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

James Aspnes[†] Keren Censor-Hillel[‡]

February 11, 2017

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 [9, 10] and on the linear lower bound for deterministic constructions [11], and does not require limiting the number of updates as in previous sublinear constructions [4]. One of the main ingredients in the construction is a novel *randomized helping* technique that allows out-of-date processes to obtain up-to-date information.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 [1, 2, 7] required $\Theta(n^2)$ steps to carry out a snapshot with n processes; subsequent work [9, 10] 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.* [11].

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** [3] (which support `ReadMax` operations that return the largest value written by `WriteMax` operations) and counters.

Limitations of the lower bound technique of [11] 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 [3], and later of snapshot objects [4]. 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 [3], 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 [5], 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 [3], where w is an upper bound for the complexity of `WriteMax` operations. That is, for every correct implementation, with high probability, there is either a `WriteMax` operation that takes w steps or a `ReadMax` operation that takes $\Omega(\log n / \log(w \log n))$ steps. This bound holds even for an oblivious adversary. This means that, curiously, for randomized implementations these lower bounds are logarithmic in n. This appeared to be a weakness of the

[1] The full definition is a bit more technical; see [11] 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.

particular proof technique used to obtain the randomized lower bounds.

We show that significantly larger lower bounds are impossible: Using a new randomized helping procedure along with a simple max register implementation, it is possible to accelerate the max register implementation of [3] 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 [4] 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 stepping stone toward implementing a limited-use snapshot, using an approach that we will summarize briefly.

Applying our techniques to the 2-component max array of [4][5] does not give the claimed complexity of $O(\log^2 n)$ steps. Instead, provide a completely different construction of a 2-component max array, which uses a randomized helping technique that is similar in spirit to that of our max register construction.

We further adapt our construction to message-passing, showing that we can obtain an improvement in performance, compared to a naïve conversion using the construction of Attiya, Bar-Noy, and Dolev [8] (commonly known as *ABD*), and remove the need for randomization by exploiting the additional powers of a message-passing system. 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 [6]. New results include a different 2-component max array construction, the 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-

[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 some constant $c \geq 1$.

[5]as done in the conference version of this paper

passing model tolerating $t < n/2$ crash failures. In both models, we represent concurrency by interleaving.

In the shared-memory model, each non-terminated non-faulty process has a pending operation on some register in each state. In each state, an adversary chooses which of these pending operations is applied next.

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.

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 also to the stronger adaptive adversary.

Complexity of an implementation in the shared-memory model is measured using **worst-case expected individual step complexity**, defined as follows. Since an adaptive adversary bases its decisions on the outcomes of the local coin flips, we can describe all adaptive adversaries as a decision tree, where each intermediate node corresponds to a possible step that is scheduled by an adversary after observing the outcomes of all current local coin flips. With each node in the tree, we associate a probability, which is obtained inductively by the probability of the preceding node and the decision of the adversary. For each operation instance, there are many intermediate nodes that correspond to its completion step. For each such node, we take the number of register operations that the corresponding operation performed, starting from the root. Together with the probability associated with this node, this gives an expected number of steps for it. The worst-case expected step complexity of an operation instance, is the maximum expected step complexity over all nodes in the decision tree that correspond to the completion of this operation. Finally, the worst-case expected individual step complexity of the implementation is then the maximum of the above over all operations.

In the message-passing model, we consider both message complexity (expected number of messages sent and received by all processes while carrying out some operation) and time complexity (the expected time from the start to finish of an operation assuming that all messages are delivered after at most one time unit).

A **max register** 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.

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

A **single-writer snapshot** 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 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 [4] is a 2-component max array. To directly build an unlimited-use snapshot object we need an unbounded-value version of a max register, and an unbounded-value version of a 2-component max array.

The max register construction of [3] is based on 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 [3] 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 [4] 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 [4] that this implementation gives a linearizable bounded-value 2-component max array.

In the snapshot algorithm of [4], 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.

1.3 Our Contributions

Our first contribution is a construction of an unbounded-value max register in which the step complexity of each operation is $O(\log n)$ with high probability, which overcomes the obstacle of the construction of [3] by combining a max-register with a novel technique of **randomized helping**. The challenge is to handle the case of an operation that is traveling down the tree on the rightmost path for too long. We refer to the rightmost path of the tree as the **spine**. In essence, the randomized helping technique allows such an operation to jump farther ahead to a point on the spine that is the correct one, that is, the first point on the spine for which the switch is unset. This is done by adopting a location in the spine used by another operation, with the challenge of making sure that this value is **fresh**—recent enough that the first operation can use it without violating linearizability. The only condition we place on the usage of the max register in order for this to work is that operations write values that are not increasing too fast. We need this condition in order to argue that once the operation found the correct node on the spine, it can safely continue to the left subtree without the worry that a new `WriteMax` operation is now writing a much larger value that is placed farther down the spine. While at first glance this might seem as a strong restriction, this is actually a very reasonable condition in applications that use max registers, and in particular it is satisfied by our implementation of an unlimited-use single-writer snapshot object.

Our second contribution is a 2-component max array that is unbounded, and whose cost per operation does not depend on the number of operations. The natural thing to try is embedding the unbounded-value max register construction in the 2-component max array construction of [4]. However, this does not work directly, since the main insight there is that values of the second component need to be propagated down while traveling the tree of the first component in order to guarantee that returned pairs are comparable. Since parts of the main tree of the first component may be skipped by a MaxScan operation due to the randomized helping mechanism, gaps may be created among the values of the corresponding copies of the second component. In the conference version of this paper [6], the solution was to take advantage of the bounded increments assumption and re-read the second component associated with the root of the primary tree instead of reading copies of the second component along the spine, which may have been skipped. This, however, causes the problem that the values that are written to a copy of the second component associated with a spine node may not comply anymore with the bounded increments assumption. In other words, this cannot be done this way within our randomized helping technique because operations may jump down the spine without accessing each node along the way.

Instead, our 2-component max array is built from two separate max registers, and MaxUpdate operations simply write to the required component. In addition, MaxUpdate operations help MaxScan operations using a randomized helping mechanism that is similar in spirit to that of the max register. In a nutshell, each value v is split into its high and low parts, which are $\lfloor v/m \rfloor$ and $v \mod m$, respectively, for a value m that is chosen later. A MaxScan operation is performed using a standard double-collect mechanism as in [1] but only over the high parts of the pair of values. The low parts enjoy the property of a bounded number of values, and can therefore be handled using a standard max register. A careful combination of both mechanisms yields a step complexity of $O(\log^2 n)$ steps per operation.

Plugging these two improvements into the snapshot implementation of [4] 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.

Finally, we adapt our implementation to a message-passing system, 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 ABD [8] 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 in $O(1)$ 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 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.

2.1 Bounded-value max registers

We begin by reviewing the bounded-value max register implementation of Aspnes *et al.* [3]. The idea is to implement the register as a fixed binary tree of one-bit atomic registers, referred to as switch bits. Initially these bits are all 0, which is interpreted as pointing to the left child of the register, while a 1 points to the right child. Each value of the max register corresponds to a leaf of the tree (which does not get a register). A `ReadMax` operation follows the path determined by the values of the switch bits until it reaches a leaf; the number of leaves to the left of this leaf (its **rank**) gives the return value. (See Algorithm 1.)

Because a `ReadMax` reads one register for each node on the path, the cost of a `ReadMax` is just the depth of the leaf it reaches. For an m-valued max register implemented as a balanced binary tree, this will be at most $\lceil \log m \rceil$. Since our implementation in this paper uses an unbalanced tree, it is important to note that the correctness of Algorithm 1 does not depend on the fact that the tree is balanced, and it is correct for any binary tree. Unbalanced trees can be used to obtain adaptive costs; it is shown in [3] that using an appropriate unbalanced tree truncated and combined with a linear-time snapshot implementation gives a cost of $O(\min(\log v, n))$ for `WriteMax`(v) or for a `ReadMax` that returns the value v.

A `WriteMax`(v) operation must set the switch bits so that subsequent `ReadMax` operations will be directed to the leaf with rank v, unless some larger v' has already been written. Intuitively, for this to happen, the `WriteMax`(v) operation needs to go over the path from the root to the leaf with rank v and set each switch along the path whenever the path descends to its right child. To implement this in a correct manner, the `WriteMax`(v)

```
Shared data:
switch: a single bit multi-writer register, initially 0
left: a $\mathsf{MaxRegister}_{m'}$ object, where $m' = \lceil m/2 \rceil$, initially 0,
right: a $\mathsf{MaxRegister}_{m-m'}$ object, initially 0

procedure WriteMax($r, v$)
    if $v < m'$ then
        if $r$.switch $= 0$ then
            WriteMax($r$.left, $v$)
    else
        WriteMax($r$.right, $v - m'$)
        $r$.switch $\leftarrow 1$

procedure ReadMax($r$)
    if $r$.switch $= 0$ then
        return ReadMax($r$.left)
    else
        return ReadMax($r$.right) $+ m'$
```

Algorithm 1: Implementation of `WriteMax`(r, v) and `ReadMax`(r) for a $\mathsf{MaxRegister}_m$ object called r.

operation needs to traverse this path from the leaf v to the root, in order for the relevant switches down any subtree to already be set once a (possibly concurrent) `ReadMax` operation traveling down from the root reaches this subtree. Moreover, to guarantee linearizability, the `WriteMax`(v) operation needs to avoid setting switches in any subtree which is a left subtree of a switch that is already set (which implies that a value v' that is larger than v is already written). Hence, a `WriteMax`(v) operation is implemented using a pass from the root to leaf v followed by a pass from leaf v back to the root: First, the downward pass starting at the root looks for larger values; these correspond to 1 bits in nodes where v's path would contain a 0 bit. If a larger value exists, the `WriteMax` exits immediately. This is needed in order for the implementation to be linearizable. Once the operation reaches the leaf v, the upward pass starting at v writes a 1 to each switch bit whose right child is on the path to v. (See Algorithm 1.) It is shown in [3] that this procedure gives a linearizable execution even with concurrent `ReadMax` and `WriteMax` operations.

The number of steps for a `WriteMax`(v) operation is at most twice the depth of the corresponding leaf. This gives a cost for `WriteMax` that is asymptotically equal to `ReadMax`, and that depends on the structure of the tree.

Aspnes *et al.* [3] show a matching $\Omega(\min(\log v, n))$ lower bound for deterministic obstruction-free max register implementations from atomic registers. For randomized implementations, they show a weaker lower bound of $\Omega(\log n/\log\log n)$ steps for `ReadMax` operations of n-bounded-value max registers, when `WriteMax` operations have polylogarithmic step complexity. This lower bound is obtained as a trade-off between the complexities of `ReadMax` and `WriteMax` operations, where $\Omega(\log n/\log(w\log n))$ steps are shown to be required for `ReadMax` operations if w is an upper-bound for `WriteMax` operations. This lower bound holds even for a weaker oblivious adversary.

We will show that with randomization, the dependence on v can be eliminated. It is possible to build a max register and a 2-component max array (and thus a snapshot object), whose cost is polylogarithmic in n with high probability for all operations, regardless of the size of the values it contains. We will only require a bounded-increments assumption, to be formally defined next, which is fulfilled in our usage for constructing a snapshot object.

2.2 An unbounded-value max register implementation

We now show how to extend the results of [3] to allow an *unbounded-value* max register that nonetheless has an $O(\log n)$ cost per operation with high probability. That is, the cost per operation is only logarithmic in n and does not depend on the value being read or written, despite allowing it to be unbounded. The first step is to bound the cost of `WriteMax` operations. We will do this under the assumption of k**-bounded increments**, which we will define by the rule that the input for each new `WriteMax` operation is a value v that is at most k more than the largest input to any previously initiated `WriteMax` operation.[6] We will later choose k such that the step complexity will be logarithmic. This assumption will be justified later by the details of our unlimited-use snapshot construction.

As in a standard max register, the core of our unbounded-value max register is a binary tree of switch bits. But now the tree is infinite, consisting of an infinite **spine** forming the rightmost path through the tree together with an infinite collection of m-valued max registers forming the rest of

[6]Note that we do not require that this previous `WriteMax` operation finished.

the tree. Each of these max registers is itself implemented as a balanced $\lceil \log m \rceil$-depth tree, where m is an integer that will be chosen later, rooted at the left child of some node in the spine (see Figure 1). Using this tree with the original algorithm, a `WriteMax`(v) operation must walk all the way from the root of the tree to the corresponding leaf, which will be found in the $\lceil v/m \rceil$-th m-valued max register. It must then walk back up to the root, setting `switch` bits as needed, giving a cost of $O(v/m + \log m)$.

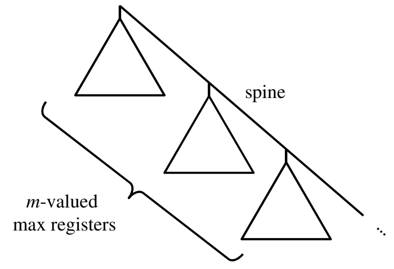

Figure 1: An unbounded-value max register.

In our algorithm, we assume that the tree is packed in memory so that a `WriteMax`(v) operation can access the root of the $\lceil v/m \rceil$-th max register directly. Within this subtree, it executes the standard algorithm; but along the spine, it sets only as many `switch` bits as are needed to guarantee that all ancestors are set; this is checked by performing an embedded `ReadMax` operation. This optimization does not affect correctness, because setting switches that are already set farther up the spine has no effect. What it does give is an improvement to the step complexity under the assumption of k-bounded increments, since between the n processes v can have increased by at most kn above the value of the last complete `WriteMax`, meaning that only kn/m steps up the spine are needed.

Setting aside for the moment the cost of the `ReadMax`, this gives a cost for the `WriteMax` of $O(\log m)$ for updating the m-valued max register plus $O(kn/m)$ for updating the segment of the spine. We will later choose k and m in a way for which the above results in $O(\log n)$ steps per `WriteMax` operation, assuming bounded increments. This complexity does not depend on the value being written or the use of randomization.

However, the `ReadMax` operations still suffer from the problem mentioned earlier: they are not wait-free in the presence of concurrent `WriteMax` operations with increasing values. For this we add the additional mechanism of randomized helping. Algorithm 2 is pseudo-code for our implementation, where $\texttt{WrapWriteMax}_i$ and $\texttt{WrapReadMax}_i$ are the operations for process p_i, which invoke WriteMax and ReadMax operations as in [3] on the m-valued max registers (in which the process id does not matter). The specifications of $\texttt{WrapWriteMax}_i$ and $\texttt{WrapReadMax}_i$ are those of WriteMax and ReadMax, respectively. Our analysis will show that $\texttt{WrapWriteMax}_i$ and $\texttt{WrapReadMax}_i$ operations are linearizable, and prove their step complexity.

We now provide a high-level description of the helping mechanism. helpVal is an array where helpVal$[i]$ holds a value that is recorded by process p_i during a $\texttt{WrapWriteMax}_i$ operation. This is the maximum between its input and a value it obtains by performing an embedded $\texttt{WrapReadMax}_i$ operation (which could be larger than its input). helpTS is a two-dimensional array where helpTS$[i][j]$ holds a timestamp for process p_j, as seen by process p_i. The pointer array holds process ids. The use of these arrays is as follows.

A $\texttt{WrapWriteMax}_i$ operation cycles over the process ids, helping one process each time it is invoked, in a round-robin fashion. The operation then reads the *timestamp*, TS$[s]$, which is the sequence number of a $\texttt{WrapReadMax}_s$ operation that is associated with the current process, s, helped by $\texttt{WrapWriteMax}_i$, and is written to TS$[s]$ by the $\texttt{WrapReadMax}_s$ operation. It then reads the value v' of the max register, and if the value v it needs to write is larger than v' it goes ahead and writes it into the max register. It then records the maximum between v and v' into the helpVal array, along with the timestamp it saw for s in the helpTS array, and updates the pointer array with its process id. A $\texttt{WrapReadMax}_i$ operation first increments its timestamp and then takes a certain number of steps reading the max register. For this, we use the notation $\texttt{ReadMax}(\ell, t)$ to denote an execution of t steps of the ReadMax operation starting from position ℓ in the spine. If the ReadMax does not finish within t steps, indicated by returning $\perp$, the $\texttt{WrapReadMax}_i$ operation tries to get help from a random process chosen from a random location in the pointer array. Getting help is done by checking whether the chosen helping process, p_j, holds the current timestamp of process p_i, performing the $\texttt{WrapReadMax}_i$ operation, and if so, taking its value from its helping array.

The idea behind the proof is that if a ReadMax operation takes too many steps trying to read the max register without finishing, it must be that there are many concurrent WriteMax operations that keep sending it down the spine. But in such a case, the $\texttt{WrapReadMax}_i$ operation finds a value in the help array that it may use, in the sense that it was updated by one of these concurrent $\texttt{WrapWriteMax}_j$ operations – specifically, after the $\texttt{WrapReadMax}_i$ operation started.

```
Shared Data:
array $\mathsf{TS}[0..n-1]$ where $\mathsf{TS}[i] = \textit{timestamp}$ for process $p_i$, initially 0
array $\mathsf{pointer}[1..n^3]$; each entry is a process id
array $\mathsf{helpVal}[0..n-1]$; entry $\mathsf{helpVal}[i]$ is the largest value seen by a
  $\mathtt{WrapWriteMax}_i$ operation
array $\mathsf{helpTS}[0..n-1][0..n-1]$; entry $\mathsf{helpTS}[i][j]$ is the most recent
  timestamp of $p_j$ seen by a $\mathtt{WrapWriteMax}_i$ operation
Persistent Local Data:
integer $\ell$; initially 0
integer $s$; initially 0
integer $\textit{spineLoc}$; initially 0
procedure $\mathtt{WrapWriteMax}_i(v)$
    $s \leftarrow s + 1 \bmod n$
    $t \leftarrow \mathsf{TS}[s]$
    $v' \leftarrow \mathtt{WrapReadMax}_i()$
    if $v > v'$ then
        $\mathtt{WriteMax}(M_{\lfloor v/m \rfloor}, v \bmod m)$ // WriteMax to the
          corresponding $m$-valued max register
        for $j = \lfloor v/m \rfloor$ to $\lfloor v'/m \rfloor$ do
            $\mathsf{spine}[j] \leftarrow 1$
    $\mathsf{helpVal}[i] \leftarrow \max(v, v')$
    $\mathsf{helpTS}[i][s] \leftarrow t$
    $\mathsf{pointer}[\ell] \leftarrow i$
    $\ell \leftarrow \ell + 1 \mod (n^3)$
procedure $\mathtt{WrapReadMax}_i()$
    $\mathsf{TS}[i] \leftarrow \mathsf{TS}[i] + 1$
    while $\textit{true}$ do
        $\textit{val} \leftarrow \mathtt{ReadMax}(\textit{spineLoc}, c' \log m)$ // For a constant $c'$, to be
          fixed in the step complexity proof
        if $\textit{val} \neq \perp$ then
            return $\textit{val}$
        else
            $j \leftarrow \mathsf{pointer}[\mathtt{random}(1, \ldots, n^3)]$
            $\textit{help} \leftarrow \mathsf{helpTS}[j][i]$
            $\textit{val} \leftarrow \mathsf{helpVal}[j]$
            if $\textit{help} = \mathsf{TS}[i]$ then
                return $\textit{val}$
            else
                $\textit{spineLoc} \leftarrow \max(\textit{spineLoc}, \lfloor \textit{val}/m \rfloor)$
```

Algorithm 2: Max register with randomized helping; code for process p_i.

Next, we proceed with the formal proof. Let spine be the array induced by the switch bits on the spine of the tree. Let M_i be the m-valued max register whose root is the left child of spine$[i]$.

We base our proof on the correctness proof of the max register construction in [3]. Loosely speaking, the proof in [3] orders `ReadMax` and `WriteMax` operations that read 0 from a switch (henceforth C_{left}) before `ReadMax` operations that read 1 from the switch and `WriteMax` operations that set the switch (henceforth C_{right}). `WriteMax` operations that would need to write to the left child of the switch but read 1 from the switch are linearized at the latest point possible before other operations start. Within each of the sets C_{left} and C_{right}, operations are ordered according to their linearization order when accessing the left and right child of the switch, respectively. It is then shown that linearizability follows by induction on the structure of the tree of switches.

We need to address two issues that differ in our implementation. First, we need to address $\texttt{WrapWriteMax}_i$ operations and show that the switches leading to a written value are indeed set by the time the $\texttt{WrapWriteMax}_i$ operation terminates, showing that our linearization is well defined. The second issue is that we need to address $\texttt{WrapReadMax}_i$ operations that return in Line 33.

It is worth mentioning that, as the proof below shows, we do not need the assumption of k-bounded increments for linearizability of the construction. This assumption is used only for bounding the step complexity.

Theorem 2.1. *Algorithm 2 is a linearizable implementation of an unbounded-value max register.*

Proof. Since `ReadMax` and `WriteMax` operations are linearizable, we can use their linearization points when linearizing `WrapReadMax` and `WrapWriteMax` operations.

We denote by S_v the set of switches on the path from the root to the leaf corresponding to v for which the path descends to their right child on the tree. We linearize a $\texttt{WrapWriteMax}_i(v)$ operation op_i at the first point during its execution in which all the switches in S_v are set, if the condition at Line 14 holds. Otherwise, we linearize op_i at the linearization point of the embedded $\texttt{WrapReadMax}_i$ operation it performs in Line 13. We linearize a $\texttt{WrapReadMax}_i$ operation op_i that returns in Line 27 at the time the $\texttt{ReadMax}(spineLoc, c' \log m)$ operation it last performed on Line 25 is linearized. For a $\texttt{WrapReadMax}_i$ operation op_i that returns in Line 33, let p_j be the process whose index j is read in Line 29, let v be the value read by $\texttt{WrapReadMax}_i$ in Line 31 and let op_j be the $\texttt{WrapWriteMax}_j$ operation that last wrote v to $\mathsf{helpVal}[j]$ prior to this read. If the value v' that op_j read in Line 13 satisfies $v' \geq v$, then we linearize op_i at the linearization point of the $\texttt{WrapReadMax}_j$ operation by p_j in Line 13. Otherwise, we linearize op_i immediately after the linearization point of op_j. In all cases, all operations linearized at a certain point either do a `ReadMax` or a `WriteMax` with the same value to the max register. Therefore, we linearize all such `WrapWriteMax` operations in an arbitrary order, and all `WrapReadMax` operations after them, in an arbitrary order.

We use an induction on the order of linearization points to prove the correctness of the linearization. Correctness means that all operations are linearized within their execution interval, and each `WrapReadMax` operation returns a value which is the largest input to any `WrapWriteMax` operation linearized before it. We add to the inductive claim the invariant that all switches in S_v for a value v written by a `WrapWriteMax` operation *op* that reads $v' < v$ in Line 13 are set by the time *op* finishes, for all `WrapWriteMax` operations *op* that have been linearized up to the current point. The inductive claim (both the basic correctness and the additional invariant) clearly holds for the base case, when no operation has yet been performed.

Assume that the linearization is correct up to some operation $t-1$ in the total order it induces, and that all switches in S_v for a value v written by a `WrapWriteMax` operation *op* that reads $v' < v$ in Line 13 are set by the time *op* finishes, for all `WrapWriteMax` operations *op* that have been linearized up to the current point. Let *op* be the t-th operation. First, assume *op* is a $\texttt{WrapWriteMax}_i(v)$ operation that reads $v' < v$ in Line 13. By construction, all appropriate switches inside $M_{\lfloor v/m \rfloor}$ are set in Line 15. By the induction hypothesis, all spine switches in $S_{v'}$ are set. The loop in Line 16 then shows that the invariant still holds.

Next, correctness for $\texttt{WrapReadMax}_i$ operations that return in Line 27 now follows from the proof in [3], since they return a value read by $\texttt{ReadMax}(\mathit{spineLoc}, c' \log m)$ and the linearization points used in this case are the same as the linearization points used in [3].

Let *op* be a $\texttt{WrapReadMax}_i$ operation that returns in Line 33. Let t be the timestamp read by *op* in Line 30. Let op' be a $\texttt{WrapWriteMax}_j(v)$ operation by p_j that last writes t to $\mathsf{helpTS}[j][i]$ prior to this read. Let op'' be the $\texttt{WrapReadMax}_j$ operation performed by op' in Line 13.

If $v' \geq v$, where v' is the value returned by op'', then the linearization point of *op* is that of op''. Note that before op' calls op'', op' reads the timestamp that *op* wrote to $\mathsf{TS}[i]$. The $\texttt{WrapReadMax}_j$ operation op'' is also performed before op' writes to $\mathsf{helpTS}[j][i]$ and hence before *op* finishes because *op* reads the value that op' writes to $\mathsf{helpTS}[j][i]$. Therefore, the linearization point of op'' is within the execution interval of op and is the same as the linearization point of op'. By the correctness of the linearization points of the construction in [3], the value returned by op, which is the maximum between the value returned by op'' and the value written by op', is the largest value written by operations that are linearized before op.

Otherwise, $v' < v$. Then the linearization point of op is immediately after that of op'. This is again within the execution interval of op because the linearization point of op' is before it writes to $\mathsf{helpTS}[j][i]$ the value read by op. Moreover, by the correctness of the linearization points of the construction in [3], the return value of op is the largest value written by operations that are linearized before it. $\square$

Having shown that this implementation is linearizable, we turn to prove its logarithmic step complexity. We denote $M = n^4$, and choose $m = Mk = n^4k$ and $k = n^d$ for some constant $d > 0$. We begin with a simple claim regarding the contents of the pointer array.

Consider any execution containing at least M linearized `WrapWriteMax` operations. Let t be any point in this execution before which at least M `WrapWriteMax` operations have been linearized. For any `WrapWriteMax` operation op linearized before t, let $v(op)$ denote the value it writes to helpVal in Line 18 and, for any set S of such operations, let $v(S)$ denote the minimum of $v(op)$ over all operations $op \in S$. Let v_t denote the maximum of $v(S)$ over all M element subsets S of `WrapWriteMax` operations linearized before t. Finally, let W_t denote the set of all `WrapWriteMax` operations op linearized before t such that $v(op) \geq v_t$.

Claim 2.2. *At any time t before which at least M* `WrapWriteMax` *operations have been linearized, all elements of the* pointer *array except at most $2n$, are process indexes i such that* $\mathsf{helpVal}[i] \geq v_t$.

Proof. Among the $M = n^4$ operations in W_t, there are at least n^3 that are performed by the same process. From Lines 20 and 21, this process writes to each of the n^3 locations in the pointer array. Clearly, at most n of the operations in W_t may be pending to write to the pointer array. In addition, at most $n - 1$ locations may be overwritten by processes p_j for which $\mathsf{helpVal}[j] < v_t$.

To prove the last observation, we argue that there is a process p_k, for some $k \in \{0, \dots n-1\}$, for which there cannot be any operation op_k that overwrites locations in the pointer array while $\mathsf{helpVal}[k] < v_t$, and that for any other process p_j, $j \neq k$, there can be at most one operation op_j hat overwrites some location in the pointer array while having

$\mathsf{helpVal}[j] < v_t$. Both of these follow from the fact that for a location in $\mathsf{pointer}$ holding a process index i with $\mathsf{helpVal}[i] = v$ to be overwritten by a process p_j with a smaller value $\mathsf{helpVal}[j] = v'$, it has to be the case that the `WrapWriteMax` operation that writes v' to $\mathsf{helpVal}[j]$ starts before the `WrapWriteMax` operation that writes v to $\mathsf{helpVal}[i]$ finishes, but finishes after it. This means the following. (A) For the first process p_k that writes to $\mathsf{pointer}$ while having $\mathsf{helpVal}[k] \geq v_t$, no operation op_k can overwrite a location in $\mathsf{pointer}$ while having $\mathsf{helpVal}[k] < v_t$, because any subsequent operation has $\mathsf{helpVal}[k] \geq v_t$. (B) For any other process p_j, $j \neq k$, there can be at most one operation op_j that overwrites a location in $\mathsf{pointer}$ while having $\mathsf{helpVal}[j] < v_t$, because any subsequent operation op'_j has $\mathsf{helpVal}[j] \geq v_t$. This is since op'_j starts after op_j finishes, and therefore also after the operation op whose location in $\mathsf{pointer}$ was overwritten is linearized, therefore the embedded `WrapReadMax` of op'_j must return a value that is at least v_t.

□

Theorem 2.3. *For $k = n^d$ and $m = n^4 k$, the step complexity of operations in Algorithm 2 is $O(\log n)$ with high probability against an adaptive adversary, provided increments are k-bounded.*

Proof. Let op_i be a $\texttt{WrapReadMax}_i$ operation by p_i. We show that there is a constant c' such that op_i finishes after $2c' \log m$ steps with high probability.

The operation first starts by trying a `ReadMax` for $c' \log m$ steps on Line 25, where the constant c' is such that $c' \log m$ steps are enough to reach and read two m-valued max registers.

Assume op_i does not finish within $2c' \log m$ steps. For this to happen, op_i takes at least $c' \log m$ steps down the spine (otherwise, it goes down some m-valued max register and terminates within another $c' \log m$ steps). Since $m = kM$, this means that $2kM$ values can be reached. By the k-bounded increments assumption, there are at least $2M$ values written or being written for this to happen, because written values can be no more than k apart, and there are $2kM$ values that are represented by the leaves of the subtrees that can be reached by op_i. This implies that by the point t in which op_i accesses the $\mathsf{pointer}$ array in Line 29, it holds that at least $2M - n \geq M$ `WrapWriteMax` operations have been linearized. Hence, by Claim 2.2, the $\mathsf{pointer}$ array contains at least $n^3 - 2n$ elements that are indexes of processes p_i for which $\mathsf{helpVal}[i]$ is among the values written by operations in W_t. Therefore, with probability at least $1 - 2n/n^3 = 1 - O(1/n)$, when p_i accesses the $\mathsf{pointer}$ array it obtains a value v whose ancestor in the spine is at a location which is at most $c' \log m$ switches above the deepest set switch on the spine.

We say that a process p_j is **current for operation** op_i if $\mathsf{helpTS}[j][i] = \mathsf{TS}[i]$, where $\mathsf{TS}[i]$ is the timestamp written by op_i. Every process p_j can perform at most n $\texttt{WrapWriteMax}_j$ operations before it becomes current for op_i, since p_j helps processes in round-robin order.

If the value v, which is obtained by p_i when it accesses the $\mathsf{helpVal}$ array of the process whose index it reads from $\mathsf{pointer}$, was not a value of an operation of a process that is current for p_i, then p_i performs another $c' \log m$ steps of the loop in Line 24, starting from the spine location where v is rooted. Recall that with probability at least $1 - O(1/n)$ when p_i accesses the $\mathsf{pointer}$ array it obtains a value v whose ancestor in the spine is at a location which is at most $c' \log m$ switches above the deepest set switch on the spine. Therefore, if p_i does not finish within this additional loop, then with probability at least $1 - O(1/n)$, there are at least M $\texttt{WrapWriteMax}$ operations that started after p_i and have been linearized by this time, i.e., they begin after $\mathsf{TS}[i]$ is incremented. Then, at most n^2 of these operations are by processes that are not current for op_i, because a process p_j becomes current for op_i by p_i every n operations of p_j because it helps processes in round-robin. There can be at most n^2 different locations in the $\mathsf{pointer}$ array written by such processes, plus at most $n-1$ locations that have operations by current processes pending to write them, but still contain previous values. The rest of the n^3 locations hold values written by processes that are current for op_i. This implies that the probability of op_i choosing a random location in $\mathsf{pointer}$ that was written by a process that is current for it is at least $1 - (n + n^2)/n^3 = 1 - O(1/n)$. Therefore, with high probability, op_i finishes within $O(\log m) = O(\log n)$ steps.

Finally, a $\texttt{WrapWriteMax}_i$ operation op_i takes $O(\log m + kn/m)$ steps in addition to calling $\texttt{WrapReadMax}_i$, because a $\texttt{WriteMax}$ operation to an m-bounded max register takes $O(\log m)$ steps and the loop in Line 16 takes $O(kn/m)$ steps, since, by the k-bounded increment assumption, v is at most kn larger than v'.

Since $k = n^d$ and $m = Mk = n^4 k$, we have that $kn/m = O(\log n)$ and therefore the number of steps required for this operation is also $O(\log n)$, completing the proof. □

3 Unbounded-value max arrays with bounded increments

A **2-component max array** A implements a pair of max registers $A[\mathsf{left}]$ and $A[\mathsf{right}]$, each of which can be updated by a $\texttt{MaxUpdate}$ operation, and a `MaxScan` operation that returns the tuple $\langle A[\mathsf{left}], A[\mathsf{right}]\rangle$, holding the maximal values written to each of the two components in all `MaxUpdate` operations linearized before it.

In this section we give an implementation of an unbounded 2-component max array using randomized helping. `MaxUpdate` and `MaxScan` operations in this implementation take $O(\log^2 n)$ steps assuming bounded increments.

The max registers $A[\mathsf{left}]$ and $A[\mathsf{right}]$ are implemented directly using the bounded-increment max registers from Algorithm 2, and indeed `MaxUpdate` operations to the two sides of the max array translate directly into writes to these max registers. However, some additional mechanisms are needed to support the `MaxScan` operation.

The main technique is splitting each max register value v into a **high part** $\mathtt{high}(v) = \lfloor v/m \rfloor$ and a **low part** $\mathtt{low}(v) = v \bmod m$. When reading the max array, the algorithm takes a snapshot of the high parts using a double collect, as in the classic atomic snapshots algorithm of Afek *et al.* [1]. As in the Afek *et al.* algorithm, a reader will either obtain a **direct scan** derived from this double collect or an **indirect scan** provided by some writer through the help array.

Because the double-collect termination test looks only at the high parts, a separate mechanism is need to make the low parts of each direct scan consistent with all other reads that return the same high parts. But these low parts are bounded by m, so for each distinct value of the high parts, we can use a standard bounded-value max array for this purpose. The parameter m is adjusted, based on the bound on increments, so that the high parts will change infrequently enough that a reader that cannot obtain a direct scan will be concurrent with many write operations. This means that the reader will obtain an indirect scan after a few attempts on average.

For convenience, we define an inverse operation $\mathtt{combine}(h, \ell) = h \cdot m + \ell$, so that $\mathtt{combine}(\mathtt{high}(v), \mathtt{low}(v)) = v$. We also abuse notation by allowing $\mathtt{high}$, low, and $\mathtt{combine}$ to distribute across tuples, so that $\mathtt{high}(\langle x, y\rangle) = \langle \mathtt{high}(x), \mathtt{high}(y)\rangle$, $\mathsf{low}(\langle x, y\rangle) = \langle \mathsf{low}(x), \mathsf{low}(y)\rangle$, and $\mathtt{combine}(\langle x, y\rangle, \langle x', y'\rangle) = \langle \mathtt{combine}(x, x'), \mathtt{combine}(y, y')\rangle$.

Let the **round** associated with a pair of values $v = \langle v[\mathsf{left}], v[\mathsf{right}]\rangle$ be $\mathtt{high}(v[\mathsf{left}]) + \mathtt{high}(v[\mathsf{right}])$. Because the high parts are always obtained from snapshots, there will be a unique tuple of high parts associated with each round r. To reach agreement on the low parts, we use a separate m-bounded max array $A.\mathsf{low}[r]$ for each r. We will also need to enforce comparability across rounds using a mechanism similar to the switch bits in the standard max register construction: the bit $A.\mathsf{closed}[r]$, implemented using an atomic register, indicates that no further values can be written to round r.

These closed bits are filled in by readers when they observe values in $A[\mathsf{left}]$ and $A[\mathsf{right}]$ that exceed the current round. Each process p maintains a persistent variable $A.\mathsf{filled}[p]$ such that p can deduce that $A.\mathsf{closed}[r] = 1$ for all $r < A.\mathsf{filled}[p]$. The value of $A.\mathsf{filled}[p]$ is updated either when p sets some $A.\mathsf{closed}[r]$ directly, or when it learns that more closed bits have been set through the helping mechanism.

The final data structure is the helping array. Each process p advertises a timestamp in an atomic register $A.\mathsf{TS}[p]$, and looks for values at least as recent as this timestamp in a random location of an array $A.\mathsf{help}[p]$ of n^2 atomic registers $A.\mathsf{help}[p][0..n^2 - 1]$. The content of each of these registers is a tuple $\langle t, x\rangle$, where t is a timestamp (initially 0) and x, if not its initial null value $\perp$, is a return value from some `MaxScan` operation embedded in some `MaxUpdate` operation.

Each process p also maintains some persistent internal variables to track which process $A.\mathsf{toHelp}[p]$ it will help as part of its next `MaxUpdate` operation and which position $A.\mathsf{helpPos}[p]$ in that process's array it will write to during that operation.

For helping the `MaxScan` operations, the `MaxUpdate` operations of each process cycle through the processes in a round-robin manner and fill in the $A.\mathsf{help}[p]$ array for each process p.

Pseudocode for the `MaxScan` and `MaxUpdate` operations is given in Algorithms 3.1 and 3.2.

3.1 Proof of correctness

We start by defining a linearization ordering. We then prove some technical lemmas about executions of the algorithm, show that the linearization ordering is consistent with the observed order of operations, and show that all `MaxScan` operations return values consistent with the linearization ordering. We defer computing the running time of the algorithm until the following section.

3.1.1 Constructing a linearization

Given a concurrent execution H of Algorithms 3.1 and 3.2, we construct a linearization in two stages. First, we define an ordering $<_{\mathsf{low}}$ of all operations on the max arrays $A.\mathsf{low}[r]$, where an operation π on $A.\mathsf{low}[r]$ is ordered before an operation π' on $A.\mathsf{low}[r']$ if $r < r'$ or if $r = r'$ and π precedes π'

```
procedure MaxScan($A$)
    $A$.TS[$p$] $\leftarrow$ $A$.TS[$p$] $+ 1$;
    while true do
        // Check help
        $\langle t, x \rangle \leftarrow$ $A$.help[$p$][random()];
        if $t = A$.TS[$p$] then
            // $x$ is a current value for $p$
            return $x$;
        else
            // $x$ implies some closed bits are set
            $A$.filled[$p$] $\leftarrow \max($$A$.filled[$p$], high($x$[left]) $+$ high($x$[right]));
        // First collect of $A$.raw
        $y$[left] $\leftarrow$ $A$.raw[left];
        $y$[right] $\leftarrow$ $A$.raw[right];
        round $\leftarrow$ high($y$[left]) $+$ high($y$[right]);
        // Do we need to fill more?
        if $A$.filled[$p$] $<$ round then
            // Yes
            $A$.closed[$A$.filled[$p$]] $\leftarrow 1$;
            $A$.filled[$p$] $\leftarrow$ $A$.filled[$p$] $+ 1$;
        else
            // No, $A$.filled[$p$] $=$ round
            // Check to see if this round is closed
            if $A$.closed[round] $= 0$ then
                // Second collect of $A$.raw
                $z$[left] $\leftarrow$ $A$.raw[left];
                $z$[right] $\leftarrow$ $A$.raw[right];
                if high($z$) $=$ high($y$) $\wedge$ $A$.closed[round] $= 0$ then
                    // Successful double collect on high($A$.raw)
                       and $A$.closed.
                    // low($z$) read after round $- 1$ closed but
                       before round closed.
                    // Use $A$.low(round) to get consistency with
                       other reads with same round.
                    $A$.low[round][left] $\leftarrow$ low($z$[left]);
                    $A$.low[round][right] $\leftarrow$ low($z$[right]);
                    return combine(high($z$), MaxScan($A$.low[round]));
```

Algorithm 3.1: Implementation of MaxScan. Code for process p.

```
1 procedure MaxUpdate($A$, side, $v$)
2     $A$.raw[side] $\leftarrow v$;
      // Perform a helping MaxScan
3     $t \leftarrow A$.TS[$A$.toHelp[$p$]];
4     $x \leftarrow$ MaxScan($A$);
5     $A$.help[$A$.toHelp[$p$]][$A$.helpPos[$p$]] $\leftarrow \langle t, x \rangle$;
      // Increment help position
6     $A$.helpPos[$p$] $\leftarrow$ ($A$.helpPos[$p$] $+ 1$) mod $n^2$;
7     if $A$.helpPos[$p$] $= 0$ then
8         $A$.toHelp[$p$] $\leftarrow$ ($A$.toHelp[$p$] $+ 1$) mod $n$;
```

Algorithm 3.2: Implementation of `MaxUpdate`. Code for process p.

in H. In other words, we sort these operations first by round and then by their actual execution order within each round.

Next, we define a linearization order $<_S$ on all operations on A, including "internal" `MaxScan` operations called by `MaxUpdate` operations in Line 4 of Algorithm 3.2, by mapping each such operation ρ to an operation $f(\rho)$ on $A.\mathsf{low}[r]$ for some r, and letting $\rho <_S \rho'$ if and only if $f(\rho) <_{\mathsf{low}} f(\rho')$. We refer to the operation $f(\rho)$ as the **critical step** of ρ, and the round r of the object $A.\mathsf{low}[r]$ that $f(\rho)$ operates on as the **round** of ρ.

In a sense, we are using the critical step to assign a linearization point for ρ in the $<_{\mathsf{low}}$ ordering. However, this is not a linearization point in the standard sense of identifying a particular time in H, because the $<_{\mathsf{low}}$ ordering may place operations in different rounds out of order with respect to their ordering in H. Nonetheless we will show that this ordering $<_S$ is consistent with the partial order $<_H$ defined by $\rho <_H \rho'$ if and only if ρ finishes in H before ρ' starts. Together with an argument that all `MaxScan` operations return values consistent with a sequential max-array execution, this will show that $<_S$ is in fact a linearization of H.

For the purposes of the argument, we assume that all operations in H are complete; if not, we can extend H by running all pending operations to completion, and apply our construction to the extended execution.

We now give the details of the mapping f. Each operation ρ of A in H falls into one of three categories:

1. Direct-scan `MaxScan` operations. These are operations that successfully complete a double collect and read from $A.\mathsf{low}[\mathsf{round}]$ in Line 22. The critical step $f(\rho)$ is the `MaxScan` operation on $A.\mathsf{low}[\mathsf{round}]$.

2. Indirect-scan `MaxScan` operations. These are operations that successfully obtain a current scan from $A.\mathsf{help}[p][i]$ for some i in Line 4. The critical step $f(\rho)$ is defined as $f(\rho')$, where ρ' is the `MaxScan` operation called by the `MaxUpdate` operation that last wrote to $A.\mathsf{help}[p][i]$.

3. `MaxUpdate` operations. Given a `MaxUpdate` operation σ that writes to $A.\mathsf{raw}[\mathsf{side}]$, consider the set of all `MaxScan` operations that both read from $A.\mathsf{raw}[\mathsf{side}]$ after σ writes to it, and subsequently write to $A.\mathsf{low}[\mathsf{round}][\mathsf{side}]$ for some round in Line 20 or 21. If an operation is a member of this set, we say that it **observes** σ.

   This set is nonempty, because the embedded `MaxScan` in σ is a member. Let r be the smallest round for which some operation in this set writes to $A.\mathsf{low}[r][\mathsf{side}]$, and $f(\sigma)$ be the earliest write to $A.\mathsf{low}[r][\mathsf{side}]$ by an operation in this set.

3.1.2 Basic properties of the algorithm

Here we prove some technical lemmas that are used in the main arguments.

The following lemma gives an invariant relating the filled values to the closed array.

Lemma 3.1. *The following invariant holds in every state of every execution of Algorithms 3.1 and 3.2:*

1. *For every pair $\langle t, x\rangle$ that appears in some register $A.\mathsf{help}[p][i]$, $A.\mathsf{closed}[j] = 1$ for all $j < \mathtt{high}(x[\mathsf{left}]) + \mathtt{high}(x[\mathsf{right}])$.*

2. *For each process p, $A.\mathsf{closed}[j] = 1$ for all $j < A.\mathsf{filled}[p]$.*

3. *For all i, if $A.\mathsf{closed}[i] = 1$, then $A.\mathsf{closed}[j] = 1$ for all $j < i$.*

Proof. Observe that the invariant holds in the initial state, as $A.\mathsf{closed}[j] = 0$ for all j. Observe also that once $A.\mathsf{closed}[j]$ is set for some j, so the invariant can only become false as the result of an operation that increases the range of j for which $A.\mathsf{closed}[j]$ must be set.

Now consider some step where the invariant holds before the step occurs, where the step increases the upper bound in j in one of the cases above. Then

1. If some process p writes $\langle t, x\rangle$ to $A.\mathsf{help}[q][i]$ as part of the helping stage of a `MaxUpdate` operation, then $\mathtt{high}(x[\mathsf{left}]) + \mathtt{high}(x[\mathsf{right}]) = \mathtt{high}(z[\mathsf{left}]) + \mathtt{high}(z[\mathsf{right}]) = \mathsf{round}$, where z and round are the values in that operation's embedded call to `MaxScan`. But the **return** line in

MaxScan is not reachable unless $A.\mathsf{filled}[p] \geq \mathsf{round}$. From the invariant preceding this step, $A.\mathsf{closed}[j] = 1$ for all $j < A.\mathsf{filled}[p]$, implying that this also holds for all $j < \mathtt{high}(x[\mathsf{right}]) + \mathtt{high}(x[\mathsf{left}]) \leq A.\mathsf{filled}[p]$.

2. If some process p increments $A.\mathsf{filled}[p]$ to r, it can only do so after setting $A.\mathsf{closed}[r-1]$ to 1. Since $A.\mathsf{closed}[j]$ must be 1 for all $j \leq r-1$, we get $A.\mathsf{closed}[j] = 1$ for all $j < r = A.\mathsf{filled}[p]$.

3. If some process p sets $A.\mathsf{closed}[i]$ then $i = A.\mathsf{filled}[p]$; it follows from the invariant that $A.\mathsf{closed}[j] = 1$ for all $j < i$.

□

For the following lemmas we need some notation. Let ρ be a MaxScan operation that ρ reaches Line 20. We denote by z_ρ and $\mathsf{round}_\rho = \mathtt{high}(z_\rho[\mathsf{left}]) + \mathtt{high}(z_\rho[\mathsf{right}])$ the values of the corresponding variables in ρ when ρ reaches Line 20. Moreover, we denote by t_{left} and t_{right} the linearization times of the ReadMax operations the last time ρ executes Lines 17 and 18, respectively.

Lemma 3.2. *Let A_t be the state of A at time t. Suppose some* MaxScan *operation ρ reaches Line 20. Then*

1. $\mathtt{high}(z_\rho) = \mathtt{high}(A_{t_{\mathsf{left}}}.\mathsf{raw})$*;*

2. $z_\rho[\mathsf{left}] = A_{t_{\mathsf{left}}}.\mathsf{raw}[\mathsf{left}]$*;*

3. $z_\rho[\mathsf{right}] = A_{t_{\mathsf{right}}}.\mathsf{raw}[\mathsf{right}]$*; and*

4. $A_s.\mathsf{closed}[\mathsf{round}_\rho - 1] = 1$ *and* $A_s.\mathsf{closed}[\mathsf{round}_\rho] = 0$*, for each s in* $\{t_{\mathsf{left}}, t_{\mathsf{right}}\}$*.*

Proof. It is immediate from the code that $z_\rho[\mathsf{left}] = A_{t_{\mathsf{left}}}.\mathsf{raw}[\mathsf{left}]$ and $z_\rho[\mathsf{right}] = A_{t_{\mathsf{right}}}.\mathsf{raw}[\mathsf{right}]$. Furthermore, these operations are guarded by the test that $A.\mathsf{filled}[p] \geq \mathsf{round}_\rho$, so $A_{t_{\mathsf{left}}}.\mathsf{closed}[\mathsf{round}_\rho - 1] = A_{t_{\mathsf{right}}}.\mathsf{closed}[\mathsf{round}_\rho - 1] = 1$ follows from Lemma 3.1. We also can't return from ρ without first reading $A.\mathsf{closed}[\mathsf{round}_\rho] = 0$ after both t_{left} and t_{right}; so $A_{t_{\mathsf{left}}}.\mathsf{closed}[\mathsf{round}_\rho] = A_{t_{\mathsf{right}}}.\mathsf{closed}[\mathsf{round}_\rho] = 0$ as well.

To show that $\mathtt{high}(z_\rho)$ captures exactly the high part of $A.\mathsf{raw}$ at time t_{left}, let t'_{left}, t'_{right} be the linearization times of the ReadMax operations the last time ρ executes Lines 9 and 10, respectively. These are the last values stored in y, and we have $\mathtt{high}(y) = \mathtt{high}(z_\rho)$ when ρ returns.

Because $A.\mathsf{raw}[\mathsf{left}]$ and $A.\mathsf{raw}[\mathsf{right}]$ are both max registers, their values are non-decreasing over time, so the fact that $\mathtt{high}(A_{t'_{\mathsf{left}}}.\mathsf{raw}[\mathsf{left}]) =$

$\mathtt{high}(A_{t_{\mathsf{left}}}.\mathsf{raw}[\mathsf{left}]) = \mathtt{high}(z_\rho[\mathsf{left}])$ and $\mathtt{high}(A_{t'_{\mathsf{right}}}.\mathsf{raw}[\mathsf{right}]) = \mathtt{high}(A_{t_{\mathsf{right}}}.\mathsf{raw}[\mathsf{right}]) = \mathtt{high}(z_\rho[\mathsf{right}])$ means that $\mathtt{high}(A_t.\mathsf{raw}) = \mathtt{high}(z_\rho)$ for any t such that $\max(t'_{\mathsf{left}}, t'_{\mathsf{right}}) = t'_{\mathsf{right}} \leq t \leq \min(t_{\mathsf{left}}, t_{\mathsf{right}}) = t_{\mathsf{left}}$. In particular, it holds for $t = t_{\mathsf{left}}$. □

Because the values returned by both components $A.\mathsf{raw}$ are nondecreasing over time, an immediate corollary of Lemma 3.2 is:

Corollary 3.3. *Let ρ and π be* `MaxScan` *operations that both reach Line 20. Then*

1. *If* $\mathsf{round}_\rho < \mathsf{round}_\pi$, $z_\rho < z_\pi$.
2. *If* $\mathsf{round}_\rho = \mathsf{round}_\pi$, $\mathtt{high}(z_\rho) = \mathtt{high}(z_\pi)$.

Proof. For the first part, Lemma 3.2 says that ρ reads $A.\mathsf{raw}[\mathsf{left}]$ and $A.\mathsf{raw}[\mathsf{right}]$ for the last time while $A.\mathsf{closed}[\mathsf{round}_\rho] = 0$; while π reads $A.\mathsf{raw}[\mathsf{left}]$ and $A.\mathsf{raw}[\mathsf{right}]$ for the last time while $A.\mathsf{closed}[\mathsf{round}_\pi - 1] = 1$. Since $\mathsf{round}_\rho \leq \mathsf{round}_\pi - 1$, Lemma 3.1 implies that $A.\mathsf{closed}[\mathsf{round}_\rho]$ is also 1 when π performs these reads. It follows that π does its reads after ρ, and so $z_\rho \leq z_\pi$. But they cannot be equal, since $\mathsf{round}_\rho \neq \mathsf{round}_\pi$. It follows that $z_\rho < z_\pi$.

For the second part, we have that there are times t_ρ and t_π such that $\mathtt{high}(z_\rho) = \mathtt{high}(A_{t_\rho}.\mathsf{raw})$ and $\mathtt{high}(z_\pi) = \mathtt{high}(A_{t_\pi}.\mathsf{raw})$. Assume without loss of generality that $t_\rho \leq t_\pi$. Then $\mathtt{high}(z_p) = \mathtt{high}(A_{t_\rho}.\mathsf{raw}) \leq \mathtt{high}(A_{t_\pi}.\mathsf{raw}) = \mathtt{high}(z_\pi)$. But we also have that $\mathtt{high}(z_\rho[\mathsf{left}]) + \mathtt{high}(z_\pi[\mathsf{right}]) = \mathtt{high}(z_\pi[\mathsf{left}]) + \mathtt{high}(z_\pi[\mathsf{right}])$. So $\mathtt{high}(z_\rho) = \mathtt{high}(z_\pi)$. □

Lemma 3.4. *Suppose that v_ρ is an indirect scan returned by a* `MaxScan` *operation ρ in Line 6 of Algorithm 3.1. Then there exists a* `MaxScan` *operation ρ' that returns $v_{\rho'} = v_\rho$, such that ρ' starts after and finishes before ρ.*

Proof. Let p be the process that carries out ρ. In order to return an indirect scan for ρ, ρ must read a tuple $\langle t, v_\rho \rangle$ from $A.\mathsf{help}[p][i]$ for some index i. From the code in Algorithm 3.2, such a value can only be written by a `MaxUpdate` operation σ that reads $A.\mathsf{TS}[p]$ after p sets it to t during ρ, and the value v_ρ is equal to the value returned by the internal `MaxScan` operation ρ' performed by σ. □

By applying Lemma 3.4 recursively, we ultimately reach some ρ' that not only returns the same value as ρ while executing within the interval of ρ, but also has the property of returning a direct scan in Line 22.

The following lemma shows consistency of the return values of different invocations of `MaxScan`, which is the first step to showing full linearizability.

3.1.3 Consistency of linearization with observable operation ordering

We begin by showing that the assigned rounds are ordered consistently with the concurrent execution. Define a round r as **active** if $A.\mathsf{closed}[r-1] = 1$ and $A.\mathsf{closed}[r] = 0$. (For the purposes of this definition, we take $A.\mathsf{closed}[-1]$ to be 1.) From Lemma 3.1, there is a unique active round at any time; in addition, the active round can only increase over time.

Lemma 3.5. *The execution interval for any complete operation in Algorithms 3.1 and 3.2 overlaps with a time at which its round is active.*

Proof. For a direct-scan `MaxScan` operation, use time t_{left}. For an indirect-scan `MaxScan` operation ρ, from Lemma 3.4 there exists a direct-scan `MaxScan` ρ' with the same round as ρ whose execution interval is a subset of that of ρ; apply Lemma 3.2 to ρ'.

This leaves the case of a `MaxUpdate`(A, side, v) operation σ. Let r be the round of σ. Let ρ be the embedded `MaxScan` operation that σ carries out in Line 4 of Algorithm 3.2. Let ρ' be the `MaxScan` operation that carries out σ's critical step $f(\sigma)$. Let t_ρ and $t_{\rho'}$ be the times t_{side} at which ρ and ρ' each read $A.\mathsf{raw}[\mathsf{side}]$ for the last time as defined in Lemma 3.2. Let $t = \min(t_\rho, t_{\rho'})$. Observe that t lies within the execution interval of σ, because (a) t_ρ occurs after σ starts and before it ends, and (b) $t_{\rho'}$ occurs after σ writes $a.\mathsf{raw}[\mathsf{side}]$.

We now argue that r is active at time t. Let $r_{\rho'} = r$ and r_ρ be the rounds of ρ' and ρ respectively. If $t = t_{\rho'}$, then the claim follows immediately from Lemma 3.2. If $t = t_\rho$, then (a) $r_{\rho'} \leq r_\rho$, since ρ' has the lowest round of any `MaxScan` operation that observes σ; and (b) $r_\rho \leq r_{\rho'}$, since r_ρ is active at t_ρ and $r_{\rho'}$ is active at the later time $t_{\rho'}$. It follows that $r_\rho = r_{\rho'} = r$ and r is active at time t. $\square$

An immediate consequence of Lemma 3.5 is:

Corollary 3.6. *Let ρ and σ be operations with assigned rounds r and s, and suppose that ρ finishes before σ starts. Then $r \leq s$.*

This gives us half of what we need for $<_S$ to be consistent with $<_H$, since if $r < s$ we have $\rho <_S \sigma$. For the other half, we must consider what happens when $r = s$.

We start with some observations about the timing of critical steps.

Lemma 3.7. *Let ρ be an operation and let c_ρ be its critical step. Then*

1. *If ρ is a* `MaxScan` *operation, c_ρ occurs during ρ's execution interval.*
2. *If ρ is a* `MaxUpdate` *operation, then*
   - (a) *c_ρ occurs after ρ starts, and*
   - (b) *If c_ρ occurs after ρ finishes, then ρ's round r_ρ is strictly less than the round of ρ's embedded* `MaxScan` *operation.*

Proof.

1. For a direct-scan `MaxScan`, this is immediate from the fact that c_ρ is an operation of ρ. For an indirect-scan `MaxScan`, this follows from Lemma 3.4.
2. Let ρ be a `MaxUpdate`(A, side, v) operation, let π be its embedded `MaxScan`, and let τ be the operation that carries out c_ρ. Note that τ may or may not be equal to π.
   - (a) Because τ observes ρ's write to $A.\mathsf{raw}[\mathsf{side}]$ before executing c_ρ, c_ρ occurs after ρ starts.
   - (b) Let r_τ and r_π be the rounds of τ and π. Since π observes ρ's write to $A.\mathsf{raw}[\mathsf{side}]$, and r_τ is the minimum round of all `MaxScan` operations that observe this write, $r_\tau \leq r_\pi$. If $r_\tau = r_\pi$, then τ writes to $A.\mathsf{low}[\mathsf{side}]$ no later than π does. It follows that in this case, c_ρ occurs before ρ finishes. So if c_ρ occurs after ρ finishes, it must hold that $r_\tau < r_\pi$.

□

Lemma 3.8. *Let ρ and σ be operations that are both assigned to round r, and suppose that ρ finishes before σ starts. Then $\rho <_S \sigma$.*

Proof. Since ρ and σ are both assigned the same round, we just need to show that ρ's critical step c_ρ precedes σ's critical step c_σ.

If ρ is a `MaxScan` operation, then from Lemma 3.7, c_ρ's occurs before ρ finishes, and c_σ occurs after σ starts. So $\rho <_S \sigma$.

Alternatively, if ρ is a `MaxUpdate`(A, side, v) operation, then either c_ρ occurs before ρ finishes, giving $\rho <_S \sigma$ as above; or c_ρ occurs after ρ finishes, but ρ's embedded `MaxScan` is assigned a round r' greater than r. But then from Lemma 3.5, $r' > r$ must be active before ρ finishes, so r cannot be active during the subsequent execution of σ, contradicting the assumption that σ is assigned round r. □

Combining Corollary 3.6 with Lemma 3.8 gives:

Lemma 3.9. *For any history H, the linearization order $<_S$ defined above is consistent with $<_H$.*

3.1.4 Correctness of `MaxScan` return values

We now show that the return values of all `MaxScan` operations are consistent with their positions in the linearization order. This means that when a `MaxScan` operation ρ returns a value v_ρ, then for each $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$, $v_\rho[\mathsf{side}]$ is equal to the largest value v written by a $\texttt{MaxUpdate}(A, \mathsf{side}, V)$ operation that precedes ρ in S, or 0 if there is no such operation. The first step is to show that reconstructing a complete value in Line 22 by combining a high part obtained from $A.\mathsf{raw}[\mathsf{side}]$ with a low part obtained from $A.\mathsf{low}[r][\mathsf{side}]$ for some r always yields a value originally read from $A.\mathsf{raw}[\mathsf{side}]$.

Lemma 3.10. *Let ρ be a direct-scan* `MaxScan` *operation with round r, and let v_ρ be the value that it returns in Line 22. Fix some $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$. Consider the set of* `MaxScan` *operations S that write $A.\mathsf{low}[r][\mathsf{side}]$ before ρ reads $A[\mathsf{low}][r]$. Note that this set is nonempty, because it includes ρ. For each $\pi \in S$, let z_π be the last tuple read from $A.\mathsf{raw}$ by π. Then $v_\rho[\mathsf{side}] = \max(z_\pi[\mathsf{side}])$.*

Proof. That $\texttt{low}(v_\rho[\mathsf{side}]) = \max(\texttt{low}(z_\pi))$ is immediate from the choice of S and the fact that $A.\mathsf{low}[r]$ is a max array. But from Corollary 3.3, we have that for every $\pi \in S$ $\texttt{high}(z_\pi[\mathsf{side}]) = \texttt{high}(v_\rho[\mathsf{side}])$. So whichever $\pi \in S$ maximizes $\texttt{low}(z_\pi[\mathsf{side}])$ also maximizes $z_\pi[\mathsf{side}] = \texttt{combine}(\texttt{high}(z_\pi[\mathsf{side}]), \texttt{low}(z_\pi[\mathsf{side}])) = \texttt{combine}(\texttt{high}(v_\rho[\mathsf{side}]), \texttt{low}(v_\rho[\mathsf{side}])) = v_\rho[\mathsf{side}]$. $\square$

It is worth noting that some such π always exists, because even if no other process previously wrote to $A.\mathsf{low}[r][\mathsf{side}]$, ρ will have done so.

Lemma 3.11. *Given a history H, every* `MaxScan` *returns the correct value for the linearized sequential execution S.*

Proof. We consider only direct-scan `MaxScan` operations, because for any indirect-scan `MaxScan` operation τ, there is a direct-scan `MaxScan` operation ρ that returns the same value as τ and linearizes in the same position relative to all `MaxUpdate` operations. So if ρ returns a correct value, so does τ.

Now consider some direct-scan `MaxScan` ρ. For each $\mathsf{side} \in \{\mathsf{left}, \mathsf{right}\}$, Lemma 3.10 says that $v_\rho[\mathsf{side}]$ is equal to $\max(z_\pi[\mathsf{side}])$, where π ranges over all `MaxScan` operations π that write $A.\mathsf{low}[\mathsf{side}]$ before ρ reads $A.\mathsf{low}$. Each such z_π is in turn read form $A.\mathsf{raw}[\mathsf{side}]$, and is equal to the maximum of all values v_σ written by `MaxUpdate` operations σ that write $A.\mathsf{raw}[\mathsf{side}]$ before π reads it for the last time. But any such operation σ linearizes before ρ, because its critical step is either π's write to $A.\mathsf{low}[r][\mathsf{side}]$, or some earlier write to $A.\mathsf{low}$. Conversely, if σ does not linearize before ρ, then there is no `MaxScan` operation π that reads $A.\mathsf{raw}[\mathsf{side}]$ after σ writes it and then writes $A.\mathsf{low}[r]$ before ρ reads $A.\mathsf{low}$.

So v_σ is included in the maximum if and only if σ linearizes before ρ, and ρ correctly returns as $v_\rho[\mathsf{side}]$ the maximum of all previous `MaxUpdate`(A, side, v) operations. Since this holds for both values of side, v_ρ is correct. ☐

Combining Lemma 3.9 and 3.11 gives:

Theorem 3.12. *Algorithms 3.1 and 3.2 implement a linearizable max array.*

3.2 Step complexity

We now turn to performance. As with Algorithm 2, we assume k-bounded increments, where $k = n^d$ for some d. Using this assumption, and with appropriate tuning of the parameters of Algorithm 2, the expected step complexity of `ReadMax` on `WriteMax` operations on $A.\mathsf{raw}[\mathsf{left}]$ $A.\mathsf{raw}[\mathsf{right}]$ will be $O(\log n)$ by Theorem 2.3.

Because Algorithm 3.2 carries out one `WriteMax` operation, a `MaxScan` operation, and a constant number of register operations, its expected step complexity is is $O(\log n + T(n))$, where $T(n)$ is the expected step complexity of the `MaxScan` operation implemented in Algorithm 3.1. We thus concentrate on showing a bound on the expected cost of Algorithm 3.1.

The essential idea is to show that no process p will execute the loop in Algorithm 3.1 too many times on average before it gets a usable value out of $A.\mathsf{help}[p]$. We will need the following lemma:

Lemma 3.13. *Consider an execution of Algorithms 3.1 and 3.2, and let t be any time during this execution. Let t' be a later time such that at least n^4* `MaxUpdate` *operations start and finish in the open interval (t, t'). Then for any p, there are at least $n^2 - n$ locations i in $A.\mathsf{help}[p]$ such that for all times $t'' \geq t$, $A.\mathsf{help}[p][i]$ contains a value written by a* `MaxUpdate` *operation that starts after t.*

Proof. By the Pigeonhole Principle, there is some process q that starts and finishes at least n^3 `MaxUpdate` operations between t and t'. The first n^3 of these operations write to $A.\mathsf{helpPos}[p][i]$ for all p and i. It follows that at time t', for each p, all n^2 positions in $A.\mathsf{help}[p]$ have been written to by at least one `MaxUpdate` operation that starts after t.

These values may be overwritten by other `MaxUpdate` operations. If they are overwritten by other `MaxUpdate` operations that start after time t, the claim in the lemma is unaffected. So we need only worry about `MaxUpdate` operations that are still in progress at time t.

There are at most n such operations, and because each `MaxUpdate` chooses which register in $A.\mathsf{help}$ to write to deterministically, there are at most n registers in all of $A.\mathsf{help}$ that can be written to by one of these pending `MaxUpdate` operations. This leaves $n^2 - n$ registers that contain for all $t'' \geq t'$ a value written by a `MaxUpdate` operation that starts after t. ☐

Theorem 3.14. *For $k = n^d$ and $m = n^4 k$, the step complexity of operations in Algorithms 3.1 and 3.2 is $O(\log^2 n)$ with high probability against an adaptive adversary, provided increments are k-bounded.*

Proof. As observed above, the main issue is bounding the step complexity of `MaxScan`.

Fix some instance ρ of `MaxScan`, and let p the process that carries out ρ. Each iteration of the loop in `MaxScan` involves $O(1)$ register operations, at most two `ReadMax` operations on each of $A.\mathsf{raw}[\mathsf{left}]$ and $A.\mathsf{raw}[\mathsf{right}]$, and at most three operations on $A.\mathsf{low}[r]$ for some r. Given the bounded-increment assumption, Theorem 2.3 says that the operations on $A.\mathsf{raw}$ take $O(\log n)$ time with high probability. The operations on $A.\mathsf{low}[r]$ take $O(\log^2 n)$ time in the worst case, since $A.\mathsf{low}$ is an m-bounded max array with m polynomial in n. This gives a total step complexity for each iteration of $O(\log^2 n)$ with high probability. If we can show that we carry out only $O(1)$ iterations with high probability, we are done.

In order to finish an iteration without returning a value, ρ must not find a current timestamp in $A.\mathsf{help}[p][\mathtt{random}()]$, and must either have $A.\mathsf{filled}[p]$ be out of date or see two different values in the collects stored in y and z. We will show both that $A.\mathsf{filled}[p]$ quickly catches up with any changes to $A.\mathsf{raw}$ that happened before ρ started, and that if $A.\mathsf{raw}$ changes too often after ρ starts, $A.\mathsf{help}[p]$ will contain enough values that are current for ρ that ρ will find a current timestamp in $A.\mathsf{help}[p][\mathtt{random}()]$ after $O(1)$ iterations with high probability.

First let us show that $A.\mathsf{raw}$ can't get too far ahead without ρ being able to finish with high probability from an indirect scan. Let t be the time at which ρ updates $A.\mathsf{TS}[p]$ in Line 2. Let $r_t = \mathtt{high}(A^t.\mathsf{raw}[\mathsf{left}]) +$

$\texttt{high}(A^t.\mathsf{raw}[\mathsf{right}])$ be the round corresponding to the values in $A.\mathsf{raw}$ at time t. Let u be the earliest time at which $r_u = \texttt{high}(A^u.\mathsf{raw}[\mathsf{left}]) + \texttt{high}(A^u.\mathsf{raw}[\mathsf{right}])$ is at least $r_t + 5$. Then there is a side side such that $\texttt{high}(A^u.\mathsf{raw}[\mathsf{side}]) \geq \texttt{high}(A^t.\mathsf{raw}[\mathsf{side}]) + 3$. Expanding the definition $\texttt{high}(x) = \lfloor x/m \rfloor$ then shows that $A^u.\mathsf{raw}[\mathsf{side}] \geq A^t.\mathsf{raw}[\mathsf{side}] + 3m$ and thus $A^u.\mathsf{raw}[\mathsf{side}] \geq A^t.\mathsf{raw}[\mathsf{side}] + 2n^4k$. Under the assumption of k-bounded increments, this implies that at least $2n^4$ `MaxUpdate` operations finish in the interval (t, u), and so at least $2n^4 - n \geq n^4$ `MaxUpdate` operations start and finish during (t, u). From Lemma 3.13, there are at least $n^2 - n$ locations i in $A.\mathsf{help}[p]$ such that from time u on, $A.\mathsf{help}[p]$ contains a value v written by a `MaxUpdate` operation that started after ρ incremented $A.\mathsf{TS}[p]$. It follows that with probability at least $1 - 1/n$, any iteration of ρ's loop that starts after u will find a current timestamp in $A.\mathsf{help}[p][\texttt{random}()]$ and return. This implies that the probability that ρ starts more than c iterations of the loop after u is at most n^{-c}, for any fixed c, giving that ρ executes $O(1)$ iterations after u with high probability.

Now we will show that $A.\mathsf{filled}[p]$ catches up quickly. Let s be the earliest time at which $r_s = \texttt{high}(A^s.\mathsf{raw}[\mathsf{left}]) + \texttt{high}(A^s.\mathsf{raw}[\mathsf{right}]) \geq r_t - 5$. Repeating the same argument as above with the interval (s, t) in place of (t, u) shows that there are $n^2 - n$ locations i such that from time t on, each $A.\mathsf{help}[p][i].\mathsf{helpVal} \geq A^s.\mathsf{raw}$. So with high probability, ρ executes at most $O(1)$ iterations before seeing such a value and updating $A.\mathsf{filled}[p]$ to be at least $r_t - 5$ in Line 8. After an additional $5 = O(1)$ iterations, $A.\mathsf{filled}[p]$ will be at least r_t. Let t' be the time at which $A^{t'}.\mathsf{filled}[p]$ first exceeds r_t.

In order for an iteration that starts after t' not to return, either $A.\mathsf{filled}[p]$ must become out of date, indicating that $\texttt{high}(A.\mathsf{raw}[\mathsf{left}]) + \texttt{high}(A.\mathsf{raw}[\mathsf{right}])$ increased; or ρ must see two different collects in y and z, also indicating that $\texttt{high}(A.\mathsf{raw}[\mathsf{left}]) + \texttt{high}(A.\mathsf{raw}[\mathsf{right}])$ increased. But $\texttt{high}(A.\mathsf{raw}[\mathsf{left}]) + \texttt{high}(A.\mathsf{raw}[\mathsf{right}])$ can increase at most 5 times before we reach time u. It follows that there are at most 5 iterations of ρ that start in (t', u).

Adding the $O(1)$ iterations that occur with high probability in each of the intervals $[0, t']$, (t', u), and $[u, \infty)$ gives $O(1)$ total iterations with high probability. Since each iteration takes $O(\log^2 n)$ steps with high probability, we get a total cost of $O(\log^2 n)$ steps with high probability. $\square$

4 Unlimited-use snapshots

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

We use the construction from [4], which for convenience we restate here in Algorithm 4.

The shared data is:

- leaf_j, for $j \in \{0, \ldots, n-1\}$: the leaf node corresponding to process p_j, with fields:
  - parent: the parent of this leaf in the tree
  - $\mathsf{view}[0, 1, \ldots]$: an infinite array, each of whose entries contains a partial snapshot, $\mathsf{view}[0]$ contains the initial value of component j and $\mathsf{view}[\ell]$ contains the ℓ-th value of component j
  - root: the root of the tree
- Each internal node has the fields:
  - left: the left child of the node in the tree
  - right: the right child of the node in the tree
  - $\mathsf{view}[0, 1, \ldots]$: an infinite array, each of whose entries contains a partial snapshot, $\mathsf{view}[0]$ contains the concatenation of $\mathsf{leaf}_j.\mathsf{view}[0]$ for all leaves leaf_j in the subtree rooted at this node, and $\mathsf{view}[\ell]$ contains the concatenation of views of the leaves after ℓ updates
  - ma: an infinite MaxArray object, initially (0,0)
- The root also has the field mr: an infinite MaxRegister object, initially 0
- Each non-root internal node also has the field parent: the parent of the node in the tree

We use this algorithm with our implementations of unbounded-value max registers and unbounded-value max arrays from the previous sections. Loosely speaking, 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 that updates its location does so by

```
procedure Update_i(s, v)
    count_i ← count_i + 1
    u ← leaf_i
    ptr ← count_i
    u.view[ptr] ← v
    while u ≠ root do
        if u = u.parent.left then
            MaxUpdate0_i(u.parent.ma, ptr)
        if u = u.parent.right then
            MaxUpdate1_i(u.parent.ma, ptr)
        u ← u.parent
        (lptr, rptr) ← MaxScan_i(u.ma)
        lview ← u.left.view[lptr]
        rview ← u.right.view[rptr]
        ptr ← lptr + rptr
        u.view[ptr] ← lview · rview // lview and rview are concatenated
    WrapWriteMax_i(root.mr, ptr)
procedure Scan(s)
    ptr ← WrapReadMax_i(root.mr)
    return root.view[ptr]
```

Algorithm 4: Unlimited-use single-writer snapshot object; code for process p_i.

updating the nodes from its leaf to the root, and a process scans the object by reading the view held by the root. We emphasize that correctness is always guaranteed in the above implementation, therefore the proof from [4] shows that this gives an unlimited-use snapshot object.

To analyze the step complexity of our construction, we show that the n-bounded increment assumption holds and use the complexity analysis of the previous sections. Intuitively, the assumption holds because every MaxRegister is used only to store the number of operations observed in the subtree of processes that it represents. Denote by v an input value of a WrapWriteMax operation *op*, and by v' the largest input to any previously initiated WrapWriteMax operation, *op*′. We show that $v - v'$ cannot be more than n, since this would imply that some process initiates an operation *op*″ between these two WrapWriteMax operations. Since *op*″ is initiated after *op*′, it observes at least $v' + 1$ operations and hence it calls WrapWriteMax with input $v'' > v'$, contradicting the definition of v'. Formally, we prove this claim in the following lemma.

Lemma 4.1. *In Algorithm 4, all* MaxRegister *and* MaxArray *objects are accessed according to the n-bounded increments assumption.*

Proof. A process that performs `MaxUpdate0` or `MaxUpdate1` on u.ma for some node u writes the value of its ptr variable. We show that ptr holds a value which is at most the number of `Update` operations invoked by processes corresponding to this subtree, and at least the number of such operations that have finished. This implies that a value being written to u.ma is larger by at most n than the largest value previously written to it, as there can be at most one operation by each process that has started but not yet finished. The claim follows by a simple induction on the height of the node that holds the object. When accessing a leaf, ptr holds the value of count_i, which is the number of operations performed by process p_i. For an intermediate node u, ptr holds the sum of the values of its two children, which, by the induction hypothesis is at most the number of `Update` operations invoked by processes corresponding to these subtrees, and at least the number of such operations that have finished, which proves the claim. Finally, the same holds for the value of ptr when the root is accessed, implying the claim also for the MaxRegister object there. □

Combining Lemma 4.1 with Theorem 3.14 gives our main theorem.

Theorem 4.2. *Algorithm 4 is an implementation of an unlimited-use single-writer snapshot object, with a step complexity of $O(\log^3 n)$ per operation with high probability.*

Proof. Plugging the n-bounded increments property given by Lemma 4.1 into Theorem 2.3 and Theorem 3.14, gives that each operation on a max register takes $O(\log n)$ steps and each operation on a 2-component max array takes $O(\log^2 n)$ steps, with high probability. Therefore, a `Scan` operation takes $O(\log n)$ steps with high probability, because it performs one `ReadMax` operation and reads a single location in the array of views root.view. An `Update` operation accesses at most $O(\log n)$ max registers and 2-component max arrays, $O(1)$ at each level of the tree, hence takes $O(\log^3 n)$ steps, with high probability (since the high probabilities of the step complexities of the max register and the 2-component max array are with an exponent larger than 1). □

5 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 classic Attiya-Bar-Noy-Dolev (ABD) register simulation [8] 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 maximal 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 max register 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 max register reads or collects to be implemented at no greater cost than the $O(1)$ time and $O(n)$ messages needed for a simple 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 procedures `MaxUpdate` and `MaxScan` can be simplified by eliminating the `pointer` array in favor of performing collects on the `help` array directly; this eliminates the need for randomization. Substituting these new implementations into Algorithm 4 gives the full result.

5.1 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 atomic register simulation of Attiya, Bar-Noy, and Dolev [8].

Pseudocode is given in Algorithm 5. Each process stores a local value `maxValue` for the max register. The `WriteMax` and `ReadMax` 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 `WriteMax`, 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 in fact stored in a majority of the processes. We denote by $\perp$ a value that is not larger than any integer value v.

```
Persistent Local Data: maxValue, initially $\bot$; $t$, initially 0
upon receiving Update$(t, v)$ from $j$ do
    maxValue $\leftarrow \max($maxValue$, v)$
    Send respond$(t,$ maxValue$)$ to $j$
procedure Update$(v)$
    $t \leftarrow t + 1$
    Send Update$(t, \bot)$ to all processes.
    Wait to receive respond$(t, v_i)$ from a set $S$ of a majority of
      processes $p_i$.
    Let $v' = \max(v, \max_{i \in S}(v_i))$.
    $t \leftarrow t + 1$
    Send Update$(t, v')$ to all processes.
    Wait to receive respond$(t, -)$ from a majority of processes.
    return $v'$.
procedure WriteMax$(v)$
    Update$(v)$
procedure ReadMax()
    return Update$(\bot)$
```

Algorithm 5: Max register in message passing using ABD.

Theorem 5.1. *Algorithm 5 gives a linearizable deterministic message-passing implementation of a max register.*

Proof. Given an execution of Algorithm 5, 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 value of v'. Finally, we put `WriteMax` operations before `ReadMax` 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`$(t, \bot)$ 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 `ReadMax` operations, it is enough to show that these values are equal to the largest input to any previous `WriteMax` 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 $\mathsf{maxValue}_p$ in some process p, or appears as the value in an `Update` or `respond` message, appears as the input to some `WriteMax` 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 `WriteMax` broadcasts its second `Update` message, and such values will either be the input to the `WriteMax` or a value that previously appeared in a `respond` message to the process carrying out the `WriteMax` operation. It follows that any value returned by a `ReadMax` corresponds to some value written in a `WriteMax`.

The computation in Line 8 ensures that any `WriteMax`(v) operation W computes $v'_W \geq v$; in the other direction, any `ReadMax` operation R returns v'_R. So for any given `ReadMax` operation R, only `WriteMax` operations with input less than or equal to v'_R can be linearized before R. From the invariant, there exists at least one `WriteMax` 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 `WriteMax` 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 `WriteMax` operation 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 to be returned as v'_R by the `ReadMax` operation R in Line 8 (since any $\mathsf{maxValue}$ 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 `WriteMax` operation linearized before R, and the sequential specification is satisfied. $\square$

5.2 Message-passing 2-component max arrays

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 is current at some time between the start and finish of the collect.

The trivial 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 correct on an unlimited number of registers with the same time and message complexity bounds.

Using the message-passing max register of the previous section and this message-passing collect, we can rewrite the `MaxUpdate` and `MaxScan` procedures from from Algorithms 3.1 and 3.2 by replacing every instance of a max array operation with an $O(1)$-time $O(n)$-message implementation, and the random probe of a single help array location in Line 4 in Algorithm 3.1 with a collect of the entirety of $A.\mathsf{help}[p]$, returning any value in $A.\mathsf{help}[p]$ that is current.

The proof of correctness carries over trivially to the revised algorithm; the only change in its behavior is that once at least one current value for p appears in $A.\mathsf{help}[p]$, p finds it immediately instead of relying on chance. So the $O(\log^2 n)$ high-probability bound on step complexity maps to an $O(\log n)$ deterministic worst-case bound on time and $O(n \log n)$ worst-case bound on message complexity for each max array operation, under the assumptions of Theorem 3.14. We summarize these results in the following theorem.

Theorem 5.2. *The message-passing implementation of Algorithms 3.1 and 3.2, with $k = n^d$ and $m = n^4 k$, completes each operation in $O(\log n)$ time and $O(n \log n)$ messages always, provided increments are k-bounded and fewer than $n/2$ processes crash.*

5.3 The full snapshot algorithm

To complete the algorithm, implement Algorithm 4 using the max arrays from Section 5.2. We then have:

Theorem 5.3. *Using message-passing max arrays, Algorithm 4 is an 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.*

6 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. 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.* [11], because the adversary cannot predict what locations a process will read from the helper array and thus cannot guarantee to cover those locations with old values. Conversely, the 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 the entire helper array 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 the conference version of this paper and the anonymous reviewers of the journal for careful comments and suggestions that greatly helped improve the presentation of this work.

References

[1] 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.

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

[3] 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.

[4] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Faster than optimal snapshots (for a while). In *2012 ACM Symposium on Principles of Distributed Computing*, pages 375–384, July 2012.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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