Impossibility Results for Asynchronous PRAM (extended abstract)

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square
Cambridge MA, 02139
herlihy@crl.dec.com
Digital Equipment Corporation
Cambridge Research Lab

CRL 91/6 June 3, 1991

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 *non-blocking* if some operation will always complete in a finite number of steps, it 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. It is known that asynchronous PRAM cannot be used to construct a non-blocking implementation of any object that solves two-process consensus, a class of objects that includes many common data types. It is natural to ask whether the converse holds: does asynchronous PRAM permit non-blocking implementations of any object that does not solve consensus? This papers shows that the answer is *no*. There is a strict infinite hierarchy among objects that do not solve consensus: there exist objects (1) without non-blocking implementations, (2) with implementations that are non-blocking but not wait-free, (3) with implementations that are wait-free but not bounded wait-free, and (4) with implementations that are K-bounded wait-free but not k-bounded wait-free for all $k > 0$ and some $K > k$.

This paper will appear in the Third Annual ACM Symposium on Parallel Algorithms and Architectures, July 21-24, 1991, Hilton Head, South Carolina.

1 Introduction

In the "classical" parallel random access machine (PRAM) model, 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 [8, 9, 14, 25]. In this model, asynchronous processes communicate by applying atomic read and write operations to the shared memory [1]. Techniques for implementing these memory locations, often called *atomic registers*, have also received considerable attention [4, 5, 20, 21, 24, 26, 27].

Much of the work on asynchronous PRAM models addresses the problem of computing functions, such as parallel summation, 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 sequence of operations, requiring that data structures be reused. It 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 *non-blocking* if some non-faulty process always completes an operation in a finite number of steps, despite failures of other processes. It is *wait-free* if every non-faulty process has this property, and it is *k-bounded wait-free*, for some fixed $k > 0$, if every non-faulty process always completes an operation within k steps. These properties form a hierarchy: bounded wait-free implies wait-free, and wait-free implies non-blocking. The non-blocking property permits individual processes to starve, but it guarantees that the system as a whole will make progress. The wait-free property excludes starvation; any process that continues to take steps will finish its operation, and the bounded wait-free property bounds how long it will take. Each 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

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

before a barrier will prevent the non-faulty processes from making progress.

Which objects have non-blocking implementations in asynchronous PRAM? Elsewhere [17, 15], we have shown that any object X can be assigned a *consensus number*, which is the largest number of processes (possibly infinite) that can achieve consensus asynchronously [13] by applying operations to a shared X. It is impossible to construct a non-blocking implementation of any object with consensus number n from objects with lower consensus numbers in a system of n or more processes, although any object with consensus number n is universal (it supports a wait-free implementation of any other object) in a system of n or fewer processes. A memory with atomic read and write operations has consensus number 1 (it cannot solve consensus between two processes), and therefore the asynchronous PRAM model is too weak to support non-blocking implementations of any object with a higher consensus number, including 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*.

It is natural to ask whether the converse holds: does asynchronous PRAM permit non-blocking implementations of the remaining objects, objects 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 the following strict infinite hierarchy among objects with consensus number 1.

- Objects without non-blocking implementations. These objects are too weak to solve two-process consensus, yet they cannot be implemented (in asynchronous PRAM) without critical sections.
- Objects with implementations that are non-blocking, but not wait-free. These objects can be implemented without critical sections, but it is impossible to guarantee fairness.
- 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.
- For all $k > 0$, objects with implementations that are K-bounded wait-free for some $K > k$, but not k-bounded wait-free.

This hieararchy is shown schematically in Figure 1. Our impossibility results (e.g., this object has no wait-free implementation) apply to systems with arbitrary numbers of processes, but some of our constructions (e.g., this object does have a non-blocking implementation) apply only to systems of two processes.

A specific contribution of this paper is the hierarchy itself, which shows that even relatively "weak" concurrent objects have a rich mathematical structure. Moreover, each level of the hierarchy requires a different kind of proof technique. Another, more general, contribution is to raise basic questions about the value of the asynchronous PRAM model. Although some synchronous PRAM

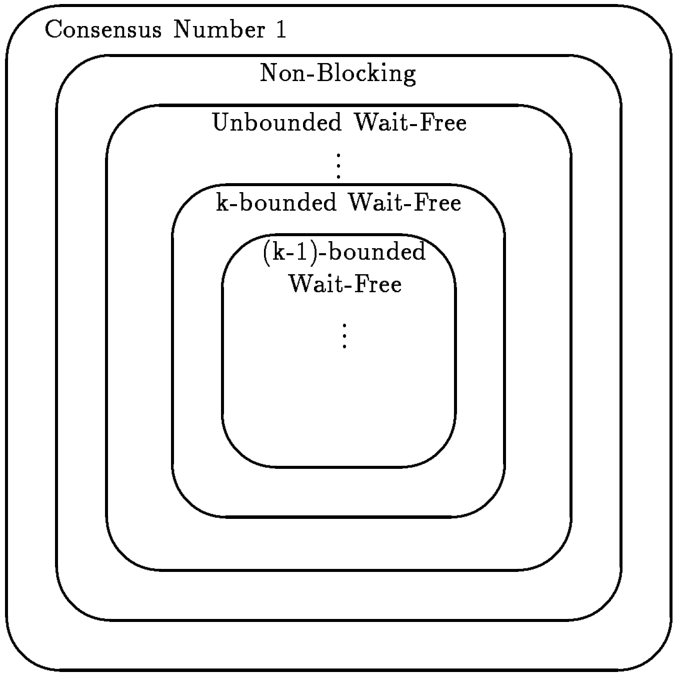

Figure 1: Hierarchy of Objects with Consensus Number 1

algorithms can be adapted to asynchronous PRAM [8, 9, 14, 25], 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 [19] 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* [16]. 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 The Model

A *concurrent system* consists of a collection of n *processes* that communicate through shared typed *objects*. Processes are *sequential* — each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. We make no fairness assumptions about processes. A process can halt, or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

Objects are data structures in memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to manipulate that object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time by a single process. For example, the behavior of a queue object can be specified by requiring that *enq* insert an item in the queue, and that *deq* remove the oldest item in the queue. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions. An object is *linearizable* [18] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of non-overlapping operations is preserved.

A *consensus protocol* is a system of n processes that communicate through a set of shared objects. The processes each start with an input value, either 0 or 1. Each process communicates with the others by applying operations to shared objects, and each process eventually chooses an output value and halts. A consensus protocol is required to be:

- *Consistent*: distinct processes never decide on distinct values.
- *Wait-free*: each process decides after a finite number of steps.
- *Valid*: the common decision value is the input to some process.

It is impossible to solve consensus for two or more processes in the asynchronous PRAM model [1, 6, 7, 10, 17, 22].

3 The 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 set of real numbers, let $range(S) = [\min(S), \max(S)]$, $midpoint(S) = (\min(S) + \max(S))/2$, and $|S| = \max(S) - \min(S)$. An *approximate agreement object* provides two operations:

Object State:
  X is a set of reals, initally $\emptyset$.
  Y is a set of reals, initally $\emptyset$.

input(P, x)
  pre: *true*
  post: $X' = X \cup \{x\}$

y := output(P)
  pre: $X \neq \emptyset$
  post: $Y' = Y \cup \{y\} \wedge$
    $range(Y) \subset range(X) \wedge$
    $|Y| < \epsilon$.

Figure 2: Sequential Specification for Approximate Agreement

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 2. 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 $range(Y) \subset range(X)$ 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 [3, 11, 12, 23], and Attiya, Lynch, and Shavit [2] independently derive upper and lower bounds for approximate agreement in shared memory that imply those given here.

A wait-free implementation of an approximate agreement object appears in Figure 3. 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 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, and discards those whose `round` fields trail its own by two or more. If the diameter of the remaining preferences is less than $\epsilon/2$,

```
input(P: process, x: real)
   if r[P].prefer = ⊥
     then r[P] := [prefer: x, round: 1]
   end if
end input

output(P: process)
   advance := false
   loop
        ℰ := entries that trail mine by 1 or less
        ℒ := leading entries
        if |ℰ| < ϵ/2
          then return r[P].prefer
          elseif |ℒ| < ϵ/2 or advance
                then r := [prefer: midpoint(ℒ),
                           round: r.round + 1]
                     advance := false
                else advance := ¬ advance
        end if
   end loop
   end output
```

Figure 3: Wait-Free Implementation of Approximate Agreement Object

P returns its own preference. If the diameter of the leaders' preferences is less than $\epsilon/2$, then P advances its entry and resumes the loop. If the diameter of the leaders' preferences exceeds $\epsilon/2$, then P rescans the entries once more before advancing its entry. For brevity, "P's r-entry" (or r-preference) refers to P's entry (or preference) with round number r.

First, we show that this implementation is correct. Let X_r denote the set of entries having round number r. (We sometimes abuse notation and use X_r to stand for the set of r-preferences; the exact meaning should be clear from context.)

Lemma 1 $X_r \subset X_{r-1}$.

Proof: By induction on round numbers. P's initial preference is trivially in $range(X_1)$. Assume the result for rounds less than r, and suppose P creates an r-preference x_p. If $\mathcal{L}_P$ is the set of leaders P observes, then $\mathcal{L}_P \subset range(X_{r-1})$ by the induction hypothesis, hence $x_p = midpoint(\mathcal{L}_P) \in range(X_{r-1})$. ∎

P *expands* X_r if it writes a preference that increases $|X_r|$.

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

Proof: They cannot have a lower round number, since P observes its own entry, and they cannot have a higher round number, since then $midpoint(\mathcal{L}_P) \in range(X_r)$. ∎

Lemma 3 $|X_r| \leq |X_{r-1}|/2$.

Proof: 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. Since both writes expand X_r, Lemma 2 implies that all entries in $\mathcal{L}_P$ and $\mathcal{L}_Q$ have round number $r-1$. One of P or Q must have observed the other's $(r-1)$-preference, so $\mathcal{L}_P \cap \mathcal{L}_Q \neq \emptyset$. Therefore, $|x_p - x_q| \leq |\mathcal{L}_P|/2 + |\mathcal{L}_Q|/2 \leq |X_{r-1}|/2$. ∎

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. Let Q be the first process to write 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 $|\mathcal{L}_P| < \epsilon/2$, and $x_p \notin \mathcal{L}_Q$, by Lemma 2.

Suppose $|\mathcal{L}_Q| < \epsilon/2$. 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 $|\mathcal{L}_P \cap \mathcal{L}_Q| \leq |\mathcal{L}_P| + |\mathcal{L}_Q| < \epsilon$. Because x_p and x_q lie within $range(\mathcal{L}_P \cup \mathcal{L}_Q)$, $|x_p - x_q| < \epsilon$.

Otherwise, if $|\mathcal{L}_Q| \geq \epsilon/2$, 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, $|\mathcal{L}'_Q| \geq \epsilon/2$. If Q finished reading $\mathcal{L}'_Q$ before Q wrote x_p, then $\mathcal{L}'_Q \subset \mathcal{L}_P$, and $|\mathcal{L}'_Q| \leq |\mathcal{L}_P| < \epsilon/2$, 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 *There exists a wait-free implementation of the approximate agreement object in asynchronous PRAM.*

Proof: We show that the protocol in Figure 3 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.

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 $X_s \subset X_r$, hence $|x_p - x_q| < \epsilon$. Finally, Lemma 3 states that $|X_r| < \epsilon/2$ for some r, implying that every process will return on or before round $r+1$. ∎

Lemma 6 *An adversary scheduler can force some process executing an output 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 uninterruptedly to conclusion. The *output* operations cannot both terminate while their preferences differ by more than ϵ. Initially, each process's preference is its input.

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 for k rounds, an adversary scheduler can ensure that the range of the preferences is at least $\Delta/(3^k)$, yielding the desired lower bound. ∎

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 $\epsilon = 1/3^k$. ∎

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

Proof: Consider an approximate agreement object with the rational numbers as potential input range. ∎

4 Non-Blocking

In this section, we construct an object having consensus number 1, a non-blocking implementation, but no wait-free implementation. This section illustrates an important difference between long-lived objects and short-lived decision problems: a decision problem, but definition, is executed once, and hence cannot distinguish between the non-blocking and wait-free properties.

We consider a system of two processes, P and Q. An *iterated approximate agreement* object has two operations:

Object State:
x_p and x_q are reals, initially $\perp$.
y_p and y_q are reals, initially $\perp$.
r_p and r_q are integers, initially 0.

input(P, x)
pre: $x_p = \perp$
post: $x'_p = x$

y := output(P)
pre: $x_p \neq \perp \wedge x_q = \perp$
post: $y'_p = x \wedge r'_p = r_p + 1$

y := output(P)
pre: $x_p \neq \perp \wedge x_q \neq \perp$
post: $y'_p = y \wedge$
$|y'_p - y_q| < \epsilon/2^{r_p} \wedge$
$y'_p \in range(x_p, x_q) \wedge$
$r'_p = r_p + 1$

Figure 4: Sequential Specification for Iterated Approximate Agreement

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

Each process P has a *starting estimate* x_p, and a *current estimate* y_p. As shown in Figure 4, P's starting estimate is initialized by *input*. Its current estimate is updated by *output* so that following P's i^{th} *output*, the range of the two processes' current estimates is less than $\epsilon/2^i$ for some fixed $\epsilon > 0$, and lies within the range of their original estimates. (The sequence of current estimates forms a Cauchy sequence that converges on a point in the range of the original estimates.) For simplicity, our specifications focus on executions in which any process that executes any operations executes an *input* followed by a sequence of *outputs*.

A non-blocking implementation of the iterated approximate agreement object is shown in Figure 5. The object is represented by a two-element array `r` of *entries*, where each entry has two fields: an integer `round`, initially 0, and a real `prefer`, initally $\perp$. Each process also has a persistent local variable `previous`, which holds a real interval, and survives from one invocation to the next. When P calls *input*, it initializes `round` to zero, `prefer` to the input value, and `previous` to the real line. When P calls *output*, it reads its entry, increments `round`, and enters the loop. Each time through the loop, it updates

```
input(P: process, x: real)
   r[P] := [prefer: x, round: 0]
   previous := [−∞, +∞]
end input

output(P: process) returns(real)
   p := r[P]
   p.round := p.round + 1
   loop
         r[P] := p
         q := r[Q]
         i := p.round + q.round
         range := [p.prefer ± ε/(2^i)]
         if q.prefer ∈ range or q.prefer ∉ previous
           then previous := range
                 return p.prefer
           elseif p.prefer < q.prefer
                 then p.prefer := p.prefer + ε/(2^i)
                 else  p.prefer := p.prefer − ε/(2^i)
           end if
         end loop
   end output
```

Figure 5: Non-Blocking Iterated Approximate Agreement Implementation

its own entry and reads Q's. It sums the two entries' `round` fields in variable `i`, and constructs a *desired* interval of radius $\epsilon/(2^i)$. If Q's preference lies outside P's previous interval, then it returns immediately. If Q's preference lies within the desired interval, then it returns, otherwise it chooses a new preference closer to the other's.

Lemma 9 *Every current estimate lies within the range of the original estimates.*

Proof: Initially, every preference lies within the original range, and each new preference lies between two earlier preferences. ∎

Lemma 10 *The operations in Figure 5 are non-blocking.*

Proof: It suffices to check that two concurrent *output* operations cannot both take an infinite number of steps without returning. If P takes only a finite number of steps, then Q's preference will converge to P's, and Q will return.

If P and Q both take an infinite number of steps, then once P and Q have each written a preference and read the other's, then each time through the loop reduces the distance between their preferences by a fixed amount, and eventually one will return. ∎

We use the following notation: p and q are output operations of P and Q, $\pi(p)$ is p's current preference (if active) or final return value (if completed), $\alpha(p)$ is the first value p assigns to variable `i`, and $\omega(p)$ is the current value of `i` (if active) or last value (if completed).

We construct an explicit linearization order as follows: if $\alpha(p) < \alpha(q)$ then p is ordered before q, and if $\alpha(p) = \alpha(q)$ then p and q are ordered arbitrarily.

Lemma 11 *The α function defines a valid linearization order.*

Proof: If p finishes before q begins, then $\alpha(p) < \alpha(q)$. ∎

Define $B(p)$ to be the open ball of radius $\epsilon/(2^{\omega(p)})$ around $\pi(p)$. If p_i is P's i^{th} *output*, then $i \leq \alpha(p_i) \leq \omega(p_i)$. In particular, if $y \in B(p_i)$, then $|y - \pi(p_i)| < \epsilon/(2^i)$.

Lemma 12 *If p is a completed* output *of P, and p' a later* output*, either completed or active, then $B(p') \subset B(p)$.*

Proof: The distances from $\pi(p')$ to the endpoints of $B(p)$ are always integral multiples of $\omega(p')$. ∎

Theorem 13 *The algorithm in Figure 5 is a non-blocking implementation of an iterated approxmiate agreement object.*

Proof: Lemma 9 states that all estimates lie within the range of the current estimates, and Lemma 10 states that one operation will always complete in a finite number of steps. It remains to check that each process's i^{th} estimate lies within $\epsilon/(2^i)$ of the other's current estimate. Suppose p_i is linearized between q_j and q_{j+1}, and that p_i is the first to violate correctness.

Suppose p_i returns after observing that Q's preference lies outside P's prevously committed range. That preference must have been written by q_j, and q_j must still be active, thus p_i cannot be the first to violate correctness.

Suppose p_i returns after observing that $\pi(p_i)$ lies within $\epsilon/(2^{\omega(p)})$ of $\pi(q_k)$. Since $\omega(q_k) \leq \omega(p_i)$, $\pi(p_i) \in B(q_k)$. If $k = j$, then we are done, since q_{j+1} has no return value yet. If $k \geq j$, Lemma 12 implies that $B(q_j) \subset B(q_{j+1}) \subseteq B(q_k)$, hence correctness is not violated. ∎

Although the algorithm in Figure 5 is non-blocking, one can easily check that it is not wait-free: an *output* operation can be forced to run forever if it is overtaken sufficiently often by other *outputs*. We now show that this property is inherent in any solution of iterated approximate agreement.

Consider an execution in which P and Q each performs an *input* followed by an *output*. Following the terminology of Fisher, Lynch, and Paterson [13], P is *bivalent* in any state where its output value is not yet determined, otherwise it is *univalent*. P's state is *x-valent* if it is univalent with eventual output value x. A *decision step* for P is an operation that carries P from a bivalent to a univalent state. P is *ambivalent* in a state s if there exists a state s' such that P is x-valent in s, y-valent in s', but s and s' are indistinguishable to Q.

Lemma 14 *From an initial state in which P and Q have different input values, it is possible to reach a state in which P is ambivalent.*

Proof: Assume without loss of generality that P has input 0, Q input 1, and $\epsilon < 1$. P is bivalent in this initial state: if P is run to completion before Q starts, its *output* returns 0, and if Q is run to completion before P starts, then P returns a value in $(1-\epsilon, 1]$.

Consider the following execution, which leaves P bivalent. In the first stage, run P until it reaches a state where it cannot continue without becoming univalent. P must eventually reach such a state, since it cannot run forever. In the second stage, run Q until it cannot continue without making P univalent, and in successive stages, alternate running P and Q until each is about to make P univalent. Because the processes cannot run forever leaving P bivalent, it must eventually reach a state s in which any subsequent step of either process forces P to be univalent. Since P is still bivalent, however, some enabled step of P carries P to an x-valent state, and some enabled step of Q carries P to a y-valent state, where x and y are distinct.

We now do a case analysis of the operations executed by P and Q. In each case, we show that certain combinations are impossible by constructing two executions starting in s, one in which P returns x, and one in which P returns y, but where the protocol states appear identical to P.

- Suppose Q is about to read a shared register. (1) P runs to completion, returning x, and (2) Q reads, and P executes to completion, returning y.
- Suppose the processes are about to write to different registers. (1) P writes, Q writes, and P executes to completion, returning x, and (2) Q writes, P writes, and P executes to completion, returning y.
- Suppose the processes are about to write to the same register. (1) P writes and P executes to completion, returning x, and (2) Q writes, P writes, and P executes to completion, returning y.

The only remaining combination is that P is about to read a register which Q is about to write. As soon as Q writes, P's state becomes ambivalent, since P must be univalent, but Q has no way to predict P's output. ∎

Bivalence arguments have been used to show the impossibility of shared-memory consensus [1, 7, 10, 17, 22], and the *ambiguous choice lemma* of Burns and Peterson [6].

Theorem 15 *The iterated approximate agreement object has no wait-free implementation.*

Proof: Consider an execution in which P and Q respectively input 0 and 1 and execute a single *output*. By Lemma 14, it is possible to reach a state where P's *output* has returned, Q's has not, and Q is ambivalent. Let y and z be two possible values Q could return in this state. Pick a k such that $\epsilon/(2^k) < |y - z|$. If P executes k more *outputs* in isolation, then it will be unable to return a value consistent with both y and z. ∎

5 Final Impossibility Results

In this section, we construct an object having consensus number 1 but no non-blocking implementation. To prove this result, we exhibit an object with the curious property that although it itself is too weak to solve two-process consensus, it can only be implemented by "stronger" objects that do solve consensus.

A *blind consensus* object's state consists of two boolean variables, called *red* and *blue*, initially both *false*. It provides two operations:

fix(c: color) returns (boolean)
blind() returns (color,boolean)

The first time *fix* is called, it sets the argument variable to *true* and returns its previous value. Subsequent calls with the same argument leave the variables unaffected, and non-deterministically return either *true* or *false*. The *blind* operation non-deterministically chooses a variable, sets it to *true*, and returns the chosen variable's name and previous value.

Lemma 16 *The blind consensus object cannot solve two-process consensus.*

Proof: Suppose otherwise. By an argument similar to the one given above in the proof of Theorem 15, any two-process consensus protocol can be maneuvered into a state where the decision value is 0 if P takes the next step, and 1 if Q takes the next step. The rest is a case analysis.

Suppose P and Q both call *fix*. If the invocations have distinct arguments, then they commute, so assume the invocations both apply to *red*. One possible execution is: P's call to *fix* returns some value v, any subsequent calls to *fix* also return v, any subsequent calls to *blind* apply to *blue*, and P eventually decides 0. Another possible execution is: Q's call to *fix* returns v, P's subsequent calls to *fix* also return v, its subsequent calls to *blind* apply to *blue*, and P

eventually decides 1. Since the two executions appear equivalent to P, we have a contradiction.

If P applies *fix* to *red* and Q calls blind, then *blind* can always choose *blue*, forcing the operations to commute. Similarly, if both processes call *blind*, the operations can always choose distinct variables. ∎

Theorem 17 *It is impossible to construct a non-blocking implementation of a blind consensus object in asynchronous PRAM.*

Proof: We show that any *implementation* of a blind consensus object can be transformed into a two-process consensus protocol.

An operation implementation is *deterministic* if it always returns the same response when run in the absence of concurrency. We may restrict our attention to deterministic implementations, since any non-determinstic implementation can always be rendered deterministic by fixing the outcome of any non-deterministic choices. Suppose the *blind* operation returns *red* when executed in isolation by P starting in the initial state (the case where it returns *blue* is symmetric). Now, we construct a consensus protocol as follows. P executes the implementation of *blind*, and Q executes the implementation of *red*. They decide 0 if P's result is (*red, false*) and Q's is *true*, and otherwise they decide 1. It is easily checked that this protocol is consistent, wait-free, and valid. ∎

We leave as an interesting open question whether every object whose sequential specification is *deterministic* has a non-blocking implementation in asynchronous PRAM.

Acknowledgments

Hagit Attiya's remarks helped improve this paper.

References

[1] J.H. Anderson and M.G. Gouda. The virtue of patience: Concurrent programming with and without waiting. University of Texas at Austin Technical Report.

[2] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? In *31st Annual Symposium on the Foundations of Computer Science*, October 1990.

[3] 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 *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 263–273, August 1988.

[4] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.

[5] 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.

[6] J.E. Burns and G.L. Peterson. The ambiguity of choosing. In *Eighth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 145–157, August 1989.

[7] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

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

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

[10] D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

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

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

[13] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[14] P.B. Gibbons. A more practical pram model. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. ACM, July 1989.

[15] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[16] 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.

[17] M.P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[18] 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.

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

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

[21] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[22] M.C. Loui and H.H. Abu-Amara. *Memory Requirements for Agreement Among Unreliable Asynchronous Processes*, volume 4, pages 163–183. JAI Press, 1987.

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

[24] 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.

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

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

[27] 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.