To appear in the Proceedings of the 10th ACM Symposium on the Principles of Distributed Computing, Montreal, Canada, August, 1991.

(2)

Efficient Parallel Algorithms on Restartable Fail-stop Processors

Paris C. Kanellakis* Alex A. Shvartsman†

Abstract

We study efficient deterministic executions of parallel algorithms on restartable fail-stop CRCW PRAMs. We allow the PRAM processors to be subject to *arbitrary stop failures* and *restarts*, that are determined by an on-line adversary, and that result in loss of private memory but do not affect shared memory. For this model, we define and justify the complexity measures of: *completed work*, where processors are charged for completed fixed-size update cycles, and *overhead ratio*, which amortizes the work over necessary work and failures. This framework is a nontrivial extension of the fail-stop no-restart model of [KS 89].

We present 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 $O(\min\{N + P\log^2 N + M\log N,\ N \cdot P^{0.6}\})$ (sub-quadratic) completed work, where M is the number of failures during this step's simulation. This 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 N processor step is $O(N/\log N)$. These results are based on a new algorithm for the *Write-All* problem "P processors write 1's in an array of size N", together with a modification of the main algorithm of [KS 89] and with the techniques in [KPS 90, Shv 89].

We observe that, on $P = N$ restartable fail-stop processors, the *Write-All* problem requires $\Omega(N\log N)$ completed work, and this lower bound holds even under the additional assumption that processors can read and locally process the entire shared memory at unit cost. Under this unrealistic assumption we have a matching upper bound. The lower bound also applies to the expected completed work of randomized algorithms that are subject to on-line adversaries. Finally, we desribe a simple on-line adversary that causes inefficiency in many randomized algorithms.

1 Introduction

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) global synchronization, (2) simultaneous access across a significant bandwidth to a shared resource (memory), and (3) that processors, memory and their interconnection must be perfectly reliable. 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 examined in [CZ 89, CZ 90, Gib 89, MSP 90, Nis 90]; this research on synchronization is related to the study of parallel reliable computations, which is the subject of this paper.

Here, we continue and extend the study of fault tolerance that was initiated in [KS 89] and show that arbitrary PRAM algorithms can be efficiently and deterministically executed on restartable fail-stop PRAMs (whose processors are subject to arbitrary dynamic patterns of failures and restarts). As it was shown in [KS 89], 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 fur-

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

†Computer Science Dept., Brown University, PO Box 1910, Providence, RI 02912, USA, and Digital Equipment Corporation, LKG2-2/T2, 550 King Street, Littleton, MA 01460, USA. aas@cs.brown.edu.

ther actions).

It was determined that efficient and fault-tolerant solutions for a certain basic problem are fundamental in making efficient parallel algorithms fault-tolerant. This problem is 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 $P = N$). Thus, in the absence of failures, this problem is solved by a trivial and optimal parallel assignment. However, fault-tolerant solutions that must be efficient for worst case adaptive adversaries are non-obvious.

The iterated *Write-All* paradigm was 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] analyzed 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]. Note that this asycnhronous simulation has very good expected performance for the problem of this paper 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] has also improved the fail-stop deterministic lower and upper bounds of [KS 89] (by $\log\log N$ factors).

The general problem of assigning active processors to tasks has similarities to the problems of resource allocation in a distributed setting. Distributed controllers have been developed for resource allocation such as the algorithms of [LGFG 86] (in a probabilistic setting), and [AAPS 87] (in a deterministic setting). Fault-tolerance of particular network architectures is also studied in [DPPU 86]. However, the underlying distributed computation models, the algorithms and their analysis are quite different from the parallel setting studied here.

Finally, the work presented here deals with dynamic patterns of faults — 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].

Contributions:

We allow PRAM processors to be subject to on-line (dynamic) failures and *restarts*. Our failure/restart errors are not the same as the errors of omission because processors lose their state after a failure, while errors of omission cause a processor to skip a number of steps without losing its context.

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 our 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. We account for the work performed by the processors in a way that discounts trivial adversaries that would otherwise force *quadratic* work *Write-All* solutions. To guarantee algorithm termination and sensible accounting of resources, we introduce an *update cycle*, that generalizes the standard PRAM read/compute/write cycle. In Section 2, we first define the model and associated complexity measures, and then discuss the reasons for the choices made. The discussion motivates the use of update cycles, the only non-obvious technical choice made.

The trivial quadratic lower bound cited above is based on a *thrashing* adversary. It depends on the adversary exploiting the separation of *read* and *write* instructions in PRAMs. When reads and writes are accounted together in update cycles it no longer applies. Instead, we show that the *Write-All* problem of size N requires $\Omega(N \log N)$ work. This lower bound holds, even if processors could read and locally process all the shared memory at unit cost. Our simple lower bound is of interest, because it is matched by an $O(N \log N)$ upper bound under these assumptions. (Remark: An $\Omega(N \log N)$ lower bound was recently shown in [KPRS 90] using a different technique and different assumptions for a fail-stop no-restart model.) The upper bound proof arguments lead to a modification of the basic algorithm of [KS 89], so that it is efficient and correct in both the original setting, and with the failure and restart errors. We describe these arguments in Section 3.

In Section 4 we present the main result and supporting algorithms. This is 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 (sub-quadratic) completed work $O(\min\{N + P\log^2 N + M \log N, N \cdot P^{0.6}\})$, where M is the number of failures during this step's simulation.

This 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 N processor step is $O(N/\log N)$. The optimality result is preserved, of course, in the absence of failures. Our approach is based on: (a) a new algorithm for *Write-All* whose completed work is $O(N \cdot P^{\log_2 \frac{3}{2}+\delta})$ for $P \leq N$ and any $\delta > 0$, and which can handle any pattern of failures/restarts, (b) a modification of an algorithm from [KS 89], and (c) the techniques developed in [KPS 90, Shv 89].

The lower bounds apply to the worst case work of deterministic algorithms as well as to the expected work of randomized and deterministic algorithms. Interestingly, randomization does not seem to help, given *on-line*, i.e, non-prespecified, patterns of failures. For example, it is easy to construct on-line failure and restart (no-restart) patterns that lead to exponential (quadratic) in N expected performance for the algorithms presented in [MSP 90]. These *stalking* adversaries are described in Section 5, where we also conclude with some open problems.

2 Definitions

2.1 Restartable fail-stop CRCW PRAM

We use the COMMON CRCW PRAM model, where all concurrently writing processors write the same value. Processors are subject to stop failures *and restarts* as in [SS 83]. Our algorithms are described in a model independent fashion using high level notation with the obvious **forall/parbegin/parend** parallel construct.

The basis of the model is the PRAM of [FW 78]:

1. There are P *initial* synchronous processors. Each processor has a unique permanent identifier (PID) in the range $0, \ldots, P-1$, and each processor always knows its PID, and the number of processors P.

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 that are 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 fixed 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 PRAM instruction set. The values quoted (4 and 2) are sufficient for our exposition.

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 during the computation at least one processor is executing an update cycle that successfully completes, and

   (ii) failures can occur before or after a write of a single bit but not during the write, i.e., bit writes are *atomic*.

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.

Note that failures here are different from the errors of omission, where processors preserve their local context. The failure and restart patterns are syntactically defined as follows:

Definition 2.1 A *failure pattern* F is a set of triples $< tag, \text{PID}, t >$ 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 PRAM 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.

A-1

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:

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

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 $|F| \leq M$:

(i) uses *completed work*:

$$S = S_{N,M,P} = \max_{I,F}\{S(I,F,P)\}\ ,$$

(ii) has *overhead ratio*:

$$\sigma = \sigma_{N,M,P} = \max_{I,F}\left\{\frac{S(I,F,P)}{|I|+|F|}\right\}. \ \square$$

Remark 1 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.

Remark 2 Consider a definition of 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.

Remark 3 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) $T(|I|) = \Theta(|I|)$.

2.2 Discussion of the technical choices

Work vs. overhead ratio:

When dealing with arbitrary processor failures and restarts, the completed 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 2.1 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 $|F| = \Omega(N^{1+\epsilon})$ for $\epsilon > 0$, then the completed work will be $\Omega(N^{1+\epsilon})$, 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. $\square$

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 from Remark 2, 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. Without it, the algorithms may not terminate, and when they do terminate the work may not be bounded by a function of N and P. 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 2.2 Let ALG be any algorithm that solves the *Write-All* problem under the arbitrary failure and restart model. Consider the standard PRAM read, compute, write cycles (if processors begin writing without reading, a simple modification of the following 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. The adversary then restarts all failed processors. Since one write operation is performed per read, compute, write 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, and not for partially completed cycles, we do not charge for thrashing adversaries. It is interesting that this change in cost measure allows subquadratic solutions.

2.3 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 multiprocessor systems whose processors are subject to fail-stop errors and restarts. Here we suggest one way of realizing our abstract model of computation.

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 we cite below are instrumental in providing the basic hardware fault-tolerance, thus providing a foundation on which the algorithmic and software fault-tolerance can be built.

Semiconductor memories are the essential components of processors and of shared memory parallel systems. These memory are being 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

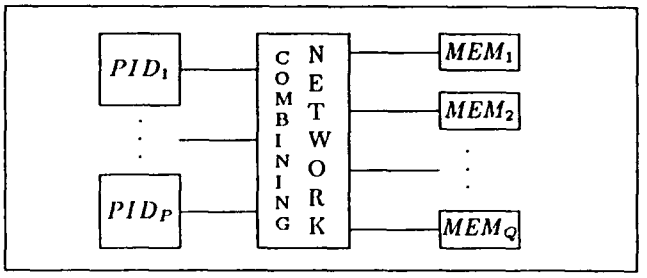

Figure 1: A robust fail-stop multiprocessor.

perfectly suited for implementing synchronous concurrent reads and writes is formally treated in [KRS 88].

Finally fail-stop processors are formally treated 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 overviewed:

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. This is true for as long as the shared memory and the interconnection network are subject to the failures within their respective design parameters.

3 Lower bounds

As we have shown in Example 2.2, without the update cycle accounting there is a thrashing adversary that exhibits a quadratic lower bound for the *Write-All* problem. With the update cycle accounting, we prove a $\Omega(N \log N)$ lower bound theorem.

Theorem 3.1 Given any N-processor CRCW PRAM algorithm that solves the *Write-All* problem of size N, then the adversary, that can cause arbitrary processor failures and restarts, can force the algorithm to perform $\Omega(N \log N)$ 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 iterative strategy:

All N processors are revived. For the upcoming cycle, the adversary determines the processors assignment to array elements. Let $U > 1$ be the number of unvisited array elements. By the pigeonhole principle, for any processor assignment to the U elements, there is a set of $\lfloor \frac{U}{2} \rfloor$ unvisited elements with no more than $\lceil \frac{N}{2} \rceil$ processors assigned to them. The adversary chooses half of the remaining previously unvisited array locations that would have had no more than $\lceil \frac{N}{2} \rceil$ processors assigned to them, and it fails these processors, allowing all others to proceed. Therefore at least $\lfloor \frac{N}{2} \rfloor$ 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 $\log N$ iterations. The work S performed by the algorithm will be $S \geq \lfloor \frac{N}{2} \rfloor \log N = \Omega(N \log N)$. □

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. Such an assumption is very strong. However we take advantage of the constructive proof strategy in the next section.

Theorem 3.2 If the fail-stop processors can read and locally process the entire shared memory at unit cost, then a solution for the *Write-All* problem can be constructed such that its completed work, when using N processors on the input of size N is $S = \Theta(N \log N)$.

Proof: We complement the previous lower bound with the following oblivious 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 $i = \lceil PID \cdot \frac{U}{N} \rceil$. This achieves load balancing with no more than $\lceil \frac{N}{U} \rceil$ processors assigned to each unvisited element.

We list the elements of the *Write-All* array according to the time at which the elements are visited in ascending order. We break this list into adjacent segments numbered sequentially starting with 1, such that segment j contains $V_j = \lfloor \frac{N}{j(j+1)} \rfloor$ elements, for $j = 1, ..., m$ and for some $m < \sqrt{N}$. When processors were assigned to the elements of the jth segment, there were no less than $U_j = N - \sum_{i=1}^{j-1} V_i \geq N - (N - \frac{N}{j}) = \frac{N}{j}$ unvisited elements. Therefore no more than $\lceil \frac{N}{U_j} \rceil$ processors were assigned to each element.

The work performed by such an algorithm is:

$$S \leq \sum_{j=1}^{m} V_j \lceil \tfrac{N}{U_j} \rceil = O(\sum_{j=1}^{m} \tfrac{N}{j(j+1)} \lceil \tfrac{N}{N/j} \rceil)$$
$$= O(N \sum_{j=1}^{\infty} \tfrac{1}{j+1}) = O(N \log N) \ . \ \square$$

4 Computation on restartable fail-stop processors

We first state the main result and then build the framework for proving it.

Theorem 4.1 Any N-processor PRAM algorithm can be executed on a 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:

(i) the completed work:
$S = O(\min\{N + P \log^2 N + M \log N, \ N \cdot P^{0.6}\})$,

(ii) the 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 4 PRIORITY CRCW PRAMs cannot be directly simulated using the same framework, for one of the algorithms used (namely algorithm X in Section 4.2) does not possess the *processor allocation monotonicity* property that assures that higher numbered processors simulate the steps of the higher numbered original processors.

We obtain this result by: (a) modifying an algorithm from [KS 89] to enable its use with restarts, (b) presenting a new algorithm that has a good overhead ratio efficiency and that terminates with sub-quadratic completed work, (c) merging the two algorithms, and using the techniques of [KPS 90] or [Shv 89] to produce efficient executions of arbitrary PRAM programs on faulty CRCW PRAMs.

We assume that N is a power of 2. Nonpowers of 2 can be handled using conventional padding techniques. All logarithms are base 2. Now the details.

4.1 Algorithm V: a modification of W of [KS 89]

Algorithm W of [KS 89] is an efficient fail-stop (no restart) *Write-All* solution. The algorithm uses full binary trees as its basic data structures. The trees are implicitly coded as heaps and are stored in linear arrays.

The algorithm uses an iterative approach in which all active processors synchronously execute the following four phases:

1. In the first phase the processors are counted and enumerated using a static bottom-up, logarithmic time traversal of the processor counting tree data structure.
2. In the second phase the processors are allocated to the unvisited array locations according to a divide-and-conquer strategy using a dynamic top-down traversal of a progress tree data structure.
3. The third phase is where the actual work (array assignments) is done.
4. In the fourth phase the progress is evaluated by a dynamic bottom-up traversal of the progress tree.

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, because 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 the processor assignment based on the proof of Theorem 3.2 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.

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 through the following three phases (we "prime" the phases to distinguish them from the phases of algorithm W):

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

2′ The processors now perform work at the leaves they reached in Phase 1′ (there are $\log N$ array elements per leaf).

3′ The processors begin at the leaves of the progress tree where they ended Phase 2′ and update the progress tree dynamically, bottom up ($O(\log N)$ time).

The following implementation detail is important in realizing processor re-synchronization after a failure and a restart. An iteration wrap-around counter is utilized, so that 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". If after a restart, a processor detects that the counter did not change for one cycle, it asserts that no processors were active at the point of the restart, and it can start a new iteration by itself – this is possible since the processors are synchronous.

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 4.2 The completed work of 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 distinguish 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 Theorem 3.2, 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. Similarly to Theorem 3.2, when the first $\frac{N}{\log N} - P$ leaves are visited, there are no more than 1 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 3.2 (not counting processor allocation). Each leaf visit takes $O(\log N)$ work steps, therefore 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.

completed work $S = O((\frac{N}{\log N} - P + P \log P) \log N) = O(N + P \log P \log N) = O(N + P \log^2 N)$.

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 3.2 with the $\lceil P/\frac{N}{\log N} \rceil$ processor allocation as a multiplicative factor. From this the completed work S is $\lceil P/\frac{N}{\log N} \rceil O(\frac{N}{\log N} \log \frac{N}{\log N}) O(\log N) = O(P \log^2 N)$.

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

The following theorem expresses the completed work of the algorithm:

Theorem 4.3 The completed work of V using $P \leq N$ processors subject to 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 4.2 does not rely on the fact that in the absence of restart, the number of active processors is non-increasing. However the lemma does not account for the work that might be spent by the processors that are active during a part of an iteration without contributing 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 4.2 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 a large $|F|$). □

4.2 Algorithm X and its analysis

We present a new algorithm X for the *Write-All* problem. We show that its completed work complexity is $S = O(N \cdot P^{0.6})$ for any failure/restart pattern using $P \leq N$ processors. The important property of X is that it has a bounded sub-quadratic completed work regardless of the failure pattern, and if a very large number of failures occures, say $|F| = \Omega(N \cdot P^{0.6})$, then the algorithm's overhead ratio σ becomes optimal: it takes a fixed number of computing steps per failure/recovery.

The algorithm utilizes a progress tree of size N as algorithm V, but it is traversed by the processors independently, and 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 neccessary work, and moves out of that subtree when no more work remains. Details follow.

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* $d[1 \ldots 2N\text{-}1]$ in shared memory. An internal tree node $d[i]$ $(i = 1, \ldots, N - 1)$ has the left child $d[2i]$ and the 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 $x[1 \ldots N]$ is 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.

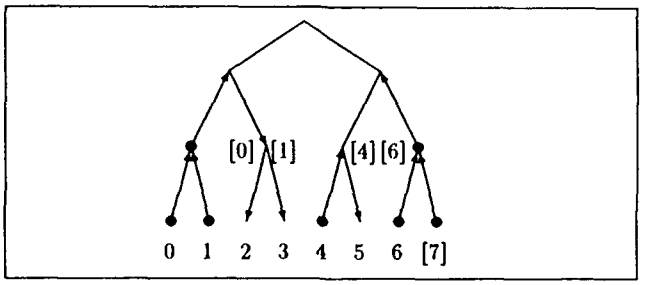

Figure 3: Processor traversal of the progress tree.

Control-flow: The algorithm consists of a single initialization and of the parallel *loop*. The high level view of the algorithm is in Figure 2 (all line numbers refer to the figure), a more detailed code is in the appendix.

This algorithm is performed by all processors that are active. The initialization (line 02) assignes 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) to: (line 04) move up the tree if the current node is marked done, (line 05) perform the work if at a leaf, (line 07) update the interior tree node if both of its subtrees are done by changing its value from 0 to 1, (line 08) move down to the left/right subtrees based on either the one of the subtrees being not done.

For the final case (line 09), the processors move down when neither child is done based on the processor identifier. This last case is where the 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.

Remark 5 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.

Example 4.1 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 in Figure 3 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. □

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 the appendix for the detailed algorithm). Therefore the body can be implemented as an update cycle.

Analysis of algorithm X:

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

Lemma 4.4 Algorithm X with N processors is a correct $\Omega(\log N)$ and $O(N)$ time fault-tolerant solution for the *Write-All* problem of size N. □

Now a lemma relating completed work when overlapping of processors occurs, and 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.

Lemma 4.5 For algorithm X, if N is the size of the input, and $N \leq P_1 \leq P_2$, then the work using P_1 processors and the work using P_2 processors relate as $S_{N,P_2} \leq \lceil \frac{P_2}{P_1} \rceil S_{N,P_1}$.

Proof sketch: This follows from the Definition 2.2 of S and the observation that if $P > N$, then exactly $\log N$ bits of the PIDs are significant during the execution of algorithm X. We observe that any two processors whose PIDs are equal modulo N, will expend no more than a single processor in the worst case at twice the cost. □

Lemma 4.6 The work complexity S of algorithm X with N initial processors for the *Write-All* problem of size N and for any pattern of failures and restarts is $S = O(N^{\log_2 3+\delta})$ for any $\delta > 0$.

Proof: We will show that for any positive δ there is a constant c, such that $S \leq cN^{\log 3+\delta}$. We proceed by induction on the height of the progress tree. For the base case: we have a tree of height 0 that corresponds to an input array of size 1, and exactly 1 processor. Since at least this processor will be active, this single leaf will be visited in a constant number of steps. Let the work expended be c' for some constant c' that depends only on the lexical structure of the algorithm. Therefore $S_{1,1} = c' \leq c \cdot 1^{\log 3+\delta}$ for all $c > c'$, and any $\delta > 0$.

For the inductive hypothesis: we assume that for the tree heights less than $\log N$, and for any $\delta > 0$, the required constant c exists. We then prove that this is true for the tree of height $\log N$.

Consider the two subtrees of the root (Figure 4). The two corresponding subtrees are of the heights $\log N - 1$. By the definition of algorithm X, no processor will leave a subtree until the subtree is finished. We have to consider the following two sub-cases: (1) both subtrees are finished simultaneously, and (2) one of the subtrees is finished before the other.

Case 1: If both subtrees are finished simultaneously, thgen the algorithm will then terminate after some small constant number of steps c' when a processor moves to the root and determines that both of the subtrees are finished. By the inductive hypothesis, there exists a c such that both the work S_L expended in the left subtree of, and the work S_R in the right subtree are bounded by $S_{\frac{N}{2},\frac{N}{2}} \leq c(\frac{N}{2})^{\log 3+\delta}$. The work needed for the algorithm to terminate is at most $c'N$, and so the total work is:

$$S \leq S_L + S_R + c'N \leq 2S_{\frac{N}{2},\frac{N}{2}} + c'N$$
$$\leq 2c(\tfrac{N}{2})^{\log 3+\delta} + c'N = c\tfrac{2}{3\cdot 2^\delta}N^{\log 3+\delta} + c'N.$$

When c is chosen sufficiently larger than c', e.g., $c \geq 3c'$, then $S \leq cN^{\log \frac{3}{2}}$.

Case 2: Assume w.l.o.g. that the left subtree is finished first with $S_L = S_{\frac{N}{2},\frac{N}{2}} \leq c(\frac{N}{2})^{\log 3+\delta}$ by the inductive hypothesis. The 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 $c' \log N$ for a predefined constant c' (the longest path from a leaf to another leaf). No more than the original $\frac{N}{2}$ processors of the left subtree will move, and so the work of moving the processors is bounded by $c'\frac{N}{2}\log N$.

By Lemma 4.5 and by the inductive hypothesis, the work S_R to complete the right subtree using N processors is bounded by $S_{\frac{N}{2},N} \leq 2S_{\frac{N}{2},\frac{N}{2}} \leq 2c(\frac{N}{2})^{\log 3+\delta}$. After this, each processor will spend some constant number of steps moving to the root and terminating the algorithm. This work is bounded by $c''N$ for some small constant c''. The total work S is:

$$S \leq S_L + c'\tfrac{N}{2}\log N + S_R + c''N$$
$$\leq c(\tfrac{N}{2})^{\log 3+\delta} + c'\tfrac{N}{2}\log N + 2c(\tfrac{N}{2})^{\log 3+\delta} + c''N$$
$$\leq \tfrac{c}{2^\delta}N^{\log 3+\delta} + c'\tfrac{N}{2} + c''N$$

When c is made sufficiently large based on δ with respect to the fixed c' and c'', e.g., $c \geq \frac{2^\delta(c'+c'')}{2^\delta-1}$, then: $S \leq cN^{\log 3+\delta}$.

Since a constant c depends only on the lexical structure of the algorithm and δ, it 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 second case and of the lemma. □

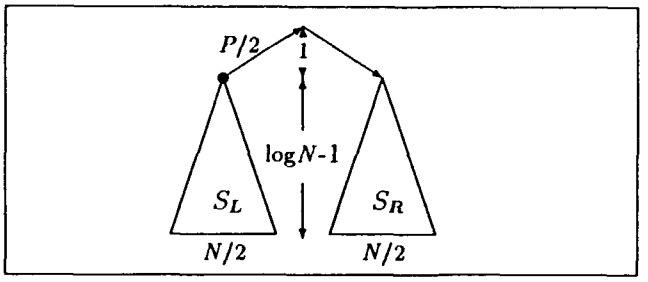

Figure 4: Inductive step for Lemma 4.6.

Now we generalize this result for $P \leq N$:

Theorem 4.7 There is an algorithm that solves the *Write-All* problem with completed work $S = O(N \cdot P^{\log \frac{3}{2}+\delta})$ for any $\delta > 0$, where N is the input array size, and $P \leq N$ is the initial number of processors.

Proof sketch: We position the P processors at the first P elements of the input array. It is easy to show that $S = O(\frac{N}{P}S_{P,P}) = O(\frac{N}{P}P^{\log 3+\delta}) = O(N \cdot P^{\log \frac{3}{2}+\delta})$. □

For example, when δ is about 0.01, $S = O(N \cdot P^{0.6})$. We next show a particular performance of algorithm X such that its completed work is asymptotically close to its upper bound.

Theorem 4.8 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 sketch: We compute the exact work performed by the algorithm when the adversary adheres to the following strategy: 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. The processors that find themselves at the same leaf as the processor 0 are (re)started, while the rest are failed. All processors with PIDs smaller than the index of the last leaf visited by processor 0 are allowed to traverse the progress tree until they reach a leaf. When processors reach a leaf, the failure/restart procedure is repeated. □

4.3 Combining the building blocks

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, and it was shown that the complexity of solving a N-size instance of the *Write-All* problem using P fail-stop processors, and the complexity of executing a single N-processor PRAM step on a fail-stop P-processor PRAM are equal. 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 4.1.

We first observe that the executions of algorithms V and X can be interleaved to yield an algorithm that achieves the following performance:

Theorem 4.9 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 $S = O(\min\{N + P\log^2 N + M\log N,\ N \cdot P^{0.6}\})$, and the overhead ratio is $\sigma = O(\log^2 N)$.

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 4.1 follows from Theorem 4.9 and the results of [KPS 90] or [Shv 89].

The following corollaries are also interesting:

Corollary 4.10 Under the hypothesis of Theorem 4.1, 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 is subsumed by Corollary 4.10. 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]. The exact analysis of algorithm V without restarts is still open.

Corollary 4.11 Under the hypothesis of Theorem 4.1:

1. when $|F|$ is $\Omega(N\log N)$, then σ is $O(\log N)$,
2. when $|F|$ is $\Omega(N^{1.6})$, then σ is $O(1)$.

Thus the efficiency of our algorithm 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.

Another interesting result is that there is a 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*×*Processors* product for that algorithm:

Corollary 4.12 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 $S = O(\tau \cdot N)$.

It also follows that optimality is preserved in the absence of failures or when during the execution of each N processor step there are $O(\log N)$ failures and restarts per each simulating processor. This is because in either of these two cases, the size of the failure/restart pattern F is bounded by: $|F| \leq O(P\log N) = O(\frac{N}{\log^2 N}\log N) = O(N/\log N)$.

5 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. First the open problems and future work:

- Lower bounds with and without restarts: We have shown an $\Omega(N \log N)$ lower bounds for failures/restarts under the assumption that processors can read and locally process the entire shared memory at unit cost. Under this assumption this is the best possible lower bound.

  Under the same assumption, it can be shown that the lower bound of [KS 89] of $\Omega(N \log N / \log \log N)$ is the best possible bound for failures without restarts.

  Under a different assumptions, an $\Omega(N \log N)$ is shown for failures without restarts in [KPRS 90]. Can these bounds be further improved using different assumptions?

- Upper bounds with restarts: Progress in this area ought to be made by finding new algorithms, or improving the analysis of existing algorithms to achieve better completed work S and the overhead ratio σ than those of algorithms V and X.

- Upper bounds without restarts: What is the worst case completed work S, and overhead ratio σ of the algorithm X in the case of fail-stop errors without restarts?

  Algorithm X appears to have a very good performance in the fail-stop (without restart) framework of [KS 89]. For example, the adversary used to show the lower bound in [KS 89] causes the worst case work of $S = \Theta(N \log^2 N / \log \log N)$ for the N-processor *Write-All* solution in [KS 89]. The same adversaary causes the known worst case work of X of $S = \Theta(N \log N \log \log N / \log \log \log N)$.

  We conjecture that the fail-stop (no restart) performance of X has work $S = \Theta(N \log N \log \log N)$ using N processors.

- For the update cycles used in this work, what is the minimum number of reads and writes that are sufficient to assure efficient solutions, and under what assumptions?

On randomization and lower bounds:

The existing upper bounds for randomized solutions for *Write-All* apply to *off-line*, i.e., non-adaptive adversaries. For example, 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 adaptive 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 their *ACC* algorithm performs expected $O(N)$ work using $P = \frac{N}{\log N \log^* N}$ processors on inputs of size N.

In contrast, we observe that a simple *stalking* adversary causes the *ACC* algorithm to perform (expected) work of $\Omega(N^2/poly \log N)$ in the case of fail-stop errors, and $\Omega((\frac{N}{poly \log N})^{\frac{N}{poly \log N}})$ work in the case of fail-stop errors with restart even when using $P \leq \frac{N}{poly \log N}$ 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 thank Jeff Vitter for helpful discussions, and Franco Preparata for reviewing an earlier draft.

6 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*, Vol.20, No.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 FOCS*, pp. 347-357, 1987.

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

[CZ 89] R. Cole and O. Zajicek, "The APRAM: Incorporating Asynchrony into the PRAM Model," in *Proc. of the 1989 ACM Symp. 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 Symp. on Parallel Algorithms and Architectures*, pp. 85-94, 1990.

[Cri 91] F. Cristian, "Understanding Fault-tolerant Distributed Systems", in *CACM*, 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 STOC*, pp. 370-379, 1986.

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

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

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

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

[IEEE 90] *IEEE Computer*, "Fault-Tolerant Systems", a 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 FOCS*, 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 (to appear in *Distributed Computing*); prel. version appears in *Proc. of the 8th ACM PODC*, pp. 211-222, 1989.

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

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

[KPS 90] Z. M. Kedem, K. V. Palem, and P. Spirakis, "Efficient Robust Parallel Computations," in *Proc. 22nd ACM STOC*, 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 STOC*).

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

[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 Symp. on Foundat. of Computer Sci.*, pp. 590-599, 1990. Also see Tech. Rep. CSE-89-6, Univ. of Calif.-Davis, 1989.

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

[Pip 85] N. Pippenger, "On networks of noisy gates", *Proc. of 26th IEEE FOCS*, pp. 30-38, 1985.

[Ran 87] A. Ranade, "How to Emulate Shared Memory", *Proc. of 28th IEEE FOCS*, 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 Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222-238, 1983.

[Sch 80] J. T. Schwartz, "Ultracomputers", *ACM Trans. on Prog. Lang. and Sys.*, 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 *Informat. Proc. Letters*).

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

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

[Val 90b] L. Valiant, "A Bridging Model for Parallel Computation," *Comm. of 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.

Appendix: Algorithm X pseudocode

Here we give a detailed pseudocode for algorithm X.

In the algorithm X 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 long binary 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.

Remark 6 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. We are not providing further details here.

Remark 7 The algorithm can be used to solve *Write-All* "in place" using the array x[] as a tree of height $\log \frac{N}{2}$ 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.