented in [5]. As in the classic snapshot of Afek *et al.* [2], a write operation must first perform a snapshot and leave the resulting value as help. Bounded increments mean that the high-order bits change infrequently: so either the double collect succeeds after $O(1)$ attempts, or sufficiently many write operations have occurred that the reader can obtain a snapshot value indirectly from one of these writes with high probability. The resulting cost of a read or write operation is $O(\log^2 n + \log^2 k)$, where k is now the bound on the increase of either array component as the result of a single write.

Plugging these two constructions into the limited-used snapshot implementation of [5] gives an implementation of an unlimited-use single-writer snapshot object with an $O(\log^3 n)$ step complexity (with high probability) for updating or scanning the object (Section 5).

Finally, we adapt our implementation to a message-passing system (Section 6), obtaining a single-writer snapshot implementation with $O(\log^2 n)$ time and $O(n \log^2 n)$ messages per operation. While a direct use of the classic Attiya-Bar-Noy-Dolev (ABD) simulation [11] gives a complexity of $O(\log^3 n)$ time and $O(n \log^3 n)$ messages per operation, we show how to leverage the capability of a message-passing system to consolidate multiple messages into a single one in order to reduce the complexity. This is done

by simply collecting all elements of an n-element array of help values in the same $O(1)$ time and $O(n)$ messages as is needed for a single read/write register, instead of sampling an n^3-element array as our shared-memory implementation does.

2 Randomized helping

Traditional helping mechanisms are designed to turn obstruction-free protocols into wait-free protocols by allowing fast processes that are obstructing the progress of slow processes to carry out operations on the slow processes' behalf. This is particularly useful for **read operations**, where the state of the object is not updated, because an operation that does change the state can carry out a read without having to know in advance whether another process will need the value or not.

The problem with helping in a deterministic algorithm is that it is expensive for the recipient to identify which process has provided it with help. For example, the helping mechanisms of Herlihy's universal construction [15] or the atomic snapshot of Afek *et al.* [2] require a process seeking help to perform a collect over n registers, which is far too expensive for our applications.

We describe instead a **randomized helping** mechanism, based on sampling from an active array containing the identities of recently active processes. This allows the recipient of help to find it quickly—in $O(1)$ operations with high probability—at the cost of potentially having to wait for polynomially many updates to fill the active array. This will work best for objects where reads do not need help unless many updates occur.

Help values are provided through an array of atomic registers $\mathsf{helpVal}[i]$, one for each helping process p_i. Each helped process p_j has a register $\mathsf{TS}[j]$ which records increasing timestamps. For each pair of processes p_i and p_j, the register $\mathsf{helpTS}[i][j]$ is a copy of a value from $\mathsf{TS}[j]$; when this value equals the current value of $\mathsf{TS}[j]$, then the value in $\mathsf{helpVal}[i]$ was read by p_i no earlier than when p_j wrote $\mathsf{TS}[j]$. This allows p_j to discard out-of-date help values.[5]

[5]To keep things simple, our implementation assumes unbounded timestamps. With some ingenuity, it might be possible to replace these timestamps with a bilateral handshaking mechanism similar to that used by Afek *et al.* [2]. However, a naive implementation would require the reader to do $\Theta(n)$ work each time it asks for help to invalidate old handshakes. We leave the question of bounding the size of registers used to coordinate helping to future work.

These arrays in principle allow a reader p_j to find help by checking all n locations in helpTS where the second index is j, but this takes $\Theta(n)$ time. So the remaining part of the mechanism is an array $\mathsf{active}[0 \ldots n^3 - 1]$, in no particular order, of ids of processes that have recently provided help.

We assume that these arrays are attached to some underlying object A that supports an $A.\texttt{read}()$ operation whose return value does not depend on the process calling it and that does not alter the state of A. The helping mechanism consists of three new operations on A: an operation $A.\texttt{incrementTS}()$ that increments the timestamp for the current process in helpTS; an operation $A.\texttt{giveHelp}()$ that is used by operations that update the state of A to provide help; and an operation $A.\texttt{takeHelp}()$ used by readers to obtain help.

Pseudocode for these operations is given in Algorithm 1. A read operation that may need help should call $A.\texttt{incrementTS}$ when it starts, and then call $A.\texttt{takeHelp}$ from time to time if it is not making progress. Other operations on A that may obstruct read operations should call $A.\texttt{giveHelp}$ once each time they are invoked.

The intuition behind the mechanism is that, given enough calls to $A.\texttt{giveHelp}$, $A.\mathsf{active}$ will eventually fill up with the ids of processes that have recently read both A and $A.\mathsf{TS}[i]$ for any particular process p_i. So if p_i cannot finish quickly enough on its own, randomly sampling a location in $A.\mathsf{active}$ will soon give it the identity of some process p_j that provided usable help in $A.\mathsf{helpVal}[j]$ and $A.\mathsf{helpTS}[j][i]$. We formalize this intuition in the following lemma:

Lemma 2.1. *Fix a partial execution Ξ of Algorithm 1 that ends at the point t where some process p_i starts an $A.\texttt{takeHelp}$ operation τ. Suppose there is a point s in Ξ such that at least $n^4 + 1$ calls to $A.\texttt{giveHelp}$ start in the interval $[s, t]$. Fix an adversary strategy for extending Ξ. Conditioning on Ξ and the adversary strategy:*

1. *With probability at least $1 - 1/n^2$, if τ returns, it returns a value value that was returned by a call to $A.\texttt{read}$ that started no earlier than s.*

2. *If τ returns* **true** *following at least one call by p_i to $A.\texttt{incrementTS}$, then the value returned by τ was previously returned by a call to $A.\texttt{read}$ that started no earlier than the start of p_i's last call to $A.\texttt{incrementTS}$ and finished no later than τ.*

3. *If p_i's last call to $A.\texttt{incrementTS}$ before t finished before s, then with probability at least $1 - 1/n$, if τ returns, τ returns* **true**.

```
1  Shared data:
2  A.TS[0 ... n − 1], an array of registers holding timestamps, initially
     all 0
3  A.helpTS[0 ... n − 1][0 ... n − 1], an array of registers holding
     timestamps, initially all 0
4  A.helpVal[0 ... n − 1], an array of registers holding help values,
     initially all ⊥
5  A.active[0 ... n^3 − 1], an array of registers holding process ids, initially
     arbitrary
6  Persistent local data for each process:
7  toHelp, a process id in the range 0 ... n − 1, initially arbitrary
8  activeNext, an index into A.active in the range 0 ... n^3 − 1, initially
     arbitrary
9  procedure A.incrementTS()
10     A.TS[i] ← A.TS[i] + 1;
11 procedure A.giveHelp()
12     t ← A.TS[toHelp];
13     v ← A.read();
14     A.helpVal[i] ← v;
15     A.helpTS[i][toHelp] ← t;
16     A.active[activeNext[i]] ← i;
17     toHelp ← (toHelp + 1) mod n;
18     activeNext ← (activeNext + 1) mod n^3;
       // Return value read, in case it is useful
19     return v;
20 procedure A.takeHelp()
21     r ← random(0 ... n^3 − 1);
22     j ← A.active[r];
23     t ← A.helpTS[j][i];
24     v ← A.helpVal[j];
25     return (t ≥ A.TS[i], v);
```

Algorithm 1: Randomized helping applied to object A. Code for process p_i.

Proof. We prove each claim in turn.

For the first claim, the adversary's goal will be to get p_i to read a process id j from $A.\mathsf{active}[r]$ such that either $A.\mathsf{helpVal}[j]$ contains a value from a call to $A.\texttt{read}$ that does not start after s, or $A.\mathsf{helpTS}[j][i]$ contains a timestamp less than $A.\mathsf{TS}[i]$. We wish to show that most choices of r will make this impossible.

By the Pigeonhole Principle, among the $n^4 + 1$ $A.\texttt{giveHelp}$ operations that start in $[s, t]$, there is some process p_j that executes at least $n^3 + 1$ of them. The first n^3 of these $n^3 + 1$ operations both start and finish in $[s, t]$, and between these n^3 operations, p_j writes j to every location in $A.\mathsf{active}$. Each of these operations also writes a value obtained from a call to $A.\texttt{read}$ that starts after s to $A.\mathsf{helpVal}[j]$, and among the first n of these operations is one that reads $A.\mathsf{TS}[i]$ and copies its value to $A.\mathsf{helpTS}[j][i]$.

In the unlikely case that p_j were the only process to write to $A.\mathsf{active}$, p_i would observe j in $A.\mathsf{active}[r]$ and the claims of the lemma would follow. But we must consider the possibility that some other process overwrites j before p_i reads $A.\mathsf{active}[r]$. Fortunately no such process p_k can write to too many locations in $A.\mathsf{active}$ before itself putting a recent value in $A.\mathsf{helpVal}[k]$ or updating the timestamp in $A.\mathsf{helpTS}[k][i]$, as we now demonstrate.

Let a_k be the value of p_k's $\mathsf{activeNext}$ variable at s. If p_k already has an $A.\texttt{giveHelp}$ operation in progress at s, it may write to $A.\mathsf{active}[a_k]$ without executing an $A.\texttt{read}$ operation starting after s or reading $A.\mathsf{TS}[i]$ after s. But if p_k writes to any other location $A.\mathsf{active}[a']$, it must first complete an $A.\texttt{giveHelp}$ operation to increment its toHelp variable, then start a new $A.\texttt{giveHelp}$ operation—after s—and run it at least to Line 16. This new operation will call $A.\texttt{read}$ in Line 13 and write the resulting value to $A.\mathsf{helpVal}[k]$ in Line 14 before reaching Line 16. It follows that if p_i chooses any value r that is not equal to a_k for some process k, it will read a process id ℓ such that $A.\mathsf{helpVal}[\ell]$ was returned by an $A.\texttt{read}$ operation that started no earlier than s. Because the a_k values are determined by s, well before p_i choose r, they are independent of the choice of r and the chance that r is equal to one of these at most n values is at most $n/n^3 = 1/n^2$. This proves the first claim in the lemma.

For the second claim, observe that only p_i can increment $A.\mathsf{TS}[i]$ in $A.\texttt{incrementTS}$, and that τ will return true only if p_i reads a timestamp T equal to $A.\mathsf{TS}[i]$ in $A.\mathsf{helpTS}[j][i]$ for the process p_j it chooses to get help from. For T to appear in $A.\mathsf{helpTS}[j][i]$, p_j must have read it from $A.\mathsf{TS}[i]$ during a call to $A.\texttt{giveHelp}$ before calling $A.\texttt{read}$, writing the return value to $A.\mathsf{helpVal}[j]$, and finally setting $A.\mathsf{helpTS}[j][i]$ to T. So τ will either return the value obtained by this $A.\texttt{read}$ operation or by some later $A.\texttt{read}$

operation also called by A.`giveHelp`; in either case, the operation will start after the $A.\mathsf{TS}[i]$ was last incremented (and thus after the start of the last call to A.`incrementTS` by p_i) and before $A.\mathsf{helpVal}[j]$ is read by τ, which occurs before τ finishes.

For the third claim, we use a similar argument as for the first. Again let p_k be some process and a_k the value of its toHelp variable at s. Then for p_k to write to $A.\mathsf{active}[(a_k + n) \bmod n^3]$, it must reach Line 16 in $n + 1$ executions of A.`giveHelp`, of which the last n must start no earlier than s. Each of these n executions will copy $A.\mathsf{TS}[\ell]$ to $A.\mathsf{TS}[k][\ell]$ for some ℓ, and because p_k increments toHelp in each operation, one of these n values ℓ will be equal to i. It follows that any process p_k can write at most n distinct locations in $A.\mathsf{active}$ without copying $A.\mathsf{TS}[i]$ to $A.\mathsf{helpTS}[k][i]$. The set of such possible locations is determined by $a_0, \ldots, a_{n-1}$ and is again independent of p_i's choice of r. So the probability that p_i reads one of these at most n^2 possible locations is at most $n^2/n^3 = 1/n$. □

3 Unbounded-value max registers with bounded increments

Recall that a max register supports operations `WriteMax`(v) and `ReadMax`, where a `ReadMax` returns the value of the largest input to any `WriteMax` operation that is linearized before it. The purpose of a max register is typically to avoid lost updates, by ensuring that old values (tagged with smaller timestamps) cannot obscure newer values, regardless of the order in which they are written. In this section, we show how to construct an unbounded-value max register that is linearizable in all executions and wait-free with $O(\log n)$ step complexity with high probability in executions which satisfy a certain restriction which we refer to as **bounded increments**.

3.1 Bounded increments and sparse increments

Let $\{x_i\} = x_1, x_2, x_3 \ldots$ be a sequence of non-negative integer values. We will write that $\{x_i\}$ has k-**bounded increments** if $x_1 \leq k$, and for all $i > 1$,

$$x_i \leq k + \max_{j<i} x_j. \tag{1}$$

This says that the maximum value in the sequence so far can only increase by k at each step.

Iterating the definition gives:

Lemma 3.1. *If $x_1, x_2, \ldots$ is a sequence with k-bounded increments, then*

1. *For each i, $x_i \leq ki$.*
2. *For each i and each $j \geq i$, $x_j \leq x_i + k(j-i)$.*

Proof. By induction on i for the first case, and on $j - i$ for the second. □

For very slow-growing sequences, we use a different definition. A sequence $x_1, x_2, \ldots$ has **t-sparse increments** if it has 1-bounded increments and the maximum value increases no more often than every t steps. Formally this second condition says that for all integer values c, if x_i is the first element of the sequence that is at least c, and x_j is the first element of the sequence that is at least $c+1$, then $j \geq i + t$.

The purpose of these definitions is that we will demand that the sequence of values written to our max register will have bounded increments, and then use bounded max registers to hold the remainder modulo some fixed value m, leaving behind quotients that have sparse increments. These sparse increments mean that the max register holding the quotients will change so slowly that a reader that fails to catch up to the current value must have many concurrent writers, allowing it to get help. This technique exploits a connection between bounded increments of a sequence $\{x_i\}$ and sparse increments of the related sequence of quotients $\{\lfloor x_i/m \rfloor\}$, given in the following lemma.

Lemma 3.2. *Let $x_1, x_2, x_3, \ldots$ be a non-negative integer sequence with k-bounded increments. Let $m \geq 2k$. Then the sequence $\lfloor x_1/m \rfloor, \lfloor x_2/m \rfloor, \lfloor x_3/m \rfloor, \ldots$ has $(m/k - 1)$-sparse increments.*

Proof. First let us show that $\{\lfloor x_i/m \rfloor\}$ has 1-bounded increments. For x_1, it holds trivially that $\lfloor x_1/m \rfloor \leq \lfloor k/m \rfloor \leq 1$.

Suppose there is some $i > 1$ such that $\lfloor x_i/m \rfloor > 1 + \max_{j<i} \lfloor x_j/m \rfloor$. Then for all $j < i$, $\lfloor x_i/m \rfloor \geq 2 + \lfloor x_j/m \rfloor$. But then $x_i/m \geq \lfloor x_i/m \rfloor \geq 2 + \lfloor x_j/m \rfloor > 2 + x_j/m - 1 = 1 + x_j/m$. Multiply both sides by m to get $x_i > m + x_j \geq k + x_j$. Since this holds for all $j < i$, $x_i > k + \max_{j<i} x_j$, contradicting k-bounded increments for $\{x_i\}$.

Next, we must show that $\max_{j \leq i} \lfloor x_j/m \rfloor$ does not increase too often. Fix some integer c. Let i be the first index for which $\lfloor x_i/m \rfloor \geq c$ and let j be the first index for which $\lfloor x_j/m \rfloor \geq c+1$.

If $i = 1$, then $x_i = x_1 \leq k$ and thus $\lfloor x_1/m \rfloor \leq \lfloor k/m \rfloor = 0$. So in this case, j is the first index for which $\lfloor x_j/m \rfloor = 1$. But then $m \leq x_j$, and thus

Lemma 3.1 gives $m \leq jk$ or $j \geq m/k$. But then $j \geq i + (m/k - 1)$, as required.

If $i > 1$, we have $\lfloor x_{i-1}/m \rfloor < c$ and thus $x_{i-1} < cm$, giving $x_{i-1} \leq cm-1$. For x_j, we have $c + 1 \leq \lfloor x_j/m \rfloor \leq x_j/m$, giving $(c + 1)m \leq x_j$. We can bound x_j in the other direction with Lemma 3.1 to get $(c + 1)m \leq x_j \leq x_{i-1} + k \cdot (j - (i - 1)) \leq cm - 1 + k \cdot (j - (i - 1)) = cm + (k - 1) + k \cdot (j - i)$. Subtracting cm from both sides then gives $m \leq k-1+k \cdot (j-i)$. Solving for $j - i$ gives $j - i \geq (m - k + 1)/k > m/k - 1$, and thus $j \geq i + (m/k - 1)$. □

We will also need to take into account the effects of reordering caused by asynchrony. For example, suppose that we have a sequence $x_1, x_2, \ldots$ with k-bounded increments, that represents the inputs to a sequence of high-level operations. If these inputs are then passed on to lower-level operations, the adversary may be able to selectively delay some of the inputs to get a different permutation of the sequence. This permutation may no longer have k-bounded increments. A simple case would be the k-bounded-increment sequence $0, k, 2k$; if reordered as $0, 2k, k$, the value k no longer serves as a stepping stone between 0 and $2k$, and the new sequence has only $2k$-bounded increments.

Fortunately, we can bound the increments in the new sequence by considering the fact that the adversary can only delay values by holding them in one of n processes. This effectively creates a bounded buffer to be used for the reordering, where each new value in the original sequence $\{x_i\}$ is placed in the buffer one at a time, and the adversary is forced to remove a value for the output sequence before inserting a new value once the buffer reaches its capacity.[6]

Call a sequence $\{y_i\}$ an **n-buffered reordering** of a sequence $\{x_i\}$ if $\{y_i\}$ can be generated from $\{x_i\}$ by this process, where n is the capacity of the buffer. Formally, this means that for each element x_i, its corresponding element $y_{i'}$ is preceded by all but at most $n - 1$ elements x_j with $j < i$.

The following lemma holds:

Lemma 3.3. *If $\{y_i\}$ is an n-buffered reordering of a sequence $\{x_i\}$ with k-bounded increments, then $\{y_i\}$ has nk-bounded increments.*

Proof. For y_1, observe that it is among the first n elements of $\{x_i\}$, and so has value at most nk by Lemma 3.1.

[6] Such a **sorting buffer** or **reordering buffer** has been considered in the context of scheduling [19, 13, 1]; a similar idea, without an explicit name, also appears in the analysis of inserting random values into concurrent binary search trees [10, 14].

For later y_j, Consider some value $y_j = x_{i_0}$. If $x_{i_0} \geq k$, then k-bounded increments implies that there exists a position $i_1 < i_0$ such that $x_{i_0} \leq x_{i_1} + k$, which we can rearrange as $x_{i_1} \geq x_{i_0} - k$. By iterating this argument we obtain a sequence of positions $i_0, i_1, i_2, i_3, \ldots i_\ell$ such that each $x_{i_{j+1}} \geq x_{i_j} - k$, ending only when we reach $x_{i_\ell} = x_0$. A simple induction argument shows that $x_{i_j} \geq x_{i_0} - jk$ for each $j \leq \ell$.

Now consider some x_{i_0} that appears in the output sequence as y_j, where $j > 0$. At the time $x_{i_0} = y_j$ leaves the buffer, there are at most $n-1$ other values in the buffer. If $\ell \geq n$, this implies at least one of the n values $x_{i_1}, x_{i_2}, \ldots x_{i_n}$ must have previously left the buffer, appearing as $y_{j'}$ for some $j' < j$. In this case, $y_{j'} = x_{i_m}$ for $m \leq n$ satisfies $y_{j'} = x_{i_m} \geq x_{i_0} - mk \geq x_{i_0} - nk = y_j - nk$, and so $y_j \leq \max_{j'<j} y_{j'} + nk$. Alternatively, if $\ell < n$, then $x_1 = x_{i_\ell} \geq x_{i_0} - \ell k \geq x_{i_0} - (n-1)k$ implies $y_j = x_{i_0} \leq x_0 + (n-1)k \leq k + (n-1)k = nk \leq y_1 + nk \leq \max j' < j y_{j'}$. □

Finally, we prove the following technical lemma that shows that as long as we retain all left-to-right maxima, removing other values from a sequence with k-bounded increments yields a sequence that still has k-bounded increments. This will turn out to be necessary for analyzing our max register implementation in Algorithm 3.

Lemma 3.4. *Let $\{x_i\} = x_1, x_2, \ldots$ have k-bounded increments, and let $x_{i_1}, x_{i_2}, \ldots$ be a subsequence of $\{x_i\}$ with the property that x_1 appears as x_{i_j} for some j, and for each $i > 1$, if $x_i > \max_{i'<i} x_{i'}$, then $i = i_j$ for some j. Then $\left\{x_{i_j}\right\}$ has k-bounded increments.*

Proof. Because a subsequence preserves ordering, the first element x_1 can only appear as x_{i_1}. So $x_{i_1} = x_1 \leq k$.

For later elements, consider some x_{i_j} with $j > 1$. Then $x_{i_j} \leq k + \max_{i'<i_j} x_{i'}$. Consider the set S of all indices ℓ such that $\ell < i_j$ and x_ℓ is maximal among all values $x_{\ell'}$ for $\ell' < i_j$. Let i' be the smallest index in S. Then $x_{i'} > \max_{i''<i'} x_{i''}$, which means that $x_{i'} = x_{i_{j'}}$ for some j'. But since $i' < i$, j' must be less than j, and so $x_{i_j} \leq k + \max_{i_{j'}<i_j} x_{i_{j'}}$, proving the claim. □

3.2 Unbounded max register with sparse increments

We start by constructing an unbounded max register that is linearizable and wait-free in all executions, and that requires only $O(1)$ steps with high probability for both `read` and `write` operations if increments are sparse enough.

Pseudocode for this construction in given in Algorithm 2. The main data structure is an unbounded array of bits, and the value v is represented by setting all bits at positions less than v to 1. A `read` operation scans the array across increasing indices until it hits the first zero. A `write`(v) operation sets all bits at positions less than v to 1 starting at $v-1$ and working down to 0. For both operations, the process performing the operation uses both its memory of previous operations and help provided by the randomized helping mechanism in Algorithm 1 to reduce redundant steps. This is what gives both `read` and `write` operations constant step complexity with high probability if the values in the `write` operations do not rise too quickly.

```
1  Shared data:
2  A[0...]: unbounded array of one-bit atomic registers, initially 0
3  Shared data for Algorithm 1
4  Local data per process: integer top, initially 0
5  procedure A.write(v)
6      r ← A.giveHelp();
7      for i ← v − 1 down to r do
8          A[i] ← 1;
9  procedure A.read()
10     A.incrementTS();
11     while A[top] = 1 do
12         ⟨current, v⟩ ← A.takeHelp();
13         if current then
14             return v;
15         else
16             top ← max(top + 1, v);
17     return top;
```

Algorithm 2: Unbounded max register

While an A.`write` operation is in progress, the set of one bits in A may not be contiguous. However, we can prove the following invariant:

Lemma 3.5. *Following any partial execution of Algorithm 2, if either*

1. *Some process's* top *variable contains the value v, or*
2. *Some A.`read` operation has returned v,*

then $A[k] = 1$ for all $k < v$.

Proof. The proof is by induction on the length of the execution. After an empty execution, all top variables are 0 and no A.read operations have returned, and the claim holds trivially. For the induction step, we consider some partial execution Ξ, and imagine extending it by one step.

Note that no $A[k]$ is ever reset to 0, so we need only consider the effect of changes to top for some process, or values returned by A.read operations.

Write top_i for the value of top held by process p_i. If the step does not change top_i for any p_i, then $A[k] = 1$ for all $k < \mathsf{top}_i$ from the induction hypothesis. If the step does change top_i, then it must carry out Line 16. If it sets top_i to the value v previously returned by A.takeHelp, then v is a value returned by some A.read operation, and $A[k] = 1$ for all $k < v$ follows from the induction hypothesis. If instead top_i is equal to one plus the previous value v' of top_i, then (a) for all $k < v'$, $A[k] = 1$ (induction hypothesis), and (b) for $k = v' < \mathsf{top}_i$, $A[k] = 1$, because to reach Line 16, p_i must first observe $A[v'] = 1$ in the while loop test in Line 11. So the first part of the invariant holds.

The other case to consider is when the step causes an A.read operation to return. But the return value v is just top_i for some p_i, so $A[k] = 1$ for $k < v$ is immediate from $A[k] < v$ for $k < \mathsf{top}_i$. □

Lemma 3.5 relates the array of bits to values returned by A.read. The next lemma relates those same bits to values written by A.write:

Lemma 3.6. *Following any partial execution of Algorithm 2:*

1. *Let k be a position in A such that $A[k] = 1$ and $A[k+1] = 0$. Then at least one call to A.write$(k+1)$ appears in the execution.*

2. *If a call to A.write(v) finishes in the execution, $A[k] = 1$ for all $k < v$.*

Proof. For the first part, observe that bits in A are set only in A.write operations. If $A[k] = 1$ and $A[k+1] = 0$ following some partial execution, then there is an A.write operation that set $A[k] = 1$ but no A.write operation that set $A[k+1]$. Because an A.write(v) operation sets positions in descending order starting at $v-1$, $A[k]$ must have been set by some A.write operations with $v - 1 \geq k$. But $v - 1$ cannot be strictly greater than k, or else this operation would have set $A[k+1]$ before setting $A[k]$. So $v - 1 = k$ exactly, giving $v = k + 1$.

For the second part, in order for A.write(v) to finish, it must set $A[k]$ for all k with $r \leq k \leq v - 1$, where r is the value returned by A.giveHelp. But A.giveHelp returns the value returned by an A.read operation, so from Lemma 3.5, $A[k] = 1$ for all $k < r$. So for any $k < v$, either $k \geq r$ and $A[k]$

is set directly by the A.`write` operation, or $k < r$ and $A[k]$ was previously set by some other A.`write` operation. In either case, $A[k] = 1$. □

Using these invariants, we can show that the algorithm is linearizable without putting any restriction on the values in the `write` operations.

Lemma 3.7. *Algorithm 2 is a linearizable implementation of a max register.*

Proof. We use the approach of assigning a linearization point to each complete operation (and some incomplete operations) at some step during its execution interval, and linearize operations by the order of their linearization points. In the case where two or more operations are assigned the same linearization point (which only occurs with A.`read` operations that return the same value), we order these operations arbitrarily with respect to each other. Because each operation's linearization point lies within its execution interval, it is immediate that the linearization is consistent with the observed ordering of operations in the concurrent execution.

For an A.`read` operation ρ, the linearization point depends on whether the process observes $A[\mathsf{top}] = 0$ in Line 11 and returns in Line 17 or obtains an indirect value from the helping mechanism in Line 12 that it returns in Line 14. In the first case, we assign ρ the step at which it reads 0 from $A[\mathsf{top}]$. In the second, we assign ρ the linearization point of the `read` operation ρ' whose value is returned by A.`takeHelp`. In either case, we assign linearization points only to A.`read` operations that finish during the execution; incomplete A.`read` operations have no effect on the effects or return values of other operations and thus do not need to be included in the linearization.

It is easy to see that in the first case, the linearization point occurs within the execution interval of ρ. In the second case, we have from Lemma 2.1 that the execution interval of ρ' is nested within that of ρ. If ρ' returns a value directly by reading $A[\mathsf{top}] = 0$, then its linearization point (and thus that of ρ as well) lies within its execution interval and thus also within the execution interval of ρ. If not, then there is a third A.`read` operation ρ'' whose execution interval is nested within that of ρ' that supplies the linearization point. Because ρ finishes at a finite time, the sequence $\rho, \rho', \rho'', \ldots$ can include only finitely many operations, the lat of which will supply the linearization point of a read operation of $A[\mathsf{top}]$, which will lie within the execution interval of ρ.

For an A.`write`(v) operation ω, we define the linearization point as the first step within the execution interval of A.`write`(v) such that $A[k] = 1$ for all $k < v$. This step will either be the first step of ω (if the condition holds

before it starts), or will be a step of some A.`write` operation that sets the last missing bit in this interval. In either case, the linearization point for ω will never be the same as for an A.`read` operation, as the linearization point for any A.`read` operation is assigned to a step of some A.`read` operation.

That a linearization point exists for any complete A.`write`(v) operations follows from Lemma 3.6, as this condition holds when ω returns. For an incomplete A.`write` operation ω, such a step might not exist; we include such an operation in the linearization if and only if it does.

The linearization S is constructed by sorting all operations that have linearization points by their linearization points, breaking ties arbitrarily. Because the execution interval of each such operation contains its linearization point, whenever some operation σ precedes an operation τ in the concurrent execution, σ will also precede τ in S. This makes S a total extension of the observed partial order on operations.

We must also show that S satisfies the sequential specification of a max register, which means that the value v_ρ returned by any A.`read` operation ρ is equal to the maximum of the inputs to all A.`write` operations that linearize before it. For any ρ, let ρ' be the direct read that supplies both the return value and linearization point for ρ. Because A.`read` and A.`write` operations do not share linearization points, the sets of A.`write` operations linearized before ρ and ρ' are the same, regardless of tie-breaking. So it is enough to show that any direct read ρ' returns the maximum input of any A.`write` operations that linearize before it, or the initial value 0 if there are no such operations.

Let ρ' be a direct read. Its linearization point is a step where the process p_i executing ρ' reads $A[\mathsf{top}] = 0$, where `top` equals the return value v. From Lemma 3.5, at this step $A[k] = 1$ for all $k < v$. If $v > 0$, this in particular gives $A[v-1] = 1$ and $A[v] = 0$, so Lemma 3.6 says that the partial execution leading up to this step includes the start of an A.`write`(v) operation, and because $A[k] = 1$ for all $k < v$, its linearization point must appear in this partial execution and thus before the linearization point of ρ'.

On the other hand, for any A.`write`(v') operation with $v' > v$, at the linearization point of ρ' it is not yet the case that $A[k] = 1$ for all $k < v'$, because $A[v] = 0$. It follows that in the case $v > 0$, at least one A.`write`(v) operation is linearized before ρ' and only A.`write`(v') operations with $v' \leq v$ are linearized before ρ', so ρ' correctly returns v in the sequential execution.

This leaves the special case of $v = 0$. In this case, at the linearization point of ρ', $A[k] = 0$ for all k. There may or may not be any A.`write` operation ω that linearizes before ρ'. If there is no such operation, then ρ' correctly returns the initial value 0 of the max register. If there is some such

operation $A.\texttt{write}(v')$, then at its linearization point $A[k] = 1$ for all $k < v'$. But as $A[k] = 0$ for all k at this point, $v' = 0$, and again the return value 0 of ρ' is correct. □

In general, Algorithm 2 is not a particularly efficient implementation of a max register. But it works well enough assuming sparse increments.

Lemma 3.8. *In any execution of Algorithm 2 where the sequence $\{x_i\}$ of input values to $A.\texttt{write}$ operations has n^4-sparse increments, with the first such input at most 1, an $A.\texttt{write}$ or $A.\texttt{read}$ operation takes $O(1)$ steps in expectation and with high probability.*

Proof. Because an $A.\texttt{write}$ operation includes an $A.\texttt{read}$ as a subroutine (inside $A.\texttt{giveHelp}$), we start by bounding the cost of $A.\texttt{read}$.

Each $A.\texttt{read}$ takes $O(1)$ steps outside the loop in Lines 11–16, and the body of the loop takes $O(1)$ steps per iteration. So we need to demonstrate that the loop performs a constant number of iterations with high probability. Specifically, we need to show that the process p_i executing $A.\texttt{read}$ observes $A[\mathsf{top}] = 0$ after a constant number of passes through the loop with high probability.

We will start by showing that even if top starts far behind the current earlier position holding a zero, it usually catches up quickly.

Fix some $A.\texttt{read}$ operation ρ and let Ξ be the partial execution leading up to the start of ρ. Let x be the largest input to any $A.\texttt{write}$ operation that starts in Ξ. Because Ξ has n^4-sparse increments, either (a) fewer than $2n^4$ $A.\texttt{write}$ operations start in Ξ, or (b) the input to each of the last $2n^4$ $A.\texttt{write}$ operations that start in Ξ is at least $x - 2$.

In the first case, $k = O(1)$, and top starts at 0; so after $O(1)$ iterations of the loop, $\mathsf{top} = x$. In the second case, we need to use the helping mechanism.

Split the $2n^4$ $A.\texttt{write}$ operations into a prefix consisting of the first $n^4 - n - 1$ such operations to start and a suffix consisting of the last $n^4 + n + 1$ such operations to start. Let s be the point at which the first write operation in the suffix starts, and let t be the end of Ξ, which is also the start of ρ.

Observe that at most n operations that start within each of these intervals might not have returned by the end of the interval; thus the prefix includes at least $n^4 - 2n - 1$ $A.\texttt{write}$ operations that finish before s and the suffix contains at least $n^4 + 1$ $A.\texttt{write}$ operations that finish before t. For n sufficiently large, $n^4 - 2n - 1 > 0$, so at least one $A.\texttt{write}$ operation with input at least $x - 2$ finishes before s. Linearizability (Lemma 3.7) then implies that any $A.\texttt{read}$ operation that starts after s returns at least $x - 2$.

The $n^4 + 1$ completed A.`write` operations in the interval $[s, t]$ embed $n^4 + 1$ calls to A.`giveHelp`. Suppose we extend this interval to an interval $[s, t']$, where t' is the point at which some call to A.`takeHelp` that starts after t chooses its random location r; and similarly extend Ξ to an execution Ξ' that ends at t'. Then $[s, t']$ also includes at least $n^4 + 1$ calls to A.`giveHelp`. From Lemma 2.1, this gives at most an n^{-2} chance that each such call to A.`takeHelp` returns a value less than $x - 2$. Since this holds even conditioning on the outcome of previous calls to A.`takeHelp` (which appear in Ξ'), the probability that ℓ such calls all return values less than $x - 2$ is at most $n^{-2\ell}$. So for any fixed exponent c, choosing $\ell \geq c/2$ gives a probability of at most n^{-c} that all of these calls to A.`takeHelp` return values less than $x - 2$. If this event does not occur, then by the end of at most $\ell = O(1)$ such calls (and thus $O(1)$ iterations of the loop), `top` is at least $x - 2$, and after two more iterations, it is at least x.

We have thus established that `top` catches up to the value of the max register at the start of ρ in $O(1)$ steps with high probability. We now need to show that this allows ρ to return in an additional $O(1)$ steps with high probability, either because there are few concurrent writes, or because there are enough concurrent writes that ρ can obtain a current value from the helping array.

Suppose that ρ completes at least three iterations of the loop after `top` $= x$. Let Ξ'' be the partial execution leading up to the end of the third iteration, let s as before be the start of ρ, and let t'' be the end of Ξ''.

If $[s, t'']$ contains the start of at most $2n^4$ A.`write` operations, then the largest input to any such operation is at most $x + 2$. This follows from the fact that the first A.`write` with input $x+1$ starts after s, and by n^4-sparsity the first A.`write` with input $x+3$ must appear at least $2n^4$ positions later in the sequence. So for the duration of the third loop iteration, no A.`write` has started with an input greater than x^2, and in particular $A[x + 2] = 0$. But because the first two iterations increment `top` to $x + 2$, the third iteration will observe $A[\texttt{top}] = 0$ and return.

If $[s, t'']$ contains the start of more than $2n^4$ A.`write` operations, then for sufficiently large n it contains at least $n^4 + 1$ A.`write` operations that both start and finish in $[s, t'']$. Each such operation will make a call to A.`giveHelp`. Lemma 2.1 gives at least a $1 - 1/n$ chance that any call to A.`takeHelp` following t'' will return true, causing ρ to finish. Again we have a geometric distribution on the number of additional calls to A.`takeHelp`, so we execute at most $O(1)$ additional iterations of the loop with high probability.

Iterating the high-probability bounds on both phases gives the same

bounds in expectation.

This concludes the proof that A.`read` takes $O(1)$ steps in expectation and with high probability. This also gives $O(1)$ expected and high-probability bounds on the step complexity of A.`write`, since each A.`write` operation performs only $O(1)$ steps beyond those in its embedded A.`read`. □

Lemma 3.8 implies that Algorithm 2 is wait-free with probability 1. With some tinkering, it can be made wait-free in all executions, by adjusting A.`takeHelp` to alternate between random sampling of A.active and stepping through all locations deterministically. This approach embeds a traditional, deterministic helping backstop in the randomized helping mechanism. Because the deterministic backstop is very slow and provides little additional utility, we do not consider it in detail.

3.3 Unbounded max register with bounded increments

The unbounded max register of the preceding section has a serious deficiency: it is only efficient for sparse increments, but in typical applications we can hope only for bounded increments. In this section, we show how to convert the latter to the former. The essential idea is that we split each max register value into high-order and low-order components, and manage the low-order components with an unbounded sequence of "inner" m-valued max implementation as in [4]. These low-order max registers are multiplexed by a single "outer" high-order max register. Assuming the full max register inputs do not increase too quickly, their high-order bits will not change too often (Lemma 3.2), and so we will be able to implement the high-order max register at low cost using Algorithm 2.

Pseudocode for this construction is given in Algorithm 3.

One way to look at Algorithm 3 is that it generalizes the recursive max register construction of [4]. In that construction, the role of the outer max register is performed by the switch bit, which we can think of as a two-valued max register. It turns out that replacing this two-valued max register with an unbounded max register does not affect linearizability, as show in the following lemma.

Lemma 3.9. *Algorithm 3 is a linearizable implementation of a max register.*

Proof. For the purposes of the proof, we treat operations on M.outer and each M.inner$[i]$ as atomic steps. We will use the ordering of these steps, together with the values read from or written to M.outer to construct an explicit linearization S of a given concurrent execution H. This linearization

```
1  Shared data:
2  unbounded array M.inner[0...] of m-valued max registers, initially 0
3  unbounded max register M.outer, initially 0
4  procedure M.write(v)
5      if M.outer.read() ≤ ⌊v/m⌋ then
6          M.inner[⌊v/m⌋].write(v mod m);
7          M.outer.write(⌊v/m⌋);
8  procedure M.read()
9      h ← M.outer.read();
10     ℓ ← M.inner[h].read();
11     return h · m + ℓ;
```

Algorithm 3: Multiplexing bounded max registers with an unbounded max register

will include all operations that finish in H, but may omit some operations that do not finish in H.

The operations included in S will consist of:

1. All M.`read` operations that return in H. We will refer to such operations as **completed reads**.
2. All M.`write`(v) operations that perform M.inner$[h]$.`write` for some h in Line 6. We will refer to such operations as **effective writes**.
3. All M.`write`(v) operations that perform M.outer.`read` in Line 5 and obtain a value $h > \lfloor v/m \rfloor$. We will refer to such operations as **ineffective writes**, because they do do not take any action other than reading M.outer.

We now assign an **epoch** h_π to each operation π included in S:

1. If π is a completed read, its epoch h_π is the value h read from M.outer in Line 9.
2. If π is an effective write of value v_π, its epoch h_π is $\lfloor v_\pi/m \rfloor$.
3. If π is an ineffective write, its epoch h_π is the value read from M.outer in Line 5.

To order the operations in S, we first sort by epoch. For operations with the same epoch, we order using linearization points, which are explicit steps in each operation's execution:

1. If π is a completed read, its linearization point is the step at which it reads $M.\mathsf{inner}[h]$ in Line 10.

2. If π is an effective write of v_π, its linearization point is the step at which it writes $M.\mathsf{inner}[\lfloor v_\pi/m \rfloor]$ in Line 6.

3. If π is an ineffective write, its linearization point is the step at which it reads $M.\mathsf{outer}$ in Line 5.

To show consistency with the observed execution order in H, we need to show that if some operation π finishes before another operation π' starts, and both appear in S, then π precedes π' in S.

First let us show that if π finishes before π' starts, then $h_\pi \leq h_{\pi'}$. Observe that at some step during its execution interval, π either reads h_π from $M.\mathsf{outer}$ or writes h_π to $M.\mathsf{outer}$. (The former case occurs for reads and ineffective writes; the latter, for effective writes.) In either case, there is some step during the execution interval of π where $M.\mathsf{outer}$ holds h_π. For π', if π' is a read or ineffective write, then $h_{\pi'}$ is a value read from $M.\mathsf{outer}$ at some step during the execution interval of π', and in particular at some step after the step of π where $M.\mathsf{outer}$ holds h_π. From the specification of a max register it follows that $h_\pi \leq h_{\pi'}$. If instead π' is an effective write of some value $v_{\pi'}$, then because π' writes $M.\mathsf{inner}[\lfloor v_{\pi'}/m \rfloor]$, it must first read a value h from $M.\mathsf{outer}$ such that $h \leq h_{\pi'} = \lfloor v'_\pi/m \rfloor$. Again we have $h_\pi \leq h \leq h_{\pi'}$.

This implies that ordering by epoch never contradicts the observed execution order in H. Within an epoch, because we are ordering by linearization points that occur within the execution interval of each operation, if π precedes π' in H then π must also precede π' in S. So the order of operations S is consistent with their partial order in H.

Next, we must show that S is in fact a correct sequential execution of a max register. This requires demonstrating that the return value of each $M.\texttt{read}$ operation ρ is the maximum input of any $M.\texttt{write}$ operation ω that is linearized before ρ, or 0 if there is no such operation.

Fix some $M.\texttt{read}$ operation ρ. Let ρ return $v = h \cdot m + \ell$, where $0 \leq \ell < m$. From inspection of the code for $M.\texttt{read}$, we see that $h = h_\rho$ and ℓ is a value read from $M.\mathsf{inner}[h]$. Any $M.\texttt{write}$ operation ω that linearizes before ρ must have $h_\omega \leq h$.

We consider two cases, depending on whether or not some $M.\texttt{write}$ operation writes $M.\mathsf{outer}$ before ρ reads it.

If no $M.\texttt{write}$ operation writes $M.\mathsf{outer}$ before ρ reads it, then $h = 0$. It is easy to see in this case that no ineffective write ω can linearize before ρ, because this can occur only if $\lfloor v_\omega/m \rfloor < h_\omega \leq 0$, and v_ω cannot be negative.

The set of effective writes that linearize before ρ consists of those $M.\texttt{write}$ operations ω such that $h_\omega = 0$ and ω writes $M.\mathsf{inner}[0]$ before ρ reads $M.\mathsf{inner}[0]$. If this set is empty, ρ reads 0 from $M.\mathsf{inner}[0]$ and correctly returns $v = 0 \cdot m + 0 = 0$. If not, ℓ is the maximum over all such ω of $v_\omega \bmod m = v_\omega$, and $v = 0 \cdot m + \ell$ is the maximum of v_ω over these writes. Again ρ returns the correct value.

Now suppose instead that at least one $M.\texttt{write}$ operation writes $M.\mathsf{outer}$ before ρ reads it. Among this set there must be some ω that supplies the maximum value $h = h_\omega = \lfloor v_\omega/m \rfloor$. Before ω does so, it must must write $v_\omega \bmod m$ to $M.\mathsf{inner}[h]$. The linearization point of ω is this write to $M.\mathsf{inner}[h]$, which precedes ρ's read of $M.\mathsf{outer}$ and thus also ρ's read of $M.\mathsf{inner}[h]$; it follows that there is at least one effective write ω with $h_\omega = h$ that linearizes before ρ. The set of all effective writes with $h_\omega = h$ that linearize before ρ consists of precisely those effective writes that write to $M.\mathsf{inner}[h]$ before ρ reads $M.\mathsf{inner}[h]$. So ℓ is the maximum of $v_\omega \bmod m$ among all effective writes ω with $h_\omega = h$ that linearize before ρ, and $v = h \cdot m + \ell$ is the maximum of v_ω among all such effective writes.

We must now show that v is the maximum over all writes linearized before ρ, including ineffective writes and effective writes ω with $h_\omega < h$. The latter class is easily dismissed, as any such effective write has $v_\omega = h_\omega \cdot m + \ell_\omega$, where $\ell_\omega < m$, so $v_\omega < (h_\omega + 1) \cdot m \leq h \cdot m \leq v$. For an ineffective write ω, to linearize before ρ, it must have $\lfloor v/\omega \rfloor < h_\omega \leq h$. But then $v_\omega < h \cdot m \leq v$, and again ω has no effect on v. This shows that the previous argument in fact demonstrates that ρ returns the maximum input value of all writes, effective or not, that linearize before ρ.

We have thus shown that every $M.\texttt{read}$ operation returns the correct value in S. This concludes the proof that S is a linearization of H, and that any execution of Algorithm 3 is linearizable. □

To implement Algorithm 3, we must supply implementations of the unbounded max register $M.\mathsf{outer}$ and the m-valued max registers $M.\mathsf{inner}[i]$. For each $M.\mathsf{inner}[i]$, we can use the m-valued max register of [4]; this gives a cost of $O(\log m)$ steps per operation. For $M.\mathsf{outer}$, we use the implementation in Algorithm 2. Showing that this works efficiently requires showing that the sequence of values $\lfloor v/m \rfloor$ written to $M.\mathsf{outer}$ by operations reaching Line 6 has n^4-sparse increments, for a suitable choice of m.

Lemma 3.10. *Consider an execution of Algorithm 3 in which the sequence of inputs to all $M.\texttt{write}$ operations, ordered by the invocation of these operations, is given by $v_1, v_2, \ldots$. Fix some k, and suppose that the sequence $v_1, v_2, \ldots$ has k-bounded increments. Let $m \geq n^2 k$.*

Then the sequence of inputs to M.outer.write operations in Line 7, ordered by invocations of M.outer.write, has t-sparse increments, where $t = m/(n^2k) - 1$.

Proof. Let $\omega_1, \omega_2, \ldots$ be the sequence of M.write operations with corresponding inputs $v_1, v_2, \ldots$. Let $\omega'_1, \omega'_2, \ldots$ be the sequence of M.write operations that execute M.outer.read in Line 5, in the order that they execute this line, and let $v'_1, v'_2, \ldots$ be the inputs to these operations. Then $v'_1, v'_2, \ldots$ is an n-buffered reordering of $v_1, v_2, \ldots$, and thus has nk-bounded increments by Lemma 3.3.

We now consider the subsequence $\{\omega''_i\} = \omega''_1, \omega''_2, \ldots$ of $\{\omega'_i\}$ containing only those operations that reach the body of the if statement, and let v''_i be the input to ω''_i. Suppose that $v'_i \geq \max_{i'<i} v'_{i'}$. Then $\lfloor v'_i/m \rfloor \geq \max_{i'<i} v'_{i'}$, and in particular $\lfloor v'_i/m \rfloor$ is greater than or equal to $\lfloor v'_{i'}/m \rfloor$ for any operation $\omega_{i'}$ that writes $\lfloor v'_{i'}/m \rfloor$ to M.outer before ω_i reads M.outer. It follows that any such ω'_i is included in $\{\omega''_i\}$, and so its corresponding input v'_i is included in $\{v''_i\}$. So the subsequence $v''_1, v''_2, \ldots$ satisfies the conditions of Lemma 3.4, and thus also has nk-bounded increments.

Next, consider the sequence $\{\omega'''_i\}$ of operations in $\{\omega''_i\}$ that reach Line 7, in the order that they invoke the M.outer.write operation in this line. This is an n-buffered reordering of $\{\omega''_i\}$, so the sequence $v'''_1, v'''_2, \ldots$ is an n-buffered reordering of $v''_1, v''_2, \ldots$. Applying Lemma 3.3 a second time shows that $v'''_1, v'''_2, \ldots$ has (n^2k)-bounded increments.

Finally, observe that the sequence of inputs to M.outer.write in this execution is given by $\lfloor v'''_1/m \rfloor, \lfloor v'''_2/m \rfloor \ldots$. Since $v'''_1 \leq n^2k + 0 = n^2k$, the first input $\lfloor v'''_1/m \rfloor \leq \lfloor n^2k/m \rfloor \leq 1$. Since $v'''_1, v'''_2, \ldots$ has (n^2k)-bounded increments, the sequence of inputs has $(m/(n^2k) - 1)$-sparse increments by Lemma 3.2. □

Using Lemmas 3.9 and 3.10, we can give a full characterization of the behavior of Algorithm 3:

Theorem 3.11. *Fix some k, and let $m \geq (n^4 + 1)(n^2k)$ be polynomial in n and k. Consider an instance M of Algorithm 3 where M.outer is implemented using Algorithm 2, and each M.inner[i] is implemented using the m-valued max register of [4]. Let $v_1, v_2, \ldots$ enumerate the inputs to M.write operations in some execution of M, and suppose that the sequence $v_1, v_2, \ldots$ has k-bounded increments. Then M is a linearizable implementation of an unbounded max register, in which each operation has step complexity $O(\log n + \log k)$ in expectation and with high probability.*

Proof. Linearizability is established in Lemma 3.9.

For the step complexity, Lemma 3.10 shows that writes to $M.\mathsf{outer}$ have t-sparse increments where $t = m/(n^2k) - 1 \geq (n^4+1)(n^2k)/(n^2k) - 1 = n^4$. Lemma 3.8 then shows that each operation on $M.\mathsf{outer}$ runs in $O(1)$ steps in expectation and with high probability. We also have that each operation on $M.\mathsf{inner}[i]$ runs in $O(\log m) = O(\log n + \log k)$ steps. Since each operation on M involves $O(1)$ operations on $M.\mathsf{outer}$ and $M.\mathsf{inner}$, the bound follows. □

4 Unbounded max array with bounded increments

Algorithm 4 gives a linearizable implementation of an unbounded 2-component max array, which is efficient for an appropriate choice of m provided the sequences of input values to $A.\texttt{write}$ operations to each of the two sides of the max array have bounded increments.

The essential idea is similar to that of Algorithm 3: we combine a simple implementation of a max array for the high-order parts $\langle \lfloor \ell/m \rfloor, \lfloor r/m \rfloor \rangle$ of the max array values $\langle \ell, r \rangle$ with an unbounded array of $(m \times m)$-bounded max arrays for the low-order parts $\langle \ell \bmod m, r \bmod m \rangle$.

For the high-order parts, we use a double collect over two unbounded max registers $A.\mathsf{raw}[\mathsf{left}]$ and $A.\mathsf{raw}[\mathsf{right}]$. This double collect either succeeds by obtaining two collects with the same high-order parts, or fails because too many $A.\texttt{write}$ operations caused the high-order parts to change. If m is large enough, this second outcome can only occur $O(1)$ times before the randomized helping mechanism is likely to return a usable value.

For the low-order parts, we define an epoch $h = \lfloor \ell/m \rfloor + \lfloor r/m \rfloor$ for each tuple $\langle \ell, r \rangle$ of values, and argue that in any execution of Algorithm 4, this epoch uniquely determines $\lfloor \ell/m \rfloor$ and $\lfloor r/m \rfloor$ for any pair of values $\langle \ell, r \rangle$ returned by an $A.\texttt{read}$ operation. Within each epoch h, we ensure consistency of the remainders $\langle \ell \bmod m, r \bmod m \rangle$ using a bounded max register $A.\mathsf{low}[h]$. This has cost $O(\log^2 m)$ per operation if we use the bounded 2-component max array construction of [5].

We refer to an $A.\texttt{read}$ operation that returns in Line 20 a value it constructs itself as a **direct read**. We refer to an $A.\texttt{read}$ operation that returns in Line 24 a value that it obtains from $A.\texttt{takeHelp}$ as an **indirect read**.

We will show correctness by first linearizing all the direct reads and $A.\texttt{write}$ operations that finish or affect the return values of $A.\texttt{read}$ operations, and then including the indirect reads after the direct reads that supply their values. We will then analyze the complexity of each operation under the assumption of bounded increments. This will give us the following

```
1  Shared data:
2  A.raw[left], A.raw[right]: unbounded max registers, initially 0.
3  A.low[0..]: unbounded array of (m × m)-bounded max arrays, initially
     all (0, 0)
4  Shared data for Algorithm 1
5  procedure A.write[side](v)
6      A.raw[side].write(v);
7      A.giveHelp();
8  procedure A.read()
9      A.incrementTS();
10     while true do
           // Double collect
11         ℓ1 ← A.raw[left];
12         r1 ← A.raw[right];
13         ℓ2 ← A.raw[left];
14         r2 ← A.raw[right];
15         if ⌊ℓ1/m⌋ = ⌊ℓ2/m⌋ and ⌊r1/m⌋ = ⌊r2/m⌋ then
               // High parts match
16             h ← ⌊ℓ2/m⌋ + ⌊r1/m⌋;
17             A.low[h][left] ← ℓ2 mod m;
18             A.low[h][right] ← r1 mod m;
19             ⟨ℓ′, r′⟩ ← A.low[h].read();
20             return ⟨⌊ℓ2/m⌋ · m + ℓ′, ⌊r1/m⌋ · m + r′⟩;
21         else
               // Try help
22             ⟨current, v⟩ ← A.takeHelp();
23             if current then
24                 return v;
```

Algorithm 4: Unbounded max array

theorem.

Theorem 4.1. *Algorithm 4 is a linearizable implementation of a max array, such that, for an appropriate choice of m, in any execution where the sequences of input values to calls to A.write[left] and A.write[right] both have k-bounded increments, each A.write and A.read operation completes in $O(\log^2 n + \log^2 k)$ steps in expectation and with high probability.*

As the proof of Theorem 4.1 is somewhat involved, we break it up into

a sequence of lemmas in the following sections.

4.1 Constructing a linearization ordering

We assume that $A.\mathsf{raw}[\mathsf{left}]$, $A.\mathsf{raw}[\mathsf{right}]$ are linearizable max register implementations and that each $A.\mathsf{low}[h]$ is a linearizable bounded max array implementation. Because linearizable implementations are indistinguishable from objects with atomic operations, we will treat operations on these objects as atomic, and will refer to the state of an object after a sequence of such atomic operations as if it were the actual state of the object during the execution.

This allows us to define an **epoch** h_σ that holds after each step. The epoch of a step σ is the epoch of the values $\langle A.\mathsf{raw}[\mathsf{left}], A.\mathsf{raw}[\mathsf{right}]\rangle$ following σ, which is just the value of $\lfloor A.\mathsf{raw}[\mathsf{left}]\rfloor + \lfloor A.\mathsf{raw}[\mathsf{right}]\rfloor$ following σ. Some basic properties of the epoch are immediate from the fact that max register values are non-decreasing over time:

Lemma 4.2. *Let α and β be steps of an execution of Algorithm 4 such that α precedes β, and let h_α and h_β be the epochs of α and β. Then $h_\alpha \le h_\beta$, and if $h_\alpha = h_\beta$ then $\lfloor A.\mathsf{raw}[\mathsf{left}]/m\rfloor$ and $\lfloor A.\mathsf{raw}[\mathsf{right}]/m\rfloor$ do not change between α and β.*

Proof. The first claim follows from the fact that $\lfloor x/m\rfloor$ and addition are both monotone functions. The second claim follows from the fact that $\lfloor \ell/m\rfloor + \lfloor r/m\rfloor$ can only equal $\lfloor \ell'/m\rfloor + \lfloor r'/m\rfloor$ when $\lfloor \ell/m\rfloor \le \lfloor \ell'/m\rfloor$ and $\lfloor r/m\rfloor \le \lfloor r'/m\rfloor$ if neither inequality is strict. □

Because it is not possible to read both max registers simultaneously, in general we do not expect operations to be able to compute the epoch of their steps. But when successful, the double collect in $A.\texttt{read}$ acts like a snapshot for the high parts that make up the epoch:

Lemma 4.3. *Consider an execution of Algorithm 4 in which some $A.\texttt{read}$ operation ρ reaches Line 16. Let ℓ_2 and r_1 be the values of these variables in ρ when it reaches Line 16.*

Then throughout the interval spanning the last execution by ρ of Line 12 and the last execution by ρ of Line 13, $\lfloor A.\mathsf{raw}[\mathsf{left}]/m\rfloor = \lfloor \ell_2/m\rfloor$ and $\lfloor A.\mathsf{raw}[\mathsf{right}]/m\rfloor = \lfloor r_1/m\rfloor$

Proof. For ρ to reach Line 16, it must read values ℓ_1 and ℓ_2 from $A.\mathsf{raw}[\mathsf{left}]$ and r_1 and r_2 from $A.\mathsf{raw}[\mathsf{right}]$ such that $\lfloor \ell_1/m\rfloor = \lfloor \ell_2/m\rfloor$ and $\lfloor r_1/m\rfloor = \lfloor r_2/m\rfloor$.

Because the values returned by a max register are non-decreasing over time, this implies that $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor = \lfloor \ell_2/m \rfloor$ throughout the interval between when ρ last reads $A.\mathsf{raw}[\mathsf{left}]$ in Line 11 and when ρ last reads $A.\mathsf{raw}[\mathsf{left}]$ again in Line 13. Similarly, $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor = \lfloor r_1/m \rfloor$ throughout the interval between when ρ last reads $A.\mathsf{raw}[\mathsf{right}]$ in Line 12 and when ρ last reads $A.\mathsf{raw}[\mathsf{right}]$ in Line 14. Both conditions hold during the intersection of these two intervals, so throughout the interval spanned by reading r_1 and ℓ_2, $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor = \lfloor \ell_2/m \rfloor$ and $\lfloor A.\mathsf{raw}[\mathsf{right}]/m \rfloor = \lfloor r_1/m \rfloor$. □

In particular, at both the step at which ρ last sets r_1 in Line 12 and the step at which ρ last sets ℓ_2 in Line 13 it holds that $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor = \lfloor \ell_2/m \rfloor$ and $\lfloor A.\mathsf{raw}[\mathsf{right}]/m \rfloor = \lfloor r_1/m \rfloor$. This means that the epoch $h = \lfloor \ell_2/m \rfloor + \lfloor r_1/m \rfloor$ calculated by $A.\texttt{read}$ in Line 16 is in fact the epoch of both of these steps.

The following lemma uses this result to show certain consistency conditions follow from the use of the double collect, both between and within epochs:

Lemma 4.4. *Consider some execution of Algorithm 4, and let ρ and ρ' be $A.\texttt{read}$ operations in this execution that both reach Line 16. Let h, ℓ_2 and r_1 be the values of h, ℓ_2, and r_1 at Line 16 in ρ; and let h', ℓ_2', and r_1' be the corresponding values of these variables in ρ'.*

1. *If $h = h'$, then $\lfloor \ell_2/m \rfloor = \lfloor \ell_2'/m \rfloor$ and $\lfloor r_1/m \rfloor = \lfloor r_1'/m \rfloor$.*
2. *If $h < h'$, then $\ell_2 \leq \ell_2'$ and $r_1 \leq r_1'$, and at least one of these inequalities is strict.*

Proof. Let I and I' be the intervals of the executions of ρ and ρ', respectively, spanning the last execution of Lines 12 and 13, as in Lemma 4.3. Let ℓ_2, r_1 and ℓ_2', r_2' be the last values read into ℓ_2, r_1 by ρ and ρ', respectively.

From Lemma 4.3, throughout I, $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor = \lfloor \ell_2/m \rfloor$ and $\lfloor A.\mathsf{raw}[\mathsf{right}]/m \rfloor = \lfloor r_1/m \rfloor$. Similarly, throughout I', $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor = \lfloor \ell_2'/m \rfloor$ and $\lfloor A.\mathsf{raw}[\mathsf{right}]/m \rfloor = \lfloor r_1'/m \rfloor$.

If I overlaps with I', then $\lfloor \ell_2/m \rfloor = \lfloor \ell_2'/m \rfloor$ and $\lfloor r_1/m \rfloor = \lfloor r_1'/m \rfloor$. This implies $h = h'$.

If I precedes I', then ℓ_2 is read before ℓ_2' and r_1 is read before r_1'. Because max register values are non-decreasing over time, $\ell_2 \leq \ell_2'$ and $r_1 \leq r_1'$. Because the floor function is also non-decreasing, $\lfloor \ell_2/m \rfloor \leq \lfloor \ell_2'/m \rfloor$ and $\lfloor r_1/m \rfloor \leq \lfloor r_1'/m \rfloor$.

If either of these inequalities is strict, we have $h = \lfloor \ell_2/m \rfloor + \lfloor r_1/m \rfloor < h' = \lfloor \ell_2'/m \rfloor + \lfloor r_1'/m \rfloor$. If neither is strict, then $h = h'$.

If I' precedes I, a symmetric argument shows that either $h = h'$, $\lfloor \ell_2/m \rfloor = \lfloor \ell'_2/m \rfloor$, and $\lfloor r_1/m \rfloor = \lfloor r'_1/m \rfloor$; or $h' < h$.

Gathering up the cases, we find that whenever $h = h'$, $\lfloor \ell_2/m \rfloor = \lfloor \ell'_2/m \rfloor$ and $\lfloor r_1/m \rfloor = \lfloor r'_1/m \rfloor$. In the only case where $h < h'$, we have $\ell_2 \leq \ell'_2$ and $r_1 \leq r'_1$, and at least one of these inequalities is strict, because otherwise $h = h'$. This proves the claim. □

For each A.`write` operation ω that executes Line 6, we assign an epoch h_ω equal to the epoch following this step. For each direct A.`read` operation ρ that finishes, we assign an epoch h_ρ equal to the value h that it calculates in Line 16. Lemma 4.3 implies that h_ρ will be equal to the epoch at the last steps at which ρ executes Lines 12 and 13. The calculation in Line 20 implies that h_ρ will also be equal to the epoch of the return value of ρ.

We use these assigned epochs to construct a linearization ordering for each execution of Algorithm 4. This will be done first by constructing a partial linearization of only some of the operations, then adjusting this partial linearization to obtain the full linearization needed by the theorem.

4.1.1 The partial linearization Σ_1

Given a concurrent execution Ξ, we will construct a sequence of operations Σ_1 from Ξ that includes all complete direct A.`read` operations (including A.`read` operations called through A.`giveHelp` from A.`write` operations), and all A.`write` operations that complete the A.raw[side].`write` operation in Line 6. This will form the scaffolding for our full linearization, which will be constructed by extending Σ_1 to include indirect A.`read` operations and then removing A.`read` operations embedded in A.`giveHelp` operations.

We sort direct A.`read` operations and A.`write` operations that complete their write to A.raw[side] first by epoch h, then by the order of steps applied to A.low[h].

1. First, group all operations by epoch, and sort these groups by increasing epoch.

2. Within each epoch h, define a **critical step** for most operations that is a step applied to A.low[h]. For an A.`read` operation ρ in epoch h, this is the step at which ρ reads A.low[h] in Line 19. For an A.`write`[side] operation in epoch h, if there exists an A.`read` operation ρ in epoch h such that (a) ρ reads A.raw[side] for the last time after ω writes A.raw[side], and (b) ρ writes A.low[h][side], then ω's critical step is the

first step at which some such A.`read` writes A.low[h][side]. If there is no such ρ, ω is not assigned a critical step.

3. For operations in an epoch that are assigned a critical step, order these operations by the order of their critical steps. Any A.`write` operations not assigned a critical step are ordered after all operations in the epoch that do have critical steps.

4. Finally, for A.`write` operations that are still not ordered with respect to each other, either because they have the same critical step or no critical step, order them by the order in which they write to either side of A.raw.

We claim that this ordering is both consistent with the observable execution order in Ξ and gives a correct sequential execution of a max array.

4.1.2 Consistency of Σ_1 with observed execution order

We start by showing consistency. The first step is to give a characterization of where an operation's critical step can occur with respect to the operation's execution interval in Ξ. Because A.`write` operations depend on direct A.`read` operations to supply their critical steps, it will be helpful to start by showing that the execution interval of a complete A.`write` operation includes the execution interval of a complete A.`read` operation; and that the execution interval of any complete A.`read` operation always includes in turn the execution interval of a complete *direct* A.`read` operation. These facts are shown in the following two lemmas.

Lemma 4.5. *Let Ξ be an execution of Algorithm 4, and let ω be an A.`write`[side] operation that starts and finishes in Ξ. Then there exists an A.`read` operation ρ such that ρ starts after ω's write to A.raw[side] and finishes before ω finishes.*

Proof. This follows from the fact that ω calls A.`giveHelp` after writing to A.raw[side], and A.`giveHelp` calls A.`read`. □

Lemma 4.6. *Let Ξ be an execution of Algorithm 4, and let ρ be an indirect A.`read` operation that starts and finishes in Ξ. Then there exists a direct A.`read` operation ρ' such that ρ' starts after ρ, ρ' finishes before ρ, and ρ returns the same value as ρ'.*

Proof. Since ρ is an indirect $A.\texttt{read}$ operation, it returns a value v obtained by a call to $A.\texttt{takeHelp}$ that returns true. This call starts after ρ calls $A.\texttt{incrementTS}$. So Lemma 2.1 (clause 2) says that v was obtained from an $A.\texttt{read}$ operation ρ_1 that started and finished between these calls to $A.\texttt{incrementTS}$ and $A.\texttt{takeHelp}$, and thus during the execution interval of ρ. If ρ_1 is a direct read, we are done: $\rho' = \rho = 1$. Otherwise, iterate the same argument to obtain a sequence of nested $A.\texttt{read}$ operations $\rho_1, \rho_2, \ldots$ that must eventually end with a direct read. □

The next lemma characterizes where the critical step of an operation in Σ_1 may occur relative to that operation's execution interval:

Lemma 4.7. *Let Ξ be an execution of Algorithm 4 and let α be an operation in the corresponding sequence Σ_1 as defined above, such that α is assigned a critical step σ_α.*

1. *If α is an $A.\texttt{read}$ operation or an $A.\texttt{write}$ operation that finishes, then σ_α occurs between the start and end of α in Ξ.*

2. *If α is an $A.\texttt{write}[\mathsf{side}]$ operation, then whether or not α finishes, σ_α occurs after α writes $A.\mathsf{raw}[\mathsf{side}]$ in Line 6.*

Proof. If α is an $A.\texttt{read}$ operation, it executes σ_α itself.

If α is an $A.\texttt{write}[\mathsf{side}]$ operation in epoch h that has a critical step, then there is some $A.\texttt{read}$ operation ρ that reads $A.\mathsf{raw}[\mathsf{side}]$ for the last time after α writes $A.\mathsf{raw}[\mathsf{side}]$, and subsequently writes $A.\mathsf{low}[\mathsf{side}][h]$. The first such write to $A.\mathsf{low}[\mathsf{side}][h]$ is σ_α, and must occur after α's write to $A.\mathsf{raw}[\mathsf{side}]$ and thus after α starts. This holds whether or not α finishes.

If α does finish, then by Lemmas 4.5 and 4.6, it includes an embedded direct $A.\texttt{read}$ operation ρ. Because ρ starts and finishes after α writes $A.\mathsf{raw}[\mathsf{side}]$, if ρ does not execute σ_α, it is only because some other $A.\texttt{read}$ operation executes σ_α first. In either case, σ_α occurs before α finishes. □

Using Lemma 4.7, we prove that Σ_1 does in fact respect the observable execution order in Ξ:

Lemma 4.8. *Let Ξ be an execution of Algorithm 4, and let Σ_1 be the corresponding sequence of operations as defined above. If α and β are operations in Σ_1, and α finished in Ξ before β starts, then α precedes β in Σ_1.*

Proof. Because α and β are included in Σ_1, they are both assigned epochs h_α, h_β, equal to the epoch of particular steps carried out on $A.\mathsf{raw}$ by each operation. Because α finishes before β starts, α's epoch-determining step

occurs before β's, and so by Lemma 4.2, $h_\alpha \leq h_\beta$. If $h_\alpha < h_\beta$, α precedes β in Σ_1. So the interesting case is when $h_\alpha = h_\beta$.

We start by making some observations about the timing of critical steps.

If α has a critical step σ_α, then σ_α occurs within the execution interval of α. This is because α is either an $A.\texttt{read}$ operation or an $A.\texttt{write}$ operation that finishes in Ξ, so the first case of Lemma 4.7 applies.

Similarly, if β has a critical step σ_β, then σ_β occurs after β starts, by one or the other cases of Lemma 4.7.

We now consider four cases, depending on which of α and β have critical steps:

1. If α and β both have critical steps, then σ_α precedes σ_β and α precedes β in Σ_1.

2. If α has a critical step and β does not, then β is ordered after α.

3. If α does not have a critical step but β does, then α is an $A.\texttt{write}[\mathsf{side}]$ operation assigned an epoch h such that no $A.\texttt{read}$ operation ρ in the same epoch h reads $A.\mathsf{raw}[\mathsf{side}]$ for the last time after α writes it. In particular, the embedded $A.\texttt{read}$ operation ρ whose existence is implied by Lemmas 4.5 and 4.6 does not do so, which can only occur if ρ has an epoch $h_\rho > h$. But in this case, because β starts after the step that assigns ρ its epoch, β must also have an epoch $h_\beta \geq h_\rho > h$. This contradicts the assumption that α and β are operations in the same epoch. We can thus exclude this case.

4. If neither α nor β have critical steps, then they are ordered by the order in which they write $A.\mathsf{raw}$, so α precedes β in Σ_1.

□

4.1.3 Correctness of return values in Σ_1

We now turn to showing that Σ_1 is in fact a sequential execution of a max array. This requires showing that for any $A.\texttt{read}$ operation ρ, the values it returns for the left and right sides of the max array are each equal to the maximum value written to each side by $A.\texttt{write}$ operations that are linearized before ρ.

Lemma 4.9. *Let Ξ be an execution of Algorithm 4, and let Σ_1 be the corresponding sequence of operations defined above. Then any $A.\texttt{read}$ operation ρ in Σ_1 returns a pair v such that for each $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$, $v[\mathsf{side}]$ is*

equal to either the maximum input value to any $A.\texttt{write}[\mathsf{side}]$ *operation that precedes* ρ *in* Σ_1*, or is equal to* 0 *if there is no such operation.*

Proof. We show first that, for each $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$, $v[\mathsf{side}]$ is either equal to 0 or to the input of some $A.\texttt{write}[\mathsf{side}]$ operation that precedes ρ in Σ_1. We then show that for any $A.\texttt{write}[\mathsf{side}](x)$ operation that precedes ρ in Σ_1, $x \leq v[\mathsf{side}]$. Together these show the claim in the lemma.

To keep the argument simple, we concentrate on $v[\mathsf{left}]$. We will note the few places where the proof for $v[\mathsf{right}]$ differs.

For ρ to return $\langle \ell, r \rangle$, ℓ must equal $\lfloor \ell_2/m \rfloor \cdot m + \ell'$, where ℓ_2 is the last value read by ρ in Line 13 and ℓ' is left-side value read by ρ from $A.\mathsf{low}[h]$ in Line 19.

Because ρ itself writes to $A.\mathsf{low}[h].\mathsf{left}$, at least one $A.\texttt{read}$ operation writes to $A.\mathsf{low}[h][\mathsf{left}]$ before ρ reads $A.\mathsf{low}$, and thus ℓ' is equal to $\ell'_2 \bmod m$ for some value ℓ'_2 read from $A.\mathsf{raw}[\mathsf{left}]$ by a read operation ρ' (which might or might not be ρ). Because ρ' writes to the same max array $A.\mathsf{low}[h]$ as ρ, ρ' and ρ have the same epoch h. Applying Lemma 4.4 thus gives $\lfloor \ell_2/m \rfloor = \lfloor \ell'_2/m \rfloor$.

But then $\lfloor \ell_2/m \rfloor \cdot m + \ell' = \lfloor \ell'_2/m \rfloor \cdot m + (\ell'_2 \bmod m) = \ell'_2$.

If ℓ'_2 is not zero, it must be the input to some $A.\texttt{write}[\mathsf{left}](\ell'_2)$ operation ω that writes to $A.\mathsf{raw}[\mathsf{left}]$ before ρ' reads it for the last time.

Since the epochs of ω and ρ' are defined by the epochs of these steps, we have $h_\omega \leq h_{\rho'} = h_\rho$. If $h_\omega < h_\rho$, then ω is ordered before ρ in Σ_1. If $h_\omega = h_\rho$, then the critical step of ω occurs no later than when ρ' writes $A.\mathsf{low}[\mathsf{left}]$. But this occurs before ρ executes its own critical step, its read of $A[\mathsf{low}]$. So in this case ω is ordered before ρ in Σ_1 as well.

We thus have that $v[\mathsf{left}]$, if nonzero, corresponds to the input value of some $A.\texttt{write}[\mathsf{left}]$ operation ω that precedes ρ in Σ_1. A similar argument substituting r_1 for ℓ_2 shows the same holds for $v[\mathsf{right}]$.

Let us now consider some $A.\texttt{write}[\mathsf{left}](v_\omega)$ operation ω that precedes ρ in Σ_1. By the construction of Σ_1, this implies that $h_\omega \leq h_\rho$.

If $h_\omega < h_\rho$, then ω writes $A.\mathsf{raw}[\mathsf{left}]$ before ρ last reads $A.\mathsf{raw}[\mathsf{left}]$ into ℓ_2 in Line 13. It follows that ρ reads a value $\ell_2 \geq v_\omega$. It then writes this value to $A.\mathsf{low}[h][\mathsf{left}]$ before reading $A.\mathsf{low}$, which by the properties of a max array gives that $v[\mathsf{left}] \geq \ell_2 \geq v_\omega$.

If $h_\omega = h_\rho$, then ω precedes ρ in Σ_1 only if there is some $A.\texttt{read}$ operation ρ' that reads $A.\mathsf{raw}[\mathsf{left}]$ after ω writes $A.\mathsf{raw}[\mathsf{right}]$, and then writes $A.\mathsf{low}[\mathsf{left}]$ (which is the critical step for ω) before ρ reads $A.\mathsf{low}$ (which is the critical step for ρ). But then we again have ρ' reading a value $\ell'_2 \geq v_\omega$ that it writes to $A.\mathsf{low}[\mathsf{left}]$, giving $v[\mathsf{left}] \geq \ell'_2 \geq v_\omega$.

As before, essentially the same argument works for $A.\texttt{write}[\mathsf{right}]$ operations.

Now fix some $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$, and let $v_1, \ldots, v_k$ enumerate the inputs to the $A.\texttt{write}[\mathsf{side}]$ operations that precede ρ in Σ_1. Let v be the value returned by ρ. We have shown that $v[\mathsf{side}] = 0$ or $v[\mathsf{side}] = v_i$ for some i. In either case, $v[\mathsf{side}] \leq \max_i v_i$ (where we adopt the convention that a maximum of an empty set is the minimum possible value 0). We also have that for any i, $v[\mathsf{side}] \geq v_i$, which implies that $v[\mathsf{side}] \geq \max_i v_i$. So $v[\mathsf{side}] = \max_i v_i$. □

Together with Lemma 4.8, Lemma 4.9 shows that Σ_1 is a linearization of the operations it includes. It remains only to edit Σ_1 to include all operations visible to the user of the max array, and no others.

4.2 The full linearization Σ

To construct the full linearization Σ, we start with Σ_1, and make two adjustments. First, for each direct read ρ, we gather up all indirect reads $\rho_1, \rho_2, \ldots, \rho_k$ that return a value originally obtained by ρ, and insert them into Σ_1 in arbitrary order immediately following ρ. Second, we remove all $A.\texttt{read}$ operations that are called internally by $A.\texttt{giveHelp}$, as these are not visible to the user. We show that the resulting sequence Σ is a linearization of the original concurrent execution Ξ.

Lemma 4.10. *Given a concurrent execution Ξ of Algorithm 4, the sequence Σ constructed above is a linearization of Ξ.*

Proof. First let us show that the indirect reads $\rho_1, \ldots, \rho_k$ inserted after some direct read ρ are (a) ordered consistently with the observable execution order in Ξ, and (b) return a value consistent with their position in Σ. Let Σ_2 be the intermediate sequence obtained by inserting these operations.

From Lemma 4.6, each ρ_i starts before and finishes after ρ. Suppose α is some operation that precedes ρ_i in Ξ. Then α also precedes ρ, so by Lemma 4.8, α precedes ρ in Σ_1. Thus α precedes ρ_i in Σ_2. Alternatively, suppose β is some operation that follows ρ_i in Ξ; then β follows ρ, and thus β follows ρ in Σ_1. Since Σ_2 inserts ρ_i between ρ and any operation that follows ρ in Σ_1, β follows ρ_i in Σ_2.

Each cluster of $A.\texttt{read}$ operations $\rho, \rho_1, \rho_2, \ldots, \rho_k$ all return the same value; since each component of the value returned by ρ is the max of values previously written by $A.\texttt{write}$ operations in Σ_1 (Lemma 4.9), and Σ_2 does not change the $A.\texttt{write}$ operations from Σ_1, ρ continues to return the correct

value in Σ_2. But this will also be the correct return value for any subsequent A.`read` operations with no intervening A.`write` operations, meaning that $\rho_1, \ldots, \rho_k$ also return the correct value.

We now almost possess the linearization of Ξ. The only remaining complication is that Σ_2 (like Ξ itself), includes internal A.`read` operations that are not visible to the user. By removing these operations we obtain the sequence Σ. This is still a linearization of the externally-visible operations in Ξ, because (a) removing an operation cannot create a pair of operations α, β whose order differs between Ξ and Σ, (b) removing A.`read` operations does not change the return value of any other operations in a sequential execution, and (c) all operations that remain in Σ return the same values as in Σ_2, which we have already shown to be correct. This concludes the proof of linearizability. □

4.3 Complexity

Finally, we bound the step complexity of A.`read` and A.`write` operations. Unlike linearizability, this requires the additional assumption that the sequences of inputs to A.`write` operations have bounded increments.

Lemma 4.11. *Fix some k. Consider an execution of Algorithm 4 where $m \geq nk(n^4 + n + 1)$ is polynomial in n and k, and the sequences of inputs to A.`write`[left] and A.`write`[right] operations have k-bounded increments when ordered by the invocation order of the operations. Then for an appropriate implementation of A.raw, A.`read` and A.`write` operations have step complexity $O(\log^2 n + \log^2 k)$ in expectation and with high probability.*

Proof. The outline of the argument is that we can use k-boundedness of the increments to show (a) that the inputs to each A.raw[side] max register are nk-bounded, giving $O(\log n + \log k)$ step complexity for operations on A.raw; and (b) that the sequence of values of $\lfloor A.\text{raw}[\text{side}]/m \rfloor$ have t-sparse increments for t large enough that an execution of A.`read` that fails three double collects will successfully obtain help with high probability on every iteration of its main loop thereafter.

We start with the claim for A.raw.

Fix some side $\in$ {left, right}. Let $v_1, v_2, \ldots$ be the sequence of inputs to A.`write`[side] operations in some execution Ξ of Algorithm 4, and suppose that this sequence has k-bounded increments. The sequence $v_{i_1}, v_{i_2}, \ldots$ of inputs to A.raw[side].`write` operations in Ξ is an n-buffered reordering of $\{v_i\}$, so by Lemma 3.3 it has nk-bounded increments. It follows

from Theorem 3.11 that each operation on $A.\mathsf{raw}[\mathsf{side}]$ has step complexity $O(\log n + \log k)$ in expectation and with high probability, assuming $A.\mathsf{raw}[\mathsf{side}]$ is implemented using Algorithm 3 with a suitable choice for its parameter m. Theorem 3.11 also tells us that the execution of $A.\mathsf{raw}[\mathsf{side}]$ is linearizable, so for the remainder of the proof, we will treat operations on $A.\mathsf{raw}[\mathsf{side}]$ as atomic.

We now show that an $A.\texttt{read}$ operation does only $O(1)$ loop iterations in expectation and with high probability.

If the double collect in Lines 11 through 14 succeeds, the $A.\texttt{read}$ operation returns. So to maximize the number of iterations of the loop, the adversary must arrange for the double collect to fail; that is, for the process p performing an $A.\texttt{read}$ operation ρ to observe a different epoch in the second collect in Lines 13 and 14 than it observed in the first collect in Lines 11 and 12. This requires that the value of at least one of $\lfloor A.\mathsf{raw}[\mathsf{left}]/m \rfloor$ or $\lfloor A.\mathsf{raw}[\mathsf{right}]/m \rfloor$ must increase between the first and second time p reads it.

Suppose now that the double collect fails three times. Then there are three such increases spread across the two max registers. It follows that for some $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$, $\lfloor A.\mathsf{raw}[\mathsf{side}]/m \rfloor$ increases twice during the initial prefix of the execution of ρ starting with its first execution of Line 11 and ending with its third execution of Line 14. Call this interval $[s, t]$.

For this to occur, these larger values must appear in the sequence of inputs to $A.\mathsf{raw}[\mathsf{side}].\texttt{write}$ operations. We have already established that this sequence of inputs $v_{i_1}, v_{i_2}, \ldots$ is nk-bounded. Applying Lemma 3.2, the scaled sequence $\lfloor v_{i_1}/m \rfloor, \lfloor v_{i_2}/m \rfloor, \ldots$ is $(m/nk - 1)$-sparse. From the stated bound on m, we have $m/nk - 1 \geq n^4 + n$; so at least $n^4 + n$ calls to $A.\mathsf{raw}[\mathsf{side}].\texttt{write}$ start during $[s, t']$. Since there are at most $n - 1$ processes other than p, at most $n - 1$ of these calls to $A.\mathsf{raw}[\mathsf{side}].\texttt{write}$ are executed by $A.\texttt{write}$ operations that are still in progress at t', so at least $n^4 + 1$ of these $A.\mathsf{raw}[\mathsf{side}].\texttt{write}$ operations are carried out by an $A.\texttt{write}$ operation that finishes during $[s, t']$. Every one of these $A.\texttt{write}$ operations calls $A.\texttt{giveHelp}$ after ρ finishes $A.\texttt{incrementTS}$.

We can thus apply apply Lemma 2.1. For any call to $A.\texttt{takeHelp}$ made by ρ following t', there is a probability of at least $1 - 1/n$ that this call returns true (conditioning on the outcome of previous calls) and that the corresponding help value can safely be returned by ρ. So after three failed double collects, on average ρ calls $A.\texttt{takeHelp}$ at most $1+1/(1-1/n) = O(1)$ times before returning, and for any fixed c, the probability that it calls $A.\texttt{takeHelp}$ more than c times without returning is at most n^{-c}. So any $A.\texttt{read}$ performs $O(1)$ iterations of its main loop in expectation and with high probability.

Counting up the cost of each iteration of the main loop, we have two calls to operations on each of $A.\mathsf{raw}[\mathsf{left}]$ and $A.\mathsf{raw}[\mathsf{right}]$, which we have already established take $O(\log n + \log k)$ steps in expectation and with high probability. We also have at most three operations on $A.\mathsf{low}$, for a total cost of $O(\log^2 m) = O((\log n + \log k)^2) = O(\log^2 n + \log^2 k)$ steps always, using the bounded max array of [5].[7] The call to $A.\texttt{takeHelp}$, if executed, has step complexity $O(1)$, as does the call to $A.\texttt{incrementTS}$ outside the loop. Adding everything up gives the claimed bound for $A.\texttt{read}$.

This leaves $A.\texttt{write}[\mathsf{side}]$ operations. Each such operation performs a call to $A.\mathsf{raw}[\mathsf{side}].\texttt{write}$ followed by a call to $A.\texttt{giveHelp}$, which embeds a call to $A.\texttt{read}$ plus $O(1)$ register operations. Both calls take $O(\log^2 n + \log^2 k)$ steps in expectation and with high probability, so the same bound applies to $A.\texttt{write}$. □

5 Unlimited-use snapshots

Using our unbounded-value 2-component max array implementation, we can now obtain an unlimited-use single-writer snapshot object.

We use the construction from [5], which for convenience we restate here. The only difference is that we instantiate this algorithm with our new implementations of unbounded-value max registers and unbounded-value max arrays from the previous sections, instead of the bounded-value implementations used in [5].

The construction is based on a balanced binary tree with n leaves, one for each process. Each intermediate node holds a 2-component max array object for its two children, that counts the number of update operations performed on each. It also stores the (unique) view corresponding to the sum of these numbers. A process updates its location by updating the nodes from its leaf to the root, and a process scans the object by reading the view held by the root. Pseudocode appears in Algorithm 5.

The leaf node in the tree corresponding to each process p_i is denoted by leaf_i. Given a node u, $u.\mathsf{parent}$, $u.\mathsf{left}$, and $u.\mathsf{right}$ represent the parent, left child, and right child of u, respectively. The root of the tree is denoted by root.

Each node u holds an unbounded array $u.\mathsf{view}[0..]$ of views, each of

[7] The $O(\log^2 m)$ bound holds even taking into account the correction to the original algorithm that appears in a subsequent erratum by the authors [6]. Though this correction increases the bound on the cost of max array write operations, these operations still take at most $O(\log^2 m)$ steps, as do max array read operations.

```
1  procedure Update_i(s, v)
2      count_i ← count_i + 1
3      u ← leaf_i
4      ptr ← count_i
5      u.view[ptr] ← v
6      while u ≠ root do
7          if u = u.parent.left then
8              u.parent.ma.write[left](ptr)
9          if u = u.parent.right then
10             u.parent.ma.write[right](ptr)
11         u ← u.parent
12         (lptr, rptr) ← u.ma.read()
13         lview ← u.left.view[lptr]
14         rview ← u.right.view[rptr]
15         ptr ← lptr + rptr
16         u.view[ptr] ← lview · rview // lview and rview are concatenated
17     root.mr.write(ptr)
18 procedure Scan(s)
19     ptr ← root.mr.read()
20     return root.view[ptr]
```

Algorithm 5: Unlimited-use single-writer snapshot object; code for process p_i.

which is either an uninitialized null value $\perp$ or a partial snapshot. Initially, $\mathsf{leaf}_i.\mathsf{view}[0]$ holds the initial value for the snapshot component for process i, and $u.\mathsf{view}[0]$ for each internal node u holds the concatenation of $\mathsf{leaf}_i[0]$ over all leaves leaf_i in the subtree rooted at u. This means that the initial value of $\mathsf{root.view}[0]$ is the initial snapshot value, a vector of all initial values for the individual components.

Each process p_i maintains a local variable count_i that is incremented with each new `Update` operation. The input to the update is written to $\mathsf{leaf}_i.\mathsf{view}[\mathsf{count}_i]$ after incrementing count_i. The process then takes responsibility for propagating this value up through the internal nodes, with the invariant that $u.\mathsf{view}[t]$ will always be a concatenation of some collection of $\mathsf{leaf}_i.\mathsf{view}[t_i]$ values where t is the sum of the t_i.

Consistency and uniqueness of these intermediate views is enforced using a max array $u.\mathsf{ma}$ at each internal node u, where $u.\mathsf{ma}[\mathsf{left}]$ holds the index

of the most recent view of u.left and u.ma[right] holds the index of the most recent view of u.right; these views are combined by concatenation to produce a new view with index equal to the sum of the two child views. Because the root has no parent, it uses an extra max register root.mr to store the index of its view; this is read by a `Scan` operation to obtain the most recent view that has been fully propagated to the root.

In the original implementation of [5], each max array and max register is limited to holding values up to some fixed bound b. Because the proof of correctness of the algorithm does not use b, we can replace these bounded max arrays and max registers with our new, unbounded variants, without affecting the original proof of correctness. So we need only show that the step complexity of each operation is likely to be low. This requires showing that appropriate bounded-increments assumptions hold.

We start with some simple invariants, which will be used to show that the sequences of values stored in each u.mr component and in root.mr have bounded increments. In the following, we define for each node u in the tree a location r_u, which is u.mr if u is the root, u.parent.ma[left] if $u = u$.parent.left, and u.parent.ma[right] if $u = u$.parent.right. Note that when u is not the root, r_u is only one of two components of a max array object. We define the value of r_u after a given partial execution in the obvious way, as the largest value written to r_u during that execution. For the purpose of proving these invariants, we assume that operations on the u.ma and root.mr objects are atomic; this is equivalent to assuming that we have fixed some linearization of the operations on these objects and are observing properties of that linearization.

Lemma 5.1. *After any partial execution Ξ of Algorithm 5, the following invariants hold:*

1. *If u is a leaf, then r_u is equal to the number of write operations to r_u in Ξ.*
2. *If u is any node, then r_u is greater than or equal to the number of write operations to r_u in Ξ.*
3. *If u is an internal node, then $r_u \leq r_{u.\mathsf{left}} + r_{u.\mathsf{right}}$.*

Proof. 1. Let $u = \mathsf{leaf}_i$. Then r_u is written only by p_i in Line 8 or 10. In either case, p_i writes the value of count_i, which is equal to the number of times p_i writes r_u up to and including this write.

2. Let u be any node. We prove the claim by induction on the height of u. If u is a leaf, the claim follows from the previous invariant. If u is

an internal node at height $h > 0$, then by the induction hypothesis, the invariant holds for its children $u.\mathsf{left}$ and $u.\mathsf{right}$.

Because u is an internal node, it can be written only by some process executing Line 8 or Line 10. Since u is not a leaf, the value written is the value ptr computed in the previous iteration by summing (in Line 15) the values of $r_{u.\mathsf{left}}$ and $r_{u.\mathsf{right}}$ previously obtained by taking a snapshot of $u.\mathsf{ma}$ in Line 12.

Consider the last such execution of Line 12 by a process p_j that subsequently writes r_u in Ξ; let Ξ' be the prefix of Ξ ending with this step.

Because every process p_k that writes r_u in Ξ must previously write either $r_{u.\mathsf{left}}$ or $r_{u.\mathsf{right}}$, and because any such write precedes p_k's snapshot of $u.\mathsf{ma}$ in Line 12, every process p_k that writes r_u in Ξ writes one of $r_{u.\mathsf{left}}$ or $r_{u.\mathsf{right}}$ in Ξ'. So the number of writes to these locations in Ξ' is at least the number of writes to r_u in Ξ. By the induction hypothesis, the values read by p_j for $r_{u.\mathsf{left}}$ and $r_{u.\mathsf{right}}$ will be at least these quantities, and so the sum of these values will be at least the number of writes to r_u in Ξ.

3. Conversely, any value written to r_u for an internal node u was computed as the sum $r'_{u.\mathsf{left}} + r'_{u.\mathsf{right}}$, where $r'_{u.\mathsf{left}}$ and $r'_{u.\mathsf{right}}$ were the values of $r_{u.\mathsf{left}}$ and $r_{u.\mathsf{right}}$ after some prefix Ξ' of Ξ. Because the values of these locations are non-decreasing over time, at the end of Ξ, $r_u = r'_{u.\mathsf{left}} + r'_{u.\mathsf{right}} \leq r_{u.\mathsf{left}} + r_{u.\mathsf{right}}$.

□

These invariants give:

Corollary 5.2. *In any execution of Algorithm 5, the sequence of values appearing in any location r_u has n-bounded increments.*

Proof. Applying Claim 3 of Lemma 5.1 inductively, any partial execution of Algorithm 5, $r_u \leq \sum_\ell r_\ell$, where ℓ ranges over all leaves in the subtree rooted at u. We will apply this to a sequence of partial executions of increasing length.

Let r_u^t be the value of r_u after t steps. Let m_u^t be the number of times processes in the subtree rooted at u call `Update` in the first t steps. Because r_ℓ counts the number of times the process assigned to ℓ writes r_ℓ, $r_\ell^t \leq m_\ell^t$, and thus $r_u^t \leq \sum_\ell r_\ell^t \leq \sum_\ell m_\ell^t = m_u^t$, where as before ℓ in each sum ranges over all leaves below u.

In the other direction, Claim 2 of Lemma 5.1 says that r_u^t is at least equal to the number of times processes in the subtree write to r_u in the first t steps. This will be at least $m_u^t - n_u$, where $n_u \leq n$ is the number of processes assigned to the subtree, because each process can have at most one `Update` operation in progress that has not yet written r_u.

We now consider the effect of a step on r_u. The only steps that change r_u are writes to r_u; a process that takes such a step does not increase m_u^t, because it has already started its `Update` in some earlier step. So the maximum increase in r_u is from $m_u^t - n$ to $m_u^{t+1} = m_u^t$, an increase of n. Because r_u is either a max register or a max array component, the previous value of r_u is the maximum of the sequence up to this point, so the sequence of r_u values has n-bounded increments. □

We finish by showing how to translate bounded increments on the sequences of values stored in particular locations to bounded increments on the sequences of values supplied as inputs to `write` operations on their parents. This gives bounded increments on the inputs to `write` operations to all internal nodes in the tree, leaving leaves as a special case.

For each internal node u, define s_u as the value of $r_{u.\mathsf{left}} + r_{u.\mathsf{right}}$, and define s_u^k as the value of $r_{u.\mathsf{left}} + r_{u.\mathsf{right}}$ after k total writes to $r_{u.\mathsf{left}}$ and $r_{u.\mathsf{right}}$. Observe that s_u can only increase as the result of one of these writes, and since a write affects only one of $r_{u.\mathsf{left}}$ or $r_{u.\mathsf{right}}$, from Corollary 5.2, the maximum increase in s_u is bounded by n. It follows that $\left\{s_u^k\right\}$ has n-bounded increments.

Define v_u^k as the value of s_u obtained by the k-th snapshot of $r_{u.\mathsf{left}}$ and $r_{u.\mathsf{right}}$ by some process executing Line 12, where these snapshots are ordered by their linearization points. Because each process takes a snapshot of $r_{u.\mathsf{left}}$ and $r_{u.\mathsf{right}}$ after updating either, a total of at most n updates can occur on these two locations between snapshots. We have already established that each such update increases s_u by at most n; so n such updates increase s_u by at most n^2. This implies that for all k, $v_u^{k+1} \leq v_u^k + n^2$: the sequence $\left\{v_u^k\right\}$ has n^2-bounded increments.

Now apply Lemma 3.3. The sequence of input values to write operations to r_u is an n-buffered reordering of the sequence $\left\{v_u^k\right\}$, because each process that takes a snapshot in Line 12 writes v_u^k as its next step in one of Lines 8, 10, or 17. The adversary can delay these writes, but can only delay up to n of them at a time; hence Lemma 3.3 applies. We have just shown, at least for internal nodes:

Lemma 5.3. *For any node u, the sequence of inputs to write operations on r_u has n^3-bounded increments.*

Proof. If u is an internal node, use the proof above.

If u is a leaf, this follows from each write having an input 1 greater than the previous write. □

Lemma 5.3 is the last piece we need for the step complexity of Algorithm 5. We state the full result for later use.

Theorem 5.4. *For an appropriate parameterization of the internal max registers and max arrays, Algorithm 5 is a linearizable implementation of a snapshot, where each A.`write` operation takes $O(\log^3 n)$ steps and each A.`read` operation takes $O(\log^2 n)$ steps, in expectation and with high probability.*

Proof. Linearizability is immediate from the proof in [5].

For the probabilistic step complexity bound, observe that Algorithm 5 carries out $O(\log n)$ iterations of its main loop, which performs $O(1)$ max array operations per iteration, with n^3-bounded increments on the sequences of inputs to write operations. Thus each max array operation has cost $O(\log^2 n + \log^2 n^3) = O(\log^2 n)$ in expectation and with high probability, so the total cost of the loop is $O(\log^3 n)$ in expectation and with high probability. This dominates the operations outside the loop, and gives the claimed bound. □

6 Extension to message passing

Our algorithm can be adapted to give an implementation of a snapshot object in an asynchronous message-passing system with fewer than $n/2$ crash failures. A direct adaptation using the ABD register simulation [11] would require $O(\log^3 n)$ time and $O(n \log^3 n)$ messages on average for each operation, where one time unit is the maximum message delay in the execution. This is because ABD implements a read/write register operation in $O(1)$ time and $O(n)$ messages, and our snapshot implementation uses $O(\log^3 n)$ accesses to read/write registers.

By taking advantage of the message-passing model, we can improve this to $O(\log^2 n)$ time and $O(n \log^2 n)$ messages, while eliminating the need for randomization. The key idea is that the inherent parallelism of a message-passing system and the ability to consolidate multiple concurrent messages

into one allows operations like collects or max-register reads to be implemented at no greater cost than the $O(1)$ time and $O(n)$ messages needed to simulate an ordinary atomic register.

We begin by describing our implementation of an unbounded-value max register using message passing; this gives the log-factor reduction in complexity. We then show how the randomized helping mechanism can be simplified by eliminating random sampling over the active array in favor of performing collects on the TS and helpTS arrays directly. This makes the guarantees for the bounded-increments max array implementation deterministic. Making these substitutions in Algorithm 5 gives the full result.

6.1 Message-passing max registers

An unbounded-value max register can be implemented directly in an asynchronous message-passing system with $f < n/2$ crash failures using a straightforward adaptation of the classic **ABD** atomic register simulation of Attiya, Bar-Noy, and Dolev [11].

Pseudocode is given in Algorithm 6. Each process stores a local value maxValue for the max register. The `write` and `read` operations are both implemented using a core `Update` subroutine. This obtains a maximum value from a majority of processes (including itself), possibly replaces it with the argument to `write`, and transmits the new value to a majority of processes (including itself). As in the ABD register, this second round is needed to ensure linearizability when the maximum value obtained in the first round is not already stored in a majority of the processes. We denote by $\perp$ a value that is not larger than any integer value v.

Theorem 6.1. *Algorithm 6 is a linearizable deterministic implementation of a max register for a message-passing system with fewer than $n/2$ crash failures, in which both* `read` *and* `write` *operations take $O(1)$ time and $O(n)$ messages.*

Proof. Both the complexity and the linearizability proofs resemble the corresponding proofs for ABD.

Since `read` and `write` are both wrappers for `Update`, their complexity bounds follow immediately from the fact that `Update` uses two round-trips ($O(1)$ time) involving a linear number of messages each ($O(n)$ messages).

To show linearizability, given an execution of Algorithm 6, we construct an explicit linearization ordering as follows. We first order all operations by the value v' obtained in Line 8, then by observable execution order (order of non-overlapping operations) within each group of operations with the same

Persistent Local Data: maxValue, initially $\perp$; t, initially 0

1 **upon receiving** `Update`(t, v) *from* j **do**
2 maxValue $\leftarrow \max(\text{maxValue}, v)$
3 Send `respond`$(t, \text{maxValue})$ to j
4 **procedure** `Update`(v)
5 $t \leftarrow t + 1$
6 Send `Update`$(t, \perp)$ to all processes.
7 Wait to receive `respond`(t, v_i) from a set S containing a majority of processes p_i.
8 Let $v' = \max(v, \max_{i \in S}(v_i))$.
9 $t \leftarrow t + 1$
10 Send `Update`(t, v') to all processes.
11 Wait to receive `respond`$(t, -)$ from a majority of processes.
12 **return** v'.
13 **procedure** `write`(v)
14 `Update`(v)
15 **procedure** `read`()
16 **return** `Update`$(\perp)$

Algorithm 6: Max register in message passing using ABD.

value of v'. Finally, we put `write` operations before `read` operations and break any remaining ties arbitrarily.

To show that this is consistent with the observed execution order, suppose that some operation A finishes before B starts. Let v'_A and v'_B be the values of v' computed by A and B, respectively. Then A broadcasts v'_A to a majority of processes before it finishes, and at least one of these processes is also in the majority that later respond to B's `Update`$(t, \perp)$ message. So the calculation of v'_B includes either v'_A or a larger value, giving $v'_B \geq v'_A$. If $v'_B = v'_A$, then B is ordered after A by the execution ordering; if $v'_B > v'_A$, then B is ordered after A by the v' ordering.

Next we argue that the linearized execution is a sequential execution of a max register. Since the only operations that return values are `read` operations, it is enough to show that these values are equal to the largest input to any previous `write` operation in the linearized execution.

Let us begin with a simple invariant: In any prefix of an execution of the algorithm, any value other than $\perp$ that appears as maxValue_p in some process p, or appears as the value in an `Update` or `respond` message, appears as the input to some `write` operation that starts in this prefix. The

proof is a straightforward induction: examination of the code shows that new values can appear only when a `write` broadcasts its second `Update` message, and such values will either be the input to the `write` or a value that previously appeared in a `respond` message to the process carrying out the `write` operation. It follows that any value returned by a `read` corresponds to some value written in a `write`.

The computation in Line 8 ensures that any `write`(v) operation W computes $v'_W \geq v$; in the other direction, any `read` operation R returns v'_R. So for any given `read` operation R, only `write` operations with input less than or equal to v'_R can be linearized before R. From the invariant, there exists at least one `write` operation W with input v_W that is equal to v'_R (for the value v'_R to be returned in Line 8). Out of all such `write` operations, there is at least one that linearizes before R, because the linearization point of operations depend on the value obtained in Line 8. If for all of the above `write` operations that linearize before R it holds that the value v'_W obtained in Line 8 is larger than the input v_W, then none of these operations use v_W in Line 10, preventing this value from being returned as v'_R by the `read` operation R in Line 8 (since any maxValue can only store a value that was the input to some previous `Update`). It follows that v'_R is in fact the largest input to any `write` operation linearized before R, and the sequential specification is satisfied. □

6.2 Bounded message-passing 2-component max arrays

By using the message-passing max register of the previous section in the bounded max array of [5], we get an immediate log-factor improvement in the cost of a max array as well.

Corollary 6.2. *For any k and ℓ, there exists a linearizable implementation of a $(k \times \ell)$-bounded max array in a message-passing system with fewer than $n/2$ crash failures, such that both* `read` *and* `write` *operations require $O(\log k)$ time and $O(n \log k)$ messages.*

Proof. An execution of a `read` or `write` operation in the $(k \times \ell)$-bounded max array construction of [5] requires $O(\log k)$ atomic register operations and $O(\log k)$ operations on ℓ-bounded max registers. By implementing the atomic registers using the ABD simulation [11] and the max registers using Algorithm 6, we obtain the claimed performance. □

6.3 Deterministic helping

Helping can be simplified in a message-passing system, because we can implement a collect operation that reads the values of arbitrarily many separate registers by combining messages used to implement each read independently.

Formally, a **collect** object is a weak version of a snapshot that does not guarantee that values read from distinct registers appear to be read atomically. It is equivalent to an array of n single-writer registers, with a write operation for each register and a collect operation that returns a value for each register that was present in that register at some time between the start and finish of the collect.

The straightforward implementation of a collect is to have the reader read each register directly. This gives a cost of $O(1)$ steps per write and $O(n)$ steps per collect. In a message-passing system, we can use the standard ABD simulation for each register and combine the messages for the n read operations used in the collect. This gives a cost of $O(1)$ time and $O(n)$ messages for both write and collect, making a collect no more expensive than reading a single simulated register. More generally, if we are only concerned with time and message complexity, we can similarly perform a collect on an unlimited number of registers with the same time and message complexity bounds.

With cheap collects, we can avoid the need for using random sampling to reduce the cost of obtaining help. This allows us to eliminate the $A.\mathsf{active}$ array in Algorithm 1 completely, have each reader read all the available help at once, and have each writer help all readers at once on every write. The resulting implementation is given in Algorithm 7.

Because Algorithm 7 is deterministic, its properties are more straightforward to describe than those of the Algorithm 1. The following lemma is analogous to Lemma 2.1. Note that unlike Algorithm 1, Algorithm 7 does not return useful information in the second return value of $A.\texttt{takeHelp}$ if the first return value is **false**. Fortunately this feature is only used in our shared-memory construction when implementing the max register. In our message-passing construction, we use a different implementation that does not require help.

Lemma 6.3. *Consider an execution Ξ of Algorithm 7. Let t be the point at which some process p_i starts an $A.\texttt{takeHelp}$ operation τ. Suppose there is a point s in Ξ such that at least $n+1$ calls to $A.\texttt{giveHelp}$ start in the interval $[s,t]$. Then:*

1. *If τ returns* **true** *following at least one call by p_i to $A.\texttt{incrementTS}$,*

```
1  Shared data:
2  A.TS[0...n−1], an array of ABD registers holding timestamps,
     initially all 0
3  A.helpTS[0...n−1][0...n−1], an array of ABD registers holding
     timestamps, initially all 0
4  A.helpVal[0...n−1], an array of ABD registers holding help values,
     initially all ⊥
5  procedure A.incrementTS()
6      A.TS[i] ← A.TS[i] + 1;
7  procedure A.giveHelp()
       // collect all timestamps
8      t[0...n−1] ← A.TS[0...n−1];
9      v ← A.read();
10     A.helpVal[i] ← v;
11     A.helpTS[i][0...n−1] ← t[0...n−1];
       // Return value read, in case it is useful
12     return v;
13 procedure A.takeHelp()
       // collect timestamps for p_i from all helpers
14     t[0...n−1] ← A.helpTS[0...n−1][i];
       // find most recent help
15     if there exists j such that t[j] ≥ A.TS[i] then
16         return (true, A.helpVal[j]);
17     else
18         return (false, ⊥);
```

Algorithm 7: Deterministic helping applied to object A. Code for process p_i.

then the value returned by τ was previously returned by a call to A.`read` *that started no earlier than the start of p_i's last call to A.*`incrementTS` *and finished no later than τ.*

2. *If p_i's last call to A.*`incrementTS` *before t finished before s, then if τ returns, τ returns* **true**.

Proof. 1. Immediate from the code: For τ to return **true**, it must see a timestamp t in $A.\mathsf{helpTS}[j]$ for some j that is equal to the current value of $A.\mathsf{TS}[i]$. Because the only place $A.\mathsf{helpTS}[j]$ is updated is in Line 11 of $A.$`giveHelp`, this can occur only if p_j makes a call to

A.giveHelp that reads t from $A.\mathsf{TS}[i]$ (implying that this call starts after p_i's last call to A.incrementTS) and subsequently calls A.read to obtain a value to write to $A.\mathsf{helpVal}[j]$ in Line 10 before writing the new timestamp in Line 11. So the value returned by τ from $A.\mathsf{helpVal}[j]$ will have been read at least as recently as this value.

2. If $n+1$ calls to A.giveHelp start in the interval $[s,t]$, then there is at least one process that starts two calls to A.giveHelp. Let p_j be such a process, and consider the first call γ by p_j to A.giveHelp that starts in $[s,t]$. Because p_j starts at least one more call before t, γ starts and finishes in $[s,t]$. This means that it starts after p_i's last call to A.incrementTS before t, and finishes before τ starts. During this time, γ copies $A.\mathsf{TS}[i]$ to $A.\mathsf{helpTS}[j][i]$. Any subsequent call to A.giveHelp by p_j will only copy this value again, so when τ reads $A.\mathsf{helpTS}[j][i]$, it obtains a value that is at least equal to $A.\mathsf{TS}[i]$, causing it to return **true**.

□

Though the ability to read n registers in parallel allows for a minimum of $n+1$ A.giveHelp operations in Lemma 6.3 instead of the n^4+1 operations in Lemma 2.1, this only improves the applicability of the result. In particular, under the conditions of Lemma 2.1, Lemma 6.3 implies that the last two claims of Lemma 2.1 apply to Algorithm 7 as well, with no probability of error.

6.4 Unbounded message-passing 2-component max arrays

We can now reimplement Algorithm 4 by replacing every instance of a max register operation with the $O(1)$-time $O(n)$-message implementation from Algorithm 6, every instance of an $(m \times m)$-bounded max array by the $O(\log m)$-time $O(n \log m)$-message implementation from Section 6.2, and replacing the help mechanism with Algorithm 7. Because these algorithms are deterministic, this gives a *deterministic* implementation of an unbounded max array for a message-passing system.

The correctness and efficiency of this implementation is described in the following theorem.

Theorem 6.4. *Algorithm 4, adapted to a message-passing system as described above, is a linearizable implementation of a max array, such that, for an appropriate choice of m, in any execution where the sequences of input*

values to calls to A.`write`[left] *and* A.`write`[right] *both have* k*-bounded increments, each* A.`write` *and* A.`read` *operation completes in* $O(\log n + \log k)$ *time and* $O(n(\log n + \log k))$ *messages, provided fewer than* $n/2$ *processes crash.*

Proof. The proof of linearizability for Algorithm 4 given in Sections 4.1 through 4.2 is not affected by replacing one linearizable implementation of a max register by another, or by using return values obtained indirectly from A.`read` operations called in A.`giveHelp`. Formally, this follows from Theorem 6.1 (which shows that the max register implementation given in Algorithm 6 is linearizable), and the first claim of Lemma 6.3 (which replaces the second claim of Lemma 2.1).

For the complexity claims, we adapt the argument in the proof of Lemma 4.11. Because the unbounded max register construction in Algorithm 6 does not require sparse increments, we can skip the initial part of the argument. Instead, we proceed directly to the argument that any A.`read` argument that fails three double collects will successfully obtain help. As in Lemma 4.11, we assume that the parameter m is at least $nk(n^4+1)$ and is polynomial in n and k.

Repeating the argument in the proof of Lemma 4.11, we have that if some A.`read` operation ρ fails to finish after three double collects, then at least n^4+1 A.`write` operations call A.`giveHelp` after ρ calls A.`incrementTS` and before ρ finishes its third double collect. Since $n^4+1 > n+1$, Lemma 6.3 says that any subsequent call by ρ to A.`takeHelp` will return **true**. So ρ executes at most $O(1)$ iterations.

Each such iteration requires $O(1)$ max register operations ($O(1)$ time and $O(n)$ messages each, from Theorem 6.1), plus $O(1)$ operations on an $(m \times m)$-bounded max array ($O(\log m)$ time and $O(n \log m)$ messages each, from Corollary 6.2). So the total cost is $O(\log m) = O(\log n + \log k)$ time and $O(n \log m) = O(n(\log n + \log k))$ messages. □

6.5 The full snapshot algorithm

To complete the algorithm, implement Algorithm 5 using the max arrays from Section 6.4. We then have:

Theorem 6.5. *Using message-passing max arrays, Algorithm 5 gives a linearizable implementation of an unlimited-use snapshot object, with a time complexity of* $O(\log^2 n)$ *and message complexity of* $O(n \log^2 n)$ *for each operation, provided fewer than* $n/2$ *processes crash.*

7 Discussion

This paper gives the first sub-linear unlimited-use snapshot implementation from atomic read/write registers. It is a randomized algorithm, with a step complexity of $O(\log^3 n)$ with high probability for each operation, where n is the number of processes. The main component of the construction is a new randomized implementation of an unbounded-value max register with a complexity of $O(\log n)$ steps per operation with high probability when the inputs to write operations have polynomially-bounded increments. The novelty of the construction is a randomized helping technique, which allows slow processes to obtain fresh information from other processes.

The use of randomization avoids in most cases the linear worst-case lower bound based on covering arguments of Jayanti *et al.* [18], because the adversary cannot predict what locations a process will read in the helping mechanism's pointer array and thus cannot be sure of covering those locations with old values. Conversely, the Jayanti *et al.* lower bound shows that some use of randomization is necessary.

Curiously, randomization does not appear to be necessary in a message-passing implementation. Here we exploit the fact that we can read multiple ABD registers in parallel at no additional cost to allow the algorithm to read all available help directly. Together with an $O(1)$-time deterministic unbounded-value max register based on the ABD construction, this gives a cost per operation of $O(\log^2 n)$ time and $O(n \log^2 n)$ messages using message-passing. It would be interesting to see if a more sophisticated use of the powers of a message-passing system could reduce this cost further.

Acknowledgements

The authors thank Faith Ellen for useful discussions. The authors are also in debt to the anonymous reviewers of both the conference and journal versions of this paper for careful comments and suggestions that greatly improved the presentation and correctness of this work.

References

[1] Anna Adamaszek, Marc P. Renault, Adi Rosén, and Rob van Stee. Reordering buffer management with advice. *Journal of Scheduling*, 20(5):423–442, Oct 2017.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[3] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[4] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, March 2012.

[5] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use snapshots with polylogarithmic step complexity. *Journal of the ACM*, 62(1):3, February 2015.

[6] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Erratum: Limited-use snapshots with polylogarithmic step complexity. To appear, JACM. Available at `http://www.cs.yale.edu/homes/aspnes/papers/limited-use-snapshots-abstract.html`., April 2017.

[7] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Danny Hendler. Lower bounds for restricted-use objects. In *Twenty-Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 172–181, June 2012.

[8] James Aspnes and Keren Censor-Hillel. Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In Yehuda Afek, editor, *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14–18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin Heidelberg, 2013.

[9] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

[10] James Aspnes and Eric Ruppert. Depth of a random binary search tree with concurrent insertions. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27–29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 371–384. Springer, 2016.

[11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[12] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[13] Ho-Leung Chan, Nicole Megow, René Sitters, and Rob van Stee. A note on sorting buffers offline. *Theoretical Computer Science*, 423:11 – 18, 2012.

[14] George Giakkoupis and Philipp Woelfel. An Improved Bound for Random Binary Search Trees with Concurrent Insertions. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*, volume 96 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[17] Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 130–140, London, UK, 1994. Springer-Verlag.

[18] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[19] Harald Räcke, Christian Sohler, and Matthias Westermann. Online scheduling for sorting buffers. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 820–832. Springer, 2002.