Are Wait-Free Algorithms Fast?

(Extended Abstract)

Hagit Attiya[†] Nancy Lynch[†] Nir Shavit[‡]

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.

1 Introduction

In shared-memory distributed systems, n 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, e.g., 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 ([11, 14]). The design of wait-free algorithms has been a very active area of research recently (see [11]).

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, i.e., in *failure-free* executions. The time measure we use is the one introduced in [13], and used to evaluate the time complexity of asynchronous algorithms (for example in [2, 6, 17, 19]). 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.

[*]This work was supported by ONR contract N00014-85-K-0168, by NSF grants CCR-8611442 and CCR-8915206, and by DARPA contracts N00014-89-J-1988 and N00014-87-K-0825.

[†]Laboratory for Computer Science, MIT.

[‡]IBM Almaden Research Center and Stanford University. Part of this work was performed while the author was at the Hebrew University and visiting the Laboratory for Computer Science, MIT.

In this paper, we address the general question by considering a specific problem—the *approximate agreement* problem studied, for example, in [7]; 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., [10]), is closely related to the important problem of synchronizing local clocks in a distributed system.

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. 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 $1/\varepsilon$. 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, i.e., it does *not* depend on the range of inputs or on ε (Section 4). The algorithm uses a novel method of overcoming the uncertainty that is inherent in an asynchronous environment, without resorting to synchronization points or other waiting mechanisms (cf. [6]): 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 5). 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 6).[1] 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. In the full paper we prove, for any wait-free approximate agreement algorithm, a tradeoff between the time complexity and the number of operations performed by a single process.

In practice, the design of distributed systems is often geared towards optimizing the time complexity in "normal executions," i.e., executions where no failures occur and processes run at approximately the same pace, while building in safety provisions to protect against failures (cf. [15]). Our results indicate that, in the asynchronous shared-memory setting, there are problems for which building in such safety provisions *must* result in 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., [8]) that tolerate failures, yet terminate in *constant* time when no failures occur.

Recent work has addressed the issue of adapting the usual synchronous shared-memory PRAM model to better reflect implementation

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

issues, by reducing synchrony (see [6]) or by requiring fault-tolerance (see [12]). 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. In [18], Martel, Subramonian and Park present efficient fault-tolerant asynchronous PRAM algorithms. Their algorithms optimize total number of operations 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.

2 Model and Time Measure

In this section we briefly describe the systems and the time measure we will consider.[2] A *system* consists of n processes $p_0, \ldots, p_{n-1}$. Each process is a deterministic state machine, with possibly an infinite number of states. Processes communicate by reading and writing to *single-writer multi-reader atomic registers.* Processes may fail, that is, undetectably stop executing operations from some point on (a *fail-stop* failure). Due to asynchrony, other processes cannot distinguish between a fail-stopped process and a slow one. An algorithm that guarantees that each non-faulty process completes its execution, even in the face of up to $n-1$ failures is called a *wait-free* algorithm.

We now define how to measure the *time* a computation takes. We assign times to events (each event is either a *read* or a *write* operation) in a given execution subject to the following constraints: *(a)* the time associated with the first event of any process is at most 1, and *(b)* the time between two events of the same process is at most 1. The time of a finite execution is the largest amount of real time that can be associated with the last event in that execution. The time between two events in an execution is the largest amount of real time that can elapse between these two events under any time assignment to this execution. Loosely speaking, this amounts to normalizing the time relative to the rate of the slowest process. For example, if in a given time interval one process takes 1 step and another takes 100, the time measured is 1. (This definition is formalized in [17]; an equivalent definition appears in [13] and is used in [6].) 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.

[2]Our definitions are standard and are similar to the ones in, e.g., [2, 11, 14, 17].

3 Basic Solutions

We start by defining the *approximate agreement* problem and describing non-wait-free and wait-free algorithms to solve it. Later in the paper (Section 5), both algorithms are used as "subroutines" in the logarithmic time approximate agreement algorithm.

In the *approximate agreement* problem, processes start with real-valued inputs, $x_0, \ldots, x_{n-1}$, and a constant $\varepsilon > 0$ (the same ε for all processes); all nonfaulty processes are required to decide on real-valued outputs $y_0, \ldots, y_{n-1}$, such that the following conditions hold:

Agreement: $|y_i - y_j| \leq \varepsilon$, for all i, j, and

Validity: $y_i \in range(\{x_0, \ldots, x_{n-1}\})$, for all i.

This problem has a simple $O(1)$ time non-wait-free solution, which we will call wait-approx. A designated 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, and on which it decides. All processes repeatedly read V_0 until it is set to a value that is not $\perp$ and decide on this value.

We next outline a wait-free algorithm wait-free-approx for approximate agreement. For lack of space we only outline a simple variant for the case of two processes. Each of the processes p_i, $i \in \{0,1\}$ has a register which it can write and the other can read. Here and elsewhere, we let $\bar{\imath}$ denote the index of the other process, i.e., $\bar{\imath} = 1 - i$. Due to the asynchrony in the system, it is impossible to have processes agree on one of the input values (see [10, 9, 16]). 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 reads $p_{\bar{i}}$'s register. If p_i reads $\perp$ from $v_{\bar{i}}$, it can never know when $p_{\bar{i}}$ will write its input value ($p_{\bar{i}}$ could fail and never write); thus, p_i must decide on its own value. If p_i reads a non-$\perp$ value from $v_{\bar{i}}$, it checks if $|v_{\bar{i}} - v_i| \leq \varepsilon$. If so, p_i decides on its own value. If not, p_i sets v_i to be $\frac{v_{\bar{i}}+v_i}{2}$ and repeats.

The following scenario is possible: $p_{\bar{i}}$ reads p_i's input value and is delayed just before writing $\frac{v_{\bar{i}}+v_i}{2}$ to its register; then p_i repeatedly reads and writes, cutting the interval in half till its value is very close to $p_{\bar{i}}$'s input; finally, $p_{\bar{i}}$ completes the write so that in fact, p_i moved "too far" towards $p_{\bar{i}}$'s old value. However, 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 $O(\log(\frac{x_{\bar{i}}-x_i}{\varepsilon}))$ time. 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 similar in flavor, but more complicated in the mechanisms used to synchronize among processes. It uses the rounds structure of [3]. In place of every read or write of the two process algorithm, it uses a single-writer atomic snapshot memory operation [1] that takes $O(n^2)$ time. In the full paper we present this algorithm and prove:

Theorem 3.1. *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*

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

4 Fast 2-Process Algorithm

We now show that, for two processes, there exists an approximate agreement algorithm whose time complexity is constant; i.e., it does *not* depend on the range of input values or ε. Key ideas from this algorithm will be used in the n process algorithm of Section 5. The intuition underlying this algorithm is as follows: It is clear from the definition of time that as long as p_i reads $p_{\bar{i}}$'s register and sees no change, either p_i is running very fast (and not much time has elapsed), or $p_{\bar{i}}$ has failed. Thus, p_i may continue to take steps as long as it sees no change, without degrading the time complexity for failure-free executions. 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. Thus, in order to get the required constant time complexity, it is necessary that p_i know to within ε what value $p_{\bar{i}}$ will decide on, if and when $p_{\bar{i}}$ does appear; similarly for $p_{\bar{i}}$. However, an inherent difficulty of the model is that a process p_i cannot know if $p_{\bar{i}}$ read its latest written value or the value written in p_i's preceding write. Taking the midpoint of the input range (as in the former 2-process algorithm) will not help, since the uncertainty among the two writes of p_i could mean a difference of $\frac{x_{\bar{i}}-x_i}{2}$ in $p_{\bar{i}}$'s decision, a difference possibly greater than ε. Our solution is to have the decisions change only gradually with each step, so that this uncertainty will result only in ε inaccuracy.

More specifically, each process p_i starts by writing its input and initializing a counter c_i in the shared memory. It then keeps incrementing this counter until either it has taken $\lceil |x_i|/\varepsilon \rceil$ steps without seeing the other and decides on its own value x_i, or $p_{\bar{i}}$ has taken a step, in which case p_i makes a decision that depends on the relative size of both processes counters, both input values, and ε. The decision is biased towards the input value of the process with the larger counter, in quanta that are less than or equal to ε. The exact decision value is given by the function bias (Figure 1).

The code for fast-2-approx for process p_i is given in Figure 2. The inputs to each process p_i are real numbers x_i and ε.[3] Each process maintains a *single-writer multi-reader atomic regis-*

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

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

Figure 1: The bias function.

```
function fast-2-approx ($x$, $\varepsilon$);
    begin
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 $c^{\bar{i}} = \bot$ then return $v^i$
4:          else return bias ($v^0, v^1, c^0, c^1, \varepsilon$);
    end;

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

Figure 2: Fast 2-process algorithm

ter with two fields: V_i—the input value, a real number, and C_i—the counter, an integer. 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. Upper case variables denote shared variables, while all lower case variables are local, and are usually local versions of the corresponding upper case shared variable. The algorithm ensures the following:

Lemma 4.1. *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 $p_{\bar{i}}$, resp., in Line 2 of* fast-2-approx. *Then, $c_i - 1 \le c_{\bar{i}} \le c_i$.* ■

This implies that values returned by bias based on the values of the counters differ by at most ε.

Theorem 4.2. *There exists a wait-free algorithm for the 2-process approximate agreement problem whose time complexity is $O(1)$.* ■

5 Fast n-Process Algorithm

In this section, we present a fast ($O(\log n)$ time) wait-free approximate agreement algorithm for n processes. We begin by showing how one can reduce, in constant time, the problem of n-process approximate agreement with n 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 then show 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 of Section 4 on two input values. Combining the two algorithms yields an $O(\log n)$ wait-free approximate agreement algorithm.

The algorithm uses an *alternated-interleaving* method of combining wait-free (resilient but slow) and non-wait-free (fast but not resilient) algorithms to obtain new algorithms that are both resilent and fast. 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 when either of them terminates.[4] When an alternation is used in an assignment statement, the value assigned is the value returned by the procedure that terminates first. It is straightforward to show that the time for execution of an alternation of two procedures is at most twice that of the fastest one.

The constant time reduction to two small intervals is encapsulated in procedure n-to-2 (Figure 3). The idea is simple: interleave the execution of the slow wait-free-approx procedure with that of the fast wait-approx. The resulting algorithm is wait-free since even if $n-1$ processes fail, wait-free-approx will terminate. It takes at most

[4]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.

$O(1)$ 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. We thus did not solve the approximate agreement problem, but we did guarantee that the values are included in the union of two small intervals. The procedure 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 $\varepsilon/12$ 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 value in each group. The solution is based on having the processes in group a (resp. b) jointly simulate a virtual process p_0 (resp. p_1) that execute the function fast-2-approx of Figure 2.

The following straightforward simulation is expressed by Lines 1-2 of the function increase-counter in Figure 3. 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 true, 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 2-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 5.2, whose termination ensures this condition. To acheive 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 5.2, 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.

5.1 The Algorithm

In addition to the shared data structures used by wait-free-approx and wait-approx, process $p_i, i \in \{0, \ldots, n-1\}$ 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 increase-counter termination flag.

In the code 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

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

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

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

Figure 3: Fast n-process algorithm

```
function fast-collect $(R)$;
  begin
1:  $l := 1$;
2:  while $l < n$ do
3:    $R_i :=$ concatenate $(R_i, R_{(i+l) \bmod n})$;
4:    $l := |R_i|$;
    od;
5:  return truncate$(R_i, n)$;
  end;
```

Figure 4: Fast information collection.

that p_j is in group g if there is any, and $\perp$, otherwise. The value v^g is calculated in a similar manner from the corresponding local copies. (Recall our convention that lower case letters stand for local variables and upper case letters for shared variables.) When g is a group name, $\bar{g}$ denotes the other group's name, e.g., 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.

5.2 Fast Information Collection and Synchronization

Before presenting the proof of correctness for the algorithm we present the procedures for information collection and synchronization. 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, e.g., max or sum, on a set of initial values that reside in the shared memory.[5] 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., [5]), 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 4, is a wait-free variation of the *pointer-jumping* technique used in PRAM algorithms (e.g., [20]). For a sequence R and a nonnegative integer n we define truncate(R, n) to be 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. In the full version of the pa-

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

```
procedure synch($R$);
  begin
    repeat until $R_i \neq \bot$;
    $l := 1$;
    while $l < n$ and $T_{i+l \bmod n} \neq \bot$ do
      repeat until $R_{i+l \bmod n} \neq \bot$;
      $R_i :=$ concatenate $(R_i, R_{(i+l) \bmod n})$;
      $l := |R_i|$;
    od;
  end;
```

Figure 5: Fast synchronization.

per we prove that any invocation of fast-collect returns within $O(\log n)$ time after all processes invoke fast-collect and that the values returned by fast-collect are not too old.

The synchronization procedure, synch, is a variant of fast-collect. Since it is used within an **alternate** construct, it is possible that synch is aborted without completing and returning "normally." To cope with this possibility, we associate with the shared array R to which synch is applied, a special *termination* array T. T_j is set to *true* if p_j terminates, i.e., aborts or returns from synch$_j$. Procedure synch guarantees that, if it is completed by some process, then either all the entries of the array are non-$\bot$ values, or for some j, T_j = *true*. The procedure is *not* wait-free. The code appears in Figure 5. In the full version of the paper we prove that any invocation of synch terminates within $O(\log n)$ time after all processes set $R_i \neq \bot$ and invoke synch.

5.3 Correctness Proof

As in the proof of the 2-process algorithm (Section 4), 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 maximal value this counter ever attains. This implies that, for each local counter, the values read by two different processes differ at most by 3. Thus, we have:

Lemma 5.1. *Suppose $i, j \in \{0, \ldots, n-1\}$ and $g \in \{a, b\}$. Assume the values returned by* fast-collect$_i$ *and* fast-collect$_j$ *calculate to c_i^g and c_j^g, respectively. Then $|c_i^g - c_j^g| \leq 3n$.* ∎

Using this lemma and the fact that when bias is applied to "close" real numbers and "close" counters it returns "close" values, we can prove that the algorithm satisfies the agreement property.

Theorem 5.2. *There exists a wait-free algorithm for the n-process approximate agreement problem whose time complexity is $O(\log n)$.* ∎

6 A $\log n$ Time Lower Bound

In this section, we show that the $\log n$ dependency exhibited by the algorithm of Theorem 5.2 is inherent: the time complexity of any wait-free algorithm for n-process approximate agreement is at least $\log n$. Together with wait-approx, 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.

We require a few definitions. 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. Let a system configuration consist of the states of the processes and the registers. Formally, a *configuration C* is a vector $\langle s_0, \ldots, s_{n-1}, v_1, \ldots \rangle$ where s_i is the state of process p_i and v_j is the value of the register R_j. An *initial configuration* is a configuration in which every process's state is an initial state and all shared variables are set to $\bot$. For a configuration C and a process p_i, let $st(p_i, C)$ be the pair consisting of the state of p_i and the value of R_i in C, i.e., $st(p_i, C) = \langle state(p_i, C), val(R_i, C) \rangle$. For the rest of this section, fix some $\varepsilon < 1$.

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. We denote the configuration resulting from the application of such a finite schedule σ_r to a configuration C by $C\sigma_r$. For any configuration C, the corresponding *synchronized execution* from C (denoted (C, σ_r)) 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 process p_i:

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 $INF(p_i, r, C) = INF(p_i, r-1, C)$.

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

Lemma 6.1. *Let C_1 and C_2 be two configurations, let p_i be any process and let $r \geq 0$. If $st(p_j, C_1) = st(p_j, C_2)$ for all $p_j \in INF(p_i, r, C_1)$, then $st(p_i, C_1\sigma_r) = st(p_i, C_2\sigma_r)$.*

Theorem 6.2. *Any wait-free algorithm for the n-process approximate agreement problem has time complexity at least $\log n$.*

Proof sketch: Assume that A is a wait-free approximate agreement algorithm. Suppose, by way of contradiction, that in all failure-free executions some process decides before time $\log n$.

Let σ be the infinite synchronized schedule, i.e., the limit of σ_r. Consider the execution of A under σ 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); without loss of generality, let p_0 be the process associated with this event. By assumption, $t < \log n$. By the validity property, p_0 must decide on 0 since all processes start with 0. Note that, by induction on r, $|INF(p_i, r, C)| \leq 2^r$, for every configuration C, $r \geq 0$ and $i \in \{0, \ldots, n-1\}$. Since $t < \log n$ it must be that $|INF(p_0, T, C_0)| \leq 2^T < n$. Thus, there exists some process that is not in $INF(p_0, t, C_0)$; without loss of generality, assume $p_{n-1} \notin |INF(p_0, T, C_0)|$.

Informally, to complete the proof, we create an alternative execution in which p_{n-1} "starts early" with input 1, runs on its own and thus must eventually decide on 1. We then let the rest of the processes execute as if they are in the synchronized execution from C_0 and use Lemma 6.1 to show that process p_0 still decides on 0, which is a contradiction to the agreement property, since $\varepsilon < 1$. ■

7 Discussion and Further Research

For approximate agreement, the answer to the question whether wait-free algorithms are fast is not binary, rather it is quantitative: we have presented a relatively fast, $O(\log n)$ time, wait-free algorithm for n-process approximate agreement. On the other hand, $\log n$ is a lower bound on the time complexity of any wait-free approximate agreement algorithm, and there exists an $O(1)$ time non-wait-free algorithm.

Using the emulators of [4], 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. Another 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 2-process approximate agreement can be generalized to *any* decision problem of size 2. 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: Are there $O(\log n)$ time wait-free algorithms for *all* problems that have wait-free solutions?

Acknowledgements: We would like to thank Jennifer Welch for careful reading of an earlier version of the paper and 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.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit, "Atomic Snapshots of Shared Memory," *Proc. 9th ACM Symp. on Principles of Dist. Comp.*, 1990, to appear.

[2] E. Arjomandi, M. Fischer and N. Lynch, "Efficiency of Synchronous Versus Asynchronous Distributed Systems," *JACM,* Vol. 30, No. 3 (1983), pp. 449–456.

[3] J. Aspnes and M. Herlihy, "Fast Randomized consensus Using Shared Memory," *J. of Algorithms,* September 1990, to appear.

[4] H. Attiya, A. Bar-Noy and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Proc. 9th ACM Symp. on Principles of Dist. Comp.*, 1990, to appear.

[5] S. Cook, C. Dwork and R. Reischuk, "Upper and Lower Time Bounds for Parallel RAMS Without Simultaneous Writes," *SIAM J. Comp.*, Vol. 15, No. 1, 1986, pp. 87–98.

[6] R. Cole and O. Zajicek, "The APRAM: Incorporating Asynchrony into the PRAM model," *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures,* 1989, pp. 169–178.

[7] D. Dolev, N. Lynch, S. Pinter, E. Stark and W. Weihl, "Reaching Approximate Agreement in the Presence of Faults," *JACM,* Vol. 33, No. 3, 1986, pp. 499–516.

[8] D. Dolev, R. Reischuk and H. R. Strong, "Eventual Is Earlier Than Immediate," *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, 1982, pp. 196–203.

[9] D. Dolev, C. Dwork and L. Stockmeyer, "On the Minimal Synchrony Needed for Distributed Consensus," *JACM,* Vol. 34, No. 1 (January 1987), pp. 77–97.

[10] M. Fischer, N. Lynch and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Processor," *JACM,* Vol. 32, No. 2 (1985), pp. 374–382.

[11] M. Herlihy, "Wait Free Implementations of Concurrent Objects," *Proc. 7th ACM Symp. on Principles of Dist. Comp.*, 1988, pp. 276–290.

[12] Z. Kedem, K. Palem and P. Spirakis, "Efficient Robust Parallel Computations," *Proc. 22nd ACM Symp. on Theory of Computing,* 1990, pp. 138–148.

[13] L. Lamport, "The Synchronization of Independent Processes," *Acta Informatica,* Vol. 7, No, 1 (1976), pp. 15–34.

[14] L. Lamport, "On Interprocess Communication. Parts I & II:," *Distributed Computing 1, 2* 1986, 77–101.

[15] B. Lampson, "Hints for Computer System Design", in *Proc. 9th ACM Symposium on Operating Systems Principles,* 1983, pp. 33–48.

[16] M. Loui and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research*, Vol. 4, JAI Press, Inc., 1987, pp. 163-183.

[17] N. Lynch and M. Fischer, "On Describing the Behavior and Implementation of Distributed Systems," *Theoretical Computer Science,* Vol. 13, No. 1 (January 1981), pp. 17-43.

[18] C. Martel, R. Subramonian and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," these proceedings.

[19] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," *Proc. 9th ACM Symp. on Theory of Computing,* 1977, pp. 91–97.

[20] J. Wyllie, *The Complexity of Parallel Computation,* Ph.D. thesis, Cornell University, August 1979.