Spreading Rumors Rapidly Despite an Adversary

James Aspnes * William Hurwood†

Abstract

In the *collect problem* [32], n processors in a shared-memory system must each learn the values of n registers. We give a randomized algorithm that solves the collect problem in $O(n \log^3 n)$ total read and write operations with high probability, even if timing is under the control of a content-oblivious adversary (a slight weakening of the usual adaptive adversary). This improves on both the trivial upper bound of $O(n^2)$ steps and the best previously known bound of $O(n^{3/2} \log n)$ steps, and is close to the lower bound of $\Omega(n \log n)$ steps. Furthermore, we show how this algorithm can be used to obtain a multi-use cooperative collect protocol that is $O(\log^3 n)$-competitive in the latency model of Ajtai et al.[3] and $O(n^{1/2} \log^{3/2} n)$-competitive in the throughput model of Aspnes and Waarts [10]; in both cases we show that the competitive ratios are within a polylogarithmic factor of optimal.

1 Introduction

Rumor spreading. The simplest problem we will consider is the following: each of n people knows a rumor. At each point in time, an adversary chooses one of the n people and hands him or her a telephone. The only restriction on the adversary's choice is that he cannot choose a person who already knows all n rumors (intuitively, we assume that such a person goes off to do something else). The person chosen by the adversary may call up any one other person (possibly choosing the other person using randomization) and will learn all the rumors that the other person currently knows. The process continues until all participants know all of the rumors. Our goal is to minimize the total number of *steps* (i.e., the total number of telephone calls).

One can think of this problem as an asynchronous version of the well-known *gossip problem* [24]. In the gossip problem, n persons wish to distribute n rumors among themselves; however, which persons communicate at each time is fixed in advance by the designer of the algorithm. By contrast, in our problem, the choice of who receives information at each time is under the control of an adversary. Furthermore, the algorithm used by each process to choose where it will look for more information can only make that choice based on the information obtained so far.

The collect problem. The rumor-spreading problem above is closely related to the *collect problem* [32]. In the collect problem, each of n processes in a shared-memory system possesses some piece of information, which it stores in one of a set of single-writer multi-reader atomic registers. We would like each of the processes to learn the values of all of the others while performing as few total read and write operations as possible. Again, we assume that timing is under the control of an adversary scheduler, which has near-total knowledge of all events in the system, and which may start and stop processes at will. The essential difference between the rumor-spreading problem above and the collect problem is that in the collect problem the operations of choosing someone to read, reading his or her values, and adding them to one's own register do not take place as a single atomic action.

The description above is of the simplest version of the collect problem, in which all values are present at the start and each process gathers the values only once. For this version of the problem, the naive solution is to have each of the n processors read each of the n registers directly, for a total cost of n^2 operations. However, the naive solution is not the best possible, as processors can learn values indirectly from other processors, thus sharing the work of reading the registers. Indeed, Ajtai et al. [3] observed that the Certified Write-All algorithm of Anderson and Woll [5] could be modified in a straightforward way to solve the collect problem in $O(n^{3/2} \log n)$ total operations. This is a substantial improvement on an upper bound of n^2, but still far from the best known lower bound of $\Omega(n \log n)$. [3].

Repeated collects. The collect problem is motivated by its frequent appearance in other algorithms. Many algorithms in the wait-free shared-memory model [1, 2, 4, 6, 7, 8, 9, 12, 13, 15, 16, 17, 19, 20, 21, 22, 23, 27, 25, 26, 28, 29, 30, 34] have an underlying structure in which processes *repeatedly* collect values using the *cooperative collect* primitive. In the cooperative collect primitive, first abstracted by Saks, Shavit, and Woll [32], processes perform the *collect* opera-

*Yale University, Department of Computer Science, 51 Prospect Street/P.O. Box 208285, New Haven, CT 06520-8285. Supported by NSF grants CCR-9410228 and CCR-9415410. E-mail: aspnes@cs.yale.edu.

†Yale University, Department of Mathematics, P.O. Box 208283, New Haven CT, 06520-8283. Supported by NSF grants CCR-8958528 and CCR-9415410. E-mail: will@math.yale.edu.

tion – an operation in which each process learns the values of a set of n registers, with the guarantee that each value learned is *fresh*: it was present in the register at some point during the collect. In a sense the cooperative collect primitive is a multi-use version of the simple collect problem, with the added difficulty of guaranteeing freshness.

Interestingly, most of these algorithms (which include nearly all algorithms in the wait-free shared-memory literature for consensus, snapshots, coin flipping, bounded round numbers, timestamps, and multi-writer registers) use the naive algorithm for performing collects in which each processor reads every register directly, at a cost of n reads per collect.[1] One reason (beyond the simplicity of the naive algorithm) may be that if one considers the performance of collect algorithms in traditional worst-case terms, the naive algorithm appears to be optimal: since the adversary can always choose to halt all but one of the processors, that lone processor running in isolation cannot carry out a collect without reading all the other processor's registers.

Competitive collect algorithms The apparent optimality of the naive algorithm for *repeated* collects is surprising given the superior performance of other algorithms for the *one-time* collect problem. Indeed, one would expect that an algorithm that solved the one-time problem quickly could be extended to an algorithm that would give better performances in many circumstances. Ajtai et al. [3] provided a tool, known as *latency competitiveness*, that can be used to show the superiority of more sophisticated algorithms. In their model the performance of a distributed algorithm is not measured in absolute terms against the worst possible schedule, but instead is measured on each schedule relative to the performance of another distributed algorithm chosen to be optimal for that schedule. In order to have good latency competitiveness, an algorithm must not only perform acceptably in hard situations (for collect, this is generally when there is little or no concurrency) but must also perform well in easy situations. More details of the latency competitiveness measure, and of the related *throughput competitiveness* measure [10], can be found in Sections 4.1 and 4.2.

1.1 Our results

We describe (Section 2) an algorithm for the rumor-spreading game which requires only $O(n \log^2 n)$ steps with high probability, slightly more than the lower bound of $\Omega(n \log n)$. Based on this algorithm, we construct (Section 3) a randomized algorithm for the collect problem that runs in $O(n \log^3 n)$ work with high probability; the extra $O(\log n)$ factor comes from the technique we use to simulate an atomic transfer of information from one processor's register to another's. This is the first solution to the problem that comes within a polylogarithmic factor of the lower bound of $\Omega(n \log n)$. Furthermore, we show (Section 4) that our algorithm can be extended in a natural way to yield an implementation of the cooperative collect primitive that is $O(\log^3 n)$-competitive in the latency model [3] and $O(\sqrt{n} \log^{3/2} n)$-competitive in the throughput model. Both of these ratios are also within a polylogarithmic factor of the best known lower bounds, and substantially improve on the best previously known ratios of $O(\sqrt{n} \log n)$ and $O(n^{3/4} \log n)$.

[1] [32, 31] present collect algorithms that do not follow the pattern of the naive algorithm. Both works, however, consider models that involve considerably stronger assumptions that either the standard wait-free shared memory model or the slightly weaker model considered here.

1.2 The model

All of our results are carried out in a model where the algorithm is allowed to generate a random value and write it out as a single atomic operation. This assumption appears frequently in early work on consensus; it is the "weak model" of Abrahamson [1] and was used in the consensus paper of Chor, Israeli, and Li [19]. In general, the weak model in its various incarnations permits much better algorithms (e.g., [11, 18]) for such problems as consensus than the best known algorithms in the more traditional "strong model". The assumption that the adversary cannot see coin-flips before they are written is justified by an assumption that in a real system failures, page faults, and similar disastrous forms of asynchrony are likely to be affected by *where* each processor is reading and writing values but not by *what* values are being read or written.

It is not clear whether this assumption can be removed while still permitting an $O(n \log^c n)$ solution to the collect problem.

2 Spreading rumors

Recall from the introduction that in the rumor-spreading problem a processor may choose what processor it will read, read that processor's state, and add the information thus obtained to its own visible state as a single atomic operation. The algorithm we analyze in this case is deceptively simple: when a processor a is chosen to move by the adversary, it reads from a processor b chosen uniformly at random from the set of all n processors. (It is possible that $b = a$.) We will refer to one of these atomic operations as a *move*.

Intuitively it seems unlikely that this is the best algorithm. For example, if a has obtained the information from $n - 1$ processors, it is clear that a should examine the sole processor whose information a does not already possess. Also if $b = a$ then no information can possibly be gained. But this algorithm has the great advantage that it is impossible for the adversary to bias a's selection of b. This makes it much easier to analyze the performance of this algorithm than it otherwise might be.

Some notation: in the following, we will use K_t^P for the set of rumors possessed by processor P at time t. We will say that a processor P *knows* a set of rumors S at time t when $S \subseteq K_t^P$. The effect of P reading Q at time t is to set K_{t+1}^P to $K_t^P \cup K_t^Q$.

Let us look at some set of rumors S and consider how they spread through the processors. For each S, we will divide moves into two classes:

- Moves by processors that already know S. We will call these moves *unproductive* (with respect to S).
- Moves by processors that do not already know S. We will call these moves *productive* (again, with respect to S).

Where it will not cause confusion we will omit a specific reference to S. Note however that a move might be unproductive with respect to some S but productive with respect to a different S'.

The following lemma shows that, with high probability, the information known by any single processor spreads to all of the processors after only $O(n \log n)$ productive moves:

Lemma 1 *Fix a starting time s and let $S = K_s^P$. Let T be the number of productive moves after s before every processor knows S and let k be a positive constant. Then*

$$Pr[T \geq kn \ln n] \leq \frac{1}{n^{k-3}}$$

Proof: If k processors know S prior to a productive move, then the probability that $k+1$ processors know S after the move is k/n. Thus the total waiting time T is given by the sum of a set of independent, geometrically distributed random variables $T_1, T_2, \ldots, T_{n-1}, T_n$ with expectations $n, n/2, \ldots, n/(n-1), 1$. This gives a total expected time of nH_n or approximately $n \ln n$. However, we wish to show a stronger claim, by bounding the tail of this sum's distribution.

We do so using moment generating functions. We have by Markov's inequality that for $c > 0$

$$\Pr[T \geq t] = \Pr\left[e^{cT} \geq e^{ct}\right] \leq \frac{\mathrm{E}\left[e^{cT}\right]}{e^{ct}}.$$

Since the T_i are independent

$$\mathrm{E}\left[e^{cT}\right] = \mathrm{E}\left[\prod_{i=1}^{n-1} e^{cT_i}\right] = \prod_{i=1}^{n-1} \mathrm{E}\left[e^{cT_i}\right].$$

We can evaluate $\mathrm{E}\left[e^{cT_i}\right]$ directly. Let $p = i/n$ and $q = 1 - i/n$. Then assuming that $qe^c < 1$ we get

$$\mathrm{E}\left[e^{cT_i}\right] = p \sum_{j=1}^{\infty} q^{j-1} e^{cj} = pe^c \sum_{j=0}^{\infty} (qe^c)^j = \frac{pe^c}{1 - qe^c}.$$

So if we let $d = e^c$ we get

$$\begin{aligned}
\Pr[T \geq t] &\leq \frac{1}{e^{ct}} \prod_{i=1}^{n-1} \frac{\frac{i}{n} e^c}{1 - (1 - \frac{i}{n})e^c} \\
&= \frac{1}{d^t} \prod_{i=1}^{n-1} \frac{id}{id + n - dn} \\
&= \frac{(n-1)!}{d^{t-n+1}} \prod_{i=1}^{n-1} \frac{1}{id + n - dn}
\end{aligned}$$

Since we want $qd < 1$ for all possible values of q, we have that $1 < d < n/(n-1)$. For d in this range and $1 < i < n$ we have

$$\frac{1}{id + n - dn} \leq \frac{n-1}{in + n(n-1) - n^2} \leq \frac{1}{i-1}.$$

Hence

$$\begin{aligned}
\Pr[T \geq t] &\leq \frac{(n-1)!}{d^{t-n+1}} \cdot \frac{1}{d + n - dn} \cdot \frac{1}{(n-2)!} \\
&= \frac{n-1}{d^{t-n+1}(d + n - dn)}.
\end{aligned}$$

Now set $d = \frac{n}{n-1} \cdot \frac{s}{s+1}$ where $s = t - n + 1$. Define l by $s = ln \ln n$. Then

$$\begin{aligned}
\Pr[T \geq t] &\leq \frac{(n-1)d^{-s}}{n + d(n-1)} \\
&= \frac{n-1}{n} \left(\frac{n-1}{n}\right)^{ln \ln n} \left(\frac{s+1}{s}\right)^{s} (s+1) \\
&\leq n^{-l} e(ln \ln n + 1).
\end{aligned}$$

Now let $t = kn \ln n$ for k some positive constant. Then $k \geq l \geq k - 1$. Assuming that n is large enough that $n^2 \geq e(kn \ln n + 1)$ we conclude

$$\Pr[T \geq kn \ln n] \leq \frac{1}{n^{k-3}} \tag{1}$$

■

What Lemma 1 tells us is that with high probability, after $kn \ln n$ productive moves K_s^P will spread to all of the processors. Thereafter any further moves must be unproductive moves. So if $3kn \ln n$ moves are performed, at least $\frac{2}{3}$ of them are unproductive— in other words, most of these $3kn \ln n$ moves are made by processors that know K_s^P. That this intuition is true simultaneously for all P with high probability is captured in the following lemma:

Lemma 2 *Let s be a time and let $t = s + 3kn \ln n$. For each processor P and time t', let $V_{t'}^P$ be the set of processors Q for which $K_{t'}^Q \supset K_s^P$. (Thus V_t^P will consist of all processors that know after an interval of $3kn \ln n$ steps everything that P knew at the beginning.) For any set of processors A, define $w(A)$ to be the number of moves made by processors in A between s and t. Then*

$$Pr\left[\exists P \; w(V_t^P) < 2kn \ln n\right] \leq \frac{1}{n^{k-4}}$$

Proof: The proof works by showing an upper bound on the number of moves *not* done by processors in V_t^P. Let $\overline{V_t^P}$ be the complement of V_t^P. Since any processor in $\overline{V_t^P}$ does not know K_s^P at time t, it cannot have known K_s^P at any time before t, and thus all of its moves prior to t are productive moves with respect to K_s^P. Using lemma 1 we get $\Pr\left[w(\overline{V_t^P}) \geq kn \ln n\right] \leq 1/n^{k-3}$. Thus:

$$\begin{aligned}
\Pr\left[\exists P \; w(V_t^P) < 2kn \ln n\right] &\leq \sum_{i=0}^{n} \Pr\left[w(V_t^P) < 2kn \ln n\right] \\
&= n \Pr\left[w(\overline{V_t^P}) \geq kn \ln n\right] \\
&\leq \frac{1}{n^{k-4}}
\end{aligned}$$

■

Because it is likely that V_t^P and V_t^Q both do at least $\frac{2}{3}$ of the work, it is likely that these sets overlap for any P and Q, i.e. that the information known by any *pair* of processors at time s is known to a *single* processor at time $s + 3kn \ln n$:

Corollary 3 *Using the notation of Lemma 2, $Pr\left[\exists P, Q \; V_t^P \cap V_t^Q = \emptyset\right] \leq 1/n^{k-4}$.*

Proof: Suppose $w(V_t^P) \geq 2kn \ln n$ and $w(V_t^Q) \geq 2kn \ln n$ Then $w(\overline{V_t^Q}) < kn \ln n$ and so $V_t^P \not\subseteq \overline{V_t^Q}$ implying that $V_t^P \cap V_t^Q \neq \emptyset$. By Lemma 2 the probability that the supposition does *not* hold is at most $1/n^{k-4}$. The result follows. ■

In particular, if at time s there is some set A of r processors that between them know all the rumors (i.e., $\bigcup_{P \in A} K_s^P \supset \bigcup_P K_0^P$), then at time $s + 3kn \ln n$ there will be a set of $\lceil r/2 \rceil$ processors that between them know all the rumors. Initially there are n processors that between them know all the rumors. Therefore after at most $1 + \log_2 n$ intervals of length $3kn \ln n$ there will be a single processor that knows all of the rumors, i.e. one that has completed its task.

The adversary cannot move a processor that knows everything, so all moves made after a processor has completed are necessarily made by processors that have not completed. So applying Lemma 1 shows that after $kn \ln n$ additional moves every processor will know everything with high probability. In summary we have the following:

Theorem 4 *Let k be some constant, and let the adversary and processors behave as described earlier in this section. Let $c = 3(\log_2 e + 1) = 7.32\cdots$. Then the probability that every processor knows every rumor after $ckn \ln^2 n$ moves is at most $\frac{1}{n^{k-5}}$.*

3 The collect problem

In the rumor-spreading problem we assumed that all of the knowledge of any particular processor was available to any other processor that wished to read it. In collect problem this is not the case; the adversary can stop a processor in between reading new information from another processor's register and writing that information to its own register. Furthermore, we allow the adversary to stop a processor between making a random choice of which register to *read* and the actual read operation. (This rule corresponds to an assumption that not all reads are equal; some might involve cache misses, network delays, and so forth.) However, as mentioned in Section 1.2, we will permit a processor to make a random choice and *write* the result of this choice to its own register as an atomic operation. (This rule corresponds to an assumption that the timing of a write is not affected by the value being written.)

Overall, the approach will be similar to that taken for the rumor-spreading problem. But it is no longer enough for each processor to simply keep reading randomly selected registers. An adversary strategy that defeats this simple algorithm is to choose one of the registers to be a "poison pill"; any processor that attempts to read this register will be halted. Since on average only one read out of every n would attempt to read the poisonous register, close to n^2 reads would be made before the adversary would be forced to let some processor actually swallow the poison pill.

We will avoid this problem by having each processor use the following algorithm. The essential idea is that before attempting to read a register, a processor will leave a note saying where it is going;[2] poison pills can thus be detected easily by the trail of corpses leading in their direction. The distance that a processor will pursue this trail will be $\lambda \ln n$, where λ is constant chosen to guarantee that the processor reaches its target with high probability.

In the following algorithm, we assume that each processor stores in its output register both the set of values S it has collected so far and its *successor*, the processor it selected to read from most recently.

- While some values are unknown:
  - Set p to be a random processor, and write out p as our successor. (We will call this the *selection step*).
  - Repeat $\lambda \ln n$ times:
    - Read the register of p. Set S to be the union of S and the values field. Set p to the successor field.
    - Write out the new S.

We would like to prove an analogue of Lemma 2 for this more sophisticated algorithm. Let us fix a starting time s. For each processor P and time $t \geq s$, define U_t^P recursively as follows. Let $U_s^P = \{P\}$. If at time t, a processor Q chooses a processor in U_t^P, then $U_{t+1}^P = U_t^P \cup \{Q\}$; otherwise $U_{t+1}^P = U_t^P$. Note that the sets U_t^P are built up by exactly the same random process as the sets V_t^P defined in Lemma 1, and so we can use Lemma 1 to show a high-probability bound on how many times the selection step can be executed by a processor not already in U^P. This bound translates into a bound on the number of operations because the number of operations executed by any processor is at most $2\lambda \ln n + 1$ times the number of times it executes the body of the outer loop, i.e., the number of times the selection step is executed.

However, it is not enough to show that many processors will be in U^P; we must also show that these processors will eventually follow the trail of successor fields to obtain K_s^P. To show this fact we view U_t^P as a rooted tree, whose root is the original node P. As each new node a is added to U^P it must select one of the processors b already in U^P; in this case we draw an edge between a and b. Notice that (conditioning on the fact that a selects a processor already in U^P) the processor b is chosen uniformly from the nodes already in U^P. In Section 3.1 we investigate the random variable M_x, which is defined to be the depth of a tree containing $x + 1$ nodes generated in precisely this fashion. We prove (equation (10)):

Lemma 5 *Let $\lambda \geq 2$, then*

$$Pr[M_{x-1} \geq \lambda \ln x] \leq \frac{1}{x^{\lambda \ln \lambda - \lambda - 1}}$$

Intuitively, the depth of the tree is likely to be bounded by the logarithm of its size because on average the i-th node to be added to the tree will choose as a parent the $(i/2)$-th node. The importance of bounding the depth of the tree is that it gives an immediate bound on the length of a trail that any processor in U^P must follow to learn K_s^P:

Lemma 6 *Suppose that the depth of the U^P tree does not exceed $\lambda \ln n$. Let Q be a processor that has completed the inner loop following its first selection of a processor in U^P. Then Q knows K_s^P.*

[2] It is here that we use the assumption that we can flip a coin and write the outcome atomically.

Proof: The result follows by induction on the size of U^P. If Q is a processor newly added to U^P, either Q successfully follows a chain of successor edges until it reaches P, or at some point it follows an edge leaving some processor R that is not an edge in U^P. But then R must have chosen a new successor after its entry into U^P and thus must have completed its inner loop following its entry into U^P. It follows by the induction hypothesis that R knows K_s^P, and thus Q learns it when it reads R's register. ■

Now we have the following extension of lemma 2.

Lemma 7 *Let the powers of the adversary, and the algorithms of the processors be as defined earlier in this section. Fix a starting time s, let $t = s + 3(kn\ln n + n)(2\lambda \ln n + 1)$, and define V_t^P as the set of processors that know K_s^P at time t and $w(A)$ to be the total number of operations executed by processors in A between s and t. Then*

$$\begin{aligned} & Pr\left[\exists P\ w(V_t^P) \le 2(kn\ln n + n)(2\lambda \ln n + 1)\right] \\ \le\ & \frac{1}{n^{k-4}} + \frac{1}{n^{\lambda \ln \lambda - \lambda - 2}} \end{aligned}$$

Proof: We use an argument similar to that used for Lemma 2. Suppose that $w(\overline{V_t^P}) \ge (kn\ln n + n)(2\lambda \ln n + 1)$. Then by Lemma 1 after $(kn\ln n)(2\lambda \ln n + 1)$ operations every processor in $\overline{V_t^P}$ is in U^P. So by Lemmas 5 and 6 after the remaining $n(2\lambda \ln n + 1)$ operations all n of them will have followed their trails back and read the information. The probability of these events *not* occuring for some P is the value given in the statement of the lemma. ■

This lemma can be used in exactly the same way as in section 2 to prove the following theorem:

Theorem 8 *Let k, λ be constants, $k \ge 1$, $\lambda \ge 2$, and let the adversary and processors behave as described earlier in this section. Assume that $n \ge 3$ and let $c = 37$. Then the probability that the cooperative collect is incomplete after $c\lambda k n \ln^3 n$ moves is at most $\frac{1}{n^{k-5}} + \frac{1}{n^{\lambda \ln \lambda - \lambda - 3}}$.*

Proof: The argument is essentially the same as used for Theorem 4. The resulting cost is given by

$$3(kn\ln + n)(2\lambda \ln n + 1)(\log_2 n + 1)$$

which is at most $37k\lambda n \ln^3 n$ under the assumptions (needed for the lemmas) that $k \ge 1$ and $\lambda \ge 2$, and the further assumption that $n \ge 3 > e$ (implying $\ln^3 n > \ln^2 n > \ln n$). ■

In particular if we take $k = \lambda \ge 9$ we can combine the terms in the probability bound to get as a special case that the probability that the cooperative collect is incomplete after $ck^2 n \ln^3 n$ moves is at most $\frac{2}{n^{k-5}}$ (where $c = 37$ as in the theorem).

3.1 Proof of Lemma 5

In this section we investigate the expected depth of a rooted tree which is built be adjoining each new vertex to one of the existing vertices chosen at random. We will show that with high probability the depth of the tree of i vertices is at most $O(\log i)$.

Let T_i be a random variable whose value is a rooted tree with $i+1$ vertices, including the root vertex. So T_0 consists of the root vertex only. Let T_{i+1} be defined by uniformly selecting one of the $i+1$ vertices in T_i and attaching a new vertex to the selected vertex.

Define random variables D_i to be the depth of the ith vertex, where the root has depth -1, a vertex adjacent to the root has depth 0 and so on. Let M_i be the depth of the tree T_i, so

$$M_i = \max_{j \le i} D_j.$$

Now define indicator variables for $i \ge 0$, $d \ge -1$,

$$X_{i\,d} = \begin{cases} 1 & \text{if } D_i = d \\ 0 & \text{otherwise} \end{cases}$$

Let $x_{i\,d} = \Pr[D_i = d] = \Pr[X_{i\,d} = 1] = \mathrm{E}[X_{i\,d}]$.

From the construction of the tree we have

$$\Pr[X_{i\,d} = 1] = \frac{1}{i}\sum_{j=0}^{i-1} X_{j\,d-1}.$$

Taking expectations we get

$$\mathrm{E}[X_{i\,d}] = \frac{1}{i}\sum_{j=0}^{i-1} \mathrm{E}[X_{j\,d-1}].$$

So the $x_{i\,d}$ are defined by the recurrence equation

$$x_{i\,d} = \begin{cases} \frac{1}{i}\sum_{j=0}^{i-1} x_{j\,d-1} & \text{if } i \ge 1 \text{ and } d \ge 0 \\ 1 & \text{if } i = 0 \text{ and } d = -1 \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

From (2) we can derive two further recurrence equations, for $i \ge 1$, $d \ge 0$

$$x_{i\,d} = \frac{i-1}{i} x_{i-1\,d} + \frac{1}{i} x_{i-1\,d-1} \tag{3}$$

$$\text{and} \quad x_{i\,d} = \frac{1}{i} \sum_{0 < i_1 < i_2 < \ldots < i_d < i} \prod_{j=1}^{d} \frac{1}{i_j}. \tag{4}$$

Now we can use (3) to find the expectation of D_i, since

$$\begin{aligned} \mathrm{E}[D_i] &= \sum_{d=0}^{\infty} d x_{i\,d} = \sum_{0}^{\infty} \left(\frac{i-1}{i} x_{i-1\,d} + \frac{1}{i} x_{i-1\,d-1} \right) \\ &= \frac{i-1}{i} \sum_{0}^{\infty} d x_{i-1\,d} + \frac{1}{i} \sum_{1}^{\infty} (d-1) x_{i-1\,d-1} \\ &\quad + \frac{1}{i} \sum_{1}^{\infty} x_{i-1\,d-1} \\ &= \mathrm{E}[D_{i-1}] + \frac{1}{i} \end{aligned}$$

Since $\mathrm{E}[D_0] = -1$ we get

$$\mathrm{E}[D_i] = \sum_{j=2}^{i} \frac{1}{j} \le \ln i \tag{5}$$

This shows that in a tree with r vertices the expected depth of any particular vertex is at most $\ln r$, which suggests that the expected depth of the entire tree is also of the order

of $\ln r$. To prove this we will need to get an upper bound on $x_{i\,d}$.

By comparing the identity

$$\left(\sum_{j=1}^{i-1}\frac{1}{j}\right)^d = \sum_{i_1=1}^{i-1}\sum_{i_2=1}^{i-1}\cdots\sum_{i_d=1}^{i-1}\prod_{j=1}^{d}\frac{1}{i_j}$$

with (4) we see that

$$\left(\sum_{j=1}^{i-1}\frac{1}{j}\right)^d = ix_{i\,d}d! + \text{terms involving squares.} \qquad (6)$$

Hence

$$x_{i\,d} \le \frac{\left(\sum_{j=1}^{i-1}\frac{1}{j}\right)^d}{i.d!} \le \frac{(1+\ln(i-1))^d}{i.d!} \qquad (7)$$

In fact we can show that as $i \to \infty$, $x_{i\,d} \to \ln^d i/(id!)$. That is, the D_i are asymptotically Poisson distributed with parameter $\ln i$.

Let $d = k\ln i$. Then using Stirling's formula we have

$$\begin{aligned}\frac{(1+\ln i)^d}{d!} &= \left(\frac{d}{k}\right)^d\frac{\left(1+\frac{k}{d}\right)^d}{d!} \le 2\left(\frac{d}{k}\right)^d\frac{e^k}{d!}\\ &\le \frac{2e^k}{\sqrt{2\pi d}}\left(\frac{d}{k}\right)^d\left(\frac{e}{d}\right)^d \le e^k e^{k(1-\ln k)\ln i}\\ &\le \frac{1}{i^{k\ln k-k-1}} \qquad (8)\end{aligned}$$

assuming that $i \ge 3$. By combining (7) and (8) we obtain

$$x_{i\,d} \le \frac{1}{i^{k\ln k-k}} \quad \text{provided } i \ge 3 \text{ and } d \ge k\ln i \qquad (9)$$

Suppose $M_x \ge y$ for some x, y. Since if there is a node with depth bigger than y there must be a node of depth exactly y we have using (2) that

$$\Pr[M_x \ge l] \le \sum_{i\le x}\Pr[D_i = y] = \sum_{i\le x} x_{i\,y} = (x+1)x_{x+1\,y+1}$$

So by applying (9) we can conclude that if k is some constant, $k \ge 2$ then

$$\Pr[M_{x-1} \ge k\ln x] \le \frac{1}{x^{k\ln k-k-1}} \qquad (10)$$

In particular if $k \ge 9$ we have that $k\ln k - k - 1 \ge k$ so

$$\Pr[M_{x-1} \ge k\ln x] \le \frac{1}{x^k} \quad \text{for } k \ge 9. \qquad (11)$$

4 Repeated collects

In this section we consider an extension of the algorithm from Section 3, which implements the *cooperative collect* primitive. For this primitive, a processor must not only be able to collect a set of values that are initially present in the registers; it must also be able to repeatedly carry out collect operations that gather n new values that are guaranteed to be *fresh* in the sense that they were present in the registers at some time during the collect operation.

Our algorithm ensures freshness by a simple timestamp scheme. Upon starting a collect a processor writes out a new timestamp. Timestamps spread through the processor's registers in parallel to register values. When a processor reads a value *directly* from its original register, it tags that value by the most recent timestamp it has from each of the other processors. Thus if a processor sees a value tagged with its own most recent timestamp, it can be sure that that value was present in the registers after the processor started its most recent collect, i.e. that the value is fresh.

The algorithm can be summarized as follows. Below, S tracks the set of values (together with their tags) known to the processor. The array T lists each processor's most recent timestamps. Both S, T, and the current successor are periodically written to the processor's output register.

- Choose a new timestamp τ and set our entry in T to τ.
- While some values are unknown:
  - Set p to be a random processor, write out p as our successor and T as our list of known timestamps.
  - Repeat $\lambda\ln n$ times:
    - Read the register of p. Set S to be the union of S and the values field. Update T to include the most recent timestamps for each processor. Set p to the successor field.
    - Write out the new S and T.
- Return S.

We can characterize the performance of this algorithm by describing its *collective latency* [3], an upper bound on the amount of work needed to complete all collects in progress at some time t:

Theorem 9 *Fix a starting time s. Let k, λ, n, and c be as in Theorem 8. Each process carries out a certain number of steps between s and the time at which it completes the collect it was working on at time s. Let T be the sum over all processors of these numbers. Then*

$$Pr\left[T > 2c\lambda kn\ln^3 n\right] \le 2\left(\frac{1}{n^{k-5}} + \frac{1}{n^{\lambda\ln\lambda-\lambda-3}}\right)$$

Proof: Divide the steps contributing to T into two classes: (i) steps taken by processors that do not yet know timestamps corresponding to all of the collects in progress at time s; and (ii) steps taken by processes that know all n of these timestamps. To bound the number of steps in class (i), observe that the behavior of the algorithm in spreading the timestamps during these steps is equivalent to the behavior of the algorithm in Section 3. Similarly, steps in class (ii) correspond to an execution of the algorithm in Section 3 when we consider the spread of values tagged by all n current timestamps. Thus the total time for both classes of steps is bounded by twice the bound from Theorem 8, except for a case whose probability is at most twice the probability from Theorem 8. ■

Having a bound on the collective latency of our repeated-collect algorithm is important because it allows us to show that the algorithm is competitive against other distributed algorithms. The competitive ratio that we obtain depends on the particular competitive model chosen; there are two natural possibilities for the collect problem, described in the following two sections.

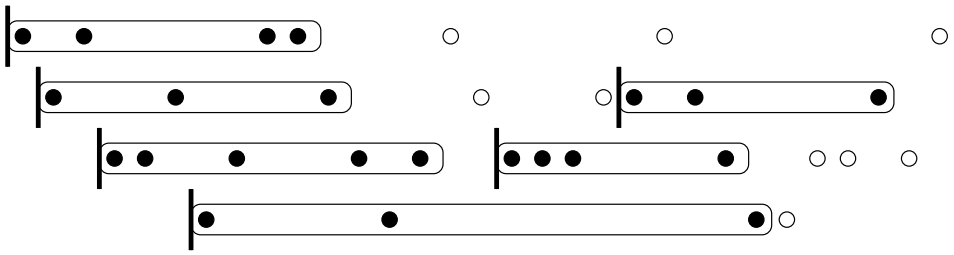

Figure 1: Latency model. New high-level operations (ovals) start at times specified by the scheduler (vertical bars). Scheduler also specifies timing of low-level operations (small circles). Cost to algorithm is number of low-level operations actually performed (filled circles).

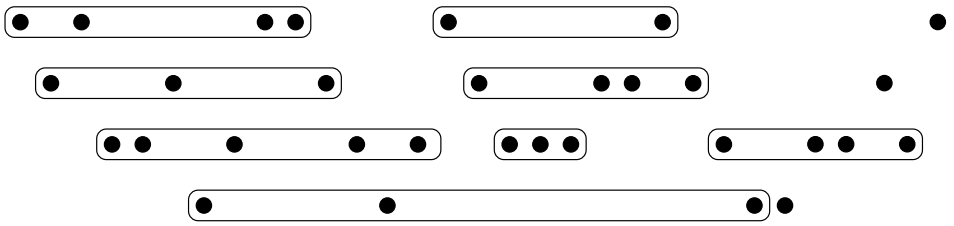

Figure 2: Throughput model. New high-level operations (ovals) start as soon as previous operations end. Scheduler controls only timing of low-level operations (filled circles). Payoff to algorithm is number of high-level operations completed.

4.1 Latency competitiveness

The competitive latency model of Ajtai et al. [3] is a mechanism for applying the technique of competitive analysis, originally developed to deal with the unknown sequences of user inputs in on-line algorithms [33], to unknown patterns of system behavior as found in fault-tolerant distributed algorithms. In the context of the repeated collect problem, it is assumed that the adversary controls the execution of an algorithm by generating (possibly in response to the algorithm's behavior) a schedule that specifies when collects start and when each processor is allowed to take a step (see Figure 1. A processor halts when it finishes a collect; it is not charged for opportunities to take a step in between finishing one collect and starting another (intuitively, we imagine that it is off doing something else). The *competitive latency* of a candidate algorithm is the least constant k, if any, that guarantees that the expected total number of operations carried out by the candidate on a given schedule σ is at most k times the cost of an optimal distributed algorithm (called the *champion* by [3]) running on the same schedule.

Ajtai et al. show that if an algorithm has a maximum collective latency of L at all times, then its competitive ratio in the latency model is at most $L/n + 1$. Unfortunately, this result is stated only for deterministic algorithms, and in any case the upper bound on the collective latency of our algorithm is only a high-probability guarantee and not absolute.

However, the proof in [3] of the relationship between collective latency and competitive latency does not really depend on these details. It proceeds by dividing an execution into segments and showing that for each such segment, the candidate algorithm carries out at most $L + n$ operations and the champion carries out at least n operations. As we show in the full paper, this construction works equally well for randomized algorithms, but the upper bound $L + n$ on the work done by the candidate for each segment becomes a random variable (whose expectation will be $O(n \log^3 n)$ for our algorithm). It follows that:

Theorem 10 *The competitive latency of the repeated collect algorithm is* $O(\log^3 n)$.

This result holds even against an *adaptive off-line* adversary [14], which is allowed to choose the champion algorithm after seeing a complete execution of the candidate.

4.2 Throughput competitiveness

More recently, Aspnes and Waarts [10] have proposed a different measure for the competitive performance of a distributed algorithm. This measure, which they call the *competitive latency*, removes the adversary control over the starting times of collects; instead, both the candidate and the champion try to complete as many collects as possible in the time available (see Figure 2). It also distinguishes between the *schedule* (the timing of events in the system), which is shared between a candidate algorithm and the champion it is competing against, and the *input* (the specification of what tasks to perform), which is assumed to be worst-case for the candidate and best-case for the champion. (In analyzing just the cooperative collect primitive, the input is irrelevant since the cooperative collect algorithm can only perform one type of task). The *throughput competitiveness* is a bound on the ratio of the number of high-level tasks (e.g., collects) completed by the champion to the number of high-level tasks completed by the candidate.

The motivation for these changes from the earlier latency model is that they permit competitive algorithms to be constructed modularly; they allow the competitive ratio of a subroutine and a function that calls it to be computed separately, with the competitive ratio of the combined algorithm simply being the product of the ratios of its components.

Unfortunately, the throughput model does not permit as good a competitive ratio for cooperative collect as the latency model: Aspnes and Waarts give a lower bound of $\Omega(\sqrt{n})$. However, it is an indication of the merits of our algorithm that (with a slight modification) it comes very close to this bound. Again, the key property is its low collective latency. By having each processor alternate between running one step of our algorithm and one step of the naive algorithm that simply reads all registers directly, we get an algorithm whose collective latency is still $O(n \log^3 n)$ and which guarantees to finish any single processor's collect in at most $2n$ work done by that processor. In [10] it is shown that any algorithm with a collective latency of L and an absolute bound of $2n$ operations on any single collect will have a competitive ratio of at most $4\sqrt{L + 2n}$; as with the competitive latency bound, this bound is stated only for deterministic algorithms, but with a bit of tinkering (as shown in the full paper) its proof can be made to apply to our algorithm as well. The result is:

Theorem 11 *The competitive throughput of the repeated collect algorithm is* $O(n^{1/2} \log^{3/2} n)$.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Proc. 7th ACM Symposium on Principles of Distributed Computing*, pp. 291–302, August 1988.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 1–13, 1990.

[3] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 401–411, November 1994. Full version available.

[4] J. Anderson. Composite registers. *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 15–30, August 1990.

[5] R. Anderson and H. Woll. Wait-free parallel algorithms for the Union-Find Problem. In *Proc. 23rd ACM Symposium on Theory of Computing*, pp. 370–380, 1991.

[6] J. Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms* 14(3):414–431, May 1993. An earlier version appeared in *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 325–331, August 1990.

[7] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. In *Journal of Algorithms 11*(3), pp.441–461, September 1990.

[8] J. Aspnes and M. P. Herlihy. Wait-free data structures in the Asynchronous PRAM Model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, July 1990, pp. 340–349, Crete, Greece.

[9] J. Aspnes and O. Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per processor. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 137–146, October 1992.

[10] J. Aspnes and O. Waarts. Modular Competitiveness for Distributed Algorithms. To appear, STOC 96.

[11] Y. Aumann and M.A. Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. To appear, ICALP '96.

[12] H. Attiya, M. Herlihy, and O. Rachman. Efficient atomic snapshots using lattice agreement. Technical report, Technion, Haifa, Israel, 1992. A preliminary version appeared in proceedings of *the 6th International Workshop on Distributed Algorithms,* Haifa, Israel, November 1992, (A. Segall and S. Zaks, eds.), Lecture Notes in Computer Science #647, Springer-Verlag, pp. 35–53.

[13] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 29–40, Aug. 1993.

[14] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Proceedings of the Twenty-Second Annual ACM Symposium on the Theory of Computing*, pages 379–386. ACM, 1990.

[15] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 41–51, August 1993.

[16] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. *Proceedings of the Fifth International Workshop on Distributed Algorithms.* Springer-Verlag, 1991.

[17] T. Chandra and C. Dwork. Using consensus to solve atomic snapshots. *Submitted for Publication*

[18] T. Chandra. Polylog randomized wait-free consensus. To appear, PODC 96.

[19] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pp. 86–97, 1987.

[20] D. Dolev and N. Shavit. Bounded concurrent timestamp systems are constructible! In *Proc. 21st ACM Symposium on Theory of Computing*, pp. 454–465, 1989. An extended version appears in IBM Research Report RJ 6785, March 1990.

[21] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time-lapse snapshots. *Proceedings of Israel Symposium on the Theory of Computing and Systems*, 1992.

[22] C. Dwork, M. Herlihy, and O. Waarts. Bounded round numbers. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 53–64, 1993.

[23] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible!, In *Proc. 24th ACM Symposium on Theory of Computing*, pp. 655–666, 1992.

[24] S. Even and B. Monien. On the number of rounds needed to disseminate information. *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[25] R. Gawlick, N. Lynch, and N. Shavit. Concurrent timestamping made simple. *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1992.

[26] S. Haldar. Efficient bounded timestamping using traceable use abstraction - Is writer's guessing better than reader's telling? Technical Report RUU-CS-93-28, Department of Computer Science, Utrecht, September 1993.

[27] M.P. Herlihy. Randomized wait-free concurrent objects. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, August 1991.

[28] A. Israeli and M. Li. Bounded time stamps. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, 1987.

[29] A. Israeli and M. Pinhasov. A concurrent time-stamp scheme which is linear in time and space. Manuscript, 1991.

[30] L. M. Kirousis, P. Spirakis and P. Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, 1991.

[31] Y. Riany, N. Shavit and D. Touitou. Practical Snapshots. In *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1994.

[32] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus — making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pp. 351–362, 1991.

[33] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM 28*(2), pp. 202–208, 1985.

[34] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th IEEE Symposium on Foundations of Computer Science*, 1986.