Are Wait-Free Algorithms Fast?

HAGIT ATTIYA

Technion, Haifa, Israel

NANCY LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

NIR SHAVIT

Tel-Aviv University, Tel-Aviv, Israel

Abstract. The time complexity of wait-free algorithms in "normal" executions, where no failures occur and processes operate at approximately the same speed, is considered. A lower bound of log n on the time complexity of any wait-free algorithm that achieves *approximate agreement* among n processes is proved. In contrast, there exists a non-wait-free algorithm that solves this problem in constant time. This implies an $\Omega(\log n)$ time separation between the wait-free and non-wait-free computation models. On the positive side, we present an $O(\log n)$ time wait-free approximate agreement algorithm; the complexity of this algorithm is within a small constant of the lower bound.

Categories and Subject Descriptors: B.3.2 [**Memory structures**]: Design styles—*shared memory*; C.1.2 [**Processor Architectures**]: Multiple data stream architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors* (*MIMD*); D.1.3 [**Programming Techniques**]: Concurrent programming—*distributed programming*; D.4.1 [**Operating Systems**]: Process management—*concurrency, multiprocessing/multiprogramming, synchronization*; F.1.2 [**Computation by Abstract Devices**]: Modes of computation—*parallelism and concurrency*

General Terms: Algorithms, performance, reliability, theory

Additional Key Words and Phrases: Approximate agreement, fault-tolerance, wait-free

A preliminary version of this work appeared in the *Proceedings of the 31st Annual Symposium on Foundations of Computer Science* (St. Louis, Mo., Oct.). IEEE, New York, 1990, pp. 55–64.

This work was supported by Office of Naval Research (ONR) contracts N00014-85-K-0168 and N00014-91-J-1046, by National Science Foundation (NSF) grants CCR 85-11442 and CCR 89-15206, and by DARPA contracts N00014-87-K-0825, N00014-89-J-1988, and N0014-92-J-4033.

Part of this work was performed while H. Attiya was at the Laboratory of Computer Science, MIT.

The work of N. Shavit was performed at the Laboratory for Computer Science, MIT, and in part also at the Hebrew University, Jerusalem, and at the IBM Almaden Research Center.

Authors' present addresses: H. Attiya, Department of Computer Science, Technion, Haifa 32000, Israel; N. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139; N. Shavit, Department of Computer Science, School of Mathematics, Tel-Aviv University, Tel-Aviv 69978, Israel.

1. *Introduction*

In shared-memory distributed systems, some number n of independent asynchronous processes communicate by reading and writing to shared memory. In such a computing environment, it is possible for processes to operate at very different speeds, for example, because of implementation issues, such as communication and memory latency, priority-based time-sharing of processors, cache misses and page faults. It is also possible for processes to fail entirely. *Wait-free algorithms* have been proposed as a mechanism for computing in the face of variable speeds and failures: a wait-free algorithm guarantees that each nonfaulty process terminates regardless of the speed and failure of other processes [Herlihy, 1991; Lamport, 1986a].[1] The design of wait-free algorithms has been a very active area of research recently.[2]

Because wait-free algorithms guarantee that fast processes terminate without waiting for slow processes, wait-free algorithms seem to be generally thought of as *fast*. However, while it is obvious from the definition that wait-free algorithms are highly resilient to failures, we believe that the assumption that such algorithms are fast requires more careful examination.

We study the *time complexity* of wait-free and non-wait-free algorithms in "normal" executions, where no failures occur and processes operate at approximately the same speed. We select this particular subset of the executions for making the comparison, because it is only reasonable to compare the behavior of the algorithms in cases where both are required to terminate. Since wait-free algorithms terminate even when some processes fail, while non-wait-free algorithms may fail to terminate in this case, the comparison should only be made in executions in which no process fails, that is, in *failure-free* executions. The time measure we use is the one introduced in [Lamport, 1976; 1977] and used to evaluate the time complexity of asynchronous algorithms, in, for example, [Arjomandi et al., 1983], [Cole and Zajicek, 1989], [Lynch and Fischer, 1981], [Lynch and Goldman, 1989], and [Peterson and Fischer, 1973]. To summarize, we are interested in measuring the time cost imposed by the wait-free property, as measured in terms of extra computation time in the most normal (failure-free) case.

In this paper, we address the general question by considering a specific problem—the *approximate agreement* problem studied, for example, in [Dolev et al., 1986], [Fekete, 1987a; 1987b], and [Mahaney and Schneider, 1985]; we study this problem in the context of a particular shared-memory primitive—single-writer multi-reader atomic registers. In this problem, each process starts with a real-valued input, and (provided it does not fail) must eventually produce a real-valued output. The outputs must all be within a given distance ϵ of each other, and must be included within the range of the inputs. This problem, a weaker variant of the well-studied problem of distributed consensus (e.g., [Fischer et al., 1985; Lamport et al., 1982]), is closely related to the important problem of synchronizing local clocks in a distributed system.

[1] Wait-free is the shared-memory analogue of the *nonblocking property* for *synchronous* transaction systems (cf. [Dwork and Skeen, 1983, Skeen, 1982]).

[2] See, for example, Afek et al. [1993], Anderson [1990], Aspnes and Herlihy [1990], Dolev et al. [1988], Herlihy [1991], Lamport [1986a; 1986b], Li et al. [1989], Peterson [1983], Peterson and Burns [1987], Schaffer [1988], and Vitanyi and Awerbuch [1986].

Approximate agreement can be achieved very easily if waiting is allowed, by having a designated process write its input to the shared memory; all other processes wait for this value to be written and adopt it as their outputs. In terms of the time measure described above, it is easy to see that the time complexity of this algorithm is constant—independent of n, the range of inputs and ϵ. On the other hand, there is a relatively simple wait-free algorithm for this problem, which we describe in Section 3, and which is based on successive averaging of intermediate values. The time complexity of this algorithm depends quadratically on n, and logarithmically on the size of the range of input values and on $1/\epsilon$. A natural question to ask is whether the time complexity of this algorithm is optimal for wait-free approximate agreement algorithms.

Our first major result is an algorithm for the special case where $n = 2$, whose time complexity is constant, that is, it does *not* depend on the range of inputs or on ϵ (Section 5). The algorithm uses a novel method of overcoming the uncertainty that is inherent in an asynchronous environment, without resorting to synchronization points (cf. [Gibbons, 1989]) or other waiting mechanisms (cf. [Cole and Zajicek, 1989]): This method involves ensuring that the two processes base their decisions on information that is approximately, but not exactly, the same.

Next, using a powerful technique of integrating wait-free (but slow) and non-wait-free (but fast) algorithms, together with an $O(\log n)$ wait-free input collection function, we generalize the key ideas of the 2-process algorithm to obtain our second major result: a wait-free algorithm for approximate agreement whose time complexity is $O(\log n)$ (Section 6). Thus, the time complexity of this algorithm does not depend on either the size of the range of input values or on ϵ, but it still depends on n, the number of processes.

At this point, it is natural to ask whether the logarithmic dependence on n is inherent for wait-free approximate agreement algorithms, or whether, on the other hand, there is a constant-time wait-free algorithm (independent of n). Our third major result shows that the $\log n$ dependency is inherent: any wait-free algorithm for approximate agreement has time complexity at least log n (Section 7).[3] This implies an $\Omega(\log n)$ time separation between the non-wait-free and wait-free computation models.

We note that the constant-time 2-process algorithm behaves rather badly if one of the processes fails. The *work* performed in an execution of an algorithm is the total number of atomic operations performed in that execution by all processes before they decide. We present a trade-off between the time complexity of and the work performed by any wait-free approximate agreement algorithm. We show that for *any* wait-free approximate agreement algorithm for 2 processes, there exists an execution in which the work exhibits a nontrivial dependency on ϵ and the range of inputs.

In practice, the design of distributed systems is often geared towards optimizing the time complexity in "normal executions," that is, executions where no failures occur and processes run at approximately the same pace, while building in safety provisions to protect against failures (cf. [Lampson, 1983]). Our results indicate that, in the asynchronous shared-memory setting, there are problems for which building in such safety provisions *must* result in

[3] The lower bound is attained in an execution where processes run synchronously and no process fails.

performance degradation in the normal executions. This situation contrasts with that occurring, for example, in synchronous systems that solve the distributed consensus problem. In that setting, there are *early-stopping* algorithms (e.g., [Dolev et al., 1982; Dwork and Moses, 1990; Moses and Tuttle, 1988]) that tolerate failures, yet still terminate in *constant* time when no failures occur. The exact cost imposed by fault-tolerance on normal executions has been studied, for example, in [Coan and Dwork, 1986], [Dwork and Moses, 1990], and [Moses and Tuttle, 1988]. For synchronous message-passing systems, it has been shown that nonblocking protocols take twice as much time, in failure-free executions, as blocking protocols [Dwork and Skeen, 1983].

Recent work has addressed the issue of adapting the usual synchronous shared-memory PRAM model to better reflect implementation issues, by reducing synchrony [Cole and Zajicek, 1989; 1990; Gibbons, 1989; Martel et al., 1989; Nishimura, 1990] or by requiring fault-tolerance [Kanellakis and Shvartsman, 1989; Kedem et al., 1990]. To the best of our knowledge, the impact of the *combination* of asynchrony and fault-tolerance (as exemplified by the wait-free model) on the time complexity of shared-memory algorithms has not previously been studied. Martel et al. [1990] present efficient fault-tolerant asynchronous PRAM algorithms. Their algorithms optimize work rather than time and employ randomization. Another major difference is that they assume that inputs are stored in the shared memory, so that every process can access the input of every other process.

The rest of the paper is organized as follows: In Section 2, we present formal definitions of the systems considered in this paper and introduce the time measure. The approximate agreement problem is defined in Section 3, where we also present a fast non-wait-free algorithm and a slow wait-free algorithm for reaching approximate agreement. Section 4 introduces a "bias" function on which the algorithms in the following sections are based. Proofs of the various properties of this function are, deferred to Section 9, to ease the presentation. A constant-time wait-free algorithm for approximate agreement between two processes is presented and proven correct in Section 5; key ideas from this algorithm are used in the $O(\log n)$ time wait-free approximate agreement algorithm presented in Section 6. Section 7 contains the $\log n$ time lower bound for wait-free approximate agreement algorithms. Section 8 presents the lower bound for the trade-off between the time complexity and the work complexity of a wait-free algorithm for approximate agreement. We conclude, in Section 10, with a discussion of the results and directions for future research.

2. *Model of Computation and Time Measure*

In this section, we describe the systems and the time measure we will consider. Our definitions are standard and are similar to the ones in, for example, [Arjomandi et al., 1983], [Herlihy, 1991], [Lamport, 1986a; 1986b], [Loui and Abu-Amara, 1987], and [Lynch and Fischer, 1981].

A *system* consists of n processes $p_0, \ldots, p_{n-1}$. Each process is a deterministic state machine, with a possibly infinite number of states. We associate with each process a set of *local states*. Among the states of each process are a subset called the *initial states* and another subset called the *decision states*. Processes communicate by means of a finite number of *single-writer multi-reader atomic* *registers* (also called *shared variables*). No assumption is made regarding the size of the registers. Each process p_i has two atomic operations available to it for accessing a shared register R:

—*write*(R, v) writes the value v to the shared variable R.
—*read*(R) reads the shared variable R and returns its value v.

A system configuration consists of the states of the processes and registers. Formally, a *configuration* C is a vector $\langle s_0, \dots, s_{n-1}, v_1, \dots \rangle$ where s_i is the local state of process p_i and v_j is the value of the shared variable R_j. Each shared variable may attain values from some *domain* that includes a special "undefined" value, $\perp$. An *initial configuration* is a configuration in which every local state is an initial state and all shared variables are set to $\perp$. For a configuration $C = \langle s_0, \dots, s_{n-1}, v_1, \dots \rangle$, *state*$(p_i, C)$ denotes the state of p_i in C and *val*(R_j, C) denotes the value of register R_j in C, that is, *state*$(p_i, C) = s_i$ and *val*$(R_j, C) = v_j$.

We consider an interleaving model of concurrency, where executions are modeled as sequences of steps. Each step is performed by a single process. A process p_i performs either a *write*(R, v) operation or a *read*(R) operation (which returns a value v), but not both, performs some local computation, and changes to its next local state. The next configuration is the result of these modifications. We assume that each process p_i follows a *local algorithm* A_i that deterministically determines p_i's next step: A_i determines a variable R and whether p_i is to read or write R as a function of p_i's local state. If p_i is to read R, then A_i determines p_i's next state as a function of p_i's current state and the value v read from R. If p_i is to write R, then A_i determines p_i's next state and the value v to be written to R as a function of p_i's current state. An *algorithm* is a function A mapping each i to a local algorithm A_i for p_i.

An *event of* p_i is simply p_i's index i. A *schedule* is a finite or infinite sequence of events. We denote by λ the empty schedule, with no events. We denote the configuration resulting from the application of a finite schedule σ to a configuration C by $C\sigma$. An *execution fragment* starting from a configuration C is a finite or infinite alternating sequence of configurations and events, $C_0, i_1, C_1, \dots, C_{k-1}, i_k, \dots$, where $C = C_0$ and $C_k = C_{k-1} i_k$, for all $k \geq 1$. We assume that a finite execution fragment ends with a configuration. The schedule *associated with* this execution fragment is $i_1, \dots, i_k, \dots$. Conversely, the (unique) execution fragment resulting from applying a schedule σ to a configuration C is denoted by (C, σ). An *execution* is an execution fragment starting with an initial configuration.

Given an infinite schedule σ, a process is *faulty* in σ if it takes a finite number of steps (i.e., has a finite number of events) in σ, and *nonfaulty*, otherwise. An infinite schedule σ is *f-admissible* if at most f processes are faulty in σ. In particular, a 0-admissible schedule is called *failure-free*. These definitions also apply to execution fragments by means of their associated schedules.

Let $\mathcal{I}$ be a fixed *input domain* and $\mathcal{D}$ be a fixed *decision domain*. Each initial state of p_i is associated with an input value in $\mathcal{I}$. For each process p_i and $d \in \mathcal{D}$ we define a subset, $D_{i,d}$, of the states of p_i. We assume that for each p_i, the sets $D_{i,d}$ are pairwise disjoint. We also assume that decisions are irrevocable, that is, the algorithm transitions are such that if p_i is in a state of

$D_{i,d}$ it will remain in a state of $D_{i,d}$. We call the set $D_{i,d}$ the *d-decision states* of p_i.

A *decision problem* (or just *problem*) Π *of size n* is a relation between $\mathscr{I}^n$ and $\mathscr{D}^n$. An algorithm *f-solves* a decision problem Π if in all executions the decisions made can be completed to a decision vector that is in the relation Π to the inputs of the processes. Furthermore, in any f-admissible execution, every nonfaulty process eventually decides. An algorithm that $(n - 1)$-solves a problem Π is also called a *wait-free* algorithm for Π. Intuitively, even if all processes but one fail when a wait-free algorithm is executed, this process eventually decides.

We now define how to measure the *time* an execution takes.[4]

We assign times to events in a schedule subject to the following constraints: (a) the time assigned to the first event of any process is at most 1, (b) the time between two events of the same process is at most 1, and (c) times are nondecreasing and, if the execution is infinite, unbounded. The time of a finite schedule σ is the largest real time that can be assigned to the last event in the schedule; denote this by *time*(σ). The time between two events in a schedule is the largest amount of real time that can elapse between these two events under any time assignment to this schedule. We define the time taken by an execution α to be the time taken by the associated schedule, and denote this time by *time*(α). (This definition follows [Lynch and Fischer, 1981; Peterson and Fischer, 1977].)

An equivalent definition (cf. [Arjomandi et al., 1983]) is obtained by externally partitioning the computation into minimal rounds: A *round* is any sequence of events such that every process takes a step at least once in the sequence. A *minimal round* is a round such that no proper prefix of it is a round. Every sequence of events can be uniquely partitioned into minimal rounds.[5] The *time* for an execution is defined to be the number of segments in the unique partition into minimal rounds. (This is the definition introduced in [Lamport, 1976; 1977] and called the *round complexity* in [Cole and Zajicek, 1989].)

The *running time for* p_i *in an execution* of an algorithm A is defined to be the time associated with the shortest finite prefix of this execution in which p_i is in a decision state (∞, if there is no such prefix). The *time complexity of an algorithm* A is the supremum of the running times over all failure-free executions of A and all processes p_i.

Note that our definition of running time applies only to failure-free executions. It is possible to extend this definition in a natural way to executions where some processes fail; for example, by explicitly modeling failure events and excluding failed processes from the requirement to take steps. In this paper, however, we concentrate on the behavior of the algorithm in the "best case," where no failures occur, and measure running time only in failure-free executions.

We conclude this section with some useful notation. Let X be a set of real numbers. Define *range*(X) to be the interval $[\min_{x \in X} x, \max_{x \in X} x]$ if X is

[4] These definitions can also be formalized in the timed automaton model [Attiya and Lynch, 1994; Merritt et al., 1988].

[5] Except, possibly, for the last segment.

nonempty and $\varnothing$, otherwise. Define *diam*(X) to be $\max_{x_1, x_2 \in X}|x_1 - x_2|$ if X is nonempty and 0, otherwise. Note that if X is nonempty, then *diam*(X) is the length of the interval *range*(X). If X is nonempty, then

$$\mathrm{mid}(X) = \frac{\min_{x \in X} x + \max_{x \in X} x}{2}.$$

3. *Basic Solutions to the Approximate Agreement Problem*

3.1. The Approximate Agreement Problem. We start by defining the *approximate agreement* problem and describing non-wait-free and wait-free algorithms to solve it. In the approximate agreement problem, processes start with real-valued inputs, $x_0, \dots, x_{n-1}$, and a constant $\epsilon > 0$ (the same ϵ for all processes); all nonfaulty processes are required to decide on real-valued outputs $y_0, \dots, y_{n-1}$, such that the following conditions hold:

Agreement. For any i, j, $|y_i - y_j| \leq \epsilon$, and
Validity. For any i, $y_i \in$ *range*$(\{x_0, \dots, x_{n-1}\})$.

3.2. Constant Time Waiting Solution. This problem has a simple $O(1)$ time non-wait-free solution, described in Figure 1. Process p_0 maintains a single-writer multi-reader atomic register, V_0, to which it writes its input value as soon as it starts the algorithm. All processes wait until V_0 is set to a value that is not $\perp$ and decide on this value. In the code, any assignment to a shared variable implies a write, and a reference to the value of a shared variable implies a read. Uppercase variables denote shared variables, while all lowercase variables are local. In this algorithm, the values returned in the **return** statements are the decision values. Later in the paper, we use this algorithm as a "subroutine" in our main algorithm; then the values returned in the **return** statements will not be the final decision values. Similar conventions hold for later algorithms in the paper. We have:

Theorem 3.1. *Procedure* wait-approx *is a non-wait-free algorithm for the approximate agreement problem whose running time is* $O(1)$.

3.3. Inefficient Wait-Free Solution. We next present a wait-free algorithm for approximate agreement. In addition to demonstrating that a wait-free solution exists for this problem, this algorithm will also be used as a "building block" in the construction of a more efficient algorithm, in Section 6.

Let us begin by outlining a simple variant of the algorithm for the case of two processes. Each of the processes p_i, $i \in \{0, 1\}$ has a register that it can write and the other can read. Here and elsewhere, we let $\bar{\imath}$ denote the index of the other process, that is, $\bar{\imath} = 1 - i$. Due to the asynchrony in the system, it is impossible to have processes agree on one of the input values (see [Dolev et al., 1987], [Fischer et al., 1985], and [Loui and Abu-Amara, 1987]). Thus, our algorithm has them gradually converge from the input values x_0 and x_1 to values that are only ϵ apart. A process p_i repeatedly does the following: It writes its value v_i (initially the input value x_i) into its register, and then reads $p_{\bar{\imath}}$'s register. If p_i reads $\perp$ from $v_{\bar{\imath}}$, it must decide on its own value, since it can never know when $p_{\bar{\imath}}$ will write its input value (if at all, because $p_{\bar{\imath}}$ could have failed before writing). If p_i reads a non-$\perp$ value from $v_{\bar{\imath}}$, it checks whether or not $|v_{\bar{\imath}} - v_i| \leq \epsilon$. If it is, p_i decides on its own value. If not, p_i sets v_i to be $(v_i + v_{\bar{\imath}})/2$ and repeats.

```
function wait-approx($x$) returns real;
    begin
        $V_0 = x$;
        return $x$;
    end;
Process $p_0$
```

```
function wait-approx($x$) returns real;
    begin
        repeat until $V_0 \neq \perp$,
        return $V_0$;
    end;
Process $p_i$, $i \neq 0$
```

FIG. 1. Fast non-wait-free n-process approximate agreement.

Due to asynchrony, processes do not necessarily converge "directly" to a value. Rather, the following type of scenario is possible: $p_{\bar{i}}$, having previously written $v_{\bar{i}}$, reads p_i's current value v_i, and is delayed just before writing $(v_i + v_{\bar{i}})/2$ to its register; then p_i repeatedly reads and writes, cutting the interval in half till its value is very close to $v_{\bar{i}}$; finally, $p_{\bar{i}}$ completes the write of $(v_i + v_{\bar{i}})/2$ to its register, so that in fact, p_i has moved "too far" towards $p_{\bar{i}}$'s old value. This can repeat itself again and again. However, it can easily be seen that in every such step of $O(1)$ time (in which both p_i and $p_{\bar{i}}$ perform a read and a write), the diameter of the proposed values, $|v_i - v_{\bar{i}}|$, is cut by at least a half, and so the values converge in

$$O\left(\log\left(\frac{x_i - x_{\bar{i}}}{\epsilon}\right)\right)$$

time.[6] The algorithm is wait-free, since each process can reach a decision independently of the other taking steps.

The algorithm for $n > 2$ processes is of the same flavor, but uses more complicated mechanisms to synchronize among processes. It uses ideas similar to those used in the randomized consensus algorithm of [Aspnes and Herlihy, 1990]. The computation proceeds in (asynchronous) phases; in each phase, each process suggests a possible decision value. In a manner similar to that of the two process scheme above, the range of suggestions shrinks by a constant factor at each phase, until after

$$O\left(\log\left(\frac{\operatorname{diam}(\{x_0, \ldots, x_{n-1}\})}{\epsilon}\right)\right)$$

phases it becomes small enough to allow processes to decide. Because there may be more than two processes, a problem may arise in the case of an execution in which certain slow processes temporarily stop taking steps (i.e., cease advancing in phases), while others (possibly more than one) continue to advance, and then those slow processes resume taking steps again. The algorithm must allow the fast processes to coordinate a decision, while at the same time guaranteeing that the ones that are temporarily slow will converge to the same decision once they resume activity. The key idea in achieving this task is to allow fast processes that have converged to approximately the same suggested value, and are ahead of all processes with different suggestions by at least two phases, to decide. As will be shown, it can be guaranteed that the processes at lower phases will accept this decision value.

[6] Here, and in the rest of the paper, we use a truncated log function whose value is always at least one.

```
shared var
        $S$ : snapshot object $[1..n]$ of array $[1..]$ of real;

function wait-free-approx($x, \varepsilon$) returns real;
        begin
1:          $phase := 1$;
            repeat forever
2:            update($S_i[phase] := x$ );
3·            $s :=$ scan($S$);
4:            $max\text{-}phase := \max_{0 \leq j \leq n-1}\{|s_j|\}$;          /* $phase \leq max\text{-}phase$ */
5:            if $phase = max\text{-}phase$ and $phase \geq 2$
                    and $s_j[r] \in n_\varepsilon[x]$
                        for all $j$ and all $r \geq phase - 1$ such that $s_j[r]$ is defined
                then return $x$;
6.              else $r := \min\{phase + 1, max\text{-}phase\}$;
7:                  $x :=$ mid($\{s_j[r] : |s_j| \geq r\}$);          /* This set is not empty. */
8:                  $phase := phase + 1$;
              fi;
            end repeat
        end;
```

FIG. 2. Slow wait-free n-process approximate agreement—Code for process i.

The algorithm appears in Figure 2. The inputs to each process p_i are real numbers x_i and ϵ.[7] For a real number x, define $n_\epsilon(x)$, the *ϵ-neighborhood* of x, to be $[x - \epsilon, x + \epsilon]$. The algorithm employs a single-writer atomic snapshot object S as a basic memory primitive. Informally, this is a data structure partitioned into n segments S_i, each of which can be updated (written) by its "owner" process p_i, and all of which can be scanned (read) by any given process in one atomic operation. Each process p_i can thus perform an update operation on S_i, replacing all or part of the contents of S_i with a new value, or a scan operation on S, returning an "instantaneous" view of the contents of all segments of S. (More precise specifications and implementations of snapshot objects from single-writer multi-reader atomic registers can be found in [Afek et al., 1993] and [Anderson, 1990].)

For each process p_i, its segment of S is an array $S_i[1\,..]$ that in any state contains a finite sequence of reals—its suggestions at different phases—indexed by phase number. Initially, each sequence is λ, the empty sequence. At each phase, after updating (writing) a suggestion to its array (Line 2), a process p_i reads the arrays of all processes (Line 3), obtaining their suggestions for all phases.[8] If p_i is at the maximum phase and all the suggestions by other processes for its phase, or the phase before it, are within ϵ of its latest suggestion, then p_i decides on its latest suggestion (Lines 4–5). Otherwise (Lines 6–8), p_i advances to the next phase taking as its new suggestion the midpoint of all the suggestions at the next phase if there are any, or of its current phase if there are none.

[7] Although ϵ is described as a parameter, it is assumed that all processes have exactly the same value of ϵ.

[8] Though one can devise algorithms that do not require a process to maintain suggestions for all past phases (cf. [Attiyah et al., 1989]), we have chosen to maintain all suggestions in order to simplify the exposition and proofs.

We now present the correctness proof for this algorithm. Since the only shared data structure used by the algorithm is the atomic snapshot object S, an execution of the algorithm can be viewed as a sequence of primitive atomic operations that are updates and scans of S. Let α be any execution, and let $r \geq 1$ be a phase number.

For any process $j \in \{0, \ldots, n-1\}$ and any execution α, define $S_j^{\alpha}[r]$ to be the value written by p_j to $S_j[r]$ in α ($\perp$ if there is no such value). Note that this value is uniquely defined. Define $S^{\alpha}[r]$ to be $\{S_j^{\alpha}[r] \neq \perp : j \in \{0, \ldots, n-1\}\}$. The following is immediate:

LEMMA 3.2. *Let α be an execution and α' be a finite prefix of α. Then $S^{\alpha'}[r] \subseteq S^{\alpha}[r]$, for every $r \geq 1$.*

Throughout the proofs in this paper, a subscript i for a procedure denotes invocation by process p_i; similarly, a subscript i for a local variable name denotes the copy of this variable at process p_i. A process p_i is said to be *in phase r* if $phase_i = r$. Denote by scan_i^r the scan performed by p_i at phase r, and by $\mathsf{update}_i^r(x)$ the update by p_i at phase r. Note that, for $r \geq 2$, the scan performed before writing a suggestion for phase r is denoted scan^{r-1}.

For a finite or infinite execution α and $r \geq 1$, denote

$$mids(\alpha, r) = \{\mathrm{mid}(S^{\alpha'}[r]) : \alpha' \text{ is a prefix of } \alpha \text{ and } S^{\alpha'}[r] \text{ is nonempty}\},$$

that is, the set of midpoints of all the sets of suggestions for phase r at earlier points of α. The next lemma is the key for proving that the algorithm is wait-free. It will be used later, in Corollary 3.7, to show that the range of suggestions decreases by a constant factor with each phase. Intuitively, it states that any suggestion for phase r must be in the range of the midpoints of all the sets of suggestions for phase $r-1$ at earlier points in the execution.

LEMMA 3.3. *For any finite execution α and phase $r \geq 2$, $range(S^{\alpha}[r]) \subseteq range(mids(\alpha, r-1))$.*

PROOF. By induction on the length of the execution. The basis holds vacuously.

For the induction step, the interesting case is when α ends with $\mathsf{update}_i^r(x)$, for some i, where $x = S_i^{\alpha}[r]$. Then scan_i^{r-1} appears in α. Let α' be the shortest prefix of α that includes scan_i^{r-1}. Note that α' is a proper prefix of α.

Let r' be the largest phase number read in scan_i^{r-1}. Since process p_i reads its own sequence, $r' \geq r-1$. If $r' = r-1$, then the code implies that x is the result of the calculation in Line 7, and hence x is the midpoint of $S^{\alpha'}[r-1]$, which suffices. If $r' \geq r$, then, by the code, $x = \mathrm{mid}(S^{\alpha'}[r])$. By the induction hypothesis on α', $range(S^{\alpha'}[r]) \subseteq range(mids(\alpha', r-1))$. Thus,

$$x = \mathrm{mid}(S^{\alpha'}[r]) \in range(S^{\alpha'}[r]) \subseteq range(mids(\alpha', r-1))$$
$$\subseteq range(mids(\alpha, r-1)),$$

as needed. □

Since $range(mids(\alpha, r-1)) \subseteq range(S^{\alpha}[r-1])$, we have:

COROLLARY 3.4. *For any finite execution α and phase $r \geq 2$, $range(S^{\alpha}[r]) \subseteq range(S^{\alpha}[r-1])$.*

For the rest of the proof, we fix some infinite execution β of the algorithm. The following lemmas are stated with respect to β. The following is a corollary of Lemma 3.3.

COROLLARY 3.5. *For any phase* $r \geq 2$, $\mathit{range}(S^{\beta}[r]) \subseteq \mathit{range}(\mathit{mids}(\beta, r-1))$.

The next lemma states that the diameter of all the possible midpoints of the suggestions in phase r is at most half the diameter of all the suggestions for phase r.

LEMMA 3.6. *For any phase* $r \geq 1$, $\mathit{diam}(\mathit{mids}(\beta, r)) \leq \frac{1}{2}\mathit{diam}(S^{\beta}[r])$.

PROOF. If $\mathit{mids}(\beta, r)$ is empty, then $\mathit{diam}(\mathit{mids}(\beta, r)) = 0$ and the claim follows immediately, so assume that $\mathit{mids}(\beta, r)$ is nonempty. Let α' and α'' be two prefixes of β such that $S^{\alpha'}[r]$ and $S^{\alpha''}[r]$ are nonempty. It suffices to show that $|\mathrm{mid}(S^{\alpha''}[r]) - \mathrm{mid}(S^{\alpha'}[r])| \leq \frac{1}{2}\mathit{diam}(S^{\beta}[r])$. Without loss of generality, suppose α'' is a prefix of α'. By Lemma 3.2, $S^{\alpha''}[r] \subseteq S^{\alpha'}[r] \subseteq S^{\beta}[r]$. Suppose first that $\mathrm{mid}(S^{\alpha'}[r]) \leq \mathrm{mid}(S^{\alpha''}[r])$. Thus, $\mathrm{mid}(S^{\alpha'}[r]) \leq \mathrm{mid}(S^{\alpha''}[r]) \leq \max(S^{\alpha''}[r]) \leq \max(S^{\alpha'}[r])$. Hence

$$|\mathrm{mid}(S^{\alpha''}[r]) - \mathrm{mid}(S^{\alpha'}[r])| \leq \frac{1}{2}\mathit{diam}(S^{\alpha'}[r]) \leq \frac{1}{2}\mathit{diam}(S^{\beta}[r]),$$

as needed. A symmetric argument applies if $\mathrm{mid}(S^{\alpha''}[r]) > \mathrm{mid}(S^{\alpha'}[r])$. □

The following lemma guarantees that suggestions become closer with each phase; it will be used together with Lemma 3.9 to ensure wait-freedom.

LEMMA 3.7. *For any phase* $r \geq 2$, $\mathit{diam}(S^{\beta}[r]) \leq \frac{1}{2}\mathit{diam}(S^{\beta}[r-1])$.

PROOF. By Corollary 3.5, $\mathit{range}(S^{\beta}[r]) \subseteq \mathit{range}(\mathit{mids}(\beta, r-1))$. Thus,

$$\begin{aligned} \mathit{diam}(S^{\beta}[r]) &\leq \mathit{diam}(\mathit{mids}(\beta, r-1)) \\ &\leq \frac{1}{2}\mathit{diam}(S^{\beta}[r-1]) \quad \text{by Lemma 3.6.} \end{aligned}$$ □

LEMMA 3.8. *If some process returns x in phase r and* $y \in S^{\beta}[r]$, *then* $y \in n_{\epsilon}(x)$.

PROOF. Assume p_i returns x in phase r. By the code, it must be that $r \geq 2$. Assume, by way of contradiction, that there exists at least one process with a suggestion for phase r that is not in $n_{\epsilon}(x)$. Let p_j be such a process with the property that scan_j^{r-1} is the earliest among the scan^{r-1} operations of these processes, and let α be the shortest prefix of β that includes scan_j^{r-1}. Let $y = S_j^{\beta}[r]$; by assumption, $y \notin n_{\epsilon}[x]$.

By the way p_j was chosen, there is no $\mathsf{update}_{j'}^{r}(y')$, with $y' \notin n_{\epsilon}(x)$ in α; thus, $\mathit{range}(S^{\alpha}[r]) \subseteq n_{\epsilon}[x]$. Let r' be the maximum phase number read in scan_j^{r-1}. If $r' \geq r$, then the minimum determined in Line 6 of p_j's code for phase $r-1$ is equal to r. Thus, in this case, the only values considered in determining $S_j^{\beta}[r]$ are values in $S^{\alpha}[r]$. Since $\mathit{range}(S^{\alpha}[r]) \subseteq n_{\epsilon}[x]$, it follows that p_j's suggestion for phase r is in $n_{\epsilon}(x)$. This is a contradiction, and hence $r' \leq r-1$. Since process p_j reads its own sequence, $r' = r-1$.

The fact that $r' = r-1$ also implies that scan_j^{r-1} precedes $\mathsf{update}_i^{r}(x)$. Let α' be the shortest prefix of β that includes scan_i^{r}. Since $\mathsf{update}_i^{r}(x)$ precedes scan_i^{r}, it follows that scan_j^{r-1} precedes scan_i^{r}, that is, α is a prefix of α'.

Since process p_i returns in phase r, it follows from the code that $range(S^{\alpha'}[r - 1]) \subseteq n_\epsilon[x]$. Since $r - 1$ is the maximum phase number read in scan_i^{r-1}, it follows that $y = \mathrm{mid}(S^\alpha[r - 1]) \in range(S^\alpha[r - 1])$. However, by Lemma 3.2, $S^\alpha[r - 1] \subseteq S^{\alpha'}[r - 1]$, and thus $y \in n_\epsilon(x)$, a contradiction. □

LEMMA 3.9. *For any phase $r \geq 1$, if $diam(S^\beta[r]) \leq \epsilon$, then every nonfaulty process returns no later than phase $r + 1$.*

PROOF. From the code it follows that every nonfaulty process either returns or reaches phase r + 1. If $diam(S^\beta[r]) \leq \epsilon$, it follows from Corollary 3.4 that $diam(S^\beta[r + 1]) \leq \epsilon$.

The proof proceeds by induction on the order in which processes perform scan^{r+1}. For the base case, let p_i be the first process to perform scan^{r+1}. Clearly, p_i has $phase_i = r + 1 = max\text{-}phase$, and by assumption $r + 1 \geq 2$. Also, $diam(S^\beta[r])$ and $diam(S^\beta[r + 1])$ are less than or equal to ϵ, and thus, p_i will pass the test in Line 5 and will return in phase $r + 1$. The induction step is similar, and uses the fact that so far no process has advanced beyond phase $r + 1$ to show that any process that reaches phase $r + 1$ passes the test in Line 5 and returns in phase $r + 1$. □

Thus, we can prove:

THEOREM 3.10. *Procedure* wait-free-approx *is a wait-free algorithm for the approximate agreement problem whose running time on input $\langle x_0, \ldots, x_{n-1} \rangle$ is at most*

$$O\left(n^2 log\left(\frac{diam(\{x_0, \ldots, x_{n-1}\})}{\epsilon}\right)\right).$$

PROOF. The validity condition clearly holds, since processes decide only on their suggestions and these are always within the range of the inputs (Corollary 3.4).

To show agreement, assume that r is the minimum phase in which some process returns, and let p_i be a processes that returns x in phase r. By Lemma 3.8, the suggestions of all processes for phase r are in $n_\epsilon(x)$. By Corollary 3.4, the same is true for phase $r + 1$. By Lemma 3.9, all nonfaulty processes return no later than phase $r + 1$, and thus, all nonfaulty processes return either in phase r or in phase $r + 1$. Since processes return only their suggestions, all returned values are in $n_\epsilon(x)$, as needed.

Since the diameter of suggestions decreases by a factor of two with each phase (by Lemma 3.7), it will eventually be less than or equal to ϵ and, by Lemma 3.9, each nonfaulty process will eventually decide. This guarantees wait-freedom.

To show the time bound, notice that, by Lemma 3.7, after

$$O\left(\log\left(\frac{diam(\{x_0, \ldots, x_{n-1}\})}{\epsilon}\right)\right)$$

phases, the diameter of the set of suggestions will be at most ϵ. By Lemma 3.9, all nonfaulty processes will return by the next phase. The time it takes a process to execute each phase is bounded from above by the number of operations it executes. Using the implementation of atomic snapshots from [Afek et al., 1993], this is bounded by $O(n^2)$. □

Since the input range is not bounded and ϵ may be arbitrarily small, the running time of the algorithm as a function of n is actually unbounded.

4. *The Bias Function*

The algorithms in Sections 5 and 6 return a decision value by performing a calculation based on an input value and a counter for each process. We name the calculated function bias, as the returned decision value is biased towards (i.e., is closer to) the input value associated with the process having the largest counter. Before presenting the algorithms, we present the function and explain its properties. The proofs of these properties are purely arithmetic, involving no arguments about synchronization between processes, and have therefore been deferred to Section 9.

In order to understand the nature of the calculation performed by the bias function, we briefly explain the structure of the algorithms using it. The new algorithms are conceptually based on the following high-level two-process algorithm. Process p_1 (similarly p_0), knowing only its own input value v^1, will repeatedly take incremental steps of size ϵ, starting at 0 and ending upon reaching the value v^1, unless it reads that the other process p_0 has also moved. In the former case it decides on v^1, and in the latter case its decision value is a function of the relative number of incremental steps both processes managed to take before each noticed the other had moved. However, since in either case process p_1's decision must be guaranteed to be in *range*$(\{v^0, v^1\})$, it cannot just be a value in the interval *range*$(\{0, v^1\})$. This is the purpose of the function bias. It provides a mapping from the processes' incremental walks in the intervals *range*$(\{0, v^0\})$ and *range*$(\{0, v^1\})$, respectively, to walks of proportional length in the allowed *range*$(\{v^0, v^1\})$. The code of bias appears in Figure 3. The function takes as inputs two real number values v^0 and v^1, two associated counters, c^0 and c^1 (integers denoting the number of incremental steps each process p_0 or p_1 took), and ϵ.

An example of the translation defined by bias is given in Figure 4 for the case $0 < v^0 < v^1$. Assume p_0 traverses a distance of length $c^0 \cdot \epsilon$ away from 0 towards v^0, and p_1 a distance of length $c^1 \cdot \epsilon$ away from 0 towards v^1. The bias function maps the respective distances of length $c^0 \cdot \epsilon$ and $c^1 \cdot \epsilon$ (within the interval $[-v^0, v^1]$), into distances of proportional length in the interval $[v^0, v^1]$. The starting point 0 in $[-v^0, v^1]$, is replaced by the point *new*-0 in $[v^0, v^1]$, which depends only on v^0 and v^1. The returned decision value is then the point associated with the larger counter (larger traversed distance).

We now introduce several lemmas that formally outline the properties of the bias function and on which the correctness proofs of the algorithms in the sequel will be based. The first is a rather simple statement, namely, that the returned value of any call to bias is in *range*$(\{v^0, v^1\})$.

LEMMA 4.1. *Let c^0, c^1 be nonnegative integers, and v^0, v^1, ϵ be real numbers, with $\epsilon > 0$. Then* bias$(v^0, v^1, c^0, c^1, \epsilon) \in$ *range*$(\{v^0, v^1\})$.

The next three lemmas deal with an additional property required of the bias function: that the values returned by different calls to bias always be approximately the same, even if the counter parameter values or the real parameter values used in these calls, are slightly different. The first lemma states that applying bias in a case where counter c^i is large yields a value close to v^i.

```
function bias ($v^0$,$v^1$,$c^0$,$c^1$,$\varepsilon$) returns *real*;
    begin
        if $v^0 = v^1 = 0$ then return 0
                else if $c^0 < c^1$ then return $v^1 + \frac{v^0 - v^1}{|v^0|+|v^1|}(|v^1| - \min\{c^1\varepsilon, |v^1|\})$
                          else return $v^0 + \frac{v^1 - v^0}{|v^0|+|v^1|}(|v^0| - \min\{c^0\varepsilon, |v^0|\})$
        fi;
    end;
```

FIG. 3. The bias function—Code for process p_i.

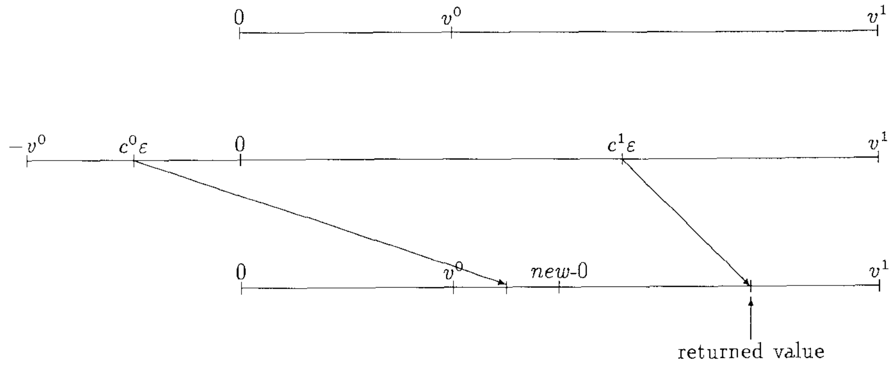

FIG. 4. The bias mapping.

LEMMA 4.2. *Let c^0, c^1 be nonnegative integers, and v^0, v^1, ϵ, m be real numbers, $\epsilon > 0$, $m \geq 0$.*

(1) *Suppose $c^1 > c^0$ and $|v^1|/\epsilon - m \leq c^1$. Then $|\text{bias}(v^0, v^1, c^0, c^1, \epsilon) - v^1| \leq m\epsilon$.*

(2) *Suppose $c^0 \geq c^1$ and $|v^0|/\epsilon - m \leq c^0$. Then $|\text{bias}(v^0, v^1, c^0, c^1, \epsilon) - v^0| \leq m\epsilon$.*

The next lemma shows that the results of two calls to bias with approximately the same values (in a sense made precise by the lemma) for c^0, c^1, and the same v^0, v^1, ϵ, are approximately the same.

LEMMA 4.3. *Let $c_0^0, c_0^1, c_1^0, c_1^1$ be nonnegative integers, and v^0, v^1, ϵ, m be real numbers, $\epsilon > 0$ and $m \geq 0$. Suppose $min\{c_0^0, c_0^1\} = min\{c_1^0, c_1^1\} = 0$ and $|c_0^0 - c_1^0| + |c_0^1 - c_1^1| \leq m$. Then*

$$\left|\text{bias}(v^0, v^1, c_0^0, c_0^1, \epsilon) - \text{bias}(v^0, v^1, c_1^0, c_1^1, \epsilon)\right| \leq m\epsilon.$$

The last lemma in this section states that applying bias, this time to real numbers v^0 and v^1 that are approximately (to within δ) the same, yet with exactly the same counters c^0, c^1, and ϵ, results in values that are approximately the same.

LEMMA 4.4. *Let c^0, c^1 be nonnegative integers, and $v_0^0, v_0^1, v_1^0, v_1^1$, ϵ, δ be real numbers, with $\epsilon > 0$, $\delta \geq 0$. Suppose $|v_0^0 - v_1^0| \leq \delta$ and $|v_0^1 - v_1^1| \leq \delta$. Then*

$$\left|\text{bias}(v_0^0, v_0^1, c^0, c^1, \epsilon) - \text{bias}(v_1^0, v_1^1, c^0, c^1, \epsilon)\right| \leq 6\delta.$$

5. *Fast Two-Process Approximate Agreement*

We now show that, for two processes, there exists a wait-free approximate agreement algorithm whose time complexity is constant; that is, it does *not* depend on the range of input values or ϵ. The n-process algorithm presented in Section 6, when specialized to the case $n = 2$, also yields a (somewhat larger) constant time complexity. We present the two-process algorithm because we believe its simplicity will help the reader develop an intuition for the ideas that will be later used in the general algorithm.

5.1. INFORMAL DESCRIPTION. The key ideas underlying this algorithm are as follows: A process, p_i, running on its own, can assume that either it is running very fast (and not much time has elapsed), or the other process, $p_{\bar{i}}$, has failed. Thus, p_i may take an unlimited number of steps without degrading the time complexity for failure-free executions, as long as $p_{\bar{i}}$ does not perform any steps. Of course, if $p_{\bar{i}}$ does not take any steps at all, then, in order to guarantee the wait-free property, p_i must *eventually* decide (unilaterally) on its own value. In this case, in order to guarantee correctness, it is necessary that if and when $p_{\bar{i}}$ does appear, it must be able to know, just by reading p_i's registers, what p_i has decided. However, an inherent difficulty of programming asynchronous systems is that, due to the uncertainty of interleaving, at least one process p_i has an "uncertainty of one step," namely, it cannot tell whether $p_{\bar{i}}$ read the value written in p_i's latest write or the value written in p_i's preceding write. A two-process solution that halves the distance between the suggested values is thus of no use, since the "uncertainty of one step" can cause processes to decide on values that are more than ϵ apart. Our solution is to have a process change its suggestions gradually with each step, more precisely, by an amount less than ϵ, so that the "uncertainty of one step" will result only in ϵ inaccuracy in the decision value.

5.2. THE ALGORITHM. The code for process p_i is given in Figure 5. Each process p_i, $i \in \{0, 1\}$ maintains a *single-writer multi-reader atomic register* with two fields: V_i—the input value, a real number, and C_i—the counter, an integer. Each process starts by writing its input and initializing a counter in the shared memory (Line 1 in increase-counter). It then keeps incrementing this counter until either it has taken a number of steps proportional to the absolute value of its input, or the other process has taken a step, whichever happens first (Line 2 of increase-counter). When the process stops, it collects all the C and V values and applies the function bias to get a decision value. As described in the former section, the decision is within the input range and biased towards the input value of the process with the larger counter. In particular, if a process runs to completion without observing the other process, it decides on its own input value. In the following subsection we show that the discrepancy in the reading of the counters among the two processes is at most 1, and thus, based on the properties of the bias function, the decisions based on the values of the counters will differ by at most ϵ.

5.3. CORRECTNESS PROOF. An execution of the algorithm can be viewed as a sequence of primitive atomic operations that are reads and writes of atomic registers (and may include changing local data). Fix some execution α of the algorithm. All lemmas in the rest of this section are stated with respect to α. In

```
shared var
        $\langle V, C \rangle$· array [0,..,1] of single writer register with
                fields V: real and C· integer;

function fast-2-approx $(x, \varepsilon)$ returns real;
1           increase-counter($x, \frac{|x|}{\varepsilon}$);
2·          $\langle v^0, v^1, c^0, c^1 \rangle := \langle V_0, V_1, C_0, C_1 \rangle$;
3           if $c^{\bar{i}} = \perp$ then return $v^i$
4·                  else return bias $(v^0, v^1, c^0, c^1, \varepsilon)$;
        end;

procedure increase-counter $(v, max)$;
1:          $\langle V_i, C_i \rangle := \langle v, 0 \rangle$;
2:          while $C_{\bar{i}} = \perp$ and $C_i < max$ do $C_i := C_i + 1$ od;
        end;
```

FIG. 5. Fast wait-free two-process approximate agreement—Code for process p_i.

the rest of this section, a value of $\perp$ is treated as -1 in arithmetic expressions. The next lemma shows a crucial property regarding how close the counter values collected by two processes are.

LEMMA 5.1. *Assume p_0 and p_1 return from* fast-2-approx. *Let $i \in \{0, 1\}$, and let c_i and $c_{\bar{i}}$ be the values of C_i read by p_i and $p_{\bar{i}}$, respectively, in Line 2 of* fast-2-approx. *Then $c_i \neq \perp$ and $c_i - 1 \leq c_{\bar{i}} \leq c_i$.*

PROOF. Since p_i returns, it must be that p_i writes to C_i. Let π_i be the last write by p_i to C_i in α. Since increase-counter returns after the last write to C_i and by definition p_i is the only one to modify C_i, it follows that c_i is the value written to C_i in π_i. Since p_i writes the value c_i to C_i, we have that $c_i \neq \perp$.

Let $\phi_{\bar{i}}$ be the read by $p_{\bar{i}}$ of C_i in Line 2 of fast-2-approx. Note that $c_{\bar{i}}$ is the value returned in $\phi_{\bar{i}}$. Since the read of C_i is atomic, it is clear that $c_{\bar{i}} \leq c_i$. We now show that $c_i - 1 \leq c_{\bar{i}}$.

If $c_i = 0$, then since $c_{\bar{i}} \leq c_i$, $c_{\bar{i}} \in \{\perp, 0\}$; since $\perp$ is mapped to -1, the claim follows. So assume $c_i > 0$. Let π'_i be the penultimate write by p_i to C_i, writing $c_i - 1$. Let ϕ_i be the latest read of $C_{\bar{i}}$ by p_i that precedes π_i; note that π'_i precedes ϕ_i. Since p_i performs at least one additional write after π'_i, it must be that the value read in ϕ_i is $\perp$. Let $\pi_{\bar{i}}$ be the write of 0 by $p_{\bar{i}}$ to $C_{\bar{i}}$ in α. From the code, it follows that $\pi_{\bar{i}}$ preceds $\phi_{\bar{i}}$. Since the value read in ϕ_i is $\perp$, and because $C_{\bar{i}}$ is written and read atomically, it follows that ϕ_i precedes $\pi_{\bar{i}}$. From the above we thus have that π'_i precedes ϕ_i which precedes $\pi_{\bar{i}}$ which precedes $\phi_{\bar{i}}$. Thus, the write π'_i precedes the read $\phi_{\bar{i}}$, and it follows that $c_i - 1 \leq c_{\bar{i}}$. □

We can now prove that the algorithm satisfies the agreement property:

LEMMA 5.2. *For processes p_0 and p_1, if* fast-2-approx$_0$ *returns y_0 and* fast-2-approx$_1$ *returns y_1, then $|y_0 - y_1| \leq \epsilon$.*

PROOF. The proof of this lemma is separated into two cases. In one case, we apply Lemma 4.2. In the other case, we show that the sum of the differences between the values of c^0 and c^1 used by p_0 and by p_1 is at most 1, and appeal to Lemma 4.3. The details follow.

Denote by π_i the first write by p_i to C_i, writing 0, for $i \in \{0, 1\}$. Since both processes decide, both π_0 and π_1 must appear in α. Assume, without loss of generality, that π_0 precedes π_1. (The other case is symmetric.) Assume that process p_0 reads $\langle v_0^0, v_0^1, c_0^0, c_0^1 \rangle$ in Line 2 before deciding, and that process p_1 reads $\langle v_1^0, v_1^1, c_1^0, c_1^1 \rangle$ in Line 2 before deciding. Note that, since p_i first writes 0 to C_i and then reads C_i, it must be that $c_i^i \geq 0$, for $i \in \{0, 1\}$.

Let ϕ be any read of C_0 by p_1, returning some value z. The code of the algorithm implies that π_1 precedes ϕ. Since π_0 precedes π_1, π_0 precedes ϕ. Since reads and writes to C_0 are atomic operations, this implies that $z \geq 0$. This implies, in particular, that $c_1^0 \geq 0$, and thus, fast-2-approx$_1$ returns in Line 4. In addition, this also implies that p_1 will not increase C_1 beyond 0, and thus, since reads and writes to C_1 are atomic, $c_1^1 = 0$ and $c_0^1 \in \{\perp, 0\}$. We separate the rest of the proof into two cases:

Case 1. $c_0^1 = \perp$. In this case, fast-2-approx$_0$ returns $v_0^0 = x_0$ in Line 3. The code of increase-counter implies that $|x_0|/\epsilon \leq c_0^0$. From Lemma 5.1, since $c_1^0 \geq 0$, it follows that $|x_0|/\epsilon - 1 \leq c_1^0$. Also, $v_1^0 = x_0$. Since $c_1^0 \geq 0 = c_1^1$, we can apply Lemma 4.2(2) with $m = 1$ and get that $|\mathsf{bias}(v_1^0, v_1^1, c_1^0, c_1^1, \epsilon) - v_0^0| \leq \epsilon$, as needed.

Case 2. $c_0^1 = 0$. Then, fast-2-approx$_0$ returns in Line 4 and $v_0^1 = v_1^1$. We have that $\min\{c_0^0, c_0^1\} = c_0^1 = 0$ and $\min\{c_1^0, c_1^1\} = c_1^1 = 0$. Also, $|c_0^0 - c_1^0| + |c_0^1 - c_1^1| = |c_0^0 - c_1^0| \leq 1$, by Lemma 5.1. The claim follows by applying Lemma 4.3 with $m = 1$. □

We have:

THEOREM 5.3. *Procedure* fast-2-approx *is a wait-free algorithm for the two-process approximate agreement problem whose time complexity is* $O(1)$.

PROOF. Agreement follows from Lemma 5.2. It follows from the code and from Lemma 4.1 that the values returned are in the range of the original input values; hence, the validity property is satisfied. Each process p_i executes at most $O(|x_i|/\epsilon)$ steps before deciding; thus, the algorithm is wait-free. Since each process executes a constant number (i.e., independent of ϵ and the range of inputs) of steps after the other process performs its first step, the time complexity of this algorithm is $O(1)$. □

6. *Fast n-Process Approximate Agreement*

In this section, we present a fast ($O(\log n)$ time) wait-free approximate agreement algorithm for n processes. The algorithm is based on an *alternated-interleaving* method of integrating wait-free (resilient, but slow) and non-wait-free (fast, but not resilient) algorithms to obtain new algorithms that are both resilient and fast.

We begin by showing how one can reduce, in constant time, the problem of n-process approximate agreement with arbitrary input values to a special case of the problem where the set of input values is included in the union of two small intervals. We do this by performing an alternated-interleaving of a wait-free and a non-wait-free algorithm. We then show, again based on an alternated-interleaving of wait-free and non-wait-free algorithms, that n processes with values in two small intervals can "simulate," in $O(\log n)$ time, two virtual processes running the fast approximate agreement algorithm

```
type
        $group = \{a, b\}$;
shared var
        $\langle V, G, C \rangle$: array [1..n] of single writer register with
                fields V: real, G. group, and C: integer;

function fast-n-approx $(x, \varepsilon)$ returns real;
        begin
            $\langle v, g \rangle$ .= n-to-2 $(x, \varepsilon)$,
            increase-counter$(v, g, \frac{|v|}{\varepsilon/6n})$;
            $\langle \vec{v}, \vec{g}, \vec{c} \rangle$ := fast-collect $(V, G, C)$;
            if $c^{\bar{g}} = \perp$ then return $v^g$
                    else return bias$(v^a, v^b, c^a, c^b, \varepsilon/6n)$;
        end;

function n-to-2 $(x, \varepsilon)$ returns $\langle real, group \rangle$;
        begin
            $\langle v, g \rangle$ := begin-alternate
                            $\langle$wait-free-approx$(x, \varepsilon/12), a\rangle$
                    and
                            $\langle$wait-approx$(x), b\rangle$;
                    end-alternate;
            return $\langle v, g \rangle$
        end;

procedure increase-counter $(v, g, max)$;
        begin
            $\langle V_i, G_i, C_i \rangle := \langle v, g, 0 \rangle$;
            begin-alternate
                while $C^{\bar{g}} = \perp$ and $C^g < max$ do $C_i := C_i + 1$ od,
            and
                synch $(C)$;
            end-alternate;
            $T_i := true$;
        end;
```

FIG. 6. Fast wait-free n-process approximate agreement—Code for process p_i.

of Section 5, thus solving the approximate agreement problem for n processes each having one of two values. Combining the two algorithms yields an $O(\log n)$ wait-free approximate agreement algorithm.

6.1. INFORMAL DESCRIPTION. The first part of the algorithm—the one that achieves the constant-time reduction to two small intervals, is encapsulated in procedure n-to-2 (Figure 6). The idea is simple: Interleave the execution of the slow wait-free-approx procedure (of Figure 2) with that of the fast wait-approx (of Figure 1), stopping when the first of them does. The resulting algorithm is wait-free since even if $n - 1$ processes fail, wait-free-approx will terminate. It takes at most $O(1)$ time in the failure-free execution since wait-approx terminates within $O(1)$ time. However, some processes (group a) might finish the alternated execution with a value from wait-approx, while others (group b) finish with a value from wait-free-approx. Thus, this strategy does not solve the approximate agreement problem, but guarantees that the returned values are included in the union of two small intervals. More specifically, the procedure n-to-2 returns an output value v_i and a group $g_i \in \{a, b\}$ to which p_i is said to belong. It is guaranteed that output values for processes in the same group $g_i \in \{a, b\}$ are at most $\epsilon/12$ apart.

The second part of the algorithm solves n-process approximate agreement in $O(\log n)$ time, assuming that processes are partitioned into two groups with approximately the same initial value in each group. The solution is based on having the processes in group a (respectively, b) jointly simulate a virtual process p_0 (respectively, p_1) that executes the function fast-2-approx of Figure 5.

The following straightforward simulation is expressed by Lines 1–2 of the procedure increase-counter in Figure 6. The counter C_0 of fast-2-approx is replaced by a joint counter, which is defined to be the sum of local counters C_i, for all i in group a. Each step of the simulated counter C_0 is implemented by $O(n)$ steps of the joint counter for a. Each step of this joint counter is, in turn, implemented by a single step of one of the individual counters in group a. Similarly, the processes in group b simulate counter C_1 of fast-2-approx. In Line 2 of increase-counter, in order to decide on the values of the joint counters of a and b, a process reads the values of all local counters. If the counter simulated by p_i's group is not large enough and the counter simulated by the other group is $\perp$, then p_i advances the counter simulated by its group (by incrementing its local counter C_i), and repeats. Otherwise, p_i exits increase-counter.

One can see that, in an execution where processes operate synchronously, each iteration of the **while** loop in Line 2 of increase-counter has $O(n)$ time complexity since reading all memory locations to calculate the simulated counter takes $O(n)$ steps. However, one can improve the time complexity based on the following observation. If p_i ever detects that all processes have set their counters in Line 1 of increase-counter, then it knows that one of the following holds: either some process from the other group has set its local counter (and hence that group's simulated counter), to a value other than $\perp$, or the other group is empty. In the former case, the loop predicate in Line 2 must be false, while in the latter case, the final value for the other group's counter will be $\perp$. In either case, p_i can stop executing increase-counter, and be guaranteed to correctly simulate the behavior of the two-process algorithm. In order to detect in less than $O(n)$ time that all processes have set their counters, we use an $O(\log n)$ non-wait-free synch procedure, described in Section 6.3.2, whose termination ensures this condition. To achieve the better time, the algorithm alternates synch with the (wait-free) loop in Line 2 of increase-counter.

The delicate synchronization provided by synch and its effect on the rest of the algorithm guarantee that after some process exits increase-counter, individual counter values increase at most by 3. Thus, after exiting increase-counter, a process can perform an $O(\log n)$ wait-free fast-collect, described in Section 6.3, in order to collect all the values needed to decide on the returned value in Lines 3–4. The above property ensures that the simulated counter values used by different processes do not differ much.

6.2. THE ALGORITHM. The code for the algorithm is presented in Figure 6. Alternated procedures are enclosed within **begin-alternate** and **end-alternate** brackets. This construct means that the algorithm alternates strictly between executing single steps of the two alternated procedures, and terminates the first time one of the procedures terminates.[9] When an alternation is used in an assignment statement, the value assigned is the value returned by the procedure that terminates first. The algorithm uses the bias procedure of Figure 3. In addition to the shared data structures used by wait-free-approx and wait-approx, process p_i, $i \in \{0, \dots, n-1\}$, has a *single-writer multi-reader atomic register* with the following fields: V_i—the value returned in p_i's first phase; G_i—denoting the group to which p_i belongs; C_i—p_i's contribution to its group's counter; T_i—p_i's Boolean synch termination flag.

In the code for process i, we abuse notation and denote by V^g, where g is a group's name, the "group's value" calculated as follows: If $g = g_i$, then it is V_i; and if $g \neq g_i$ then it is an arbitrary V_j, such that p_j is in group g if it is non-empty, and $\perp$, otherwise. The value v^g is calculated in a similar manner from the corresponding local copies. (Recall our convention that lowercase letters stand for local variables and uppercase letters for shared variables.) When g is a group name, $\bar{g}$ denotes the other group's name, for example, if $g = a$, then $\bar{g} = b$. The notation C^g, for $g \in \{a, b\}$, stands for the sum of those C_i such that $G_i = g$ and $C_i \neq \perp$, if there is any such C_i, and $\perp$, otherwise. The value c^g is calculated in a similar manner from the corresponding local copies.

6.3. FAST INFORMATION COLLECTION AND SYNCHRONIZATION. We now present the procedures for information collection and synchronization and prove their properties.

6.3.1. *Fast Information Collection.* We start with a wait-free algorithm for *input collection*—returning the current values in the entries of an array R. The time complexity of the algorithm is $O(\log n)$.

This problem is interesting on its own as it underlies any problem of computing a function, for example, max or sum, on a set of initial values that reside in the shared memory.[10] Once a process collects all the values, computing the function can be done locally in constant time. Since $\Omega(\log n)$ is a lower bound on the time for the information collection problem (see, e.g., [Cook et al., 1986]), this implies that for problems whose output depends on all the initial values in memory, and only on them, there exists an optimally fast wait-free solution.

Our algorithm, presented in Figure 7, is a wait-free variation of the *pointer-jumping* technique used in PRAM algorithms (e.g., [Wyllie, 1979]). Think of the registers R_i, $i \in \{1 \dots n\}$, as being arranged in a circle (hence, indices are modulo n). To achieve logarithmic time complexity, a process writes in the register R_i not only its value, but also all other values it has learned about. Proceeding in a cyclic fashion, p_i first reads R_{i+1}. If R_{i+1} has already collected, say, 3 values R_{i+1}. R_{i+4}, then p_i next reads R_{i+5}. It continues in this fashion until it has transitively collected values from all n registers.

[9] We remark that this is just a coding convenience, used to simplify the control structure of the algorithm. It is implemented locally at one process and does not cause spawning of new processes

[10] Note that these problems are very different form the *decision problems* considered until now in this paper, where inputs are local to the processes and do not reside in the shared memory.

```
type
        string = array [1..n] of register values;
shared var
        R : array [1..n] single writer register;

function fast-collect (R) returns string;
        begin
            $l := 1$;
            while $l < n$ do                          /* i knows fewer than n values. */
                $R_i$ := concatenate ($R_i$, $R_{(i+l) \bmod n}$);     /* Read what $p_{(i+l)}$ knows. */
                $l := |R_i|$;
            od;
            return truncate($R_i$, $n$);
        end;
```

FIG. 7. Fast wait-free information collection—Code for process p_i.

We use the following functions in the algorithm: For sequences R, R' and a nonnegative integer n, we define concatenate(R, R') as returning the concatenation of R' to R, and truncate(R, n) as returning the first n elements of R if $|R| > n$, and R, otherwise. The initial value $\perp$ is treated like any other value and may be returned by the algorithm for entries that have not yet been set.

Fix some execution α of the fast-n-approx algorithm. We clearly have:

LEMMA 6.1. *Assume* fast-collect$_i$ *is invoked by* p_i *in* α, *and let* α' *be the shortest prefix of* α *that includes some invocation of* fast-collect. *Then* fast-collect$_i$ *returns a vector containing, for each* p_j, *a value that appears in* R_j *at some point at or after* α'. *Moreover,* fast-collect$_i$ *returns within at most* $2n$ *steps by* p_i.

PROOF. Each iteration of the **while** loop in procedure fast-collect takes at most two steps, and the loop is executed at most n times. □

The next lemma is the crux of the time analysis for this algorithm.

For the rest of this Section, let t be the time of the last event in the shortest finite prefix of α that includes an invocation of fast-collect by every p_i, $i \in \{0, \ldots, n-1\}$, if such a prefix exists, ∞ otherwise.

LEMMA 6.2. *Assume* $t < \infty$. *For every* $i \in \{0, \ldots, n-1\}$ *and every integer* r, $0 \le r \le \lceil \log n \rceil$, $|R_i| \ge \min\{2^r, n\}$ *at time* $t + 3r$.

PROOF. The proof is by induction on r. The base case, $r = 0$, is trivial.

For the induction step, assume that $r \ge 1$. If at time $t + 3r$, $|R_i| \ge n$, the claim follows. So suppose, $|R_i| < n$ at time $t + 3r$. Then also $|R_i| < n$ at time $t + 3(r-1)$. Then by the induction hypothesis, $|R_i| \ge 2^{r-1}$ at time $t + 3(r-1)$.

By the code, there must be some time t', where $t + 3(r-1) < t' \le t + 3(r-1) + 2$, at which p_i reads some R_j. Fix j to be the index of the first such read that occurs. By the induction hypothesis, $|R_j| \ge \min\{2^{r-1}, n\}$ at time $t + 3(r-1)$. Since p_i reads R_j by time $t + 3(r-1) + 2$, the code implies that p_i subsequently writes R_i by time $t + 3r$. It follows that $|R_i| \ge 2^{r-1} + \min\{2^{r-1}, n\} \ge \min\{2^r, n\}$ at time $t + 3r$. □

In particular, at time $t + 3\lceil \log n \rceil$, we have $|R_i| \geq n$ for every i. Thus, fast-collect$_i$ returns by time $t + 3\lceil \log n \rceil$. We have:

LEMMA 6.3. *Let α' be a finite prefix of α. Assume that in α'*, fast-collect$_i$ *is invoked by p_i, for every $i \in \{0, \ldots, n-1\}$. Then for every $i \in \{0, \ldots, n-1\}$*, fast-collect$_i$ *returns within at most $O(\log n)$ time after time(α').*

6.3.2. *Fast Synchronization.* The synchronization procedure, synch, is used to guarantee that at least one of two events has occurred: (a) all processes have started executing increase-counter, or (b) some process has completed executing increase-counter. It uses a similar transitive information collection strategy to that used by fast-collect, but it is not wait-free. In case the processes run synchronously, it is guaranteed to terminate within time $O(\log n)$.

The code appears in Figure 8. In the code, each process p_j uses a flag T_j to indicate that it has completed executing increase-counter. If a process, while executing synch, ever finds any other process' flag equal to *true*, it terminates execution of synch.

In the absence of such early termination, a process executing synch attempts to determine that all processes have written their fields of the shared array R. It does so using the transitive collection strategy represented in Lines 5–6. The waiting loop in Line 4 ensures that (in the absence of early termination) the process does not terminate until all processes have written their fields of the array R. That is, when a process terminates, it must be that either all R_j are non-$\perp$ or some $T_j = true$. The fact that the information collection is done transitively implies a logarithmic upper bound in case all processes run synchronously.

For the rest of this subsection, fix some execution α of fast-n-approx.

The first lemma gives the correctness claim. Its proof is straightforward.

LEMMA 6.4. *Let α' be a finite prefix of α. Assume that in α'*, synch$_i$ *returns, for some p_i. Then, at the end of α' either all R entries are $\neq \perp$ or $T_j = true$ for some j.*

The next lemma gives a linear upper bound on the time required by synch.

LEMMA 6.5. *Let α' be a finite prefix of α and let $i \in \{0, \ldots, n-1\}$. Assume that in α' all R entries are set to values $\neq \perp$, and that* synch$_i$ *is invoked by p_i. Then*, synch$_i$ *returns within at most $6n$ steps by p_i after the end of α'.*

PROOF. Each iteration of the **while** loop in procedure synch takes at most six steps. (There are three operations, and because of alternation they might require six steps.) The claim follows, since the loop will be executed at most n times. □

The following lemma gives the $O(\log n)$ time bound.

LEMMA 6.6. *Let α' be a finite prefix of α. Assume that in α' all R entries are set to values $\neq \perp$, and* synch$_i$ *is invoked by p_i, for every $i \in \{0, \ldots, n-1\}$. Then, every process terminates* synch *within at most $O(\log n)$ time after the end of α'.*

PROOF. Let t be the time of the last event of α'. We prove that for every process p_i and for every integer r, $0 \leq r \leq \lceil \log n \rceil$, by time $t + 10r$, either p_i sets $T_i = true$ or $|R_i| \geq \min\{2^r, n\}$. The claim follows by taking $r = \lceil \log n \rceil$: by

```
shared var
        $R$ : array [1..n] of single writer register;

procedure synch($R$);
        begin
            repeat until $R_i \neq \bot$;                        /* $i$ has written. */
            $l := 1$;
            while $l < n$ and $T_{i+l \bmod n} = \bot$ do /* $p_{i+l \bmod n}$ has not yet terminated. */
                repeat until $R_{i+l \bmod n} \neq \bot$;         /* $p_{i+l \bmod n}$ has written. */
                $R_i$ := concatenate ($R_i$, $R_{(i+l) \bmod n}$);
                $l := |R_i|$;
            od;
        end;
```

FIG. 8. Fast non-wait-free synchronization—Code for process p_i.

time $t + 10\lceil \log n \rceil$, either p_i sets $T_i = true$ or $|R_i| \geq n$. If p_i sets $T_i = true$, then p_i has already terminated synch$_i$. On the other hand, if $|R_i| \geq n$, then p_i returns from synch$_i$ within $O(1)$ time.

The proof is by induction on r. The base case, $r = 0$, is trivial.

For the induction step, assume that $1 \leq r \leq \lceil \log n \rceil$. If p_i sets $T_i = true$ by time $t + 10r$, then the claim is immediate, so assume that T_i is not *true* by time $t + 10r$. In particular, T_i is not *true* by time $t + 10(r - 1)$. Hence, by the induction hypothesis, $|R_i| \geq \min\{2^{r-1}, n\} = 2^{r-1}$ by time $t + 10(r - 1)$.

By the code, there must be some time t', where $t + 10(r - 1) < t' \leq t + 10(r - 1) + 6$, at which p_i reads some T_j. (This bound takes into account the fact that the synch procedure is executed in strict alternation with another task.) Fix j to be the index of the first such read that occurs. If $T_j = true$ by time $t + 10(r - 1)$, then when p_i reads T_j the value is *true* and p_i sets $T_i = true$ by at most 2 time units later, that is, by time $t + 10(r - 1) + 8 \leq t + 10r$. This is a contradiction, so it must be that $T_j \neq true$ by time $t + 10(r - 1)$. By the induction hypothesis for $r - 1$, $|R_j| \geq 2^{r-1}$ by time $t + 10(r - 1)$. Since p_i reads T_j by time $t + 10(r - 1) + 6$, the code implies that p_i reads R_j and then writes R_i by time $t + 10r$. Then, the length of R_i at time $t + 10r$ is at least $2^{r-1} + 2^{r-1} = 2^r$, as needed. □

6.4. CORRECTNESS PROOF. We remind the reader that an execution of the algorithm is viewed as a sequence of primitive atomic operations that are reads and writes of atomic registers. We now fix some execution α of fast-n-approx.

As in the proof of the two-process algorithm (Section 5), the crucial point in the proof of the algorithm is showing that, in Lines 3–4 of fast-n-approx, processes use "close" values for c^a and c^b. We show that the value of an arbitrary counter when some process invokes fast-collect is at most 3 less than the maximum value that this counter ever attains. This is formalized and proved in the next lemma: (As before, we identify $\bot$ with -1 in arithmetic expressions.)

LEMMA 6.7. *Assume that p_i invokes* fast_collect$_i$ *in α. Fix some process p_j; let k be the value of C_j returned by* fast-collect$_i$. *Let k' be the maximum value attained for C_j in α. Then, $k' - 3 \leq k \leq k'$.*

PROOF. The inequality $k \leq k'$ follows immediately from the fact that reads and writes of the shared register are atomic. To prove the other inequality, let $p_{i'}$ be the first process to execute the write operation in Line 4 of increase-counter. Such a process exists because p_i performs this write operation before invoking fast-collect$_i$. Let α' be a shortest prefix of α that includes $p_{i'}$'s write to $T_{i'}$. Let k'' be the value of C_j at the end of α'. Since any invocation of fast-collect follows this last write operation in Line 4, Lemma 6.1 and the fact that reads and writes to C_j are atomic imply that $k'' \leq k$. Thus, it suffices to show that $k' - 3 \leq k''$. There are two cases according to the way $p_{i'}$ exits the **alternate** construct in Lines 2–3 of increase-counter:

Case 1. $p_{i'}$ exits the **while** loop. It must be that one of the halting conditions of the **while** loop is false for $p_{i'}$. If $p_{i'}$ and p_j are in the same group, that is, $g_{i'} = g_j$, then p_j will perform at most one iteration of the **while** loop after α' before p_j also sees the corresponding condition to be false. If $p_{i'}$ and p_j are not in the same group, that is, $g_{i'} \neq g_j$, then p_j will perform at most one iteration of the **while** loop after α' before p_j sees the first condition to be false (by observing $C_{i'} \neq \perp$). The claim follows.

Case 2. $p_{i'}$ returns from synch$_{i'}$. By definition, for all $l \in \{0, \ldots, n-1\}$, $T_l = \perp$ when $p_{i'}$ terminates synch$_{i'}$. It follows from Lemma 6.4 that, for all $l \in \{0, \ldots, n-1\}$, the value of C_l at the end of α' is $\neq \perp$. By Lemma 6.5, p_j will exit synch$_j(C)$ after performing at most $6n$ of its own steps after α'. It follows from the definition of **alternate** that p_j will perform at most $3n$ steps in the **while** loop in Line 2 of increase-counter, before synch$_j(C)$ terminates. However, each iteration of the **while** loop takes at least n steps (since n registers have to be read). Thus, p_j will perform at most three additional iterations of the **while** loop, before synch$_j(C)$ terminates. The claim follows. □

This implies that, for each local counter, the values read by two different processes differ at most by 3. Hence, the values used by different processes for the joint counters c^a and c^b differ at most by $3n$. Formally, we have:

LEMMA 6.8. *Suppose $i, j \in \{0, \ldots, n-1\}$ and $g \in \{a, b\}$. Assume the values returned by* fast-collect$_i$ *and* fast-collect$_j$ *are c_i^g and c_j^g, respectively. Then, $|c_i^g - c_j^g| \leq 3n$.*

We can now prove that the algorithm satisfies the agreement property:

LEMMA 6.9. *If* fast-approx$_i$ *returns y_i and* fast-approx$_j$ *returns y_j, then $|y_i - y_j| \leq \epsilon$.*

PROOF. The general outline of the proof parallels that of Lemma 5.2; however, some of the details are different. First, the discrepancy between processes' view of the joint counters might be $3n$; to compensate for that, we use bias with $\epsilon/6n$. In addition, we must allow for the possibility of using different values from the same group (by applying Lemma 4.4). The details follow.

We start with the proof for the case where p_i and p_j are not in the same group; without loss of generality, assume $g_i = a$ and $g_j = b$.

Assume that the values computed by p_i based on fast-collect$_i$ to be used in Lines 3–4 of fast-n-approx are $\langle v_i^a, v_i^b, c_i^a, c_i^b \rangle$; similarly, assume that the values computed by p_j based on fast-collect$_j$ to be used in Lines 3–4 of fast-n-approx are $\langle v_j^a, v_j^b, c_j^a, c_j^b \rangle$. Note that since p_i is in group a, $c_i^a \geq 0$ and $v_i^a \neq \perp$; similarly, since p_j is in group b, $c_j^b \geq 0$ and $v_j^b \neq \perp$.

For any process p_k, denote by π_k the write by process p_k in Line 1 of increase-counter (if it appears in α). Since p_i and p_j decide, π_i and π_j must appear in α. Let $p_{i'}$ be such that $\pi_{i'}$ is the first write of Line 1 of increase-counter in α. Assume, without loss of generality, that $p_{i'}$ is in group a. Intuitively, we assume that the first process to start the second phase of the algorithm belongs to p_i's group, a.

The code of the algorithm implies that, for any $p_{j'}$ in group b, $\pi_{j'}$ precedes any calculation of C^a by $p_{j'}$. Since $\pi_{i'}$ precedes $\pi_{j'}$ it follows that $p_{j'}$ will always calculate $C^a \neq \perp$. Thus, $c_j^a \geq 0$ and hence fast-n-approx$_j$ returns in Line 4 and $v_j^a \neq \perp$. Also, the above implies that C^b never increases beyond 0. Thus, $c_j^b = 0$ and $c_i^b \in \{\perp, 0\}$. We separate the rest of the proof into two cases:

Case 1. $c_i^b = \perp$. Then, fast-n-approx$_i$ returns v_i^a in Line 3. From the code, it follows that $c_i^a \geq |v_i^a| 6n/\epsilon$. By Lemma 6.8, $c_j^a \geq |v_i^a| 6n/\epsilon - 3n$. Since $c_j^a \geq 0 = c_j^b$, applying Lemma 4.2 (2) with $m = 3n$ we get that

$$\left| \mathsf{bias}\left(v_i^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) - v_i^a \right| \leq \frac{\epsilon}{2}. \tag{1}$$

Also, Theorem 3.1 implies that $|v_i^a - v_j^a| \leq \epsilon/12$. Applying Lemma 4.4 with $\delta = \epsilon/12$, $c^0 = c_j^a$, $c^1 = c_j^b$, $v_0^0 = v_j^a$, $v_0^1 = v_j^b$, $v_1^0 = v_i^a$, $v_1^1 = v_j^b$, we get that

$$\left| \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_i^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) \right| \leq \frac{6\epsilon}{12} = \frac{\epsilon}{2}. \tag{2}$$

From (1) and (2), it follows that

$$\left| \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) - v_i^a \right| \leq \epsilon,$$

as needed.

Case 2. $c_i^b = 0$. Thus, fast-n-approx$_i$ returns in Line 4 and $v_i^b \neq \perp$. We have that $\min\{c_i^a, c_i^b\} = c_i^b = 0$ and $\min\{c_j^a, c_j^b\} = c_j^b = 0$. Also, $|c_i^a - c_j^a| + |c_i^b - c_j^b| = |c_i^a - c_j^a| \leq 3n$ by Lemma 6.8. Applying Lemma 4.3 with $m = 3n$ we get

$$\left| \mathsf{bias}\left(v_j^a, v_j^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) \right| \leq 3n \cdot \frac{\epsilon}{6n} = \frac{\epsilon}{2}. \tag{3}$$

Also, Theorems 3.1 and 3.10 imply that $|v_i^a - v_j^a| \leq \epsilon/12$ and $|v_i^b - v_j^b| \leq \epsilon/12$. By applying Lemma 4.4 with $\delta = \epsilon/12$, we get

$$\left| \mathsf{bias}\left(v_i^a, v_i^b,' c_i^a, c_i^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_j^a, v_j^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) \right| \leq \frac{6\epsilon}{12} = \frac{\epsilon}{2}. \tag{4}$$

From (3) and (4), it follows that

$$\left| \mathsf{bias}\left(v_i^a, v_i^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) \right| \leq \epsilon,$$

as needed.

We now consider the case where p_i and p_j are in the same group; without loss of generality, assume $g_i = g_j = a$. Let $p_{i'}$ be such that $\pi_{i'}$ is the first write of Line 1 of increase-counter in α. (As before, π_k is the write by process p_k in Line 1 of increase-counter.) Assume first that p_i, p_j belong to the group that wrote first, that is, $g_{i'} = g_i = g_j$. In this case, $c_i^b, c_j^b \in \{\perp, 0\}$ (by arguments similar to those above). We separate the rest of the proof into three cases:

Case 1. $c_i^b = \perp$. Then fast-n-approx$_i$ returns v_i^a in Line 3. If $c_j^b = \perp$, then fast-n-approx$_j$ returns v_j^a in Line 3, and the claim follows, since Theorem 3.1 implies that $|v_i^a - v_j^a| \leq \epsilon/12$. Otherwise, $c_j^b = 0$. From the code, it follows that $c_i^a \geq |v_i^a| 6n/\epsilon$. By Lemma 6.8, $c_j^a \geq |v_i^a| 6n/\epsilon - 3n$. Since $c_j^a \geq 0 = c_j^b$, applying Lemma 4.2(2) with $m = 3n$, we get that

$$\left| \mathsf{bias}\left(v_i^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) - v_i^a \right| \leq \frac{\epsilon}{2}. \tag{5}$$

Also, Theorem 3.1 implies that $|v_i^a - v_j^a| \leq \epsilon/12$. Applying Lemma 4.4 with $\delta = \epsilon/12$, $c^0 = c_j^a$, $c^1 = c_j^b$, $v_0^0 = v_j^a$, $v_0^1 = v_j^b$, $v_1^0 = v_i^a$, $v_1^1 = v_j^b$, we get that

$$\left| \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_i^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) \right| \leq \frac{6\epsilon}{12} = \frac{\epsilon}{2}. \tag{6}$$

From (5) and (6), it follows that

$$\left| \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) - v_i^a \right| \leq \epsilon,$$

as needed.

Case 2. $c_j^b = \perp$ is symmetric to Case 1.

Case 3. $c_i^b = c_j^b = 0$. Thus, fast-n-approx$_i$ and fast-n-approx$_j$ return in Line 4 and $v_i^b, v_j^b \neq \perp$. We have that $\min\{c_i^a, c_i^b\} = c_i^b = 0$ and $\min\{c_j^a, c_j^b\} = c_j^b = 0$. Also, $|c_i^a - c_j^a| + |c_i^b - c_j^b| = |c_i^a - c_j^a| \leq 3n$ by Lemma 6.8. Applying Lemma 4.3 with $m = 3n$, we get

$$\left| \mathsf{bias}\left(v_j^a, v_j^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) \right| \leq 3n \cdot \frac{\epsilon}{6n} = \frac{\epsilon}{2}. \tag{7}$$

Also, Theorems 3.1 and 3.10 imply that $|v_i^a - v_j^a| \leq \epsilon/12$ and $|v_i^b - v_j^b| \leq \epsilon/12$. By applying Lemma 4.4 with $\delta = \epsilon/12$, we get

$$\left| \mathsf{bias}\left(v_i^a, v_i^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_j^a, v_j^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) \right| \leq \frac{6\epsilon}{12} = \frac{\epsilon}{2}. \tag{8}$$

From (7) and (8), it follows that

$$\left| \mathsf{bias}\left(v_i^a, v_i^b, c_i^a, c_i^b, \frac{\epsilon}{6n} \right) - \mathsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n} \right) \right| \leq \epsilon,$$

as needed.

Assume now that p_i and p_j are not in the group that wrote first, that is, $g_{i'} \neq g_i$. By arguments similar to those above, $c_i^a = c_j^a = 0$, $c_i^b \geq 0$ and $c_j^b \geq 0$. Thus, fast-n-approx$_i$ returns in Line 4 and $v_i^b \neq \perp$. We have that $\min\{c_i^a, c_i^b\}$

$= c_i^a = 0$ and $\min\{c_j^a, c_j^b\} = c_j^a = 0$. Also, $|c_i^a - c_j^a| + |c_i^b - c_j^b| = |c_i^b - c_j^b| \leq 3n$ by Lemma 6.8. Applying 4.3 with $m = 3n$, we get

$$\left|\textsf{bias}\left(v_j^a, v_j^b, c_i^a, c_i^b, \frac{\epsilon}{6n}\right) - \textsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n}\right)\right| \leq 3n \cdot \frac{\epsilon}{6n} = \frac{\epsilon}{2}. \quad (9)$$

Also, Theorems 3.1 and 3.10 imply that $|v_i^a - v_j^a| \leq \epsilon/12$ and $|v_i^b - v_j^b| \leq \epsilon/12$. By applying Lemma 4.4 with $\delta = \epsilon/12$, we get

$$\left|\textsf{bias}\left(v_i^a, v_i^b, c_i^a, c_i^b, \frac{\epsilon}{6n}\right) - \textsf{bias}\left(v_j^a, v_j^b, c_i^a, c_i^b, \frac{\epsilon}{6n}\right)\right| \leq \frac{6\epsilon}{12} = \frac{\epsilon}{2}. \quad (10)$$

From (9) and (10), it follows that

$$\left|\textsf{bias}\left(v_i^a, v_i^b, c_i^a, c_i^b, \frac{\epsilon}{6n}\right) - \textsf{bias}\left(v_j^a, v_j^b, c_j^a, c_j^b, \frac{\epsilon}{6n}\right)\right| \leq \epsilon,$$

as needed. □

We have:

THEOREM 6.10. *Procedure* fast-n-approx *is a wait-free algorithm for the n-process approximate agreement problem whose time complexity is $O(\log n)$.*

PROOF. Agreement follows from Lemma 6.9. Validity follows immediately since the values returned by wait-free-approx and wait-approx are in the range of the original inputs, and the bias function preserves this property (by Lemma 4.1).

The algorithm is wait-free because the first alternative of each alternation construct and fast-collect are wait-free.

Within $O(1)$ time all processes finish n-to-2. Thus, within $O(1)$ time all processes start procedure increase-counter, write to C_i and invoke synch. By Lemma 6.6, within $O(\log n)$ time, each process terminates synch. Thus, within $O(\log n)$ time, all processes exit increase-counter and invoke fast-collect. By Lemma 6.3, all processes return from fast-collect within $O(\log n)$ time. Hence, the total time complexity is $O(\log n)$. □

7. *A log n Time Lower Bound*

In this section, we show that the $\log n$ dependency exhibited by the algorithm of Theorem 6.10 is inherent: The time complexity of any wait-free algorithm for n-process approximate agreement is at least $\log n$. Together with Theorem 3.1, this result shows that there are problems for which wait-free algorithms take more time (by an $\Omega(\log n)$ factor) than non-wait-free algorithms.

In the rest of this section, we assume that each process has only one register to which it can write. Since the size of registers is not restricted and since only one process may write to each register, there is no loss of generality in this assumption. Let R_i be the register to which p_i writes. For a configuration C and a process p_i, let $st(p_i, C)$ be the pair consisting of the local state of p_i and the value of R_i in C, (that is, $st(p_i, C) = \langle state(p_i, C), val(R_i, C)\rangle$.

The *synchronized schedule* is the schedule in which processes take steps in round-robin order starting with p_0, essentially operating synchronously. The sequence of r rounds in the round-robin order is denoted σ_r. For any configuration C, the corresponding *synchronized execution* from C is uniquely determined by the algorithm. Note that this is a failure-free execution.

We now define the set of processes that could have influenced p_i's state at time r in the synchronized execution from a configuration C. Let C be a configuration; by induction on $r \geq 0$, define the set $INF(p_i, r, C)$, for every $i \in \{0, \ldots, n-1\}$, using the following rules:

(1) $r = 0$: $INF(p_i, r, C) = \{p_i\}$, for every $i \in \{0, \ldots, n-1\}$.
(2) $r \geq 1$: If p_i's rth step in (C, σ_r) is a read of R_j, then $INF(p_i, r, C) = INF(p_i, r-1, C) \cup INF(p_j, r-1, C)$. If p_i's rth step is a write (to R_i), then $INF(p_i, r, C) = INF(p_i, r-1, C)$.

LEMMA 7.1. *$|INF(p_i, r, C)| \leq 2^r$ for every configuration C, $r \geq 0$ and $i \in \{0, \ldots, n-1\}$.*

PROOF. By induction on r. □

The next lemma formalizes the intuition that INF includes all the processes that can influence p's state up to time r.

LEMMA 7.2. *Let C_1 and C_2 be two configurations, let p_i be any process and let $r \geq 0$. If $st(p_k, C_1) = st(p_k, C_2)$ for all $p_k \in INF(p_i, r, C_1)$, then $st(p_i, C_1\sigma_r) = st(p_i, C_2\sigma_r)$.*

PROOF. The proof is by induction on r. For the base case, $r = 0$, we have $INF(p_i, 0, C_1) = \{p_i\}$ and $\sigma_0 = \lambda$. Then, the claim follows immediately from the assumption.

To prove the induction step, assume $r \geq 1$ and the claim holds for $r-1$, and suppose that $st(p_k, C_1) = st(p_k, C_2)$ for all $p_k \in INF(p_i, r, C_1)$. Since, by definition, $INF(p_i, r-1, C_1) \subseteq INF(p_i, r, C_1)$, it follows that $st(p_k, C_1) = st(p_k, C_2)$ for all $p_k \in INF(p_i, r-1, C_1)$. Then, by the induction hypothesis, $st(p_i, C_1\sigma_{r-1}) = st(p_i, C_2\sigma_{r-1})$. We consider two cases:

If p_i's rth step in (C_1, σ_r) is a write, then the fact that $st(p_i, C_1\sigma_{r-1}) = st(p_i, C_2\sigma_{r-1})$ implies that $st(p_i, C_1\sigma_r) = st(p_i, C_2\sigma_r)$, as needed.

On the other hand, suppose that p_i's rth step in (C_1, σ_r) is a read, say from R_j. By definition, $INF(p_j, r-1, C_1) \subseteq INF(p_i, r, C_1)$, and hence, $st(p_k, C_1) = st(p_k, C_2)$ for all $p_k \in INF(p_j, r-1, C_1)$. Then, by the induction hypothesis, $st(p_j, C_1\sigma_{r-1}) = st(p_j, C_2\sigma_{r-1})$. Since also $st(p_i, C_1\sigma_{r-1}) = st(p_i, C_2\sigma_{r-1})$, it follows that $st(p_i, C_1\sigma_r) = st(p_i, C_2\sigma_r)$, as needed. □

We can now prove:

THEOREM 7.3. *Any wait-free algorithm for the n-process approximate agreement problem has time complexity at least $\log n$.*

PROOF. Assume that A is a wait-free approximate agreement algorithm. We prove a slightly stronger claim: There exists a failure-free execution α in which no process decides before time $\log n$. Suppose, by way of contradiction, that in all failure-free executions some process decides before time $\log n$.

Fix some $\epsilon < 1$. Let σ be the infinite synchronized schedule. Consider the execution (C_0, σ) of A from the initial configuration C_0 where processes start with inputs $\langle 0, \ldots, 0 \rangle$. Let t be the time associated with the first decision event in (C_0, σ), and let p_i be the process associated with this event; by assumption, $t < \log n$. By the validity property, p_i must decide on 0 since all processes start with 0.

By Lemma 7.1, we have that $|INF(p_i, t, C_0)| \leq 2^t < n$. Thus, there exists some process, say p_j, that is not in $INF(p_i, t, C_0)$.

Intuitively, to complete the proof, we create an alternative execution in which p_j "starts early" with input 1, runs on its own and thus must eventually decide 1. We then let the rest of the processes execute as if they are in the synchronized execution from C_0 and use Lemma 7.2 to show that process p_i still decides on 0, which is a contradiction to the agreement property, since $\epsilon < 1$.

More precisely, apply τ, an infinite schedule consisting of steps of p_j only, to the initial configuration C_2, where processes start with inputs $\langle 1, \ldots, 1 \rangle$. The resulting execution (C_2, τ) is $(n-1)$-admissible, and thus, since A $(n-1)$-solves the approximate agreement problem, and since p_j is nonfaulty in τ, there exists a finite prefix τ' of τ in which p_j decides. By validity, p_j decides on 1. Now apply τ' to the initial configuration C_1 where all processes but p_j start with input 0, and p_j starts with input 1. By induction on the prefixes of τ', it follows that $st(p_j, C_1\tau') = st(p_j, C_2\tau')$. Thus, p_j decides on 1 in $C_1\tau'$. Since p_j can write only to R_j, it follows that for all processes $p_k \neq p_j$, $st(p_k, C_1\tau') = st(p_k, C_0)$. By Lemma 7.2, $state(p_i, C_1\tau'\sigma_t) = state(p_i, C_0\sigma_t)$. Thus, p_i decides on 0 in $C_1\tau'\sigma_t$, and p_j decides 1, which is a contradiction to agreement, since $\epsilon < 1$. □

8. *A Trade-Off between Work and Time*

We now consider the performance of wait-free algorithms when failures occur. A drawback of the fast algorithms we have presented in this paper is that, if a failure *does* occur, then the remaining processes will have to take many steps before halting. We show that this phenomenon is unavoidable. Roughly speaking, we prove that if an algorithm terminates in a small number of steps in executions where failures do occur, then it is slow in normal executions. In the rest of this section, we restrict our attention to the two-process case.

Let the work performed by an algorithm be defined as the maximum, over all executions, of the total number of operations performed by all processes before deciding. To bound the work from below, we show a stronger bound: We prove a lower bound on the number of operations a *single* process performs before deciding when running on its own. Clearly, this also gives a lower bound on the work.

Let $k \geq 1$ be an integer. An algorithm is *k-bounded* if from any reachable configuration, a process that executes k consecutive steps on its own must decide. Fix a k-bounded wait-free algorithm A for approximate agreement; all definitions and lemmas in the rest of this section are with respect to A. For each process p_i and each configuration C reachable in an execution of A, define $pref_i(C)$, the *preference* of p_i in C, to be the value on which p_i decides in the execution fragment starting from C in which it runs alone until it decides.

A finite schedule is a *block* if it consists of a positive number of events by p_0 followed by one event by p_1, or vice versa.

LEMMA 8.1. *Let σ be a finite schedule, and let C_0 be an initial configuration. Let $C = C_0\sigma$. Then, there exists a finite block schedule σ' such that*

$$|pref_0(C\sigma') - pref_1(C\sigma')| \geq \frac{1}{2k} |pref_0(C) - pref_1(C)|.$$

PROOF. The proof considers the tree of all block schedules applied to C. A case analysis, according to the types of steps taken, similar to the one in [Loui and Abu-Amara, 1987], is used to show that it cannot be that all the pairs of preferences associated with leaves of this tree are close together. The details follow:

Let $\tau_0 = 0^k$, that is, the schedule consisting of k events of p_0. Similarly, let $\tau_1 = 1^k$. Let $(C, \tau_0) = C, C_1, \ldots, C_k$, and $(C, \tau_1) = C, C'_1, \ldots, C'_k$. For any l, $1 \leq l \leq k$, define $D_l = C_l 1$, that is, the configuration that results from applying an event of p_1 to C_l. Similarly, for any l, $1 \leq l \leq k$, define $D'_l = C'_l 0$. Define $v_0^l = pref_0(D_l)$, $v_1^l = pref_1(D_l)$, $u_0^l = pref_0(D'_l)$ and $u_1^l = pref_1(D'_l)$.

Since A is k-bounded, it must be that p_0 decides in $C\tau_0$; by definition, it must decide on $pref_0(C)$. Similarly, p_1 decides on $pref_1(C)$ in $C\tau_1$. Note that $pref_0(C) = pref_0(C_k) = pref_0(C_k 1) = v_0^k$, and $pref_1(C) = pref_1(C'_k) = pref_1(C'_k 0) = u_1^k$.

We show that for all l, $1 \leq l < k$, either $v_0^l = v_0^{l+1}$ or $v_1^l = v_1^{l+1}$. There are four cases, depending on the type of operation taken in p_0's step from C_l to C_{l+1} and in p_1's step from C_l to D_l:

(1) p_0 writes and p_1 writes: commutativity implies that $v_0^l = v_0^{l+1}$.
(2) p_0 reads and p_1 reads: commutativity implies that $v_0^l = v_0^{l+1}$.
(3) p_0 writes and p_1 reads: $v_0^l = v_0^{l+1}$, since the state of p_0 is the same in $D_l 0$ and D_{l+1}.
(4) p_0 reads and p_1 writes: $v_1^l = v_1^{l+1}$, since the state of p_1 is the same in D_l and D_{l+1}.

By symmetric arguments we can show that for all l, $1 \leq l < k$, either $u_0^l = u_0^{l+1}$ or $u_1^l = u_1^{l+1}$. In a similar manner, we show that either $v_1^1 = u_1^1$ or $v_0^1 = u_0^1$, by case analysis, depending on the type of operation taken in p_0's step from C to C_1 and in p_1's step from C to C'_1:

(1) p_0 writes and p_1 writes: commutativity implies that $v_0^1 = u_0^1$ and $v_1^1 = u_1^1$.
(2) p_0 reads and p_1 reads: commutativity implies that $v_0^1 = u_0^1$ and $v_1^1 = u_1^1$.
(3) p_0 writes and p_1 reads: $v_0^1 = u_0^1$, since the state of p_0 is the same in D_1 and D'_1.
(4) p_0 reads and p_1 writes: $v_1^1 = u_1^1$, since the state of p_1 is the same in D_1 and D'_1.

Suppose, for instance, that $v_1^1 = u_1^1$. (The argument is analogous if $v_0^1 = u_0^1$.) It is possible to show (e.g., by induction) that $|v_0^k - v_1^1| \leq \Sigma_{l=1}^k |v_0^l - v_1^l|$, and that $|u_1^k - u_1^1| \leq \Sigma_{l=1}^k |u_1^l - u_0^l|$. Therefore, $|v_0^k - u_1^k| \leq \Sigma_{l=1}^k |v_0^l - v_1^l| + \Sigma_{l=1}^k |u_1^l - u_0^l|$. By simple calculations, this implies that either there exists some l such that $|v_0^l - v_1^l| \geq 1/2k|v_0^k - u_1^k|$, or there exists some l such that, $|u_0^l - u_1^l| \geq 1/2k|v_0^k - u_1^k|$. Recall that $pref_0(C) = v_0^k$, and $pref_1(C) = u_1^k$. Therefore, either there exists some l such that $|v_0^l - v_1^l| \geq 1/2k|pref_0(C) - pref_1(C)|$, or there exists some l such that, $|u_0^l - u_1^l| \geq 1/2k|pref_0(C) - pref_1(C)|$. In the first case, the claim follows by taking $\sigma' = 0^l 1$; in the second case, the claim follows by taking $\sigma' = 1^l 0$.

These facts can be used to show (e.g., by induction) that $|v_0^k - v_1^0| \leq \Sigma_{l=1}^k |v_0^l - v_1^l|$, and that $|u_1^k - u_0^0| \leq \Sigma_{l=1}^k |u_1^l - u_0^l|$. By simple calculations, this implies that either there exists some l such that $|v_0^l - v_1^l| \geq 1/2k|v_0^k - u_1^k|$, or there exists some l such that, $|u_0^l - u_1^l| \geq 1/2k|v_0^k - u_1^k|$. Recall that $pref_0(C) = v_0^k$, and $pref_1(C) = u_1^k$. Therefore, either there exists some l such that $|v_0^l - v_1^l| \geq 1/2k|pref_0(C) - pref_1(C)|$, or there exists some l such that, $|u_0^l - u_1^l| \geq 1/2k|pref_0(C) - pref_1(C)|$. In the first case, the claim follows by taking $\sigma' = 0^l 1$; in the second case, the claim follows by taking $\sigma' = 1^l 0$. □

Note that the validity condition implies that if p_i's input in an initial configuration C is v_i then $pref_i(C) = v_i$. Starting with this fact and applying Lemma 8.1 iteratively, we can bound the rate at which a k-bounded algorithm converges. We get:

THEOREM 8.2. *Let A be a k-bounded wait-free algorithm for approximate agreement between two processes, and let x_0 and x_1 be arbitrary real numbers, $x_0 \neq x_1$. Then there exists an execution of A where processes start with inputs $\langle x_0, x_1 \rangle$, in which the time complexity is $\Omega(\log_{2k}(|x_0 - x_1|/\epsilon))$.*

PROOF. Let C be an initial configuration in which the two processes have inputs x_0 and x_1, respectively. We construct, inductively, a schedule σ_l such that σ_l is a sequence of l blocks and for $C_l = C\sigma_l$,

$$|pref_0(C_l) - pref_1(C_l)| \geq \left(\frac{1}{2k}\right)^l |pref_0(C) - pref_1(C)|.$$

This is done by repeatedly applying Lemma 8.1. We have that $time(\sigma_l) = l$, since σ_l consists of l blocks. The validity condition implies that $pref_i(C) = x_i$. Thus, $|pref_0(C) - pref_1(C)| = |x_0 - x_1|$. The claim follows by noticing that it cannot be that both p_0 and p_1 have decided in a configuration D if $|pref_0(D) - pref_1(D)| > \epsilon$. □

REMARK 8.3. The case analysis in the proof of Lemma 8.1 can be extended to handle multi-writer multi-reader registers; thus, the above trade-off applies also to algorithms that use *multi-writer* multi-reader atomic registers.

9. *Properties of the Bias Function*

In this section, the interested reader may find the long postponed proofs of Lemma 4.1 through 4.4. We begin with the rather straightforward proof of Lemma 4.1.

LEMMA 4.1. *Let c^0, c^1 be nonnegative integers, and v^0, v^1, ϵ be real numbers, with $\epsilon > 0$. Then*

$$\mathsf{bias}(v^0, v^1, c^0, c^1, \epsilon) \in range(\{v^0, v^1\}).$$

PROOF. Let $y = \mathsf{bias}(v^0, v^1, c^0, c^1, \epsilon)$. The claim is trivial if y is calculated in Line 1. If y is calculated in Line 2, then

$$y = v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}(|v^1| - \min\{c^1\epsilon, |v^1|\}).$$

If the min is attained in the second term, then $y = v^1$ and the claim follows. So assume $c^1\epsilon \leq |v^1|$, so

$$y = v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}(|v^1| - c^1\epsilon).$$

Assume $v^1 \geq v^0$. (A symmetric argument applies when $v^1 < v^0$.) Then, $v^0 - v^1 \leq 0$, so $y \leq v^1$. Since

$$\left|\frac{v^0 - v^1}{|v^0| + |v^1|}(|v^1| - c^1\epsilon)\right| \leq v^1 - v^0,$$

it follows that $y \geq v^0$.

The case where y is calculated in Line 3 is symmetric. □

The following is the proof of Lemma 4.2:

LEMMA 4.2. *Let c^0, c^1 be nonnegative integers, and v^0, v^1, ϵ, m be real numbers, $\epsilon > 0$, $m \geq 0$.*

(1) *Suppose $c^1 > c^0$ and $|v^1|/\epsilon - m \leq c^1$. Then $|\mathsf{bias}(v^0, v^1, c^0, c^1, \epsilon) - v^1| \leq m\epsilon$.*
(2) *Suppose $c^0 \geq c^1$ and $|v^0|/\epsilon - m \leq c^0$. Then $|\mathsf{bias}(v^0, v^1, c^0, c^1, \epsilon) - v^0| \leq m\epsilon$.*

PROOF. We present the proof only for (2); the proof for (1) follows from symmetric arguments. Let $y = \mathsf{bias}(v^0, v^1, c^0, c^1, \epsilon)$. If y is calculated in Line 1 of the bias code, then $y = 0$ and $v^0 = 0$ and the claim follows. hence, since $c^0 \geq c^1$, it follows that y is calculated in Line 3 of bias, that is,

$$y = v^0 + \frac{v^1 - v^0}{|v^0| + |v^1|}(|v^0| - \min\{c^0\epsilon, |v^0|\}).$$

If the min attains its value in the second term, then $y = v^0$, and the claim follows. Otherwise, $c^0\epsilon \leq |v^0|$; thus,

$$\begin{aligned}|y - v^0| &= \left|\frac{v^1 - v^0}{|v^0| + |v^1|}(|v^0| - c^0\epsilon)\right| \\ &= \frac{|v^1 - v^0|}{|v^0| + |v^1|}||v^0| - c^0\epsilon| \\ &\leq ||v^0| - c^0\epsilon| = |v^0| - c^0\epsilon \leq m\epsilon,\end{aligned}$$

by the hypothesis of the lemma. □

Next is the proof of Lemma 4.3.

LEMMA 4.3. *Let $c_0^0, c_0^1, c_1^0, c_1^1$ be nonnegative integers, and v^0, v^1, ϵ, m be real numbers, $\epsilon > 0$ and $m \geq 0$. Suppose $min\{c_0^0, c_0^1\} = min\{c_1^0, c_1^1\} = 0$ and $|c_0^0 - c_1^0| + |c_0^1 - c_1^1| \leq m$. Then*

$$|\mathsf{bias}(v^0, v^1, c_0^0, c_0^1, \epsilon) - \mathsf{bias}(v^0, v^1, c_1^0, c_1^1, \epsilon)| \leq m\epsilon.$$

PROOF. Let $y_0 = \mathsf{bias}(v^0, v^1, c_0^0, c_0^1, \epsilon)$, and $y_1 = \mathsf{bias}(v^0, v^1, c_1^0, c_1^1, \epsilon)$.

If $v^0 = v^1 = 0$, then both y_0 and y_1 are calculated in Line 1 of bias, that is, $y_0 = y_1 = 0$ and the claim follows.

Now assume y_0 is calculated in Line 2 of bias, while y_1 is calculated in Line 3 of bias (the reverse case is symmetric). Thus, $c_0^0 < c_0^1$, while $c_1^1 \leq c_1^0$. Thus, by assumption, $c_0^0 = c_1^1 = 0$. Since $|c_0^0 - c_1^0| + |c_0^1 - c_1^1| \leq m$, it follows that $|c_1^0| + |c_0^1| = c_1^0 + c_0^1 \leq m$. Thus, $\min\{c_1^0, |v^0|/\epsilon\} + \min\{c_0^1, |v^1|/\epsilon\} \leq m$. So, $\min\{c_1^0\epsilon, |v^0|\} + \min\{c_0^1\epsilon, |v^1|\} \leq m\epsilon$. We have

$$y_0 = v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}\left(|v^1| - \min\{c_0^1\epsilon, |v^1|\}\right)$$

and

$$y_1 = v^0 + \frac{v^1 - v^0}{|v^0| + |v^1|}\left(|v^0| - \min\{c_1^0\epsilon, |v^0|\}\right).$$

Thus,

$$\begin{aligned}
|y_0 - y_1| &= \left| v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}\left(|v^1| - \min\{c_0^1\epsilon, |v^1|\}\right)\right. \\
&\qquad \left. - v^0 - \frac{v^1 - v^0}{|v^0| + |v^1|}\left(|v^0| - \min\{c_1^0\epsilon, |v^0|\}\right)\right| \\
&= \left| v^1 - v^0 + \frac{v^0 - v^1}{|v^0| + |v^1|}(|v^0| + |v^1|)\right. \\
&\qquad \left. - \frac{v^0 - v^1}{|v^0| + |v^1|}\left(\min\{c_0^1\epsilon, |v^1|\} + \min\{c_1^0\epsilon, |v^0|\}\right)\right| \\
&= \frac{|v^0 - v^1|}{|v^0| + |v^1|}\left|\min\{c_0^1\epsilon, |v^1|\} + \min\{c_1^0\epsilon, |v^0|\}\right| \\
&\leq \left|\min\{c_0^1\epsilon, |v^1|\} + \min\{c_1^0\epsilon, |v^0|\}\right| \\
&= \min\{c_0^1\epsilon, |v^1|\} + \min\{c_1^0\epsilon, |v^0|\} \leq m\epsilon,
\end{aligned}$$

as needed.

Now assume that both y_0 and y_1 are calculated in Line 2 of bias (the case where both are calculated in Line 3 of bias is symmetric), that is,

$$y_0 = v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}\left(|v^1| - \min\{c_0^1\epsilon, |v^1|\}\right)$$

and

$$y_1 = v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}\left(|v^1| - \min\{c_1^1\epsilon, |v^1|\}\right).$$

If, for y_0, the min is attained in the second term, then $c_0^1\epsilon \ge |v^1|$, and $y_0 = v^1$; since $|c_0^1 - c_1^1| \le m$, it follows that $c_1^1 \ge |v^1|/\epsilon - m$. Because y_1 is calculated in Line 2, $c_1^0 < c_1^1$ and the claim follows from Lemma 4.2(1). A similar argument applies if for y_1 the min is attained in the second term. So assume that for both y_0 and y_1 the min is attained in the first term. Thus,

$$\begin{aligned}
|y_0 - y_1| &= \left| v^1 + \frac{v^0 - v^1}{|v^0| + |v^1|}\left(|v^1| - c_0^1\epsilon\right) - v^1 - \frac{v^0 - v^1}{|v^0| + |v^1|}\left(|v^1| - c_1^1\epsilon\right)\right| \\
&= \left| \frac{v^0 - v^1}{|v^0| + |v^1|}\left(c_1^1\epsilon - c_0^1\epsilon\right)\right| \\
&= \frac{|v^0 - v^1|}{|v^0| + |v^1|}\left|\left(c_1^1\epsilon - c_0^1\epsilon\right)\right| \\
&\le \left|\left(c_1^1\epsilon - c_0^1\epsilon\right)\right| = \epsilon|c_1^1 - c_0^1| \le m\epsilon,
\end{aligned}$$

as needed. □

In the proof of the next lemma, we use the following two facts:

CLAIM 9.1. *If x, y, x', y' are real numbers, such that $|x| + |y| \ne 0$ and $|x'| + |y'| \ne 0$, and for some δ, $|x - x'| \le \delta$ and $|y - y'| \le \delta$, then*

$$\left| \frac{|x|(y - x)}{|x| + |y|} - \frac{|x'|(y' - x')}{|x'| + |y'|} \right| \le 3\delta.$$

We prove this claim by first showing that

$$\left| \frac{x(y - x)}{x + y} - \frac{x'(y' - x')}{x' + y'} \right| \le 3\delta,$$

using calculus, then handling the absolute values by case analysis.

CLAIM 9.2. *If x, y, x', y' are real numbers, such that $|x| + |y| \ne 0$ and $|x'| + |y'| \ne 0$, and for some δ, $|x - x'| \le \delta$ and $|y - y'| \le \delta$, then*

$$\left| \frac{(y - x)}{|x| + |y|} - \frac{(y' - x')}{|x'| + |y'|} \right| \le \frac{2\delta}{\min(|x| + |y|, |x'| + |y'|)}.$$

We prove this claim by straightforward calculations and a case analysis. Finally, we can prove Lemma 4.4.

LEMMA 4.4. *Let c^0, c^1 be nonnegative integers, and $v_0^0, v_0^1, v_1^0, v_1^1, \epsilon, \delta$ be real numbers, with $\epsilon > 0$, $\delta \ge 0$. Suppose $|v_0^0 - v_1^0| \le \delta$ and $|v_0^1 - v_1^1| \le \delta$. Then*

$$\left|\mathsf{bias}(v_0^0, v_0^1, c^0, c^1, \epsilon) - \mathsf{bias}(v_1^0, v_1^1, c^0, c^1, \epsilon)\right| \le 6\delta.$$

PROOF. Let $y_0 = \mathsf{bias}(v_0^0, v_0^1, c^0, c^1, \epsilon)$, and $y_1 = \mathsf{bias}(v_1^0, v_1^1, c^0, c^1, \epsilon)$. If $v_0^0 = v_0^1 = 0$, then $y_0 = 0$. Thus, $|v_1^0| \le \delta$ and $|v_1^1| \le \delta$. So, from Lemma 4.1, it follows that $|y_1| \le \delta$ and the claim follows. The case $v_1^0 = v_1^1 = 0$ follows from symmetric arguments. So assume at least one of v_0^0, v_0^1 is nonzero and similarly for at least one of v_1^0, v_1^1.

Assume that $c^0 < c^1$, that is, y_0 and y_1 are calculated in Line 2. (The other case, where $c^1 \leq c^0$ and y_0 and y_1 are calculated in Line 3, is symmetric.) Then

$$y_0 = v_0^1 + \frac{v_0^0 - v_0^1}{|v_0^0| + |v_0^1|}\left(|v_0^1| - \min\{c^1\epsilon, |v_0^1|\}\right)$$

and

$$y_1 = v_1^1 + \frac{v_1^0 - v_1^1}{|v_1^0| + |v_1^1|}\left(|v_1^1| - \min\{c^1\epsilon, |v_1^1|\}\right).$$

First, assume the min for y_0 is attained in the second term; then $y_0 = v_0^1$. In this case, if the min for y_1 is also attained in the second term, then $y_1 = v_1^1$, and the claim follows. On the other hand, suppose the min for y_1 is attained in the first term. Since the min for y_0 is attained in the second term, $c^1\epsilon \geq |v_0^1| \geq |v_1^1| - \delta$. Applying Lemma 4.2(1) with $m = \delta/\epsilon$, we get that $|y_1 - v_1^1| \leq \delta$. Since $|v_0^1 - v_1^1| \leq \delta$, we have $|y_0 - y_1| \leq 2\delta$.

Now assume that in both cases the min is attained in the first term. In particular, $c^1\epsilon \leq |v_1^1|$ and $c^1\epsilon \leq |v_0^1|$. We have,

$$\begin{aligned}
|y_0 - y_1| &= \left| v_0^1 + \frac{v_0^0 - v_0^1}{|v_0^0| + |v_0^1|}\left(|v_0^1| - c^1\epsilon\right) - v_1^1 - \frac{v_1^0 - v_1^1}{|v_1^0| + |v_1^1|}\left(|v_1^1| - c^1\epsilon\right) \right| \\
&\leq |v_0^1 - v_1^1| + \left| \frac{v_0^0 - v_0^1}{|v_0^0| + |v_0^1|}\left(|v_0^1| - c^1\epsilon\right) - \frac{v_1^0 - v_1^1}{|v_1^0| + |v_1^1|}\left(|v_1^1| - c^1\epsilon\right) \right| \\
&\leq \delta + \left| \frac{v_0^0 - v_0^1}{|v_0^0| + |v_0^1|}\left(|v_0^1| - c^1\epsilon\right) - \frac{v_1^0 - v_1^1}{|v_1^0| + |v_1^1|}\left(|v_1^1| - c^1\epsilon\right) \right| \\
&\leq \delta + \left| \frac{|v_0^1|(v_0^0 - v_0^1)}{|v_0^0| + |v_0^1|} - \frac{|v_1^1|(v_1^0 - v_1^1)}{|v_1^0| + |v_1^1|} \right| \\
&\quad + \left| \frac{v_0^0 - v_0^1}{|v_0^0| + |v_0^1|}c^1\epsilon - \frac{v_1^0 - v_1^1}{|v_1^0| + |v_1^1|}c^1\epsilon \right| \\
&\leq 4\delta + c^1\epsilon \left| \frac{v_0^0 - v_0^1}{|v_0^0| + |v_0^1|} - \frac{v_1^0 - v_1^1}{|v_1^0| + |v_1^1|} \right|, \qquad \text{by Claim 9.1,} \\
&\leq 4\delta + c^1\epsilon \frac{2\delta}{\min\left(|v_0^1| + |v_0^0|, |v_1^1| + |v_1^0|\right)}, \qquad \text{by Claim 9.2,} \\
&\leq 4\delta + c^1\epsilon \frac{2\delta}{\min\left(|v_0^1|, |v_1^1|\right)} \\
&\leq 4\delta + c^1\epsilon \frac{2\delta}{c^1\epsilon} \leq 6\delta.
\end{aligned}$$

□

10. *Discussion and Further Research*

We have presented a relatively fast, $O(\log n)$ time, wait-free algorithm for n-process approximate agreement. This shows that wait-free algorithms for approximate agreement can be fast, but not as fast as the best non-wait-free algorithms for this problem: we have shown that $\log n$ is a lower bound on the time complexity of any wait-free approximate agreement algorithm, while there exists an $O(1)$ time non-wait-free algorithm.

Using the emulators of [Attiya et al., 1990], our algorithms can be translated into algorithms that work in message-passing systems. The algorithms have the same time complexity (in complete networks) and are resilient to the failure of a majority of the processes.

There are many ways in which our work can be extended. An interesting direction is to consider the impact on our results of using other shared memory primitives. For example, if powerful *Read-Modify-Write* registers are used, then a constant time wait-free approximate agreement algorithm can be devised. What happens if *multi-writer* multi-reader registers are used? The existence of faster wait-free algorithms using these primitives will imply a lower bound on the *time complexity* (in normal executions) of any implementation of multi-writer registers from single-writer registers.

Another avenue of research is to see whether the techniques presented in this paper, both for algorithms and lower bounds, can be applied to other problems. We believe, for example, that the $O(1)$ time algorithm for two-process approximate agreement can be generalized to *any* decision problem of size 2, using the characterization result of [Biran et al., 1990]. It is interesting to explore whether similar results can be proved for problems that require repeated coordination (e.g., *l-exclusion*).

Finally, there remains the fundamental unanswered question raised by this work: Can wait-free (highly resilient) computation be performed at the price of no more than a logarithmic slowdown? Even more strongly, are there $O(\log n)$ time wait-free algorithms for *all* problems that have wait-free solutions?

Since the preliminary presentation of our work, first steps have been made towards answering this question in the context of randomized computation [Saks et al., 1991]. Based on the alternated-interleaving method presented in Section 6.2, Saks et al. [1991] are able to show that any *decision problem* that has a wait-free or expected wait-free[11] solution algorithm, has an expected wait-free algorithm with the same worst-case time complexity, that takes only $O(\log n)$ expected time[12] in fault-free executions. However, the above question itself is still far from being answered.

ACKNOWLEDGMENTS. We would like to thank Jennifer Welch, Ophir Rachman, Marios Mavronicolas, and the anonymous referees for reading earlier versions of the paper and for many helpful comments. Thanks are also due to Cynthia Dwork, Maurice Herlihy, Mike Saks, Marc Snir, and Heather Woll, for helpful discussions on the topic of this paper.

[11] An expected wait-free algorithm is a randomized algorithm that is only expected, rather than guaranteed, to terminate within a finite number of steps.

[12] This is optimal by a straightforward extension of our lower bound to the case of randomized computation (see Saks et al. [1991]).

REFERENCES

AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1993. Atomic snapshots of shared memory. *J. ACM 40*, 4 (Sept.), 873–890.

ANDERSON, J. 1990. Composite registers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Dıstributed Computıng* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, pp. 15–30.

ARJOMANDI, E., FISCHER, M. J., AND LYNCH, N. A. 1983. Efficiency of synchronous versus asynchronous distributed systems. *J. ACM 30*, 3 (July), 448–456.

ASPNES, J., AND HERLIHY, M. 1990. Fast randomized consensus using shared memory. *J. Algorithms 11* (Sept.), 441–461.

ATTIYA, H., BAR-NOY, A., AND DOLEV, D. 1990. Sharing memory robustly in message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Princıples of Dıstributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, pp. 363–376. (Expanded version: Technical Memo MIT/LCS/TM-423, Laboratory for Computer Science, MIT, February 1990.)

ATTIYA, H., DOLEV, D., AND SHAVIT, N. 1989. Bounded polynomial randomized consensus. In *Proceedings of the 8th Annual ACM Symposium on Prınciples of Distributed Computıng* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 281–293.

ATTIYA, H., AND LYNCH, N. 1994. Time bounds for real-time process control in the presence of timing uncertainty. *Inf. Comput., 110*, 1, 183–232. (Also in *Proceedings of the 10th IEEE Real-Time Systems Symposium* (Santa Monica, Calif., Dec.). IEEE, New York, 1989, pp. 268–284.

BIRAN, O., MORAN, S., AND ZAKS, S. 1990. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *J. Alg. 11* (Sept.) 420–440.

COAN, B., AND DWORK, C. 1986. Simultaneity is harder than agreement. In *Proceedıngs of the 5th IEEE Symposium on Reliabılity in Distributed Software and Database Systems*. IEEE, New York, pp. 141–150.

COOK, S., DWORK, C., AND REISCHUK, R. 1986. Upper and lower time bounds for parallel RAMS without simultaneous writes. *SIAM J. Comput. 15*, 1, 87–98.

COLE, R., AND ZAJICEK, O. 1989. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures* (Santa Fe, N.M., June 18–21). ACM, New York, pp. 169–178.

COLE, R., AND ZAJICEK, O. 1990. The expected advantage of asynchrony. In *Proceedıngs of the 2nd ACM Symposium on Parallel Algorithms and Architectures* (Island of Crete, Greece, July 2–6). ACM, New York, pp. 85–94.

DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan.), 77–97.

DOLEV, D., GAFNI, E., AND SHAVIT, N. 1988. Toward a non-atomic era: *l*-exclusion as a test case. In *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, (Chicago, Ill., May 2–4). ACM, New York, pp. 78–92.

DOLEV, D., LYNCH, N. A., PINTER, S. S., STARK, E. W., AND WEIHL, W. E. 1986. Reaching approximate agreement in the presence of faults. *J. ACM 33*, 3 (July), 499–516.

DOLEV, D., REISCHUK, R., AND STRONG, H. R. 1982. Eventual is earlier than immediate. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 196–203.

DWORK, C., AND MOSES, Y. 1990. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Inf. Comput., 88*, 2, 156–186.

DWORK, C., AND SKEEN, D. 1983. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 1–11.

FEKETE, A. 1987a. Asymptotically optimal algorithms for approximate agreement. *Dist. Comput. 4*, 1, 9–30.

FEKETE, A. 1987b. Asynchronous approximate agreement. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 64–76.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty processor. *J. ACM 32*, 2 (Apr.), 374–382.

GIBBONS, P. 1989. A more practical PRAM model. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Fe, N.M., June 18–20). ACM, New York, pp. 158–168.

Herlihy, M. P. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan.), 124–129.

Kanellakis, P. C., and Shvartsman, A. A. 1989. Efficient parallel algorithms can be made robust. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 211–221.

Kedem, Z. M., Palem, K. V., and Spirakis, P. G. 1990. Efficient robust parallel computations. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (Baltimore, Md., May 12–14). ACM, New York, pp. 138–148.

Lamport, L. 1976. The synchronization of independent processes. *Acta Inf. 7*, 1, 15–34.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3*, 2 (Mar.), 125–143.

Lamport, L. 1986a. On interprocess communication. Part I: Basic formalism. *Dist. Comput. 1*, 2, 77–85.

Lamport, L. 1986b. On interprocess communication. Part II: Algorithms. *Dist. Comput. 1*, 2, 86–101.

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst. 4*, 3 (July), 382–401.

Lampson, B. 1983. Hints for computer system design In *Proceedings of the 9th Annual ACM Symposium on Operating Systems Principles*. ACM, New York, pp. 33–48.

Li, M., Tromp, J., and Vitanyi, P. M. B. 1989. How to share concurrent wait-free variables. *Proceedings of ICALP 89*. Lecture Notes in Computer Science, vol. 372. Springer-Verlag, New York, pp. 488–505. (Expanded version: Report CS-R8916, CWI, Amsterdam, April 1989.)

Loui, M., and Abu-Amara, H. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Adv. Comput. Res. 4*, 163–183.

Lynch, N., and Fischer, M. 1981. On describing the behavior and implementation of distributed systems. *Theoret. Comput. Sci 13*, 1 (Jan.), 17–43.

Lynch, N., and Goldman, K. 1989. Lecture notes for 6.852 MIT/LCS/RSS-5. Laboratory for Computer Science, MIT, Cambridge, Mass.

Mahaney, S. R., and Schneider, F. B. 1985. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug 5–17). ACM, New York, pp 237–249.

Martel, C., Park, A., and Subramonian, R. 1989. Optimal asynchronous algorithms for shared memory parallel computers. Tech. Rep. CSE-89-8. Division of Computer Science, University of California at Davis, Davis, Calif., July.

Martel, C., Subramonian, R., and Park, A. 1990. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 590–599.

Merritt, M., Modugno, F., and Tuttle, M. 1991. Time constrained automata. In *Proceedings of the 2nd CONCUR* (Amsterdam, the Netherlands, Aug.) Lecture Notes in Computer Science, vol. 527. Springer-Verlag, New York.

Moses, Y., and Tuttle, M. 1988. Programming simultaneous actions using common knowledge. *Algorithmica 3*, 121–169.

Nishimura, N. 1990. Asynchronous shared memory parallel computation. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures* (Island of Crete, Greece, July 2–6). ACM, New York, pp. 76–84.

Peterson, G. L. 1983. Concurrent reading while writing. *ACM Trans. Prog. Lang. Syst. 5*, 1 (Jan.) 46–55.

Peterson, G., and Burns, J. 1987. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 383–392.

Peterson, G. L., and Fischer, M. J. 1977. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing* (Boulder, Colo., May 2–4). ACM, New York, pp. 91–97.

Saks, M., Shavit, N., and Woll, H. 1991. Optimal time randomized consensus—Making resilient algorithms fast in practice. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, Calif., Jan. 28–30). ACM, New York, pp. 351–362.

Schaffer, R. 1988. On the correctness of atomic multi-writer registers. MIT/LCS/TM-364. MIT, Cambridge, Mass.

SKEEN, D. 1982. Crash recovery in a distributed database system. Memorandum No. UCB/ERL M82/45, Electronics Research Laboratory, Berkeley, Calif., May.

VITANYI, P., AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 233–243.

WYLLIE, J. 1979. The complexity of parallel computation. Ph.D. thesis. Cornell Univ., Ithaca, N.Y. Technical Report TR 79-387, Department of Computer Science.

RECEIVED JULY 1991; REVISED APRIL 1993; ACCEPTED MAY 1993