

A Theory of Competitive Analysis for Distributed Algorithms *

Miklos Ajtai[†] James Aspnes[‡] Cynthia Dwork* Orli Waarts[§]

June 10, 2003

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 [18], and of Awerbuch, Kutten, and Peleg [16], in the context of data management and job scheduling. In these papers, as well as in other subsequent work [4, 15, 19, 14], the cost of a distributed algorithm is compared to the cost of an optimal *global-control* algorithm. (This is also done implicitly in the earlier work of Awerbuch and Peleg [17].) 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 [57]. We present two algorithms (with different strengths) for this primitive, and provide a competitive analysis for each one.

*A preliminary version of this paper was presented at the 35th Symposium on Foundations of Computer Science, November 1994.

[†]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. Supported by NSF grant CCR-9410228. Part of this research was performed while at IBM Almaden. E-mail: aspnes-james@cs.yale.edu.

[§]Computer Science Division, U. C. Berkeley, CA 94720. Work supported in part by an NSF postdoctoral fellowship. During part of this research the fourth author was at IBM Almaden. E-Mail: waarts@cs.berkeley.edu

1 Introduction

Introducing a Notion of Competitive Analysis for Distributed Algorithms The technique of competitive analysis was proposed by Sleator and Tarjan [58] 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.

Awerbuch, Kutten, and Peleg [16], and Bartal, Fiat, and Rabani [18], 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, 15, 14, 19], the cost of a distributed on-line algorithm¹ is compared to the cost of an optimal *global-control* algorithm. (This is also done implicitly in the earlier work of Awerbuch and Peleg [17].) As has been observed elsewhere (see, *e.g.* [15], 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.

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

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

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 any other *distributed* algorithm.

This is the approach introduced in this paper. An algorithm that is k -competitive according to the competitive notion of all current literature on competitive distributed algorithms [4, 15, 16, 18, 19, 14], 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 [16, 18], 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: 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 using the *cooperative collect* primitive, first abstracted by Saks, Shavit, and Woll [57]. 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, bounded round numbers, timestamps, and multi-writer registers [1, 2, 5, 7, 8, 9, 10, 11, 13, 20, 21, 24, 25, 26, 28, 30, 32, 38, 34, 35, 39, 40, 46, 59]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.

²An exception is the consensus algorithm of Saks, Shavit, and Woll [57]. We discuss their results in Section 2.

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*: 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 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 they are 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.

The example above shows that a worst-case measure is not very useful for evaluating cooperative collect algorithms. Unfortunately, a similar example shows that a competitive analysis that proceeds by comparing a distributed algorithm to an ideal global-control algorithm gives equally poor results. The underlying difficulty arises because a global-control algorithm knows when registers are updated. Thus in the case where none of the registers have changed since a process’s last collect, it can simply return the values it previously saw, doing *no* read or write operations. On the other hand, any distributed algorithm must read all n registers to be sure that new values have not appeared, which gives an infinite competitive ratio, for *any* distributed algorithm. Thus the competitive measure of [16, 18] 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

³This is analogous to the *regularity* property for registers [47]: 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 $U_1 \rightarrow U_2 \rightarrow R$.

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 collect algorithm, 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 deterministic 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.

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 $n^{(7/4)+\epsilon} \log^2 n$, where ϵ tends to zero as n goes to infinity. This will yield a competitive latency of $O(n^{(3/4)+\epsilon} \log^2 n)$. 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.

Our Upper Bounds versus a Lower Bound For comparison, we show that no algorithm can have expected competitive latency that is better than $\Omega(\log n)$. (This of course implies that for each deterministic algorithm, there is a fixed schedule on which its competitive latency is at least $\Omega(\log n)$.) Our lower bound holds also for randomized algorithms and against an adversary that does not need to know the algorithm, *i.e.*, an adversary weaker than the usual oblivious adversary. In particular, the adversary determines a probability distribution over

schedules under which any cooperative collect algorithm, even one that uses randomization, will have an expected latency competitiveness of $\Omega(\log n)$. In contrast, our algorithms are deterministic and their competitiveness is measured against a stronger, worst-case (*adaptive*) adversary.

The $\Omega(\log n)$ lower bound matches our upper bound rather tightly in the following sense. Our algorithms have a structure in which processes are partitioned into \sqrt{n} 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. Loosely stated we show that for each of these groups, the *work ratio*, between the amount of work performed by the group to the work done by all processes in the best possible algorithm for this schedule, is $O(p \log n)$ for our first algorithm and $O(pn^{(1/4)+\epsilon} \log n)$ for our second algorithm, where pn is the number of atomic steps a process needs to perform in order to complete an atomic snapshot scan operation on n registers. The competitive latency is obtained by multiplying these ratios by the number of groups (cf. end of Section 5.5).

Since the best known atomic snapshot in our model of computation, due to Attiya and Rachman [13], requires a process to perform $O(n \log n)$ atomic steps for each atomic snapshot, it follows that for each of the \sqrt{n} groups, the ratio between the amount of work performed by it in our faster algorithm, to the work done by all processes in the best possible algorithm for this schedule is $O(\log^2 n)$. Thus, despite the powerful adversary, each of these groups comes very close to our almost trivial lower bound of $\Omega(\log n)$. This also shows that in order to improve significantly the upper bound obtained in this paper, one has to allow the groups to collaborate in some way. The combinatorial problem of reasoning about the interactions between the processes, which is already quite hard even when collaboration is done solely within the groups, becomes significantly harder.

Organization The remainder of the paper is organized as follows. Section 2 describes some additional related work. Section 3 describes our model of computation. Section 4 presents our competitive measure. Section 5 describes our faster cooperative collect algorithm and its competitive analysis. Section 6 describes our second cooperative collect algorithm, whose construction is explicit, and analyses its competitiveness. Section 7 derives a lower bound for the competitive latency of any cooperative collect algorithm.

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 [57]. 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 [41], a group of processes must together write to every register in some set (the *write-all*), and every process must learn that every register has been written to (the *certification*). This paper was followed by a number of others that consider variants of the basic problem (see, for example, [6, 23, 41, 42, 44, 45, 48, 49]). 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 the *asynchronous message-passing* model, Bridgeland and Watro studied the problem of performing a number t of tasks in a system of n processors [22]. 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 [29]. 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 [56].

A related notion to the competitive measure suggested in this paper is the idea of comparing algorithms with partial information only against other algorithms with partial information, which was introduced by Papadimitriou and Yannakakis [54] in the context of linear programming; their model corresponds to a distributed system with no communication. A generalization of this approach has recently been described by Koutsoupias and Papadimitriou [43].

In addition, there is a long history of interest in *optimality* of a distributed algorithm given certain conditions, such as a particular pattern of failures [27, 31, 37, 50, 52, 53], or a particular pattern of message delivery [12, 33, 55]. These and related works are in the spirit of our paper, but differ substantially in the details and applicability to distinct situations. In a sense, work on optimality envisions a fundamentally different role for the adversary in which it is trying to produce bad performance both in the candidate algorithm and in what we would call the champion algorithm; in contrast, the adversary used in competitive analysis usually cooperates with the champion.

3 Model of Computation

We assume a system of n processes p_1, \dots, 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. Our algorithms will assume that the registers are single-writer-multi-reader, *i.e.* that each register can be written to only by a single processor (its owner) and read by all processors. We assume a completely asynchronous system. Each process can receive stimuli (requests) from the outside world. A process's 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 still 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 states. (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 $\text{work}(A, s)$, the *competitive ratio with respect to latency* of an algorithm is defined to be:

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

where s ranges over all finite schedules that are compatible with A and B ranges over all correct distributed algorithms that are compatible with s . The restriction to finite schedules is required for the ratio to be defined. For any fixed *infinite* schedule, we can define the competitive ratio to be the limit supremum of the ratio for increasingly long finite prefixes of the schedule. But this leads to the same result once the supremum over all schedules is taken.

Note that the definition above 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.

⁴This 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 $\{1, \dots, n\}$. 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. Anderson and Woll showed that 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 $\{1, \dots, n\}$ 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 [51] obtained an explicit construction in which this quantity is $O(n^{1+\epsilon} \log n)$, where ϵ tends to zero when n tends to infinity.⁵

In the speedy collect algorithm, we also have each process choose its actions (which are now read operations instead of the write operations of Certified Write-All) according to a fixed permutation that is distinct for each process. Similarly, we adopt a rule that a process does not carry out a read of a particular register if some other process has already supplied it with a fresh value for that register, with freshness detected by the use of timestamps. This additional requirement that the secondhand values be fresh, however, means that the set of reads done by a process does not correspond to a greedy monotonic increasing subsequence of the process's permutation, and in consequence the known results bounding the sizes of LGMIS do not help us find a "good" set of permutations that will make our algorithm competitive. Instead, we can show that a "good" set of permutations exists, using a probabilistic argument based on a demonstration that the effect of the adversary can be described in a small number of bits.

5.1 Description of the 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

⁵Specifically, $\epsilon = O(\log \log \log n / \log \log n)$.

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 $\text{SCAN-UPDATE}(v)$, where v can be null. When a process p performs $\text{SCAN-UPDATE}(v)$ 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 $\text{SCAN-UPDATE}(v)$ for a null v , it simply returns the snapshot of A . In the following, all $\text{SCAN-UPDATE}()$ 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 $\text{SCAN-UPDATE}()$ operations that preserves the real time order of the operations and that corresponds to the apparent ordering determined by examining which $\text{SCAN-UPDATE}()$ operations return which values. The $\text{SCAN-UPDATE}(v)$ 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 $O(\sqrt{n} \log n)$.

For technical reasons, not every atomic snapshot algorithm can be used here without modification. The main restriction is that the algorithm must have, in addition to the above properties common to all snapshot algorithms, the property that the very first step of a $\text{SCAN-UPDATE}(v)$ operation is a write of the new value v , and that this value must be included in the result of any $\text{SCAN-UPDATE}()$ that starts after the value is written, even if the $\text{SCAN-UPDATE}(v)$ operation has done no work yet beyond that very first write. Fortunately, the Attiya-Rachman algorithm has this property.

Each process p is given a fixed permutation π_p of the blocks. On first waking (beginning a collect), p performs $\text{SCAN-UPDATE}(\text{NEWVIEW}_p)$, 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 $\text{SCAN-UPDATE}()$ ($O(\sqrt{n} \log n)$ 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, is not tagged with a timestamp whose p th component is COLLECT-NUM_p (\sqrt{n} operations).
3. **WRITE-VIEW:** Tag the block just read with the current timestamp. Let NEWVIEW_p consist of (a) this block, (b) the most recent values of each block b seen in the snapshot, along with their timestamp tags, and (c) the value COLLECT-NUM_p (which is unchanged). Update $\text{VIEW}[p]$ by invoking $\text{SCAN-UPDATE}(\text{NEWVIEW}_p)$ ($O(\sqrt{n} \log n)$ operations).

This loop repeats until all \sqrt{n} blocks appear in $\text{VIEW}[p]$ tagged with a timestamp whose p th component is COLLECT-NUM_p .

To ensure that in the worst case process p performs at most $O(n)$ operations, we interleave the execution of this loop with a naïve collect. That is: after each atomic operation performed in the loop above, insert a single read operation for a register for which the process does not yet have a trusted value (i.e., a value in a block tagged with a timestamp whose p th component is `COLLECT-NUMp`). The values learned through these single reads are not included in the process’s view and do not affect its behavior in the above loop; however, if the union of its view and these directly-read values include fresh values for all n registers, the process finishes its collect as soon as it is done with its current `SCAN-UPDATE()` operation.

It is worth noting that instead of using as a timestamp the vector of collect numbers seen by the `READ-GROUP` operation (through an atomic snapshot), one can use the sum of the \sqrt{n} components of this vector. Refer to this sum as a *round number*, and refer to the sum of the collect numbers obtained by the first `READ-GROUP` of a collect as an *initial* round number in this collect. From the fact that the collect numbers of a process increase monotonically and the linearizability of the snapshot object, it follows that a process can trust values tagged by round numbers that are not less than its initial round number in this collect. Use of this shorter timestamps will decrease the registers’ size by a factor of n . For simplicity of exposition, we overlook this possible optimization.

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 particular, we would like to show that the effect of the adversary can be described in only $O(n \log n)$ bits. In this case we can show that a “good” set of permutations exists, using a combinatorial lemma described in Section 5.2. In Section 5.3 we describe a succinct representation of the adversary. This representation, and the “good” permutations that follow from its existence, are then used to prove the competitiveness of the algorithm in Sections 5.4 and 5.5.

5.2 Good Permutations Exist

In Anderson and Woll’s Certified Write All algorithm, it is possible to describe the schedule as a single permutation σ that specifies the order in which values are first written. The reason for this simplicity is that in the Certified Write All problem freshness is not an issue, so writing a register advances the goals of all processes equally. This symmetry does not hold for the cooperative collect problem, and in consequence it is necessary to model each adversary a as a collection $R_1^a, R_2^a, \dots, R_n^a$ of permutations, one for each process.

Intuitively, R_i^a captures the behavior of adversary a that is relevant to process p_i , in that we can show that 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 $1 \dots n$ 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 (that is, values tagged with a timestamp that includes `COLLECT-NUMp_i`). 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 5.4, 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 trusted by p_i will also be trusted by p_j .

The fact that we can describe an adversary succinctly 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 set of permutations for which the sum of the longest greedy monotone increasing subsequences for each pair (R_i^a, π_i) is $O(n \log n)$.

Denote by $\ell(\sigma, \pi)$ the length of the greedy monotone increasing subsequence (LGMIS) of a permutation π according to the ordering σ .

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 $a = R_1^a, \dots, R_n^a$ where R_i^a is an ordering of the set $\{1, \dots, n\}$. For all $c_1, c_2 > 0$, there exists c_3 , so that if n is sufficiently large, $|A| \leq 2^{c_1 n \log n}$, and the permutations π_1, \dots, π_n are taken at random independently and with uniform distribution on the set of all permutations of $\{1, \dots, n\}$, then with a probability of at least $1 - e^{-c_2 n \log n}$ we have that for all $a \in A$, $\sum_i \ell(R_i^a, \pi_i) \leq c_3 n \log n$.*

Proof: The main step in proving the lemma is to show that for any single adversary a , the sum of the LGMIS's is small with high probability. To simplify the argument, note that if we fix an adversary a and a permutation σ , $\sum_i \ell(R_i^a, \pi_i) = \sum_i \ell(\sigma(R_i^a)^{-1} R_i^a, \sigma(R_i^a)^{-1} \pi_i) = \sum_i \ell(\sigma, \pi_i')$, where the π_i' are independent, uniformly distributed random permutations. Consequently, as long as we are considering only the distribution of the random variable $\sum_i \ell(R_i^a, \pi_i)$ for a single fixed a , we may assume without loss of generality that $R_i^a = \sigma$ for all i .

First we show:

Claim 5.2 *There is an $\alpha > 0$ so that if n is sufficiently large, σ is a fixed ordering on $\{1, \dots, n\}$, and π_i is a random permutation of $\{1, \dots, n\}$, then for all r , where $\alpha \log n < r \leq n$, we have that $P(\ell(\sigma, \pi_i) > r) < e^{-r/4}$.*

Proof: We may assume that σ is the natural ordering of the numbers $\{1, \dots, n\}$. Let u_1, \dots, u_t be the LGMIS of permutation π_i with respect to the ordering σ . Note that for any $m = 1, \dots, n - 1$ and $j = 1, \dots, n - 1$ the distribution of u_{j+1} , conditioned on $u_j = m$, is uniform on the set $\{m + 1, \dots, n\}$.

Let D_j , $j = 1, 2, \dots$ be the following random variable: If u_j does not exist then D_j is 0 or 1 with a probability of $1/2$, $1/2$. If u_j is defined, then $D_j=1$ if $n - u_j < \frac{1}{2}(n - u_{j-1})$, and $D_j = 0$ otherwise.

Observe that the inequality $n - u_j < \frac{1}{2}(n - u_{j-1})$ may hold for at most $\log_2 n$ distinct

values of j . Thus, for any fixed integer $r \leq n$, if $l(\sigma, \pi_i) > r$, then $\sum_{i=1}^r D_i \leq \log_2 n$. Therefore, $P(l(\sigma, \pi_i) > r) \leq P(\sum_{i=1}^r D_i \leq \log n)$.

We estimate the probability $p_r = P(\sum_{i=1}^r D_i \leq \log n)$. Clearly for any $j = 1, \dots, n$ and for any 0, 1 sequence $\delta_1, \dots, \delta_{j-1}$, we have that

$$P(D_j = 1 | D_1 = \delta_1, \dots, D_{j-1} = \delta_{j-1}) \leq 1/2.$$

(Because, if u_j is not defined then the above probability is 1/2; if it is defined then, as we have said earlier, u_j is uniform in $\{u_{j-1} + 1, \dots, n\}$ and therefore $P(u_j > \frac{1}{2}(n + u_{j-1})) \leq 1/2$.) Thus, if D'_i , $i = 1, \dots, r$ are mutually independent random variables and $D'_i = 0$, or 1 with probabilities 1/2, 1/2 for all $i = 1, \dots, r$, then $p_r \leq p'_r = P(\sum_{i=1}^r D'_i \leq \log n)$.

To complete the proof of the claim we estimate p'_r . If $r > 2 \log n$ then

$$p'_r = \sum_{j=0}^{\log n} \binom{r}{j} 2^{-j} 2^{j-r} = 2^{-r} \sum_{j=0}^{\log n} \binom{r}{j} \leq 2^{-r} (1 + \log n) \binom{r}{\lfloor \log n \rfloor}.$$

Let $d = \lfloor \log n \rfloor$, using that $d! \geq (d/e)^d$, we get that

$$2^{-r} \binom{r}{d} \leq 2^{-r} r^d / d! \leq 2^{-r} r^d (d/e)^{-d} = e^{-r \log 2 + d \log r - d \log d + d} \stackrel{\text{def}}{=} K(r).$$

Assume now that $\alpha > 0$ is picked so that for all $x > \alpha$ we have $x \log 2 > 2(\log x + 1)$. Let $\rho = r/d$. Then for all $r > \alpha d$,

$$\begin{aligned} K(r) &= K(\rho d) \\ &= e^{-\rho d \log 2 + d \log d + d \log \rho - d \log d + d} \\ &= e^{-\rho d \log 2 + d \log \rho + d} \\ &\leq e^{-(\rho/2)d \log 2} = e^{-\frac{1}{2}r \log 2}. \end{aligned}$$

Consequently for all $r > \alpha \log n$, $P(l(\sigma, \pi_i) > r) \leq p_r \leq p'_r \leq (1 + \log n) e^{-\frac{1}{2}r \log 2} < e^{-r/4}$, and we are done. \blacksquare

Next we show:

Claim 5.3 *Assume $\alpha > 0$ is the constant in Claim 5.2, $\beta > 0$ is an arbitrary real number, n is sufficiently large, σ is a fixed ordering on $\{1, \dots, n\}$, and, for each $i = 1 \dots, n$, π_i is a random permutation of $\{1, \dots, n\}$. Let B be the event*

$$\sum \{l(\sigma, \pi_i) | l(\sigma, \pi_i) > \alpha \log n\} > (4\beta + 12)n \log n,$$

then $P(B) < e^{-\beta n \log n}$.

Proof: Let $X = \{i | l(\sigma, \pi_i) > \alpha \log n\}$. Then $X = \{x_1, \dots, x_t\}$ for some t . Let $u = \langle x_1, \dots, x_t, l(\sigma, \pi_{x_1}), \dots, l(\sigma, \pi_{x_t}) \rangle$. For each t_0 that is a fixed number in $\{1, \dots, n\}$, and for each $u_0 = \langle v_1, \dots, v_{t_0}, w_1, \dots, w_{t_0} \rangle$ that is a fixed sequence of length $2t_0$ whose elements are integers between 1 and n , define B_{t_0, u_0} to be the following event: “ B and $t = t_0$ and $u = u_0$ ”.

We estimate the probability of B_{t_0, u_0} . By Claim 5.2, for any fixed $j = 1, \dots, t_0$ the probability that $l(\sigma, \pi_{v_j}) = w_j$ is at most $e^{-w_j/4}$. (Here we used the fact that $w_j \geq \alpha \log n$.) Since these events for each $j = 1, \dots, t_0$ are independent, we have that

$$P(B_{t_0, u_0}) < \prod_{j=1}^{t_0} e^{-w_j/4} \leq e^{-1/4 \sum w_j} \leq e^{-(\beta+3)n \log n}.$$

Since t_0 is an integer between 1 and n , u_0 is a sequence of length $2t_0 \leq 2n$, and each element of u_0 is an integer between 1 and n , there are at most n^{2n+1} choices for the pair t_0, u_0 . Thus,

$$P(B) \leq \sum_{t_0, u_0} P(B_{t_0, u_0}) < n^{2n+1} e^{-(\beta+3)n \log n} \leq e^{(2n+1) \log n} e^{-(\beta+3)n \log n} \leq e^{-\beta n \log n},$$

which completes the proof of the claim. ■

Since each number $l(\sigma, \pi_i)$ which is not included in the sum in Claim 5.3 is at most $\alpha \log n$, their total is at most $\alpha n \log n$, and it follows that

Claim 5.4 *Assume $\alpha > 0$ is the constant in Claim 5.2, $\beta > 0$ is an arbitrary real number, n is sufficiently large, σ is a fixed ordering on $\{1, \dots, n\}$, and, for each $i = 1, \dots, n$, π_i is a random permutation of $\{1, \dots, n\}$. Let B be the event*

$$\sum l(\sigma, \pi_i) > (4\beta + 12 + \alpha)n \log n,$$

then $P(B) < e^{-\beta n \log n}$. ■

Because $\sum_i l(\sigma, \pi_i)$ has the same distribution as $\sum_i l(R_i^a, \pi_i)$, the above claim applies equally well to the more general case where we replace σ with a possibly distinct permutation for each i . This completes the main step of the proof.

To obtain the full lemma, given c_1, c_2 , choose $c_3 = 4(c_1 + c_2) + 12 + \alpha$, where α is the constant of Claim 5.2. For each fixed $a \in A$, let D_a be the event $\sum_i l(R_i^a, \pi_i) \leq c_3 n \log n$. Claim 5.4 implies that for each fixed a , $P(D_a) \geq 1 - e^{-(c_1+c_2)n \log n}$. Therefore,

$$\begin{aligned} P(\forall a \in A, D_a) &\geq 1 - |A| e^{-(c_1+c_2)n \log n} \\ &\geq 1 - 2^{c_1 n \log n} e^{-(c_1+c_2)n \log n} \\ &= 1 - e^{-c_2 n \log n}, \end{aligned}$$

which completes the proof of the lemma. ■

5.3 Representing the Scheduling Adversary as a Combinatorial Object

Given a set Π of m permutations on $\{1, \dots, m\}$, 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 first 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 $O(m \log m)$.

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 $\{1, \dots, m\}$ 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 abuse notation slightly and denote by $l(\pi'_i, \sigma)$ the length of 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 5.5 *There is a constant c such that for each m there is a set Π of m permutations π_1, \dots, π_m of m values each such that $\Gamma(\Pi) \leq cm \log m$.*

Proof: By construction, the sets S_i can together be described using only $O(m \log m)$ bits. For each i , let OLD_i denote the set of elements in π_i but not in π'_i . For each i , let us define R_i to be the elements of OLD_i (in any order) followed by the sequence S_i . Note that the assumptions on the concise representation of the S_i 's and the containment chain property of the old subsets imply that the R_i 's can be represented using only $O(m \log m)$ bits. Moreover, for every i , and every permutation π_i , the length of the LGMIS of π_i with respect to R_i is at least as great as the length of the LGMIS of π'_i with respect to S_i . To see this, consider two cases. In the first case π_i begins with an element in π'_i . In this case, the elements in OLD_i are irrelevant to the LGMIS of π_i with respect to R_i , by definition of LGMIS and the fact that all elements in OLD_i are smaller in R_i than elements in π_i . We actually have equality in this case. In the second case, π_i begins with an element in OLD_i . Here, elements in OLD_i can add to the LGMIS, but their addition cannot affect the addition of elements in π'_i because elements in OLD_i are all less, according to R_i , than anything in π'_i . Once the first element in π'_i has been added to the LGMIS, no other elements in OLD_i will be added, and they will be irrelevant after this point.

Now, let us start with a good set Π of permutations guaranteed by Lemma 5.1. We have shown that the adversary can be described with $O(m \log m)$ bits. We have also shown how, given Π and the description of the adversary, we can construct the permutations π'_i and the sequences S_i . Applying the construction of the previous paragraph, we can construct from these the orderings R_i , so that together these can be described with $O(m \log m)$ bits. Thus the orderings satisfy the conditions of the Lemma, and we conclude that the sum over all i of the length of the LGMIS of π_i with respect to R_i is at most $cm \log m$ for some constant c that does not depend on m . Finally, we have shown that this quantity is greater than or equal to $\Gamma(\Pi)$, so $\Gamma(\Pi) \leq cm \log m$. ■

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 5.5 yields an algorithm with good latency.

5.4 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 5.5 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 5.6 *Let Π be a set of $m = \sqrt{n}$ permutations on $\{1, \dots, m\}$. Suppose that the set of permutations Π for each group satisfies $\Gamma(\Pi) = O(T(m))$, where $\Gamma(\Pi)$ is as defined in Section 5.3. Then the collective latency for each group using our algorithm is $O(T(m)\sqrt{n} \log n)$.*

Proof: Fix an arbitrary time t and let G be the set of processes performing collects at time t . Since for each READ-BLOCK phase there is at most one READ-GROUP and one WRITE-VIEW phase, together requiring $O(\sqrt{n} \log n)$ operations, we can get a bound on the number of operations after t by bounding the number of READ-BLOCKS. We distinguish between READ-BLOCKS after time t whose values are trusted by every process in G (“globally trusted” READ-BLOCKS), and other (“partially trusted”) READ-BLOCKS.

Bounding the number of partially trusted READ-BLOCKS. It is not difficult to see that each process performs at most one partially trusted READ-BLOCK after time t , as the READ-GROUP preceding its second READ-BLOCK will get the current COLLECT- NUM_p values for all processes in G . (This is the place where we use the requirement that a SCAN-UPDATE() operation returns all values that have already been written when it starts, and that its very

first operation writes out the new value. Without this requirement, it would be possible for a process to start a collect without affecting other processes' views of its `COLLECT-NUMp` value.)

Overview of bounding the number of globally trusted READ-BLOCKS. To bound the number of globally trusted READ-BLOCKS, we start by constructing a three-part adversary σ consistent with the representation described in Section 5.3. 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 shows that the READ-BLOCKS done by process i correspond to a greedy monotonic increasing subsequence according to the sequence S_i described in Section 5.3. Applying Theorem 5.5 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 $O(\sqrt{n} \log n)$.

Partitioning the READ-BLOCKS into new and old. Keeping this overview in mind, let us get into the details of the proof. For every process $p \in G$, let `G-COLLECT-NUMp` denote the collect number of p current at t , and let `G-INITIAL-UPDATEp` denote the `SCAN-UPDATE()` operation in which p updates its collect number to have the value `G-COLLECT-NUMp`. Since the `G-INITIAL-UPDATE` operations are serialized by the `SCAN-UPDATE()` procedure, we can define `LAST-G-INITIAL-UPDATE` to be the last, according to this ordering, over all $p \in G$, of the operations `G-INITIAL-UPDATEp`. Recall that during a `READ-GROUP` phase a process takes a snapshot of the collect numbers of all processes in the group. We partition the `READ-BLOCK` phases performed by processes $p \in G$ into those that started after p takes a snapshot serialized after `LAST-G-INITIAL-UPDATE` and to those that started before this point. We refer to the first class of `READ-BLOCK` phases as *new* READ-BLOCKS and to the second class as *old* READ-BLOCKS. We refer to the corresponding `READ-GROUPS` as *new* and *old*, respectively. The new READ-BLOCKS are globally trusted: they obtain values that will be accepted as trustworthy by all processes in G .

The bulk of the remainder of the proof obtains a bound on the number of new globally trusted READ-BLOCKS. This is a little different from counting the number of READ-BLOCKS that occur after time t , and indeed it must be. This is because the atomic snapshot scan algorithm yields no real notion of time, just a serialization order on snapshots and updates that is consistent with the real-time order of events. On the other hand, the definition of collective latency requires us to bound the number of operations (reads and writes of shared registers) that occur after time t . We first argue that no process can take more than $O(\sqrt{n} \log n)$ steps after time t and before beginning a new `READ-GROUP`.

Bounding the work corresponding to old globally trusted READ-BLOCKS. Consider a process $q \in G$. When q begins its collect current at t , the first thing it does is perform an atomic snapshot update $\text{SCAN-UPDATE}(\text{G-COLLECT-NUM}_q)$, and the first step of this update operation is a write of G-COLLECT-NUM_q . Moreover, it is a property of the Attiya-Rachman $\text{SCAN-UPDATE}()$ algorithm that any atomic snapshot scan begun after this write will be serialized after this update.

Now, by definition, every process $q \in G$ writes G-COLLECT-NUM_q no later than time t (since these are the collects in process at t). It follows from the above discussion that any atomic snapshot scan begun after time t is serialized after $\text{LAST-G-INITIAL-UPDATE}$. In particular, any READ-GROUP begun after time t returns a snapshot serialized after $\text{LAST-G-INITIAL-UPDATE}$ and is therefore a new READ-GROUP by definition. It thus follows from the algorithm that each process p can perform at most $O(\sqrt{n} \log n)$ steps after time t and before beginning a new READ-GROUP .

Bounding the number of new globally trusted READ-BLOCKS. Next we must bound the number of new READ-BLOCKS . Define an adversary permutation σ as follows.

1. Let $m = \sqrt{n}$. Consider the linearization order on the updates $\text{G-INITIAL-UPDATE}_p, p \in G$. Attach a trusting threshold between 1 and m to each process in G consistent with this order and such that the process p whose update $\text{G-INITIAL-UPDATE}_p$ is serialized last among all updates $\text{G-INITIAL-UPDATE}_q, q \in G$ will have trusting threshold m .
2. We construct σ' by merging two lists of \sqrt{n} pairs each. We start with the first list. Recall that the processes are partitioned into \sqrt{n} groups, and G is a (not necessarily proper) subset of the processes in a particular group. Consider all the WRITE-VIEWS that are done by processes in this group and that correspond to new READ-GROUPS . We call these WRITE-VIEWS *new* WRITE-VIEWS . Since we are using atomic snapshots to perform WRITE-VIEWS and READ-GROUPS , these operations are totally ordered. Hence, for each block there is a first time (according to this ordering) in which its values are written by a new WRITE-VIEW . Let $i_1, \dots, i_{\sqrt{n}}$ be this order, *i.e.* for each $j = 1, \dots, \sqrt{n} - 1$, the values of block i_j are written by some new WRITE-VIEW before the values of block i_{j+1} are written by any new WRITE-VIEW . Let σ' initially contain the \sqrt{n} pairs of elements $(i_1, \sqrt{n}), \dots, (i_{\sqrt{n}}, \sqrt{n})$, in order. This gives us an ordered list of \sqrt{n} pairs.

The second list of \sqrt{n} pairs is constructed as follows. For each process p in the \sqrt{n} -sized group that contains G consider the last old READ-BLOCK done by p (*i.e.*, the last READ-BLOCK by p that follows a READ-GROUP whose atomic snapshot was not serialized after $\text{LAST-G-INITIAL-UPDATE}$). We refer to the corresponding WRITE-VIEW by p as an *almost new* WRITE-VIEW . There are at most \sqrt{n} such WRITE-VIEWS (one for each process in the group). For each such almost new WRITE-VIEW , we construct a pair whose value is the index of the block read in the corresponding READ-BLOCK , and whose trustworthiness is determined by the timestamp on the block as follows. The timestamp is a vector of collect numbers, some of which were written by updates $\text{G-INITIAL-UPDATE}_q, q \in G$. Of these updates, consider the update that is serialized last, and let r be the process

executing this last update. Then the trustworthiness of the pair is the trusting threshold of r , determined above. Order this set of pairs according to the serialization order of the WRITE-VIEWS corresponding to the pairs: if p 's almost new WRITE-VIEW is serialized before q 's almost new WRITE-VIEW, then order the pair corresponding to p 's almost new WRITE-VIEW before the pair corresponding to q 's almost new WRITE-VIEW. This gives us a second list of \sqrt{n} pairs.

Finally, we merge the two lists as follows. A pair (i, \sqrt{n}) from the first list precedes a pair (j, t) from the second list if and only if the first WRITE-VIEW containing block i tagged with trusting threshold corresponding to \sqrt{n} (that is, the first write of block i trusted by everyone in G) is serialized before the almost new WRITE-VIEW that gave rise to the pair (j, t) . Since all the WRITE-VIEWS are serialized this merging is well-defined and results in an ordered list of $2\sqrt{n}$ elements.

3. For the third part of the adversary, consider all (block, timestamp) pairs written (at any time) by a WRITE-VIEW done by a process in the group containing G , in which the timestamp is serialized before LAST-G-INITIAL-UPDATE. For each process p in the group containing G , remove from this set of pairs the last pair to be written by p ; that is, remove the pair written in p 's almost new WRITE-VIEW. We call the resulting set *old pairs*. Now, for each process $p \in G$, we define *old subset* associated with p 's trusting threshold to be the set of all block names b for which there exists an old pair (b, ts) where ts is a timestamp serialized after G-INITIAL-UPDATE $_p$. The values in this subset are trusted by p , but not necessarily by processes with a higher trusting threshold than p has. Observe that since there is a total order on the timestamps and the updates G-INITIAL-UPDATE $_p$, for $p \in G$, these old subsets are also totally ordered.

Clearly, the adversary above is a legitimate adversary according to our definition of the previous section. Hence it follows from the theorem assumption that the sum of the LGMISS of the π'_i with respect to this adversary σ is $O(T(m))$. Thus, to complete the proof we show:

Lemma 5.7 *For each process i , the id's of the blocks read by it while performing new READ-BLOCKS appear in the longest greedy monotonic increasing subsequence of π'_i with respect to σ .*

Proof: Suppose the claim does not hold. Then there is a process i that does a READ-BLOCK of block q , and q either does not appear in π'_i , or there is some block p that precedes q in π'_i but is larger than q according to S_i .

Recall that π'_i is obtained from π_i by erasing all the elements that are contained in the old subset corresponding to i 's trusting threshold. Thus, if q does not appear in π'_i , then q is in the old subset associated with i 's trusting threshold. By definition of the old subsets, this means that q was written by an old WRITE-VIEW but not by an almost new WRITE-VIEW. Any READ-GROUP serialized after LAST-G-INITIAL-UPDATE is by definition a new READ-GROUP (that is, it is associated with a new READ-BLOCK). Thus, no almost new READ-GROUP can be serialized after LAST-G-INITIAL-UPDATE. It follows that no old WRITE-VIEW can be serialized after

LAST-G-INITIAL-UPDATE, since such a WRITE-VIEW must be followed by an almost new READ-GROUP, but any READ-GROUP begun after such a WRITE-VIEW will necessarily be serialized after LAST-G-INITIAL-UPDATE, and hence will be new. Thus, q was written by a WRITE-VIEW serialized before LAST-G-INITIAL-UPDATE, and was tagged by a timestamp trusted by i . Thus i learns a trusted value for q no later than during its first new READ-GROUP, contradicting the assumption that it reads q during a new READ-BLOCK.

Finally, assume there is some block p that precedes q in π'_i but is larger than q according to S_i . Since p precedes q , i must have read a trustworthy value of it before it READ-BLOCKS q . It follows from S_i , that a trustworthy value of q was already written before the trustworthy value of p . Thus i must have read also a trustworthy value for q at the same time, contradicting the fact it READ-BLOCKS it later. ■

Combining the result of the lemma with the fact shown above that the sum of the LGMISS of the π'_i with respect to the adversary σ is $O(T(m))$, thus yields a bound of $\Gamma(\Pi) = O(T(m))$ on the number of new globally trusted READ-BLOCKS. To complete the proof of Theorem 5.6, observe that the collective latency for each group is bounded by \sqrt{n} times the maximum amount of work performed by any process p after time t and before p begins a new READ-GROUP, plus $O(T(m))$ (the upper bound on the number of new globally trusted READ-BLOCKS) times the cost of an iteration of the main loop of the algorithm. Thus, the collective latency for each group using our algorithm is $O(n \log n + T(m)\sqrt{n} \log n) = O(T(m)\sqrt{n} \log n)$. ■

5.5 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.

Note that in this section we consider only *finite* schedules.

Theorem 5.8 *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.*

Proof: By the definition of latency competitiveness (Section 4.1), it is enough to consider finite prefixes of the schedule. Consider an arbitrary finite prefix, and let t_e be the time at which the last atomic step in this prefix takes place. 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.

This slicing is carried out recursively. To simplify its description, let us treat the schedule as assigning operations not to processes but to individual collects; an operation will be assigned to a particular collect C if it is assigned to C 's process and occurs after C starts but before C 's process starts any other collect.

As long as $t_i < t_e$, we will choose $t_{i+1} \leq t_e$ so that $I_i = (t_i, t_{i+1}]$ is the longest interval starting at t_i in which the number m of operations assigned to collects that start during I_i and that are performed by processes in G is at most n . Let C_i be the set of collects that begin during I_i . We claim that either:

1. $t_{i+1} = t_e$, or
2. $t_{i+1} < t_e$, and $m = n$.

The proof of this claim is that if $(t_i, t_{i+1}]$ contains fewer than n operations from collects in C_i , and there is some first operation α after t_{i+1} , we can increase t_{i+1} to include α without violating the definition. Since these n operations must be carried out by somebody, we also have that if $t_{i+1} < t_e$ then C_i is nonempty.

Let k be the number of intervals defined above. First we show that if $1 \leq i < k$ (*i.e.* I_i is not the last interval), then the champion must perform at least n operations during I_i . Observe that any collect that starts after t_i cannot trust any value read before t_i , so if C_i is nonempty, at least n operations must be carried out by the champion after t_i before any collect in C_i can finish. If $1 \leq i < k$, then as observed above, C_i is nonempty, and by the definition of I_i , the processes performing collects in C_i are together scheduled to perform n operations during I_i . At each such step the process in the champion algorithm must perform its operation, *unless* it has already finished its collect. So if none of these processes finish their collects, all n of these scheduled operations are performed; but if a process has already finished its collect, by the earlier observation some n operations must have been performed since t_i (though possibly by processes not performing collects in C_i). In either case, n operations are performed in I_i by the champion.

Similarly, if C_k is nonempty, then the reasoning above immediately implies that the champion performs in I_k at least $\min\{n, p\}$ operations, where p is the number of operations that the processes performing collects in C_k are together scheduled to perform in I_k . By our definition of I_i , it follows that $p \leq n$. Thus, if C_k is nonempty, the champion performs at least p operations in I_k .

To complete the proof it is enough to show that in our algorithm the collects starting in each interval require at most $L + n$ operations. For each $1 \leq i < k$, divide the operations performed as part of collects in C_i into those occurring during I_i and those occurring after I_i , *i.e.* after t_{i+1} . The first set consists of at most n operations by the definition of I_i ; the second

consists of at most L operations since it is bounded by the collective latency of the group of processes carrying out collects at time t_{i+1} .

We now have two cases, according to whether or not C_k is empty. If C_k is empty, then our algorithm performs at most $(k-1)n + (k-1)L$ steps, while the champion performs at least $(k-1)n$ steps. If C_k is nonempty, then let p be the number of steps that the processes performing collects in C_k are together scheduled to perform in I_k . Our algorithm performs at most $(k-1)n + (k-1)L + p$ steps, while the champion performs at least $(k-1)n + p$ steps. In either case the work ratio is at most $L/n + 1$. ■

Corollary 5.9 *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 at most $mL/n + m$.*

Proof: Let w be the number of operations done by the champion. Then from Theorem 5.8, no single group in the candidate algorithm does more than $w(L/n + 1)$ operations. It follows that the m groups together do a total of at most $wm(L/n + 1)$ operations. ■

Lemma 5.10 *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 5.6, 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 5.9 ■

The set of permutations Π from Theorem 5.5 have $T(\sqrt{n}) = O(\sqrt{n} \log n)$, and thus:

Theorem 5.11 *The competitive latency of the Speedy Collect algorithm is $O(n^{1/2} \log^2 n)$.*

Observe that for each schedule, the work performed by each of the \sqrt{n} -sized groups into which the processes are partitioned is at most $O(\log^2 n)$ times the total work performed by all groups in the best possible algorithm for this schedule. Moreover, as can be seen in the beginning of the proof of Theorem 5.6, one of the $\log n$ factors in the competitive latency is due to the fact that the READ-GROUPS are performed using the atomic snapshot algorithm of Attiya-Rachman. In this atomic snapshot a process takes $O(n \log n)$ operations to complete an atomic snapshot of n registers. If one had a linear algorithm for snapshot, then the work performed by each of the \sqrt{n} -sized groups would be just $O(\log n)$ times the total work performed by all groups in the best possible algorithm for this schedule, matching (within each group) our trivial lower bound of Section 7. Thus it is unlikely that substantial further improvements are possible as long as a division of the processes into isolated groups is necessary.

6 A Constructive Algorithm

The preceding section describes an $O(n^{1/2} \log^2 n)$ competitive algorithm that is based on a set Π of permutations which satisfy a very strong constraint. The existence of such a set is shown probabilistically. In this section we develop an algorithm whose competitive latency is only $O(n^{(3/4)+\epsilon} \log^2 n)$, but which is based on an explicit set of permutations. The new algorithm requires more machinery to make up for this set's weaker properties.

As before, the processes are divided into \sqrt{n} groups of \sqrt{n} processes each, and will read registers in blocks of \sqrt{n} registers at a time. However, in this algorithm each process will simulate $O(n^{1/4})$ *virtual* processes running a one-time collect algorithm. For each of these virtual processes, the process will maintain a separate view marked by a *round number*. Virtual processes with a particular round number will cooperate only with other virtual processes with the same round number.

The round numbers are assigned so that every virtual process in a given round can trust other processes in the same round to supply it with only fresh values. This eliminates the use of timestamps in deciding what values to read, and is the key technique that allows the use of a simpler set of permutations than required by the non-constructive algorithm.

To obtain these round numbers, we need to use a slightly more elaborate version of the timestamps used in the non-constructive algorithm. Each process p will include in its register a value COLLECT-NUM_p that is a count of how many collects it has started. To initiate a collect, the process will increase this value by one using a $\text{SCAN-UPDATE}()$ operation as provided by the Attiya and Rachman protocol. (This is the only time a process will change its COLLECT-NUM value.) Its initial round number in this collect is given by the sum of all the COLLECT-NUM values it sees in this $\text{SCAN-UPDATE}()$ operation; essentially, it is this process's view of how many collects have been initiated by processes in its group.

The process starts by simulating just one virtual process, which will be marked with its initial round number. It then proceeds through the following loop. (Note that the READ-GROUP step during the very first pass through the loop is implemented using the same $\text{SCAN-UPDATE}()$ operation that is used to obtain the initial round number.)

1. READ-GROUP : p performs a $\text{SCAN-UPDATE}()$ on the registers of all processes in the group to obtain the views of all virtual processes being simulated by the group ($\sqrt{n} \log n$ operations). For each virtual process that p is simulating, p will add to its view the union of the views of all virtual processes with the same round number.

At this step p may also begin simulating a new virtual process. Let i be p 's starting round number in this collect. For each round number $j > i$ that is a multiple of $n^{1/4}$, if p observes a round number greater than or equal to j and p does not already simulate a virtual process at round j , then p will start simulating a virtual process in round j with an initial view equal to the union of the views of all virtual processes in round j .

2. MULTI-READ-BLOCK : For each virtual process that p is simulating, p performs a READ-BLOCK on behalf of that process, reading the first block in p 's permutation that is not in

the virtual process's view (\sqrt{n} operations for each active virtual process). Add this block to that process's view, tagged with a timestamp that consists of the `COLLECT-NUMq` values read by the previous `READ-GROUP`.

3. `WRITE-VIEW`: p writes the current views of all simulated processes to its register using `SCAN-UPDATE()` ($O(\sqrt{n} \log n)$ operations).

If at any time the union of the views read by p during its most recent `READ-GROUP` and of the views of p 's own virtual processes contains a value for each register that is tagged with a timestamp including `COLLECT-NUMp`, p skips immediately to its next `WRITE-VIEW` step and exits thereafter. In doing so, it stops simulating any virtual processes. However, it must still maintain the views of all the virtual processes it has ever simulated in its register.⁶

To ensure that in the worst case process p performs at most $O(n)$ operations, we interleave the execution of this loop with a naïve collect, as done in Section 5.1. That is: after each atomic operation performed in the loop above, insert a single read operation for a register for which the process p does not yet have a trusted value. The values learned through these single reads are not included in the process's view and do not affect its behavior in the above loop; however, if the union of its view and these directly-read values include fresh values for all n registers, the process finishes its collect as soon as it is done with its current `SCAN-UPDATE()` operation.

As observed in Section 5.1, also here a value read during a `READ-BLOCK` operation, can be tagged with only the round number of the virtual process that read it, instead of with the vector of `COLLECT-NUMS` read by the simulating process during the preceding `READ-GROUP` operation. Again, for simplicity of exposition, we overlook this possible optimization.

Some properties of the algorithm do not depend on the choice of permutations. In particular, we note:

Lemma 6.1 *Any value returned by the above algorithm is fresh.*

Proof: At the end of a collect p returns a vector of values each of which was tagged with a timestamp including the current value of `COLLECT-NUMp`. But such a value must have been read after a `READ-GROUP` that obtained this current value, which in turn must have followed p 's write at the start of its collect. ■

The following lemma shows that virtual processes in the same round can always trust each other's values to be fresh.

Lemma 6.2 *Let r be the initial round number calculated by a process p as part of a collect C in which p sets `COLLECT-NUMp` to c . Let S be any `SCAN-UPDATE()` operation that returns*

⁶This is done for simplicity of exposition; the proofs hold unchanged also if each process only maintains the views of the virtual processes it simulated in the most recent collect it completed.

a vector of COLLECT-NUM values that sum to $r' \geq r$. Then S returns a value $c' \geq c$ for COLLECT-NUM $_p$.

Proof: From the fact that the vector of COLLECT-NUM values increase monotonically and the linearizability of the snapshot object, it follows that $r' \geq r$ if and only if the vector of COLLECT-NUM values obtained by S is componentwise greater than or equal to the vector obtained by the first SCAN-UPDATE() operation in C . But the first SCAN-UPDATE() operation in C returns a vector in which COLLECT-NUM $_p = c$. ■

An immediate corollary of the lemma is that it is not difficult to show that the number of virtual processes active as part of any one collect is bounded:

Lemma 6.3 *No process ever simulates more than $n^{1/4} + 1$ active virtual processes during a single collect.*

Proof: Consider a single collect of some process p and let i be its starting round number for this collect. Suppose p sees a round number greater than or equal to $i + \sqrt{n}$ during a READ-GROUP phase; then at least \sqrt{n} collects have been started by the $\sqrt{n} - 1$ processes other than p since the beginning of p 's collect. By the Pigeonhole Principle, some process must have started two of these collects. The first of these two collects has a starting round number j greater than i , and must have written out a complete set of values in the views of virtual processes with round numbers j or greater before it exited (and thus before the second collect started). By Lemma 6.2, these values will be tagged with timestamps that include the current value for COLLECT-NUM $_p$; so p takes this view and finishes. Consequently p never starts a virtual process with a round number greater than or equal to $i + \sqrt{n}$, and the bound follows. ■

It remains to show that not too much work is done on behalf of the virtual processes. To do so, we must choose our permutations carefully.

Recall that J. Naor and R. Roth [51] have constructed a set of m permutations on the numbers $1 \dots m$ such that the sum of the lengths of all longest greedy monotonic increasing subsequences on the set with respect to any ordering σ is $O(m^{1+\epsilon} \log m)$, where ϵ tends to zero as n goes to infinity. Taking m to be \sqrt{n} gives us a set of permutations for each of our processes that has the property that the sum of the lengths of the LGMIS is $O((\sqrt{n})^{1+\epsilon} \log n)$. Using this fact we prove the following lemma:

Lemma 6.4 *At most $O((\sqrt{n})^{1+\epsilon} \log n)$ READ-BLOCKS are done by virtual processes in each round.*

Proof: Fix an arbitrary round r , and define the adversary permutation σ of the blocks by the order in which they are first written as parts of views of virtual processes in this round,

as determined by the serialization order of the SCAN-UPDATE() operations. Note that it is an immediate consequence of the serializability of the SCAN-UPDATE() operations that any view for round r that is obtained in a READ-GROUP step will be a prefix of σ .

Let p be a virtual process in round r and let π be its permutation. We will show that every block read by p as part of a READ-BLOCK is in λ , the LGMIS of π with respect to σ . The full result then follows from the fact that the bound on the sum of the lengths of the LGMISs for all virtual processes in round r is $O((\sqrt{n})^{1+\epsilon} \log n)$.

Let b be a block that is *not* in λ . From the fact that b is not in λ it follows that there is some a in λ that precedes b in π but follows b in σ .

Since the view for round r obtained in any READ-GROUP step is a prefix of σ , any such view that contains a must also contain b . So after a READ-GROUP, either p 's view does not contain a , in which case p will not read b in the immediately following READ-BLOCK because b cannot be the first unread block in the π ordering; or p 's view does contain a , in which case it also contains b , and again p will not read b . Thus any block that p *does* read must be in λ , as claimed above. ■

Recall that the collective latency of a set of processes at some time t is the sum the number of operations done by each process between t and the time it finishes the last collect that it started at or before t .

Theorem 6.5 *The collective latency for each group of \sqrt{n} cooperating processes in our algorithm is at most $n^{(5/4)+\epsilon} \log^2 n$.*

Proof: Fix a time t . Analogously to the proof of Theorem 5.6, the bound on the number of operations done by the processes will follow from a bound on the number of READ-BLOCKS done on behalf of their virtual processes as part of the current collect. We restrict our attention to READ-BLOCKS (and by extension MULTI-READ-BLOCKS) that are part of the current collect of some process in the group.

Call a MULTI-READ-BLOCK *old* if its preceding READ-GROUP starts before t , and *new* otherwise. There is at most one old MULTI-READ-BLOCK for each of the \sqrt{n} processes in the group, for a total of \sqrt{n} old MULTI-READ-BLOCKS. From Lemma 6.3 these old MULTI-READ-BLOCKS contribute $O(n^{3/4})$ READ-BLOCKS.

To bound the number of new MULTI-READ-BLOCKS, we must look more closely at the round structure. Let i be the largest round number of any virtual process being simulated by a process in the group at time t . Let j^- be the largest multiple of $n^{1/4}$ less than or equal to i and let j^+ be the smallest multiple of $n^{1/4}$ greater than or equal to i . Divide the new MULTI-READ-BLOCKS into three classes based on the round numbers of the virtual processes carrying out their component READ-BLOCKS: (i) those that include round j^- but not round j^+ ; (ii) those that include neither round j^- nor round j^+ ; and (iii) those that include round j^+ . By Lemma 6.4 there are no more than $O((\sqrt{n})^{1+\epsilon} \log n)$ MULTI-READ-BLOCKS in each of

classes (i) and (iii). Each such MULTI-READ-BLOCK consists of $O(n^{1/4})$ READ-BLOCKS for a total of $O(n^{(3/4)+\epsilon} \log n)$ READ-BLOCKS for classes (i) and (iii) together.

Consider now a process p that performs a new MULTI-READ-BLOCK of class (ii). Since its preceding READ-GROUP was after time t , it must have observed virtual processes at all rounds less than or equal to i . So the fact that p is not simulating a virtual process for round j^- implies that its starting round number i' exceeds j^- . But its starting round number must also be at most i , which puts it between j^- and j^+ . Since p does not do a READ-BLOCK for round j^+ , it must not have observed any process at round j^+ or higher. But then it is simulating only a single round i' virtual process and its MULTI-READ-BLOCK consists of a single READ-BLOCK for that virtual process.

Since each class (ii) MULTI-READ-BLOCK consists of a single READ-BLOCK in a round i' with $j^- < i' < j^+$, and there are $O((\sqrt{n})^{1+\epsilon} \log n)$ READ-BLOCKS for each of these rounds, there are $O(n^{(3/4)+\epsilon} \log n)$ class (ii) MULTI-READ-BLOCKS altogether. The total number of READ-BLOCKS contributed by both old MULTI-READ-BLOCKS and new MULTI-READ-BLOCKS of classes (i), (ii), and (iii) is thus $O(n^{(3/4)+\epsilon} \log n)$, and since for each READ-BLOCK that a process does it carries out $O(\sqrt{n} \log n)$ operations (including operations that are parts of the corresponding READ-GROUP and WRITE-VIEW phases) the collective latency is $O(n^{(5/4)+\epsilon} \log^2 n)$. ■

Combining the above lemma with Corollary 5.9 gives:

Theorem 6.6 *The competitive latency of our algorithm using the explicit construction of Naor and Roth [51] is $O(n^{(3/4)+\epsilon} \log^2 n)$.*

7 Lower Bound on Competitive Latency

It is not difficult to show a lower bound of $\Omega(\log n)$ on the competitive latency of any cooperative collect protocol. The essential idea is that the adversary will run the processes one at a time until each finishes a collect. In the champion, the first process will read all the registers (n operations) and write out a summary of their values (1 operation). The other processes each carry out one read operation to obtain this summary. In the candidate algorithm, however, the adversary can arrange things so that later processes do not know which process went first. So the second process will have to read (on average) $n/2$ output registers before it finds the first process's summary; the third process will have to read $n/3$ before it finds either the first or the second process, and so forth, for a total of at least $n(1 + 1/2 + 1/3 + \dots + 1/n) = \Omega(n \log n)$ expected operations.

To make this idea work, we need to deal with two technical issues. The first issue is that we must consider freshness. In order for the processes in the champion algorithm to profit from the efforts of the first process, that first process must be able to certify the values it collects as fresh for the other processes. This means that we must prefix the main body of the schedule with a preamble in which each process carries out one write (of a timestamp, in the

champion) or possibly some other operation. The second issue is that we must ensure that the processes in the candidate cannot guess which processes run first in the main body. This issue is resolved by choosing the order in which the processes run uniformly at random, but some care is needed to show that no information about the random permutation leaks.

Theorem 7.1 *There exists a distribution over schedules such that the expected competitive latency of any collect algorithm on a schedule drawn from this distribution is at least $\Omega(\log n)$.*

Proof: We define a probability distribution over schedules for which the expected competitive latency of any candidate algorithm is at least $\Omega(\log n)$. It follows that for any particular algorithm, there is some fixed schedule that produces a competitive latency of $\Omega(\log n)$.

The schedules we will consider consist of a preamble, in which each process starts a collect and takes one step; and a main body, in which the processes are run sequentially. The structure of the preamble is fixed: at the k -th time unit in the preamble, process k starts a new collect and is given one step. The structure of the main body is slightly more complicated: it consists of n intervals, one for each process. In each interval, only the process associated with that interval may take steps; the length of the interval is $2n$. Which interval is associated with which process is controlled by a random permutation π , chosen from a uniform distribution.

In the champion algorithm: (1) each process writes out a timestamp during the preamble (n operations total); (2) the first process in the main body reads the n output registers of all processes, then the n registers containing the values to be collected (these registers may be identical to the output registers of the processes), and writes out the result, tagged with the timestamps of the other processes ($2n$ operations); finally, (3) each of the other processes reads the output register of the first process to obtain all of the values ($n - 1$ total operations). The cost to the champion of completing one round is thus $4n$ operations.

In the candidate algorithm, first observe that the preamble is too short to allow any process p to learn a secondhand value from another process q , since q would have to carry out at least two operations to read an input register and write its value to the output register. Furthermore, as the schedule in the preamble is fixed, it reveals no information about π .

Now fix a process p , and suppose that it is the k -th process to run in the main body. We wish to show that p executes at least $\frac{n}{k} - 1$ operations in the main body before finishing its collect. If $k = 1$, p is the first process, and it must execute at least $n - 1$ reads of input registers to obtain all n values (it may have read one value during the preamble, but it cannot learn any values secondhand from other processes).

Alternatively, if $k > 1$, either p finishes its collect by reading all values directly (at the cost of $n - 1$ main body operations), or at some point it learns a value secondhand. We will count the expected number of steps until this latter event occurs. To do so we must be very careful about what information p has at each point in time about the distribution of π .

Let t_i be the earliest time in the main body at which p has read the output registers of i other processes at least once. Let P_i be the distribution of values for π conditioned on the

information obtained by p up to time t_i and define P_0 to be the uniform distribution.

Claim 7.2 *If at time t_i , p has not yet read the output register of any process that precedes it in π , then each of the $(n - i)!$ permutations of p and the $(n - i - 1)$ processes it has not yet read are equally likely according to P_i .*

Proof: Suppose the claim is true for P_{i-1} , and that at time t_i , p reads the output register of a process q that follows p in π . Since q has not performed any operations in the main body, the value in its register reveals nothing about the relative ordering of any processes except p and q , and the claim follows for P_i . ■

From the claim it is immediate that if p has not yet found a process that precedes it, its odds of finding one on its next read of an output register are no better than chance. Thus since p is looking for the output of one of $k - 1$ earlier processes in $n - 1$ registers, it takes an expected n/k reads until it finds one that could contain secondhand values that are fresh for this round. This gives a lower bound of $\frac{n}{k} - 1$ on the expected number of operations executed by p in the main body.

Summing these expectations over all processes p (which we can do even though the corresponding random variables may not be independent) gives a total of $n(1/1 + 1/2 + \dots + 1/n) = \Omega(n \log n)$ operations per round. Since the champion needs only $4n$ operations for each round, this gives a competitive ratio of $\Omega(\log n)$. ■

In the above schedules each process performs at most one collect. However, roughly speaking, for any integer p , one can construct a schedule in which each process performs p collects, by simply iterating the above schedule p times. More specifically, the new schedule will consist of a sequence of p rounds. Each round consists of a preamble in which each process starts a collect and takes one step and a main body in which each process is given enough steps to finish its collect. (We allow each process to complete a collect in each round in order to guarantee that by the time a process receives a new request to perform a collect, it has already completed its previous collect; this requirement was not necessary in the construction of Theorem 7.1 since there a process receives only a single request to perform a collect.) The structure of the preamble is fixed: at the k -th time unit in the preamble, process k starts a new collect and is given one step. The main body consists of n intervals, one for each process. In each interval, only the process associated with that interval may take steps; the length of the interval is the longer of $2n$ or the time needed by the process to complete its collect in the candidate algorithm. Which interval is associated with which process is controlled by a random permutation π , chosen from a uniform distribution independently of the choices made in other rounds.

Exactly as in the proof of Theorem 7.1, it follows that the expected competitive latency of any collect algorithm over the schedules constructed above is at least $\Omega(\log n)$ per round, and hence is $\Omega(\log n)$ when we sum over all rounds. This shows that the lower bound of Theorem 7.1 arises in arbitrarily long schedules and is not merely a side effect of giving the candidate too few steps.

However, the above construction, in contrast to the construction of Theorem 7.1, assumes that the adversary is either adaptive or knows *a-priori* some bound on the number of steps a process may take to finish a collect. This is a consequence of the requirement that the adversary supply a schedule that allows the candidate to complete its collects.

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 *Inform. 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 $O(n \log n)$ Operations. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 29–40, Aug. 1993.
- [14] B. Awerbuch and Y. Azar. Local optimization of global objectives: Competitive distributed deadlock resolution and resource allocation. In *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pp. 240–249, November 1994.
- [15] 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.
- [16] 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.
- [17] B. Awerbuch and D. Peleg. Sparse Partitions. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pp. 503–513, November 1990.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] J. Buss and P. Ragde. Certified Write-All on a Strongly Asynchronous PRAM. *Manuscript*, 1990.
- [24] T. Chandra and C. Dwork. Using Consensus to solve Atomic Snapshots. *Submitted for Publication*

- [25] 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.
- [26] 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.
- [27] 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.
- [28] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time-lapse snapshots. *Proceedings of Israel Symposium on the Theory of Computing and Systems*, 1992.
- [29] 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.
- [30] C. Dwork, M. Herlihy, and O. Waarts. Bounded Round Numbers. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 53–64, 1993.
- [31] 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.
- [32] 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.
- [33] M. Fischer and A. Michael, Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. Research Report 221, Yale U., Feb. 1982.
- [34] R. Gawlick, N. Lynch, and N. Shavit. Concurrent Timestamping Made Simple. *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1992.
- [35] 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.
- [36] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment, *Journal of the Association for 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.
- [37] 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.
- [38] M.P. Herlihy. Randomized wait-free concurrent objects. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, August 1991.

- [39] A. Israeli and M. Li. Bounded Time Stamps. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, 1987.
- [40] A. Israeli and M. Pinhasov. A Concurrent Time-Stamp Scheme which is Linear in Time and Space. Manuscript, 1991.
- [41] 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
- [42] P. Kanellakis and A. Shvartsman. Efficient Robust Parallel Computations. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, pp. 23–36, 1991.
- [43] E. Koutsoupias and C. Papadimitriou. Beyond Competitive Analysis. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 394–400, November 1994.
- [44] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining Tentative and Definitive Algorithms for Very Fast Dependable Parallel Computing. In *Proc. 23rd ACM Symposium on Theory of Computing*, pp. 381–390, 1991.
- [45] A. Kedem, K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 138–148, 1990.
- [46] 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.
- [47] L. Lamport. On Interprocess Communication, Parts I and II. *Distributed Computing 1*, pp. 77–101, 1986.
- [48] C. Martel and R. Subramonian. On the Complexity of Certified Write-All Algorithms. *Manuscript*, 1993.
- [49] 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.
- [50] 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.)
- [51] J. Naor and R. M. Roth. Constructions of permutation arrays for certain scheduling cost measures. *Manuscript*.
- [52] G. Neiger. Using knowledge to achieve consistent coordination in distributed systems. *Manuscript*, 1990.
- [53] G. Neiger and M. R. Tuttle. Common knowledge and consistent simultaneous coordination. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, 1990.
- [54] C.H. Papadimitriou and M. Yannakakis. Linear Programming Without the Matrix. In *Proc. 25th ACM Symposium on Theory of Computing*, pp. 121–129, May 1993.

- [55] 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.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] 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.