Parallel Algorithms with Processor Failures and Delays

Jonathan F. Buss* Paris C. Kanellakis† Prabhakar L. Ragde‡ Alex A. Shvartsman§

July 19, 1991

Abstract

We study efficient deterministic parallel algorithms on two models: restartable fail-stop CRCW PRAMs and strongly asynchronous PRAMs. In the first model, synchronous processors are subject to arbitrary stop failures and restarts determined by an on-line adversary and involving loss of private but not shared memory; the complexity measures are *completed work* (where processors are charged for completed fixed-size update cycles) and *overhead ratio* (completed work amortized over necessary work and failures). In the second model, the result of the computation is a serializaton of the actions of the processors determined by an on-line adversary; the complexity measure is *total work* (number of steps taken by all processors). Despite their differences the two models share key algorithmic techniques.

We present new algorithms for the *Write-All* problem (in which P processors write ones into an array of size N) for the two models. These algorithms can be used to implement a simulation strategy for any N processor PRAM on a restartable fail-stop P processor CRCW PRAM such that it guarantees a terminating execution of each simulated N processor step, with $O(\log^2 N)$ overhead ratio, and $O(\min\{N + P\log^2 N + M\log N,\ N \cdot P^{0.59}\})$ (sub-quadratic) completed work (where M is the number of failures during this step's simulation). We also show that the *Write-All* requires $N - P + \Omega(P \log P)$ completed/total work on these models for $P \leq N$.

Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada. Electronic mail: jfbuss@maytag.waterloo.edu. The research of this author was supported by NSERC Operating Grant OGP0009171.

†*Computer Science Dept., Brown University, PO Box 1910, Providence, RI 02912, USA.* Electronic mail: pck@cs.brown.edu. The research of this author was supported by NSF grant IRI-8617344 and ONR grant N00014-91-J-1613.

‡*Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.* Electronic mail: plragde@maytag.waterloo.edu. The research of this author was supported by NSERC Operating Grant OGP0041913.

§*Computer Science Dept., Brown University, PO Box 1910, Providence, RI 02912, USA,* and *Digital Equipment Corporation, Telecomm. & Networks, 550 King Street, Littleton, MA 01460, USA.* Electronic mail: aas@cs.brown.edu.

Contents

1 Introduction

1.1 Context of this work

The model of parallel computation known as the Parallel Random Access Machine or PRAM [FW 78] has attracted much attention in recent years. Many *efficient* and *optimal* algorithms have been designed for it; see the surveys [EG 88, KR 90]. The PRAM is a convenient abstraction that combines the power of parallelism with the simplicity of a RAM, but it has several unrealistic features. The PRAM requires: (1) simultaneous access (requiring significant bandwidth) to a shared resource, namely memory; (2) global processor synchronization; and (3) perfectly reliable processors, memory and interconnection between them. The gap between the abstract models of parallel computation and realizable parallel computers is being bridged by current research. For example, memory access simulation in other architectures is the subject of a large body of literature surveyed in [Val 90a]; for some recent work see [HP 89, Ran 87, Upf 89]. Asynchronous PRAMs are the subject of [CZ 89, CZ 90, Gib 89, MSP 90, Nis 90]. Here we address the issues of synchronization and reliability of PRAM processors.

In [KS 89] it is shown that it is possible to combine efficiency and fault-tolerance in many key PRAM algorithms in the presence of arbitrary dynamic fail-stop processor errors (when processors fail by stopping and do not perform any further actions). The key to such algorithm design is the following fundamental problem, called the *Write-All* problem:

Given a P-processor PRAM and a 0-valued array
of N elements, write value 1 into all array locations.

This problem was formulated to capture the essence of the computational progress that can be naturally accomplished in unit time by a PRAM (when $P = N$). In the absence of failures, this problem is solved by a trivial and optimal parallel assignment. However, it is not obvious how to design solutions that are efficient in the presence of failures or asynchrony. [KS 89] give an algorithm for the *Write-All* problem that does a total of $O(N \log^2 N)$ work.

The iterated *Write-All* paradigm is employed (independently) in [KPS 90] and [Shv 89] to extend the results of [KS 89] to arbitrary PRAM algorithms (subject to fail-stop errors without restarts). In addition to the general simulation technique, [KPS 90] analyzes the expected behavior of several solutions to *Write-All* using a particular random failure model. [Shv 89] presents a deterministic optimal work execution of PRAM algorithms subject to worst case failures given *parallel slackness* (as in [Val 90b]).

A simple randomized algorithm that serves as a basis for simulating arbitrary PRAM algorithms on an asynchronous PRAM is presented in [MSP 90]. This randomized asynchronous simulation has very good expected performance for the *Write-All* problem when the adversary is off-line. Recently, [KPRS 90] further refined the results of [KPS 90] to produce an approach that leads to constant expected slowdown of PRAM algorithms when the power of the adversary is restricted. [KPRS 90] also improved the fail-stop deterministic lower and upper bounds of [KS 89] (by $\log \log N$ factors).

The work presented here deals with dynamic patterns of faults and the dynamic assignment of processors to tasks. Processors in our algorithms have very little private information and communicate via shared memory. For recent advances on coping with static fault patterns, see [K* 90].

We consider fault granularity at the processor level; for recent work on gate granularities, see [AU 90, Pip 85, Rud 85]. The general problem of assigning active processors to tasks has similarities to the problems of resource management in a distributed setting, such as distributed controllers of [LGFG 86] and [AAPS 87]. Fault-tolerance of particular network architectures is also studied in [DPPU 86]. However, the distributed computation models, the algorithms, and their analysis are quite different from the parallel setting studied here.

1.2 Contributions of this paper

In this paper, we extend the fail-stop model of [KS 89] by allowing arbitrary dynamic restarts of processors (with loss of private memory). We also consider a model in which private memory is safe, but the interactions of processors with each other through shared memory can no longer be assumed to be synchronous. Although the models differ in their formal definition, some algorithms work equally well in both models.

In the restartable fail-stop model, defined precisely in Section 2.1, PRAM processors are subject to on-line (dynamic) *failures* and *restarts*. We concentrate on the worst case analysis of the *completed work* of deterministic algorithms that are subject to arbitrary adversaries, and on the *overhead ratio*, which amortizes the work over the necessary work and failures/restarts. In this model, processors fail and then restart in a way that makes it possible to develop *terminating* algorithms, while relaxing the requirement that one processor must never fail (which was necessary in the fail-stop without restart model). To guarantee algorithm termination and sensible accounting of resources, we introduce an *update cycle*, that generalizes the standard PRAM read/compute/write cycle. In the absence of update cycles, a *thrashing* adversary exploiting the separation of *read* and *write* instructions in PRAMs can force quadratic work for any *Write-All* solution. The restartable PRAM model is defined in Section 2.1, which also contains a discussion of the technical choices made.

The strongly asynchronous model is defined in Section 2.2. In this model, we use Lamport's notion of *serializability* [Lam 86], which states that the effect of a parallel computation should be consistent with some serialization of atomic processor actions. We consider the serialization to be chosen by an on-line adversary, and use atomic reads and atomic writes (other primitives are considered as well). This model is related to other models known as asynchronous PRAMs [CZ 89, CZ 90, Gib 89, MSP 90, Nis 90]; perhaps the closest of these is [MSP 90], although this reference considers only off-line (pre-specified) adversaries and randomized algorithms. The relationship of the two models in Sections 2.1 and 2.2 is discussed in Section 2.3; some practical motivation is also discussed in Section 1.3 below.

In Section 3, we present lower bounds for the *Write-All* problem. When reads and writes are accounted together in update cycles of the restartable fail-stop model, the quadratic lower bound mentioned above no longer applies. Instead, we show that the *Write-All* problem of size N requires $N-P+\Omega(P\log P)$ completed work for $P \le N$. This bound also holds in the strongly asynchronous model. It holds when processors can read and locally process the entire shared memory at unit cost. (Note that an $\Omega(N\log N)$ lower bound when $P = N$ was recently shown in [KPRS 90] using a different technique and different assumptions for a fail-stop no-restart model.) We also demonstrate a lower bound of $N + \Omega(P\log N)$ (when $3 \le P \le N$) for the strongly asynchronous PRAM, when certain atomic primitives (such as compare-and-swap or test-and-set) are used to access shared memory.

In Section 4 we present three efficient algorithms for the *Write-All* problem. The first (algorithm V) is a modification of the algorithm developed in [KS 89] for the fail-stop no-restart model, and runs on the restartable fail-stop model with completed work $O(N + P\log^2 N + M\log N)$, where M is the number of failures. This algorithm is based on an analysis of the lower bounds in Section 3. The second (algorithm X) runs on both models in time $O(N \cdot P^{\log_2 \frac{3}{2}})$. The third (algorithm T) runs on both models in the case $P = 3$, using $N + O(\log N)$ compare-and-swap operations on the strongly asynchronous model and $N + O(\log N)$ update cycles in the fail-stop restart model. This matches the lower bound when three processors are used.

In Section 5, we show how to use algorithms V and X to simulate any N processor PRAM on a restartable fail-stop P processor CRCW PRAM. A terminating execution of each simulated N processor step is guaranteed with $O(\log^2 N)$ overhead ratio, and (sub-quadratic) completed work $O(\min\{N + P\log^2 N + M\log N,\ N \cdot P^{\log_2 \frac{3}{2}}\})$, where M is the number of failures during the simulation of the particular step. The strategy is work-optimal when the number of simulating processors is $P \leq N/\log^2 N$ and the total number of failures per each simulated step is $O(N/\log N)$.

The lower bounds presented in Section 3 apply to the worst-case work of deterministic algorithms and to the expected work of randomized and deterministic algorithms. Randomization does not seem to help, given on-line (non-prespecified) patterns of failures. For example, it is easy to construct on-line failure and restart (resp. no-restart) patterns that lead to exponential (resp. quadratic) in N expected performance for the algorithms presented in [MSP 90]. These *stalking* adversaries are described in Section 6, where we also conclude with some open problems.

Preliminary versions of this work were reported in [BR 90, KS 91].

1.3 Motivation and relation to physical systems

The models we present and study are intended to capture certain features of actual systems.

Processor delay is a feature of any multi-user environment, in which processing priorities are not specified by a single user. Processing time may be required at a moment's notice by another user or by the underlying operating system.

Processor failure may occur either because of a physical fault or because another entity in the system preempts processing time without saving the old state.

Communication delay is a well-known feature of multi-processor systems. Small communication delays are compatible with synchronization if the step time is sufficient for the longest possible access time, but synchronizing by counting up to the longest possible access time eliminates any advantages due to caching and similar techniques.

Communication failure may be due to memory operations of other processors. The interacting operations need not involve the same memory module. If the communication network reports the failure of an operation, the processor can re-attempt the access, and the situation can be modelled as a communication delay. If unannounced failures can occur, an algorithm must either explicitly check its write operations or ensure in some other way that omission of a write is not detrimental to performance.

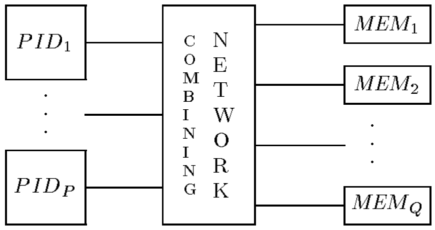

Figure 1: A robust fail-stop multiprocessor.

In this paper, we treat delay and/or failure as occurring to the processors only. If memory operations are atomic and serializable, they may be assumed to be instantaneous, and the communication delays or access failures may be charged to the processor.

An architecture for a restartable fail-stop multiprocessor: The main goal of this work is to study algorithmic techniques that enable efficient parallel computation on realizable multiprocessor systems. We now suggest one way of realizing the abstract model of computation where processors are subject to fail-stop errors and restarts, i.e., the model formalized in Section 2.1.

Engineering and technological approaches exist that allow implementing electronic components and systems that operate correctly when subjected to certain failures (for examples, see [IEEE 90, Cri 91]). The technologies cited in the next paragraph are instrumental in providing basic hardware fault-tolerance for a foundation on which the algorithmic and software fault-tolerance can be built.

Semiconductor memories are the essential components of shared memory parallel systems. Memories are routinely manufactured with built-in fault tolerance using replication and coding techniques without appreciably degrading performance (see the survey [SM 84]). Interconnection networks are typically used in a multiprocessor system to provide communication among processors, memory modules and other devices (e.g., as in the *Ultracomputer* [Sch 80]). The fault-tolerance of interconnection networks has been the subject of much work in its own turn. The networks are made more reliable by employing redundancy (see the survey [AAS 87]). A *combining* interconnection network that is perfectly suited for implementing synchronous concurrent reads and writes is formally treated in [KRS 88]. Finally, fail-stop processors are formally presented and justified in [SS 83].

The abstract model that we are studying can be realized (Figure 1) in the following architecture, using the components we have just discussed:

1. There are P *fail-stop* processors, each with a unique address and some amount of local memory. Processors are unreliable.
2. There are Q addressable shared memory cells. The input of size $N \leq Q$ is stored in shared memory. This memory is assumed to be reliable.
3. Interconnection of processors and memory is provided by a synchronous combining interconnection network. This network is assumed to be reliable.

With this architecture, our algorithmic techniques become completely applicable; i.e., the algorithms and simulations we develop will work correctly, and within the complexity bounds (under the unit cost memory access assumption) for all patterns of processor failures and restarts when the underlying components are subject to the failures within their respective design parameters.

2 Models of computation

2.1 The restartable fail-stop CRCW PRAM

We use as a basis the PRAM model [FW 78], where all concurrently writing processors write the same value (COMMON CRCW). Processors are subject to stop failures and restarts as in [SS 83]. Our algorithms are described using the **forall/parbegin/parend** parallel construct.

1. There are P synchronous processors. Each processor has a unique permanent identifier (PID) in the range $0, \ldots, P-1$, and each processor has access to P and its own PID.
2. The global memory accessible to all processors is denoted as **shared**; in addition, each processor has a constant size local memory denoted as **private**. All memory cells are capable of storing $O(\log \max\{N, P\})$ bits on inputs of size N.
3. The input is stored in N cells in shared memory, and the rest of the shared memory is cleared (i.e., contains zeroes). The processors have access to the input and its size N.

In all our algorithms:

- The PRAM processors execute sequences of instructions grouped in *update cycles*. Each update cycle consists of reading a small fixed number of shared memory cells (e.g., 4), performing some fixed time computation, and writing a small number of shared memory cells (e.g., 2).

The parameters of the update cycle, i.e., the number of read and write instructions, are fixed, but depend on the instruction set of the PRAM; see [FW 78] for a typical PRAM instruction set. The values quoted (4 and 2) are sufficient for our exposition. It is an interesting question whether smaller values would suffice to implement efficient algorithms.

We use the *fail-stop with restart* failure model, where time instances are the PRAM synchronous clock-ticks:

1. A failure pattern F (i.e., failures and restarts) is determined by an *on-line adversary*, that knows everything about the algorithm and is unknown to the algorithm.
2. Any processor may fail at any time during any update cycle, or having failed it may restart at any time, provided that:

   (i) at any time at least one processor is executing an update cycle that successfully completes;

   (ii) single bit writes are *atomic*, i.e., failures can occur before or after a write of a single bit.
3. Failures do not affect the shared memory, but the failed processors lose their private memory. Processors are restarted at their initial state with their PID as their only knowledge.

The failure and restart patterns are syntactically defined as follows:

Definition 2.1 A *failure pattern* F is a set of triples $<tag$, PID, $t>$ where *tag* is either **failure** indicating processor failure, or **restart** indicating a processor restart, PID is the processor identifier, and t is the time indicating when the processor stops or restarts. The *size* of the failure pattern F is defined as the cardinality $|F|$. □

For simplicity of presentation, we assume that the shared memory writes of $O(\log \max\{N, P\})$ bit words are atomic. Algorithms using this assumption can be easily converted to use only single bit atomic writes as in [KS 89].

We investigate two natural complexity measures, completed work and overhead ratio. The completed work measure generalizes the standard *Parallel-time* × *Processors* product and the available processor steps of [KS 89]. The overhead ratio is an amortized measure.

Definition 2.2 Consider an algorithm with P initial processors that terminates in parallel-time τ after completing its task on some input data I and in the presence of a failure pattern F. If $P_i(I, F) \leq P$ is the number of processors completing an update cycle at time i, and c is the time required to complete one update cycle, then we define $S(I, F, P)$ as:

$$S(I, F, P) = c\sum_{i=1}^{\tau} P_i(I, F). \quad \square$$

Update cycles are units of accounting. They do not constrain the instruction set of the PRAM, and failures can occur between the instructions of an update cycle. However, note that in $S(I, F, P)$ the processors are not charged for the read and write instructions of update cycles that are not completed.

Definition 2.3 A P-processor PRAM algorithm on any input data I of size $|I| = N$, and in the presence of any pattern F of failures and restarts of size $|F| \leq M$,

- uses *completed work* $S = S_{N,M,P} = \max_{I,F}\{S(I, F, P)\}$, and
- has *overhead ratio* $\sigma = \sigma_{N,M,P} = \max_{I,F}\left\{\dfrac{S(I, F, P)}{|I| + |F|}\right\}$. □

Consider a definition of *total work* $S'(I, F, P)$ that also counts incomplete update cycles. Clearly $S'(I, F, P) \leq S(I, F, P) + c|F|$. Thus, using S' does asymptotically affect the measure of work (when $|F|$ is very large), but it does not asymptotically affect σ.

One might also generalize the overhead ratio as $\frac{S(I,F,P)}{T(|I|)+|F|}$, where $T(|I|)$ is the time complexity of the best sequential solution known to date for the particular problem at hand. For the purposes of this exposition, it is sufficient to express σ in terms of the ratio $\frac{S(I,F,P)}{|I|+|F|}$. This is because for *Write-All* (by itself and as used in the simulation) $T(|I|) = \Theta(|I|)$.

Now let us briefly comment on the technical choices made in Definitions 2.2 and 2.3.

Work vs. overhead ratio: When dealing with arbitrary processor failures and restarts, the completed work measure S (or total work measure S') depends on the size N of the input I, the number of processors P, and the size of failure pattern F. The ultimate performance goal for a parallel fault-tolerant algorithm is to be able to perform the required computation at a work cost as close as possible to the work performed by the best sequential algorithm known. Unfortunately, this goal is not attainable when an adversary succeeds in causing too many processor failures during a computation.

Example A: Consider a *Write-All* solution, where it takes a processor one instruction to recover from a failure. If an adversary in a failure pattern F with the number of failures and restarts $|F| = \Omega(N^{1+\varepsilon})$ for $\varepsilon > 0$, then the completed work will be $\Omega(N^{1+\varepsilon})$, and thus already non-optimal and potentially large, regardless of how efficient the algorithm is otherwise. Yet the algorithm may be extremely efficient, since it takes only one instruction to handle a failure. □

This illustrates the need for a measure of efficiency that is sensitive to both the size of the input N, and the number of failures and restarts $M = |F|$. When $M = O(P)$ as in the case of the stop failures without restarts in [KS 89], S properly describes the algorithm efficiency, and $\sigma = O(\frac{S_{N,M,P}}{N})$. However, when F can be large relative to N and P (as is the case when restarts are allowed) σ better reflects the efficiency of a fault-tolerant algorithm. Recall that σ is insensitive to the choice of S or S', and to using update cycles, as a measure of work. However, update cycles are necessary for the following two reasons.

Update cycles and termination: Our failure model requires that at any time, at least one processor is executing an update cycle that completes. (This condition subsumes the condition of [KS 89] that one processor does not fail during the computation). This requirement is formulated in terms of update cycles and assures that some progress is made. Since the processors lose their context after a failure, they have to read something to regain it. Without at least one active update cycle completing, the adversary can force the PRAM to thrash by allowing only these reads to be performed. Similar concerns are discussed in [SS 83].

Update cycles as a unit of accounting: In our definition of completed work we only count completed update cycles. Even if the progress and termination of a computation is assured (by always completely executing at least one update cycle), but the processors are charged for incomplete update cycles, the work S' of any algorithm that simulates a single N processor PRAM step is at least $\Omega(P \cdot N)$. The reason for this quadratic behavior in S' is the following simple and rather uninteresting *thrashing* adversary.

Example B: We evaluate the work of any solution for the *Write-All* problem under the arbitrary failure and restart model. Consider the standard PRAM read-compute-write cycle (if processors begin writing without reading, a simple modification of the argument leads to the same result). A *thrashing* adversary allows all processors to perform the read and compute instructions; then it fails all but one processor for the write operation. Failed processors are then restarted. Since one write operation is performed per cycle, N cycles will be required to initialize N array elements. Each of the P processors performs $\Theta(N)$ instructions which results in work of $\Theta(P \cdot N)$. □

By charging the processors only for the completed fixed size update cycles we do not charge for thrashing adversaries. This change in cost measure allows sub-quadratic solutions.

2.2 The strongly asynchronous PRAM

The strongly asynchronous PRAM model departs from the standard PRAM models in that the processors are completely asynchronous. The only synchronizing assumption is that reads and writes to memory are atomic and serializable, in the sense of Lamport [Lam 86]. Serializability means that the result of a computation is consistent with some total ordering of atomic actions. (Note that this does not mean that the actions are in fact ordered this way, but that the effect of the computation is as if they were.) This is a restriction on the possible outcome of simultaneous events. With asynchronous processors, the distinction between exclusive writes and concurrent writes disappears. Among the traditional synchronous PRAM models, the ARBITRARY CRCW PRAM is closest to the strongly asynchronous model.

One important situation that is modelled by the strongly asynchronous PRAM is the case in which the processors are "nearly synchronous." If identical processors access shared memory across a common communication channel or network, then they will run at approximately the same speed, but the precise interleaving of memory operations may not be under the direct control of the processors. To model the lack of control over the interleaving, we posit an on-line adversary that chooses the interleaving to maximize the cost of the computation. The adversary is free to delay any processor for any length of time.

Definition 2.4 We define an *interleaving* to be a sequence of processor numbers, each in the range $[0, P-1]$. An *execution* of a PRAM algorithm consistent with a particular interleaving is the execution of steps by the processors in the order specified by the interleaving. □

Definition 2.5 The measure of the efficiency of a strongly asynchronous PRAM is the total number of steps completed, which we term the *total work* of the computation (expressed in terms of P and the input size N). To define total work, we assume that each processor executes a halt instruction when it terminates work on the algorithm. In order for the algorithm to be correct, it must be the case that at this point, the postconditions for the algorithm are satisfied. The total work of an algorithm with respect to a given interleaving is the length of the smallest halt-free prefix of that interleaving. The total work required by an algorithm is then the maximum total work over all possible interleavings of the processors. (Note that in this worst case, all processors will be ready to execute halt instructions.)□

Previous work along these lines has assumed either that randomized algorithms can be used to defeat off-line adversaries ([MSP 90]) or that interleavings are chosen according to some probabilistic distribution ([CZ 90, Nis 90]). Some of the models in these last two papers are similar to our restartable fail-stop model, but failures are probabilistic and restarts do not destroy private memory. Because of our worst-case assumptions, these analyses are inappropriate. Furthermore, notions of time used in [CZ 90] do not work here, because our scheduling adversary may introduce arbitrarily long delays.

The notion of *wait-free* asynchronous computation, in which any one processor terminates in a finite number of steps regardless of the speeds of the other processors, is introduced in [Her 88]. In the strongly asynchronous PRAM, by definition any algorithm with bounded work must be wait-free. The same paper shows that atomic reads and writes are insufficient to solve two-processor consensus, and demonstrates a hierarchy of stronger primitives for accessing memory (such as test-and-set or compare-and-swap). A later paper ([AH 90]) demonstrates wait-free data structures using only atomic reads and writes.

Finally, we note that the strongly asynchronous model is a very general one, and it is subject to fewer definitional restrictions than is its fail-stop restartable counterpart. However, as a result of such restrictions, the fail-stop model can be used for general synchronous PRAM simulations (as we show in Section 5), while the strongly asynchronous model cannot be used for such simulations due to impossibility results such as [Her 88].

2.3 Comparison of the models

On the surface, the two models of restartable fail-stop processors and of asynchronous processors are designed for quite different situations. The fail-stop model treats failure as an abnormal event, which occurs with sufficient frequency that it cannot be ignored. The asynchronous model treats delay as a normal occurrence. Nevertheless, the two models are closely related.

Consider an execution of an asynchronous algorithm. Because the events are serializable, we may assume without loss of generality that the events occur at discrete times. In other words, a set of time slices is fixed in advance, and the scheduling adversary chooses at each time slice whether or not each processor will start running during that time slice. From this viewpoint, the two models differ in the following ways.

1. Processors that miss a time slice lose their internal state in the restartable fail-stop case, and keep their internal state in the asynchronous case.
2. The adversary can stop a processor after any memory operation within a time slice in the restartable fail-stop case while this has no effect on the asynchronous case.
3. The time slices are long enough for several memory operations in the restartable fail-stop case but allow only a single operation in the asynchronous case.

From the algorithmic point of view, the difference between the models concerns the number of failures during an execution of the algorithm. In the restartable fail-stop model, failure is treated as a significant event, and the number of failures may be taken into account when measuring the efficiency of the algorithm. In the asynchronous model, delay is the rule rather than the exception, and the number of delays is not a particularly meaningful quantity. A normal execution may involve many delays of each processor between each consecutive step.

An algorithm that performs a bounded amount of work for any number of failures, and has a small amount of state information, is suitable for either model. An algorithm whose performance degrades significantly as the number of failures increases, however, may only be suitable for the restartable fail-stop model. Algorithms W and V (as presented in Section 4) are examples of the latter case; algorithms X and T exemplify the former case.

3 Lower bounds for the *Write-All* problem

3.1 Lower bounds with memory snapshots

As we have shown in Example B in Section 2.1, without the update cycle accounting there is a thrashing adversary that exhibits a quadratic lower bound for the *Write-All* problem in the restartable fail-stop model. With the update cycle accounting and for the asynchronous model, we show $N - P + \Omega(P \log P)$ work lower bounds (when $P \leq N$) for both models, even when the processors can take unit time *memory snapshots*, i.e., processors can read and locally process the entire shared memory at unit cost.

Theorem 3.1 Given any P-processor CRCW PRAM algorithm that solves the *Write-All* problem of size N ($P \leq N$), an adversary (that can cause arbitrary processor failures and restarts) can force the algorithm to perform $N - P + \Omega(P \log P)$ completed work steps.

Proof: Let Z be any algorithm for the *Write-All* problem subject to arbitrary failure/restarts using update cycles. Consider each PRAM cycle. The adversary uses the following strategy:

Let $U > 1$ be the number of unvisited array elements. For as long as $U > P$, the adversary induces no failures. The work needed to visit $N - P$ array elements when there were no failures is at least $N - P$.

As soon as a processor is about to visit the element $N - P + 1$ making $U \leq P$, the adversary fails and then restarts all N processors. For the upcoming cycle, the adversary determines how the algorithm assigns processors to write to array elements. By an averaging argument, for any processor assignment to the U elements, there is a set of $\lfloor \frac{U}{2} \rfloor$ unvisited elements with no more than $\lceil \frac{P}{2} \rceil$ processors assigned to them. The adversary fails these processors, allowing all others to proceed. Therefore at least $\lfloor \frac{P}{2} \rfloor$ processors will complete this step having visited no more than half of the remaining unvisited array locations.

This strategy can be continued for at least $\log P$ iterations. The work performed by the algorithm will be $S \geq N - P + \lfloor \frac{P}{2} \rfloor \log P = N - P + \Omega(P \log P)$. $\square$

Note that the bound holds even if processors are only charged for writes into the array of size N and do not have to only write the value 1. The simplicity of this strategy ensures that the results hold in the strongly asynchronous model.

Theorem 3.2 Any N-processor strongly asynchronous PRAM algorithm that solves the *Write-All* problem of size N has total work $N - P + \Omega(P \log P)$.

Proof: Any possible execution of an algorithm on the restartable fail-stop model can be duplicated by an appropriate interleaving on the strongly asynchronous model. The argument in Theorem 3.1 works even if failed processors do not lose local state, and so the same strategy will work in the strongly asynchronous model. $\square$

This lower bound is the tightest possible bound under the assumption that the processors can read and locally process the entire shared memory at unit cost. Although such an assumption is very strong, we present the matching upper bound for two reasons. First, it demonstrates that any improvement to the lower bound must take account of the fact that processors can read only a constant number of cells per update cycle. Second, it presents a simple processor allocation strategy that we use to advantage in the next section.

Theorem 3.3 If processors can read and locally process the entire shared memory at unit cost, then a solution for the *Write-All* problem in the restartable fail-stop model can be constructed such that its completed work using P processors on input of size N is $S = N - P + O(P \log P)$, when $P \leq N$.

Proof: The processors follow the following simple strategy: at each step that a processor PID is active, it reads the N elements of the array $x[1..N]$ to be visited. Say U of these elements are still not visited. The processor numbers these U elements from 1 to U based on their position in the array, and assigns itself to the ith unvisited element such that $i = \lceil PID \cdot \frac{U}{P} \rceil$. This achieves load balancing with no more than $\lceil \frac{P}{U} \rceil$ processors assigned to each unvisited element. The reading and local processing is done as a snapshot at unit cost.

We list the elements of the *Write-All* array in ascending order according to the time at which the elements are visited (ties are broken arbitrarily). We divide this list into adjacent segments numbered sequentially starting with 0, such that the segment 0 contains $V_0 = N - P$ elements, and segment $j \geq 1$ contains $V_j = \lfloor \frac{P}{j(j+1)} \rfloor$ elements, for $j = 1, ..., m$ and for some $m \leq \sqrt{P}$. Let U_j be the least possible number of unvisited elements when processors were being assigned to the elements of the jth segment. U_j can be computed as $U_j = N - \sum_{i=0}^{j-1} V_i$. U_0 is of course N, and for $j \geq 1$, $U_j = P - \sum_{i=1}^{j-1} V_i \geq P - (P - \frac{P}{j}) = \frac{P}{j}$. Therefore no more than $\lceil \frac{P}{U_j} \rceil$ processors were assigned to each element.

The work performed by such an algorithm is:

$$S \leq \sum_{j=0}^{m} V_j \left\lceil \frac{P}{U_j} \right\rceil \leq V_0 + \sum_{j=1}^{m} \left\lfloor \frac{P}{j(j+1)} \right\rfloor \left\lceil \frac{P}{P/j} \right\rceil = V_0 + O\left(P \sum_{j=1}^{m} \frac{1}{j+1} \right) = N - P + O(P \log P) \ . \ \Box$$

A similar situation holds in the strongly asynchronous model.

Theorem 3.4 If processors can read and locally process the entire shared memory at unit cost, then a solution for the *Write-All* problem in the strongly asynchronous model can be constructed with total work $N - P + O(P \log P)$ using P processors on input of size N, for $P \leq N$.

Proof: We use the same algorithm as in the previous proof. The proof itself applies to the strongly asynchronous model with the following modifications: (1) one unit of total work is charged for each read and the write that (potentially) follows; (2) as soon as a processor performs a read, it is charged one unit work; this is done to take care of the situation when a processor performs a write only after all elements in a given segment have been initialized. □

3.2 Lower bounds with test-and-set operations

Under certain assumptions on the way that memory is accessed in the strongly asynchronous model, we can prove a different lower bound. Assume for the moment that, instead of atomic reads and writes, memory is accessed by means of *test-and-set* operations. That is, memory can only contain zeroes and ones, and a single test-and-set operation on a memory cell sets the value of that cell to 1 and returns the old value of the cell. (We will discuss shortly how this assumption can be generalized.)

Theorem 3.5 Any strongly asynchronous PRAM algorithm for the *Write-All* problem which uses test-and-set as an atomic operation requires $N + \Omega(P \log(N/P))$ total work, for $P \geq 3$.

Proof: Consider the following class of interleavings. A *round* will be a length of time in which processors take one step each in PID order; formally, it is the sequence of PIDs $\langle 1, 2, \ldots P\rangle$. We will run the algorithm in phases. To define a phase, suppose that U cells out of the original N remain unset at the beginning of a phase. We imagine running the algorithm in rounds until a *collision* occurs; that is, until a test-and-set operation is done on a cell that is already set to one. Suppose this happens in the tth round. The actual definition of the phase depends on the nature of the collision; there are two cases.

If the cell involved in the collision was set in this round, then it was initially set by some processor with PID i, and set again by some processor with PID j. Then to define the phase, we let only processors i and j alternate steps, instead of running all processors; that is, the phase consists of the PIDs i, j repeated t times. A total of $2t$ steps are taken and one of them is wasted work.

On the other hand, if the cell was set in a previous round, then consider the processor with PID j that set it in this round and let only this processor take steps. That is, the phase consists of the PID j repeated t times, for a total of t steps and one wasted step.

We now note that t must be at most $\lceil U/P\rceil$, and so a recurrence for the amount of wasted work $W(U)$ is $W(U) \geq 1 + W(U - 2\lceil U/P\rceil + 1)$. By induction, we can show that $W(U) \geq cP\ln(U/2P)$ for a suitable constant $c > 0$; the result follows by noting that unwasted work N is necessary.

The trivial base case of the induction is $U \leq 2P$. Now suppose that the inequality $W(x) \geq cP\ln(x/2P)$ holds for all integer $x < U$. By the induction hypothesis, we have $W(U) \geq cP\ln((U - 2\lceil U/P\rceil + 1)/2P) \geq 1 + cP\ln(U/2P) + cP\ln(1 - 2/P - 1/U)$. It thus suffices to prove $1 + cP\ln(1 - 2/P - 1/U) \geq 0$. But

$$1 + cP\ln(1 - 2/P - 1/U) \geq 1 + cP\ln(1 - 5/(2P)) \geq 1 + cP(-5/(2P - 5)) \geq 0.$$

The first inequality is valid because $U > 2P$; the second inequality uses $\ln(1 - z) \geq -z/(1 - z)$, which can be seen by comparing power series; the third inequality is valid for $P \geq 3$ and any choice of $c \leq 1/15$. No attempt was made to optimize the constant c. $\square$

The argument used in this lower bound can be applied equally well if the atomic operation is compare-and-swap, or to any set of atomic read-modify-write operations where the read and writes are constrained to be to the same cells. It also applies to atomic read and atomic write, but in this case there is no known matching upper bound, whereas algorithm T (presented in the next section) can match the lower bound (for some choices of atomic operation) in the case $P = 3$. The above proof technique also applies to the fail-stop restartable model, when each update cycle accesses only one array element used by the *Write-All* problem.

4 Algorithms for the *Write-All* problem

The original motivation for studying the *Write-All* problem was that it intuitively captured the essential nature of a single synchronous PRAM step. This intuition was made concrete when it was shown ([KPS 90, Shv 89]) how to use any algorithm for the *Write-All* problem in general PRAM simulations. This application is discussed in the next section; in this section, we will present new algorithms for the *Write-All* problem.

In what follows, we assume that the number of array elements N and the number of processors P are powers of 2. Nonpowers of 2 can be handled using conventional padding techniques. All logarithms are base 2.

4.1 Algorithm V: a modification of a no-restart algorithm

Algorithm W of [KS 89] is an efficient fail-stop (no restart) *Write-All* solution. The algorithm uses two full binary trees as its basic data structures (the processor counting and the progress measurement trees). The algorithm uses an iterative approach in which all active processors synchronously execute the following four phases:

W1: Processors are counted and enumerated using a static bottom-up, logarithmic time traversal of the processor counting tree data structure.

W2: Processors are allocated to the unvisited array locations according to a divide-and-conquer strategy using a dynamic top-down traversal of the progress tree data structure.

W3: Array assignments are done.

W4: Progress is evaluated by a dynamic bottom-up traversal of the progress tree data structure.

This algorithm has efficient completed work when subjected to arbitrary failure patterns without restarts. It can be extended to handle processor restarts by introducing an iteration counter, and having the revived processors wait for the start of a new iteration. However, this algorithm may not terminate if the adversary does not allow any of the processors that were alive at the beginning of an iteration to complete that iteration. Even if the extended algorithm were to terminate, its completed work is not bounded by a function of N and P.

In addition, the proof framework of [KS 89] does not easily extend to include processor restarts: the processor enumeration and allocation phases become inefficient and possibly incorrect, since no accurate estimates of active processors can be obtained when the adversary can revive any of the failed processors at any time.

On the other hand, the second phase of algorithm W can implement processor assignment (in a manner similar to that used in the proof of Theorem 3.3) in $O(\log N)$ time by using the permanent processor PID in the top-down divide-and-conquer allocation. This also suggests that *the processor enumeration phase of algorithm W does not improve its efficiency when processors can be restarted.*

Therefore we present a modified version of algorithm W, that we call V. To avoid a complete restatement of the details of algorithm V, the reader is urged to refer to [KS 89].

V uses the data structures of the optimized algorithm W of [KS 89] (i.e., full binary trees with $\frac{N}{\log N}$ leaves) for progress estimation and processor allocation. There are $\log N$ array elements associated with each leaf. When using P processor such that $P > \frac{N}{\log N}$ on such data structures, it is sufficient for each processor to take its PID modulo $\frac{N}{\log N}$ to assure that there is a uniform initial assignment of at least $\lfloor P/\frac{N}{\log N}\rfloor$ and no more than $\lceil P/\frac{N}{\log N}\rceil$ processors to a work element.

Algorithm V is an iterative algorithm using the following three phases.

V1: Allocate processors using PIDs in a dynamic top-down traversal of the progress tree to assure load balancing ($O(\log N)$ time).

V2: The processors now perform work at the leaves they reached in phase V1 (there are $\log N$ array elements per leaf).

V3: The processors begin at the leaves of the progress tree where they ended phase V2 and update the progress tree dynamically, bottom up ($O(\log N)$ time).

Processor re-synchronization after a failure and a restart is an important implementation detail. One way of realizing processor re-synchronization is through the utilization of an iteration wrap-around counter that is based on the synchronous PRAM clock. If a processor fails, and then is restarted, it waits for the counter wrap-around to rejoin the computation. The point at which the counter wraps around depends on the length of the program code, but it is fixed at "compile time".

Analysis of algorithm V:

We now analyze the performance of this algorithm first in the fail-stop, and then in the fail-stop and restart setting.

Lemma 4.1 The completed work of algorithm V using $P \leq N$ processors that are subject to fail-stop errors without restarts is $S = O(N + P\log^2 N)$.

Proof: We factor out any work that is wasted due to failures by charging this work to the failures. Since the failures are fail-stop, there can be at most P failures, and each processor that fails can waste at most $O(\log N)$ steps corresponding to a single iteration of the algorithm. Therefore the work charged to the failures is $O(P\log N)$, and it will be absorbed by the rest of the work.

We next evaluate the work that directly contributes to the progress of the algorithm by distinguishing two cases below. In each of the cases, it takes $O(\log \frac{N}{\log N}) = O(\log N)$ time to perform processor allocation, and $O(\log N)$ time to perform the work at the leaves. Thus each iteration of the algorithm takes $O(\log N)$ time. We use the allocation technique of Theorem 3.3, where instead of reading and locally processing the entire memory at unit cost, we use an $O(\log N)$ time iteration for processor allocation.

Case 1: $1 \leq P < \frac{N}{\log N}$. In this case, at most 1 processor is initially allocated to each leaf. As in the proof of Theorem 3.3, when the first $\frac{N}{\log N} - P$ leaves are visited, there is no more than one processor allocated to each leaf by the balanced allocation phase. When the remaining P or less leaves are visited, the work is $O(P\log P)$ by Theorem 3.3 (not counting processor allocation). Each leaf visit takes $O(\log N)$ work steps; therefore the completed work is:

$$S = O\left(\left(\frac{N}{\log N} - P + P\log P\right)\cdot \log N\right) = O(N + P\ \log P\ \log N) = O(N + P\log^2 N).$$

Case 2: $\frac{N}{\log N} \leq P \leq N$. In this case, no more than $\lceil P/\frac{N}{\log N}\rceil$ processors are initially allocated to each leaf. Any two processors that are initially allocated to the same leaf, should they both survive, will behave identically throughout the computation. Therefore we can use Theorem 3.3 with the $\lceil P/\frac{N}{\log N}\rceil$ processor allocation as a multiplicative factor. From this, the completed work is:

$$S = \left\lceil P/\frac{N}{\log N}\right\rceil \cdot O\left(\frac{N}{\log N}\log\frac{N}{\log N}\right) \cdot O(\log N) = O(P\log^2 N).$$

The results of the two cases combine to yield $S = O(N + P\log^2 N)$. $\square$

The following theorem expresses the completed work of the algorithm in the presence of restarts:

Theorem 4.2 The completed work of algorithm V using $P \leq N$ processors subject to an arbitrary failure and restart pattern F of size M is: $S = O(N + P\log^2 N + M\log N)$.

Proof: The proof of Lemma 4.1 does not rely on the fact that in the absence of restarts, the number of active processors is non-increasing. However, the lemma does not account for the work that might be performed by processors that are active during a part of an iteration but do not contribute to the progress of the algorithm due to failures. To account for all work, we are going to charge to the array being processed the work that contributes to progress, and any work that was wasted due to failures will be charged to the failures and restarts. Lemma 4.1 accounts for the work charged to the array. Otherwise, we observe that a processor can waste no more than $O(\log N)$ time steps without contributing to the progress due to a failure and/or a restart. Therefore this amount of wasted work is bounded by $O(M\log N)$. This proves the theorem. (Note that the completed work S of V is small for small $|F|$, but it is not bounded by a function of P and N for large $|F|$). $\square$

4.2 Algorithm X: a binary tree algorithm

We present a new algorithm X for the *Write-All* problem, and show that its completed/total work complexity is $S = O(N \cdot P^{\log\frac{3}{2}})$ using $P \leq N$ processors in the restartable fail-stop and the strongly asynchronous models of computation. The important property of X is that it has bounded sub-quadratic completed work; in the restartable fail-stop model, this is independent of the failure pattern. If a very large number of failures occurs, say $|F| = \Omega(N \cdot P^{0.59})$, then the algorithm's overhead ratio σ becomes optimal: it takes a fixed number of computing steps per failure/recovery.

Like algorithm V, algorithm X utilizes a progress tree of size N, but it is traversed by the processors independently, not in synchronized phases. This reflects the local nature of the processor assignment in algorithm X as opposed to the global assignments used in algorithms V and W. Each processor, acting independently, searches for work in the smallest immediate subtree that has work that needs to be done. It then performs the necessary work, and moves out of that subtree when no more work remains. We present the algorithm on the restartable fail-stop model.

Input: Shared array $x[1..N]$; $x[i] = 0$ for $1 \leq i \leq N$.

Output: Shared array $x[1..N]$; $x[i] = 1$ for $1 \leq i \leq N$.

Data-structures: The algorithm uses a full binary tree of size $2N-1$, stored as a *heap* $d[1\ldots 2N-1]$ in shared memory. An internal tree node $d[i]$ $(i = 1, \ldots, N-1)$ has the left child $d[2i]$ and the

```
01  forall processors PID=0..P − 1 parbegin
02      Perform initial processor assignment to the leaves of the progress tree
03      while there is still work left in the tree do
04          if current subtree is done then move one level up
05          elseif this is a leaf then perform the work at the leaf
06          elseif this is an interior tree node then
07              if both subtrees are done then update the tree node
08              elseif only one is done then go to the one that is not done
09              else move to the left/right subtree according to PID bit values
10              fi
11          fi
12      od
13  parend
```

Figure 2: A high level view of the algorithm X.

right child $d[2i+1]$. The tree is used for progress evaluation and processor allocation. The values stored in the heap are initially 0.

The N elements of the input array $x[1 \ldots N]$ are associated with the leaves of the tree. Element $x[i]$ is associated with $d[i+N-1]$, where $1 \leq i \leq N$. The algorithm also utilizes an array $w[0..P-1]$ that is used to store individual processor locations within the progress tree d.

Each processor uses some constant amount of private memory to perform simple arithmetic computations. An important private constant is PID, containing the initial processor identifier.

Thus, the overall memory used is $O(N+P)$ and the data-structures are simple.

Control-flow: The algorithm consists of a single initialization and of the parallel *loop*. A high level view of the algorithm is in Figure 2; all line numbers refer to this figure. More detailed code can be found in Appendix A.

The initialization (line 02) assigns the P processors to the leaves of the progress tree so that the processors are assigned to the first P leaves by storing the initial leaf assignment in w[PID]. The *loop* (lines 03-12) consists of a multi-way decision (lines 04-11). If the current node is marked done, the processor moves up the tree (line 04). If the processor is at a leaf, it performs work (line 05). If the current node is an unmarked interior node and both of its subtrees are done, the interior node is marked by changing its value from 0 to 1 (line 07). If a single subtree is not done, the processor moves down appropriately (line 08).

For the final case (line 09), the processors move down when neither child is done. This last case is where a non-trivial (*italicized*) decision is made. The PID of the processor is used at depth h of the tree node based on the value of the h^{th} most significant bit of the binary representation of the PID: bit 0 will send the processor to the left, and bit 1 to the right.

Regardless of the decision made by a processor within the *loop* body, each iteration of the body consists of no more than four shared memory reads, a fixed time computation using private memory, and one shared memory write (see Appendix A for the detailed algorithm). Therefore the body can be implemented as an update cycle.

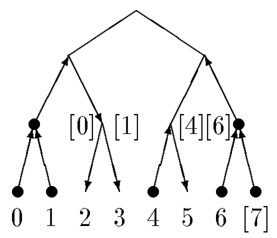

Figure 3: Processor traversal of the progress tree.

Example C: Consider algorithm X for $N = P = 8$. The progress tree d of size $2N - 1 = 15$ is used to represent the full binary progress tree with eight leaves. The 8 processors have PIDs in the range 0 through 7. Their initial positions are indicated in Figure 3 under the leaves of the tree. The diagram illustrates the state of a computation where the processors were subject to some failures and restarts. Heavy dots indicate nodes whose subtrees are finished. The paths being traversed by the processors are indicated by the arrows. Active processor locations (at the time when the snapshot was taken) are indicated by their PIDs in brackets. In this configuration, should the active processors complete the next cycle, they will move in the directions indicated by the arrows: processors 0 and 1 will descend to the left and right respectively, processor 4 will move to the unvisited leaf to its right, and processors 6 and 7 will move up. □

Analysis of algorithm X:

We begin by showing the correctness and termination of algorithm X in the following simple lemma.

Lemma 4.3 Algorithm X with N processors is a correct, terminating and fault-tolerant solution for the P-processor *Write-All* problem of size N. The algorithm terminates in at least $\Omega(\log N)$ and at most $O(P \cdot N)$ time steps.

Proof: We first observe that the processor loads are localized in the sense that a processor exhausts all work in the vicinity of its original position in the tree, before moving to other areas of the tree. If a processor moves up out of a subtree then all the leaves in that subtree were visited. We also observe that it takes exactly one update cycle to: (i) change the value of a progress tree node from 0 to 1, (ii) to move up from a (non root) node, or (iii) to move down left, or (iv) down right from a (non leaf) node. Therefore, given any node of the progress tree and any processor, the processor will visit and spend exactly one complete update cycle at the node no more than four times.

Since there are $2N - 1$ nodes in the progress tree, any processor will be able to execute no more than $O(N)$ completed update cycles. If there are P processors, then all processors will be able to complete no more than $O(P \cdot N)$ update cycles. Furthermore, at any point in time, there is at least one update cycle that will complete. Therefore it will take no more than $O(P \cdot N)$ sequential update cycles of constant size for the algorithm to terminate.

Finally, we also observe that all paths from a leaf to the root are at least $\log N$ long, therefore at least $\log N$ update cycles per processor will be required for the algorithm to terminate. □

Now we prove the main work lemma. In the rest of this section, the expression "$S_{N,P}$" denotes the completed work on inputs of size N using P initial processors and for *any* failure pattern. Note that in this lemma we assume $P \geq N$.

Lemma 4.4 The completed work of algorithm X for the *Write-All* problem of size N with $P \geq N$ initial processors and for any pattern of failures and restarts is $S_{N,P} = O(P \cdot N^{\log \frac{3}{2}})$.

Proof: We show by induction on the height of the progress tree that there are positive constants c_1, c_2, c_3 such that $S_{N,P} \leq c_1 P \cdot N^{\log \frac{3}{2}} - c_2 P \log N - c_3 P$.

For the base case: we have a tree of height 0 that corresponds to an input array of size 1 and at least as many initial processors P. Since at least one processor, and at most P processors will be active, this single leaf will be visited in a constant number of steps. Let the work expended be $c'P$ for some constant c' that depends only on the lexical structure of the algorithm. Therefore $S_{1,P} = c'P \leq c_1 P \cdot 1^{\log \frac{3}{2}} - c_2 P \cdot 0 - c_3 P$ when c_1 is chosen to be larger than or equal to $c_3 + c'$.

Now consider a tree of height $\log N$ (≥ 1). The root has two subtrees (left and right) of height $\log N - 1$. By the definition of algorithm X, no processor will leave a subtree until the subtree is *marked-one*, i.e., the value of the root of the subtree is changed from 0 to 1. We consider the following sub-cases: (1) both subtrees are marked-one simultaneously, and (2) one of the subtrees is marked-one before the other.

Case 1: If both subtrees are marked-one simultaneously, then the algorithm will terminate after the two independent subtrees terminate plus some small constant number of steps c' (when a processor moves to the root and determines that both of the subtrees are finished). Both the work S_L expended in the left subtree of, and the work S_R in the right subtree are bounded by $S_{N/2,P/2}$. The added work needed for the algorithm to terminate is at most $c'P$, and so the total work is:

$$S \leq S_L + S_R + c'P \leq 2S_{N/2,P/2} + c'P \leq 2\left(c_1 \frac{P}{2}\left(\frac{N}{2}\right)^{\log \frac{3}{2}} - c_2 \frac{P}{2} \log \frac{N}{2} - c_3 \frac{P}{2}\right) + c'P$$

$$= c_1 \frac{2}{3} P N^{\log \frac{3}{2}} - c_2 P \log \frac{N}{2} - c_3 P + c'P \leq c_1 P \cdot N^{\log \frac{3}{2}} - c_2 P \log N - c_3 P$$

for sufficiently large c_1 and any c_2 depending on c', e.g., $c_1 \geq 3(c_2 + c')$.

Case 2: Assume without loss of generality that the left subtree is marked-one first with $S_L = S_{N/2,P/2}$ work being expended in this subtree. Any active processors from the left subtree will start moving via the root to the right subtree. The path traversed by any processor as it moves to the right subtree after the left subtree is finished is bounded by the maximum path length from a leaf to another leaf $c' \log N$ for a predefined constant c'. No more than the original $P/2$ processors of the left subtree will move, and so the work of moving the processors is bounded by $c'(P/2) \log N$.

We observe that the cost of an execution in which P processors begin at the leaves of a tree (with $N/2$ leaves) differs from the cost of an execution where $P/2$ processors start at the leaves, and $P/2$ arrive at a later time via the root, by no more than the cost $c'(P/2) \log N$ accounted for above. This can be simply shown by constructing a scenario in which the second set of $P/2$ processors do not arrive through the root, but instead start their execution with a failure, and then traverse along a path of 1's (if any) in the progress tree, until they reach a 0 node that is either a leaf, or whose descendants are marked. Having accounted for the difference, we see that the work S_R to complete the right subtree using up to P processors is bounded by $S_{N/2,P}$ (by the definition of S, if $P_1 \le P_2$, then $S_{N,P_1} \le S_{N,P_2}$). After this, each processor will spend some constant number of steps moving to the root and terminating the algorithm. This work is bounded by $c''P$ for some small constant c''. The total work S is:

$$S \le S_L + c'\frac{P}{2}\log N + S_R + c''P \le S_{N/2,P/2} + c'\frac{P}{2}\log N + S_{N/2,P} + c''P$$

$$\le c_1\frac{P}{2}\left(\frac{N}{2}\right)^{\log\frac{3}{2}} - c_2\frac{P}{2}\log\frac{N}{2} - c_3\frac{P}{2} + c_1P\left(\frac{N}{2}\right)^{\log\frac{3}{2}} - c_2P\log\frac{N}{2} - c_3\frac{P}{2} + c''P$$

$$= c_1PN^{\log\frac{3}{2}} - c_2P\log N\left(\frac{3}{2} - \frac{c'}{2c_2}\right) - c_3P\left(\frac{3}{2} - \frac{c''}{c_3} - \frac{3c_2}{2c_3}\right) \le c_1P\cdot N^{\log\frac{3}{2}} - c_2P\log N - c_3P$$

for sufficiently large c_2 and c_3 depending on fixed c' and c'', e.g., $c_2 \ge c'$ and $c_3 \ge 3c_2 + 2c''$.

Since the constants c', c'' depend only on the lexical structure of the algorithm, the constants c_1, c_2, c_3 can always be chosen sufficiently large to satisfy the base case and both the cases (1) and (2) of the inductive step. This completes the proof of the lemma. □

The quantity $P \cdot N^{\log\frac{3}{2}}$ is about $P \cdot N^{0.59}$. We next show a particular pattern of failures for which the completed work of algorithm X matches this upper bound.

Lemma 4.5 There exists a pattern of fail-stop/restart errors that cause the algorithm X to perform $S = \Omega(N^{\log 3})$ work on the input of size N using $P = N$ processors.

Proof: We can compute the exact work performed by the algorithm when the adversary adheres to the following strategy:

(a) The processor with PID 0 will be allowed to sequentially traverse the progress tree in post-order starting at the leftmost leaf and finishing at the rightmost leaf.

(b) The processors that find themselves at the same leaf as processor 0 are (re)started and are allowed to traverse the progress tree until they reach a leaf, where they are failed.

(c) Procedure (b) is repeated until all leaves are visited.

Thus the leaves of the progress tree are visited left to right, from the leaf number 1 to the leaf number N. At any time, if i is the number of the rightmost visited leaf, then only the processors with PIDs 0 to $i-1$ have performed at least one update cycle thus far.

The cost of such strategy can be expressed inductively as follows:

The cost C_1 of traversing a tree of size 1 using a single processor is 1 (unit of completed work).

The cost C_{i+1} of traversing a tree of size 2^{i+1} is computed as follows: first, there is the cost C_i of traversing the left subtree of size 2^i. Then, all processors move to the right subtree and participate (subject to failures) in the traversal of the right subtree at the cost of $2C_i$ — the cost is doubled, because the two processors whose PIDs are equal modulo i behave identically. Thus $C_{i+1} = 3C_i$, and $C_{\log N} = 3^{\log N} = N^{\log 3}$. □

Now we show how to use algorithm X with P processors to solve *Write-All* problems of size N such that $P \le N$. Given an array of size N, we break the N elements of the input into $\frac{N}{P}$ groups of P elements each (the last group may have fewer than P elements). The P processors are then used to solve $\frac{N}{P}$ *Write-All* problems of size P one at a time. We call this algorithm X', and we will use X' in the general simulations.

Remark: Strictly speaking, it is not necessary to modify algorithm X for $P \leq N$ processors. Algorithm X can be used with $P \leq N$ processors by initially assigning the P processors to the first P elements of the array to be visited. It can also be shown that X and X' have the same asymptotic complexity; however, the analysis of X' is very simple, as we show below.

Theorem 4.6 Algorithm X' with P processors solves the *Write-All* problem of size N $(P \leq N)$ using completed work $S = O(N \cdot P^{\log \frac{3}{2}})$. In addition, there is an adversary that forces algorithm X' to perform $S = \Omega(N \cdot P^{\log \frac{3}{2}})$ work.

Proof: By Lemma 4.4, $S_{P,P} = O(P \cdot P^{\log \frac{3}{2}}) = O(P^{\log 3})$. Thus the overall work will be $S = O(\frac{N}{P} S_{P,P}) = O(\frac{N}{P} P^{\log 3}) = O(N \cdot P^{\log \frac{3}{2}})$.

Using the strategy of Lemma 4.5, an adversary causes the algorithm to perform work $S_{P,P} = \Omega(P^{\log 3})$ on each of the $\frac{N}{P}$ segments of the input array. This results in the overall work of: $S = \Omega(\frac{N}{P} P^{\log 3}) = \Omega(N \cdot P^{\log \frac{3}{2}})$. □

Remark: Lemma 4.3 gives only a loose upper bound for the worst time performance of algorithm X — there we are primarily concerned with termination. The actual worst case time for algorithm X can be no more than the upper bound on the completed work. This is because at any point in time there is at least one update cycle that will complete. Therefore, for algorithm X' with $P \leq N$, the time is bounded by $O(N \cdot P^{\log \frac{3}{2}})$. In particular, for $P = N$, the time is bounded by $O(N^{\log 3})$. In fact, using the worst case strategy of Lemma 4.5, an adversary can "time share" the completed cycles of the processors so only one processor is active at any given time, with the processor with PID 0 being one step ahead of other processors. The resulting time is then $\Omega(N^{\log 3})$.

In algorithm X, processors work for the most part independently of other processors; they attempt to avoid duplicating already-completed work but do not co-ordinate their actions with other processors. It is this property which allows the algorithm to run on the strongly asynchronous model with the same work and time bounds.

Lemma 4.7 Algorithm X with P processors solves the *Write-All* problem of size N $(P \geq N)$ on the strongly asynchronous model with total work $O(P \cdot N^{\log \frac{3}{2}})$.

Proof: If we let $S_{N,P}$ be the total work done by algorithm X on a problem of size N with P processors, then $S_{N,P}$ satisfies the same recurrence as given in the proof of Lemma 4.4. The proof, which never uses synchroneity, goes through exactly as in that lemma, except that case 1 (where left and right subtrees have their roots marked simultaneously) does not occur. □

The final result of this section is similar to Theorem 4.6:

Theorem 4.8 Algorithm X' with P processors solves the *Write-All* problem of size N $(P \leq N)$ on the strongly asynchronous model with total work $O(N \cdot P^{\log \frac{3}{2}})$.

4.3 Algorithm T: a three-processor algorithm

Quite different techniques are necessary when designing a parallel algorithm in which the number of processors is much smaller than the size of the input. The goal in this situation, when the underlying machine is synchronous, is to find a method whose parallel time complexity is at most the sequential time complexity divided by the number of processors plus a small *additive* overhead; see [And 90] for an example of such an algorithm. Note that constant factors are important and cannot be hidden in O-notation. When considering algorithms on fail-stop or asynchronous models, the goal is to have the parallel work complexity be equal to the sequential complexity plus small overhead.

For the *Write-All* problem, it is easy to achieve this goal with two processors. The processor with PID 0 (henceforth, P_0) reads and then writes locations sequentially starting at 1 and moving up; processor P_1 reads and then writes locations sequentially starting at N and moving down. Both processors stop when they read a 1. The completed work is exactly $N+1$.

The first non-trivial case is that of three processors. Here is an intuitive description of an algorithm that works in this situation. Processor P_0 works left-to-right, processor P_1 works right-to-left, and P_2 fills starting from the middle and alternately expanding in both directions. If P_0 and P_2 meet, they both know that an entire prefix of the memory cells has been written. Processor P_0 then jumps to the leftmost cell not written by itself or P_2, and P_2 jumps to the new "middle" of unwritten cells. A meeting of P_1 and P_2 is symmetric. When P_0 and P_1 meet, the computation is complete. Intuitively, processors can maintain an upper bound on the number of empty cells remaining that starts at N and is halved every time a collision occurs. Thus at most $\log N$ collisions are experienced by each processor. High-level pseudo-code for the algorithm is given in figure 4.

Implementation of the high-level algorithm requires some form of communication among the asynchronous processors. At a collision, a processor must determine which processor previously wrote the cell. In the case of a collision with P_2, a processor must also determine what portion of the array to jump over. This communication may be implemented either by writing additional information to the cells of the array or by using auxiliary variables.

If the array in which processors are writing is also used to hold auxiliary information, implementation is straightforward. When processor P_2 writes to a cell at the left (resp. right) end of its area, it writes the location of the next unwritten cell to the right (resp. left). P_0 and P_1 write the values -1 and $N+1$ respectively, to signal no unwritten cells. A total of $N+O(\log N)$ reads and $N+O(\log N)$ writes are required on the asynchronous model. If an atomic compare-and-swap is used, the total work is reduced to $N+O(\log N)$ operations.

To solve the pure *Write-All* problem, in which only 1's are written to the array, auxiliary shared variables are required. These variables must be carefully managed to ensure that the processors maintain a consistent view of the progress of the algorithm. Because a processor may be delayed between reading an auxiliary variable and writing to the array, complete consistency is impossible. Approximate consistency is sufficient, however, if the processors are appropriately pessimistic. The precise code is presented and analyzed in Appendix B.

In summary, algorithm T provides the following bounds.

Theorem 4.9 The *Write-All* problem for three processors can be solved with $N+O(\log N)$ writes to and $N+O(\log N)$ reads from the array.

T_0:
    set current position to 1
    repeat
        if no collision **then** write 1 and increment current position
        else if collision with P_1 **then** exit
        else (collision with P_2) set current position at the right of P_2's area
        fi
    until current position $> N$.

T_1:
    set current position to N
    repeat
        if no collision **then** write 1 and decrement current position
        else if collision with P_0 **then** exit
        else (collision with P_2) set current position at the left of P_2's area
        fi
    until current position < 1.

T_2:
    initialize middle and boundaries of current write area
    repeat
        if no collision **then** write the next 2 cells away from the middle
        else if collision with P_0 **then**
            set left boundary at rightmost cell written by P_2
            set middle halfway between left and right boundaries
        else (collision with P_1)
            set right boundary at leftmost cell written by P_2
            set middle halfway between left and right boundaries
        fi
    until done

Figure 4: A high-level description of algorithm T. Processor P_i executes T_i.

In most applications, the array also has room for communication variables, and no auxiliary variables are necessary.

5 General simulations on restartable fail-stop processors

We now present a major extension to the algorithms presented so far. This is an efficient deterministic simulation of any N-processor synchronous PRAM on P restartable fail-stop processors ($P \leq N$). Note that due to the impossibility results for asynchronous models [Her 88], we are able to show this result only for the restartable fail-stop model.

We first formally state the main result and then discuss its proof.

Theorem 5.1 Any N-processor PRAM algorithm can be executed on a restartable fail-stop P-processor CRCW PRAM, with $P \leq N$. Each N-processor PRAM step is executed in the presence of any pattern F of *failures and restarts* of size M with:

- completed work: $S = O(\min\{N + P\log^2 N + M\log N,\ \ N \cdot P^{\log\frac{3}{2}}\})$,
- overhead ratio: $\sigma = O(\log^2 N)$.

EREW, CREW, and WEAK and COMMON CRCW PRAM algorithms are simulated on fail-stop COMMON CRCW PRAMs; ARBITRARY and STRONG CRCW PRAMs are simulated on fail-stop CRCW PRAMs of the same type. □

Remark: PRIORITY CRCW PRAMs cannot be directly simulated using the same framework, for one of the algorithms used (namely algorithm X in Section 4) does not possess the *processor allocation monotonicity* property that assures that higher numbered processors simulate the steps of the higher numbered original processors.

An approach for executing arbitrary PRAM programs on fail-stop CRCW PRAMs (without restart) was presented independently in [KPS 90] and [Shv 89]. The execution is based on simulating individual PRAM computation steps using the *Write-All* paradigm. It was shown that the complexity of solving a N-size instance of the *Write-All* problem using P fail-stop processors is equal to the complexity of executing a single N-processor PRAM step on a fail-stop P-processor PRAM. Here we describe how algorithms V and X' are combined with the framework of [KPS 90] or [Shv 89] to yield efficient executions of PRAM programs on PRAMs that are subject to stop-failures and restarts as stated in Theorem 5.1.

Theorem 5.2 There exists a *Write-All* solution using $P \leq N$ processors on instances of size N such that for any pattern F of failures and restarts with $|F| \leq M$, the completed work is $S = O(\min\{N + P\log^2 N + M\log N,\ \ N \cdot P^{\log\frac{3}{2}}\})$, and the overhead ratio is $\sigma = O(\log^2 N)$.

Proof: The executions of algorithms V and X' can be interleaved to yield an algorithm that achieves the performance as stated. The completed work complexity is asymptotically equal to the minimum of the completed work performed by V and X'. This is because the number of cycles performed by each algorithm in the interleaving differs by at most a multiplicative constant. The overhead ratio is directly inherited from algorithm V by the same reasoning because of the Definition 2.3 of σ and S. □

The simulations of the individual PRAM steps are based on replacing the trivial array assignments in a *Write-All* solution with the appropriate components of the PRAM steps. These steps are decomposed into a fixed number of assignments corresponding to the standard *fetch/decode/execute* RAM instruction cycles in which the data words are moved between the shared memory and the internal processor registers. The resulting algorithm is then used to interpret the individual cycles using the available fail-stop processors and to ensure that the results of computations are stored in temporary memory before simulating the synchronous updates of the shared memory with the new values. For the details on this technique, the reader is referred to [KS 89, KPS 90, Shv 89]. Application of these techniques in conjunction with the algorithms V and X' yield efficient and terminating executions of any non-fault-tolerant PRAM programs in the presence of arbitrary failure and restart patterns. Theorem 5.1 follows from Theorem 5.2 and the results of [KPS 90] or [Shv 89]. The following corollaries are also interesting:

Corollary 5.3 Under the hypothesis of Theorem 5.1, and if $|F| \leq P \leq N$, then:
$S = O(N + P\log^2 N)$, and $\sigma = O(\log^2 N)$.

The fail-stop (without restarts) behavior of the combined algorithm is subsumed by this corollary. The exact analysis of algorithm V without restarts is still unknown. Without restarts, [KPRS 90] have an algorithm with $S = O(N + P\frac{\log^2 N}{\log\log N})$, and [Mar 91] has shown that the same performance is achieved by algorithm W from [KS 89].

Corollary 5.4 Under the hypothesis of Theorem 5.1:

(i) when $|F|$ is $\Omega(N\log N)$, then σ is $O(\log N)$,

(ii) when $|F|$ is $\Omega(N^{1.59})$, then σ is $O(1)$.

Thus the overhead efficiency of our algorithm actually improves for large failure patterns. These results also suggest that it is harder to deal efficiently with a few worst case failures than with a large number of failures.

Our next corollary demonstrates a non-trivial range of parameters for which the completed work is optimal; i.e., the work performed in executing a parallel algorithm on a faulty PRAM is asymptotically equal to the *Parallel-time*$\times$*Processors* product for that algorithm.

Corollary 5.5 Any N-processor, τ-time PRAM algorithm can be executed on a $P \leq N/\log^2 N$ processor fail-stop CRCW PRAM, such that when during the execution of each N-processor step of that algorithm the total number of processor failures and restarts is $O(N/\log N)$, then the completed work is $S = O(\tau \cdot N)$.

6 Discussion and Open Problems

We conclude with a brief discussion of open problems and the effects of on-line adversaries on the expected performance of randomized algorithms.

Lower bounds: We have shown an $\Omega(N\log N)$ lower bound (when $N = P$) for the *Write-All* problem in both the restartable fail-stop and the strongly asynchronous models under the assumption that processors can read and locally process the entire shared memory at unit cost. Under this assumption, these are the best possible lower bounds.

Under the same assumption, it can be shown that the $\Omega(N\log N/\log\log N)$ lower bound of [KS 89] is the best possible bound for failures without restarts. This is done by adapting the analysis of algorithm W by [Mar 91]. According to the analysis, the number of "block-steps" of W for $P = N$ is $O(N\log N/\log\log N)$ and each block-step can be realized at unit cost each, under the above assumptions.

Under different assumptions, an $\Omega(N \log N)$ lower bound is shown for failures without restarts in [KPRS 90].

Can these lower bounds be further improved? Can the lower bound of $N + \Omega(P \log N)$ be proved for the restartable fail-stop model, or improved for the strongly asynchronous model with atomic reads and writes?

Upper bounds: Is $O(N \log^{O(1)} N)$ completed/total work for solving *Write-All* with N processors and input of size N achievable in the restartable fail-stop/strongly asynchronous model? Recently, an existence proof for an algorithm achieving $O(N^{1+\epsilon})$ work was given in [AW 91].

What is the worst-case completed work S, and overhead ratio σ of the algorithm X in the fail-stop (without restart) framework of [KS 89]? Algorithm X appears to perform well in this context. For example, the adversary used to show the lower bound in [KS 89] causes completed work $S = \Theta(N \log^2 N / \log\log N)$ for the N-processor *Write-All* solution in [KS 89]. The same adversary causes algorithm X to do completed work $S = \Theta(N \log N \log\log N / \log\log\log N)$. We *conjecture* that the fail-stop (no restart) performance of X has $S = O(N \log N \log\log N)$ using N processors.

Can algorithm T be generalized to work with more than three processors, or can another (more general) algorithm be found that achieves truly optimal speedup for small numbers of processors?

Model issues: What is the minimum number of reads and writes necessary in an update cycle to ensure efficient algorithms? What is the precise relationship between the complexity of problems (as opposed to algorithms) on the two models presented here? Finally, are there efficient algorithms for important problems that do not come from simulation of synchronous PRAM algorithms?

On randomization and lower bounds: Analyses of randomized solutions for *Write-All* have so far considered only *off-line* (non-adaptive) adversaries. In contrast, the lower bounds of Section 3 apply to both the worst case performance of deterministic algorithms and the expected performance of randomized algorithms subject to on-line adversaries.

A randomized *asynchronous coupon clipping* (ACC) algorithm for the *Write-All* problem was analyzed in [MSP 90]. Assuming off-line adversaries, it was shown in [MSP 90] that ACC algorithm performs expected $O(N)$ work using $P = N/(\log N \log^* N)$ processors on inputs of size N.

In the on-line case, we observe that a simple *stalking* adversary causes the ACC algorithm to perform (expected) work of $\Omega(N^2/\operatorname{polylog} N)$ in the case of fail-stop errors, and $\Omega\Big(\big(\frac{N}{\operatorname{polylog} N}\big)^{\frac{N}{\operatorname{polylog} N}}\Big)$ work in the case of fail-stop errors with restart even when using $P \leq \frac{N}{\operatorname{polylog} N}$ processors. The stalking adversary strategy consists of choosing a single leaf in a binary tree employed by ACC, and failing all processors that touch that leaf until only one processor remains in the fail-stop case, or until all processors simultaneously touch the leaf in the fail-stop/restart case. This performance is not improved even when using the completed work accounting. On a positive note, when the adversary is made off-line, the ACC algorithm becomes efficient in the fail-stop/restart setting.

Acknowledgements:

We would like to thank Jeff Vitter, Marc Snir, and Naomi Nishimura for helpful discussions, and Franco Preparata for reviewing an earlier draft of this paper.

References

[AAS 87] G. B. Adams III, D. P. Agrawal, H. J. Seigel, "A Survey and Comparison of Fault-tolerant Multistage Interconnection Networks", *IEEE Computer*, 20, 6, pp. 14-29, 1987.

[AAPS 87] Y. Afek, B. Awerbuch, S. Plotkin, M. Saks, "Local Management of a Global Resource in a Communication Network", *Proc. of the 28th IEEE Symposium on Foundations of Computer Science*, pp. 347-357, 1987.

[And 90] R. Anderson, "Parallel Algorithms for Generating Random Permutations on a Shared Memory Machine", *Proc. of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pp. 95-102, 1990.

[AW 91] R. Anderson and H. Woll, "Wait-Free Parallel Algorithms for the Union-Find Problem", *Proc. of the 23rd ACM Symposium on Theory of Computing*, pp. 370-380, 1991.

[AH 90] J. Aspnes and M. Herlihy, "Wait-Free Data Structures in the Asynchronous PRAM Model", *Proc. of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pp. 340-349, 1990.

[AU 90] S. Assaf and E. Upfal, "Fault Tolerant Sorting Network," in *Proc. of the 31st IEEE Symposium on Foundations of Computer Science*, pp. 275-284, 1990.

[BR 90] J. Buss and P. Ragde, "Certified Write-All on a Strongly Asynchronous PRAM", manuscript, 1990.

[CZ 89] R. Cole and O. Zajicek, "The APRAM: Incorporating Asynchrony into the PRAM Model," in *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 170-178, 1989.

[CZ 90] R. Cole and O. Zajicek, "The Expected Advantage of Asynchrony," in *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pp. 85-94, 1990.

[Cri 91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", in *Communications of the ACM*, vol. 3, no. 2, pp. 56-78, 1991.

[DPPU 86] C. Dwork, D. Peleg, N. Pippenger, E. Upfal, "Fault Tolerance in Networks of Bounded Degree", *in Proc. of the 18th ACM Symposium on Theory of Computing*, pp. 370-379, 1986.

[EG 88] D. Eppstein and Z. Galil, "Parallel Techniques for Combinatorial Computation", *Annual Computer Science Review*, 3 (1988), pp. 233-83.

[FW 78] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines", *Proc. the 10th ACM Symposium on Theory of Computing*, pp. 114-118, 1978.

[Gib 89] P. Gibbons, "A More Practical PRAM Model," in *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 158-168, 1989.

[Her 88] M. P. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization", in *Proc. of the 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[HP 89] S. W. Hornick and F. P. Preparata, "Deterministic P-RAM: Simulation with Constant Redundancy," in *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures.*, pp. 103-109, 1989.

[IEEE 90] *IEEE Computer*, "Fault-Tolerant Systems", (special issue), vol. 23, no. 7, 1990.

[K* 90] C. Kaklamanis, A. Karlin, F. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, A. Tsantilas, "Asymptotically Tight Bounds for Computing with Arrays of Processors," in *Proc. of the 31st IEEE Symposium on Foundations of Computer Science*, pp. 285-296, 1990.

[KS 89] P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms Can Be Made Robust", Brown Univ. Tech. Report CS-89-35 (revision pending for *Distributed Computing*); preliminary version appears in *Proc. of the 8th ACM Symposium on Principles of Distributed Computing*, pp. 211-222, 1989.

[KS 91] P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms On Restartable Fail-Stop Processors", to appear in *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, 1991.

[KR 90] R. M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines", in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), North-Holland, 1990.

[KPS 90] Z. M. Kedem, K. V. Palem, and P. Spirakis, "Efficient Robust Parallel Computations," in *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 138-148, 1990.

[KPRS 90] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis, "Combining Tentative and Definite Executions for Dependable Parallel Computing," Univ. of Maryland, Tech. Report UMIACS-TR-90-122, CS-TR-2537, 1990 (to appear in *Proc 23d ACM Symposium on Theory of Computing*).

[KRS 88] C. P. Kruskal, L. Rudolph, M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 4, pp. 579-601 1988.

[Lam 86] L. Lamport, "On Interprocess Communication", *Distributed Computing*, 1 (1986), pp. 77-101.

[LGFG 86] N.A. Lynch, N.D. Griffeth, M.J. Fischer, L.J. Guibas, "Probabilistic Analysis of a Network Resource Allocation Algorithm", *Information and Control*, vol. 68, pp. 47-85, 1986.

[MSP 90] C. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," in *Proc. 32d IEEE Symposium on Foundations of Computer Science*, pp. 590-599, 1990. Also see Tech. Rep. CSE-89-6, Univ. of Calif.-Davis, 1989.

[Mar 91] C. Martel, personal communication, March, 1991.

[Nis 90] N. Nishimura, "Asynchronous Shared Memory Parallel Computation," in *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 76-84, 1990.

[Pip 85] N. Pippenger, "On Networks of Noisy Gates", *Proc. of 26th IEEE Symposium on Foundations of Computer Science*, pp. 30-38, 1985.

[Ran 87] A. Ranade, "How to Emulate Shared Memory", *Proc. of 28th IEEE Symposium on Foundations of Computer Science*, pp. 185-194, 1987.

[Rud 85] L. Rudolph, "A Robust Sorting Network", *IEEE Trans. on Comp.*, vol. 34, no. 4, pp. 326-335, 1985.

[SM 84] D. B. Sarrazin and M. Malek, "Fault-Tolerant Semiconductor Memories", *IEEE Computer*, vol. 17, no. 8, pp. 49-56, 1984.

[SS 83] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: an Approach to Designing Fault-tolerant Computing Systems", *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 222-238, 1983.

[Sch 80] J. T. Schwartz, "Ultracomputers", *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 4, pp. 484-521, 1980.

[Shv 89] A. A. Shvartsman, "Achieving Optimal CRCW PRAM Fault-Tolerance", *Tech. Report CS-89-49*, Brown University, 1989 (to appear in *Information Processing Letters*).

[Upf 89] E. Upfal, "An O(log N) Deterministic Packet Routing Scheme," in *Proc. 21st ACM Symposium on Theory of Computing*, pp. 241-250, 1989.

[Val 90a] L. Valiant, "General Purpose Parallel Architectures," in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), North-Holland, 1990.

[Val 90b] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103-111, 1990.

```
forall processors PID=0..P − 1 parbegin
    shared x[1..N];                 −−shared memory
    shared d[1..2N-1];              −−"done" heap (progress tree)
    shared w[0..P-1];               −−"where" array
    private done, where;            −−current node done/where
    private left, right;            −−left/right child values
    action,recovery
        w[PID] := 1 + PID;  −−the initial positions
    end ;
    action,recovery
        while w[PID] ≠ 0 do  −−while haven't exited the tree
            where := w[PID];  −−current heap location
            done := d[where];  −−doneness of this subtree
            if done then w[PID] := where div 2;  −−move up one level
            elseif not done ∧ where ≥ N − 1 then  −−at a leaf
                if x[where−N] = 0 then x[where−N] := 1;  −−initialize leaf
                elseif x[where−N] = 1 then d[where] := 1;  −−indicate "done"
                fi
            elseif not done ∧ where < N − 1 then  −−interior tree node
                left := d[2∗where]; right := d[2∗where+1];  −−read left/right child values
                if left ∧ right then d[where] := 1;  −−both children done
                elseif not left ∧ right then w[PID] := 2∗where;  −−go left
                elseif left ∧ not right then w[PID] := 2∗where+1;  −−go right
                elseif not left ∧ not right then  −−both subtrees are not done
                  −−move down according to the PID bit
                    if not PID[log(where)] then w[PID] := 2∗where;  −−move left
                    elseif PID[log(where)] then w[PID] := 2∗where+1;  −−move right
                    fi
                fi
            fi
        od
    end
parend .
```

Figure 5: Algorithm X.

A Algorithm X pseudocode

Here we give detailed pseudocode for algorithm X on the restartable fail-stop model.

In the pseudocode, the **action, recovery end** construct of [SS 83] is used to denote the actions and the recovery procedures for the processors. In the algorithm this signifies that an action is also its own recovery action, should a processor fail at any point within the action block.

The notation "PID[log(k)]" is used to denote the binary true/false value of the $\lfloor \log(k) \rfloor$-th bit of the $\log(N)$-bit representation of PID, where the most significant bit is the bit number 0, and the least significant bit is bit number $\log N$. Finally, **div** stands for integer division with truncation.

The action/recovery construct can be implemented by appropriately checkpointing the instruction counter in stable storage as the last instruction of an action, and reading the instruction counter upon a restart. This is amenable to automatic implementation by a compiler.

```
T0:

shared I0 := 1;
shared I1;
private temp0;
shared x[1..N];

repeat
    --Invariant: x[k] = 1 for all k < I0
    if x[I0] = 0 then
        x[I0] := 1;
        I0 := I0 + 1;
    elseif I0 ≥ I1 or I0 ≥ Right2 then
        --Collision with P1
        I0 := N + 1;
    else
        --Collision with P2
        temp0 := Mid2;
        if I0 < Left2 then
            --Left2 has been updated
            I0 := Left2
        else
            --The correct Mid2 was read
            I0 := max{2 ⋆ temp0 − I0, I0 + 1}
        fi
    fi
until I0 ≥ N + 1;
```

```
T1:

shared I1 := N;
shared I0;
private temp1;
shared x[1..N];

repeat
    --Invariant: x[k] = 1 for all k > I1
    if x[I1] = 0 then
        x[I1] := 1;
        I1 := I1 − 1;
    elseif I1 ≤ I0 or I1 ≤ Left2 then
        --Collision with P0
        I1 := 0;
    else
        --Collision with P2
        temp1 := Mid2;
        if I1 > Right2 then
            --Right2 has been updated
            I1 := Right2
        else
            --The correct Mid2 was read
            I1 := min{2 ⋆ temp1 − I1, I1 − 1}
        fi
    fi
until I1 ≤ 0;
```

Figure 6: Algorithms T_0 and T_1

It is possible to perform local optimization of the algorithm by: (i) evenly spacing the P processors N/P leaves apart by when $P < N$, and by (ii) using the integer values at the progress tree nodes to represent the known number of descendent leaves visited by the algorithm. Our worst case analysis does not benefit from these modifications.

The algorithm can be used to solve *Write-All* "in place" using the array x[] as a tree of height $\log(N/2)$ with the leaves x[N/2..N-1], and doubling up the processors at the leaves, and using x[N] as the final element to be initialized and used as the algorithm termination sentinel. With this modification, array d[] is not needed. The asymptotic efficiency of the algorithm is not affected.

B Algorithm T pseudocode

The code for algorithm T in Figures 6 and 7 is given in three parts, one for each of the three processors (algorithm T_i for processor P_i). The code given is designed for easy proof of correctness, rather than optimality.

```
T₂:
    shared Left2 := 1;  −−left boundary of current write area
    shared Right2 := N;  −−right boundary of current write area
    shared Mid2 := ⌈N/2⌉;  −−middle of current write area
    shared I₀, I₁;
    shared x[1..N];
    private i := 0;  −−number of writes in current area

    repeat
        −−Invariant: At all times, x[k] = 1 for all values of k that satisfy
        −−1 ≤ k < Left2 or Mid2 − i < k < Mid2 + i or Right2 < k ≤ N
        case (x[Mid2 − i], x[Mid2 + i]) is
        (0,0):  −−Continue writing in current area
            x[Mid2 − i] := 1;
            x[Mid2 + i] := 1;
            i := i + 1;
        (1,0):  −−jump to the right
            jumpright;
        (0,1):  −−jump to the left
            jumpleft;
        (1,1):
            i := i + 1
            if I₀ ≥ mid then jumpright else jumpleft fi
        esac
    until Left2 ≥ Right2 or Mid2 − i < Left

procedure jumpright:
    Left2 := Mid2 + i;
    i := 0;
    Mid2 := ⌈(Left2 + Right2)/2⌉;
end

procedure jumpleft:
    Right2 := Mid2 − i;
    i := 0;
    Mid2 := ⌈(Left2 + Right2)/2⌉;
end
```

Figure 7: Algorithm T_2

T_0 and T_1 terminate because I_0 increases and I_1 decreases with every loop iteration. T_2 terminates because every loop iteration either increases i or decreases *Right2* − *Left2*. Since any execution of algorithm T is equivalent to some serialized execution, the following lemma implies that all cells of the array x are 1 at termination.

Lemma B.1 Every serialized execution of algorithm T maintains the following invariants.

1. For all k such that $1 \leq k < I_0$, cell k contains 1.

2. For all k such that $I_1 < k \leq N$, cell k contains 1.
3. For all k such that $1 \leq k < \mathit{Left2}$, cell k contains 1.
4. For all k such that $\mathit{Right2} < k \leq N$, cell k contains 1.
5. For all k such that $\mathit{Mid2} - i < k < \mathit{Mid2} + i$, cell k contains 1.

If some cell k has value 1, then at least one of the following holds.

6. Cell k was written by P_0 at a time when I_0 had the value k, or
7. Cell k was written by P_1 at a time when I_1 had the value k, or
8. Cell k was written by P_2 at a time when the values of *Mid2* and i satisfied $k = \mathit{Mid2} \pm i$.

Proof: Inspection of the code reveals that the consecutive values of I_0 and of *Left2* are nondecreasing, and the values of I_1 and of *Right2* are nonincreasing. Also, no processor writes to the same cell twice, and 0 is never written.

The invariants are vacuous at the start of the algorithm. It is necessary and sufficient to show that every operation preserves the invariants. The last three are trivial.

The assignments $I_0 := I_0 + 1$, $I_0 := N + 1$ and $I_0 := \mathit{Left2}$ preserve the invariants because of the comparisons preceding their execution and the monotonicity properties. The assignment $I_0 := 2 \star \mathit{temp0} - I_0$ is executed only after cell I_0 has been found to have been written by P_2 only. The variable *temp0* holds a value of *Mid2* that was valid at some time after the write and before *Left2* was increased by a subsequent execution of procedure jumpright. If P_2 had not yet jumped, conditions 8 and 5 imply the preservation of condition 1. Otherwise, P_2 jumped to the left because of a collision with P_1, and the entire array has been written, satisfying all of the invariants.

The case of assignments to I_1 is symmetrical.

The assignment $\mathit{Left2} := \mathit{Mid2} + i$ is executed only after P_0 has written to cell $\mathit{Mid2} - i$, and hence conditions 1, 5 and 6 imply preservation of condition 3. Similarly, $\mathit{Right2} := \mathit{Mid2} - i$ is executed only after P_1 has written to cell $\mathit{Mid2} + i$, and hence conditions 2, 5 and 7 imply preservation of condition 4. □

To prove the desired work bound, we use the following definition of a collision between processors.

Definition B.1 P_0 collides with P_j ($j \in \{1, 2\}$) if P_0 executes the code block labelled "collision with P_j." P_1 collides with P_j ($j \in \{0, 2\}$) if P_1 executes the code block labelled "collision with P_j." P_2 collides with P_0 if P_2 executes procedure jumpright. P_2 collides with P_1 if P_2 executes procedure jumpleft.

A redundant write does not imply that the writing processors collide with one another. Nevertheless, the number of collisions is a bound on the number of redundant writes.

Lemma B.2 Suppose two processors both write to cell k. Then one (or both) of the processors will collide in its next loop iteration.

Proof: One of the two processors must be P_0 or P_1. If it is P_0, then the other will next attempt to write to cell $k-1$ and collide. If it is P_1, then the other will next attempt to write to cell $k+1$ and collide. (In either case, the collision may involve the third processor.) $\Box$

Lemma B.3 There are $O(\log N)$ collisions.

Proof: When P_2 jumps, the quantity *Right2* $-$ *Left2* decreases by a factor of at least 2. Hence P_2 collides at most $\log N$ times. Also, P_0 can collide with P_1, and P_1 with P_0, at most once each.

Suppose P_0 collides with P_2 in attempting to write to cell k. Because P_0 did not collide with P_1, P_2 wrote to cell k with some value m in *Mid2* and the value $m-k$ in i. If P_2 continues to process, it will collide with either P_0 or P_1 after at most two iterations, when the value of i has become $m-k+2$. (The worst case occurs if P_0 and P_2 both write cell $k-1$.) Hence the only cells that P_2 writes with m in *Mid2* are in the interval $[k-1, 2m-k+1]$. Thus P_0 attempts to write at most four cells in the interval (i.e., cells $k-1$, k, $2m-k$ and $2m-k+1$), and can collide only at the latter three. Therefore, the number of collisions of P_0 with P_2 is at most three times the number of collisions of P_2.

Similarly, the number of collisions of P_1 with P_2 is at most three times the number of collisions of P_2. Hence the total number of collisions in $O(\log N)$, as required. $\Box$

Each collision involves only a constant number of memory accesses. Thus the algorithm satisfies the required work bounds.

Theorem B.4 Algorithm T solves the *Write-All* problem for three processors using $N+O(\log N)$ writes to and $N+O(\log N)$ reads from the array. There are at most $N+O(\log N)$ writes and $O(\log N)$ reads involving auxiliary variables.

Proof: The result follows directly from the above discussion. $\Box$

If the cells of array x can hold arbitrary integer values, then the information communicated by the values of the shared auxiliary variables can be stored directly in the array. Processors P_0 and P_1 write -1 and -2 respectively. Processor P_2 writes the value *Mid2* $+\, i$ when writing to the left of *Mid2* and the value *Mid2* $-\, i$ when writing to the right of *Mid2*. In this case, only private local variables are required.