A Theory of Competitive Analysis for Distributed Algorithms

Miklos Ajtai* James Aspnes† Cynthia Dwork* Orli Waarts‡

Abstract

We introduce a theory of competitive analysis for distributed algorithms. The first steps in this direction were made in the seminal papers of Bartal, Fiat, and Rabani [17], and of Awerbuch, Kutten, and Peleg [15], in the context of data management and job scheduling. In these papers, as well as in other subsequent work [14, 4, 18], the cost of a distributed algorithm is compared to the cost of an optimal global-control *algorithm. Here we introduce a more refined notion of competitiveness for distributed algorithms, one that reflects the performance of distributed algorithms more accurately. In particular, our theory allows one to compare the cost of a distributed on-line algorithm to the cost of an optimal* distributed *algorithm. We demonstrate our method by studying the* cooperative collect *primitive, first abstracted by Saks, Shavit, and Woll [50]. We provide the first algorithms that allow processes to cooperate to finish their work in fewer steps. Specifically, we present two algorithms (with different strengths), and provide a competitive analysis for each one.*

1 Introduction

Introducing a Notion of Competitive Analysis for Distributed Algorithms The technique of competitive analysis was proposed by Sleator and Tarjan [51] to study problems that arise in an *on-line* setting, where an algorithm is given an unpredictable sequence of requests to perform operations, and must make decisions about how to satisfy its current request that may affect how efficiently it can satisfy future requests. Since the worst-case performance of an algorithm might depend only on very unusual or artificial sequences of requests, or might even be unbounded if one allows arbitrary request sequences, one would like to look instead at how well the algorithm performs relative to some measure of difficulty for the request sequence. The key innovation of Sleator and Tarjan was to use as a measure of difficulty the performance of an optimal *off-line* algorithm, one allowed to see the entire request sequence before making any decisions about how to satisfy it. They defined the *competitive ratio*, which is the supremum, over all possible input sequences σ, of the ratio of the performance achieved by the on-line algorithm on σ, to the performance achieved by the optimal off-line algorithm on σ, where the measure of performance depends on the particular problem.

In a distributed setting there are additional sources of nondeterminism, other than the request sequence. These include process step times, request arrival times, message delivery times (in a message-passing system) and failures. Moreover, a distributed algorithm has to deal not only with the problems of lack of knowledge of future requests and future system behavior, but also with incomplete information about the *current* system state. Due to the additional type of nondeterminism in the distributed setting, it is not obvious how to extend the notion of competitive analysis to this environment.

Bartal, Fiat, and Rabani [17], and Awerbuch, Kutten, and Peleg [15], took the first steps in this direction. Their work was in the context of job scheduling and data management. In these papers, and in subsequent work [4, 14, 18], the cost of a distributed on-line

*IBM Almaden Research Center, 650 Harry Road, San Jose CA 95120. E-mail: ajtai@almaden.ibm.com, dwork@almaden.ibm.com, resp.

†Yale University, Department of Computer Science, 51 Prospect Street/P.O. Box 208285, New Haven CT 06520-8285. E-mail: aspnes-james@cs.yale.edu

‡Computer Science Division, U. C. Berkeley. Work supported by an NSF postdoctoral fellowship. During part of this research the fourth author was at IBM Almaden. E-Mail: orli@cs.stanford.edu

algorithm is compared to the cost of an optimal *global-control* algorithm[1]. (This is also done implicitly in the earlier work of Awerbuch and Peleg [16].) As has been observed elsewhere (see, *e.g.* [14], paraphrased here), this imposes an additional handicap on the distributed on-line algorithm in comparison to the optimal algorithm: In the distributed algorithm the decisions are made based solely on local information. It is thus up to the algorithm to learn (at a price) the relevant part of the global state necessary to make a decision. The additional handicap imposed on the on-line distributed algorithm is that it is evaluated against the off-line algorithm that does *not* pay for overhead of control needed to make an intelligent decision. It is to the credit of the works cited above the algorithms enjoy small competitive ratios even against so powerful a competitor.

We claim that in some cases a more refined measure is necessary, and that to achieve this the handicap of incomplete system information should be imposed not only on the distributed on-line algorithm but also on the optimal algorithm with which the on-line algorithm is compared. Otherwise, two distributed on-line algorithms may seem to have the same competitive ratio, while in fact one of them totally outperforms the other. Our approach is ultimately based on the observation that the purpose of competitive analysis for on-line algorithms is to allow comparison between *on-line* algorithms; the fictitious off-line algorithm is merely a means to this end. Therefore, the natural extension of competitiveness to distributed algorithms is to define a distributed algorithm as k-competitive if for each sequence of requests, and each scheduling of events, it performs at most k times worse than another *distributed* algorithm.

This is the approach introduced in this paper. An algorithm that is k-competitive according to the competitive notion of all current distributed competitive literature [14, 15, 4, 17, 18], is at most k competitive according to our notion, but may be much better. (A concrete example appears below.) Thus, the competitive notion in this paper captures the performance of distributed algorithms more accurately than does the definition used in the literature.

Under both the definition of Sleator and Tarjan and the one introduced by [15, 17], one only has to show that the competitive algorithm performs well in comparison with any other algorithm that deals with one type of nondeterminism: the nondeterminism of not knowing the future requests and system behavior. In contrast, using the new definition one must show that the competitive algorithm performs well in comparison with any other algorithm that deals with *two* types of nondeterminism, *i.e.* the nondeterminism of not knowing the future requests and system behavior, and the nondeterminism of having only partial information about the current system state. Our measure is defined formally in Section 4 and is one of the central contributions of the paper.

Cooperative Collect To demonstrate our technique we study the problem of having processes repeatedly collect values, by the *cooperative collect* primitive, first abstracted by Saks, Shavit, and Woll [50]. In many shared-memory applications processes repeatedly read all values stored in a set of registers. If each process reads every register itself, then the communication costs increase dramatically with the degree of concurrency, due to bus congestion and contention. Interestingly, this is the (trivial) solution that is used in current literature on wait-free shared-memory applications, including nearly all algorithms known to us for consensus, snapshots, coin flipping, timestamps, and multi-writer registers [1, 2, 5, 7, 8, 9, 10, 11, 13, 19, 22, 23, 24, 26, 28, 30, 35, 32, 36, 37, 42, 52][2]. Indeed, the cost of this naïve implementation is easily shown to be a lower bound on the worst-case cost of any implementation. Here, the *worst case* is taken over the set of adversarially chosen *schedules* of events (we give more details below). In this paper we show that in the interesting cases – those in which concurrency is high – it is possible to do much better than in the naïve solution. This suggests that a competitive analysis of the problem may be fruitful.

We assume the standard model for asynchronous shared-memory computation, in which n processes communicate by reading and writing to a set of single-writer-multi-reader *registers*. (We confine ourselves to single-writer registers because the construction of registers that can be written to by more than one process is one of the principal uses for the cooperative collect primitive.) As usual, a *step* is a read or a write to a shared variable. We require our algorithms to be *wait-free*: there is an *a priori* bound on the number of steps a process must take in order to satisfy a request, independent of the behavior of the other processes.

In the *cooperative collect* primitive, processes perform the *collect* operation – an operation in which the process learns the values of a set of n registers, with the guarantee that each value learned is *fresh*:

[1] Because most distributed algorithms have an on-line flavor, we use the terms *distributed algorithm* and *distributed on-line algorithm* interchangeably.

[2] An exception is the consensus algorithm of Saks, Shavit, and Woll [50]. We discuss their results in Section 2.

it was present in the register at some point during the collect.[3] If each process reads every register, then this condition is trivially satisfied. However, more sophisticated protocols may allow one process say, p, to learn values indirectly from another process, q. The difficulty is that these values may be *stale*, in that q obtained them before p started its current collect, and the contents of the registers have changed in the interim. Thus, additional work must be done to ascertain that the values are fresh, and if not, to obtain fresh values.

Competitive Analysis of Cooperative Collect Algorithms We assume that the *schedule* – which processes take steps at which times, when requests for collects arrive, and when the registers are updated – is under the control of an adversary. Intuitively, if the adversary schedules all n processes to perform collect operations concurrently, the work can be partitioned so that each process performs significantly fewer than n reads. However, suppose instead that the adversary first schedules p_1 to perform a collect in isolation. If p_2 is later scheduled to perform a collect, it cannot use the values obtained by p_1, since they might not be fresh. For this reason p_2 must read all the registers itself. Continuing this way, we can construct a schedule in which every algorithm must have each process read all n registers. Thus, the worst-case cost for any distributed algorithm is always as high as the cost of the naïve algorithm.

Moreover, since a global control algorithm knows when registers are updated, if in the above example no register was updated since the time that p_1 completed its collect, then in the global control algorithm p_2 will need to perform at most one read (of the information collected by p_1) to make its collect. Thus no distributed algorithm can be competitive against such an algorithm. Hence also the competitive measure of [15, 17] does not allow us to distinguish between the naïve algorithm and algorithms that totally dominate it.

The competitive measure presented here allows us such a distinction. To characterize the behavior of an algorithm over a range of possible schedules we define the *competitive latency* of an algorithm. Intuitively, the competitive latency measures the ratio between the amount of work that an algorithm needs to perform in order to carry out a particular set of collects, to the work done by the best possible algorithm for carrying out those collects given the same schedule. As discussed above, we refine previous notions by requiring that this best possible algorithm be a distributed algorithm. Though the choice of this *champion* algorithm can depend on the schedule, and thus it can implicitly use its knowledge of the schedule to optimize performance (say, by having a process read a register that contains many needed values), it cannot cut corners that would compromise safety guarantees if the schedule were different (as it would if it allowed a process not to read a register because it "knows" from the schedule that the register has never been written to).

Our Algorithms Using the trivial solution, even if n processes perform collects concurrently, there are a total of n^2 reads. We present the first algorithms that cross this barrier. The basic technique is a mechanism that allows processes to read registers cooperatively, by having each process read registers in an order determined by a fixed permutation of the registers. The proof of competitiveness for our algorithms has two parts, each of which introduces a different technique. In the first part, we partition the execution into intervals, each of which can be identified with a different set of collect operations, and in each of which any distributed algorithm must perform at least n steps. This technique demonstrates how an algorithm can be compared with an optimal *distributed* algorithm, *i.e.,* with an algorithm that does not have global control. In the second part we show how to construct a set of permutations so that a set of concurrent collect operations will take at most kn steps to be completed, for some k, independent of the scheduling of the processes' steps. A first step in this direction was made by Anderson and Woll in their elegant work on the *certified write-all problem* [6] (see Section 2). Due to the requirement of freshness, our adversary is less constrained than the adversary in [6], where freshness is not an issue. Thus, we need additional insight into the combinatorial structure of the schedule. In particular, for this part of the proof we prove that if the adversary has a short description, then there exists a good set of permutations. We then show that the adversary is sufficiently constrained in its choices by our algorithm, that it has a short description.

We present two algorithms; the differences between them come from using different sets of permutations. The first algorithm uses a set of permutations with strong properties that allows a very simple and elegant algorithm; however, the construction of the permutations is probabilistic, although suitable permutations can be found with high probability. The second algorithm uses a constructible but weaker set of permutations, and requires some additional machinery.

[3] This is analogous to the *regularity* property for registers [43]: if a read operation R returns a value that was written in an update operation U_1, there must be no update operation U_2 to the same register such that $U_1 \rightarrow U_2 \rightarrow R$.

In the non-constructive algorithm, the number of reads for n overlapping collects is at most $O(n^{3/2}\log^2 n)$. We show this yields a competitive latency of $O(n^{1/2}\log^2 n)$. In the explicit construction, the number of reads for n overlapping collects is at most $n^{7/4}\log^2 n$. This will yield a competitive latency of $O(n^{3/4}\log^2 n)$. These bounds are in contrast to the $\Omega(n)$-competitiveness of the trivial solution. In addition, we have an absolute worst-case bound on the work done for each collect: both algorithms are structured so that no collect ever takes more than $2n$ operations, no matter what the schedule.

For lack of space, this abstract describes only the non-constructive result, and most proofs are omitted or only sketched.

2 Other Related Work

Saks, Shavit, and Woll were the first to recognize the opportunity for improving the efficiency of shared-memory algorithms by finding a way for processes to cooperate during their collects [50]. They devised an elegant randomized solution, which they analyzed in the so-called *big-step* model. In this model, a time unit is the minimal interval in the execution of the algorithm during which each *non-faulty* process executes at least one step. In particular, if in one time interval one process takes a single step, while another takes 100 steps, only one time unit is charged. Thus, the big-step model gives no information about the number of accesses to shared memory ("small" steps) performed by the processes during an execution of the algorithm. This stands in contrast to our work, which focuses on shared-memory accesses.

The cooperative collect resembles the problem of arranging for processes to collaborate in order to perform a set of tasks. The closest problem in the literature is the *certified write-all* problem (CWA). In this problem, the first variant of which was introduced by Kanellakis and Shvartsman [38], a group of processes must together write to every register in some set, and every process must learn that every register has been written into. This paper was followed by a number of others that consider variants of the basic problem (see, for example, [6, 21, 38, 39, 40, 41, 44, 45]). All of the work on the CWA assumes some sort of multi-writer registers. In a model that provides multi-writer registers, the cooperative collect would be equivalent to the *certified write-all* (CWA) problem, were it not for the issue of freshness. The reason for the equivalence is that if a process learns that some register has been written to, it must be because of information passed to it from some process that wrote to that particular register. Thus, given a certified write-all algorithm, one can replace each of the writes to the registers by a read, and pass the value read along with the certification that that particular register was touched. Thus when each process finishes, because it possesses a certification that each register was touched, it must also possess each register's value.

The CWA is useful in simulating a synchronous PRAM on an asynchronous one. Specifically, the CWA can be used as a synchronization primitive to determine that a set of tasks – those performed at a given step in the simulated algorithm – have been completed, and it is therefore safe to proceed to the simulation of the next step. If each instance of the CWA is carried out on a different set of registers (a solution not relevant to our problem), then issues of freshness do not arise. If registers are re-used the problem becomes more complicated, particularly in a deterministic setting. We know of no work on deterministic algorithms for the CWA problem that addresses these issues in our model of computation. (For example, [6] assumes *Compare&Swap* and a *tagged* architecture, in which associated with each register is a tag indicating the last time that it was written.) In contrast, our algorithms are deterministic.

In the *asynchronous message-passing* model, Bridgeland and Watro studied the problem of performing a number t of tasks in a system of n processors [20]. In their work, processors may fail by crashing and each processor can perform at most one unit of work. They provide tight bounds on the number of crash failures that can be tolerated by any solution to the problem. In the synchronous message-passing model, Dwork, Halpern, and Waarts studied essentially the same problem [27]. Their goal was to design algorithms that minimized the total amount of *effort*, defined as the sum of the work performed and messages sent, in order for each non-faulty process to ensure that all tasks have been performed. Their results were recently extended by Prisco, Mayer and Yung [49].

There is a long history of interest in *optimality* of a distributed algorithm given certain conditions, such as a particular pattern of failures [25, 29, 34, 46], or a particular pattern of message delivery [12, 31, 48]. These and related works are in the spirit of our paper, but differ substantially in the details and applicability to distinct situations.

3 Model of Computation

We assume a system of n processes $p_1, \ldots, p_n$, that communicate through shared memory. Each location in memory is called a *register*. Registers can be read or written in a single atomic step. We assume a completely asynchronous system. Each process can receive stimuli (requests) from the outside world. A process' local state can change only when it takes a step (performs a read or a write of a register) or in response to an external stimulus. Each process has a set of halting states. A process in a halting state takes no steps and cannot change state except in response to an outside stimulus. A process in a non-halting state is, intuitively, ready to take a step. On being activated by the scheduler, such a process accesses a register and enters a new state. The new state is a function of the previous state and any information read, if the step was a read of a register. This formalizes the usual assumption in an asynchronous algorithm, that a process cannot change state solely in response to being given, by the scheduler, the opportunity to take a step. We assume that the *schedule* of events, that is, the interleaving of step times and outside stimuli, is under the control of an adversary.

4 Competitive Analysis

Traditionally, the competitiveness of an algorithm has been measured by comparing its performance to the performance of an omniscient being (the off-line algorithm). The intuition is that if the on-line algorithm "does well" when measured against an omniscient being, then it certainly "does well" when compared to any other algorithm that solves the problem. This notion of competitiveness can be extended naturally by restricting the class of things (omniscient beings, or algorithms) against which the given algorithm is to be compared, provided the resulting comparison says something interesting about the algorithm studied.

As discussed in the Introduction, in order to get a more refined measure of the performance of a distributed algorithm, we compare its performance to that of other *distributed* algorithms: algorithms in which processes get no "free" knowledge about the current state of the system. To measure the competitiveness of an algorithm for a certain problem $\mathcal{P}$, we compare its cost on each schedule σ, to the cost of the best distributed algorithm on σ. We refer to the algorithm being measured as the *candidate*, and we compare it, on each schedule σ to the *champion* for σ. Thus, we can imagine that the champion guesses σ and optimizes accordingly, but even if the schedule is not σ the champion operates correctly. Note that we have restricted our comparison class by requiring that the champion actually be a *distributed* algorithm for $\mathcal{P}$ – that is, that it solve problem $\mathcal{P}$ correctly on *all* schedules. On the other hand, we permit a different champion for each σ. This is a departure from the usual model, in which there is a *single* off-line algorithm.

In this paper we focus on a particular cost measure based on the work done by an algorithm. The result is a competitive ratio which we call *competitive latency*.

4.1 Competitive Latency

In this paper we are interested in algorithms for carrying out a sequence of tasks. Each request from the scheduler is a request to carry out a particular task. To complete a task a process must enter into one of its halting state. (Naturally, to be correct, the algorithm must in fact have successfully carried out the specified task when it enters into this halting state.)

We consider only schedules in which each process in the candidate algorithm completes its current task before being asked to start a new one. (This is consistent with the use of the cooperative collect in all the algorithms mentioned above.) Similarly, for each such schedule, we will only consider as possible champions algorithms in which each process happens to finish its task before the next task arrives. Algorithms that have this property will be said to be *compatible* with the given schedule. We will charge both the candidate and the champion for every read or write operation that they carry out as part of the tasks.

The *total work* done by an algorithm A under an adversary schedule s is just the number of reads and writes in s.[4] Writing this quantity as $\mathrm{work}(A, s)$, the *competitive ratio with respect to latency* of an algorithm is defined to be:

$$\sup_s \frac{\mathrm{work}(A, s)}{\inf_B \mathrm{work}(B, s)}$$

where B ranges over all correct distributed algorithms that are compatible with s. This definition is sufficient for our purposes as we consider only deterministic algorithms; for a randomized algorithm it would be necessary to take expectations over both the algorithm's choices and the adversary's responses to them.

[4] This quantity is not simply the length of the schedule since a process does no work while in its halting state.

5 The Speedy Collect Algorithm

In this section we present a non-constructive algorithm that is $O(\sqrt{n}\log^2 n)$-competitive with respect to latency.

Our starting point is the Certified Write-All algorithm of Anderson and Woll [6]. In their algorithm every process p_i has a fixed permutation π_i of the integers $\{1,\ldots,n\}$. When p_i takes a step it writes to the first location in π_i that has not yet been written. Intuitively, it is to the adversary's advantage if many processes write to the same location at the same time, since this causes wasted work. For each adversary scheduler Anderson and Woll showed that the number of cells that are written can be bounded above as follows.

A *longest greedy monotonic increasing subsequence* (LGMIS) of a permutation π with respect to an ordering σ is constructed by starting with the empty sequence, then running through the elements of π in order and adding each to the subsequence if and only if it is larger (according to σ) than all elements already in the subsequence. Let σ be the order in which the cells are first written, under this adversary schedule. The total number of writes performed by each p_i in this schedule is bounded above by the length of the longest *greedy* monotonic increasing subsequence of π_i with respect to σ. It was shown probabilistically in [6] that there exists a set of n permutations on the numbers $\{1,\ldots,n\}$ such that the sum of the lengths of all longest greedy monotonic increasing subsequences on the set with respect to any ordering σ is $O(n\log n)$. Later, J. Naor and R. Roth [47] obtained an explicit construction in which this quantity is $O(n(\log n)^{1+\varepsilon})$.

This, then, is our starting point. We observe that the adversary scheduler in [6] can be described in $n\log n$ bits. Due to freshness considerations, our problem is harder, and our adversary has more flexibility, and therefore may require significantly more bits to describe. This is important in light of the following combinatorial lemma, which ties the existence of a good set of permutations to the length of the description of the adversary. Specifically, if the adversary can be described in $O(n\log n)$ bits, then there is a "good" set of permutations:

Lemma 5.1 *Assume that n is a positive integer and A is a set so that each element of $a \in A$ is a sequence $a = R_1^a, \ldots, R_n^a$ where R_i^a is an ordering of the set $\{1,\ldots,n\}$. For all $c_1, c_2 > 0$, there exists c_3, so that if n is sufficiently large, $|A| \le 2^{c_1 n\log n}$, and the permutations $\pi_1,\ldots,\pi_n$ are taken at random independently and with uniform distribution on the set of all permutations of $\{1,\ldots,n\}$, then with a probability of at least $1 - e^{-c_2 n\log n}$ we have that for all $a \in A$, $\sum_i l(R_i^a, \pi_i) \le c_3 n\log n$.*

Intuitively, R_i^a captures the behavior of adversary a that is relevant to process p_i, in the sense that, as we show, the reads performed by each process are bounded above by the longest greedy monotonic increasing subsequence of π_i with respect to R_i^a. The relation between the R_i's and the adversary scheduler is as follows. In the algorithm of [6], R_i^a describes the order in which the cells $1\ldots n$ are first written; thus $R_i^a = R_j^a$ for all i, j, a. In our scenario, for each process p_i we are concerned with the ordering of the writes of blocks of registers which can be *trusted* by p_i to be fresh. Thus in our scenario R_i^a describes the order of the first *trustworthy* writes of each block (values that p_i trusts to be fresh). Therefore, for each a it is not necessarily the case that $R_i^a = R_j^a$, and thus a naïve representation of our adversary requires more than $n\log n$ bits. Nonetheless, as we show in Section 5.2, it is actually possible to describe the adversary in $O(n\log n)$ bits. This is because the relationship between R_i^a and R_j^a is not completely unconstrained. For example, intuitively, if p_j begins its collect before p_i does, then values fresh for p_i will be considered fresh by p_j.

We now describe the algorithm, which we call the *Speedy Collect* algorithm.

We partition the processes into groups of size $\sqrt{n}$. The processes in each group will collaborate to read $\sqrt{n}$ blocks of $\sqrt{n}$ registers; there is no collaboration between groups. Each process p has a shared variable COLLECT-NUM$_p$, initially zero and incremented each time p begins a new collect. Throughout the algorithm, p repeatedly computes timestamps. A timestamp is an array of collect numbers, one for each process. Intuitively, p will trust any value tagged with a timestamp whose component for p equals COLLECT-NUM$_p$ because these values are necessarily read after p's collect began.

The views of processes in a group are read and updated using the atomic snapshot algorithm of Attiya and Rachman [13]. The basic operation of the Attiya-Rachman algorithm on an array A is SCAN-UPDATE(v), where v can be null. When a process p performs SCAN-UPDATE(v) for a non-null v, it has the effect of updating p's current value to v and returning a copy of the entire contents of A (a *snapshot*), with $A[p] = v$. When it performs SCAN-UPDATE(v) for a null v, it simply returns the snapshot of A. In the following, all SCAN-UPDATE() operations are applied to the array VIEW. Since the Attiya-Rachman

algorithm is an atomic snapshot algorithm, there is a total *serialization order* on the SCAN-UPDATE( )operations that preserves the real time order of the operations and that corresponds to the apparent ordering determined by which SCAN-UPDATE() operations return which values. The SCAN-UPDATE(v) operation has a cost of $O(m \log m)$, where m is the number of processes (and also the size of the array); in this paper we will generally be using snapshots only within a group of $\sqrt{n}$ processes, in which case the cost will be $O(\sqrt{n} \log n)$.

Each process p is given a fixed permutation π_p of the blocks. On first waking (beginning a collect), p performs SCAN-UPDATE(NEWVIEW$_p$), where NEWVIEW$_p$ contains only p's newly incremented collect number. From then on, p repeatedly performs the following operations.

1. READ-GROUP: Obtain an atomic snapshot of the current view of all processes in the group by invoking SCAN-UPDATE() ($O(\sqrt{n} \log n)$ operations). Extract from this a snapshot of the vector of collect numbers, but do not write this snapshot to shared memory, at this point. Call this snapshot a *timestamp*.

2. READ-BLOCK: Read the registers in the first block in π_p that, in the union of the views obtained in the snapshot, are not tagged with a timestamp whose pth component is COLLECT-NUM$_p$ ($\sqrt{n}$ operations).

3. WRITE-VIEW: Tag the block just read with the current timestamp. Let NEWVIEW$_p$ be the union of this and, for each block b seen in the snapshot, the most recent value of b, tagged with its timestamp; and in addition COLLECT-NUM$_p$ (which is unchanged). Update VIEW[p] by invoking SCAN-UPDATE(NEWVIEW$_p$) ($O(\sqrt{n} \log n)$ operations).

This loop repeats until all $\sqrt{n}$ blocks appear in VIEW[p] tagged with a timestamp whose pth component is COLLECT-NUM$_p$. However, to ensure that in the worst case process p performs at most $O(n)$ operations, every time it performs a single atomic step it also performs a simple read of a register for which it does not yet have a value tagged with a timestamp whose pth component is COLLECT-NUM$_p$. It completes its current collect as soon as it knows a fresh value for every register.

The key to the performance of the algorithm is the choice of good permutations. In order to be able to choose the permutations well we need to formulate a more precise description of the effect of the adversary scheduler. In the next section we show how to do this.

5.1 Representing the Scheduling Adversary as a Combinatorial Object

Given a set Π of m permutations on $\{1, \ldots, m\}$, the adversary, denoted by σ, consists of three parts, as described below. We remark that the definition below is purely combinatorial; the interpretations given below of each of the parts in terms of what values are "trusted" by processes is intended solely to give an intuitive explanation of why this representation was chosen.

1. The first part of the adversary attaches to each process a number between 1 and m, which will be called the process' trusting threshold. At least one process will have trusting threshold m. Intuitively, the trusting thresholds reflect the serialization order of updates to the vector of collect numbers. A process p will trust only values attached with a snapshot that contains p's current collect number. Thus, p only trusts values tagged with timestamps that are serialized after p's most recent update of COLLECT-NUM$_p$. A lower trusting threshold corresponds to an earlier timestamp and represents a process that is more likely to trust other processes' values. Specifying the trusting thresholds takes $m \log m$ bits.

2. The second part of the adversary, denoted by σ', is an ordered list of at most $2m$ elements. Each element in σ' is an ordered pair of numbers, each of which is an integer between 1 and m. The first number appearing in a pair is referred to as the *value* of the pair, and the second is referred to as the *trustworthiness* of the pair. The value of the pair represents the index of a block of registers, while the trustworthiness reflects a timestamp with which the block was tagged.

   The sequence σ' is constructed by mixing two sequences of length m. The first sequence contains one element for each block between 1 and m; this element has as its value the number of the block, and has trustworthiness m. These pairs are ordered according to the order in which universally trusted versions of these blocks are written. The second sequence consists of a pair for each process p recording p's last write of a block that is *not* universally trusted (if there is such a block). The elements of the two sequences are interleaved together according to the serialization order of the

corresponding write operations. Since each of the at most $2m$ elements of σ' can be specified in $2\log m$ bits, the number of bits needed for this part of the adversary is again $O(\log m)$.

3. The third and last part of the adversary provides for each trusting threshold (i.e. each number between 1 and m) a subset of the integers $\{1,\ldots,m\}$ that will correspond to it. This subset is called an *old subset* corresponding to the trusting threshold. Old subsets are required to be totally ordered under inclusion; that is, the old subset for a particular threshold must be contained in the old subset for any lower threshold (the containment need not be proper). Intuitively, values in an old subset of trusting threshold t are trusted to be fresh only by processes of trusting threshold less than or equal to t. Intuitively, the union of the old subsets will contain all values that are trusted only by some of the processes.

   Because the old subsets are ordered by inclusion, they too can be represented in only $O(m\log m)$ bits.

Observe that the adversary is fully defined using $O(m\log m)$ bits.

Now we show how the above adversary imposes an order R_i on π_i. First, erase from π_i all elements that are contained in the old set corresponding to i's trusting threshold. Call the remaining permutation π_i'. We first define the sequences S_i as follows:

- S_i contains exactly the elements that appear in π_i'.
- An element p precedes q in S_i iff the first occurrence of a pair with value p that is trusted by i (according to i's trusting threshold) appears in σ' before the first occurrence of a pair with value q that is trusted by i. Said differently, consider only the pairs in σ' with second component (trustworthiness) at least as large as the trusting threshold assigned to i. Then p precedes q in S_i if in this restricted list of pairs the first pair of the form $(p,\cdot)$ precedes the first pair of the form $(q,\cdot)$.

Note that the sequences S_i are together completely determined by the adversary σ and can therefore be described using only $O(m\log m)$ bits.

We denote by $l(\pi_i',\sigma)$ the greedy monotonic increasing subsequence in π_i' according to S_i, where π_i', S_i are constructed using σ as described above. Define

$$\Gamma(\Pi) = \max_{\sigma}\sum_{i=1}^{n}\|l(\pi_i',\sigma)\|.$$

Padding the S_i with a prefix containing the elements in π_i but not in π_i', we can then use Lemma 5.1 to prove the following theorem:

Theorem 5.2 *There is a constant c such that for each m there is a set Π of m permutations $\pi_1,\ldots,\pi_m$ of m values each such that $\Gamma(\Pi)\le cm\log m$.*

In the next section we show that the effect of an adversary scheduler on the Speedy Collect algorithm can be completely captured by a 3-part adversary σ of the type described above. Thus, choosing the set of permutations Π whose existence is guaranteed by Theorem 5.2 yields an algorithm with good latency.

5.2 Collective Latency of the Speedy Collect Algorithm

Define the *collective latency* of a set of processes G at a point t in time as the sum over all $p\in G$ of the number of operations done by process p between t and the time that it completes the last collect that it started at or before t. (Recall that a process is considered as having completed its collect only when it enters into a halting state.) We show that for a suitable set of permutations our algorithm gives a small collective latency for each of the $\sqrt{n}$-sized groups of processes; in Section 5.3 we use this fact to show that our algorithm is competitive with respect to latency when all of the processes are taken together. The following theorem is at the core of our proof of competitiveness:

Theorem 5.3 *Let Π be a set of $m=\sqrt{n}$ permutations on $\{1,\ldots,m\}$. Suppose that the set of permutations Π for each group satisfies $\Gamma(\Pi)=O(T(m))$. Then the collective latency for each group using our algorithm is $O\left(T(m)\sqrt{n}\log n\right)$.*

Sketch of Proof: Fix an arbitrary time t and let G be the set of processes performing collects at time t. The proof separately analyzes READ-BLOCKs after time t whose values are trusted by every process in G ("globally trusted" READ-BLOCKs), and other ("partially trusted") READ-BLOCKs. It is easily shown that each process performs at most one partially trusted READ-BLOCK after time t.

The first half of the analysis of globally trusted READ-BLOCKs is the construction of a three-part adversary σ consistent with the representation described in Section 5.1. Intuitively, the three parts are as follows. The first part of σ is determined by the serialization order on the processes' initial calls to SCAN-UPDATE(), in which they write their new collect numbers for the collects that are in process at t. The

ordering σ' is determined by the serialization order on the globally and partially trusted blocks performed in the WRITE-VIEW phases. Finally, the serialization order on the timestamps orders the sets trusted by the individual members of G by inclusion.

The remainder of the analysis of globally trusted READ-BLOCKs shows that the globally trusted READ-BLOCKs done by process i correspond to a greedy monotonic increasing subsequence according to the sequence S_i described in Section 5.1. Applying Theorem 5.2 yields a bound of $\Gamma(\Pi) = O(T(m))$ on the number of globally trusted READ-BLOCKs. The cost of each of these READ-BLOCKs and their associated WRITE-VIEW and READ-GROUP phases is $O(\sqrt{n}\log n)$. ■

5.3 Using Collective Latency to Bound the Latency Competitiveness

The following theorem is the key to the relationship between collective latency and the competitive latency measure. To make this connection, it is useful to have the following definition: given a particular schedule, the *work ratio* for a group of processes G is the ratio between the total number of operations performed by processes in G in the candidate algorithm to the total number of operations performed by all processes in the champion algorithm.

Theorem 5.4 *For any cooperative collect algorithm A, any group G of processes, and any schedule that is compatible with A, if there exists a bound L such that for all times t the collective latency for G at t is at most L, then the work ratio for G is at most $L/n + 1$, where n is the number of values to be collected.*

Sketch of Proof: The key to the proof is that whenever some process starts a collect in algorithm A, the same process starts a collect in the champion algorithm. So we can define a partition of the schedule into intervals I_1, I_2, etc., which have the property that: (a) the champion performs at least n operations during each interval; and (b) algorithm A requires at most $L + n$ operations to complete the collects performed by processes in G that start during each interval. Fact (a) is proved by demonstrating that during each interval every register must be read at least once to obtain fresh values. Fact (b) is proved by applying the definition of collective latency to the endpoint of each interval. The result follows. ■

Corollary 5.5 *For any collect algorithm, if the processes can be divided into m groups such that for all times t each group has a maximum collective latency of L at t, then the competitive latency is $mL/n + m$.*

Lemma 5.6 *Suppose that the set of permutations Π for each group satisfies $\Gamma(\Pi) = O(T(\sqrt{n}))$. Then our algorithm has a competitive latency of $O(T(\sqrt{n})\log n)$.*

Proof: By Theorem 5.3, the collective latency for each group is $L = O(T(\sqrt{n})\sqrt{n}\log n)$. There are $m = \sqrt{n}$ groups, so the result follows from Corollary 5.5 ■

The set of permutations Π from Theorem 5.2 have $T(\sqrt{n}) = O(\sqrt{n}\log n)$, and thus:

Theorem 5.7 *The competitive latency of the Speedy Collect algorithm is $O(n^{1/2}\log^2 n)$.*

Acknowledgements

We are grateful to Noga Alon, Richard Anderson, Joe Halpern, Paris Kanellakis, Chip Martel, Nimrod Megiddo, and Yuval Rabani for their help in this work.

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] N. Alon. Generating pseudo-random permutations and maximum-flow algorithms. In *Infor. Proc. Letters* 35, 201–204, 1990.

[4] N. Alon, G. Kalai, M. Ricklin, and L. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 334-343, October 1992.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] H. Attiya, A. Herzberg, and S. Rajsbaum. Optimal Clock Synchronization under Different Delay Assumptions. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 109–120, Aug. 1993.

[13] H. Attiya and O. Rachman. Atomic Snapshots in $0(n \log n)$ Operations. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 29–40, Aug. 1993.

[14] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proc. 25th ACM Symposium on Theory of Computing*, pp. 164–173, May 1993.

[15] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proc. 24th ACM Symposium on Theory of Computing*, pp. 571–580, May 1992.

[16] B. Awerbuch and D. Peleg. Sparse Partitions. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pp. 503–513, November 1990.

[17] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symposium on Theory of Computing*, pp. 39–50, 1992.

[18] Y. Bartal, and A. Rosen. The distributed k-server problem – A competitive distributed translator for k-server algorithms. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 344–353, October 1992.

[19] 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.

[20] M. F. Bridgeland and R. J. Watro. Fault-Tolerant Decision Making in Totally Asynchronous Distributed Systems. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pp. 52–63, 1987.

[21] J. Buss and P. Ragde. Certified Write-All on a Strongly Asynchronous PRAM. *Manuscript,* 1990.

[22] T. Chandra and C. Dwork. Using Consensus to solve Atomic Snapshots. *Submitted for Publication*

[23] 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.

[24] D. Dolev and N. Shavit. Bounded Concurrent Time-Stamp 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.

[25] D. Dolev, R. Reischuk, and H.R. Strong. Early Stopping in Byzantine Agreement. *JACM* 34:7, Oct. 1990, pp. 720–741. First appeared in: Eventual is Earlier than Immediate, IBM RJ 3915, 1983.

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

[27] C. Dwork, J. Halpern, and O. Waarts. Accomplishing Work in the Presence of Failures. In *Proc. 11th ACM Symposium on Principles of Distributed Computing*, pp. 91–102, 1992.

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

[29] C. Dwork and Y. Moses. Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures. In *Information and Computation 88*(2) (1990), originally in *Proc. TARK* 1986.

[30] 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.

[31] M. Fischer and A. Michael, Sacrificing Serializability to Attain High Availablity of Data in an Unreliable Network. Research Report 221, Yale U., Feb. 1982.

[32] R. Gawlick, N. Lynch, and N. Shavit. Concurrent Timestamping Made Simple. *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1992.

[33] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment, *Journal of the Associationfor Computing Machinery*, Vol 37, No 3, January 1990, pp. 549–587. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[34] J.Y. Halpern, Y. Moses, and O. Waarts. A Characterization of Eventual Byzantine Agreement. In *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 333–346, August 1990.

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

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

[37] A. Israeli and M. Pinhasov. A Concurrent Time-Stamp Scheme which is Linear in Time and Space. Manuscript, 1991.

[38] P. Kanellakis and A. Shvartsman. Efficient Parallel Algorithms Can Be Made Robust. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pp. 211–222

[39] P. Kanellakis and A. Shvartsman. Efficient Robust Parallel Computations. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, pp. 23–36, 1991.

[40] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining Tentative and Definitie Algorithms for Very Fast Dependable Parallel Computing. In *Proc. 23rd ACM Symposium on Theory of Computing*, pp. 381–390, 1991.

[41] A. Kedem, K. Palem, and P. Spiriakis. Efficient Robust Parallel Computations. In *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 138–148, 1990.

[42] 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.

[43] L. Lamport. On Interprocess Communication, Parts I and II. *Distributed Computing 1*, pp. 77–101, 1986.

[44] C. Martel and R. Subramonian. On the Complexity of Certified Write-All Algorithms. *Manuscript*, 1993.

[45] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMS are (Almost) as Good as Synchronous PRAMs. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pp. 590–599, 1990.

[46] Y. Moses and M.R. Tuttle. Programming Simultaneous Actions Using Common Knowledge. *Algorithmica 3*(1), pp. 121–169, 1988 (Also appeared in *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pp. 208–221, 1986.)

[47] J. Naor and R. M. Roth. Constructions of permutation arrays for certain scheduling cost measures. *Manuscript.*

[48] B. Patt-Shamir and S. Rajsbaum. A Theory of Clock Synchronization. In *Proc. 26th ACM Symposium on Theory of Computing*, pp. 810–819, May 1994.

[49] R. D. Prisco, A. Mayer, and M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proc. 30th ACM Symposium on Principles of Distributed Computing*, pages 161-172, 1994.

[50] 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.

[51] 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.

[52] 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.