Wait-Free Data Structures in the Asynchronous PRAM Model

James Aspnes *
Department of Computer Science
Yale University
New Haven, CT 06520-8285

Maurice Herlihy†
Computer Science Department
Brown University
Providence, RI 02912

December 31, 2000

Abstract

In the asynchronous PRAM model, processes communicate by atomically reading and writing shared memory locations. This paper investigates the extent to which asynchronous PRAM permits long-lived, highly concurrent data structures. An implementation of a concurrent object is *wait-free* if every operation will complete in a finite number of steps, and it is *k-bounded wait-free*, for some $k > 0$, if every operation will complete within k steps. In the first part of this paper, we show that there are objects with wait-free implementations but no k-bounded wait-free implementations for any k, and that there is an infinite hierarchy of objects with implementations that are k-bounded wait-free but not K-bounded wait-free for some $K > k$. In the second part of the paper, we give an algebraic characterization of a large class of objects that do have wait-free implementations in asynchronous PRAM, as well as a general algorithm for implementing them. Our tools include simple iterative algorithms for wait-free approximate agreement and atomic snapshot.[1]

*During much of the work described in this paper, J. Aspnes was funded by an IBM graduate fellowship.

†Much of the work described in this paper was completed while M. Herlihy was at DEC Cambridge Research Laboratory

[1]A preliminary version of this paper appeared in the *Proceedings of the Second Annual*

1 Introduction

In the "classical" parallel random access machine (PRAM) model [20], a set of processes executing in lock-step communicate by applying read and write operations to a shared memory. Existing shared memory architectures, however, are inherently *asynchronous:* processors' relative speeds are unpredictable, at least in the short term, because of timing uncertainties introduced by variations in instruction complexity, page faults, cache misses, and operating system activities such as preemption or swapping. A number of researchers have noted this mismatch, and have proposed the *asynchronous PRAM* model as an alternative [16, 17, 21, 41]. In this model, asynchronous processes communicate by applying atomic read and write operations to the shared memory [2]. Techniques for implementing these memory locations, often called *atomic registers*, have also received considerable attention [13, 14, 32, 35, 40, 43, 44].

Much of the work on asynchronous PRAM models addresses the problem of computing functions, such as parallel summation or substring matching, whose inputs reside in the shared memory. Many practical applications, however, such as operating systems and data bases, are not organized around functional computation. Instead, they are organized around long-lived *data objects* such as sets, queues, directories, and so on. In this paper, we investigate the extent to which the asynchronous PRAM model supports long-lived, highly-concurrent data objects. There are several reasons why long-lived objects are inherently more difficult than functional computation. A data object has an unbounded lifetime during which each process can execute an arbitrary dynamically-chosen sequence of operations. The that data structures must be reused, but they must retain enough information to ensure that "sleepy" processes that arbitrarily suspend and resume execution can continue to progress, while discarding enough information to keep the object size bounded. Care must be taken to guard against starvation, since one operation can be "overtaken" by an arbitrary sequence of other operations.

An implementation of a concurrent object is *wait-free* if every process must complete an operation after taking a finite number of steps, despite failures of other processes. It is *k-bounded wait-free*, for some fixed $k > 0$, if every process completes an operation after taking k steps. The wait-free

ACM Symposium on Parallel Architectures and Algorithms, Crete, Greece, July 1990 [7], and in the *Proceedings of the Third Annual ACM Symposium on Parallel Architectures and Algorithms*, Hilton Head, North Carolina, July 1991 [25].

[2] Some of these models also include primitives for barrier synchronization.

property excludes starvation: any process that continues to take steps will finish its operation. The bounded wait-free property bounds how long it will take. Either of these properties rules out conventional synchronization techniques such as barrier synchronization, busy-waiting, conditional waiting, or critical sections, since the failure or delay of a single process within a critical section or before a barrier will prevent the non-faulty processes from making progress.

Which objects have wait-free implementations in asynchronous PRAM? Elsewhere [23, 26], we have shown that any object X that solves consensus for two or more processes cannot be implemented without randomization in a model that provides only simple reads and writes of shared memory. Thus the asynchronous PRAM model does not permit deterministic implementations of common data types such as sets, queues, stacks, priority queues, or lists, most if not all the classical synchronization primitives, such as *test&set*, *compare&swap*, and *fetch&add*, and simple memory-to-memory operations such as *move* or *swap*.

In the first part of the paper, we give some additional impossibility results for the asynchronous PRAM model. Given that one cannot construct a wait-free implementation of any object that solves two-process consensus, it is natural to ask whether the converse holds: does asynchronous PRAM permit wait-free implementations of all remaining objects, i.e., those that do *not* solve two-process consensus? In this paper, we show that the answer is *no*. In a system of two processes, we demonstrate the existence of a strict infinite hierarchy among objects that are still too weak to solve consensus:

- Objects with implementations that are wait-free, but not bounded wait-free. Each operation requires a finite number of steps, but there is no bound common to all operations. (Theorem 8.)

- For all $k > 0$, objects with implementations that are K-bounded wait-free for some $K > k$, but not k-bounded wait-free. (Theorem 7.)

In the second part of this paper, we give a new characterization of a wide class of objects that *do* have wait-free implementations in the asynchronous PRAM model. This characterization is algebraic in nature, in the sense that it is expressed in terms of simple commutativity and overwriting properties of the data type's sequential specification. We present a technique for transforming a sequential object implementation into an n-process wait-free implementation requiring a worst-case synchronization overhead of $O(n^2)$ reads and writes per operation. Examples of objects that can be

implemented in this way include counters, logical clocks [33], and certain kinds of set abstractions.

One contribution of this paper is the impossibility hierarchy, which shows that even relatively "weak" concurrent objects have a rich mathematical structure. A second contribution is the characterization of a large class of constructible objects, implying that despite the weakness of the model, certain problems do have wait-free solutions. Perhaps the most general contribution is to raise basic questions about the value of the asynchronous PRAM model. Although some synchronous PRAM algorithms can be adapted to asynchronous PRAM [16, 17, 21, 41], our results show that there is little hope of constructing useful highly-concurrent long-lived data structures in this model. Fortunately, however, one can argue that asynchronous PRAM is an incomplete reflection of current practice. Starting with the IBM System/370 architecture [30] in the early 1970's, nearly every major architecture has provided some kind of atomic read-modify-write primitive. We have shown elsewhere that one can construct a bounded wait-free implementation of any object by augmenting the read and write operations with sufficiently powerful read-modify-write primitives, such as *compare&swap* [24]. It is not our intent here to suggest a specific alternative model, but we do believe that the research community would benefit from a more realistic and powerful model of concurrent shared-memory computation.

2 Related Work

Although the work on atomic registers has a long history, it is only recently that researchers have attempted to formalize the computational power of the resulting model. Cole and Zajicek [16, 17] and Nishimura [41] propose complexity measures and basic algorithms for an "asynchronous PRAM" model in which asynchronous processes communicate through shared atomic registers. Gibbons [21] proposes an asynchronous model in which shared atomic registers are augmented by a form of barrier synchronization. Our work extends these approaches in two ways: we consider data structures rather than the usual numeric or graph algorithms, and we focus on wait-free computation, since we feel that algorithms that require processes to wait for one another are poorly suited to asynchronous models.

Recently, a number of researchers have investigated the problem of compiling classical PRAM programs onto other models [31, 15], some still synchronous, some not. It must be emphasized that these researchers are addressing very different kinds of applications. Their programs have the prop-

erty that all the program's arguments are present in public memory when the program starts. Because new information does not arrive dynamically, these programs are not subject to the consensus-related impossibility results that define the computational power of concurrent objects in this model. This assumption is a legitimate one for "off-line" applications such as scientific computation, but not for reactive systems such as operating systems, file systems, databases, and any other kind of long-lived system.

Two other atomic scan algorithms were developed independently of the one presented here: by Afek et al. [2] and by Anderson [4]. The former has time complexity comparable to ours, while the latter uses time exponential in the number of processes. Both of these proposals use bounded counters, while the most straightforward implementation of our scan algorithm uses unbounded counters to represent lattice elements.

Anderson [5] gives a bounded implementation of *pseudo read-modify-write* instructions in asynchronous PRAM. Let F be a set of functions that commute with one another. A pseudo read-modify-write instruction is parameterized by a function f from F. When applied to a memory location holding a value v, it replaces the contents with $f(v)$, but does not return a value. This construction uses bounded counters, unlike our construction, but it does not permit overwriting operations.

An object implementation is *randomized wait-free* if each operation completes in a *fixed* expected number of steps. Elsewhere [6], we have shown that the asynchronous PRAM model is universal for randomized wait-free objects.

Our approximate agreement algorithm and lower bounds give similar asymptotic results to the independent work of Attiya, Lynch, and Shavit [9]. Hoest and Shavit [28] have recently shown that, when translated to an iterated snapshot model, the constant factors in our results are the best possible.

Since the first appearance of the preliminary versions of this paper [7, 25], there have been many advances in the study of wait-free objects built from atomic registers. In particular, there has been considerable improvement in algorithms for atomic snapshots. The *lattice agreement* technique [8], where processes agree on a chain in a lattice, is closely related to the semi-lattice construction we use in Section 6. By allowing processes to obtain values spread throughout the lattice instead of pushing all processes toward the top, lattice agreement allows for faster snapshot protocols such as the asymptotically optimal $O(n \log n)$ protocol of Attiya and Rachman [10].

3 Model

Informally, our model of computation consists of a collection of sequential threads of control called *processes* that communicate through shared data structures called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the only means to manipulate that object. Each process applies a sequence of operations to objects, issuing an invocation and receiving the associated response. The basic correctness condition for concurrent systems is *linearizability* [27]: although operations of concurrent processes may overlap, each operation appears to take effect instantaneously at some point between its invocation and response. In particular, operations that do not overlap take effect in their "real-time" order.

3.1 Automata

A complete formal exposition of our model is given elsewhere [26]. Here we give an abbreviated version, omitting some technical details for brevity. We model objects as automata, using a simplified form of I/O automata formalism of Lynch and Tuttle [38]. Because the wait-free condition does not require any fairness or liveness conditions, and because we consider only finite sets of processes and objects, we do not make use of the full power of the I/O automata formalism. (For brevity, our algorithms are expressed using pseudocode, although it is straightforward to translate this notation into automata definitions.)

An *automaton* A is a non-deterministic automaton with the following components[3]: $States(A)$ is a finite or infinite set of states, including a distinguished set of starting states, $In(A)$ is a set of *input events*, $Out(A)$ is a set of *output events*, $Steps(A)$ is a transition relation given by a set of triples (s', e, s), where s and s' are states and e is an event. Such a triple is called a *step*, and it means that an automaton in state s' can undergo a transition to state s, and that transition is associated with the event e. An *execution fragment* of an automaton A is a finite sequence $s_0, e_1, s_1, \ldots e_n, s_n$ or infinite sequence $s_0, e_1, s_1, \ldots$ of alternating states and events such that each (s_i, e_{i+1}, s_{i+1}) is a step of A. An *execution* is an execution fragment where s_0 is a starting state. A *history fragment* of an automaton is the subsequence of events occurring in an execution fragment, and a *history* is the

[3] To remain consistent with the terminology of [27], we use "event" where Lynch and Tuttle use "operation," and "history" where they use "schedule."

subsequence occurring in an execution. A *prefix* of an execution or history is a prefix of the sequence that is itself an execution or history, respectively.

3.2 Linearizable Objects

An *object* is an automaton with input events INVOKE(P,op) where P is a process and op is an operation of the object [4], and output events RESPOND(P,res), where res is a result value. We refer to these events as *invocations* and *responses*. If H is a history, then $H|P$ is the subsequence of invocations and responses labeled with P. Two invocations and responses *match* if their process names agree. To capture the notion that a process represents a single thread of control, we say that a process history is *well-formed* if it begins with an invocation and alternates matching invocations and responses. An invocation is *pending* if it is not followed by a matching response. A history is *well-formed* if, for each process P, $H|P$ is well-formed. We restrict our attention to well-formed histories.

An execution is *sequential* if its first event is an invocation, and it alternates matching invocations and responses. A history is sequential if it is derived from a sequential execution. (Notice that a sequential execution permits process steps to be interleaved, but at the granularity of complete operations.) If we restrict our attention to sequential histories, then the behavior of an object can be specified in a particularly simple way: by giving pre- and postconditions for each operation. We refer to such a specification as a *sequential specification*. In this paper, we consider only objects whose sequential specifications are *total* and *deterministic*: if the object has a pending invocation, then it has a unique matching enabled response. We consider only total operations because it is unclear how to interpret the wait-free condition for partial operations. For example, the most natural way to define the effects of a partial *dequeue* operation of a shared queue in a concurrent system is to have it wait until the queue becomes non-empty, a specification that clearly does not admit a wait-free implementation. We consider only deterministic operations because one can always use a deterministic implementation to satisfy a non-deterministic specification, e.g., using the *dequeue* operation for queues to implement a non-deterministic *choose* operation for sets.

If H is a history, let $complete(H)$ denote the maximal subsequence of H consisting only of invocations and matching responses. Each history H induces a partial "real-time" order $\prec_H$ on its operations: $p \prec_H q$ if the

[4] *Op* may also include argument values.

response for p precedes the invocation for q. Operations unrelated by $\prec_H$ are said to be *concurrent*. If H is sequential, $\prec_H$ is a total order. An object is *linearizable* if each history H can be extended to a well-formed history H', by adding zero or more responses, such that there exists a sequential history S such that:

- For all P, $complete(H')|P = S|P$
- $\prec_H \subseteq \prec_S$

In other words, the history "appears" sequential to each individual process, and this apparent sequential interleaving respects the real-time precedence ordering of operations. Equivalently, each operation appears to take effect instantaneously at some point between its invocation and its response. A linearizable object is thus "equivalent" to a sequential object, and its operations can also be specified by simple pre- and postconditions.

As discussed in more detail elsewhere [27], linearizability differs from related correctness conditions such as sequential consistency [34] or strict serializability [42] because it is a *local* property: a set of objects is linearizable if and only if each individual object is linearizable.

3.3 Implementations

An implementation of an object is itself an automaton composed from a collection of smaller automata. When a process invokes an operation, that invocation is handled by a *front-end* automaton, which applies a sequence of operations to a *representation* automaton, and eventually returns a response to the process. The front-end models the procedure that implements the operation, and the representation models shared memory in the obvious way: it accepts *read* and *write* invocations, and its state is just the Cartesian Product of each of the registers.

An implementation is *k-bounded* wait-free if each invocation returns provided its front-end takes at least k steps, it is *wait-free* if each pending invocation eventually returns provided its front-end continues to take steps.

4 A Wait-Free Hierarchy

In this section, we construct a family of objects with the property that, for all k, there exists an object whose implementations are K-bounded wait-free but not k-bounded wait-free, for some $K > k$. There also exists an object whose implementations are wait-free but not k-bounded wait-free for any k.

We prove the lower bounds by reducing the (difficult) problem of analyzing all possible implementations of a particular object to the (more tractable) problem of analyzing solutions to a related decision problem.

If S is a nonempty set of real numbers, let $range(S) = [\min(S), \max(S)]$, $midpoint(S) = (\min(S) + \max(S))/2$, and let $|S| = \max(S) - \min(S)$. For the empty set, define $range(\emptyset) = \emptyset$ and $|\emptyset| = 0$.

An *approximate agreement object* provides two operations:

```
input(P: process, x: real)
output(P: process) returns (real)
```

A sequential specification for these operations, expressed in terms of pre- and post-conditions, appears in Figure 1. The object's abstract state has two components: a set of real *input values* X and a set of real *output values* Y, initially both empty. In postconditions, X' and Y' denote the components' new states. The *input* operation inserts its argument value in X. The *output* operation is defined only when X is non-empty. It inserts its result in Y, ensuring that $range(Y) \subseteq range(X)$ and $|Y| < \epsilon$ for some fixed $\epsilon > 0$. For brevity, we leave unspecified how *output* behaves when X is empty.

As a decision problem, approximate agreement has been studied in a variety of message-passing models [12, 18, 19, 39]. Attiya, Lynch, and Shavit [9] independently derive upper and lower bounds for approximate agreement in shared memory that can be adapted obtain asymptotic bounds similar to those given here. Their approximate agreement algorithm is optimized for a best-case model where processes run approximately synchronously, and so involves some additional machinery that exploits the efficiencies possible in this model. In contrast, our algorithm is relatively simple, but does not perform as well for best-case executions.

A wait-free implementation of an approximate agreement object appears in Figure 2. The object is represented by an n-element array `r` of *entries*, where each entry has two fields: an integer `round` initially zero, and a real `prefer`, initially $\perp$. A process is a *leader* if its `round` field is greater than or equal to any other process's `round` field. P *advances* its entry by setting its preference to the midpoint of the leaders' preferences (line 16) and by incrementing its `round` field by one. P *scans* the entries by reading them in an arbitrary order.

The first time P calls *input*, it sets `prefer` to its input value. Subsequent calls have no effect. When P calls *output*, it returns the results of executing a wait-free approximate agreement protocol. This protocol consists of a loop in which P scans the entries (line 10), and discards those whose `round` fields trail its own by two or more (line 11). If the diameter of the remaining

Object State:
  X is a set of reals, initially $\emptyset$.
  Y is a set of reals, initially $\emptyset$.
input(P, x)
  pre: *true*
  post: $X' = X \cup \{x\}$
y := output(P)
  pre: $X \neq \emptyset$
  post: $Y' = Y \cup \{y\} \wedge$
    $range(Y) \subseteq range(X) \wedge$
    $|range(Y)| < \epsilon$.

Figure 1: Sequential Specification for Approximate Agreement

```
1   proc input(P: process, x: real)
2     if r[P].prefer = ⊥ then
3       r[P] := [prefer: x, round: 1]
4     end if
5   end input
7   proc output(P: process)
8     advance := false
9     loop
10      Scan r
11      E := {r[Q].prefer : r[Q].round ≥ r[P].round − 1}
12      L := {r[Q].prefer : r[Q].round = max_Q r[Q].round}
13      if |range(E)| < ε/2 then
14        return r[P].prefer
15      elseif |range(L)| < ε/2 or advance then
16        r := [prefer: midpoint(L),
17              round: r.round + 1]
18        advance := false
19      else advance := ¬ advance
20      end if
21    end loop
22  end output
```

Figure 2: Wait-Free Implementation of Approximate Agreement Object

preferences is less than $\epsilon/2$, P returns its own preference (lines 13–14). If the diameter of the leaders' preferences is less than $\epsilon/2$, then P advances its entry and resumes the loop (line 16). If the diameter of the leaders' preferences exceeds $\epsilon/2$, then P rescans the entries once more before advancing its entry. This rescan is implemented using the `advance` flag, set in lines 18 and 19.

In analyzing the algorithm, it will be useful to keep track of all values that a process writes to its register during the entire execution, as opposed to just the most recent value.. We will denote by "P's r-entry" (or r-preference) the unique value of $r[P]$ (or the `prefer` field in that value) with round number r, of all values written to $r[P]$ during an execution of the algorithm.

The essential idea of the algorithm is that the range spanned by the set of all processes r-entries shrinks in each round, and when it gets small enough, the processes correctly detect termination. Consider some prefix of an execution of the algorithm, and let X_r denote the set of all processes' r-preferences in that prefix, i.e., all `prefer` values written during that prefix with round number r. Lemmas 1, 2, and 3 bound how each X_r relates to the preceding X_{r-1}, and how these quantities change over time.

Lemma 1 *In any prefix of an execution of the approximate agreement algorithm, for all $r > 1$, $range(X_r) \subseteq range(X_{r-1})$.*

Proof: The proof is by induction on the length of the prefix. The base case is an empty prefix, in which the claim holds since $range(X_r) = \emptyset$ for all r.

If a process creates a preference as part of an input operation in line 3, then it may increase $range(X_1)$ but does not increase the range of any larger round; thus writing inputs preserves the induction hypothesis.

Now suppose P creates an r-preference x_p by writing a new entry in line 16. For each s, let X'_s be the set of s-preferences in the prefix preceding this write operation. By the induction hypothesis, $range(X'_s) \subseteq range(X'_{r-1})$ for all $s \geq r-1$.

If $\mathcal{L}_P$ is the set of leaders P computes (in the preceding line 12), then $\mathcal{L}_P$ consists of preferences at some round $r_{\max} \geq r-1$ (as the maximum round includes P's own round $r-1$). Thus $x_p = midpoint(\mathcal{L}_P) \in range(X'_{r_{\max}}) \subseteq range(X'_{r-1}) = range(X_{r-1})$. ■

We will say that P *expands* X_r if it writes a preference that increases $|range(X_r)|$.

Lemma 2 *If P expands X_r after observing the set of leaders $\mathcal{L}_P$, then the entries corresponding to preferences in $\mathcal{L}_P$ have round number $r-1$.*

Proof: As in the proof of Lemma 1, we use the fact that all preferences in $\mathcal{L}_P$ correspond to entries with the same round number $r_{\max} \geq r-1$.

Suppose now that $r_{\max} > r-1$. Then when P executes line 16, it chooses as its new preference $midpoint(\mathcal{L}_P) \in range(\mathcal{L}_P) \subseteq range(X_{r_{\max}}) \subseteq range(X_r)$, where the last inclusion follow from Lemma 1. But this contradicts the fact that P expands X_r. Thus $r_{\max} = r-1$. ■

Lemma 3 *In any prefix of an execution of the approximate agreement algorithm, for all $r > 1$, $|range(X_r)| \leq |range(X_{r-1})|/2$.*

Proof: We will show that the endpoints of $range(X_r)$ are the midpoints of overlapping subranges of $range(X_{r-1})$, from which the claim follows by a simple case analysis.

Let P be the first process to write $x_p = \min(X_r)$, Q be the first process to write $x_q = \max(X_r)$, and let $\mathcal{L}_P$ and $\mathcal{L}_Q$ their respective sets of leaders as computed in line 12 immediate preceding their writes of x_p and x_q in line 16. Since both writes expand X_r, Lemma 2 implies that all entries in $\mathcal{L}_P$ and $\mathcal{L}_Q$ have round number $r-1$. Note also that $\mathcal{L}_P$ contains P's $(r-1)$-preference and $\mathcal{L}_Q$ contains Q's $(r-1)$-preference.

We will now show that at least one of these preference also occurs in the set of leader values observed by the other process, and thus that $\mathcal{L}_P \cap \mathcal{L}_Q$ is nonempty.

Let P_1, Q_1 be the events that P and Q write their $(r-1)$-preferences, respectively; and let P_2, Q_2 be the events that P and Q start their following scans (line 10). Suppose P does not observe Q's $(r-1)$-preference in $\mathcal{L}_P$. Then Q_1 occurs after P_2, in which case Q_2 occurs after P_1, and thus Q's scan includes P's $(r-1)$-preference. Thus at least one of P's or Q's $(r-1)$-preferences appears in both $\mathcal{L}_P$ and $\mathcal{L}_Q$.

It follows that $range(\mathcal{L}_P) \cap range(\mathcal{L}_Q)$ is nonempty. Let $[a,b] = range(\mathcal{L}_P)$ and $[c,d] = range(\mathcal{L}_Q)$, so that $x_p = midpoint(range(\mathcal{L}_P)) = \frac{a+b}{2}$ and $x_q = midpoint(range(\mathcal{L}_P)) = \frac{c+d}{2}$. Then $x_q - x_p = \frac{c+d-a-b}{2}$. If $a \leq c \leq b \leq d$, then $c-a$ and $d-b$ are the lengths of non-overlapping intervals contained in $range(X_{r-1}$ and so $x_q - x_p \leq \frac{(c-a)+(d-b)}{2} \leq |range(X_{r-1})|/2$. If $a \leq c \leq d \leq b$, then $x_q \in [a,b]$ and $x_q - \frac{a+b}{2} \leq \frac{b-a}{2} \leq |range(X_{r-1})|/2$. The remaining case $c \leq a \leq b \leq d$ follows similarly. In each case, we have $|range(X_r)| = |x_p - x_q| \leq |range(X_{r-1})|/2$. ■

Lemma 3 says that the range of preferences shrinks exponentially in the number of rounds. Thus the range will eventually drop below $\epsilon/2$, the threshold for the termination test in line 13. In Lemma 4, we show that if

this test is true and a process executes the return in line 14, then later actions by other processes will not produce values outside the ϵ range permitted by the specification.

Lemma 4 *If P returns x_p at round r, and Q writes x_q at round r, then $|x_p - x_q| < \epsilon$.*

Proof: By contradiction. Without loss of generality, let Q the *first* process to write an r-preference x_q such that $|x_p - x_q| \geq \epsilon$. Let $\mathcal{L}_P$ be the set of leaders observed by P *after* writing x_p, and let $\mathcal{L}_Q$ be the set of leaders observed by Q *before* writing x_q. Note that $x_p \in range(\mathcal{L}_P)$ and $x_q \in range(\mathcal{L}_Q)$. Moreover, $x_q \notin \mathcal{L}_P$ because $|range(\mathcal{L}_P)| < \epsilon/2$ (from line 13), and P's write of x_p is not observed by Q when computing $\mathcal{L}_Q$, by Lemma 2.

Suppose $|range(\mathcal{L}_Q)| < \epsilon/2$. Because each process wrote its $(r-1)$-entry before reading the other's entry, and because neither process read the other's r-entry, one of the two processes must have read the other's $(r-1)$-entry, and therefore $\mathcal{L}_P \cap \mathcal{L}_Q \neq \emptyset$. It follows that $|range(\mathcal{L}_P \cup \mathcal{L}_Q)| \leq |range(\mathcal{L}_P)| + |range(\mathcal{L}_Q)| < \epsilon$. Because x_p and x_q lie within $range(\mathcal{L}_P \cup \mathcal{L}_Q)$, $|x_p - x_q| < \epsilon$.

Otherwise, if $|range(\mathcal{L}_Q)| \geq \epsilon/2$, then Q reads twice before writing x_q. Let $\mathcal{L}'_Q$ be the set of leaders it saw during the first read. Since Q reads twice, $|range(\mathcal{L}'_Q)| \geq \epsilon/2$. If Q finished reading $\mathcal{L}'_Q$ before Q wrote x_p, then $\mathcal{L}'_Q \subseteq \mathcal{L}_P$, and $|range(\mathcal{L}'_Q)| \leq |range(\mathcal{L}_P)| < \epsilon/2$, a contradiction. If Q finished reading $\mathcal{L}'_Q$ after Q wrote x_p, then it started reading $\mathcal{L}_Q$ afterwards, and $x_p \in \mathcal{L}_Q$, a contradiction. ■

Theorem 5 *Let Δ be an upper bound on the size of the range of the inputs. There exists a wait-free implementation of the approximate agreement object in asynchronous PRAM, in which each process executes at most $(2n+1)\log_2(\Delta/\epsilon) + O(n)$ steps before finishing.*

Proof: We show that the protocol in Figure 2 is correct. There are three points to check: (1) that every output value lies within the original input range, (2) that the diameter of the output set is less than ϵ, and (3) that the algorithm is wait-free and runs within the specified time bound.

The first point is an immediate consequence of Lemma 1. For the second point, suppose P returns x_p after round r and Q returns x_q after round s, where $r \leq s$. Lemma 4 states that every element of X_r lies within ϵ of x_p, and Lemma 1 that $range(X_s) \subseteq range(X_r)$, hence $|x_p - x_q| < \epsilon$. Finally, Lemma 3 implies that $|X_r| \leq \Delta \cdot 2^{-r+1}$, so that for some $r = \log_2(\Delta/\epsilon) + O(1)$ we

have states that $|X_r| < \epsilon/2$ in any prefix of the execution. Thus no process ever sees a larger range among the leaders at round r, and every process returns on or before round $r+1$. To get the bound stated in the theorem, note that each process takes at most $(2n+1)$ steps in each round. ■

Lemma 6 *Let Δ be the size of range of the inputs. An adversary scheduler can force some process executing any deterministic implementation of the output operation of an approximate agreement object to execute $\lfloor \log_3(\Delta/\epsilon) \rfloor$ steps before finishing.*

Proof: It is enough to prove the result for two processes. Consider an execution in which P and Q have distinct input values, and each executes an *output*. Define a process's *preference* at any point to be the value it returns if it runs by itself until termination. Note that the preference is well-defined as long as the process is deterministic, that the preference of a process that returns is equal to its return value, and that once one process returns the other will eventually return its own preference (as the first process is no longer running). Thus the *output* operations cannot both terminate while their preferences differ by more than ϵ.

We will show a lower bound on the number of steps it takes for the preferences of the two processes to converge. Initially, each process's preference is its input, for if it returns some other value without seeing any inputs of other processes, it may violate the condition that $range(Y) \subseteq range(X)$.

It is immediate from the definition that a process's preference can only change as the result of a step by another process. Consider the following scenario. Run P until it is about to change Q's preference, then do the same for Q. Alternate P and Q in this way as long as neither process changes preference. Eventually, since the operations cannot run forever, the object reaches a state where each process is about to change the other's preference. The adversary now has a choice of running P, Q, or both. Let p_0 be P's current preference, p_1 its preference if Q takes the next step, and let q_0 and q_1 be defined similarly. Depending on whom the adversary schedules next, the new preferences will differ by either $|p_0 - q_1|$, $|p_1 - q_0|$, or $|p_1 - q_1|$. The sum of these quantities is at least $|p_0 - q_0|$, thus the adversary can always choose one that is greater than or equal to $|p_0 - q_0|/3$, preventing the gap between the preferences from shrinking by more than one third. Repeating this strategy k times, an adversary scheduler can ensure that the range of the preferences is at least $\Delta/(3^k)$. Since each iteration of the strategy involves at least one operation by each process, we get the desired lower bound. ■

Curiously, the gap between the $\log_2(\Delta/\epsilon)$ rounds of the upper bound in Theorem 5 and the $\log_3(\Delta/\epsilon)$ rounds of the lower bound in Lemma 6 is not an accident. Since the first appearance of our results [25], Hoest and Shavit [28] have shown using topological methods that in an iterated snapshot model with a structure similar to that of our algorithm, $\log_3(\Delta/\epsilon)$ is in fact a tight bound for two processes, while $\log_2(\Delta/\epsilon)$ is tight for three or more.

Theorem 7 *For all $k > 0$, there exists an object with a K-bounded wait-free implementation, for $K > k$, that is not k-bounded wait-free.*

Proof: Consider an approximate agreement object with the unit interval as potential input range, and $\epsilon = 1/3^k$. From Lemma 6, this object is not k-bounded wait-free, but it is K-bounded wait-free for $K = O(nk)$ by Theorem 5. ■

Theorem 8 *There exists an object with a wait-free implementation but no bounded wait-free implementation.*

Proof: Consider an approximate agreement object with an unbounded input range. For any particular set of inputs, $\Delta = |range(X_1)|$ is bounded, and Theorem 5 shows that the approximate agreement algorithm eventually terminates. But by setting Δ large enough, any implementation can be force to run longer than any fixed bound by Lemma 6. ■

5 A Class of Constructible Objects

In this section, we describe a class of objects that can be constructed in the asynchronous PRAM model. These objects are characterized by a simple algebraic property of their operations, described in detail in Section 5.1. The property says that any two operations of the object must either commute, meaning that the state of the object after both have occurred does not reveal which happened first; or at least one must overwrite the other, meaning that if the overwriter occurs last it is impossible to determine if the other operation occurred at all. Some technical consequences of this definition are elaborated in Section 5.2. These are used in Section 5.3 to show that any history of an object satisfying the characterization can be described by a *linearization graph*, with the properties that (a) all linearizations of the graph correspond to histories of the object that are equivalent (in a formally

defined sense); and (b) appropriately-defined subgraphs of the linearization graph produce linearizations that correspond to histories of the object that are equivalent to prefixes of the full history. An algorithm that simulates objects by constructing families of consistent linearization graphs for each process, together with a proof of its correctness, is given in Section 5.4.

5.1 Commuting and Overwriting

We are now ready to state the algebraic conditions an object must satisfy for us to provide a wait-free implementation.

These conditions are defined in terms of the set of *legal* histories, defined as those meeting the object's sequential specification. If p is an operation, p_i denotes p's invocation, and p_r its response. We use "$\cdot$" to denote concatenation, and $H \cdot p$ to denote $H \cdot p_i \cdot p_r$, where H is a sequential history.

Definition 9 *Two sequential histories H and H' are* equivalent *if, for all sequential histories G, $H \cdot G$ is legal if and only if $H' \cdot G$ is legal.*

Definition 10 *Invocations p_i and q_i* commute *if, for all sequential histories H, if $H \cdot p$ and $H \cdot q$ are legal then $H \cdot p \cdot q$ and $H \cdot q \cdot p$ are legal and equivalent.*

Definition 11 *Invocation q_i* overwrites *p_i if, for all sequential histories H, if $H \cdot p$ and $H \cdot q$ are legal then $H \cdot p \cdot q$ is legal and equivalent to $H \cdot q$.*

This particular notion of commutativity is due to Weihl [45]. For brevity, we say that two operations commute when their invocations commute.

We will show how to construct a wait-free asynchronous PRAM implementation for any object whose sequential specification satisfies the following property:

Property 1 *For all operations p and q, either p and q commute, or one overwrites the other.*

For example, one data type that satisfies these conditions is the following *counter* data type, providing the following operations:

inc(c: counter, amount: integer)
dec(c: counter, amount: integer)

respectively increment and decrement the counter by a given amount,

reset(c: counter, amount: integer)

reinitializes the counter to *amount*, and

read(c: counter) returns(integer)

returns the current counter value. Note that *inc* and *dec* operations commute, every operation overwrites *read*, and *reset* overwrites every operation. Such a shared counter appears, for example, in randomized shared-memory algorithms [6], and in the implementation of logical clocks [33].

5.2 Preliminary Lemmas

Lemma 12 *The* overwrites *relation is transitive.*

Proof: Suppose r overwrites q, and q overwrites p.

By the definition of overwrites, there exists a sequential history H such that $H \cdot p$, $H \cdot q$, and $H \cdot r$ are legal, $H \cdot p \cdot q$ is equivalent to $H \cdot q$, and $H \cdot q \cdot r$ is equivalent to $H \cdot r$.

Since operations are total, there exists a response r'_r such that $G = H \cdot p \cdot q \cdot r_i \cdot r'_r$ is legal. Since q overwrites p, G is equivalent to $H \cdot q \cdot r_i \cdot r'_r$. Since $H \cdot q \cdot r$ is legal, and since operations are deterministic, $r_r = r'_r$.

Since r overwrites q, G is equivalent to $H \cdot p \cdot r$. Since q overwrites p, G is also equivalent to $H \cdot r$. We have shown that if $H \cdot p$ and $H \cdot r$ are legal, then $H \cdot p \cdot r$ is legal and equivalent to $H \cdot r$, hence r overwrites p. ■

Lemma 13 *Let H be a history with operations p, q, r, and s such that p precedes q, r precedes s, and p and s are concurrent. We claim that r must precede q.*

Proof: Since p and s are concurrent, s_i appears before p_r in H. Since r precedes s, r_i and r_r also appear before p_r. Finally, since p precedes q, q_i and q_r appear after p_r, and therefore r and q do not overlap, and r precedes q in H. ■

Our object simulation algorithm works by implicitly constructing sequential histories consistent with a concurrent execution. A central problem is to get all processes to agree on the order of operations in those cases where the order matters. In general, we will try to put overwritten operations before their overwriters, since this destroys the most evidence that might otherwise be used to convict us of non-linearizability. Unfortunately this heuristic is not enough to order all operations, as some pairs of operations might overwrite each other. For such groups of mutually overwriting operations, we

break ties using the indices of the processes carrying out the operations. This gives us an extended notion of overwriting, which we call *dominance*.

For the following definition, processes are ordered by their indices: $P_i < P_j$ if and only if $i < j$.

Definition 14 *An operation p of process P* dominates *operation q of Q if either (1) p overwrites q but not vice-versa, or (2) p and q overwrite each other and $P > Q$.*

Lemma 15 *The dominance relation is a strict partial order.*

Proof: First we show that dominance is transitive. Suppose r dominates q, and q dominates p, where operations p, q, and r are respectively executed by processes P, Q, R. By the definition of dominance, r overwrites q, and q overwrites p, hence, by transitivity (Lemma 12), r overwrites p. If p does not overwrite r, we are done, so suppose p also overwrites r. Since p overwrites r and r overwrites q, p overwrites q. Since p and q overwrite one another, and q dominates p, it must be that $P < Q$. Similarly, since q overwrites p, and p overwrites r, q overwrites r, and, by similar reasoning, $Q < R$. It follows that $P < R$, hence r dominates p.

We must also show that dominance is antisymmetric. Suppose an operation p of process P dominates an operation q of process Q. Then either (1) q does not overwrite p and thus does not dominate p; or (2) p and q overwrite each other, but since $P > Q$, q does not dominate p. ■

5.3 Precedence and Linearization Graphs

In this section, we define the precedence and linearization graphs used in the algorithm presented in Section 5.4.

A *precedence graph* is a directed acyclic graph that represents the partial order of operations in some history; each node in the graph corresponds to an operation, and there is an edge from p to q if p precedes q, i.e., if the response of p occurs before the invocation of q in the history.

Any linearization of the history is a linear extension of the partial order represented by the precedence graph, and thus corresponds to a topological sort of the graph. However, not all linear extensions give equivalent sequential histories. To ensure that all processes see a consistent picture, we augment the precedence graph with additional *dominance edges* based on the dominance relation of Definition 14. A dominance edge is directed from p to q if q dominates p; their direction is thus the reverse of the *precedence edges*, since a precedence edge runs from p to q if p precedes q. The

intuition is that we would like dominated operations to be placed earlier in the history, so that evidence of their presence or absence does not propagate in ways that might overly constrain the story that the implementation tells about the sequential execution it is claiming to simulate.

Because the combination of precedence and dominance edges might create cycles, not all possible dominance edges are added to the precedence graph. Instead, we add a maximal set that does not create a cycle, using the `lingraph` procedure from Figure 3. The result of this procedure is called a *linearization graph*, because its topological sort defines a linearization of the concurrent history.

In the actual algorithm, the purpose of the linearization graph is to ensure that no operation's result is affected by concurrent operations. In this respect, linearization graphs owe something to the *serialization graphs* [11] used in database theory, although the technical details are different.

Given a precedence graph $\mathcal{G}$, the associated linearization graph $L(\mathcal{G})$ is defined by the `lingraph` algorithm shown in Figure 3. Here, $\{p_1, \ldots, p_k\}$ represent the operations sorted in any order consistent with the precedence order. The algorithm constructs a sequence of *intermediate graphs* $\mathcal{L}_{i,j}$, for $0 \leq i < j \leq k$. For brevity, we say that the construction *visits* p_i when it compares p_i to p_j, for $i < j$.

Lemma 16 *If p and q are concurrent in $\mathcal{G}$, and p dominates q, then there is either a path from p to q or a path from q to p in $L(\mathcal{G})$.*

Proof: When `lingraph` visits the first of p or q, either there is already a path from p to q, or the edge $q \rightarrow p$ will be added in line 8 or line 11. ■

Lemma 17 *If there is no path between p and q in $L(\mathcal{G})$, then they commute.*

Proof: First observe that p and q must be concurrent, as otherwise they are adjacent in the precedence graph $\mathcal{G}$.

Suppose p and q do not commute. Then at least one overwrites the other and so one dominates the other. Applying Lemma 16, there is a path between them. ■

Lemma 18 *$L(\mathcal{G})$ is acyclic.*

Proof: By induction on the sequence of intermediate $\mathcal{L}_{i,j}$ graphs. Since $\mathcal{G}$ is acyclic, $\mathcal{L}_{1,0} = \mathcal{G}$ is acyclic. But because of the tests in lines 7 and 10, no new cycles are created by adding dominance edges. ■

```
1   proc lingraph(G: precedence graph)
2     L_{0,k} := G
3     for i in 1...k do
4       L_{i,i} := L_{i-1,k}
5       for j in i+1...k do
6         if p_i dominates p_j and
7           adding p_j → p_i to L_{i,j-1} does not create a cycle
8           then L_{i,j} := L_{i,j-1} ∪p_j → p_i
9         elseif p_j dominates p_i and
10          adding p_i → p_j to L_{i,j-1} does not create a cycle
11          then L_{i,j} := L_{i,j-1} ∪ p_i → p_j
12        else L_{i,j} := L_{i,j-1}
13        end if
14      end for
15    end for
16    return L_{k,k}
17  end lingraph
```

Figure 3: The Linearization Graph Construction

Lemma 18 tells us that the linearization graph contains no cycles, and can thus be topologically sorted to give a total order on operations. Lemma 17 tells us that this total order will correctly order all operations whose order we care about. In Lemma 20, below, we show that this fact is sufficient to show that all orderings of the linearization graph yield equivalent histories.

Definition 19 *A* linearization *of a precedence graph $\mathcal{G}$ is a sequential history constructed by a topological sort of $L(\mathcal{G})$.*

Lemma 20 *If $\mathcal{G}$ has a legal linearization, then all linearizations of $\mathcal{G}$ are legal and equivalent.*

Proof: By induction on the number of operations in $\mathcal{G}$. The result is immediate when the graph has a single operation.

Pick an operation p such that p has no outgoing edges in $L(\mathcal{G})$. Let $H = H_1 \cdot p \cdot H_2$ be the legal linearization of $\mathcal{G}$, and $G = G_1 \cdot p \cdot G_2$ any other linearization. Let $\mathcal{G}$' be $\mathcal{G}$ with p removed.

Since p has no outgoing edges in $L(\mathcal{G})$, each operation in H_2 and G_2 is concurrent with p, and hence commutes with p (Lemma 17), so H is equivalent to $H_1 \cdot H_2 \cdot p$. Now, $h' = H_1 \cdot H_2$ is a legal linearization of G',

$G' = G_1 \cdot G_2$ is a linearization of G', hence by the induction hypothesis, G' is legal and equivalent to H'. It follows that H is equivalent to $G_1 \cdot G_2 \cdot p$, and since p commutes with each operation in G_2 (see above), H is also equivalent to $G_1 \cdot p \cdot G_2$. ■

We now prove a few technical lemmas that will be used to show that appropriate partial views of the linearization graph yield consistent histories.

Lemma 21 *Let $\mathcal{G}$ be a precedence graph, and p_0 and p_1 operations concurrent in $\mathcal{G}$, such that there is a path from p_0 to p_1 in the intermediate graph $\mathcal{L}_{i,j}$ in the construction of $L(\mathcal{G})$. Any path of minimal length from p_0 to p_1 in $\mathcal{L}_{i,j}$ contains at most one edge from $\mathcal{G}$.*

Proof: If there is more than one precedence edge, then there exist operations p, q, r, and s in the path such that p precedes q, there is a path from q to r, and r precedes s. If q precedes s, then the path can be shortened, and therefore p and s are concurrent. By Lemma 13, however, r would then precede q, which contradicts the assumption that there is path from q to r. ■

Lemma 22 *If p dominates q, and there is a path from p to q in $L(\mathcal{G})$, then there exists an r such that r dominates p and r precedes q.*

Proof: Consider the first intermediate graph in the construction of $L(\mathcal{G})$ to contain a path from p to q. We claim that any path of minimal length from p to q in this graph contains exactly one precedence edge. It cannot contain more than one (Lemma 21), and if it contains none, then q dominates p by transitivity (Lemma 15), which is impossible because p dominates q.

This path traverses operations $p_0 = p, p_1, \ldots, p_m$ and $q_0, q_1, \ldots, q_\ell = q$, such that dominance edges link p_i to p_{i+1} and q_i to q_{i+1}, and p_m precedes q_0. Suppose $p \neq p_k$ and $q \neq q_0$. To construct the paths from p to p_k and q to q_0, the construction must add at least one edge between two of the p_i and at least one edge between two of the q_j. When the construction visits p_i, it adds a dominance edge from p_0 to p_i (unless $p_0 = p_i$), and from p_i to p_m (unless $p_m = p_i$). Although p dominates q, and hence so does p_i, the construction does not add an edge from q to p_i, implying that there must already be a path from p_i to q. Visiting p_i thus completes the path from p to q, implying that p_i must be the last operation visited. A symmetric argument, however, also shows that visiting q_j also completes a path from p to q, implying that q_j must also be the also last operation visited, a contradiction.

Suppose $p_m = p$. Consider the first intermediate graph in the construction of $L(\mathcal{G})$ to contain a path from q_0 to some q', concurrent with q_0, that dominates p. Pick a path of minimal length, and let q'' be the operation immediately before q' in this path. We claim that p and q' must be concurrent, since otherwise the path could be shortened. Lemma 13, however, implies that q'' precedes q_0, contradicting the assumption that there is a path from q_0 to q''.

It follows that $q_0 = q$, and the r in the lemma statement is $p_k \neq p$. ■

Lemma 23 *Let $\mathcal{G}$ be a precedence graph, p an operation of $\mathcal{G}$ with no outgoing edges, and let $\mathcal{G}' = \mathcal{G} - p$ be the graph obtained by removing p from $\mathcal{G}$. Then $L(\mathcal{G}')$ is a subgraph of $L(\mathcal{G})$.*

Proof: Suppose there is an edge from q to r in $L(\mathcal{G}')$ but not in $L(\mathcal{G})$. Because $\mathcal{G}$ is a subgraph of $\mathcal{G}$, the missing edge must be a dominance edge. The construction for $L(\mathcal{G})$ fails to insert this edge only if it completes a path from r to q before it can add an edge from q to r.

By Lemma 22, there exists r' in $L(\mathcal{G})$ such that r' dominates r, and r' precedes q. Since p does not precede any operations, r' and p are distinct, therefore r' is in G'. Since r' precedes q, the construction visits either r or r' before it visits q. Either way, it constructs a path from r to r' before it compares r and q, thus it completes a path from r to q, a path that does not exist in $L(\mathcal{G}')$. ■

Lemma 24 *Let p be an operation; let H_1 and H_2 be sequential histories such that $H_1 \cdot p$ and $H_2 \cdot p$ are both legal; and suppose that for any q in H_2 that is dominated by p, there exists an r in H_2 that precedes q and dominates p. Then $H_1 \cdot p \cdot H_2$ is legal.*

Proof: By induction on the length of H_2. The result is immediate if H_2 is empty. Otherwise, H_2 can be written as $q \cdot H_2'$, where q is an operation that p does not dominate. Either q dominates p, in which case the result is immediate, or p and q commute, in which case $H_1 \cdot p \cdot q \cdot H_2'$ is equivalent to $H_1 \cdot q \cdot p \cdot H_2'$, where the latter satisfies the conditions of the lemma, and the result follows from the induction hypothesis. ■

5.4 The Algorithm

A wait-free algorithm for implementing an object satisfying Property 1 is shown in Figure 4. The object is represented by its precedence graph. Each

```
1   % Shared data
2   root: array[1..n] of pointer to entry
4   proc execute(p_i: invocation) returns(response)
5     % Step 1: construct a response
6     view := atomic scan of root array
7     H := linearization of view
8     e := new entry
9     e.invocation := p_i
10    e.response := p_r such that H · p_i · p_r is legal
11    for i in 1 ... n do
12      e.preceding[i] := view[i]
13    end for
14    % Step 2: write out the response
15    root[P] := address of e
16    return p_r
17  end execute
```

Figure 4: A Wait-Free Implementation

operation is represented by an *entry*, a data structure with fields for the invocation, the response, and n pointers to each process's preceding entry. The graph is rooted in an *anchor* array whose P^{th} entry holds a pointer to the entry for process P's most recent operation.

A process executes an operation in two steps:

1. It takes an instantaneous snapshot of the anchor array using the *atomic scan* procedure described in Section 6. It then constructs a linearization graph from the precedence graph rooted at the snapshot array, and then constructs a linearization, called its *view*. Using a sequential implementation of the object, it determines the response to the invocation consistent with the view. It creates an entry for the operation, filling in the response and the precedence edges from the snapshot array.

2. The process updates the precedence graph by storing a pointer to the new entry in its position in the anchor array.

Each of these steps makes a single access to shared data: Step 1 uses the atomic scan algorithm given below, and Step 2 writes a single pointer into the shared *root* array. Informally, this algorithm exploits the commutativity and overwriting properties of operations to ensure that each process sees

"enough" of the object state to choose a correct response independently of any operations that may be taking place concurrently. We will show that the shared precedence graph always has a legal linearization.

Lemma 25 *Let $H_1 \cdot p \cdot H_2$ be a linearization of the shared precedence graph $\mathcal{G}$. If p and q are concurrent in $\mathcal{G}$, p dominates q, and q is in H_2, then there exists an r such that r dominates p and r precedes q.*

Proof: Since p and q are concurrent and do not commute, $L(\mathcal{G})$ contains a path from one to the other (Lemma 16). Since p appears before q in the linearization, this path must go from p to q. The result now follows directly from Lemma 22. ■

An entry that has been initialized but not yet written out is *pending*.

Theorem 26 *The following property is invariant: if the shared precedence graph is linearizable, then it remains linearizable after writing out any pending entry.*

Proof: By induction. The property holds trivially in the object's initial state, when the precedence graph is empty and no entries are pending. The property is preserved when P executes Step 1, since the result of writing out P's entry is linearizable by construction, and the result of writing out any other entry is unchanged.

It remains to check that writing out P's pending entry does not violate linearizability by "invalidating" any other process's pending operation. Suppose P and Q respectively have pending operations p and q. Let $\mathcal{G}$ be the current precedence graph, $\mathcal{G}_p$ the precedence graph after writing out p, $\mathcal{G}_q$ the precedence graph after writing out q, and $\mathcal{G}_{pq}$ the precedence graph after writing out both.

Let $H_1 \cdot p \cdot H_2 \cdot q \cdot H_3$ be a linearization of $L(\mathcal{G}_{pq})$. By Lemma 23, $L(\mathcal{G}_p)$ and $L(\mathcal{G}_q)$ are subgraphs of $L(\mathcal{G}_{pq})$, hence $H_1 \cdot p \cdot H_2 \cdot H_3$ is a linearization of $\mathcal{G}_p$ and $H_1 \cdot H_2 \cdot q \cdot H_3$ a linearization of $\mathcal{G}_q$. By the induction hypothesis, these are both legal sequential histories.

In particular, $H_1 \cdot p$ is legal, $H_1 \cdot H_2 \cdot q \cdot H_3$ is legal, and if p dominates any operation r in $H_2 \cdot q \cdot H_3$, then there exists an r' in $H_2 \cdot q \cdot H_3$ that precedes r and dominates p (Lemma 25). By Lemma 24, $G = H_1 \cdot p \cdot H_2 \cdot q \cdot H_3$ is legal. ■

Corollary 27 *The object implementation in Figure 4 is linearizable.*

Because of the generality of the algorithm, there is quite a bid of overhead in the construction and maintenance of the precedence and linearization graphs. For any particular data type, it should be possible to apply type-specific optimizations to discard most of the precedence graph, and to avoid reconstructing the entire linearization graph for each operation.

6 Atomic Scan

```
1   proc Scan(P: process, v: value) returns(value)
2     scan[P][0] := v ∨ scan[P][0]
3     for i in 1...n+1 do
4       for Q in 1...n do
5         scan[P][i] := scan[P][i] ∨ scan[Q][i-1]
6       end for
7     end for
8     return scan[P][n+1]
9   end Scan
```

Figure 5: The Scan Procedure

In this section, we show how to take an atomic snapshot scan of an array of multi-reader, single-writer registers in which process P writes the P^{th} array element. It is convenient to cast this problem in slightly more general form: since the array's state does not depend on the order in which distinct processes update their array elements, it is natural to treat the array state as the join in a $\vee$-semilattice of the input values[5] The snapshot scan simply returns the join of the register values.

Fix a $\vee$-semilattice L; for convenience we will assume that L contains a bottom element $\perp$ such that $\perp \vee x = x$ for all x in L. The atomic scan object has an operation $\text{Write}_L(P, v)$ for each process P and element v of L, and an operation $\text{ReadMax}(P)$ for each process P. The serial semantics of the object are straightforward: in any history H, the value returned by a $\text{ReadMax}(P)$ operation is the join of the values written by earlier $\text{Write}_L(Q, v)$ operations, for all Q.

The processes share an array $\text{scan}[1 \ldots n][0 \ldots n{+}1]$ of multi-reader/single-writer atomic registers, where P alone writes to each $\text{scan}[P][i]$. The opera-

[5] A $\vee$-semilattice is a partial order with a join operation (written as $\vee$); the join $a \vee b$ of a and b is the unique least element of the partial order that is greater than or equal to both a and b.

tions $\text{Write}_L(P, v)$ and $\text{ReadMax}(P)$ are each implemented using a stronger primitive operation, $\text{Scan}(P, v)$, defined in Figure 5. The Write_L operation is implemented by executing $\text{Scan}(P, v)$ and discarding the return value, while the ReadMax operation is implemented by executing $\text{Scan}(P, \perp)$.

6.1 Proof of Correctness

We demonstrate the correctness of the atomic scan algorithm in two steps. First, we show that any two values returned by Scan operations are comparable within the lattice L. Second, we use the lattice ordering of the returned values to order the implemented Write_L and ReadMax operations in any concurrent history H; this ordering will produce an equivalent serial history of the atomic scan object, thus proving linearizability. We use the usual order symbols $<, >, \geq, \leq$ for the semilattice order in L.

An *implementation* history is one in which *high-level* Scan invocations and responses are interleaved with *low-level read* and *write* invocations and responses in a constrained way: each Scan invocation is separated from its matching response by a sequence of *read* and *write* operations of the same process. Since *read* and *write* operations are linearizable by assumption, we may assume without loss of generality that the subsequence of low-level operations is a sequential history.

Let H be fixed implementation history, p a Scan operation in H executed by process P, and q a Scan operation by P. We use $p[k]$ as an abbreviation for the *write* operation to $\text{scan}[P][k]$ executed on behalf of the high-level operation p. We sometimes abuse this notation by using $p[k]$ also to refer to the value it writes. We say that $p[k]$ *directly-sees* $q[k-1]$ if P's *read* of $\text{scan}[P][k-1]$ appears after $q[k-1]$ in H. We say that $p[k]$ *sees* $q[l]$ if they lie in the in the reflexive, transitive closure of *directly-sees*. Note that for $p[k]$ to see $q[l]$ it is not enough that $p[k] \geq q[l]$; it must also occur later in time after a sequence of intermediate reads and writes that would allow the value $q[l]$ to be incorporated in the value $p[k]$.

Certain facts about the *sees* relation are important enough to state as lemmas. The proofs are straightforward and are omitted for brevity.

Lemma 28 *If $i \leq j$, then $p[j]$ sees $p[i]$.*

Lemma 29 *If $p \prec_H q$ and $q[k]$ and $p[k]$ exist, then $q[k] \geq p[k]$.*

It is also not difficult to see that any value written by a process is the join of the values seen by that process; more formally, we state:

Lemma 30 *For any $p[k]$ in H, if $0 \leq l < k$, then $p[k] = \bigvee \{q[l] \mid p[k] \text{ sees } q[l]\}$.*

The following lemma describes the principle on which the atomic scan algorithm depends:

Lemma 31 *If $p[k]$ and $q[k]$ both appear in H, for $k > 0$, then either $p[k]$ sees $q[k-1]$ or $q[k]$ sees $q[k-1]$.*

Proof: Suppose $p[k-1]$ precedes $q[k-1]$. Since Q's *read* of scan$[Q][k-1]$ appears after $q[k-1]$, it appears after $p[k-1]$, and $q[k]$ sees $p[k-1]$. otherwise, if $q[k-1]$ precedes $p[k-1]$, then $p[k]$ sees $q[k-1]$. ■

We now prove the consistency of the atomic scan operation.

Lemma 32 *Either $p[n+1] \geq q[n+1]$ or $q[n+1] \geq p[n+1]$.*

Proof: Let p', q' be Scan operations such that $p[n+1]$ sees $p'[0]$, and $q[n+1]$ sees $q'[0]$. We claim that:

$$p[n+1] \geq q'[0] \text{ or } q[n+1] \geq p'[0]. \tag{1}$$

Let $\{p_0, \ldots, p_{n+1}\}$ be an indexed set of Scan operations (not necessarily distinct) such that $p_0 = p'$, $p_{n+1} = p$, and for each k, $0 < k < n+1$, $p_k[k]$ directly-sees $p_{k-1}[k-1]$. Define $\{q_0, \ldots, q_{n+1}\}$ similarly; the existence of the sets follows from the definition of *sees*.

For each p_k, q_k, where $k > 0$, Lemma 31 implies that either $p_k[k]$ sees $q_k[k-1]$ or $q_k[k]$ sees $p_k[k-1]$, and thus one of p_k or q_k has the property that its $(k-1)^{st}$ *write* is seen by both $p_k[k]$ and $q_k[k]$. Denote this operation by x_k, and the associated process by X_k.

Now consider the indexed set $\{x_0, \ldots, x_{n+1}\}$. By the pigeonhole principle, there exist distinct i and j such that $i < j$ and $X_i = X_j$. If $x_i = x_j$, Lemma 28 immediately implies that $x_j[j-1]$ sees $x_i[i]$.

Otherwise, x_i must precede x_j, because $x_j[j]$ sees either $q_i[i]$ or $p_i[i]$, both of which see $x_i[i-1]$. Thus, by Lemma 29, $x_j[j-1] \geq x_i[j-1]$, but since $j - 1 \geq i$ Lemma 28 implies that $x_i[j-1]$ sees $x_i[i]$. Thus in either case $x_j[j-1] \geq x_i[i]$. $p[n+1]$ and $q[n+1]$ see $x_j[j-1]$, and $x_i[i]$ sees one of $p'[0]$, $q'[0]$, showing that Equation 1 holds.

Now suppose that $p[n+1]$ and $q[n+1]$ are incomparable. By Lemma 30, there must then exist a $p'[0]$ which $p[n+1]$ alone sees and a $q'[0]$ which $q[n+1]$ alone sees — contradicting Equation 1. ■

Theorem 33 *The atomic scan object implementation is linearizable.*

Proof: Consider any two operations x and y. Let $x \prec'_S y$ if either $x[n+1] < y[n+1]$ or $x[n+1] = y[n+1]$, x is a Write_L operation and y is a ReadMax operation. Extend $\prec'_S$ to a total order $\prec_S$; by Lemma 29 $\prec_S$ extends $\prec_H$, and thus we can use it to linearize H. That the resulting sequential history is legal follows directly from Lemma 32. ■

To implement the atomic snapshot algorithm used in the previous section, we make each value an n-element array of pointers, where the entire array is kept in a single register. (As noted above, numerous techniques exist for constructing large atomic registers from smaller ones.) Each array entry has an associated tag, and the maximum of two entries is the one with the higher tag. The join of two values is the element-wise maximum of the two arrays. The $\perp$ value is just an array whose tags are all zero. P writes the P^{th} position in the *anchor* array by initializing scan$[P][0]$ to an array whose P^{th} element has a higher tag than P's latest entry, and whose other elements have tag zero. (As a simple optimization, the other elements can simply be omitted.)

6.2 Running Time

Each Scan operation requires one *read* and one *write* operation to set scan$[P][0]$, plus n *read* and one *write* operations for each of $n+1$ passes through the loop. Thus a single Scan operation requires a total of $n^2 + n + 1$ *read* and $n+2$ *write* operations, where, as usual, n is the number of processes. Some minor gains arise by eliminating superfluous operations that simplify the proof: the very last write (to scan$[P][n+1]$) is unnecessary, as are the reads that a process does of its own registers. After eliminating these operations, a Scan requires $n^2 - 1$ *read* and $n+1$ *write* operations.

7 Conclusions

In this paper, we have explored some of the mathematical structure underlying the asynchronous PRAM model. We have seen that it encompasses a rich impossibility hierarchy, but it still supports wait-free implementations of a large class of objects that have a simple algebraic characterization.

Although we believe that asynchronous PRAM is considerably more realistic than its synchronous predecessor, it is still far from ideal. In one sense, asynchronous PRAM is too weak to be realistic. The only way for processes to synchronize is by read and write operations. One might justify this restriction in the same way one justifies ruler-and-compass constructions in

classical geometry: simply as an intellectual challenge. One cannot justify it as a realistic reflection of current practice. Nearly every major architecture since the 1970's has provided some form of *read-modify-write* operation that atomically reads and modifies memory. Examples include *test-and-set*, *compare-and-swap*, *fetch-and-add*, atomic *swap*, and many others. (Glew and Hwu [22] give an excellent survey of synchronization primitives provided by current architectures.) Today, it would be inconceivable to design a shared-memory multiprocessor without such atomic instructions.

There is another sense in which asynchronous PRAM may be too strong to be realistic. Many modern shared-memory multiprocessors do not guarantee that memory is sequentially consistent [34]: reads and writes to shared memory do not appear to occur atomically (e.g., [1, 36] and many commercial multiprocessors). In modern architectures, processors are fast, while memory and communication are slow, and as a result the cache coherency protocols necessary to enforce sequential consistency are expensive, and architects are often unwilling to pay this cost on every memory access. Recently, a number of researchers have started exploring the implications of such "weak" memories [3, 29, 37]. A satisfactory trade-off between ease of implementation and ease of use has yet to be established for shared-memory semantics.

In conclusion, although the asynchronous PRAM model explored in this paper has its limitations, we believe that the model is interesting in its own right, and we hope that the questions we have raised and the techniques we have developed here will be useful and informative when the "right" model comes along.

Acknowledgments

Hagit Attiya's remarks helped improve this paper. We are also grateful for the insightful comments and patience of the anonymous referees.

References

[1] S.V. Adve and M.D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots. Ninth ACM Symposium on Principles of Distributed Computing, to appear., 1990.

[3] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed shared memory. Technical Report TR GIT-CC-90-49, Georgia Institute of Technology, December 1990.

[4] Anderson. Composite registers. Technical Report TR-89-25, University of Texas at Austin, September 1989.

[5] J. Anderson and B. Grošelj. Pseudo read-modify-write operations: Bounded wait-free implementations. In 5th International Workshop on Distributed Algorithms, September 1991.

[6] J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, September 1990.

[7] J. Aspnes and M.P. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

[8] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[9] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, July 1994.

[10] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, April 1998.

[11] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[12] O. Biran, S. Moran, and S. Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 263–273. ACM, August 1988.

[13] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12):1506–1514, December 1988.

[14] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[15] J.F. Buss, P.C. Kanellakis, P.L. Ragde, and A.A. Shvartsman. Parallel algorithms with processor failures and delays. Technical Report CS-91-54, Brown University, Providence RI, August 1991.

[16] R. Cole and O. Zajicek. The APRAM: incorporating asynchrony into the PRAM model. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 169–178, Santa Fe, NM, June 1989.

[17] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures*, pages 85–94, Crete, Greece, July 1990.

[18] D. Dolev, N.A. Lynch, S.S. Pinter, E.W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[19] A. Fekete. Asymptotically optimal algorithms for approximate agreement. In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 73–87. ACM, August 1986.

[20] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[21] P.B. Gibbons. A more practical PRAM model. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 158–168, Santa Fe, NM, June 1989.

[22] A. Glew and W. Hwu. A feature taxonomy and survey of synchronization primitive implementations. Technical Report CRHC-91-7, University of Illinois at Urbana-Champagne, 1101 W. Springfield, Urbana, IL 61801, December 1990.

[23] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[24] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming, pages 197–206, March 1990.

[25] M.P. Herlihy. Impossibility results for asynchronous PRAM. In *Proceedings of the 3rd Annual Symposium on Parallel Algorithms and Architectures*, pages 327–336, 1991.

[26] M.P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

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

[28] Gunnar Hoest and Nir Shavit. Towards a topological characterization of asynchronous complexity (preliminary version). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 199–208, Santa Barbara, California, 21–24 August 1997.

[29] P. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. Technical Report TR GIT-ICS-89/39, Georgia Institute of Technology, Atlanta GA, October 1989.

[30] IBM. System/370 principles of operation. Order Number GA22-7000.

[31] P.C. Kannelakis and A.A. Shvartsman. Efficient parallel algorithms on restartable fail-stop processors. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*. ACM, August 1991.

[32] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[34] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[35] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[36] D Lenoski, J. Laudon, K Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, May 1990.

[37] R. Lipton and J. Sandberg. A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[38] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, November 1988.

[39] S. Mahaney and F.B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 237–249. ACM, August 1985.

[40] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[41] N. Nishimura. Asynchronous shared memory parallel computation. In *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures*, pages 76–84. ACM, July 1990.

[42] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[43] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[44] G.L. Peterson and J.E. Burns. Concurrent reading while writing ii: the multi-writer case. Technical Report GIT-ICS-86/26, Georgia Institute of Technology, December 1986.

[45] W.E. Weihl. Specification and implementation of atomic data types. Technical Report TR-314, MIT Laboratory for Computer Science, March 1984.