Fault-Tolerant and Efficient Parallel Computation

Alexander Allister Shvartsman

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-23
April 1992

Fault-Tolerant and Efficient Parallel Computation

by

Alexander Allister Shvartsman

B. S., Computer Science, Stevens Institute of Technology, 1979

M. S., Computer Science, Cornell University, 1981

Sc. M., Computer Science, Brown University, 1988

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1992

© Copyright 1992
by

Alexander Allister Shvartsman

This dissertation by **Alexander Allister Shvartsman**

is accepted in its present form by the Department of
Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Paris C. Kanellakis

Recommended to the Graduate Council

Date _____ _____

Franco P. Preparata

Date _____ _____

Jeffrey S. Vitter

Approved by the Graduate Council

Date _____ _____

Abstract

RECENT advances in computer technology made parallel machines a reality. Massively parallel systems use many general-purpose, inexpensive processing elements to attain computation speed-ups comparable to or better than those achieved by expensive, specialized machines with a small number of fast processors. In such setting, however, one would expect to see an increased number of processor failures attributable to hardware or software. This may eliminate the potential advantage of parallel computation. We believe that this presents a reliability bottleneck that is among fundamental problems in parallel computation.

We investigate *algorithmic ways of introducing fault-tolerance in multiprocessors under the constraint of preserving efficiency.* This research demonstrates how in certain models of parallel computation it is possible to combine efficiency and fault-tolerance. We show that in the models we study, it is possible to develop efficient parallel algorithms without concern for fault-tolerance, and then correctly and efficiently execute these algorithms on parallel machines whose processors are subject to arbitrary dynamic fail-stop errors. By ensuring efficient executions for any patterns of failures, the efficiency is also maintained when failures are infrequent, or when the expected number of failures is small.

The efficient algorithmic approaches to multiprocessor fault-tolerance presented in this thesis make a contribution towards bridging the gap between the abstract models of parallel computation and realizable parallel architectures.

Acknowledgements

I THANK my advisor Paris C. Kanellakis who provided the initial vision for this work. His scientific intuition, well-founded optimism, research discipline and contagious energy greatly contributed to the success of our work. Working together through long brainstorming sessions were the most memorable and enjoyable times during my stay at Brown. Now Paris is "losing" a student, but he has gained a colleague and a friend.

I thank my committee members Franco Preparata and Jeffrey S. Vitter. I had the privilege of consulting with Jeff and Franco well before they joined my committee, and their scientific insight has improved the quality of the papers that became an important part of this thesis. It is a pleasure to acknowledge both of them here as well.

I thank John E. Savage and Peter Wegner for encouraging me to come to Brown, and for many quality discussions we had during my stay at Brown.

I acknowledge with gratitude the funding of Digital's Graduate Engineering Education Program. I thank the Program's manager Shirley Stahl for her encouragement, thoughtful guidance and advice. Thanks are due to my technical advisors, Bert Halstead at Cambridge Research Lab and Colin Strutt at Networks and Communications for sponsoring me. I also thank my support team members, John C. Egolf, Anne A. Pelagatti, and Peter Savage and Al Lathrop for reviewing parts of this work.

Parts of my research were also funded by the Office of Naval Research grant N00014-91-J-1613. I am grateful for this ONR support.

I also have great many (old and new) debts to my family. My scientific curiosity was nurtured by Maria A. Charzewska-Theodoroff, my grandmother and a teacher of mathematics. I thank my wife and my best friend Robin for everything. My work was made more enjoyable by the companionship of my daughter Ginger, who touches my soul, and my son Ted, who gives me strength. Finally, I thank my parents who sacrificed everything for their children.

Credits

MAJOR parts of this dissertation are based on the works [27, 55, 56, 57, 92, 94]. These papers correspond to the sections of the thesis, chronologically, as follows:

[55] This paper is a joint work with Paris Kanellakis, and it is in the Proceedings of *PODC'89*. This work was extended and it will appear in *Distributed Computing*, vol. 5, 1992. It maps to Sections 2.1-2.5, 3.2.1, 4.1, 5.1, 5.5.2 and 6.2.

[92] This work is singly authored and it appears in *Information Processing Letters*, vol. 39, 1991. It corresponds to Section 3.2.3, 5.2-5.3.

[56] This is a joint work with Paris Kanellakis, and it appeared in the Proceedings of *PODC'91*. This work maps to Sections 2.6, 3.2.2, 3.3, 4.2 and 5.4.

[57] This is a survey based on [55, 56]. It is in the Proceedings of *ONR Workshop on Ultradependable Multicomputers*, 1991. The new material maps to Section 3.4.1.

[27] This is a joint paper with Buss, Kanellakis and Ragde. It includes the full version of [56] and the independent results of Buss and Ragde. This work was submitted to *SIAM Journal of Computing*.

[94] This is a singly authored work. This University technical report was submitted to *Information Processing Letters*, and it corresponds to Sections 5.5.1 and 6.1.

A more detailed credit is warranted for the following:

The frameworks and the algorithms of [55] and [56] were developed jointly with Kanellakis. The lower bounds in [55] and [56] and the treatment of atomicity are due to Shvartsman. The determinization of algorithm Y is due to Kanellakis [57].

Algorithm X in Section 3.3.1 was independently discovered by Buss and Ragde for the strongly asynchronous model [27]. The general simulation in [92] was independently developed by Kedem et al. [59]. Martel [71] proved a better bound for algorithm W of Section 3.2.1. For completeness we include this proof in Appendix A.6.

Contents

Abstract vii

Acknowledgements ix

Credits xi

1 Introduction 1

1.1 Overview and Motivation 1

1.2 Contributions 5

1.3 Related Work 7

1.3.1 Fault-tolerant parallel computation 7

1.3.2 Fault-tolerant distributed computation 9

1.3.3 Technology for fault tolerance 10

1.4 Relation to Physical Systems 12

1.5 Structure of the Document 15

2 Models and Definitions 17

2.1 Base Model of Computation 17

2.2 Measures of Efficiency 18

2.3 The *Write-All* Problem 21

2.4 Models of Failure 22

2.5 No-restart Fail-stop CRCW PRAM 25
2.5.1 Failure model 25
2.5.2 Measure of efficiency: available processor steps 26
2.5.3 Discussion of the technical choices made 27
2.6 Restartable Fail-stop CRCW PRAM 28
2.6.1 Failure and restart model 29
2.6.2 Measures of efficiency: completed work and overhead ratio 30
2.6.3 Discussion of the technical choices made 31

3 Write-All Algorithms **35**

3.1 Processor Allocation Paradigms 35
3.2 Global Allocation Paradigm 37
3.2.1 Algorithm W 37
3.2.2 Algorithm V 51
3.2.3 Processor Allocation Monotonicity 56
3.3 Local Allocation Paradigm 57
3.3.1 Algorithm X 57
3.3.2 Algorithms X_{coin} and X_{die} 67
3.4 Hashed Allocation Paradigm 70
3.4.1 Algorithm Y 70

4 Write-All Lower Bounds With Memory Snapshots **75**

4.1 Lower Bounds for the No-Restart Fail-Stop Model 76
4.2 Lower Bounds for the Restartable Fail-Stop Model 79
4.3 Other bounds 81
4.3.1 A lower bound for CREW PRAM 81
4.3.2 Lower bounds with test-and-set operations 83

5 Algorithm Simulations and Transformations **85**

5.1 General Parallel Assignment 86

5.2 A PRAM Interpreter 88

5.3 General Simulations on Fail-stop Processors Without Restart 90

5.4 General Simulations on Restartable Fail-Stop Processors 96

5.5 Improving Oblivious Simulations 99

5.5.1 Parallel prefix 99

5.5.2 Pointer doubling 101

5.6 On Parallel Complexity Classes and Fault-Tolerance 103

5.6.1 Fail-stop model without restarts 105

5.6.2 Restartable fail-stop model 107

6 Simplifying Memory Assumptions **109**

6.1 Solving Write-All Using Contaminated Memory 109

6.1.1 Write-All assumptions 110

6.1.2 Model: fail-stop PRAM with contaminated memory 111

6.1.3 Write-All algorithms using contaminated memory 112

6.1.4 General simulations and algorithm transformations 117

6.2 Atomic Access and Word Size 119

7 Discussion and Open Problems **121**

Bibliography and References **125**

A Pseudocode for algorithm W and Two Lemmas **135**

A.1 Main Procedure for Algorithm W 136

A.2 Static Bottom Up Traversal 137

A.3 Dynamic Bottom Up Traversal 138

A.4 Dynamic Top Down Traversal 139
A.5 Dynamic Dop Down Traversal Lemma 3.2 140
A.6 Martel's Improved Lemma 3.4 142

B Algorithm X pseudocode **145**

C Mathematical lemmas used for lower bounds **147**

List of Tables

5.1 Closure under the fail-stop transformation ξ (for $P = P(N)$). 106

5.2 Closure under the restartable fail-stop transformation ρ (for $P = P(N)$). 108

List of Figures

1.1 A robust fail-stop multiprocessor. 14

2.1 Work in the absence (W) and presence (S) of processor failures. 20

3.1 A high level view of algorithm W 39
3.2 A high level view of algorithm V 53
3.3 A high level view of algorithm X. 58
3.4 An example of processor traversals of the progress tree 59
3.5 Inductive step for Lemma 3.15 61
3.6 A fail-stop scenario for algorithm X with $P = N = 64$. 65
3.7 Stages 1 and 2 of a general fail-stop scenario for algorithm X 66
3.8 An example of an inefficient progress tree traversal 69
3.9 A high level view of the algorithm Y – hashed allocation paradigm. . . . 71
3.10 A detailed view of the deterministic algorithm Y. 72

5.1 Universal PRAM Interpreter. 89
5.2 Modified *UPI* using two generations of shared memory. 92
5.3 Pseudo-code for a robust *UPI*. 93
5.4 Second stage of robust prefix computation. 101
5.5 A high level view of the robust pointer doubling algorithm 102

6.1 A high level view of the bootstrap algorithm. 113

6.2 Contamination robust processor enumeration Z_{enum}. 115

A.1 Main procedure of algorithm W. 136

A.2 Phase W1 procedure — Static bottom up traversal 137

A.3 Phase W4 procedure — Dynamic bottom up traversal 138

A.4 Phase W2 procedure — Dynamic top down traversal 139

A.5 Proof outline of the phase W2 top down traversal 141

B.1 Algorithm X detailed pseudo-code. 146

List of Examples

2.1 Work subject to failures 20

2.2 Write-All 21

2.3 Work subject to a large number of recoveries 31

2.4 Thrashing adversary 32

3.1 Processor enumeration 41

3.2 Progress estimation 43

3.3 Progress tree traversal 59

3.4 Stalking adversary 67

3.5 Inefficient progress tree traversal 69

5.1 General parallel assignment 86

Chapter 1

Introduction

1.1 Overview and Motivation

MASSIVELY parallel machines and networks consisting of hundreds and thousands of processors are in existence today, and the multiprocessor technology is continuing to evolve. In order to take advantage of the processing power of these parallel computing environments, there is a corresponding need for efficient algorithms. Significant research in the past decade was dedicated to the development of efficient parallel algorithms that assume perfectly reliable parallel computers. However, the algorithms that are to be executed on realizable parallel machines must be able to deal with unpredictable system failures. In this dissertation we address *efficient algorithmic approaches to multiprocessor fault-tolerance.*

In parallel systems, as the number of inexpensive processing units grows, one would expect to see an increased number of failures per individual processing element. The hardware failures may be caused by fabrication defects or by intermittent errors. It would be too costly to make each parallel processing element as reliable in hardware as a single processor machine. When the number of processors is in the thousands, it is likewise impractical to provide fault tolerance and fault masking at the level that can be achieved by the more expensive, specialized machines with a small number of fast processors such as Tandem [17], Stratus [101] or VAXft [26]. In addition, the software for these systems is typically more complex and thus less reliable than the software of the more conventional uniprocessors. Therefore, it is critical that fault-tolerant versions

of existing or new algorithms be developed, which preserve efficiency under adverse conditions.

In order for practical multiprocessor software to incorporate specific fault-tolerant algorithms and utilize particular techniques in assuring fault tolerance, these algorithms and techniques must meet the following criteria:

Efficiency : algorithms must use well defined and bounded computation and network resources. Most of the theoretical work to date has concentrated on this goal.

Scalability : an algorithm should exhibit stable and predictable performance within parallel environments of increasing sizes. The traditional asymptotic analysis guarantees this goal only to the extent that reliability and important implementation details are modeled by the theory (see next two points).

Reliability : an algorithm must remain operational in a parallel environment subject to failures. Reliability, efficiency and scalability are the main subject of this thesis.

Feasibility : an algorithm must be implementable using state of the art network and computing technology, so that it takes advantage of the considerable computing resources available. Model and algorithm simplicity are keys to being able to integrate abstract solutions with realizable hardware and software systems.

The efficiency and scalability of parallel algorithms have been the subject of research since the seventies. A model of parallel computation known as the Parallel Random Access Machine or PRAM [44] has attracted much attention, and many "efficient" and "optimal" algorithms have been designed for it (the surveys [40, 58] contain a wealth of information on the subject). The PRAM is a convenient abstraction that combines the power of parallelism with the simplicity of a RAM (Random Access Machine) [39], but it has several unrealistic features. The PRAM has the following requirements:

1. Simultaneous access across a significant bandwidth to a shared resource, 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 [98]; for some recent work see [49, 87, 97]. Computation on asynchronous PRAMs are the subject of [29, 31, 45, 75, 78]. The reliability of semiconductor memories has been thoroughly studied, and a survey can be found in [89], while the theory of error detecting and correcting codes is reviewed in [76]. The fault-tolerant issues of the interconnection networks used to integrate processors and memory modules are discussed in [2]. Fault tolerance of systolic arrays — a particular class of parallel machines — has been studied to some extent, and some of the achievements in that area are surveyed in [1]. All these areas are extremely important for dependable massively parallel computing. In this work we address the following issues: the reliability and synchronization of parallel processors that can be modeled by PRAMs, and the efficiency of computation on such processors.

The model of parallel computation that serves as the basis for this work is the synchronous PRAM of Fortune and Wyllie [44], with concurrent reads and concurrent writes (CRCW). The convention for determining which processor or processors succeed, when concurrently writing to shared memory, is immaterial in our algorithms. We investigate fault-prone PRAMs whose processors exhibit *fail-stop* processor behavior, such as that of Schlichting and Schneider [90]. The only atomicity requirement is the *atomic* concurrent writing of single bits. The redundancy provided by concurrent reads and writes is essential to our model, e.g., when the writes are exclusive we show that efficiency and fault tolerance cannot be combined.

This thesis includes and extends the study of fault tolerance that was first formalized by Kanellakis and Shvartsman in [55]. As it was shown there, it is possible to combine efficiency and fault tolerance in many key PRAM algorithms in the presence of arbitrary dynamic 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. The first algorithm for the *Write-All* problem with poly-logarithmic overhead in work was shown in [55].

Using solutions to the *Write-All* problem, we show that arbitrary PRAM algorithms can be efficiently and deterministically executed on fail-stop PRAMs, whose processors are subject either to arbitrary dynamic patterns of failures, or the dynamic patterns of failures and restarts.

An important part of this research addresses the definition of models for fault-tolerant parallel computation based on selections from a spectrum of types of failures and on the architecture of the system used. Such modeling must necessarily precede the development of techniques for constructing parallel algorithms.

There is an interesting body of research in distributed algorithms addressing problems similar to those we consider in this thesis. Some of the high level concerns addressed by our parallel algorithms are analogous to those encountered by other researchers in their work on distributed algorithms. Both the parallel and distributed techniques share the goals of *failure detection, load scheduling* and *progress evaluation*. For example, fault tolerance is the subject of significant current research in the setting of dynamic asynchronous network protocols. Distributed controllers have been developed for resource allocation in network protocols, where the total number of messages sent is the resource monitored [4, 70]. Other developments [3, 13, 15] solve the problems of executing distributed algorithms in the presence of dynamic network changes, i.e., dynamic changes of the computation medium.

One of the important results of the work in the distributed model is that it is possible to take synchronous algorithms or algorithms that are designed for fixed network topologies and "compile" them so that they can be used with asynchronous networks or the networks whose topology changes dynamically [15]. Our research begins to yield similar results for certain shared memory parallel models.

The potential reliability advantage of distributed computing systems is due to the replication of resources. The resulting redundancy in computation is a trade-off of efficiency (measured in terms of available resources) for fault tolerance. Mullender, in the

1989 *Distributed Systems* edition of the ACM's *Frontier Series* [77], considers independent failures essential for distributed systems, i.e., a failure of a single processing node must not lead to the failure of the entire distributed system. He gives the disqualifying disadvantage of multiprocessors with shared memory [77, page 6]:

> What disqualifies multiprocessors is that there is no independent failure: when one processor crashes, the whole system stops working. . . .

Although Mullender primarily refers to the problems of the available general purpose multiprocessor architectures, it is nevertheless true that most efficient parallel algorithms assume perfect processor reliability, and therefore these algorithms crash or produce incorrect results even if a single processor failure is encountered. The underlying theme that we address in this thesis in the context of parallel algorithms is

> *Combining the reliability potential of distributed computing with the speed-up potential of parallel computing.*

1.2 Contributions

We now overview the contributions of the research included in this thesis. Our main results have established that it is possible to combine efficiency and fault-tolerance for two particular parallel models: no-restart fail-stop CRCW PRAMs and restartable fail-stop CRCW PRAMs.

We formalized a model of computation and failures (at the granularity of fail-stop processing units), and defined a complexity measure for evaluating the algorithms' efficiency and fault tolerance. The key complexity measure that we define generalizes the notion of work of parallel algorithms commonly expressed as the *Parallel-time* $\times$ *Processors* product (in the absence of failures). We introduced the *Write-All* paradigm and showed that efficient parallel algorithms can be made robust, that is, be efficient and correct in the presence of arbitrary fail-stop errors without restarts as long as a single processor remains active. Specifically, the efficiency of parallel algorithms is degraded by no more than a multiplicative factor that is square in the logarithm of the size of the input. In many circumstances we achieve even lower overheads.

We have also shown that concurrent write model (CRCW) is essential in achieving robustness. Concurrent writes is the source of redundancy in our approach. Without concurrent writes, algorithms may incur up to a linear multiplicative overhead as is the case with the concurrent read exclusive write (CREW) model.

For the no-restart fail-stop CRCW PRAMs we developed a general fault-tolerant PRAM algorithm simulation. We showed how to execute correctly and efficiently any PRAM algorithm on a fault-prone fail-stop PRAM. The simulation is based on a solution to the *Write-All* problem and the techniques used in implementing the *general parallel assignment*. We have also shown that any parallel algorithm can be optimally executed in the presence of arbitrary fail-stop errors using a slightly smaller number of processors by taking advantage of *parallel slackness* as advocated by Valiant in [98]. By *optimal execution* we mean that the asymptotic efficiency of the source algorithm is not degraded even if arbitrary fail-stop errors are encountered. Given a N-processor PRAM algorithm we simulate it efficiently by a P-processor fail-stop CRCW PRAM algorithm, for $P \leq N$. The simulation is optimal for $P \leq N/(\log^2 N - \log N \log\log N)$.

Extending the fail-stop no-restart model, we formulated a *restartable fail-stop* CRCW PRAM and computations in this model. We allow the PRAM processors to be subject to arbitrary stop failures and restarts that are determined by an on-line adversary. The failures 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.

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 $O(\min\{N + P\log^2 N + M\log N, N \cdot P^{0.59}\})$ (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 $P \leq N/(\log^2 N - \log N \log\log N)$ 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, together with a modification of our main fail-stop algorithm.

We studied the lower bounds for the no-restart and for the restartable fail-stop PRAMs.

We show that there exist no optimal *Write-All* solutions for N-processor no-restart PRAM, even if each processor can perform *memory snaphsots*, i.e., read and locally process the entire shared memory at unit cost. The result showed that under this hypothesis, $\Omega(N \log N / \log\log N)$ work will be required. This is the strongest possible bound under this assumption.

For the restartable PRAM model, we showed that the *Write-All* problem requires $\Omega(N \log N)$ completed work when $P = N$, and this lower bound holds even under the additional assumption of *memory snapshots*. Under this assumption we have a matching upper bound.

Despite the memory snapshot assumption, these lower bounds are of interest also because we use these results to show the lower and upper bounds for some of the algorithms we develop in this work. The lower bounds for both models also apply to the expected work of randomized algorithms.

We also show that for some fundamental algorithms, it is possible to construct fault-tolerant algorithms that improve on the efficiency of naïve general simulations. For example, we show how to use the *Write-All* technique to achieve savings in computing parallel prefixes for any associative operation, and compute list ranking.

Finally, using a deterministic bootstrapping and balancing argument, we show how to solve the *Write-All* problem when auxiliary memory is contaminated with arbitrary values. All previous *Write-All* solutions use $\Omega(P)$ auxiliary shared memory and assume that this memory is cleared or initialized to some known value. For any dynamic pattern of fail-stop, no-restart errors on a CRCW PRAM with at least one surviving processor, our new algorithm writes all 1's using $O(N + P \log^3 N/(\log\log^2 N))$ work *without any initialization assumption.* This technique can be combined with any *Write-All* algorithm to yield efficient simulations of any PRAM and even optimal simulations given processor slack. It can also be used with restartable fail-stop processor simulations.

1.3 Related Work

1.3.1 Fault-tolerant parallel computation

The study of PRAM fault tolerance was initiated by Kanellakis and Shvartsman in [55], where a new complexity measure for fault tolerant PRAM algorithms was defined, where

the notion of parallel robustness was introduced, and where the *Write-All* problem was defined.

The techniques presented in [55] can readily be employed in making arbitrary PRAM algorithms fault-tolerant. The iterated *Write-All* paradigm was employed (independently) by Kedem et al. in [59] and by Shvartsman in [92] to extend the results of [55] to arbitrary PRAM algorithms (subject to fail-stop errors without restarts). In addition to the general simulation technique, [59] analyzes the expected behavior of several solutions to *Write-All* using a particular random failure model. The algorithms analyzed included algorithms from [55] and [75], and a new algorithm based on pointer doubling that has a good expected behavior for the failure model defined. The deterministic execution of PRAM algorithms in [92] is optimal for any adversary when parallel slackness (as in [99]) is exploited to our advantage.

Asynchronous versions of the PRAM is a subject of recent research. Various means of relaxing the strict synchronization requirements of the standard PRAM have been used to show that efficient algorithms can be efficiently executed on asynchronous models [29, 31, 45, 78, 75].

A simple randomized algorithm that serves as a basis for simulating arbitrary PRAM algorithms on an asynchronous PRAM is presented by Martel et al. in [75]. This randomized asynchronous simulation has very good expected performance for the *Write-All* problem when the adversary is off-line. Other algorithms in this model are given by Martel et al. in [72] and Martel and Subramonian in [73]. Kedem et al. [61] further refined the results in [59] to produce an approach that leads to constant expected slowdown of PRAM algorithms when the power of the adversary is restricted. The fail-stop deterministic lower and upper bounds of [55] were also improved in [61] by $\log\log N$ factors. Recently, Kedem et al. [60] further investigated the use of randomization for resilient parallel computation. Martel [71] has improved the analysis of the main algorithm in [55] by a $\log\log N$ factor. This improvement also leads to the upper bound that matches the lower bound in [55] under the *memory snapshot* assumption as we show in this work.

A parallel algorithm animation tool was developed by Apgar [9] to aid in the analysis of *Write-All* algorithms using Stasko's [95] TANGO animation system.

Our modeling of fault tolerance where a processor is an entity subject to failures has some similarities with the design of "robust" sorting networks using fault-prone

switches, as those of Rudolph in [88], and in general with the design of reliable systems from unreliable components, as done by Pippenger in [82] using gates or by Dwork et al. in [38] for networks. The notion of robustness that we target in this research differs from that of the sorting network in [88], and in that network a linear number of operations is still critical. Another example is the emulation of PRAMs on faulty hypercubes. See the the recent result of Aumann and Ben-Or on high probability emulation [12].

Interesting impossibility results for asynchronous shared memory models are given by Herlihy in [47, 48]. General synchronous PRAM simulations are impossible using *bounded* resources on asynchronous PRAMs. Buss et al. [27] show that some deterministic computations can be performed using subquadratic work, even when arbitrary asynchrony of PRAM processors is allowed. Anderson and Woll [8] also showed an efficient randomized solution for *Write-All*, as well as the existence of *Write-All* solutions with work $O(N^{1+\varepsilon})$ for $P = N$ and any $\varepsilon > 0$ that can be used with the models we define here.

Finally, our work here deals with dynamic patterns of faults; for recent advances on coping with static fault patterns, for example, are addressed by Kaklamanis in [54]. The granularity of faults in our work is at the processor level; for recent work on gate granularities see [11, 82, 88].

1.3.2 Fault-tolerant distributed computation

Adding fault tolerance to algorithms is the subject of significant current research in the qualitatively different setting of dynamic asynchronous network protocols (recent results and an overview of this area is well represented by [3, 4, 13, 15]).

The general problems encountered in fault-tolerant parallel computation and in particular the problems of allocating active processors to tasks have similarities to the problems of resource management in a distributed setting. Distributed controllers have been developed for resource allocation in network protocols, where the total number of messages sent is the resource controlled. For instance, the algorithms of Lynch et al. [70] (with a probabilistic setting) and of Awerbuch et al. [4] (with a deterministic setting) are among the most sophisticated in that area. The problem we address in this thesis is, at an intuitive level, one of controlling resource allocation. The resource controlled is all available PRAM processor steps, and the reason we are forced to control it, is the

requirement to complete the computation in the presence of faults. Note that unreliable PRAM processor steps must control all available PRAM processor steps. This introduces difficulties that recall the presence of network changes in [3, 13, 15], i.e., dynamic changes of the computation medium. Fault tolerance of particular network architectures is also studied in [38]. However, the distributed computation models, the algorithms, and their analysis are quite different from the parallel setting studied here.

It is interesting that the concept of a "communication complexity controller" first developed for distributed computing has an analog in parallel computing, i.e., "an algorithmic transformation that guarantees robustness". Note that the parallel setting is simpler to define and has easier to describe solutions, immediately applicable to a large body of existing work on parallel algorithms.

Parallel computation in the setting where the shared memory is initially contaminated has some similarities with the notion of a *self-stabilizing system* introduced by Dijkstra in [34]. Paraphrasing [34], a system is self-stabilizing if and only if, regardless of the initial state the system can always make a state transition into another state, and the system is guaranteed to find itself in a legitimate state after a finite number of transitions. Our computations using initially contaminated memory can be viewed as self-stabilizing with respect to the state of shared memory. In order to describe our technical contributions we must now review the state-of-the-art of the algorithmics of *Write-All*. For the most recent results in the area of distributed self-stabilizing systems see the works of Awerbuch et al. [14, 16].

Finally, the synchronous parallel setting with fail-stop processor errors is free from the limitations inherent in the asynchronous environment, or the situations where the processors can perform malicious actions (see [41, 69, 79] for surveys of the topic, and [37, 42, 43] for lower bounds results).

1.3.3 Technology for fault tolerance

Several engineering and technological approaches exist to implementing parallel systems that enable them to operate correctly when they are subjected to certain failures. Although these research and engineering areas are not as directly relevant to our research as the work cited earlier, they are nevertheless extremely important. The methods and technologies summarized 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.

Fault-tolerant memories

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. The three main techniques used in providing memory fault-tolerance are: (1) Coding: in addition to the bits being stored, this technique utilizes additional (parity) bits in conjunction with various error detecting and/or correcting codes (see McEliece [76] for a grand tour). (2) Replication or shadowing: two or more copies of the memory are maintained with either the majority vote being taken, or the faulty units being shut off in a hybrid approach. (3) Reconfiguration: spare memories are used to replace faulty units by reconfiguring memory units. The survey of Sarrazin and Malek [89] covers these techniques that are used to make memory (cache and main) more reliable without appreciably degrading its performance.

Robust interconnection networks

Another important subject that has been the target of work is the area of fault-tolerant interconnection networks. Interconnection networks are typically used in multiprocessor systems to provide communication among processors, memory modules and other devices [52]. An encyclopaedic survey of the interconnection networks is given by Almasi and Gottlieb in [6, Chapter 8]. Theoretical foundations for such networks are summarized by Pippenger in [83]. The networks are made more reliable by employing redundancy. A survey of fault tolerant interconnection networks is presented by Adams et al. in [2]. An interesting interconnection network routing strategy was described by Preparata [85], in which fast routing is achieved by allowing for some messages to be lost and using a redundancy scheme [84, 86] to reconstruct lost information.

The area of fault tolerance and efficiency of interconnection networks is extremely important as an enabling technology for fault-tolerant parallel computation. In this thesis, we limit our work to the design of algorithmic techniques that assume that a robust interconnection medium such as those surveyed in [2] is available.

Fault tolerance in special purpose parallel computers

Systolic arrays are special purpose parallel computers that lend themselves to being engineered with fault-tolerant features. Algorithm-based fault-tolerant techniques are designed for specific algorithms that are implemented as systolic arrays, and where occasional and intermittent failures are expected. For example such techniques are used for various calculations on, or with matrices. Typically a limited number of faults can be handled by systems that utilize various checksumming methods to locate faults that caused incorrect values to be computed, and then reconstruct the correct values.

Reconfigurable VLSI-based arrays are used when permanent faults (i.e., due to fabrication defects) are the primary concern. The arrays are manufactured with spare modules, such that a number of failures can be tolerated by detecting faulty modules and either bypassing them or automatically replacing them with the spare modules. The survey of Abraham et al. [1] overviews the algorithm-based fault tolerance in systolic arrays and reconfigurable VLSI-based systolic arrays. Relevant theoretical bounds are given by Kaklamanis et al. in [54].

In his thesis, Hughey [50] presents a programmable systolic array, and he also describes several techniques used for on-board fault detection along with software techniques that enable the bypassing of certain processing element failures.

In the next section we apply some of the technologies cited above to show an example of a realizable system that is consistent with the models we study in this work.

1.4 Relation to Physical Systems

The abstract models of parallel computation we present and study must be able to reflect or capture the characteristics of actual systems.

Processor delay is a feature of any multi-programming environment, in which processing priorities are not centrally or predictably specified. A processor may be temporarily or permanently suspended due to an external event, and processing resources may be unexpectedly required by another task as determined by the underlying system. In the synchronous parallel environment that we study, a processor delay is treated as a processor failure subject to a possible subsequent restart.

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. If a processor discontinues its operations due to an internal or external event for the duration of a computation, we must assure that the computation in progress will proceed to a successful completion.

Communication delay is a well-known aspect of multi-component systems when information available at certain components is needed by other components. Small communication delays can be consistent with a system that is designed to operate synchronously. On the other hand, unpredictable or non-uniform delays introduce additional complexities to the design of algorithms. In the synchronous parallel setting we assume that the communication delay is uniform for all processors. This will allow for the complexity measures to be meaningfully applied when the communications delay is a function of the size of the task and the number of processing elements.

Communication failure may be due to memory failures or as the result of memory operations by other processors. If the communication network reports the failure of an operation, the processor can re-attempt the access, and the situation can be modeled 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.

In this work, we treat delay and failure as occurring to the processors only. If memory operations are atomic and synchronous, they may be assumed to be instantaneous, and the communication delays or failures may be attributable to the processor, and accountable at the processor level of abstraction.

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 we formalize in Sections 2.5 and 2.6.

Engineering and technological approaches exist that allow implementing electronic components and systems that operate correctly when subjected to certain failures as was

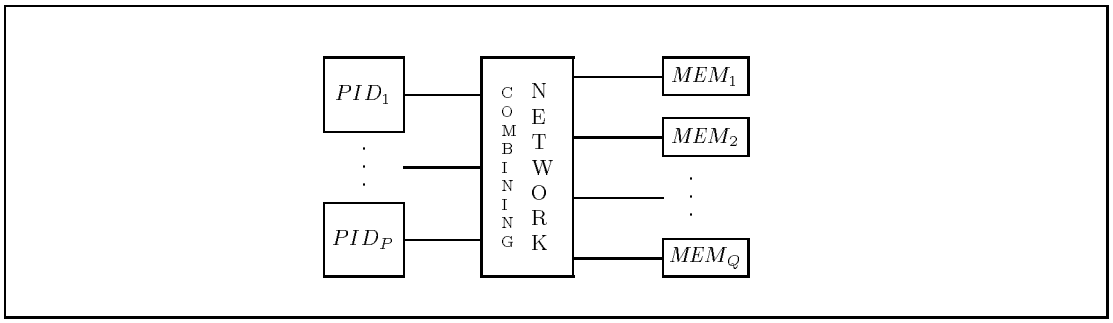


Figure 1.1: A robust fail-stop multiprocessor.

overviewed in the previous section (for surveys and examples, see [33, 51, 53]). We will now cite the particular technologies that are instrumental in providing basic hardware fault tolerance and employ these technologies in 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 [89]. 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 [91]. 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 [2]. A *combining* interconnection network that is perfectly suited for implementing synchronous concurrent reads and writes is formally treated in [62] (the combining properties are used in their simplest form only to implement concurrent access to memory). Finally, fail-stop processors are formally presented and justified in [90].

The abstract model that we are studying can be realized (Figure 1.1) in the following architecture, using the components just cited:

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.

If there is a cost L associated with reading or writing a single shared memory cell, then the work complexity of the algorithms and the simulations that we studied increases by the factor L.

1.5 Structure of the Document

The rest of this thesis is structured as follows. Chapter 2 defines and motivates the models employed by this research, the associated measures of complexity, the models of failure, and the key *Write-All* problem. Chapter 3 contains the definitions and the analysis of fault-tolerant parallel algorithms using three processor allocation paradigms: global allocation, local allocation and hashed allocation. In Chapter 4 we address the lower bounds under the memory snapshot assumption. In Chapter 5, the building blocks of the previous chapters are used to implement a general simulation of parallel algorithms. There we also discuss of improvement to the oblivious simulations. In Chapter 6 we solve the *Write-All* problem when the shared memory is contaminated and we eliminate the requirement of atomic writes of logarithmic number of bits. We conclude with a discussion in Chapter 7.

The bibliography is followed by three appendices. Appendix A contains the detailed pseudocode for algorithm W and two lemmas. Appendix B contains the pseudocode for algorithm X. Appendix C is reserved for mathematical lemmas used in the lower bounds proofs.

Chapter 2

Models and Definitions

MODELING parallel computation and processor failures must go hand in hand with the study of algorithms and their complexity. In this chapter we define the base models of the computation that are the subject of our research, the models of failure that we are studying, the two major variations of the fail-stop parallel random access machine and the rationale behind the technical decisions that we made. We discuss the definitions of the complexity measures that characterize the efficiency of algorithms for the models selected and in the context of particular failure models. We also formalize the key *Write-All* problem.

2.1 Base Model of Computation

We study fault-tolerant algorithms for the closely coupled synchronous shared memory multiprocessor systems where the processors need to cooperate in working towards a common computational goal. Specifically, we study algorithms for the systems that can be modeled by the Parallel Random Access Machine (PRAM) of Fortune and Wyllie [44].

The PRAM model is used widely in the parallel algorithms research community as a convenient and elegant model, and a wealth of efficient algorithms exist and are continually being developed for this model. The surveys of Eppstein and Galil [40] and Karp and Ramachandran [58] cover all of the important variations of the PRAM model, and give most of the fundamental PRAM algorithms. Instead of reiterating the rationale for studying the PRAM model and listing the variations of the PRAM models,

we refer the reader to the two excellent surveys [40, 58] that succinctly address the topic in the respective introductory sections. For the base model in this work, we use the following definition of the PRAM [44]:

1. There are P initial processors with unique identifiers (PID) in the range $1, \ldots, P$. Each processor has access to its PID, and the number of processors P.

2. The global memory accessible to all processors is denoted as **shared**, each processor also 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 size N.

We use the concurrent read, concurrent write (CRCW) variation of the PRAM, in which multiple processors can concurrently read or write to the same memory location. In the algorithms we present, all concurrently writing processors write the same value making our approach independent of the CRCW conventions for writes.

Our algorithms are described in a model independent fashion using a consistent high level notation with the obvious **forall/parbegin/parend** parallel construct. Such high level notation can be formalized as a programming language that can be compiled using standard compilation techniques and the techniques specific to PRAMs as discussed by Wyllie [102].

2.2 Measures of Efficiency

Computation speed-up is one of the central reasons for using parallel computers. In this section, we introduce and discuss a particular way of flexibly factoring fault tolerance into the conventional definition of parallel work. This definition can be adapted for the particular failure models that we examine in later sections.

If a task can be done in time T using a single processor, we would like to perform the same task in parallel time $\tau = T/P$ using P processors. This optimal linear speed-up is one of the important goals of parallel algorithm design.

More formally, let *parallel work* be the product of the number of processors P and the parallel time τ. Parallel algorithms are considered optimal when the parallel work is within a multiplicative constant of the best known sequential time (of course for the computation to be practical, the constant must be small). Even if not optimal, parallel algorithms are generally considered efficient, if they attain a near linear speed-up. That is, using P processors on inputs of size N, the parallel time achieved is $T(N)/P$ (within a multiplicative factor polylogarithmic in N), where $T(N)$ is the best known sequential bound and P ranges over $1, \ldots, N$.

Efficient or optimal parallel algorithms have been developed for many fundamental computation tasks such as manipulating integers (e.g., add or sort N integers), manipulating lists and trees (e.g., compute the rankings of the elements of a N-size list, and compute a preorder numbering or subtree sizes, ..., of a N-size tree). These algorithms play an important role in realizing the promise of high speed-ups using massive parallelism.

Unfortunately, the quest for high speed-ups has led to efficient parallel algorithms that are very tightly designed, so that every processor is fully utilized doing something essential for resolving the input task. Thus, *parallel algorithm efficiency* implies a *minimization of redundancy* in the computation that leaves very little room for fault tolerance. It is interesting to note that most of the known efficient parallel algorithms do not terminate correctly or become quite inefficient if they are perturbed by simple processor errors. These perturbations are of course outside the original setting, but are nonetheless realistic.

Once processor failures are introduced into a parallel computation the $\tau \times P$ measure is no longer that relevant. This is because the computational resource is no longer under the control of computation — it varies due to failures, and only limited resources may be available at any given time. The efficiency of fault-tolerant parallel computation is more appropriately measured in terms of the processor work steps that are available to the computation.

Consider a computation with P initial processors that terminates in parallel-time τ after completing its task on some input data I of size N and in the presence of fail-stop error pattern F. If $P_i(I, F) \leq P$ is the number of processors completing an instruction at step i, then we define the following measure: $S = S(I, F, P) = \sum_{i=1}^{\tau} P_i(I, F)$.

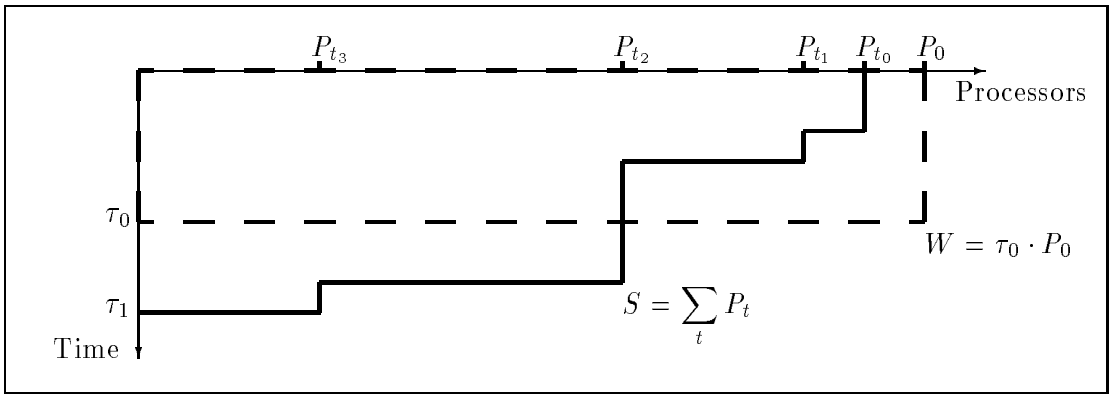


Figure 2.1: Work in the absence (W) and presence (S) of processor failures.

Example 2.1 *Work subject to failures:* Consider a P_0-processor algorithm that terminates in time τ_0 in the absence of failures. The fault-free work $W = \tau_0 \cdot P_0$ is the area of the dashed rectangle in Figure 2.1. Due to failures the algorithm begins with $P_{t_0} < P_0$ processors, and then the number of active processors is reduced to P_{t_1}, P_{t_2} and P_{t_3} at times t_1, t_2, t_3 respectively until the computation terminates at time τ_1. The work in the presence of these failures is more appropriately described by the area bounded by the "staircase" solid lines and the two axis. This area is S. □

S is used as the basis for measuring efficiency of fault-tolerant algorithms. The algorithms are studied in the context of the chosen failure models and in conjunction with natural measures of efficiency. We use S to define *available processor steps* (S^*) and the notion of *robustness* for the fail-stop no-restart model in Section 2.5, and we use S to define *completed work* (S^+) and *overhead ratio* (σ) for the restartable fail-stop model in Section 2.6.

The use of generalized parallel work as the primary complexity measure has the additional benefits of being able to compare meaningfully the efficiency of fault-tolerant and non-fault-tolerant algorithms. For example, the well-known class $\mathcal{NC}$ of algorithms characterizes efficiency primarily in terms of (polylogarithmic) time efficiency, even if the computational agent is large (polynomial) relative to the size of a problem [30, 81]. To characterize better the efficiency of parallel algorithms, the efficiency measures need to take into account both the parallel time and the size of the computational resource, i.e., parallel work. Such characterizations of parallel algorithm efficiency are defined by Vitter and Simons in [100] and expanded on by Kruskal et al. in [63].

2.3 The *Write-All* Problem

In order to deal with failures, it is necessary for the correct processors to detect the failures and reschedule the work of the failed processors. The main problem here is that the minimization of redundancy in the computation does not leave many resources for *failure detection* and *load rescheduling*. It is fairly easy to see that naïve processor failure detection and reassignment strategies, e.g., use of a master control or clustering of processors, are inadequate. A master control strategy is sensitive to particular patterns of simple failures. Clustering can degrade the performance, measured as the worst case work S by a linear or greater multiplicative factor. Let us illustrate this discussion with an example.

Example 2.2 *Write-All:* One of the simplest tasks performed by a PRAM is: *given a zero-valued array of N elements and P processors, write value 1 into each array location.* We call this task *Write-All*. When $P = N$, this *Write-All* problem is trivially solved in constant time by the following (PRAM) program. However, even a single processor failure will prevent the establishment of $\{x[i]=1\ (i=1, \ldots, N)\}$ as the postcondition.

```
forall processors PID = 1..N parbegin
    shared integer array x[1..N];
    x[PID] := 1
parend
```

Simple fixes are available that will make the above program more fault-tolerant. For example, for a small number of failures $< k$, consider the clustering algorithm below. This algorithm performs well for dynamic failure patterns with few errors, but poorly if there are many failures. For example, if k is fixed and N variable then it cannot handle $N/2$ failures, and if k is allowed to grow to $N/2$ then S becomes quadratic in N.

```
forall processors PID = 1..N parbegin
    shared integer array x[1..N];
    for i = PID to PID + k do
        if i ≤ N then x[i] := 1 else x[i − N] := 1 fi
    od
parend
```

□

We will show the existence of an N-processor CRCW PRAM algorithm for the *Write-All* problem of Example 2.2, for which $S = O(N \log^2 N)$ for any pattern of fail-stop processor failures. This solution also illustrates the notion of *robustness*. The original *Parallel-time* $\times$ *Processors* product, N, is increased by at most a polylogarithmic in the size of the input multiplicative factor, for any dynamic pattern of failures with at least one surviving processor. Note that there is no knowledge of how many, when, or which processors will fail.

Our techniques for deriving fault-tolerant and efficient algorithms are based on robust solutions for the *Write-All* problem, and so we grace it with its own name:

Definition 2.1 Given a zero-valued array of N elements and P fail-stop PRAM processors, the *Write-All* problem is to set each element of the array to 1. □

Remark 2.1 The main point of *Write-All* problem is not initializing an array of N elements, but transforming the contents of shared memory from one state to another. *Write-All* is formulated to capture the computational progress that can be naturally accomplished in unit time by a PRAM in the absence of failures. *Write-All* can be equally defined in terms of computing the absolute values of elements of an array, or copying to the target array values from another array.

2.4 Models of Failure

We now present several dimensions of failure modeling and the models we are concerned with in this work. Significant research in the past decade was devoted to the study of fault-tolerant, distributed computation in the presence of arbitrary (and even malicious) system faults. Byzantine and other simpler failures were extensively studied in the context of distributed algorithms for the *consensus* problem (e.g., Pease et al. [79, 80], also see a survey by Fischer [41]). The failure models for parallel computation are constructed from failure definitions along the spectrum of failure models discussed below.

Types of failures, such as byzantine, omission failures, fail-stop failures, etc., have been part of the consensus literature as distributed algorithms were being developed to deal with failures (e.g., as discussed by Lamport and Lynch in the survey [66]). In the area of parallel computation, where processors are more tightly coupled as compared

to a distributed environment, failures need to be classified further with the emphasis placed on the more benign fail-stop case. For example, the synchronous parallel setting with fail-stop processor errors is free from the limitations inherent in the asynchronous environment, or the situations where the processors can perform malicious actions (see [79, 41, 69] for surveys, and [42, 43, 37] for lower bounds).

Using state of the art technology, processing elements are being designed with built-in diagnostics capabilities. Upon detecting failures, such processors can isolate themselves from the rest of the computing environment without harmful effects. Such processors are modeled as fail-stop processors. It was shown by Schlichting and Schneider in [90] that using a formal methodology and an appropriate programming language framework, it is possible to construct correct algorithms for fail-stop processors. In this work we in turn show that it is possible to construct efficient and fault-tolerant algorithms for certain classes of parallel fail-stop processors. In the context of closely coupled parallel computation the fail-stop failures are both accurate and tractable.

Adversaries: The notion of adversary is useful and important for the study of parallel computations under different models of failure. An adversary determines which processors can fail at what step of the computation and which errors can be caused by the failures. When the adversary is *omniscient*, it has complete knowledge of the computation. The adversary can be restricted to have *time or space limited knowledge* of the actions (e.g., those performed in the past or just in a subset of the processors). When the adversary is in addition *on-line*, it can decide during the computation what processors will fail, as in this work. Alternatively, the adversary can be *off-line* as in Martel et al. [75], in which case all failure decisions are made prior to the start of a computation. Finally, an adversary might be limited probabilistically as in Kedem et al. [59], where the faults are occur with certain probability. Here we deal with the omniscient on-line adversaries.

Granularity: Whereas, there is a fair amount of analysis based on types of failures and kinds of adversaries, there has been less attention paid to granularity of failures. For parallel systems granularity seems to be a key concept. Failure granularity defines the extent to which sub-system failures affect the overall system. Granularity also defines the smallest system components, such that a failure within the component is either completely masked by the component or causes the failure of the entire component. In practice, many parallel programs are implemented using *threads* packages [22, 36]. It is

also reasonable to study failure granularity at the level of a single thread.

Our modeling of fault tolerance has some similarities with the design of "robust" sorting networks, as those of Rudolph [88], and in general with the design of reliable systems from unreliable components, as in Pippenger [82] or Dwork et al. [38]. The distinguishing characteristic of our approach is the investigation of fault tolerance at the processor granularity as opposed to gate or switch granularities [82] and [88] respectively.

Magnitude: Many hardware oriented fault tolerance techniques provide fault masking up to a pre-determined limit. In a distributed setting, some algorithms can handle processor failures when the number of failures does not exceed a certain fraction of the total number of processors. It is important to develop techniques that can deal with *any* number of failures, however such techniques should also yield good results when the number of failures is relatively small. The efficiency of fault-tolerant solution may depend on the maximum allowable number of failures, but it is imperative that computations remain correct for any number of failures (when one or more processors remains operative). Here we study arbitrary failures and arbitrary failures and restarts.

Recovery: In some models it is reasonable to assume that faulty processors never recover. For example, manufacturing defects may permanently disable some of the systolic array processors, while the array remains functional when equipped with on-board fault-tolerance [1, 28, 50]. It is also reasonable for processors to recover at some point and rejoin a computation in progress. Failures may be quantified by the duration of a processor's absence from a computation. We consider both the no-restart and restartable models.

Frequency: A final but significant dimension is the frequency and timing pattern of failures. Assumptions about failure frequency must underlie any probabilistic analysis. In addition, we believe that the fault tolerant algorithm must show graceful degradation of performance so that when the failures are infrequent the algorithms must be near the peak of their efficiency. In this work we place no restriction on the frequency of failures.

In the next sections we define two variations of the PRAM whose processors are subject to stop failures. The two models are:

1. The fail-stop PRAM, where the processors do not restart after a failure, and
2. The restartable fail-stop PRAM, where the processors can restart after a failure.

We study fail-stop processor errors [90] that are determined by the worst case omniscient on-line (adaptive) adversary that is not limited as far as the frequency and magnitude of errors are concerned.

In each of the models, the patterns of processor failures will be specified as sets of triples $<tag, \text{PID}, t>$ where *tag* is label indicating the type of an event (i.e., **failure** or **restart**), PID is the processor identifier, and t is the time instance indicating when the event occurs. The *size* of the failure pattern F is defined as the cardinality $|F|$.

2.5 No-restart Fail-stop CRCW PRAM

We begin with the PRAM model given in Section 2.1. This model is extended with a failure model, and a complexity measure that captures the work of a fault-tolerant algorithm when its processors are subject to failures.

2.5.1 Failure model

The fail-stop CRCW PRAM extends the basic model by allowing processor failures. The failure model for the fail-stop CRCW PRAM is defined as follows:

1. We allow *any dynamic pattern* F of processor fail-stop errors provided one processor survives (one processor is necessary if anything is to be done). F describes which processors fail and when. This pattern is determined by an *adversary*, who knows everything about the structure and the dynamic behavior of the algorithm.

2. We only consider *fail-stop* (no restart) behavior: processors fail by stopping and not performing any further actions. Fail-stop models are reasonable approximations of what is desirable and achievable in practice [90].

3. We assume that the *shared memory writes* of the individual PRAM steps are *atomic* with respect to failures: failures can occur before or after a shared write of $O(\log \max\{N, P\})$ bit words, but not during the write. This non-trivial assumption is made only for simplicity of presentation. Algorithms using this assumption can be automatically converted to use only single bit atomic writes as we show in Section 6.2.

The failure patterns are syntactically defined as follows:

Definition 2.2 A fail-stop no restart *failure pattern* F is a set of triples $<tag$, PID, $t>$ where *tag* is **failure** indicating processor failure, PID is the processor identifier, and t is the time indicating when the processor stops or restarts. □

Remark 2.2 Since at least one processor must survive if a computation is to terminate, we need only consider failure patterns F of size $|F| < P$ for computations with P initial processors.

2.5.2 Measure of efficiency: available processor steps

We define the complexity measure of *available processor steps* defined using the measure introduced in Section 2.2. This measure generalizes the standard *Parallel-time* $\times$*Processors* product. As we discussed earlier, this measure appropriately evaluates the efficiency of fault-tolerant parallel algorithms. We formally define S as follows:

Definition 2.3 Consider a computation with P initial processors that terminates in parallel-time τ after completing its task on some input data I of size N and in the presence of fail-stop error pattern F. If $P_i(I,F) \leq P$ is the number of processors completing an instruction at step i, then we define $S(I,F,P)$ as:

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

We now define *available processor steps* S^* in terms of S:

Definition 2.4 A P-processor PRAM algorithm on any input data I of size $|I| = N$ and in the presence of any pattern F of failures of size $|F| \leq M < N$ uses *available processor steps*

$$S^* = S^*_{N,M,P} = \max_{I,F}\{S(I,F,P)\} \; . \; \square$$

From the definition of S, we immediately observe the following property:

Property 2.5 Given any fault-tolerant parallel algorithm that uses up to P processors, on inputs of size N, if $M_1 < P_1 \leq P_2 \leq P$ and $M_2 < P_2 \leq P$, then $S^*_{N,M_1,P_1} \leq S^*_{N,M_2,P_2}$.

This is so because by Definition 2.4, when the smaller number of processors P_1 is used, we maximize over the failure patterns with $P_2 - P_1$ failures at time zero.

The measure S^* is used in turn to define the notion of algorithm *robustness* that combines fault tolerance and efficiency:

Definition 2.6 Let $T(N)$ be the best sequential (RAM) time bound known for N-size instances of a problem. We say that a parallel algorithm for this problem is a *robust parallel algorithm* if: for any input I of size N and for any number of initial processors P $(1 \leq P \leq N)$ and for any failure pattern F of size M with at least one surviving processor $(M < N)$, this algorithm completes its task and it has $S^* = S^*_{N,M,P} \leq c\ T(N) \log^{c'} N$, for fixed c, c'. □

Remark 2.3 Note that the available processor steps S^* defines the worst case efficiency of a computation. In some cases it may be sufficient for a computation to be efficient with high probability, and allow worst case inefficiency. Such an approach was taken in [59] using a probabilistically restricted adversary. In an analogous fashion it is possible to modify the definition of "robustness" based on the failure model used.

2.5.3 Discussion of the technical choices made

Fail-stop errors vs. malicious processor behavior: We have chosen to consider only the failure models where the processors do not write any erroneous or maliciously incorrect values to shared memory. While malicious processor behavior is often considered in conjunction with message based systems, it makes less sense to consider malicious behavior in tightly coupled shared memory systems. This is because even a single faulty processor has the potential of invalidating the results of a computation in unit time, and because in a parallel system all processors are normally "trusted" agents, and so the issues of security are not applicable.

Concurrent writes vs. exclusive writes: The choice of CRCW (concurrent read, concurrent write) is justified in the discussion of lower bounds in Chapter 4, where a simple result shows that the CREW (concurrent read, exclusive write) model does not admit fault-tolerant efficient algorithms.

Clear vs. contaminated initial memory: We require that a linear amount of shared memory location be initially clear, i.e., initialized to zero. While this is consistent with

definitions of PRAM such as [44], it is nevertheless a requirement that fault-tolerant systems ought to be able to do without. We address this issue in Section 6.1 where we develop an efficient procedure that solves the *Write-All* problem even when the shared memory is *contaminated*, i.e., contains arbitrary values.

Atomicity and word size: Thus far, the model we have defined assumes the ability to perform $\log N$-bit word parallel writes atomically. That is the model allows: (1) $\log N$-bit words to be written in unit time, and (2) the adversary could cause failures either before or after the write cycle of the PRAM, but not during the write cycle. The algorithms in these models can be modified so that these two restrictions are relaxed.

The new definition of atomicity becomes: (1) $\log N$-size words are written using $\log N$ bit write cycles, and (2) the adversary can cause arbitrary fail-stop errors either before or after the *single bit write cycle* of the PRAM, but not during the bit write cycle. We formally show this in Section 6.2.

2.6 Restartable Fail-stop CRCW PRAM

We begin with the PRAM model given in Section 2.1. This PRAM model is first augmented with the concept of an *update cycle*.

In all our algorithms for the restartable fail-stop PRAM:

- 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 (number of read and write instructions) are fixed, but depend on the instruction set of the PRAM. The values quoted (4 and 2) are sufficient for our exposition.

We next define the model of failures and restarts model, and two natural complexity measures.

2.6.1 Failure and restart model

We use the *fail-stop with restart* failure model, where time instances are the PRAM 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* (see Remark 2.5 below).

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 formally defined as follows:

Definition 2.7 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.

Remark 2.4 The failures and restarts we are considering are different from the errors of omission, e.g., where processors may skip a step but preserve their local context.

Remark 2.5 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 we show in Section 6.2.

2.6.2 Measures of efficiency: completed work and overhead ratio

We investigate two natural complexity measures, completed work and overhead ratio. The completed work measure generalizes the standard *Parallel-time* $\times$ *Processors* product and the available processor steps of Definition 2.4. The overhead ratio is an amortized measure.

Definition 2.8 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 2.9 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* $S^+ = S^+_{N,M,P} = \max_{I,F}\{S'(I,F,P)\}$ and

(ii) has *overhead ratio* $\sigma = \sigma_{N,M,P} = \max_{I,F}\left\{\dfrac{S'(I,F,P)}{|I|+|F|}\right\}$. $\square$

Remark 2.6 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 2.7 For the fail-stop no-restart execution of restartable algorithms, the measures S^* and S^+ are equal asymptotically. When the restarts do not occur, then the maximum work spent in the incomplete cycles is bounded by $O(P)$, since there can be no more than P failures. Therefore, for the fail-stop no-restart model, using the work S^* yields the same results as using the S^+ measure. The only difference is that S^* accounts abstract instructions, while S^+ accounts update cycles that might contain a small constant number of instructions.

Remark 2.8 Consider the definition of work $S(I,F,P)$ (Definition 2.3) that accounts for the incomplete update cycles. Clearly $S(I,F,P) \leq S'(I,F,P) + c|F|$. Thus, using S does affect asymptotically the measure of work (when $|F|$ is very large), but it does not asymptotically affect σ is given in Definition 2.9(ii).

Remark 2.9 One might also generalize the overhead ratio in terms of $\frac{S'(I,F,P)}{T(I)+|F|}$, 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 the *Write-All* problem (by itself and as used in the general simulation) $T(I) = \Theta(|I|)$.

Remark 2.10 Another way to generalize the overhead ratio is in terms of $\frac{S'(I,F,P)}{\tau_P(I)\cdot P+|F|}$, where $\tau_P(I)$ is the parallel time complexity of the best P-processor solution known to date for the particular problem at hand. Again, 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 the *Write-All* problem (by itself and as used in the general simulation) $\tau_P(I) \cdot P = \Theta(|I|)$.

2.6.3 Discussion of the technical choices made

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 2.3 *Work subject to a large number of recoveries:* Consider a *Write-All* solution, where it takes a processor one instruction to recover from a failure. If an adversary inflicts a failure pattern F with the number of failure/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, 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 2.9, 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 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 non-restartable fail-stop model 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 be unbounded. Since the processors lose their context after a failure, they have to read something to regain it. Without at least one update cycle completing, the adversary can force the PRAM to thrash by allowing only these reads to be performed. Similar concerns are discussed in [90].

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 (in Remark 2.9) of any algorithm that simulates a single N processor PRAM step is at least $\Omega(P \cdot N)$. The reason for this quadratic behavior is the following simple and rather uninteresting *thrashing* adversary.

Example 2.4 *Thrashing adversary:* Let ALG be any algorithm that solves the *Write-All* problem under the arbitrary failure/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 $\Theta(P \cdot N)$. $\square$

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 sub-quadratic solutions.

Chapter 3

Write-All Algorithms

DEMONSTRATING the existence of efficient algorithms for the *Write-All* problem given in Definition 2.1 is essential for the general simulation we develop and analyze in Chapter 5. In this chapter we present and analyze several algorithms for the *Write-All* problem using three processor allocation paradigms.

In the chapter and in the detailed description of the algorithms in Appendices A and B we assume that N is a power of 2. Nonpowers of 2 can be handled using conventional padding techniques. All logarithms are to the base 2, and **div** stands for integer division with truncation.

3.1 Processor Allocation Paradigms

Processor allocation is often the key problem on the way to achieving efficient solutions. Processors need to be allocated so that the relative processor loads are balanced according to appropriate criteria. We present three processor allocation paradigms used in constructing efficient solutions for the *Write-All* problem.

In the overview of the processor allocation paradigms and algorithms within each paradigm we give the complexity results in terms of N, the size of the *Write-All* array.

Global allocation paradigm

The allocation of processors in the global allocation paradigm is performed using the knowledge of the global state of the computation. The processors compute and reduce the information that is in turn used to synchronize and allocate processors. We present two deterministic algorithms that use global allocation paradigm:

Algorithm W: this is a deterministic fail-stop no restart algorithm for which $S^* = O(N \log^2 N / \log\log N)$; this algorithm has an optimal range of processors for which the work of the algorithm is optimal for any pattern of failures for $P \leq N/\log^2 N$.

Algorithm V: this is an algorithm that can be used with both models; in the non-restartable model it has $S^* = O(N \log^2 N)$ and a range of optimality similar to that of algorithm W, and for the restartable fail-stop model it has $S^+ = O(N \log^2 N + M \log N)$, where M is the size of the failure pattern encountered during the execution.

Local allocation paradigm

Here processors make allocation decisions based on the information that is local to the processors, or that is immediately available in constant number of memory accesses from the shared data structures. No global synchronization is necessary, and therefore local allocation algorithms can also be used with asynchronous systems (as shown by Buss et al. in [27]). We present and analyze a deterministic algorithm, and we present (without analysis) two randomized algorithms based on the deterministic algorithm:

Algorithm X: this is a deterministic algorithm that can be used in both the fail-stop and restartable models. In the fail-stop model we show and analyse a particular failure scenario (this scenario was refined by López-Ortiz [68] to exhibit the known worst fail-stop work for algorithm X). For the restartable model we provide complete algorithm analysis and show that for any pattern of failures and restarts the algorithm has subquadratic work $O(N^{1.59})$ and efficient overhead ratio $\sigma = O(\log^2 N)$ (when interleaved with algorithm V).

Algorithms X_{coin} and X_{die}: these are two variations of algorithm X using coin tossing and die casting respectively. The analysis of these algorithms is an open question.

Hashed allocation paradigm

Here processors are allocated in a hashed fashion, either according to a randomized scheme or using a deterministic scheme that approximates a particular randomized scheme. Hashed allocation algorithms can be used in both restartable and non-restartable failure models.

Algorithm Y: this is an efficient determinization of a randomized algorithm that was defined by Anderson and Woll in [8]. We present this algorithm without analysis. Some experimental work suggests that the algorithm is a very efficient algorithm. The analysis of algorithm Y is stated as an open problem that shows an interesting linkage between group theory, combinatorics and multi-processor scheduling.

3.2 Global Allocation Paradigm

This section consists of three parts. In the first two we present two algorithms using the global allocation paradigm: algorithm W and algorithm V. In the final part we define the *processor allocation monotonicity* property and show that these two algorithms have this property. This property will enable us to develop fault-tolerant simulations for PRIORITY PRAMs in Chapter 5.

3.2.1 Algorithm W

We now define and analyze a robust parallel algorithm for the *Write-All* problem in the fail-stop no-restart model. We call it algorithm W. This solution illustrates the notion of *robustness* in the fail-stop model. The original *Parallel-time* $\times$ *Processors* product, N, is increased by at most a $c \log^2 N / \log\log N$ multiplicative factor, for any dynamic pattern of failures with at least one surviving processor. Note that we have no knowledge of how many, when, or which processors will fail.

Algorithm W solution for the *Write-All* problem is based on a parallel *loop* through (1) a failure detecting phase, (2) a load rescheduling phase, (3) a work phase where assignments ($x[i]$:=1) are performed, and (4) a phase that estimates the work remaining and controls the parallel *loop*. The entire algorithm is moderately involved, but fairly modular. Phases 1 and 4 involve bottom up traversal of two different heaps and phase

2 involves a top down traversal of these heaps. Algorithm W uses the ability of the PRAM to atomically write words of $O(\log N)$ bits. However this is only for convenience of presentation, and in Section 6 we remove this assumption.

This solution is simple enough to capture certain engineering intuitions (e.g., the rescheduling involves divide-and-conquer) and to be easily implementable (e.g., we include detailed description of the code in Appendix A). Proving robustness is the subject of Section 3.2. The phases of the algorithm are such that reasoning about the failure patterns involves few cases and the algorithm analysis uses recurrences and inequalities. By exploiting *parallel slackness* as as advocated by Valiant [98], and using a slightly smaller number of processors ($1 \leq P \leq N/(\log^2 N - \log\log N \log N)$, where N is the size of the input array) we show that *Write-All* can be solved optimally with $S^* = O(N)$.

For simplicity of presentation, in the rest of this section we assume that the initial number of processors P is N, where N is the input size. Our results immediately extend to any P in the range $1, \ldots, N$ by assuming that the algorithm starts with N processors, and that $N - P$ processors fail prior to the first step of the algorithm.

Algorithm W definition

Algorithm W is a four phase iterative algorithm. It uses full binary trees to (1) enumerate surviving processors, (2) allocate processors, (3) perform work ($x[i] := 1$), and (4) measure progress.

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: We use four full binary trees, each of size $2N - 1$, stored as *heaps* in shared memory. By heap $h[1..2N-1]$ we mean that array h codes a *full binary tree* structure by using $h[i]$ ($i = 1, \ldots, N-1$) as an internal tree node with the left child $h[2i]$ and the right child $h[2i+1]$.

The heaps are $c[1..2N-1]$ (for processor counting and allocation), $cs[1..2N-1]$ (for keeping step numbers), $d[1..2N-1]$ (for progress counting) and $a[1..2N-1]$ (for top-down auxiliary accounting). They are initially 0.

The input is in shared array $x[1..N]$, where the N elements of this array are associated with the leaves of the heaps d and a. Element $x[i]$ is associated with $d[i+N-1]$

```
01 forall processors PID=1..N parbegin
02     Phase W3: Visit the leaves based on PID to perform work on the input data.
03     Phase W4: Traverse the d heap bottom up to measure progress.
04     while the root of the d heap is not N do
05         Phase W1: Traverse the c, cs heaps bottom up to enumerate processors.
06         Phase W2: Traverse the d, a, c heaps top down to reschedule work.
07         Phase W3: Perform rescheduled work on the input data.
08         Phase W4: Traverse the d heap bottom up to measure progress
09     od
10 parend
```

Figure 3.1: A high level view of algorithm W

and $a[i + N - 1]$, where $1 \leq i \leq N$. Similarly processors are initially associated with the leaves of the heap c, such that processor PID is associated with $c[\text{PID}+N - 1]$.

Each processor uses some constant amount of local memory. For example, this local memory may be used to perform some simple arithmetic computations. Important local variables are PID, containing the initial processor identifier, and pn, containing a dynamically changing processor number. Note that PID's do not change but pn's do.

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

Control-flow: Due to the omniscience of the adversary, we employ an oblivious iterative approach in the sense that the pool of the available processors is treated uniformly and is assigned evenly to the tasks that need to be done. The basic idea of the *loop* is: (a) For *failure detection* use bottom up, fast parallel summation to estimate the surviving processors and to estimate the progress they have made. (b) For *load rescheduling* use a top down, divide-and-conquer strategy based on the estimate of progress made. This idea is realized as follows.

The algorithm consists of the parallel *loop* given in Figure 1. This loop is performed, in a synchronous way, by all processors that have not stopped. It consists of four phases of steps, and the first time only part of it is executed (phases W3 and W4). Of course, processors can fail-stop at any time during the algorithm. We next proceed with a high level description of the phases, and then provide additional details with examples.

Phase W1 – *the failure detection/processor enumeration phase*: In this phase all processors traverse a full binary tree used for processor counting starting with the

leaves associated with processor identifiers (PIDs) and finishing at the root. A version of the standard parallel addition algorithm is used for counting.

Phase W2 – *the processor allocation phase*: Here, the processors begin at the root of the full binary tree that represents the progress of the algorithm, and traverse it starting with the root and finishing at the leaves associated with the unfinished work. The processors are allocated in a divide-and-conquer fashion according to the hierarchy of the progress tree.

Phase W3 – *the work phase*: The processors now perform work they find at the leaves they reached in phase W2.

Phase W4 – *the progress measurement phase*: The processors begin at the leaves of the progress tree where they ended phase W3 and traverse it to the root to estimate the progress of the algorithm. A version of the standard parallel addition algorithm is used to count the number of leaves where the work of phase W3 was successfully done.

Algorithm W technical details:

In **phase W1** each processor PID traverses heaps c and cs bottom up from from the location PID$+N-1$. The $O(\log N)$ path of this traversal is the same (static) for all the loop iterations. As processors perform this traversal they calculate an overestimate of the surviving processors. This is done using a standard $O(\log N)$ parallel-time version of a CRCW summation algorithm. Heap c holds the sums and heap cs the timestamps (or step numbers) for the current *loop* iteration. This allows reusing c without having to initialize it each time. Also, during this traversal surviving processors calculate new processor numbers pn for themselves, based on the same sums. Detailed code for this procedure is given in Appendix A.2.

Each processor PID starts by writing a 1 in the leaf $c[\text{PID}+N-1]$ of the tree c. If a processor fails *before* it writes 1 then its action will not contribute to the overall count. If a processor fails *after* it writes 1 then this number can still contribute to the overall sum if one or more processors were active at a sibling tree node and remained active as they moved to the ancestor tree node. The same observation applies to counts written subsequently at internal nodes, which are the sums of the counts of the children nodes in tree c.

Example 3.1 *Processor enumeration:* Consider phase W1 for $N = 4$. There are 4 processors with PIDs 1, 2, 3, and 4, and the counting tree is represented as the heap $c[1..7]$. If processor 1 failed prior to the start of phase W1, processor 3 failed right after writing 1 into its leaf $c[6]$, and processor 4 failed after calculating $c[3] = 2$ as the sum of its ($c[4] = 1$) and processor 3's ($c[3] = 1$) contributions, then the heap will look like this after the completion of the phase. Observe that the root value $c[1] = 3$ yet the actual number of active processors is 1. □

$c[1]$:	3			
$c[2,3]$:	1	2		
$c[4,5,6,7]$:	0	1	1	1
PID:	1	2	3	4

It is easy to show that phase W1 will always compute in $c[1]$ an *overestimate* of the number of processors, which are surviving at the time of its completion (see Lemma 3.1).

We also need to enumerate the surviving processors. This is accomplished by each processor assuming that it is the only one, and then adding the number of the surviving processors it estimates to its left. This enumeration creates the dynamic processor number pn.

Finally, in phase W1 we must be able to reuse our heap several times. This presents a problem. For example, if a processor had written 1 into its heap leaf and then failed then the value 1 will remain there for the duration of the computation, thus preventing us from computing monotonically tighter estimates of the number of surviving processors. This is corrected by associating a step number with each node of the count heap c and storing it in heap cs, thus time stamping valid data. The count step is initially zero, and during each successive *loop* iteration, gets incremented by each surviving processor. Failed processors will not increment their step numbers, thus enabling the surviving processors to detect counts that are out-of-date and treat them as zeroes. We need not worry about time stamping overflow, since we have words of $O(\log N)$ bits and in the worst case the *loop* iterates N times (see Lemma 3.3).

In **phase W2** all surviving processors start at the root of the progress tree d. In $d[i]$ there is an underestimate of the work already performed in the subtree defined by i. Now the processors traverse d top down and get rescheduled dynamically according to the work remaining to be done in the subtrees of i.

It is essential to balance the work loads of the surviving processors. In the next section, we formally show that the algorithm meets the goal of balancing (Lemma 3.2).

Although the divide-and-conquer idea based on d is sound, some care has to be put into its implementation.

In the remaining discussion of phase W2 we explain our implementation, which is based on *auxiliary progress tree* a. The values in a are *defined* from the values in d. All values in a are defined given d, although only part of a is actually *computed*. The important points are that (i) a represents the progress made *and* fully recorded from leaves to the root, and (ii) the value of each $a[i]$ is defined based only on the values of d seen along the unique path from the root to the node i.

At each internal node i, the processors are divided between the left and right subtrees in proportion to the leaves that either have not been visited *or* whose visitation was not fully recorded in d. This is accomplished by computing $a[2i], a[2i+1]$ and using these values instead of $d[2i], d[2i+1]$ in order to discard partially recorded progress (caused by failures and recorded by the processors in the dynamic bottom up traversal of d only part way to the root). We detect partially recorded progress in d when a value of an internal node in d is less than the sum of the values of its two descendants. Thus, at i, after computing the values $a[2i], a[2i+1]$, the scheduling of work is done using divide-and-conquer according to the values $N - a[2i]$ and $N - a[2i+1]$.

Formally, the nonnegative integer values in a are constrained top down as follows:

The root value is $a[1] = d[1]$. For the children of an interior node i $(1 \leq i \leq N-1)$ we have $a[2i] \leq d[2i]$, $a[2i+1] \leq d[2i+1]$, and $a[2i] + a[2i+1] = a[i]$

These constraints do not uniquely define a. However, we realize a unique definition by making $a[2i]$ and $a[2i+1]$ proportional (up to round-off) to the values $d[2i]$ and $d[2i+1]$. Thus, our dynamic top-down traversal (given in detail in Appendix A.4) implements one way of uniquely defining the values of a satisfying these constraints.

The constraints on the values of a assure that (i) there are exactly $d[1] = a[1]$ number of leaves whose d and a values are 1 — such leaves are called *accounted*, and no processor will reach these leaves, and (ii) the processors reach leaves with the a values of 0 — such leaves are called *unaccounted*. Also see Example 3.2 below for additional intuition on a.

Remark 3.1 Strictly speaking, the auxiliary progress tree a need not be represented as a shared heap. Since the values of the a heap are computable from the d heap, it is

sufficient for each processor to have three local scalar variables to represent a node and its two descendants in a "virtual" heap a. However it is convenient to use a as defined above in the proofs of the next section. In any case, a linear amount of storage is used.

In **phase W3** all processors are at the leaves reached in phase W2. Each processor writes 1 in the array element associated with the leaf it has been rescheduled to. Prior to the start of the first iteration of the *loop* each processor PID tries to write in location x[PID]. phase W3 is where the work of the original non-robust algorithm gets done. It is contained within procedure Main in Appendix A.1.

In **phase W4** the processors record the progress made by traversing the d heap bottom up and using the standard summation method. The $O(\log N)$ paths (dynamically) traversed by processors can differ in each *loop* iteration, since processors start from the leaves where they were in phase W3. What is computed each time is an underestimate of the progress made. No timestamps are needed here because the progress recorded increases monotonically. This dynamic bottom up traversal is given in Appendix A.3.

Phase W4 is a simple variant of phase W1, except for the fact that the path traversed bottom up is dynamically determined. One can easily show that the progress recorded in $d[1]$ by phase W4 increases *monotonically* and it *underestimates* the actual progress (see Lemma 3.3). This guarantees that the algorithm terminates after at most N iterations, since $d[1] \neq N$ is the guard that controls the main *loop*.

The following example illustrates phase W4, and provides intuition for why the heap a is used in phase W2 and why it is needed by the proof framework presented in the next section.

Example 3.2 *Progress estimation:* Consider phase W4 for $N = 4$. There are 4 processors with PIDs 1, 2, 3, and 4, and the progress tree is represented as the heap d[1..7]. If, during a phase W4 bottom-up traversal of the progress heap d, processor 4 failed prior to the start of the phase, and processor 3 failed after the first step of the traversal having written 1 into the leaf d[6], then the d heap will look like this after the completion of the phase.

d[1]:	2			
d[2,3]:	2	0		
d[4,5,6,7]:	1	1	1	0
PID:	1	2	3	4

Let $P' = 2$ be the number of surviving processors. We see that $d[1] = 2$ is an underestimate of the actual number, i.e. 3, of visited leaves. If the d heap is used directly in

phase W2 to allocate processors to the unvisited leaves, then the leaf associated with d[7] will be allocated all P' surviving processors. On the other hand, by knowing the (overestimate) number of surviving processors P' and the (underestimate) of the visited leaves $d[1]$, we would like to prove that the allocation is balanced, and that no leaf is allocated more than $\lceil P'/(N - d[1]) \rceil = 1$ processors. We use the heap a in phase W2, where the surviving processors compute $a[6] = a[7] = 0$, with each reaching a distinct leaf thus assuring balanced processor allocation. □

Analysis of algorithm W

We now outline the proof of robustness for algorithm W. Lemma 3.1 shows that in each loop iteration, the algorithm computes (over)estimates of the remaining processors. In Lemma 3.2 we prove that processors are only allocated to the unaccounted leaves, and that all such leaves are allocated a balanced number of processors. Lemma 3.3 assures monotonic progress of the computation, and thus its termination. In Lemma 3.4 we develop an upper bound on the work performed by the processors prior to the algorithm termination. Lemmas 3.1 and 3.3 are proved using simple inductions on the structure of the heaps used by the algorithm. Lemma 3.2 is shown by using an invariant for the algorithm of phase W2. Lemma 3.4 is the central lemma of this section and its proof consists of a relatively involved induction on the size of the estimated work remaining at some step of the algorithm.

These lemmas are used to show the main Theorem 3.5, and, by exploiting parallel slackness, we obtain the optimality result Theorem 3.7.

We first introduce some terminology. Let us consider the i-th iteration of the *loop* ($1 \leq i \leq N$). Note that the first iteration consists only of phases W3 and W4. Define: (1) U_i to be the estimated remaining work, the value of $N - d[1]$ right before the iteration starts, i.e. right after phase W4 of the previous iteration (U_1 is N); (2) P_i to be the real number of surviving processors, right before the iteration starts, i.e. right after phase W4 of the previous iteration (P_1 is P); (3) R_i to be the estimated number of surviving processors, that is the value of $c[1]$ right after phase W1 of the i-th iteration (R_1 is P). The following is shown by straightforward induction on tree c.

Lemma 3.1 In algorithm W, for all loop-iterations i we have: $P_i \geq R_i \geq P_{i+1}$, as long as at least one processor survives.

Proof: The basis $P = P_1 = R_1 \geq P_2$ is obvious. P_i is the number of processors active prior to the first PRAM instruction of the phase W1 algorithm for static bottom-up traversal. By the definition of the model, we immediately have $P_i \geq P_{i+1}$. We will first show that $P_i \geq R_i \geq P_{i+1}$ by induction on the structure of the c tree after the completion of the phase W1 static bottom-up traversal. For simplicity we will treat the values of the c tree with incorrect cs version numbers as virtual zeroes, and not involve cs tree further in this proof. The proof will involve two inductions: one to show the first part of the inequality, and one for the second part.

(1) *Inequality* $P_i \geq R_i$: Let $s(t)$ denote the number of processors that *initiated* phase W1 of the algorithm in the subtree of the c tree rooted at node t. Clearly, we have that $s(1) = P_i$.

Basis: For all subtrees of height 0 rooted at t, $c[t] \leq s(t)$, because some processors may have stop-failed after the initiation of phase W1, but before the initialization of the leaves of c tree.

Inductive hypothesis: assume that for all subtrees of height h rooted at nodes t, we have $c[t] \leq s(t)$.

Inductive step: consider nodes t of height $h + 1$. By the inductive hypothesis: $c[2t] \leq s(2t)$ and $c[2t + 1] \leq s(2t + 1)$. If any processor reached a node t, then $c[t] = c[2t] + c[2t + 1] \leq s(2t) + s(2t + 1) = s(t)$. If no processors reached the node t, then $c[t] = 0 \leq s(t)$.

The induction stops at $t = 1$ where $R_i = c[1] \leq s(1) = P_i$, and so $P_i \geq R_i$.

(2) *Inequality* $R_i \geq P_{i+1}$: This can be shown using similar induction, but instead of $s(t)$ we define $r(t)$, to be the number of processors that initiated phase W1 in the subtree of the node t and that *completed* the phase W1 traversal. $r(1)$ is the upper bound for P_{i+1}, and the induction will show that $r(1) \leq c[1]$. □

In the dividing done during the dynamic top down traversal in W, we will allocate processors to tasks that either have not been completed, or have been completed, but not yet accounted for at the root $d[1]$. Recall that a leaf of d is *accounted* if it has value 1 and if the corresponding defined value in the leaf of heap a is also 1 (there are exactly $d[1]$ accounted leaves). In Algorithm W, the processors get allocated in a balanced fashion to the *unaccounted* leaves, i.e. the leaves whose associated (defined and) computed value

in heap a is 0. The next lemma shows that the processors allocation to the unaccounted leaves is balanced. It involves a detailed but straightforward assertional proof. Below we give the lemma with a proof sketch, and its full proof is found in Appendix A.6.

Lemma 3.2 In phase W2 of each loop-iteration i of algorithm W: (1) processors are only allocated to unaccounted leaves, and (2) no leaf is allocated more than $\lceil R_i/U_i \rceil$ processors.

Proof sketch: The lemma is shown by proving an invariant for the phase W2 algorithm. For each active processor, the main assertions of the invariant is that during the top down traversal, at each node j of the progress tree d and the auxiliary progress tree a: (1) $a[j]$ is strictly less than the number of leaves in the subtree of node j, and $a[j] \leq d[j]$, and (2) the maximum number of active processors allocated to the progress subtree of node j is equal (up to a round-off) to R_i/U_i times the number of unaccounted leaves in that subtree. When the surviving processors reach the leaves, it follows from the invariant that $a[j] = 0$, i.e., the leaf is unaccounted, and that the number of processors at that leaf is no more than $\lceil R_i/U_i \rceil$. □

The following lemma shows that for each loop-iteration, the number of unvisited leaves is decreasing monotonically, thus assuring termination of the main loop after at most N iterations. The worst case of exactly N iterations corresponds to a single processor surviving at the outset of the algorithm.

Lemma 3.3 In algorithm W, for all loop-iterations i we have: $U_i > U_{i+1}$, as long as at least one processor survives.

Proof: To prove this, we define $a_i[1..2N\text{-}1]$ and $d_i[1..2N\text{-}1]$ to be the values of trees a and d after the completion of iteration i. $d_0[1..2N\text{-}1]$ are the initial zero values. We first show that if an iteration $i+1$ is started with tree d satisfying $d_i[t] \leq d_i[2t] + d_i[2t+1]$ $(1 \leq t < N)$, then after the termination of loop-iteration $i+1$, tree d will satisfy $d_{i+1}[t] \leq d_{i+1}[2t] + d_{i+1}[2t+1]$ $(1 \leq t < N)$, and along a path completely traversed from leaf to root by a processor in phase W4: $a_i[t] < d_{i+1}[t]$ $(1 \leq t < 2N$, t along the path traversed).

This can be shown using straightforward induction on the structure of the tree d and the loop-iteration number i. From this, since $U_i = N - d_i[1] = N - a_i[1]$ and $U_{i+1} = N - d_{i+1}[1]$, we have $N - U_i < N - U_{i+1}$ which leads to the desired result. □

We now come to the main lemma. We will treat the three $\log N$ time tree traversals performed by a single processor during each phase of the algorithm as a single *block-step* of cost $O(\log N)$. We will charge each processor for each such block step, regardless of whether the processor actually completes the traversals or whether it fail-stops somewhere in-between. This coarseness will not distort our results; since we can have at most P processor failures it amounts to a one time overcharge of $O(P \log N)$. Let us take a snapshot of the algorithm after completion of several loop-iterations. We are right before loop-iteration i. V_i stands for the total number of block-steps performed by the processors in trying to complete all remaining work (at most U_i).

Lemma 3.4 For any failure pattern with at least one surviving processor, and starting at each loop-iteration i, algorithm W completes all remaining work. Its total number of block-steps V_i is less than or equal to $P_i + U_i + P_i \log(U_i)$, where $1 \leq P_i, \;\; U_i \leq N$.

Proof: We proceed by induction on the size of U_i. For the base case: We have at most one unaccounted leaf and some number of processors ($U_i = 1, P_i \geq 1$). As long as at least one processor survives, we are going to visit the single remaining leaf in one phase in which at most P_i processors participate and $P_i \leq P_i + 1 + P_i \log(1)$.

For the inductive hypothesis: we assume the lemma is true for all $U_i < U, \;\; P_i \geq 1$, where $U \leq N$. We will then prove it for $U_i = U, \;\; P_i \geq 1$.

We divide the proof in two cases: (1) as many unaccounted leaves at least as processors, i.e., $P_i \leq U_i$, and (2) more processors than unaccounted leaves, i.e., $P_i > U_i$.

In both cases, by Lemma 3.2, we have that the (accounted) progress for iteration i is at least the number of surviving processors P_{i+1} divided by $\lceil R_i/U_i \rceil$. This is because each one of these processors returns to the root $d[1]$, reporting some progress, and at most $\lceil R_i/U_i \rceil$ processors report information about the same leaf.

Also, by Lemma 3.1, $P_i \geq R_i \geq P_{i+1}$, and we can assume that $kP_i = P_{i+1}$, for some k with $0 < k \leq 1$ (at least one processor survives). Thus, for both the above cases, we have:

$$U_{i+1} \leq \left(U_i - \frac{P_{i+1}}{\lceil R_i/U_i \rceil} \right) \leq \left(U_i - \frac{P_{i+1}}{1 + R_i/U_i} \right) \leq U_i \left(1 - \frac{k}{1 + U_i/P_i} \right)$$

For the case (1) it is easy to see that we will have at most one processor allocated to each unaccounted leaf so: $U_{i+1} \leq U_i - P_{i+1}$. For the case (2) by the above inequality

and $P_i > U_i$ we have $U_{i+1} \leq U_i(1 - k/2)$. Now we use the inductive hypothesis (but for iteration $i + 1$) in both cases.

Case (1): The survival of at least one processor and $U_{i+1} \leq U_i - P_{i+1}$ imply that $U_{i+1} < U_i$. The total work (in block-steps) is at most $P_i + V_{i+1}$, where by the hypothesis $V_{i+1} \leq P_{i+1} + U_{i+1} + P_{i+1} \log(U_{i+1})$. Thus, it suffices to show that $P_{i+1} + U_{i+1} + P_{i+1} \log(U_{i+1})$ is less than or equal to $U_i + P_i \log(U_i)$. This is trivial given $U_{i+1} \leq U_i - P_{i+1}$ and Lemmas 3.1 and 3.3.

Case (2): There are two subcases. If $k = 1$ the algorithm completes correctly in one iteration and the work $P_i = R_i = P_{i+1}$ trivially satisfies the Lemma. The second subcase is the most interesting one and is if $0 < k < 1$. For this subcase we use $U_{i+1} \leq U_i(1 - k/2)$, which implies $U_{i+1} < U_i$. As in case (1), the total work (in block-steps) is at most $P_i + V_{i+1}$, where by the hypothesis $V_{i+1} \leq P_{i+1} + U_{i+1} + P_{i+1} \log(U_{i+1})$. Thus, it suffices to show that $P_{i+1} + U_{i+1} + P_{i+1} \log(U_{i+1})$ is less than or equal to $U_i + P_i \log(U_i)$. For $2 > U_{i+1} = 1$ this is trivial.

By simple manipulation it suffices to show that $kP_i + U_i(1-k/2) + kP_i \log(U_i(1-k/2))$ is less than or equal to $U_i + P_i \log(U_i)$. This is equivalent to showing that

$$k\left(1 - \frac{U_i}{2P_i}\right) + k \log\left(1 - \frac{k}{2}\right) \leq (1 - k) \log U_i$$

Recall that all logarithms are base 2 and therefore $(\log(1/2) = -1)$. Since $U_i \geq 2$ ($U_i = 1$ was taken care of by base case) we have $\log U_i \geq 1$. Also, in this case $1 - U_i/2P_i \leq 1$. It thus suffices to show the inequality

$$k \log(2 - k) \leq (1 - k), \quad \text{for } 0 < k < 1 \qquad (*)$$

Inequality $(*)$ is true by elementary calculus (it is tight only for $k = 1$). This completes the proof of the second subcase, of case (2) and of the Lemma. □

Remark 3.2 This lemma also shows, that if an algorithm existed that could balance the load of the surviving processors and allocate them in constant time, then the *Write-All* problem could be solved with logarithmic overhead ($\log U_1 = \log N$). The lower bound result in Section 4 provides some intuition for the multiplicative factor $\log U_i$.

Theorem 3.5 Algorithm W is a robust parallel algorithm for the *Write-All* problem with $S^* = O(N \log N + P \log^2 N)$, where N is the input array size, and the initial number of processors P is between 1 and N.

Proof: This immediately follows from Definition 2.6 and Lemmas 3.1-3.4. Note that although we assumed N processors in Algorithm W, we only used the fact that $P \leq N$ in the lemmas. In fact, as indicated earlier, we accommodate $P < N$ processors by considering that $N - P$ processors failed prior to the beginning of the algorithm. This contributes a single charge of $O(N-P)$ to the cost, and does not distort the asymptotic result that consists of the product of the total block-steps V_1 performed by the algorithm since the first iteration of the algorithm (inclusive) times the per-block-step cost of $O(\log N)$:

$$S = V_1 \cdot O(\log N) = (P_1 + U_1 + P_1 \log U_1) \cdot O(\log N) \quad \text{(using Lemma 3.4)}$$

$$= (P + N + P \log N) \cdot O(\log N) \quad \text{(using } P_1 = P \text{ and } U_1 = N\text{)}$$

$$= O(P \log N + N \log N + P \log^2 N) = O(N \log N + P \log^2 N)\ . \ \square$$

One immediate observation of this result shows that fewer processor steps will be expended by the algorithm if it is started with less than N processors. For example we reach a $S^* = O(N \log N)$ bound when using $P = N/\log N$ processors. A question can be posed: could an optimal algorithm for the *Write-All* problem be constructed using a non-trivial number of processors? This question is positively answered below.

We first observe that each block-step takes $\Theta(\log N)$ time and therefore each processor can be asked to perform $\Theta(\log N)$ processing steps in phase W3 without affecting the asymptotic complexity. To take advantage of this, we parameterize algorithm W as follows:

1. Let N be the size of the input.
2. Let $H \leq N$ be the instance size for the algorithm, thus the height of the trees used is $\log H$.
3. Let $G = N/H$ be the number of the input array elements mapped to each leaf of the heaps.
4. Let $P \leq H$ be the initial number of processors.

With these data structures, the performance of algorithm W is described by the following lemma:

Lemma 3.6 Algorithm W with P processors, the progress tree with H leaves ($P \leq H$) and $2H-1$ total nodes all initialized to zero and G array elements at each leaf, has the work of $S^* = O((H + P \log H) \cdot (\log H + G))$ for any pattern of stop failures.

Proof: The cost of a single block-step C_B is $O(\log H + G) = O(\log H + N/H)$. By Lemma 3.4 the algorithm will verifiably visit all leaves of the progress heap after spending $V_1 = P_1 + U_1 + P_1 \log U_1 = P + H + P \log H = H + P \log H$ block-steps. Therefore $S^* = V_1 \cdot C_B$, and so:

$$S^* = O(H + P \log H) \cdot O(\log H + N/H) = O(H \log H + P \log^2 H + N + \tfrac{PN \log H}{H}) \ .$$

□

To achieve work optimality, we would like to choose the parameters in the lemma so that $S^* = O(N)$. While the exact solution is involved, we observe that the following values for parameters G, H and P produce the desired result:

$$G = \log N \ , \ H = N/\log N \ \text{ and } \ P = H/\log H = N/(\log^2 N - \log N \log\log N) \ .$$

Thus by exploiting parallel slackness, we achieve work optimality using a number of processors smaller than N:

Theorem 3.7 Parameterized algorithm W with $\log N$ array elements mapped to each leaf of the progress heap is a robust parallel algorithm that solves the *Write-All* problem of size N with $S^* = O(N)$, when $P \leq N/(\log^2 N - \log N \log\log N)$.

However, as we show in the chapter on lower bounds, no optimal N-processor algorithm exists for *Write-All*.

The parameterized algorithm W as in the last theorem above can also be used with any number of processors P such that $1 \leq P \leq N$. When using P processor such that $P > \frac{N}{\log N}$, it is sufficient for each processor to take its PID modulo $\frac{N}{\log N}$ to assure 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.

Worst case adversary for algorithm W

We are going to show in Chapter 4 that an adversary strategy can be constructed so that any *Write-All* algorithm with $P = N$ processors will be forced to perform $S = \Omega(N \frac{\log N}{\log\log N})$ work steps (Theorem 4.4).

That theorem can be directly utilized to produce the following result:

Theorem 3.8 There is a processor failure pattern for algorithm W that results in $S = \Theta(N \log^2 N / \log\log N)$, for $P = N$.

Proof: This is accomplished by using the adversary that fail-stops processors according to the strategy as in the proof of the Theorem 4.4, except that instead of PRAM steps, the adversary uses *block-steps*, and the processors are stopped only during the phase W3 where the operations on the actual data take place. This corresponds to the fixed per block-step *processor survival* coefficient k defined in Lemma 3.4 being equal to $1 - 1/\log N$. □

Using a slightly different strategy, it is possible to construct failure patterns that force the algorithm to take $\Omega(N \log N \log\log N)$ steps (this example is due to Jeff Vitter). This is done by utilizing a *variable* per block processor survival coefficient $k = 1 - 1/\sqrt{U_b}$, where U_b, is the underestimate of the unvisited leaves in Algorithm W after the completion of block iteration b.

It was shown by Martel [71] that the worst case performance of algorithm W is no worse than $S = \Theta(N \log^2 N / \log\log N)$. We state this result here and give its proof in Appendix A.6.

Theorem 3.9 [71] Algorithm W is a robust parallel algorithm for the *Write-All* problem with $S^* = O(N \log^2 N / \log\log N)$, where N is the input array size, and the initial number of processors P is between 1 and N.

3.2.2 Algorithm V

Algorithm W in the previous section is an efficient fail-stop (no restart) *Write-All* solution. It 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 for algorithm W 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 4.7) 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 W, the reader is urged to refer to the previous section (3.2.1).

Definition of algorithm V

We formulate algorithm V using the data structures of the optimized algorithm W.

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 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 of the progress tree. Each processor instead of using its PID during the computation uses the PID modulo $\frac{N}{\log N}$. When the number of processors P is such that $P > \frac{N}{\log N}$, this assures 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 the work elements at each leaf.

Control-flow: Algorithm V is an iterative algorithm using the following three phases.

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

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

```
01  forall processors PID=1..N parbegin
02    Phase V2: Visit the leaves based on PID to perform work on the i nput data.
03    Phase V3: Traverse the d heap bottom up to measure progress.
04    while the root of the d heap is not N do
05        Phase V1: Traverse the d, a, c heaps top down to reschedule work.
06        Phase V2: Perform rescheduled work on the input data.
07        Phase V3: Traverse the d heap bottom up to measure progres s
08    od
09  parend
```

Figure 3.2: A high level view of algorithm V

Phase V3 – *progress measurement*: 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 3.10 The 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 4.7, 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 4.7, 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 4.7 (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 4.7 with the $\lceil P/\frac{N}{\log N} \rceil$ processor allocation as a multiplicative factor. From this, the 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 $S^* = O(N + P \log^2 N)$. □

The following corollary extracts the slightly better bound analyzed in the case (1) above, and it also covers the processor range for which the work of the algorithm is optimal.

Corollary 3.11 The work of algorithm V using $P \leq N/\log N$ processors that are subject to fail-stop errors without restarts is $S^* = O(N + P \log N \log P)$.

The upper bound analysis is tight:

Theorem 3.12 There is a fail-stop adversary that causes the work of algorithm V to be $S^* = \Omega(P \log^2 N)$ for the number of processors $N/\log N \leq P \leq N$, and $S^* = \Omega(N + P \log N \log P)$ for the number of processors $1 \leq P \leq N/\log N$.

Proof: Consider the following adversary for $P = N/\log N$. At the outset the adversary fail-stops all processors that are initially assigned to the, say, left subtree of the progress tree. Let the number of unvisited array elements be U. By the balanced allocation lemma (3.2) the N processors (dead or alive) will be assigned in a balanced fashion to the left and right segments of the contiguous U unvisited elements. Initially, U_0 is $N/\log N$, and so the algorithm will terminate in $\log U_0 = \Theta(\log N)$ block-steps when such an adversary is encountered. Each block-step takes $\Theta(\log N)$ time using the remaining $P/2$ processors. Thus the work is $S^* \leq \frac{P}{2}\Theta(\log N)\Theta(\log N) = \Theta(P\log^2 N) = \Omega(N\log N)$.

When P is larger than $N/\log N$, then each leaf is allocated at least $\lfloor P/\frac{N}{\log N}\rfloor$ and no more than $\lceil P/\frac{N}{\log N}\rceil$ processors. All processors allocated to the same leaf have their PIDs equal modulo $N/\log N$. Therefore the work is increased by at least a factor of $\lfloor P/\frac{N}{\log N}\rfloor$ as compared to the case $P = N/\log N$. I.e., $S^* = \lfloor P/\frac{N}{\log N}\rfloor\Omega(N\log N) = \Omega(P\log^2 N)$.

Finally, when $P < N/\log N$, the result follows similarly using the strategy of the case (1) of Lemma 3.10. □

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

Theorem 3.13 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 3.10 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 3.10 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 not bounded by a function of P and N for large $|F|$). □

Remark 3.3 Recall that when the failure patterns are such that the size M is bounded by P, then the measures S^* and S^+ are asymptotically equal.

3.2.3 Processor Allocation Monotonicity

One of the advantages of the two algorithms that we presented for the global allocation paradigm is that both algorithms have what we define as the *processor allocation monotonicity* property:

Definition 3.1 A given *Write-All* algorithm has a *processor allocation monotonicity* property if whenever during the execution of any step of the algorithm there are two array location a_1 and a_2 (without loss of generality let $a_1 < a_2$) that are being concurrently written to by two processors with PIDs p_1 and p_2 respectively, then $p_1 < p_2$.

This property becomes important when parallel algorithms are simulated on fault-prone PRIORITY PRAMs. These simulations will be addressed in Chapter 5. Neither the local allocation algorithm nor the hashed allocation algorithm presented further in this chapter have this property.

To show that algorithms W and V satisfy the *processor allocation monotonicity* property, we need to examine the processor enumeration and allocation phases of algorithm W, and the allocation phase of algorithm V. In the enumeration phase, a surviving processor is enumerated by adding one to the overestimate of surviving processors with PIDs smaller than that processor's PID. Thus processors are enumerated monotonically: larger PIDs are given larger processor numbers. In the allocation phase, the enumerated processors are assigned in a divide-and-conquer strategy according to a binary tree: lower (higher) numbered processors are assigned to the subtrees that contain lower (higher) numbered work elements. Therefore the processor allocation is monotonic. The same holds for algorithm V, since processors always have their permanent PIDs and enumeration is not used. This proves the following:

Property 3.2 Algorithms V and W satisfy the *processor allocation monotonicity*.

3.3 Local Allocation Paradigm

In this section we present and analyze an algorithm that can be used in both the non-restartable and restartable models. We call it algorithm X. We also propose (without analysis) two randomized versions of this algorithm that have the potential of being more efficient (in the expected work analysis) than algorithm X, even when subjected to the omniscient adaptive adversaries.

3.3.1 Algorithm X

We present an algorithm for the *Write-All* problem, and show that its completed work complexity is $S = O(N \cdot P^{\log \frac{3}{2}}) = O(N \cdot P^{0.59})$ using $P \leq N$ processors in the restartable fail-stop model 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 $|F| = \Omega(N \cdot P^{\log \frac{3}{2}})$, then the algorithm's overhead ratio σ becomes optimal: it takes a fixed number of computing steps per failure/recovery.

Definition of algorithm X

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

```
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 3.3: A high level view of algorithm X.

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 processor's own 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 3.3; all line numbers refer to this figure. More detailed code can be found in Appendix B.

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.

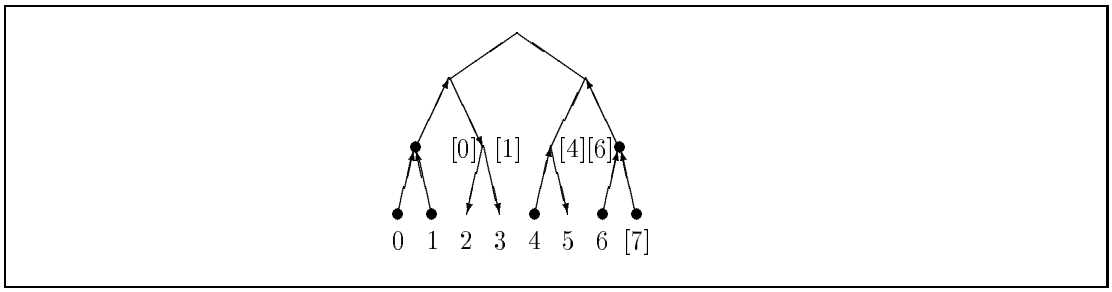


Figure 3.4: An example of processor traversals of the progress tree

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

Example 3.3 *Progress tree traversal:* 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 3.4 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 3.14 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 proceed to 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 3.15 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

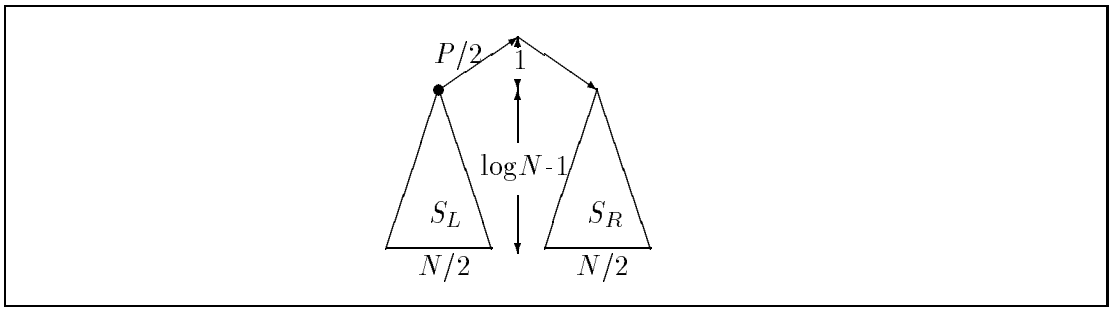


Figure 3.5: Inductive step for Lemma 3.15

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 $\log N \ (\geq 1)$. 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:

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

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 $P_1 \leq P_2$, then $S_{N,P_1} \leq S_{N,P_2}$). 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:

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

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

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 3.16 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 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 $P \leq N$. 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 3.4 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 3.17 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 3.15, $S_{P,P} = O(P \cdot P^{\log \frac{3}{2}}) = O(P^{\log 3})$. 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 3.16, 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 $S = \Omega(\frac{N}{P} P^{\log 3}) = \Omega(N \cdot P^{\log \frac{3}{2}})$. □

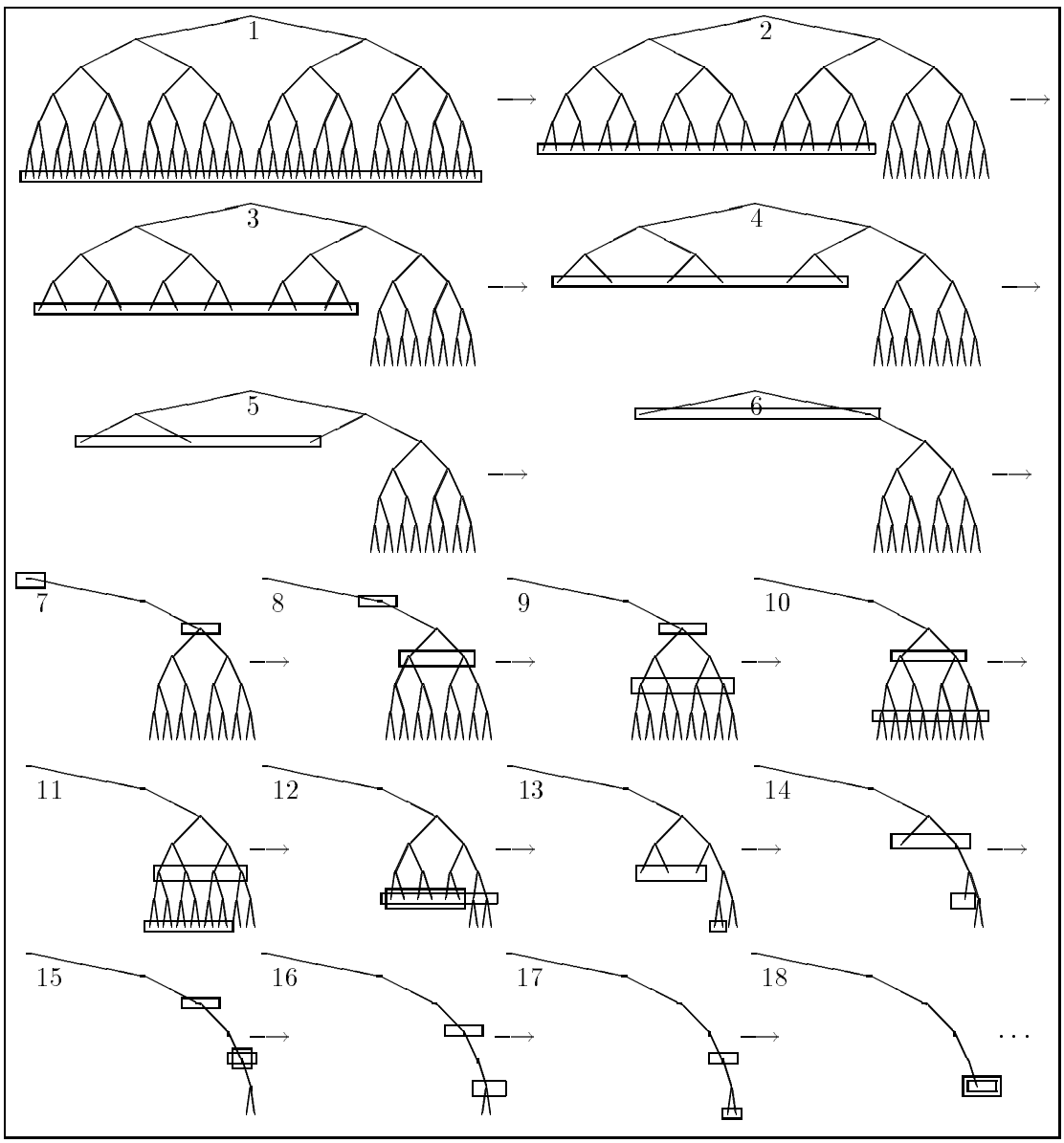
Remark 3.5 Lemma 3.14 gives only a loose upper bound for the worst performance of algorithm X — there we are concerned with termination. The actual worst case time for algorithm X can be no more than the upper bound on the completed work. This is because at any point in time there is at least one update 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 3.16, 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 $\Omega(N^{\log 3})$.

Remark 3.6 In algorithm X the processors work independently; they attempt to avoid duplicating already-completed work but do not co-ordinate their actions with other processors. In [27], Buss et al. show that this property allows the algorithm to run on a strongly asynchronous PRAM with the same work and time bounds.

A fail-stop lower bound for algorithm X

The analysis of the upper bound for algorithm X for the fail-stop no-restart model is still an open question. In this section we show and analyse a particular failure scenario.

The failure scenario is based on the strategy of the adversary that is used in the proof of Theorem 4.4. For algorithm X, this strategy is that if U is the number of unvisited progress tree leaves, then in each step where the unvisited leaves have processors assigned to them, the adversary stops the processors that are assigned to the rightmost $U/\log U$ leaves. For $N = P = 64$, this strategy is illustrated in Figure 3.6.



Horizontal boxes represent processor "waves" traversing the tree. The nodes that are marked by the processors as "visited" are never again traversed. These nodes and the corresponding tree edges are erased for clarity – the upward moving waves appear to be "consuming" the tree. When two waves "collide", they are depicted as two overlapping boxes (steps 12,15,17). After a collision the waves are merged.

The adversary stops only the processors that are visiting the leaves. At step 1, the processors assigned to the 16 ($= N/\log N$) rightmost leaves are stopped. No processor is stopped in steps 2-10. At step 11, the processors assigned to the 4 ($= 16/\log 16$) rightmost leaves are stopped. At step 13, the processors assigned to the 2 ($= 4/\log 4$) rightmost leaves are stopped. The computation terminates in five more steps after step 18, as the single remaining wave "gobbles up" the path to the root.

Figure 3.6: A fail-stop scenario for algorithm X with $P = N = 64$.

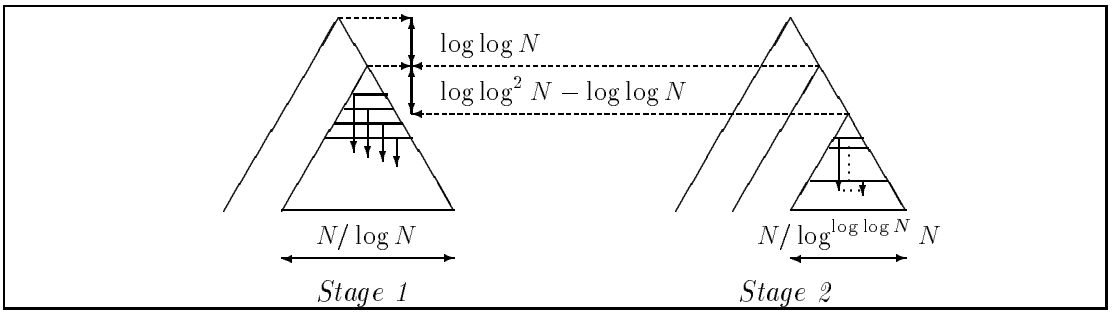


Figure 3.7: Stages 1 and 2 of a general fail-stop scenario for algorithm X

Theorem 3.18 There is a processor failure pattern for algorithm X that results in $S = \Theta(N \log N \log\log N / \log\log\log N)$ for $P = N$.

Proof: The adversary uses the strategy similar to that in Theorem 4.4, except that instead of PRAM steps, the adversary stops the processors only when the waves of processors reach the unvisited leaves. We examine the following stages (stages 1 and 2 are illustrated in Figure 3.7):

Stage 1: The adversary stop $N/\log N$ processors assigned to the rightmost $N/\log N$ leaves. The surviving $N - N/\log N$ processors are allowed to traverse the tree bottom-up and then top-down (after reaching a node such that only one child is marked 1) until the first wave of processors reaches the leaves. There will be: $\log\log N = \log N - \log(N/\log N)$ such waves. (This is illustrated in Figure 3.7, *Stage 1.*)

As each wave reaches the leaves, the processors that are assigned to the rightmost logarithmic fraction of the leaves is stopped (in this proof, it is sufficient to use a $\log N$ fraction, while in the example in Figure 3.6 we use a logarithm of the remaining number of unvisited leaves – asymptotically, both logarithms are equal). After the last wave reaches the leaves, there would be $\Theta(N/\log^{\log\log N} N)$ leaves left. Assume for simplicity that this number is a power of two (it will be for some sufficiently large N, which is sufficient for lower bounds). Note that the number of surviving processors is still $\Theta(N)$ $(= N - N/\log N)$.

Stage 2: The processors will have to traverse a path of length $\Theta(\log(N/\log^{\log\log N} N)) = \Theta(\log N - \log\log^2 N)$ to find unvisited leaves. This time there will be: $(\log N - \log\log N) - (\log N - \log\log^2 N) = \log\log^2 N - \log\log N = \Theta(\log\log^2 N)$ waves (subtracting subtree heights). This is illustrated in Figure 3.7, *Stage 2.* The number of

surviving processors is still $\Theta(N)$, as only a diminishing polylogarithmic fraction of each wave is stopped.

In *Stage i*, there will be $\Theta(\log\log^i N)$ waves, and the total number of live processors is still $\Theta(N)$ processors. Each wave will have to traverse paths of length $\Theta(\log N - \log\log^i N)$.

The algorithm terminates in stage τ, such that $\tau = \log\log N/\log\log\log N$ at which time the remaining leaf (or a constant number of leaves) is visited. While there are still $\Theta(N)$ processors are active. The work performed is as follows:

$$S = N\sum_{i=1}^{\tau}(\log N - \log\log^i N) = N\tau\log N - \sum_{i=1}^{\tau}\log\log^i N$$
$$= \Theta(N\log N\log\log N/\log\log\log N - N\log N/\log\log N)$$
$$= \Theta(N\log N\log\log N/\log\log\log N)\ .$$

□

Recently López-Ortiz refined the particular scenario used in the proof above to exhibit the known worst fail-stop work for algorithm X of $\Theta(N\log^2 N/\log\log N)$ [68]. As the corollary of this result, the upper bound for algorithm X is no better than the upper bound for algorithm W for the fail-stop no-restart model.

3.3.2 Algorithms X_{coin} and X_{die}

In this thesis we investigate deterministic solutions and concentrate on the worst case analysis. Randomized algorithms for the *Write-All* problem are capable of improving on the upper bounds of the deterministic algorithms only when the adversary is off-line as in [75] or when the adversary is limited probabilistically as in [59, 61].

A randomized *asynchronous coupon clipping* (ACC) algorithm for the *Write-All* problem was analyzed by Martel et al. in [75]. Assuming off-line adversaries, it was shown in [75] that ACC algorithm has expected work $O(N)$ using $P = N/(\log N\log^* N)$ processors on inputs of size N. However, when the adversary is on-line then the algorithm becomes inefficient even for simple on-line adversaries.

Example 3.4 *Stalking adversary:* 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 $\Omega\Big((\frac{N}{\text{polylog}N})^{\frac{N}{\text{polylog}N}}\Big)$ work in the case of fail-stop errors with restart even when using $P \leq \frac{N}{\text{polylog}N}$ processors.

Algorithm ACC uses as its main data structure a full binary tree similar to the progress tree of algorithm X. In algorithm ACC, processors randomly visit the nodes of that binary tree. The stalking adversary strategy consists of choosing a single leaf in the 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.

Stalking adversaries can and do occur in practice. Consider a processor or processors that repeatedly attempt to read from a "bad" memory location, or a processor that executes an instruction whose microcode is partially corrupt – all such processors may be failing at the same instruction either indefinitely, or in an intermittent pattern. □

Here were present two randomized versions of algorithm X, and we conjecture that these algorithms have the potential of improving on the performance of algorithm X even when the adversary is the worst case on-line adversary. In particular, either algorithm does not allow the stalking adversary to cause quadratic or worse amount of work.

Algorithm X_{coin}

This algorithm uses the data structures of algorithm X and it is identical to algorithm X, except that in line 09 of Figure 3.3 instead of making a move according to the PID bits, a processor chooses to move left or right after flipping a coin.

The potential advantages of this randomization is that if more than one processor find themselves at the same interior node, then, should they all survive the next step, it is expected that half of them will move right and half left if the corresponding subtrees are not completed.

This strategy might yield better expected performance than algorithm X, because the processors from the same subtrees will fan-out to the uncompleted portion of the progress tree sooner.

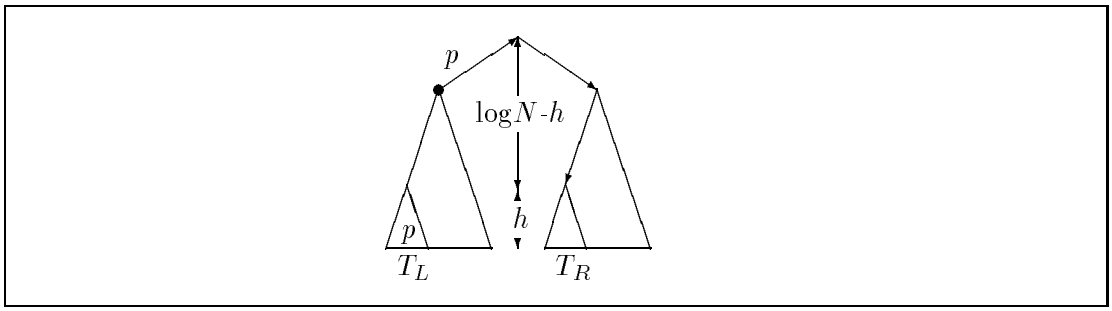


Figure 3.8: An example of an inefficient progress tree traversal

Example 3.5 *Inefficient progress tree traversal:* In Figure 3.8, let T_L and T_R be subtrees of heights h that are identically positioned within the left and right subtrees of the progress tree respectively. Suppose, either in algorithm X or algorithm X_{coin}, all p processors from the subtree T_L reach the root, and the right subtree is not completed.

In algorithm X the processors might synchronously reach the root of subtree T_R at height h without fanning out throughout the right subtree because the processors will make identical down-moves along the path of length $\log N - h$ from the root since they use identical PID bits along the path having originated in the same subtree (N is the number of leaves).

However in algorithm X_{coin}, the processors will probabilistically fan-out at the root of the right subtree. Thus, instead of p processors being at the leaves of tree T_R, the processors will be balanced (probabilistically) throughout the leaves of the entire right subtree. □

Algorithm X_{die}

This algorithm uses the data structures of algorithm X and it is similar to algorithm X with the following two differences:

1. Instead of using binary values at the progress tree nodes indicating whether the subtree is completed or not, algorithm X_{die} uses the integer values at the nodes to represent the known number of descendent leaves visited by the algorithm as done in algorithms V and W.

2. Instead of making a move according to the PID bits (line 09 of Figure 3.3), a processor at an interior progress tree node casts an N-sided die to produce a

random number r in the range $[1, N]$, reads the values d_1 and d_2 of the left and right children of the progress tree and then:

- if $d_1 = d_2 = 0$, then the processor moves left and right with probabilities $\frac{1}{2}$;
- if $d_1 + d_2 \neq 0$, then the processor moves left if $1 \leq r \leq N\frac{2^{h-1}-d_1}{2^h-d_1-d_2}$, where h is the height of the interior progress tree node, and it moves right otherwise — this is done to send processors left or right with probabilities that are proportional to the remaining work in the left $(2^{h-1}-d_1)$ and right $(2^{h-1}-d_2)$ subtrees. (This is similar to the deterministic divide and conquer strategy used in algorithm W.)

This algorithm benefits from the earlier fan-out of the processors just like in algorithm X_{coin}. In addition, in algorithm X_{die} the processors take advantage of the knowledge of the remaining amount of work in the left and right subtree at any node of the progress tree, and attempt to move down to the leaves in numbers that are proportional to the remaining work in the subtrees.

We conjecture that the expected work of both the algorithm X_{coin} and X_{die} is better than the worst case work of algorithm X subject to the worst case on-line adversary.

3.4 Hashed Allocation Paradigm

The final technique is demonstrated using a new heuristic for determinizing an efficient randomized *Write-All* solution proposed by Anderson and Woll in [8].

3.4.1 Algorithm Y

A family of randomized algorithms for *Write-All* was presented in [8]. The basic technique in all of these algorithms is abstracted and given as a high level code in Figure 3.9. The basic algorithm in [8] is obtained by randomly choosing the permutation in line 03. In this case the expected work of the algorithm is $O(N \log N)$, for $P = \sqrt{N}$ (assume N is a square).

We propose the following way of determinizing the algorithm of [8]: Given $P = \sqrt{N}$, we chose the smallest prime m such that $P < m$. Primes are sufficiently dense, so that

```
01    forall processors PID = 1..√N parbegin
02      Divide the N array elements into √N work groups of √N elements
03      Each processor PID obtains a private permutation π_PID of {1, 2, ..., √N}
04      for i = 1..√N do
05          if π_PID[i]th group is not finished
06          then perform sequential work on the √N elements of the group
07                  and mark the group as finished
08          fi
09      od
10    parend
```

Figure 3.9: A high level view of the algorithm Y – hashed allocation paradigm.

there is at least one prime between P and $2P$ (e.g, see the discussion in [32, Sec. 33.8]), so that the complexity of the algorithms is not distorted when P is not a prime. We then construct the multiplication table for the numbers $1, 2, \ldots m-1$ modulo m. It is not difficult to show that each row of this table is a permutation and that this structure is a group (using the basic group theory facts, e.g., [23]). Processor with PID i uses the ith permutation as its schedule.

Note that the table need not be pre-computed, as any item can be computed directly by any processor with the knowledge of its PID, and the number of work elements w it has processed thus far as $(PID \cdot w)$ **mod** m. A detailed pseudo-code for the deterministic algorithm Y is given in Figure 3.10.

We conjecture that the worst case work of this deterministic algorithm is no worse than the expected work of the randomized algorithm.

The open problem below contains an interesting observation of a group-theoretic aspect of a multi-processor scheduling problem [93].

An open problem: What is the completed work of algorithm Y with the proposed determinization? We have performed some experimental analysis and all cases it resulted in the the work being is $O(N \log N)$. This is the same as the expected work using random permutations. We next briefly state the relevant framework, and then state the open problem.

In [8], the analysis of the randomized algorithm is based on the definition of a measure, called *contention*, that evaluates the "overlap" of a permutation representing a processor schedule with respect to a permutation representing a scheduling adversary:

```
01  forall processors PID = 1..P = √N parbegin
02     --The x[1..N] array elements are viewed as divided into √N work groups
03     --of √N elements in each: x[1..√N], x[1+√N..2√N], etc.
04       shared x[1..N];              --shared memory
05       shared done[1..√N];          --done markers
06       shared w[1..√N];             --done by each processor
07       private k;                   --local workgroup number per processor
08       w[PID] := 1; --the initial workgroup to do
09       while w[PID] ≠ 0 do --While not all √N groups done
10            k := (PID · w[PID]) mod m --current workgroup number
11            if not done[k] --group is not marked finished
12            then --perform sequential work on the √N elements of the group
13                 for i = 1..√N do
14                    x[(w[PID] − 1)√N + i] := 1;
15                 od
16                 done[k] := true --mark workgroup as finished
17            fi
18            w[PID] := w[PID] + 1; --advance processor's groups done counter
19       od
20  parend
```

Figure 3.10: A detailed view of the deterministic algorithm Y.

Definition 3.3 [8] Given two permutations π and α represented as lists of integers $\{1, ..., p\}$, the *contention* of π with respect to α, denoted $C(\pi, \alpha)$ is defined as follows: scan α left-to-right, and for each encountered item, delete that item from π, $C(\pi, \alpha)$ is the number of times the deleted item is at the head of the list π.

For example, $C(\langle 2,4,1,3\rangle, \langle 1,2,3,4\rangle) = 2$, and, $C(\langle 3,2,1\rangle, \langle 2,1,3\rangle) = 1$.

Contentions for a set of permutations is defined as follows:

Definition 3.4 [8] Given a set of permutation $\Pi = \{\pi_1, ..., \pi_p\}$ and a permutation α, $C(\Pi, \alpha)$ is defined as $\sum_1^p C(\pi_i, \alpha)$.

Intuitively, contention measures redundant work performed by an algorithm that uses hashed allocation paradigm. The key relevant results shown in [8] are:

Lemma 3.19 [8] For permutations π and α, $C(\pi, \alpha)$ is equal to the number of left-to-right maxima in $\alpha^{-1} \circ \pi$, where $\circ$ is the permutation composition.

Lemma 3.20 [8] If in an algorithm Y with p processors, each processor uses a permutation from $\Pi = \{\pi_1, ..., \pi_p\}$ as its schedules, then the worst case work of the algorithm is $O(p \cdot max_\alpha C(\Pi, \alpha))$.

When we consider the group of all permutations of integers $\{1, ..., p\}$ with the permutation composition $\circ$ being the multiplication operation for the group, the following corollary follows:

Corollary 3.21 For any two permutations α, β: $C(\beta, \alpha) = C(\alpha^{-1} \circ \beta, e)$, where e is the identity permutation.

We now use the deterministic algorithm Y with the permutations being computed deterministically as we proposed, and reduce our conjecture to a problem below that exhibits an interesting connection between multiprocessor scheduling on one hand and group theory and combinatorics on the other.

The P permutations that are computed by the processors constitute a group. Call it Π. Using the above corollary, we observe that the contention of the set of P permutations with respect to any permutation α is $C(\Pi, \alpha)$ and it is the same as the contention of the left coset of Π, $\alpha^{-1} \circ \Pi$ with respect to the identity permutation, that is $C(\alpha^{-1} \circ \Pi, e)$. This allows us to reduce the problem to the following (the details are left as an exercise).

In order to show that the worst case work of Y is $O(N \log N)$, using the above framework, it is sufficient to show that:

> Given a prime m, consider the group $G = \langle \{1, 2, \ldots, m-1\}, \bullet \pmod m \rangle$. The multiplication table for G, when the rows of the table are interpreted as permutations of $\{1, \ldots, m-1\}$, is a group K of order $m-1$ (a subgroup of all permutations). Show that, for each left coset of K (with respect to all permutations) the sum of the number of left-to-right maxima of all elements of the coset is $O(m \log m)$.

Chapter 4

Write-All Lower Bounds With Memory Snapshots

ADVERSARIES considered in this work are very powerful — no optimal N-processor solutions exist for the *Write-All* problem even if the processors have the ability of taking *instant memory snapshots*.

In this chapter we show that for any algorithm that implements an N-processor robust solution to the *Write-All* problem in either the no-restart fail-stop or the restartable failstop model, a *failure pattern* can be constructed that will cause the algorithm to perform a *superlinear* number of processing steps. That is, there are no work-optimal N-processor solutions in these models (in contrast, optimality can be achieved by exploiting parallel slack, i.e., using fewer than N processors).

The lower bound results apply to the worst case work of the deterministic algorithms, and the expected work of deterministic and randomized algorithms that are subject to dynamic *on-line* adversaries. These results hold even under the additional assumption that processors can read and locally process all the shared memory at unit cost. We also show that concurrent writes are necessary for the existence of robust algorithms — concurrent writes are an important source of redundancy in our approach.

4.1 Lower Bounds for the No-Restart Fail-Stop Model

We now present a lower bound that holds for the fail-stop PRAMs as well as for a much stronger model where the processors can take unit time *memory snapshots*, i.e., processors can read and locally process the entire shared memory at unit cost.

We first list three simple mathematical lemmas (whose proofs are given in Appendix A), then state the main lower bounds theorem and its proof.

Lemma 4.1 Given a sorted list of m ($m > 1$) nonnegative integers $a_1, a_2, \ldots, a_m$ then we have for all j ($1 \leq j < m$) that $\left(1 - \frac{j}{m}\right) \sum_{i=1}^{m} a_i \leq \sum_{i=j+1}^{m} a_i$.

Lemma 4.2 Given $G > 1, N > G$, and integer σ such that $\sigma < \frac{\log N}{\log G} - 1$, then the following inequality holds: $\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor /G\rfloor \ldots /G\rfloor}_{\sigma \text{ times}} > 0$ (where σ is the number of divisions by G).

Lemma 4.3 For $N \to \infty$: $\left(1 - \frac{1}{\log N}\right)^{\frac{\log N}{\log \log N}} = 1 - \frac{1}{\log \log N} + O\left(\frac{1}{(\log \log N)^2}\right)$.

In the theorem, we will make use of Lemma 4.1 with $G = \log N$.

Theorem 4.4 Given any (deterministic or randomized) N-processor CRCW PRAM algorithm that solves the *Write-All* problem, the adversary can force fail-stop errors that result in $\Omega(N \frac{\log N}{\log \log N})$ steps being performed by the algorithm, even if the processors can read and locally process all shared memory at unit cost.

Proof: We are going to present a strategy for the adversary that results in this worst case behavior. Let A be the best possible algorithm that implements a robust solution for the *Write-All* problem. Each processor participating in the algorithm is allowed to read the entire shared memory, and locally perform arbitrary computation on it in unit time.

Let $P_0 = N$ be the initial number of processors, and $U_0 = N$ be the initial number of unvisited array elements. The strategy of the adversary is outlined below. Step numbers refer to the PRAM steps (not to be confused with block-steps or loop-iterations used

in Section 3.2.1). For each step, the adversary will be determining what processors are going to write to what shared memory locations.

Step 1: The adversary chooses $U_1 = \lfloor U_0/\log U_0 \rfloor$ array elements with the least number of processors assigned to them. This can be done since the adversary knows all the actions to be performed by A. The adversary then fail-stops the processors assigned to these array elements, if any.

To estimate the number of surviving processors and to express this mathematically, we will be using Lemma 4.1 with the following definitions:

Let $m = U_0$, and let $a_1, \ldots, a_m$ be the sorted in ascending order quantities of processors assigned to each array element, moreover, let a_m also include the quantity of any un-assigned processors (i.e., a_1 is the least number of processors assigned to an array element, a_2 is the next least quantity of processors, etc.). Let $j = U_1$. Thus the adversary failed exactly $\sum_{i=1}^{j} a_i$ processors. The initial number of processors is: $\sum_{i=1}^{m} a_i = P_0$, therefore, the number of surviving processors P_1 is: $\sum_{i=j+1}^{m} a_i = P_1$. Using Lemma 4.1, we get:

$$P_1 \geq (1 - U_1/U_0)P_0$$

or, after substituting for U_1 and using the properties of *floor*:

$$P_1 \geq \left(1 - \frac{\lfloor U_0/\log U_0 \rfloor}{U_0}\right) P_0 \geq \left(1 - \frac{1}{\log U_0}\right) P_0$$

Step 2: The adversary again chooses among the U_1 remaining unvisited array elements $U_2 = \lfloor U_1/\log U_0 \rfloor$ elements with the least number of processors assigned to them. Using Lemma 4.1 again in a similar way:

$$\begin{aligned} P_2 &\geq \left(1 - \frac{\lfloor U_1/\log U_0 \rfloor}{U_1}\right) P_1 = \left(1 - \frac{\lfloor \lfloor U_0/\log U_0 \rfloor / \log U_0 \rfloor}{\lfloor U_0/\log U_0 \rfloor}\right) P_1 \\ &\geq \left(1 - \frac{\lfloor U_0/\log U_0 \rfloor / \log U_0}{\lfloor U_0/\log U_0 \rfloor}\right) P_1 = \left(1 - \frac{1}{\log U_0}\right) P_1 \geq \left(1 - \frac{1}{\log U_0}\right)^2 P_0 \end{aligned}$$

Step i: The adversary chooses among U_{i-1} unvisited array elements $U_i = \lfloor U_{i-1}/\log U_0 \rfloor$ elements with least number of processors assigned to them. Again, applying Lemma 4.1:

$$P_i \geq \left(1 - \frac{\lfloor U_{i-1}/\log U_0 \rfloor}{U_{i-1}}\right) P_{i-1} \geq \left(1 - \frac{1}{\log U_0}\right) P_{i-1} \geq \left(1 - \frac{1}{\log U_0}\right)^i P_0$$

This process is repeated for as long as there are any unvisited array elements, at which point the surviving processors will successfully terminate the algorithm. Let ρ be the step at which the last unvisited element is finally visited. Let us use Lemma 4.2 with $G = \log N$ and σ the largest integer such that $\sigma < \log N/\log\log N - 1$. Then

$U_\sigma = \lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor/G\rfloor \ldots /G\rfloor}_{\sigma \text{ times}} > 0$, and so ρ must be greater than σ because $U_\rho = 0$. Thus we have $\rho \geq \dfrac{\log U_0}{\log\log U_0} - 1 = \dfrac{\log N}{\log\log N} - 1 > \sigma$.

We want to estimate $S = \sum_{i=0}^{\rho} P_i$. By the adversary strategy given above, for all PRAM steps i: $P_i \geq (1 - \frac{1}{\log U_0})^i P_0$. Therefore:

$S = \displaystyle\sum_{i=0}^{\rho} P_i \geq \sum_{i=0}^{\rho} \left(1 - \frac{1}{\log N}\right)^i P_0$; Using the summation of a geometric progression we obtain:

$$\begin{aligned} S &\geq P_0 \frac{1 - (1 - \frac{1}{\log N})^\rho}{1 - (1 - \frac{1}{\log N})} = P_0 \log N \left(1 - \left(1 - \frac{1}{\log N}\right)^\rho\right) \\ &\geq P_0 \log N \left(1 - \left(1 - \frac{1}{\log N}\right)^{\frac{\log N}{\log\log N}}\right) \end{aligned}$$

Using the result of Lemma 4.3: $(1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}} = 1 - \frac{1}{\log\log N} + O(\frac{1}{(\log\log N)^2})$, we obtain the following lower bound on the number of PRAM processor steps:

$$\begin{aligned} S &\geq P_0 \log N \left(1 - 1 + \frac{1}{\log\log N} + O\left(\frac{1}{(\log\log N)^2}\right)\right) \\ &= P_0 \frac{\log N}{\log\log N} + O\left(P_0 \frac{\log N}{(\log\log N)^2}\right) \end{aligned}$$

Therefore $S = \Omega(N \dfrac{\log N}{\log\log N})$. □

We use this result in Section 3.2.1 to exhibit a processor failure pattern for algorithm W that results in the worst case behavior of algorithm W that corresponds to work $S = \Theta(N \log^2 N / \log\log N)$.

Remark 4.1 The lower bound of $\Omega(N \frac{\log N}{\log\log N})$ is the strongest possible bound for the fail-stop model without restarts under the memory snapshot assumption. This can be shown in a straightforward way by adapting the analysis of algorithm W by Martel [71] (Theorem 3.4bis in Appendix A.6). 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 memory snapshot assumption.

We close this section with a comment on the knowledge of the adversary used in the proofs. The adversary dynamically fails processors based solely on their intent to write into the array to be initialized by the algorithm. The adversary uses no knowledge

about the methods that the algorithms use in selecting the array elements to write into. Therefore the worst case behavior caused by such adversary will also apply to the cases where the algorithm uses a stronger PRAM model (such as the *ideal* PRAM of Beame and Hastad [20]) with arbitrarily powerful instruction set or the cases where the algorithm makes probabilistic decisions (such as using a coin toss).

On the other hand, when the adversary is limited as in Martel et al. [75] and Kedem et al. [59] by using off-line, oblivious adversaries, or the adversaries that are limited stochastically, better expected work has been reported for CRCW PRAMs.

4.2 Lower Bounds for the Restartable Fail-Stop Model

As we have shown in Example 2.4 in Section 2.6, 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. When the update cycle accounting is introduced, we showed in Section 3.3.1 that there is a sub-quadratic solution. With the update cycle accounting we now show $N - P + \Omega(P \log P)$ work lower bound (when $P \leq N$), 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 4.5 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 $S \geq N - P + \lfloor \frac{P}{2} \rfloor \log P = N - P + \Omega(P \log P)$. □

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 4.6 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 4.5 works even if failed processors do not lose local state, and so the same strategy will work in the strongly asynchronous model. □

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 Section 3.2.2 when we analize algorithm V.

Theorem 4.7 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:

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

□

Finally, the lower bounds in this section apply, just as the results for the non-restartable model, to the worst case work of the deterministic algorithms, and to the expected work of the deterministic and the randomized algorithms subject to the worst case dynamic on-line adversaries (that cannot affect a coin toss or a random number selection).

4.3 Other bounds

4.3.1 A lower bound for CREW PRAM

In the absence of failures, any P-processor CREW (concurrent read exclusive write) or EREW (exclusive read exclusive write) PRAM can simulate a P-processor CRCW

PRAM with only a factor of $O(\log P)$ more parallel work [58]. We now show that a more severe difference exists between CRCW and CREW PRAMs (and thus also EREW PRAMs) when the processors are subject to failures.

Theorem 4.8 Given any (deterministic or randomized) N-processor CREW PRAM algorithm that solves the *Write-All* problem, the adversary can force fail-stop errors that result in $\Omega(N^2)$ steps being performed by the algorithm, even if the processors can read and locally process all shared memory at unit cost.

Proof: To prove this, we first define an auxiliary *Write-One* problem as follows: *Given a scalar variable s whose value is initially 0, store 1 in this variable.*

Let B be the most efficient asymptotically CREW algorithm that solves the *Write-One* problem that is able toread and process all shared memory at unit cost. Such an algorithm is no more efficient asymptotically than the algorithm below that utilizes an *oracle* to predict the best selection of a processor that is chosen to exclusively write the value 1:

```
forall processors PID=1..N parbegin
    shared integer s;
    while s = 0 do
        if PID = Oracle() then s := 1 fi
    od
parend
```

To exhibit the worst case behavior the adversary fails the processor that was selected by the *Oracle*() to perform the exclusive write until a single processor remains. The remaining processor is then allowed to write 1 by the adversary. Clearly $S = \Omega(N^2)$.

Finally, it can be shown by a simple reduction to the *Write-One* problem that any N-processor CREW solution to the *Write-All* problem has the worst case of $S = \Omega(N^2)$. □

For the CREW PRAMs, Martel and Subramonian show a randomized *Write-All* algorithm in [73] that, when adversary is oblivious and non-adaptive, has expected work of only $O(N \log N)$ using N processors, and expected work (N) when using $N/\log N$ processors. Thus it appears that adaptive adversary are significantly more powerful than the oblivious adversaries as far as randomized CREW algorithms are concerned.

4.3.2 Lower bounds with test-and-set operations

Under certain assumptions on the way that memory is accessed in the strongly asynchronous model, a different lower bound is shown by Buss et al. in [27]. Assume 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.

Theorem 4.9 [27] 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 $P \geq 3$.

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. The result also applies to the fail-stop restartable model, when each update cycle accesses only one array element used by the *Write-All* problem.

Chapter 5

Algorithm Simulations and Transformations

IN THIS chapter we develop a general technique for efficient simulation of *any* PRAM algorithm by a fail-stop CRCW PRAM using the *Write-All* paradigm. We formulate a *universal PRAM interpreter (UPI)*, then develop a fault-tolerant *UPI* that can execute any PRAM algorithm by storing the programs and processor registers in shared memory. Our simulation is based on executing individual PRAM computation steps using the *Write-All* paradigm in such a way 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.

The fault-tolerant algorithms are executed on CRCW PRAMs whose processors are subject to fail-stop errors. As we have shown in Chapter 4 on lower bounds, the CREW (concurrent read, exclusive write) model is not sufficient due to the fact that very simple adversaries can cause at least a quadratic amount of work for any *Write-All* solution, even if the restarts are not allowed.

We will show that in the no-restart fail-stop model the algorithms that can be made fault-tolerant include the following models: WEAK (only zeros can be written concurrently), COMMON (concurrent writes of identical values are permitted), ARBITRARY (some single processor succeeds), PRIORITY (highest numbered processor succeeds), and STRONG (processor writing the largest value succeeds). In the restartable model, we can make fault-tolerant all of the above, except for the PRIORITY PRAM (for detailed

surveys of PRAM variations see [40, 58]).

For the no-restart model, we show that the simulation of PRAM algorithms can be done using optimal work in the presence of arbitrary fail-stop errors. This is accomplished using our simulation together with algorithm W that is optimal on inputs of size N when using P processors such that W is in its range of optimality. The resulting fault-tolerant execution does not degrade the asymptotic efficiency of the source algorithm. For the restartable model we show a strategy that is work-optimal when the number of simulating processors is P is within the optimality range of algorithm V and the total number of failures per each simulated N processor step is $O(P \log N)$.

We also show that in some cases it is possible to develop fault-tolerant algorithms that improve on the efficiency of general but "naïve" simulations of these algorithms.

Finally, we briefly discuss some parallel efficiency classes and "closures" with respect to fault tolerance.

5.1 General Parallel Assignment

In this section we introduce the basic techniques that are going to be used in the PRAM simulations.

Given a solution for the *Write-All* problem, it can readily be used as a building block for transforming efficient parallel algorithms into robust ones. The techniques used are illustrated by producing a robust *general parallel assignment* in the following example:

Example 5.1 *General parallel assignment:* Consider computing and storing in an array $x[1..N]$ the values of a function f that depend on the processor numbers PID and the initial values of the array x. Also, for simplicity, assume f can be computed in $O(1)$ sequential time.

```
forall processors PID = 1..N parbegin
    shared integer array x[1..N];
    x[PID] := f(PID, x[1..N])
parend
```

We convert the assignment to a form that remains correct when processors fail and when multiple attempts are made to execute the assignment, assuming there are means

for reassigning surviving processors that have accomplished their initial task. This is done using binary version numbers and two generations of the array:

```
forall processors PID = 1..N parbegin
    shared integer array x[0..1][1..N];
    bit integer v;
    x[v + 1][PID] := f(PID, x[v][1..N]);
    v := v + 1
parend
```

Here, v is the current bit (**mod** 2) version number (or tag), so that $x[v][1 \ldots N]$ is the array of current values. Function f will use only these values of x as its input. The values of f are stored in $x[v+1][1 \ldots N]$ creating the next generation of array x. After all the assignments are performed, the binary version number is incremented (**mod** 2).

At this point, a simple transformation of a solution to the *Write-All* problem, with the general parallel assignment replacing the trivial "$x[i] = 1$" assignment, will yield a robust N-processor algorithm. □

The preceding example directly yields the following:

Proposition 5.1 The asymptotic work complexities of solving the *general parallel assignment* problem and the *Write-All* problem are equal. □

Similarly to the *general parallel assignment*, any *Write-All* solution can be directly used to zero an array of size N, copy N computed values from one array to another, ro perform any other single step PRAM computations.

Write-All algorithms usually assume that an amount of shared memory proportional to either the number of processors P or the problem size N is available, and that it is initialized to zero (we relax this assumption in Section 6.1). This is true of all known *Write-All* solutions [8, 27, 55, 56, 59, 61, 75], and this also applies to algorithms that can be adapted to serve as an *Write-All* solution, e.g., [29]. Furthermore, iterative use of the *Write-All* technique in a single algorithm requires that the heaps contain zeroes at the start of each iteration. This can be accomplished by utilizing three identical sets of the heaps. One is used in the main algorithm, another set of heaps is used for zeroing the other two in a *Write-All* style, and the third set is saved for the next use of the

zeroing clean-up step. This step does not affect the asymptotic efficiency, and we will assume that the heaps are appropriately cleared in the rest of this section. We are going to revisit this technique in more detail in the next two sections.

5.2 A PRAM Interpreter

PRAM programs are normally presented as high level programs that can be compiled into assembly level instructions using conventional techniques (see a discussion in Wyllie's thesis [102]). As is the case with sequential processors, the instructions are stored in memory, the address of an instruction to be executed next is stored in an *instruction counter* register, and in order to execute an instruction, it is fetched into an *instruction buffer*. When control structures such as **while-do** and **if-then-else** are used, they the branching of control is compiled as assignments to instruction counters. Other processor private memory cells are stored in general purposes *registers*. Here we will use a formalization of the definition of PRAM such as the one used by Karp and Ramachandran in [58]. Informally, PRAM instructions consist of three synchronous cycles:

1. *Read cycle*: a processor reads a value from a location in shared memory into private memory,
2. *Compute cycle*: a processor performs a computation using private memory,
3. *Write cycle*: a processor writes a value from a location in private memory to a location in shared memory.

To formalize PRAM programs that are specified in terms of a PRAM "machine language", we formulate a definition given in Figure 5.1 in conjunction with the code for a PRAM interpreter. The simple PRAM program in Definition 5.1 in the figure implements the synchronous computation performed by a PRAM.

The number of registers per processor, $l = |\mathtt{r}|$, is typically constant for uni-processors, however in parallel processing it is important to provide each processor with larger private memories in order to allow them to perform as much computation as possible without having to access the shared memory. We will consider private memories of sizes up to $O(\log^k N)$ for some constant k. This does not diminish the computational power of the model, since if an algorithm assumes a larger than available private memory, a dedicated portion of shared memory can be used by such algorithms.

```
01    forall processors PID=1..P parbegin
02        shared SM[1..M]; --shared memory
03        shared PROG[1..P,1..size]; --P programs of length size
04        private IB --instruction buffer
05        private r record IC, RR, WR, ... end --private registers
06        r.IC := 1; --start at the first instruction
07        while r.IC ≠ 0 do --while not HALT
08            IB       := PROG[PID,r.IC]; --fetch of <R() C() W() J()> into IB
09            r.RR     := SM[R(r)]; --read cycle
10            r        := C(r);     --compute cycle
11            SM[W(r)] := r.WR;     --write cycle
12            r.IC     := J(r);     --next instruction
13        od
14    parend
```

Definition 5.1 A P-processor PRAM program on inputs of size N is defined as follows:

1. The P processors have unique identifiers PID in the range 1 to P.
2. Each processor has an instruction buffer IB, and a set of internal registers collectively referred to as the record `r`. The number of registers per processor, $|\mathtt{r}|$, is l. The registers in `r` include an instruction counter IC, a read register RR and a write register WR used for reads/writes from/to shared memory.
3. Q uses shared memory cells SM[1..m] for some m, with the first N cells containing the input. Shared memory cells and registers are capable of storing $O(\log \max\{N, P\})$ bits each.
4. Program instructions are stored in a shared array PROG[1..P,1..$size$], where $size$ is a constant. Program for processor i is in PROG[i,1..$size$]), one instruction per array element.
5. Instructions consist of four encoded operations <R() C() W() J()>. After reading an instruction into IB, a processor interprets this code as follows (line numbers refer to Figure 5.1):

   Read cycle (line 09): read into RR the contents of shared memory location R(r),

   Compute cycle (line 10): assign to registers in r the result of computation C(r),

   Write cycle (line 11): write the contents of WR to shared memory location W(r),

   Update instruction counter (line 12): compute next instruction address; IC = 0 is a halt.

□

Figure 5.1: Universal PRAM Interpreter.

Also note that we allow for processors used by a PRAM program to read and update the entire private memory of size l in unit time. When simulating PRAMs on fail-stop PRAMs, the fail-stop processors will not need and do not use this capability. This allows for the PRAM programs to be more flexible, without imposing any restrictions on the fail-stop processors used to simulate these programs.

In Definition 5.1, R(), W() and J() are expressions involving registers, and C() is the code for individual processors' compute cycles. PRAM programs are executed by the *UPI* that accepts a PRAM program and its input as data.

5.3 General Simulations on Fail-stop Processors Without Restart

We say that a PRAM algorithm is *fault-tolerant* if it completes its task in the presence of arbitrary fail-stop errors. We say that a PRAM algorithm is simulated by a fault-tolerant PRAM algorithm when both algorithms exhibit identical input/output behavior. Such simulation is *robust* if it is *efficient*, and it is *optimal* if the efficiency of the simulating algorithm (measured as work) is within a constant factor of the efficiency of the simulated algorithm:

Definition 5.2 Let $\mathcal{M}$ be a simulation of a P-processor PRAM algorithm Q, whose parallel-time on inputs of size N is less than or equal to $\tau(N)$, by a fault-tolerant P'-processor algorithm Q'.

(1) $\mathcal{M}$ is *robust*, if Q' has $S = O(P \cdot \tau(N) \cdot \log^c N)$ for a constant c, and

(2) $\mathcal{M}$ is *optimal*, if Q' has $S = O(P \cdot \tau(N))$. □

Remark 5.1 Note that a *robust simulation* of an algorithm is not the same as a *robust algorithm*. This is because robustness is defined in relation to the best sequential algorithm, and not in relation to the work of an arbitrary parallel algorithm which may be inefficient. However a robust simulation of an efficient algorithm yields a robust algorithm.

Finally, it is relatively easy to construct robust simulations using a small number of processors, e.g., using P processors to simulate N processors when P is polylogarithmic

in N. We show next that this is possible for P in $1 \leq P \leq N$, and optimally so for P in $1 \leq P \leq N/\log^2 N$.

The *UPI* in Figure 5.1 models arbitrary computation performed by a PRAM, and in this section we develop an efficient fault-tolerant *UPI*.

Let us define $S_w(X, P)$ to be the cost of solving the *Write-All* problem of size X, using P ($P \leq X$) processors. We measure this cost as S, and we will make use of the Property 2.5 that states that $S_w(X, P') \leq S_w(X, P)$ when $P' \leq P$.

Now the main lemma.

Lemma 5.1 Let Q be a P-processor COMMON (ARBITRARY) CRCW PRAM algorithm that uses m shared and $l = O(P \log^k N)$ (constant k) local memory on inputs of size N. Q can be simulated on a P'-processor ($P' \leq P$) fail-stop COMMON (ARBITRARY) CRCW PRAM using $O(m + l)$ shared memory, at the cost of $S = O(S_w(P, P') \cdot \log^k N)$ per parallel PRAM step.

Proof: The desired result is achieved by constructing a robust version of *UPI* of Figure 5.1. Figure 5.2 illustrates an intermediate step of the construction, and Figure 5.3 is the final pseudo-code for the robust *UPI*. The proof consists of the following six steps: (1) local memory is stored in shared memory, (2) shared memory is split into two generations, "current" and "future", (3) PRAM step computations are changed to use current memory as input and use future as output and as a computation scratch-pad, (4) current and future memories are reconciled to produce new current memory, (5) instruction counters are examined to detect termination, and finally, (6) each of the modified groups of actions is placed in the work phase of a *Write-All* algorithm. Now the details.

First, we have to store processor local memory in shared memory, since after processors fail, the local memory is lost. We observe that l local memory cells can be stored in shared memory without affecting program semantics. To do this, registers are subscripted by PID (lines 04-05, all lines refer to Figure 5.2).

We then separate all shared memory and registers into two generations: current (using subscript 0) and future (subscript 1) (lines 02 and 05). All memory references are now made using the generation subscript 0 or 1 (lines 10-19). This does not affect

```
01 forall processors PID=1..P parbegin
02   shared SM[0..1,1..M];          --two generations of shared memory
03   shared PROG[1..P,1..size];     --P programs of length size; one generation
04   shared IB[1..P];               --instruction buffers, shared
05   shared r[0..1,1..P] record IC, RB, WB, ... end  --two sets of shared registers
06    --Initialize instruction counters
07   r[0,PID].IC := 1;  --start at the first instruction
08   while r[0,PID].IC ≠ 0 do  --while not HALT
09     --Tentative computation
10       IB[PID]              := PROG[PID,r[0,PID].IC]; --fetch <R() C() W() J()>
11       r[1,PID]             := r[0,PID];              --copy registers to scratchpad
12       r[1,PID].RB          := SM[0,R(r[1,PID])];     --read cycle
13       r[1,PID]             := C(r[1,PID]);           --compute cycle
14       SM[1,[W(r[1,PID])]   := r[1,PID].WB;           --write cycle
15       r[1,PID].IC          := J(r[1,PID]);           --next instruction
16     --Reconcile shared memory
17       SM[0,W(r[1,PID])]    := SM[1,W(r[1,PID])];
18     --Reconcile registers
19       r[0,PID]             := r[1,PID]
20   od
21 parend
```

Figure 5.2: Modified *UPI* using two generations of shared memory.

the asymptotic memory requirement of size $O(m + l)$. This is done to assure that the memory that can be accessed by processors that have not yet completed a particular action (due to failures) is not changed. This allows the PRAM instructions to be restarted when active processors are re-allocated.

Next we group statements into four actions to compute future memory values and to reconcile current and future memories. The initialization of ICs takes place in line 07. The contents of the **while** loop of Figure 5.1 are grouped into 3 actions: the action on lines 09-15 performs the tentative PRAM step computation using current memory as input, and future memory as output and as a scratchpad; the action on lines 16-17 reconciles shared memory; and the action on lines 18-19 reconciles processor registers.

Now a *Write-All* algorithm is used with each of the four actions as work phases, see Figure 5.3, steps `a`, `b`, `c` and `d`. This assures that all actions of a given phase are performed before any of the actions of the next phase are attempted.

Algorithm W utilizes workspace memory of size $O(P)$ on inputs of size P using P' processors ($P' \leq P$). This memory initially contains zeroes. Consecutive use of the algorithm W in a single algorithm requires that this workspace is zeroed. This is

```
01    forall processors PID=1..P' parbegin
02        shared PROG[...], SM[...], IB[...], r[...], H1, H2, H3, HALT initial false;
03        a: Initialize ICs to 1 using H1; Clear H1 using H3;
04        while not HALT do --while not all processors halte d
05            b: Perform a tentative PRAM step using H1;
06                Clear H1 and H3 using H2;
07            c: Reconcile shared memory using H1;
08                Clear H1 and H2 using H3;
09            d: Reconcile registers using H1;
10                Clear H1 and H3 using H2;
11            e: Compute, using H1, HALT=false iff ∃PID such that IC ≠ 0;
12                Clear H1 and H2 using H3;
13        od
14    parend
```

Figure 5.3: Pseudo-code for a robust *UPI*.

accomplished by utilizing three interchangeable workspaces, call them *H1*, *H2* and *H3*. *H1* is used in the actual algorithm. *H2* is used in zeroing *H1* and *H3* using a *Write-All* algorithm. *H3* is saved for the next use of the zeroing cleanup step. *H2* and *H3* then alternate. This simple, but modular cleanup technique affects neither the overall asymptotic memory usage, nor the asymptotic efficiency of robust algorithms when the cleanup stages are interleaved with the computation stages (Figure 5.3).

Lastly, we need to check for the algorithm termination by verifying that all processors halted. This is done by computing shared HALT to be *true*, iff for all PIDs, IC=0. This can be accomplished in unit time on a CRCW PRAM in the absence of failures. Here we compute HALT as follows: it is initialized to *true*, then using the *Write-All* technique, processors examine the simulated ICs and execute "HALT:=*false*" for each IC $\neq 0$ (Figure 5.3, step `e`).

In this simulation, due to failures, the synchronous tentative PRAM step computations may occur asynchronously (in the sense that the synchronous PRAM steps of the simulated processors can be performed at different times by the simulating processors). However since no processor reads or writes registers of other processors, and since the program PROG[...] is either COMMON or ARBITRARY CRCW, it does not matter in what order the shared memory is written by the simulated processors. Therefore the simulated PRAM steps will write values to shared memory in a way that is consistent with both the COMMON and ARBITRARY models.

To analyze the complexity of the resulting fault-tolerant computation, we first examine the use of registers. In tentative computation and reconciliation of registers in Figure 5.2, each PRAM processor may need to compute the values of, and copy the shared memory that represents a processor's registers as the result of interpreting the computation C(). This must be done sequentially by each processor, thus incurring a polylogarithmic in N multiplicative overhead. However this overhead still falls within the definition of robustness (Definition 2.6).

The complexity of applying this technique to a single PRAM step is bounded by the complexity of solving the *Write-All* problem in steps `a`, `c` and `e` of Figure 5.3, plus the complexity of robustly writing and copying $l = O(P \log^k N)$ shared memory in steps `b` and `d`. To copy l memory, we apply the *Write-All* technique l/P times at the cost of $S_w(P, P')$ per application. Therefore, the total cost per single PRAM step is

$$S = O[S_w(P,P') + (l/P)S_w(P,P')] = O[S_w(P,P') + \log^k N \cdot S_w(P,P')]$$

$$= O(S_w(P,P') \cdot \log^k N).$$

□

A construction for PRAM simulation, similar to the one used in Lemma 5.1 was independently developed by Kedem et al. in [59] using the *Write-All* technique to simulate any COMMON or ARBITRARY CRCW PRAM algorithm that uses arbitrary shared and no local memory. That construction does not allow any local memory, while we allows up to poly-log local memory. Note that we allow the processors to update the local memory in *unit time*. This makes our simulation nominally more general because the use of local memory is not mandated but is allowed up to a certain limit, while [59] does not allow any local memory. In [59] it is also observed that since P processors can only read P and write P shared memory cells in a single PRAM step, the overhead in shared memory need only be $O(P)$ when processors have no local memory. The results that follow do not take advantage of this optimization.

Theorem 5.2 Any P-processor PRAM (EREW, CREW, and WEAK, COMMON, ARBITRARY, PRIORITY or STRONG CRCW) algorithm that uses arbitrary shared and polylogarithmic in the input size local memory can be robustly simulated on a fail-stop P'-processor COMMON or ARBITRARY CRCW PRAM, when $P' \leq P$.

Proof: We use any robust *Write-All* solution, e.g., algorithm W. For COMMON or ARBITRARY CRCW PRAMs, Theorem 3.5 establishes $S_w(P, P') = O(P \log^2 P)$. Then the proof follows from Lemma 5.1 and Definition 5.2(1).

Correct EREW (exclusive read, exclusive write), CREW and WEAK CRCW PRAM programs can be directly executed on a CRCW PRAM without changing the program semantics. Therefore the technique of Lemma 5.1 can also be used with CREW and EREW algorithms that are to be executed on a CRCW PRAM. Thus the result holds for EREW, CREW, and WEAK CRCW models.

To extend the simulation to stronger CRCW models such as PRIORITY and STRONG, we use an efficient algorithm transformation technique that preserves algorithms' efficiency to within a logarithmic in the number of processors factor as shown in [40] (attributed to folklore): "A parallel computation that can be performed in time τ on a P-processor STRONG CRCW PRAM, can also be performed in time $\tau \log P$ using P EREW processors". Therefore we first preprocess PRIORITY and STRONG PRAM algorithms, and then robustly simulate the transformed algorithms on fail-stop COMMON or ARBITRARY CRCW PRAMs. □

To achieve optimal simulation, we need to use an optimal *Write-All* solution. For the purposes of the proof we are going to use the algorithm W within its range of optimality:

Theorem 5.3 Any P-processor PRAM algorithm that uses arbitrary shared and constant local memory per processor can be optimally simulated on a fail-stop P'-processor CRCW PRAM, when $P' \leq P/\log^2 P$. EREW, CREW, and WEAK and COMMON CRCW PRAM algorithms are simulated on fail-stop COMMON CRCW PRAMs; ARBITRARY, PRIORITY and STRONG CRCW PRAMs are simulated on fail-stop CRCW PRAMs of the same type.

Proof: For the optimality result we use Theorem 3.7 that establishes $S_w(P, P/\log^2 P) = O(P)$. We also eliminate the overhead of copying local memories by allowing constant local memory per processor. In this case the cost of copying local memory is absorbed by the cost of optimal simulation of PRAM instructions by using Lemma 3.1 with $k = 0$ (constant local memory). This proves the result for for EREW, CREW, and WEAK, COMMON and ARBITRARY CRCW PRAMs.

To prove the result for PRIORITY PRAM, we need to show that (1) when two or more PRIORITY processors write concurrently, then the processor that simulates the highest numbered processor will succeed, and (2) lower numbered processors do not overwrite cells written by higher numbered processors during several phases of the single PRAM step simulation.

To show (1) it is sufficient to demonstrate that the simulation has the *processor allocation monotonicity* property. This property is defined as follows: if PRAM steps of processors with PIDs p_1 and p_2 (without loss of generality let $p_1 < p_2$) are simulated respectively by processors with PIDs p'_1 and p'_2, then $p'_1 < p'_2$. This is assured when using algorithm W (or algorithm V) as the *Write-All* solution, since the algorithm has this property as we have shown in Section 3.2.3.

Property (2) can be assured using auxiliary storage as in the transformation in Eppstein and Galil [40]: before writing, processors first write the PID of the simulated processor and then the data, but only if the previously written PID is lower.

For the simpler case of STRONG PRAM, the concurrent writes are properly handled by zeroing the future memory cells to which processors will write, and then performing writes only if the value in the cell is smaller than the value to be written (without loss of generality use unsigned integers). This assures the correctness of asynchronous writes. Synchronous writes are properly handled by the STRONG PRAM itself since regardless of the processor used, the writes of larger values will succeed. □

5.4 General Simulations on Restartable Fail-Stop Processors

We now extend the results presented in the previous section to the restartable fail-stop model. We begin by formally stating the main result for a deterministic simulation of any N-processor synchronous PRAM on P restartable fail-stop processors ($P \leq N$), and then discuss its proof.

Theorem 5.4 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 5.2 PRIORITY CRCW PRAMs cannot be directly simulated using the same framework, for one of the algorithms used (namely algorithm X in Section 3.3.1) does not possess the *processor allocation monotonicity* property that assures that higher numbered processors simulate the steps of the higher numbered original processors.

In the previous section we had shown in Lemma 5.1 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. That result also holds in the restartable fail-stop model, since the proof of Lemma 5.1 does not utilize the knowledge that there are no restarts.

Here we describe how algorithms V and X' are combined with the framework we have established to yield efficient executions of PRAM programs on PRAMs that are subject to stop-failures and restarts.

Theorem 5.5 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 2.9 of σ and S^+. □

Application of the simulation techniques from the previous section 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 5.4 follows from Theorem 5.5, Lemma 5.1 and Theorem 5.2. The following corollaries are also interesting:

Corollary 5.6 Under the hypothesis of Theorem 5.4, 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 next result gives additional insight into the efficiency of our solution:

Corollary 5.7 Under the hypothesis of Theorem 5.4:

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

Thus the overhead efficiency σ 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 5.8 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 $S = O(\tau \cdot N)$.

Of course it is also true 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.

5.5 Improving Oblivious Simulations

In addition to serving as the basis for oblivious simulations, any solution for the *Write-All* problem can also be readily used as a building block for custom transformations of efficient parallel algorithms into robust ones [55]. Custom transformations are interesting because in some cases it is possible to improve on the work of the naïve oblivious simulation. These improvements are most significant for fast algorithms when a full range of processors is used, i.e., when N are used to simulate N processors, because in this case the parallel slack cannot be taken advantage of. For example in the models with clear initial memory, a factor of $\log N / \log\log N$ was saved off the pointer doubling simulations [55], and using randomization and off-line adversaries, improvements can be obtained in expected work of other algorithms [72, 75].

Using the general simulation techniques, such as [59, 75, 92], if $S_w(N,P)$ is the efficiency of solving a *Write-All* instance of size N using P processors, then a single N-processor PRAM step can be simulated using P fail-stop processors and work $S_w(N,P)$. Thus if the *Parallel-time* $\times$ *Processors* of an original N-processor algorithm is $\tau \cdot N$, then the work S of the fault-tolerant version of the algorithm will be no better than $O(\tau \cdot S_w(N,P))$.

One immediate result that improves on the general simulations follows from the fact that algorithms V, W and X, by their definition, implement an associative operation on N values.

Proposition 5.2 Given any associative operation $\oplus$ on integers, and an integer array $x[1..N]$, it is possible to robustly compute $\bigoplus_{i=1}^{N} x[i]$ using P fail-stop processors at a cost of of a single application of any of the algorithms V, W or X.

This saves a full $\log N$ factor for all simulations. The savings are also possible for the important prefix sums and pointer doubling algorithms.

5.5.1 Parallel prefix

We now show how to obtain deterministic improvements in work for the prefix sums algorithm that occurs in solutions of several important problems [21]. Efficient parallel algorithms and circuits for computing prefix sums were given by Ladner and Fischer in

[64], where the *prefix problem* is defined as follows: Given an associative operation $\oplus$ on a domain $\mathcal{D}$, and $x_1, \ldots, x_n \in \mathcal{D}$, compute, for each k, $(1 \leq k \leq n)$ the sum $\bigoplus_{i=1}^{k} x_i$.

In order to compute the prefix sums of N values using N processors, at least $\log N / \log\log N$ parallel steps are required [20, 67], and the known algorithms require at least $\log N$ steps. Therefore an oblivious simulation of a known prefix algorithm will require simulating at least $\log N$ steps. When using $P = N$ processors, the work of such simulation will be $O(S_w \cdot \log N)$. Here we extend Proposition 5.2 and show a robust prefix sum algorithm whose work complexity is $O(S_w)$, thus improving oblivious deterministic simulation by a factor of $\log N$.

In the no-restart fail-stop model we have the following result:

Theorem 5.9 Parallel prefix for N values can be computed using N non-restartable fail-stop processors using $O(N)$ clear memory with $S = O(N \log^2 N / \log\log N)$.

Proof: The prefix summation algorithm that we are going to use as the basis, is an iterative version of the recursive algorithm of [64]. The algorithm consists of two stages: (1) first a binary summation tree is computed, (2) then the individual prefix sums are computed from the summation tree obtained in the first stage. Each prefix sum requires no more than logarithmic number of additions.

Each stage can be performed in logarithmic time in parallel by up to N processors. To produce the robust version of the above algorithm, we use algorithm W twice to implement these two stages. Fir each stage the controls of algorithm W areused with appropriate modifications as follows:

1. A binary summation tree is computed in bottom up traversals at the same time when the progress tree of algorithm W is being updated. This modification to the algorithm does not affect its asymptotic complexity.

2. This stage uses the work phase of algorithm W modified to include the logarithmic time summation operations using the tree computed in stage 1.

   In the code, shown in Figure 5.4, $\langle\langle i \rangle\rangle$ is a binary string representing the value i in binary, where most significant bit is bit number 0, and $\langle\langle i \rangle\rangle_h$ is the true/false value of the h^{th} most significant bit of the binary string representing i. The loop

```
01    forall processors PID = 0..N parbegin
02        shared integer array sum[1..2N − 1]; −−summation tree
03        shared integer array prefix[1..N]; −−prefix sums
04        private integer j, j1, j2, −−current/left/right indices
05                        h; −−depth in the summation tree
06        j := 1; −−begin at the root,
07        h := 0; −−and at depth 0
08        prefix[PID] := 0; −−initialize the sum
09        while h ≠ 0 do −−traverse from root to leaf
10            h := h + 1
11            j1 := 2 ∗ j −−left index
12            j2 := j1 + 1 −−right index
13            if ⟨⟨PID⟩⟩_h −−Is the sub-sum at this level included?
14            then prefix[PID] := prefix[PID] + sum[j1] −−add the left sub-sum
15                 j := j2 −−go down to the right
16            else j := j1 −−go down to the left
17            fi ;
18        od
19    parend
```

Figure 5.4: Second stage of robust prefix computation.

in lines 09-18 is the top-down traversal of the summation tree. In lines 13-17 the appropriate subtree sum is added (line 14) at depth h only if the corresponding bit value of the processor PID is *true*.

□

Note that because of the lower bounds of Beame and Hastad [20] and Li and Yesha [67], at least $\log N/\log\log N$ parallel time and at least $N\log N/\log\log N$ work will be required by $P = N$ processors to compute the prefix sums in the absence of failures. Therefore the multiplicative overhead in work of our parallel prefix algorithm is only $\log N$ when using alogirthm W in the fail-stop model.

5.5.2 Pointer doubling

Another important improvement for the fail-stop case is a robust *pointer doubling* operation that is a basic building block for many parallel algorithms. This is accomplished using algorithm W, the most efficient to date algorithm for the fail-stop model.

```
01 forall processors PID=1..N parbegin
02   for 1..log(N)/loglog(N) do
03     --Perform a single stage to double each pointer at least log(N) times
04       Phase W3: Each processor doubles its leaf's pointer log(N) times.
05       Phase W4: Bottom up traversal to (under)estimate the no. of leaves visited
06       while the underestimate of the visited leaves is not N do
07           Phase W1: Perform bottom up traversal to enumerate remaining processors
08           Phase W2: Perform top down traversal to reschedule work
09           Phase W3: Each processor doubles its leaf's pointers log(N) times.
10           Phase W4: Perform bottom up traversal to measure progress made
11       od
12   od
13 parend
```

Figure 5.5: A high level view of the robust pointer doubling algorithm

Proposition 5.3 There is a robust list ranking algorithm for the fail-stop model with $S^* = O(\log N \cdot S_w(N,P)/\log\log N)$, where N is the input list size and $S_w(N,P)$ is the complexity of algorithm W for the initial number of processors $P : 1 \leq P \leq N$.

Proof: The robust algorithm is implemented using a variation of algorithm W, *general parallel assignment*, and the standard pointer doubling algorithm. A high level algorithm description is given in Figure 5.5, where Phases W1 through W4 refer to the phases of algorithm W.

We associate each list element with a progress tree leaf. Phase W3 uses the *general parallel assignment* approach to double pointers and update distances. As before, we use two generation of the arrays representing the pointers, and the running ranks of the list elements — one is used as the "current" values, and the other as the "next" values being computed. Binary tags are used to determine which generation is "current", and which is "next". The generations alternate as the computation progresses.

The $\log N$ pointer doubling operations in phase W3 makes it sufficient for the outmost **for** loop to iterate $\log N/\log\log N$ times, and it does not affect the complexity of the approach in algorithm W, since Phases W1, W2 and W4 take $\Theta(\log N)$ time. This results in $S = O(\frac{\log N}{\log\log N} S_w(P,N))$. □

The technique of Proposition 5.3 for the pointer doubling algorithm achieves a $\log\log N$ improvement in work over the naïve simulations, i.e., instead of $S = O(\log N \cdot$

S_w) we achieve $S = O(\log N \cdot S_w / \log\log N)$. This improvement can be used with several important robust algorithms that are based on pointer doubling:

Proposition 5.4 There is a robust parallel algorithm for computing the tree functions of Tarjan and Vishkin [96] with $S = O(\log N \cdot S_w(N,P)/\log\log N)$, where N is the input tree size and $S_w(N,P)$ is the complexity of algorithm W for the initial number of processors $P : 1 \leq P \leq N$.

The robust algorithms obtained using our technique are optimized for the worst case behavior in the presence of arbitrary fail-stop error patterns. These algorithms incur a $O(\log^2 N)$ *multiplicative overhead* relative to the source algorithm when using N processors, and this overhead is reduced to $O(\log N)$ in the absence of failures. However the robust list ranking algorithm can be tailored to yield $S = O(N \log N)$ in the absence of failures. This is accomplished by preceding the algorithm with $\log N$ pointer doubling operations and a phase 4 bottom-up traversal. This results in $O(N \log N)$ *additive* overhead. Therefore, for the algorithms that are dominated by pointer doubling with a cost of $\Theta(N \log N)$, e.g., Tarjan and Vishkin [96], there is no asymptotic degradation in the absence of failures. This optimization can be used with, for example, the sorting techniques such as Batcher [18] to reduce the overall multiplicative cost to $O(\log N)$ in the absence of failures over the $O(N \log^2 N)$ cost associated with sorting networks.

Finaly, when a *Write-All* solution is used within its range of optimality (by taking advantage of parallel slackness) as the basis for fault-tolerant algorithms, then we obtain fault-tolerant solutions to all of the above problems, such that the available processor steps S is asymptotically equal to the *Parallel-time* $\times$ *Processors* of the original algorithms.

5.6 On Parallel Complexity Classes and Fault-Tolerance

We have briefly mentioned in the discussion of efficiency measures in Chapter 2 that it is important for parallel algorithms to have efficient work both in the failure-free environment and when they are subject to failure. We have also remarked in this chapter that efficient simulations of parallel algorithms result in efficient algorithms only if the simulated algorithm is efficient to begin with. Here we address these topics further.

Many efficient parallel algorithms belong to the class $\mathcal{NC}$, however the inverse is not necessarily true. This is because the algorithms in $\mathcal{NC}$ allow for polynomial inefficiency in work. In $\mathcal{NC}$ the efficiency is characterized in terms of (polylogarithmic) time, but the computational agent can be large (polynomial) relative to the size of a problem [30, 81]. Further critique of the notion that $\mathcal{NC}$ class of algorithms is the class of efficient parallel algorithms is given by Kruskal et al. in [63].

In the context of fault-tolerant computation we suggest that while a definition of robustness can be made in terms of $\mathcal{NC}$, this definition is not very meaningful in terms of the resulting algorithm efficiency. An $\mathcal{NC}$ algorithm can be made fault-tolerant by clustering a polynomial number of processors and assigning them to the work that is normally performed by a single processor. The resulting algorithm is correct for as long as at least one processor remains active in each cluster. But clearly such algorithm is extremely inefficient, and such technique is extremely wasteful, even if the resulting algorithm still meets the $\mathcal{NC}$ criteria of efficiency, i.e., polylogarithmic time and and polynomial resources.

To reiterate: in order to characterize better the efficiency of parallel algorithms, the efficiency measures need to take into account both the parallel time and the size of the computational resource, i.e., parallel work. Such characterization of parallel algorithm efficiency are defined by Vitter and Simons in [100] and expanded on by Kruskal et al. in [63]. The efficiency classes defined in [63] are as follows:

Let A be a problem such that the (RAM) time complexity of the best known sequential algorithm is $T(N)$. A parallel algorithm that solves an N-size instance of A using $P(N)$ processors in $\tau(N)$ parallel time belongs to the class:

1. ENC if $\tau(N) = \log^{O(1)}(T(N))$ and $\tau(N) \cdot P(N) = O(T(N))$.
2. EP if $\tau(N) \leq T(N)^{\varepsilon}$ (const $\varepsilon < 1$) and $\tau(N) \cdot P(N) = O(T(N))$.
3. ANC if $\tau(N) = \log^{O(1)}(T(N))$ and $\tau(N) \cdot P(N) = T(N) \cdot \log^{O(1)}(T(N))$.
4. AP if $\tau(N) \leq T(N)^{\varepsilon}$ (const $\varepsilon < 1$) and $\tau(N) \cdot P(N) = T(N) \cdot \log^{O(1)}(T(N))$.
5. SNC if $\tau(N) = \log^{O(1)}(T(N))$ and $\tau(N) \cdot P(N) = T(N)^{O(1)}$.
6. SP if $\tau(N) \leq T(N)^{\varepsilon}$ (const $\varepsilon < 1$) and $\tau(N) \cdot P(N) = T(N)^{O(1)}$.

Analogously with our definition of *robustness*, the complexity characterizing these classes is defined with respect to the time complexity of the best sequential algorithm. There are two complexity criteria for each class: (a) parallel time $\tau(N)$ and, (b) parallel work $\tau(N) \cdot P(N)$.

In the next two subsections we define criteria via which we can evaluate whether our algorithm transformations preserve the efficiency of the algorithms in each the classes above.

In order to be able to use the time complexity of the original algorithm as a comparison metric, we need to introduce some measure of time for the fault-tolerant algorithms. In order to do that, we will use the maximum time that is required by the fault-tolerant algorithm to complete its computation provided that a linear number of processors are still active. For the fail-stop model this corresponds to an execution in which at least cP processors survive for some constant $c > 0$, while for the restartable model we ask that each update cycle is completed by at least cP processors.

If we do not make this assumption then the best we can conclude about the running time of the fault-tolerant versions is that it is at least the time of the best sequential algorithm, because time can be severely degraded when the remaining number of processors becomes small. For example, the algorithms become sequential when only one processor is active.

5.6.1 Fail-stop model without restarts

We first examine whether the classes of [63] are closed with respect to our fault-tolerant transformations in the fail-stop model. We assume that $P = P(N)$ processors are used. Also, if P is polynomial in N, then $\log P = O(\log N)$.

For any algorithm A, let $\xi(A)$ be the fault-tolerant algorithm that can be constructed using the techniques in this chapter (as either a simulation or a dedicated algorithm). We formulate the following definition:

Definition 5.3 Let $\mathcal{C}$ be a class in which the parallel time of algorithms is in the complexity class τ_C and the parallel work is in the complexity class w_C. We say that $\mathcal{C}$ is closed with respect to a fail-stop without restart fault-tolerant transformation ξ if for any algorithm A in $\mathcal{C}$:

Complexity Class	Time with $\geq cP$ processors $c_2\tau(N)\log^2 N/\log\log N$	Fail-Stop Work $c_2\log^{O(1)} N\cdot\tau(N)\cdot P(N)$	Closed under ξ?
ENC	$= O(\log^{O(1)}(T(N)))$	$> O(T(N))$	No
EP	$= O(T(N)^{\varepsilon})$	$> O(T(N))$	No
ANC	$= \log^{O(1)}(T(N))$	$= T(N)\cdot\log^{O(1)}(T(N))$	Yes
AP	$= O(T(N)^{\varepsilon})$	$= T(N)\cdot\log^{O(1)}(T(N))$	Yes
SNC	$= \log^{O(1)}(T(N))$	$= T(N)^{O(1)}$	Yes
SP	$= O(T(N)^{\varepsilon})$	$= T(N)^{O(1)}$	Yes

Table 5.1: Closure under the fail-stop transformation ξ (for $P = P(N)$).

1. the worst case work S of $\xi(A)$ is such that S is in w_C, and

2. the running time t is such that t is in τ_C when the minimum number of processors active during the computation is cP for some constant $c > 0$. □

An immediate observation is that $\mathcal{NC}$ is trivially closed with respect to our fault-tolerant transformations.

In the fail-stop model, using for example algorithm W as the basis for transforming non-fault-tolerant algorithms, we have the following:

- the multiplicative overhead in work is $O(\log N^2/\log\log N)$, and so if the work of the initial algorithm A is $\tau(N)\cdot P(N)$ then the worst case work of the fault-tolerant version $\xi(A)$ is $c_1\log^{O(1)} N\cdot\tau(N)\cdot P(N)$ for some constant $c_1 > 0$,

- algorithm W terminates in $O(S_w/cP) = O(\log^2 N/\log\log N)$ time when at least cP processors are active, therefore if the parallel time of algorithm A is $\tau(N)$, then the parallel time of execution for $\xi(A)$ using at least cP active processors is $c_2\tau(N)\log^2 N/\log\log N$ for some constant $c_2 > 0$,

The resulting closure properties of the classes defined in [63] under our fail-stop transformation is summarized in Table 5.1.

5.6.2 Restartable fail-stop model

We next examine whether the classes of [63] are closed with respect to our transformations in the restartable fail-stop model. Again we have $\log P = O(\log N)$.

For any algorithm A, let $\rho(A)$ be the restartable fault-tolerant algorithm that can be constructed using the techniques in this chapter We formulate the following definition:

Definition 5.4 Let $\mathcal{C}$ be a class in which the parallel time of algorithms is in the complexity class τ_C and the parallel work is in the complexity class w_C. We say that $\mathcal{C}$ is closed with respect to a restartable fail-stop fault-tolerant transformation ρ if for any algorithm A in $\mathcal{C}$:

1. the worst case overhead σ of $\rho(A)$ is such that $\sigma \cdot \tau(N) \cdot P(N)$ is in w_C, and
2. the running time t is such that t is in τ_C when the number of processors completing each update cycle of the computation is at least cP for constant $c > 0$. □

In the fail-stop restartable model we are going to take advantage of the existential result by Anderson and Woll in [8], who showed that for every $\varepsilon > 0$, there exists a deterministic algorithm for P processors that simulates P PRAM instructions with $O(P^{1+\varepsilon})$ work. This result was developed for the asynchronous model, but it also applies for fail-stop model with restarts.

In this model we will provide existential closure properties. Assume we have an algorithm such as the one characterized in the theorem above. This algorithm can be interleaved with algorithm V, for example, so that the overhead σ of the combined algorithm is $O(\log^2 N)$. When this combined algorithm is used as the basis for transforming non-fault-tolerant algorithms, we have the following:

- if the work of the initial algorithm A is $\tau(N) \cdot P(N)$ then $\sigma \cdot \tau(N) \cdot P(N) = \log^2 N \cdot \tau(N) \cdot P(N)$,
- a single *Write-All* step terminates in $O(P^{1+\varepsilon}/cP) = O(P^{\varepsilon})$ time when at least cP processors are active, therefore if the parallel time of algorithm A is $\tau(N)$, then the parallel time of execution for $\rho(A)$ using at least cP active processors is $c_2\tau(N) \cdot P^{\varepsilon}$ for some constant $c_2 > 0$,

Complexity Class	Time with $\geq cP$ processors $c \cdot \tau(N) \cdot P^{\varepsilon}$	Work $\log^2 N \cdot \tau(N) \cdot P(N)$	Closed under ρ?
ENC	$> O(\log^{O(1)}(T(N)))$	$> O(T(N))$	No
EP	$= O(T(N)^{\varepsilon})$	$> O(T(N))$	No
ANC	$> \log^{O(1)}(T(N))$	$= T(N) \cdot \log^{O(1)}(T(N))$	Yes
AP	$= O(T(N)^{\varepsilon})$	$= T(N) \cdot \log^{O(1)}(T(N))$	Yes
SNC	$> \log^{O(1)}(T(N))$	$= T(N)^{O(1)}$	No
SP	$O(T(N)^{\varepsilon})$	$= T(N)^{O(1)}$	Yes

Table 5.2: Closure under the restartable fail-stop transformation ρ (for $P = P(N)$).

The closure properties of the classes of [63] under the restartable fail-stop transformation is summarized in Table 5.2.

Chapter 6

Simplifying Memory Assumptions

IN ALL our algorithms we assume that the shared memory is in a known state (i.e., contains zeros) prior to the very first execution of a *Write-All* algorithm, and that the writes of a logarithmic number of bits are atomic. In the chapter we formally relax these model requirements.

6.1 Solving Write-All Using Contaminated Memory

As we have shown in Chapter 5, the problem of *Write-All* —using P-processors write 1's into all locations of an array of size N, where $P \leq N$— can and has been used as the basic building block for constructing efficient and fault-tolerant parallel algorithms. All previous *Write-All* solutions use $\Omega(P)$ auxiliary shared memory and assume that this memory is cleared or initialized to some known value. When *Write-All* building blocks are used in polylogarithmic parallel time algorithms (e.g., to compute prefix sums or list ranking) auxiliary memory initialization cannot be amortized over the computation. Thus, assuming clear memory is a very strong precondition, and for *Write-All* itself raises a legitimate "chicken-or-egg" objection.

In this section, using a deterministic bootstrapping and balancing argument, we show how to *Write-All* when auxiliary memory is contaminated with arbitrary values. For any dynamic pattern of fail-stop, no-restart errors on a CRCW PRAM with at least one

surviving processor, our new algorithm writes all 1's using $O(N + P\log^3 N/(\log\log^2 N))$ work, *without any initialization assumption.* This technique can be combined with any *Write-All* algorithm to yield efficient simulations of any PRAM and even optimal simulations given processor slack. It can also be used with restartable fail-stop processor simulations. In addition, we show that for the parallel prefix computation it is possible to improve on the best deterministic simulations to date: by a factor of $\log N$ when the memory is clear, and by a factor of $\log\log N$ when the memory is contaminated.

6.1.1 Write-All assumptions

Write-All captures the computational progress that can be naturally accomplished in unit time by a PRAM (when $P = N$). In the presence of asynchrony or failures, efficient solutions to *Write-All* (increasing the fault-free work by polylogarithmic factors only) are non-obvious. Note that, in all existing solutions it does not matter what is the initial state of the size N array. For example, up to now, we assumed it is all 0's, but the algorithms would work even if the N locations were initialized using arbitrary 0's and 1's. A much more important assumption in all previous *Write-All* solutions (both in this thesis, and by other authors, e.g, [29, 59, 61, 75]) was regarding the initial state of additional auxiliary memory used (typically of $\Omega(P)$ size). The basic assumption has been that:

The $\Omega(P)$ auxiliary shared memory is cleared or initialized to some known value.

In theory, this is a natural, even if unstated assumption, for PRAMs [44] and RAMs (cf., Turing Machine auxiliary tapes are initially blank). However, given the definition of *Write-All* this dependence on clear space raises a legitimate "chicken-or-egg" objection. In practice, memory locations typically contain unpredictable values, and processes that need to use large blocks of memory cannot assume that it is cleared or is initialized to a known value. In fact operating systems usually provide explicit services that allocate clear memory, e.g., **calloc()** in standard C libraries. Such allocation is predictably much more time consuming, even in the absence of failures.

It is easy to construct simple *Write-All* algorithms that do not assume clear shared memory, but they appear to use quadratic work. If the overall computation involves many steps, one can perhaps afford an expensive initialization phase and amortize its cost over subsequent efficient steps. Unfortnately, when *Write-All* building blocks are

used in very fast (i.e., polylogarithmic parallel time) algorithms (e.g., to compute prefix sums or list ranking) auxiliary memory initialization cannot be amortized over the computation. Fortunately, we show that there is a way around this dilemma:

> *We present Write-All algorithms and algorithm simulations that do not require that the auxiliary memory is cleared prior to the computation.*

Algorithms in this setting have some similarities with the notion of a *self-stabilizing system* introduced by Dijkstra in [34]. Paraphrasing [34], a system is self-stabilizing if and only if, regardless of the initial state the system can always make a state transition into another state, and the system is guaranteed to find itself in a legitimate state after a finite number of transitions. Our computations using initially contaminated memory can be viewed as self-stabilizing with respect to the state of shared memory.

We eliminate the assumption that any amount of clear initial memory is available for the fail-stop and fail-stop restartable algorithms. We develop *deterministic* fault-tolerant algorithms that can be used to simulate PRAMs using contaminated memory, i.e., when the shared memory not containing the input is initially in an arbitrary and possibly illegal state. We also improve on the state-of-the-art robust prefix sums computations.

6.1.2 Model: fail-stop PRAM with contaminated memory

The basis of our model is the restartable fail-stop CRCW PRAM of Sections 2.5-2.6 except that the shared memory that does not contain the input is *contaminated*:

1. There are P processors. Each has a unique processor identifier PID in the range $\{0, \ldots, P-1\}$.
2. *Shared* memory is accessible to all processors; each processor has a constant size *private* memory. Each memory cell stores one word of size $O(\log \max\{N, P\})$.
3. The input is stored in N cells in shared memory.
4. The shared memory not containing the input is *contaminated*.

We use the notation "*Write-All*(N, P, L)" to stand for an instance of fault-tolerant *Write-All* that uses P processors and *clear* auxiliary memory of size L to initialize to 1 an array of size N.

Definition 6.1 An algorithm that uses P processors to solve a *Write-All* problem of size N is *contamination-tolerant*, if it is a *Write-All*$(N, P, 0)$ algorithm. □

6.1.3 Write-All algorithms using contaminated memory

The *Write-All* algorithms and simulations based on *Write-All* paradigm, e.g., [55, 59, 61, 92], or the algorithms that can serve as *Write-All* solution, e.g., the addition algorithm in [29] or the maximum finding algorithm in [75], invariably assume that a linear portion of shared memory is either cleared or is initialized to known values. Starting with a non-contaminated portion of memory, such algorithms and simulations are able to perform their computation by "using up" the clear memory, and concurrently or subsequently clearing additional segments of memory needed for future iterations. We develop an efficient *Write-All* solution that requires no clear shared memory.

A Bootstrap procedure

We formulate a *bootstrap* approach to the design of fault-tolerant *Write-All* algorithms, such that the auxiliary memory is initially contaminated. The bootstrapping proceeds in stages:

In stage 1 of our procedure, all P processors clear an initial segment of N_0 locations in the auxiliary memory.

At the stage i of the procedure, we use P processors to clear N_{i+1} memory locations with the help of N_i memory locations that were cleared in the stge $i-1$.

If $N_{i+1} > N_i$ and $N_0 \geq 1$, then this procedure will clear the required N memory location in at most N stages. Say τ is the final stage number, i.e., $N_\tau = N$.

Let P_i be the number of active processors that initiate phase i, and define $N_{-1} = 0$. The cost of such a procedure is: $S_{boot} = \sum_{i=1}^{\tau} S_i(N_i, P_i, N_{i-1})$ where S_i is the cost of the *Write-All*(N_i, P_i, N_{i-1}) algorithm used in stage i.

The efficiency of the resulting algorithm depends on the choices of the particular *Write-All* solution(s) used in each stage and the parameters N_i.

One specific approach is to define a series of multipliers $G_0, G_1, \ldots, G_\tau$ such that $N_i = \prod_{j=0}^{i} G_j$. The high level view of such algorithm is given in Figure 6.1. The

```
01    forall processors PID=0..P − 1 parbegin  −−P processors clear N memory
02        Clear the initial block of N_0 = G_0 elements sequentially using P processors
03        i := 0  −−Iteration counter
04        while N_i < N do
05            Use a Write-All solution with data structures of size N_i
06            and G_{i+1} elements at the leaves
07            to clear memory of size N_{i+1} = N_i · G_{i+1}
08            i := i + 1
09        od
10    parend
```

Figure 6.1: A high level view of the bootstrap algorithm.

algorithm consists of an initialization (lines 02-04) and a parallel loop (lines 04-09). We use a variation of this scheme below.

We next use the bootstrap approach to construct and analyze contamination-tolerant *Write-All* algorithms in the fail-stop and restartable fail-stop models.

Algorithm Z for the fail-stop model

We will algorithm W in each phase of the bootstrap procedure, and we call the resulting algorithm, algorithm Z.

We analyze algorithm Z for the following choice of parameters: we use $G_0 = \log N$, and $G_i = G_{i-1} \log N$ (for $i > 0$). In the initialization, all P processors traverse a list of size G_0 sequentially and clear it. Then, iteratively, the processors use algorithm W to clear increasingly larger sections of memory using the auxiliary memory cleared in the previous iteration (Fig. 6.1, lines 05-07).

Recall that algorithm W is a fail-stop (no restart) *Write-All* solution. It uses two full binary trees (represented as heaps in memory) and it consists of a loop in which the active processors synchronously iterate through four tree traversal phases. To avoid a complete restatement, the reader is urged to refer to Section 3.2.1. When we use a parameterized algorithm W, with the result of Martel (Appendix A.6), the work of the algorithm is (similarly to Lemma 3.6):

Theorem 6.1 Algorithm W with P processors, the progress tree with H leaves ($P \leq H$) and $2H-1$ total nodes all initialized to zero and G array elements at each leaf, has the work of $S = O((H + P \log H / \log \log H) \cdot (\log P + \log H + G))$ for any pattern of stop-failures.

Note that the above result and algorithm W can be used when $P > H$. As we have already described in Section 3.2.1, when there are P processors and the progress tree has $H < P$ leaves, then it is sufficient for each processor to take its PID modulo H to assure uniform initial assignment of processors and to preserve the result.

Algorithm W stores its binary trees as linear arrays interpreted as heaps. Therefore the structure of the trees is unaffected by the state of the memory, because the heaps are implicit. We next observe that the enumeration of the processors in phase W1 of algorithm W can be done in a bottom-up traversal of a *contaminated* processor tree. The pseudocode for this algorithm is given in Figure 6.2.

We call this algorithm Z_{enum}. The surviving processors enumerate themselves using a standard logarithmic time algorithm based on addition. The contaminated memory cells are distinguished from the cells that contain valid values via the use of a single bit associated with each cell (a so called "deadman flag"). When a processor arrives at a node, it clears the bit associated with its sibling, then it sets its own bit (lines 16-17). Only cells that have valid values written in them by active processors will have the bit set. The enumeration itself is as in phase W1.

Theorem 6.2 Algorithm Z is a contamination-tolerant *Write-All*$(N, P, 0)$ algorithm that fo any pattern of fail-stop errors has $S = O(N + P \log^3 N/(\log \log N)^2)$ for $1 \leq P \leq N$.

Proof: We first evaluate and then total the work of the algorithm during each of the finite numbers stages of its execution. In each use of algorithm W, we will have $G = \log N$ as the number of memory locations associated with each leaf of the progress tree, and we will apply Thm 6.1 with different instantiations of H to evaluate the upper bound of work.

Stage 0: Enumerate processors using Z_{enum}, then sequentially clear $\log N$ memory using all surviving processors. The work using the initial $P_0 \leq P$ processors is: $W_0 = P_0 \cdot \log P + P_0 \cdot \log N$.

```
01    forall processors PID = 0..P − 1 parbegin
02        shared integer array c[1..2N − 1]; −−processor counts
03        shared bit array alive[1..2N − 1]; −−alive/dead markers
04        private integer pn −−enumerated processor number
05        private integer j1, j2, −−left/right siblings indices
06            t; −−predecessor index of j1 and j2
07        j1 := PID + (N − 1); −−heap-leaf init
08        pn := 1; −−assume this processor is no. 1
09        c[j1] := 1; −−a processor is counted once in this step
10        for 1.. log(P) do −−traverse the tree from leaf to root
11            t := j1 div 2; −−parent of j1 and j2
12            if 2 ∗ t = j1
13            then j2 := j1 + 1 −−j1 came from left
14            else j2 := j1 − 1 −−j1 came from right
15            fi ;
16            alive[j2] := 0 −−mark siblings dead
17            alive[j1] := 1 −−mark self alive
18            if alive[j2] = 1 −−both sub-trees have active processors?
19            then c[t] := c[j1] + c[j2] −−both branches are active
20                if j1 > j2 −−j1 came from right, update processor n umber
21                then pn := pn + c[j2]
22                fi
23            else c[t] := c[j1] −−all siblings failed
24            fi ;
25            j1 := t −−advance up the heap
26        od
27    parend
```

Figure 6.2: Contamination robust processor enumeration Z_{enum}.

Stage 1: $P_1 \leq P_0 \leq P$. Using instance of Thm 6.1 where $H = \log N$, the work is:

$$W_1 = (\log N + P_1 \log\log N / \log\log\log N) \cdot (\log P_1 + \log N + \log\log N).$$

Stage i: $P_i \leq P_{i-1} \leq N$. Using instance where $H = \log^i N$:

$$W_i = (\log^i N + P_i \cdot i \log\log N / (\log i + \log\log\log N)) \cdot (\log P_i + \log N + i \log\log N)$$

The *Final Stage* τ is when $\log^\tau N = N/\log N$, i.e., $\tau = \frac{\log N}{\log\log N} - 1$.

Totalling the work in all phases yields:

$$S = \sum_{i=0}^{\tau} W_i = W_0 + \sum_{i=1}^{\tau} \left(\log^i N + P_i \frac{i \log\log N}{\log i + \log\log\log N} \right) (\log P_i + \log N + i \log\log N)$$

Simplifying the sum results in $S = O(N + P \log^3 N/(\log\log N)^2)$. □

This approach has the following range of optimality:

Theorem 6.3 For any pattern of fail-stop errors, algorithm Z is a contamination-tolerant *Write-All*$(N, N(\log\log N)^2/\log^3 N, 0)$ algorithm with $S = O(N)$.

Algorithm Z_r for the restartable fail-stop model

Algorithm Z_r is similar to algorithm Z, except that in each stage we will be utilizing a restartable *Write-All* algorithm. (Algorithm W is not suitable when restarts are allowed.) Other parameters of the bootstrap procedure are the same as for the fail-stop case.

In this analysis, we will be using an algorithm that was described and characterized with the following result by Anderson and Woll:

Theorem 6.4 [8] There exists a *Write-All*(H, H, H) solution with H processors that has work $O(H^{1+\varepsilon})$ for every $\varepsilon > 0$.

This is an existential result, and we call this algorithm AW. The best known constructed deterministic algorithm has $\varepsilon = \log_2 3 - 1 < 0.59$ is algorithm X (it can also be used with the bootstrap). Note that algorithm AW was developed for the asynchronous model, but it can be used in the restartable fail-stop model as well. The work of the algorithm in the asynchronous model is the same as its completed work in the restartable fail-stop model.

Theorem 6.5 Algorithm Z_r is a contamination-tolerant *Write-All*$(N, N, 0)$ algorithm that for any pattern of fail-stop errors has $S = O(N^{1+\varepsilon})$ for any $\varepsilon > 0$.

Proof: We first note that there exists a *Write-All*(H, P, H) solution with $P \geq H$ processors that has work $O(P^{1+\varepsilon})$ for every $\varepsilon > 0$. We use algorithm AW, except all processors use their PIDs modulo H. The worst case work is achieved when up to $\lceil \frac{P}{H} \rceil$ processors that have the same PID module H operate synchronously as a single processor. The work of the algorithm in this case is: $S = \lceil \frac{P}{H} \rceil \cdot O(H^{1+\varepsilon}) = O(P^{1+\varepsilon})$. Using this algorithm at each stage of the bootstrap procedure, and evaluating the total work as in Thm 6.2 yields the desired result:

We evaluate and then sum the work of the algorithm during each of the finite numbers stages of its execution. In each stage $i > 1$ of algorithm Z_r, we will use

algorithm AW $\log N$ times to clear $\log^{i+1} N$ memory locations. In each instance of use of Theorem 6.4, we will use $\delta > 0$ as the exponent, such that $\varepsilon/2 = \delta$. This is done to simplify the final sum using the property that $\log N = O(N^{\delta})$ for any $\delta > 0$ We also use $P = N$ for clarity.

Stage 0: All processors linearly initialize the segment of shared memory of length $\log N$ using The work is: $W_0 = P \cdot \log N$.

Stage 1: The algorithm is applied $\log N$ times to clear a segment of shared memory of size $\log^2 N$. Using instance where $H = \log N$, the work is: $W_1 = (P \log^{\delta} N) \cdot \log N$.

Stage i: Using instance $H = \log^i N$: $W_i = (P(\log^i N)^{\delta} N) \cdot \log N = (P \log^{i\delta} N) \cdot \log N$.

Final Stage τ where $\log^{\tau} N = N/\log N$, i.e., $\tau = \log N/\log\log N - 1$. Using the instance where $H = \log^{\tau} N = N/\log N$, the work is: $W_{\tau} = (P(\log^{\tau} N)^{\delta}) \cdot \log N = (P(N/\log N)^{\delta}) \cdot \log N = P \cdot N^{\delta} \log^{1-\delta} N$.

$$S = \sum_{i=0}^{\tau} W_i = W_0 + \sum_{i=1}^{\tau} (P \log^{i\delta} N) \cdot \log N = O(N^{1+\delta} \log^{1-\delta} N)$$
$$= O(N^{1+\delta} \log N) = O(N^{1+\varepsilon}).$$

□

6.1.4 General simulations and algorithm transformations

Using the contamination-tolerant *Write-All* solutions we have developed for the fail-stop no-restart and fail-stop restartable models, we obtain the general simulation results and some improvements for algorithm transformations.

Oblivious simulations

For the setting with initially contaminated shared memory, using algorithms Z and Z_r with the general algorithm simulation techniques from Chapter 5, we obtain the following results:

Theorem 6.6 Any N-processor, τ parallel time PRAM algorithm can be simulated using $O(N)$ contaminated memory and P fail-stop CRCW processors with

$S = O(N + P \log^3 N/(\log\log N)^2 + \tau \cdot P \log^2 N/\log\log N)$ for $1 \leq P \leq N$.

This simulation has optimal ranges:

Corollary 6.7 Any N-processor, τ parallel time PRAM algorithm can be simulated using $O(N)$ contaminated memory and P fail-stop CRCW processors with $S = O(\tau \cdot N)$ when:

1. $1 \leq P \leq N(\log\log N)^2/\log^3 N)$, or
2. $1 \leq P \leq N\log\log N/\log^2 N)$ and $\tau > \log N/\log\log N$.

In the restartable fail-stop model we get:

Theorem 6.8 Any N-processor, τ parallel time PRAM algorithm can be simulated using $O(N)$ contaminated memory and N restartable fail-stop CRCW processors with $S = O(\tau \cdot N^{1+\varepsilon})$.

Remark 6.1 We can also use the complexity measure of *overhead ratio* σ to evaluate the efficiency of simulations by that amortizing the work of a simulation over the necessary work and the number of failures that are encountered. The simulation in the restartable fail-stop model has overhead ratio per PRAM step of $\sigma = N^{\varepsilon}$. This overhead ratio can be made polylogarithmic by interleaving algorithm Z_r with algorithm V as presented in Section 5.4.

Improving oblivious simulations

As we have discussed in Chapter 5, custom transformations of algorithms are interesting because in some cases it is possible to improve on the work of the naïve oblivious simulation. These improvements are most significant for fast algorithms when a full range of processors is used. In the case of parallel prefix (Section 5.5.1), additional savings can be carried over to the fail-stop no-restart model in the setting when the shared memory is contaminated. Using the result of Theorem 5.9 together with a contamination-tolerant *Write-All* solution we obtain the following:

Theorem 6.9 Parallel prefix for N values can be computed using N fail-stop processors and $O(N)$ contaminated memory with $S = O(N\log^3 N/(\log\log N)^2)$.

Note that using N processors to simulate a parallel prefix algorithm that uses $P = N$ processors and time $\log N$ would require the work $S = O(N \log^3 N / \log\log N)$ (Theorem 6.6), and so the custom algorithm saves a $\log\log N$ factor relative to the oblivious simulation.

6.2 Atomic Access and Word Size

Thus far, we relied on the property of our model to perform $\log N$-bit word parallel writes atomically. That is, the model allows the following: (1) $\log N$-bit words are written in unit time, and (2) the adversary can cause failures either before or after the write cycle of the PRAM, but not during the write cycle. The fault-tolerant algorithms we developed can be modified so that these two restrictions are relaxed.

The new definition of atomicity becomes:

(1) $\log N$-size words are written using $\log N$ bit write cycles, and

(2) the adversary can cause arbitrary fail-stop errors either before or after the *single bit write cycle* of the PRAM, but not during the bit write cycle.

Proposition 6.1 Any fault-tolerant algorithm using $O(\log N)$ bit atomic writes on inputs I of size N, and using P processors for $1 \leq P \leq N$ can be adapted to use $O(1)$ bit atomic writes so that there is: (1) preservation of $O(S(I, F, P))$ steps used by the algorithm (counting $\log N$ bit write cycles as one time unit), and (2) preservation of the space used (counting $O(\log N)$ bits as one word).

Proof: The algorithms are adapted by *simulating* one atomic write of an $O(\log N)$ bit word atomic writes using $O(\log N)$ atomic $O(1)$ bit writes. We implement $\log N$-size words using a single bit *tag* and two $\log N$-size words. The two words are numbered 0 and 1, and the bit tag (initially 0) indicates which of the two words has valid contents.

Thus each shared memory location is represented as:

```
record
    bit integer t;  --current valid version number
    integer X[0..1];  --log N-size values indexed by t
end
```

Each read cycle of the shared memory now becomes:

```
begin --macro read cycle
    read tag from t; --read current tag
    for i = 1 to log(N) do --read current contents
        read bit i of value from bit i of X[tag];
    od
end
```

The write cycle to the shared memory becomes:

```
begin --macro write cycle
    read tag from t; --read current tag
    tag := tag + 1 (mod 2);
    for i = 1 to log(N) do --write new contents
        write bit i of value to bit i of X[tag];
    od
    write tag to t; --update the tag
end
```

Since the single bit tag is the last bit written during the write cycle, a failure anywhere during this high level write cycle will prevent the tag value to be updated, and so any subsequent read will be able to read the previous value stored. This approach is similar to that of Bloom in [24], but it is somewhat simpler due to the fact that we are dealing with the *synchronous* model.

Fault-tolerant algorithms can be automatically transformed using the macro read and write cycles above to versions that only require single bit atomic writes. Clearly, the number of $\log N$-size words read or written by each macro cycle is $O(1)$ as before, and the shared memory requirements are within a factor of two of the original memory size. Therefore, the asymptotic performance of the algorithm has not changed. □

Remark 6.2 It is sufficient to use non-atomic $\log N$-size word reads/writes instead of the $\log N$ single bit reads/writes. Thus the simplest atomicity requirement is that the write of the single bit tag must be atomic per each write of a single $\log N$-bit word.

Remark 6.3 This approach is consistent with the restartable fail-stop PRAM, where since synchronous restarts cannot occur in the middle of a read or a write of a word. In that setting, the above simulation can be used as is with restartable processors.

Chapter 7

Discussion and Open Problems

WE presented a study fault tolerance and efficiency for two models of fault-prone parallel computation: fail-stop no-restart PRAM and restartable fail-stop PRAM. Both models are direct extensions of the standard PRAM model, so that all existing algorithms can be executed on either model without any changes in the absence of failures. When failures are introduced or when the shared memory is initially contaminated, existing parallel algorithms can be mechanically transformed so that the algorithms become fault-tolerant while the efficiency is degraded slightly for a large family of failure patterns. We furthermore can take advantage of parallel slack. In the fail-stop no-restart model, we can simulate algorithms so that the work is preserved asymptotically. In the restartable model the work is preserved for as long as the number of failures per each processor is logarithmic per each simulated step.

The area of efficient and fault-tolerant parallel computation remains a fertile ground for further research and improvements of the existing results:

Model building: The definitions of the two models of computation that we studied, the fail-stop PRAM and the restartable fail-stop PRAM, are obtained by combining: (i) a model of parallel processing (i.e., shared-memory based multiprocessing), and (ii) an associated model of failures (e.g., failure types, strength of adversaries, granularity of failures and frequency). In order to evaluate the efficiency of various fault-tolerant algorithmic methods for the defined models, we have defined new and generalized existing measures of complexity (i.e., overhead ratio available

processor steps and overall work). We have also showed that the fail-stop CREW PRAM does not admit efficient solutions to the *Write-All* problem.

Productive and promising research areas include identifying further natural models with the goal of classifying the models that: (a) either admit efficient fault-tolerant algorithms, or (b) that are inherently prohibitive to efficient computation.

For the update cycles that we use in the restartable model, it is interesting to determine the minimum number of reads and writes necessary to enable the existence of efficient algorithms. Other questions of merit include: 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 can be derived independently and do not come from simulation or transformation of synchronous PRAM algorithms?

Algorithms and upper bounds: We designed and analized several efficient and fault-tolerant algorithms for the parallel models studied.

The design of even more efficient algorithms subject to the constraints of efficiency, reliability, scalability and feasibility remains a challenging topic for research. More efficient algorithms could be developed for the *Write-All* problem that serves as the basis for general algorithm simulations, and also in order to improve the efficiency of naïve general simulations. There is still a $\log N / \log\log N$ gap that remains between the most efficient known deterministic *Write-All* algorithm, i.e., W, and the corresponding lower bound, i.e., $O(N \log N)$ ([61]).

Another open problem is to determine the overhead ratio σ for algorithm X in the original setting of failures and restarts.

Recently, an existence proof for an algorithm achieving $O(N^{1+\epsilon})$ work was given in [8]. Is $O(N \log^{O(1)} N)$ completed work for solving *Write-All* with N processors and input of size N achievable in the restartable fail-stop model?

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

Under the same assumption we showed an $\Omega(N \log N / \log\log N)$ lower bound for the no-restart fail-stop model, and that this is the best possible bound.

In order to further improve these bounds, the strong assumption of memory snapshots must be removed. There was some progress in this direction. Under different assumptions, an $\Omega(N \log N)$ lower bound is shown for failures without restarts in [61].

Can these lower bounds be further improved? For the no-restart fail-stop model, the improvements can be modest at best, since only a $\log N / \log\log N$ gap remains between the upper and lower bounds.

Experimental analysis: The design of new and the analysis of existing fault-tolerant parallel algorithms can be aided by using experimentation. Algorithm animation [25, 95] has the promise of providing additional insights into algortihms' behavior through visualization. A tool for animating *Write-All* algorithms was developed by Apgar [9] using Stasko's TANGO system [95]. Using the that animation, an observer can monitor the progress of a parallel computation and dynamically inject processor faults and restarts.

Concluding remarks

It is often claimed that distributed computing systems have the potential advantage of higher reliability over centralized systems. This advantage of distributed computing can be applied also to parallel systems, because the fault-tolerance in distributed systems is precisely due to the replication of resources. The resulting redundancy in computation is a trade-off of efficiency (measured as all available resources) for fault-tolerance.

The fundamental question we studied in the context of parallel algorithms depend on the exact form of this trade-off, in summary:

> *How can the reliability advantage of distributed computing be combined with the speed-up potential of parallel computing?*

Many interesting related work areas can be defined, and we believe that there are worthy research areas beyond the immediate results of this thesis. Our framework may be generalized in many ways based on assumptions about system architecture and structure of faults. Some concrete and relevant questions that seem promising follow:

1. Formulate a common framework for merging our results on fault-tolerant parallelism and the existing results on distributed network protocols in the presence of topological changes.

2. What about randomized robust parallel computation? Could a formalization along the lines of [70] be used? Another good area here seems to be the average case analysis of fail-stop CREW processors [73].

3. Evaluate the feasibility of implementing fault-tolerant algorithms based on different multiprocessing paradigms.

4. In actual multiprocessor practice, *threads* packages provide a basis for the implementation of a wide variety of parallel paradigms, e.g., [22, 36]. The available *threads* packages typically support shared-memory lightweight processes. How are parallel programs, implemented using *threads* packages, affected by processor failures? How can the fault-tolerance that can be built into the *threads* packages themselves. What fault-tolerant programming methodologies, can be designed for the commonly used *threads* packages.

5. Fault-tolerant multiprocessor scheduling – develop the models and strategies for efficient and fault-tolerant processor scheduling (e.g., the three processor allocation paradigms) and concurrency control for the purposes of robust computation and preservation of invariants of persistent data structures.

6. Robust multiprocessing software package – a practical goal could be to develop a multiprocessing *threads* package based on the accumulated research and experimental results.

Bibliography

[1] J.A. Abraham, P. Banerjee, C.-Y. Chen, W. K. Fuchs, S.-Y. Kuo, A.L. Narasimha Reddy, "Fault tolerance techniques for systolic arrays", *IEEE Computer*, Vol.20, No.7, pp. 65-76, 1987.

[2] 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.

[3] Y. Afek, B. Awerbuch, E. Gafni, "Applying static network protocols to dynamic networks", in *Proc. of the 28th IEEE Symposium on Foundations of Computer Science*, pp. 358-370, 1987.

[4] 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.

[5] A. Aggarwal, A.K. Chandra, M. Snir, On communication latency in PRAM computations, in *Proc. of the 30 IEEE FOCS*, pp. 11-22, 1989.

[6] G. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummins, 1989.

[7] 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.

[8] 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.

[9] S.W. Apgar, "Interactive Animation of Fault-Tolerant Parallel Algorithms," forthcoming Master's thesis, 1992.

[10] 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.

[11] 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.

[12] Y. Aumann and Michael Ben-Or, "Asymptotically Optimal PRAM Emulation on Faulty Hypercubes," in *Proc. of the 31st IEEE Symposium on Foundations of Computer Science*, pp. 440-446, 1991.

[13] B. Awerbuch, "On the effects of feedback in dynamic network protocols", *in Proc. of the 29th IEEE Symposium on Foundations of Computer Science*, pp. 231-242, 1988.

[14] B. Awerbuch, B. Patt, G. Varghese, "Self-stabilization by Local Checking and Correction," in *Proc. of the 33rd IEEE Symposium on Foundations of Computer Science*, pp. 258-267, 1991.

[15] B. Awerbuch, M. Sipser, "Dynamic networks are as fast as static networks", *in Proc. of the 29th IEEE Symposium on Foundations of Computer Science*, pp. 206-219, 1988.

[16] B. Awerbuch, G. Varghese, "Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols," in *Proc. of the 33rd IEEE Symposium on Foundations of Computer Science*, pp. 258-267, 1991.

[17] J. Bartlett et al., "Fault Tolerance in Tandem Computer Systems," in *The Theory and Practice of Reliable System Design*, by D.P. Siewiorek and R.S. Swarz, Digital Press, 1991.

[18] K.E. Batcher, Sorting networks and their applications, in *Proc. of the AFIPS Spring Joint Comp. Conf.*, vol. 32, pp. 307-314, 1968.

[19] G. Baudet, Asynchronous iterative methods for multiprocessors, *JACM*, vol. 25, no. 2, pp. 226-244, 1978.

[20] P. Beame and J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM," *Journal of the ACM*, vol. 36, no. 3, pp. 643-670, 1989.

[21] G. Bilardi and F. P. Preparata, "Size-Time Complexity of Boolean Networks for Prefix Computation," *Journal of the ACM*, vol. 36, no. 2, pp. 363-382, 1989.

[22] A. Birrell, "An Introduction to Programming with Threads", SRC, Digital Equip. Corp., Technical Report 35, January, 1989.

[23] G. Birkhoff, S. MacLane, *A Survey of Modern Algebra*, 4th ed., Macmillan, 1977.

[24] B. Bloom, "Constructing two-writer atomic registers," *IEEE Trans. on Computers*, vol. 37, no. 12, pp. 1506-1514, 1988.

[25] M.H. Brown, "Exploring algorithms using Balsa-II", *IEEE Computer*, Vol.21, No.5, pp. 14-36, 1988.

[26] W. Bruckert, C. Alonso, J. Melvin, "Verification of the First Fault-tolerant VAX System," *Digital Technical Journal*, vol. 3, no. 1, pp. 79-85, 1991.

[27] J. Buss, P.C. Kanellakis, P. Ragde, A.A. Shvartsman, "Parallel algorithms with processor failures and delays", Brown Univ. Tech. Report CS-91-54, August 1991.

[28] M. Chean and J.A.B. Fortes, "A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays," *IEEE Computer*, vol. 23, no. 1, pp. 55-69, 1990.

[29] 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.

[30] S.A. Cook, "An Overview of Computational Complexity," in *Comm. of the ACM*, vol. 9, pp. 400-408, 1983.

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

[32] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, The MIT Press, 1989.

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

[34] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. of the ACM*, Vol. 17, No. 11, pp. 643-644, 1976.

[35] E. W. Dijkstra, *The Discipline of Programming*, Prentice-Hall, 1976.

[36] T. Doeppner, "Threads: A system for the Support of Concurrent Programming," Computer Science Technical Report CS-87-11, Brown University, 1987.

[37] D. Dolev, C. Dwork, L. Stockmeyer, "On the minimal synchronism needed for distributed consensus", *in Proc. of the 24th IEEE FOCS*, pp. 393–402, 1983.

[38] 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.

[39] P. van Emde Boas, "Machine Models and Simulations," in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), vol. 1, North-Holland, 1990.

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

[41] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)", *Yale Univ. Tech. Rep.*, DCS/RR-273, 1983.

[42] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency", *IPL*, vol. 14., no. 4, pp. 183-186, 1982.

[43] M. J. Fischer, N. A. Lynch, M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *JACM*, vol. 32, no. 2, pp. 374-382, 1985.

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

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

[46] R. Graham, D. Knuth, O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, 1989.

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

[48] M. P. Herlihy, "Impossibility Results for Asynchronous PRAM", in *Proc. of the Third ACM Symposium on Parallel Algorithms and Architectures*, pp. 327-336, 1991.

[49] 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.

[50] R. P. Hughey, *Programmable Systolic Array*, Ph.D. Dissertation, Brown University, CS-TR-34, May, 1991.

[51] *IEEE Computer*, "Fault-Tolerant Computing", special issue, vol. 17, no. 8, 1984.

[52] *IEEE Computer*, "Interconnection Networks", special issue, vol. 20, no. 6, 1987.

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

[54] 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.

[55] P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms Can Be Made Robust", *Distributed Computing*), vol. 5, no. 4, 1992; preliminary version appears in *Proc. of the 8th ACM Symposium on Principles of Distributed Computing*, pp. 211-222, 1989.

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

[57] P. C. Kanellakis and A. A. Shvartsman, "Robust Computing with Fail-Stop Processors", in *Proceedings of the Second Annual Review and Workshop on Ultradependable Multicomputers*, Office of Naval Research, pp. 55-60, 1991.

[58] 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), vol. 1, North-Holland, 1990.

[59] 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.

[60] Z. M. Kedem, K. V. Palem, M. O. Rabin and A. Raghunathan, "Efficient Program Transformations for Resilient Parallel Computation via Randomization," to appear in *Proc 24th ACM Symposium on Theory of Computing*, 1992.

[61] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis, "Combining Tentative and Definite Executions for Dependable Parallel Computing," in *Proc 23d ACM Symposium on Theory of Computing*, pp. 381-390, 1991.

[62] 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.

[63] C. P. Kruskal, L. Rudolph, M. Snir, "A Complexity Theory of Efficient Parallel Algorithms," *Theoretical Computer Science* **71**, pp. 95-132, 1990.

[64] L. E. Ladner, M. J. Fischer, "Parallel Prefix Computation", *Journal of the ACM*, vol. 27, no. 4, pp. 831-838, 1980.

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

[66] L. Lamport and N. Lynch, "Distributed Computing: Models and Methods," in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), vol. 2, North-Holland, 1990.

[67] M. Li and Y. Yesha, "New Lower Bounds for Parallel Computation," *Journal of the ACM*, vol. 36, no. 3, pp. 671-680, 1989.

[68] A. López-Ortiz, "Algorithm X takes work $\Omega(n \log^2 n / \log\log n)$ in a synchronous fail-stop (no restart) PRAM", unpublished manuscript, 1992.

[69] N.A. Lynch, "One Hundred Impossibility Proofs for Distributed Comuting", *Proc. of the 8th ACM Symposium on Principles of Distributed Computing*, pp. 1-27, 1989.

[70] 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.

[71] C. Martel, personal communication, March, 1991.

[72] C. Martel, A. Park, and R. Subramonian, "Work-optimal Asynchronous Algorithms for Shared Memory Parallel Computers," to appear in *SIAM Journal on Computing* in 1992.

[73] C. Martel and R. Subramonian, "On the Complexity of Certified Write-All Algorithms," Tech. Report CSE-91-24, Univ. of Calif.-Davis, 1991.

[74] C. Martel and R. Subramonian, "Asynchronous PRAM Algorithms for List Ranking and Transitive Closure," Tech. Report CSE-90-12, Univ. of Calif.-Davis, revised 1991.

[75] 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.

[76] R. McEliece, *The Theory of Information and Coding*, Addison-Wesley, 1977.

[77] S. Mullender (Editor), *Distributed Systems*, ACM Press, Frontier Series, 1989.

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

[79] M. Pease, R. Shostak, L. Lamport, "Reaching agreement in the presence of faults", *JACM*, vol. 27, no. 2, pp. 228-234, 1980.

[80] M. Pease, R. Shostak, L. Lamport, "The Byzantine Generals Problem", *ACM TOPLAS*, vol. 4, no. 3, pp. 382-401, 1982.

[81] N. Pippenger, "On Simultaneous Resource Bounds", in *Proc. of 20th IEEE Symposium on Foundations of Computer Science*, pp. 307-311, 1979.

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

[83] N. Pippenger, "Communications Networks," in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), vol. 1, North-Holland, 1990.

[84] F.P. Preparata, "Holographic Dispersal and Recovery of Information," in *IEEE Trans. on Info. Theory*, vol. 35, no. 5, pp. 1123-1124, 1989.

[85] F.P. Preparata, "Parallel Optical Interconnections and Discrete Holography," a presentation at Brown. Univ. Industr. Partners Symp., March, 1991.

[86] M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance", *J. of ACM*, vol. 36, no. 2, pp. 335-348, 1989.

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

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

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

[90] 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.

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

[92] A. A. Shvartsman, "Achieving Optimal CRCW PRAM Fault-Tolerance", in *Information Processing Letters*, vol. 39, pp. 59-66, 1991.

[93] A. A. Shvartsman, "A Group-Theoretic Aspect of a Multi-Processor Scheduling Problem", unpublished manuscript, (presented in the open session of the *16th Internat. Symp. on Mathematical Foundations of Computer Science*, Poland, 1991).

[94] A. A. Shvartsman, "How to Write-All Efficiently Even with Contaminated Memory", Tech. Report CS-92-09, Brown Univ., 1992.

[95] J.T. Stasko, "Tango: A framework and system for algorithms animation", *IEEE Computer*, Vol.23, No.9, pp. 27-39, 1990.

[96] R.E. Tarjan, U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time", in *Proc. of the 25th IEEE FOCS*, pp. 12-22, 1984.

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

[98] L. Valiant, "General Purpose Parallel Architectures," in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), vol. 1, North-Holland, 1990.

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

[100] J. S. Vitter, R. A. Simons, "New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems for $\mathcal{P}$," *IEEE Trans. Comput.*, vol. 35, no. 5, 1986.

[101] S. Webber, "The Stratus Architecture," in *The Theory and Practice of Reliable System Design*, by D.P. Siewiorek and R.S. Swarz, Digital Press, 1991.

[102] J. C. Wyllie, *The Complexity of Parallel Computation*, Ph.D. Thesis, Cornell University, TR 79-387, 1979.

Appendix A

Pseudocode for algorithm W and Two Lemmas

THE first four sections of this chapter contain the detailed pseudocode for algorithm W and brief comments. Fifth section contains a formal proof of Lemma 3.2. The final sixth section gives an alternative proof of Lemma 3.4 that was communicated by Martel [71].

A.1 Main Procedure for Algorithm W

The main *loop* in Figure A.1 consists of the four phases outlined in Section 3.2.1. Processor counting and enumeration is implemented as a static bottom up traversal in procedure S_BU() in Appendix A.2, work assignment is done in a dynamic top down traversal in procedure D_TD() in Appendix A.4, the work itself is a simple assignment "x[k]:=1", and the progress is measured via a dynamic bottom up traversal in procedure D_BU() given in Appendix A.3. Parameter passing is by reference in all cases.

```
forall processors PID=1..N
parbegin
    shared integer array
      x[1..N], --input array
      c[1..2N-1], --processor counts
      cs[1..2N-1], --count step numbers
      d[1..2N-1], --progress/done tree
      a[1..2N-1]; --accounted tree
    private integer
      pn, --dynamic processor no.
      k, --array index PID will be assigned to
      step; --time stamp

    step := 0; --initial processor counting step
    k := PID; --initially work data item PID
    x[k] := 1; --visit leaf
    D_BU(k); --measure progress

  --Main loop
    while d[1] ≠ N do

        S_BU(PID,step,pn); --enumerate proc-s
        D_TD(pn,k); --assign work
        x[k] := 1; --do work: visit leaf
        D_BU(k); --measure progress

    od
parend ;
```

Figure A.1: Main procedure of algorithm W.

A.2 Static Bottom Up Traversal

This procedure is given in Figure A.2. All processors traverse heap c to compute the overestimate of the number of processors in c[1], and each processor computes its processor number pn that is used in the work assignment phase. The heap cs is used to synchronize processor counting across multiple calls to S_BU().

```
procedure S_BU(integer PID,  --processor id
               integer step,  --timestamp
               integer pn)  --processor no.
    shared integer array
        c[1..2N-1],  --processor counts
        cs[1..2N-1];  --count step numbers
    private integer
        j1, j2,  --siblings indices
        t;  --parent of j1 and j2

    step := step + 1;  --new time stamp
    j1 := PID + (N-1);  --heap-leaf init
    pn := 1;  --assume this processor is no. 1
    c[j1] := 1;
    cs[j1] := step;  --count the processor once

  --Traverse the tree from leaf to root
    for 1..log(N) do

        t := j1 div 2;  --parent of j1 and j2

        if 2*t = j1
        then j2 := j1 + 1  --j1 came from left
        else j2 := j1 - 1  --j1 came from right
        fi ;

        if cs[j1] = cs[j2]  --both sub-trees active?
        then c[t] := c[j1] + c[j2]  --both active
            if j1 > j2  --j1 came from right
            then pn := pn + c[j2]
            fi
        else c[t] := c[j1]  --all siblings failed
        fi ;
        cs[t] := step;  --time stamp, and
        j1 := t  --advance up the heap
    od
end ;
```

Figure A.2: Phase W1 procedure — Static bottom up traversal

A.3 Dynamic Bottom Up Traversal

This procedure is given in Figure A.3. Heap d contains the underestimates for the number of leaves visited in each subtree, with d[1] containing the underestimate of the total number of leaves visited. This number is used in terminating the overall program (when d[1]=N).

```
procedure D_BU(integer k  --current leaf
                          )
    shared integer array
        d[1..2N-1]; --done/progress tree
    private integer
        i1, i2, --siblings indices
        t; --parent of i1 and i2

    i1 := k + (N-1); --heap-leaf init.
    d[i1] := 1; --done for good

  --Traverse the tree from leaf to root
    for 1..log(N) do

        t := i1 div 2; --parent of i1 and i2

      --compute left/right indices
        if 2*t = i1
        then i2 := i1 + 1  --j1 came from left
        else i2 := i1 - 1  --j1 came from right
        fi ;

        d[t] := d[i1] + d[i2]; --update progress
        i1 := t  --advance to the predecessor
    od
end ;
```

Figure A.3: Phase W4 procedure — Dynamic bottom up traversal

A.4 Dynamic Top Down Traversal

This procedure, given in Figure A.4, implements load rescheduling of the remaining active processors. Heaps *c* and *d* are traversed top down. Heap *a* is used to traverse paths to the *unaccounted* leaves according to the discipline implemented by this algorithm. Heap *c* is used to partition the remaining processors between the left and right tree branches, and heap d contains the progress information for the subtrees being traversed. Processors are allocated in proportion to the remaining work.

```
procedure D_TD(integer pn  --dynamic processor no.
               integer k)  --data item
    shared integer array
        c[1..2N-1],  --processor counts
        d[1..2N-1],  --progress/done tree
        a[1..2N-1];  --accounted tree
    private integer
        j, j1, j2;  --current/left/right indices
    j := 1;  --start at the root
    size := N;  --the whole tree is visible
    a[1] := d[1];  --no. of all accounted nodes
  --traverse from root to leaf
    while size ≠ 1 do
        j1 := 2*j; j2 := j1 + 1;  --left/right indices
      --compute accounted node values
        if d[j1]+d[j2] = 0 → a[j1] := 0
        [] d[j1]+d[j2] ≠ 0 → a[j1] := a[j]*d[j1] div (d[j1]+d[j2])
        fi
        a[j2] := a[j]−a[j1];
      --processor alloc. to left/right sub-trees
        c[j1] := c[j]*(size/2−a[j1]) div (size−a[j]);
        c[j2] := c[j] − c[j1];
      --go left/right based on proc. no.
        if pn ≤ c[j1] → j := j1
        [] pn > c[j1] → j := j2; pn := pn − c[j1]
        fi ;
        size := size div 2  --half of leaves visible
    od ;
    k := j − (N−1)  --assign processor based on j
parend
```

Figure A.4: Phase W2 procedure — Dynamic top down traversal

A.5 Dynamic Dop Down Traversal Lemma 3.2

The next lemma shows that the algorithm correctly achieves the desired load balancing.

Lemma 3.2 loop-iteration i of algorithm W: (1) processors are only allocated to unaccounted leaves, and (2) no leaf is allocated more than $\lceil R_i/U_i \rceil$ processors.

Proof: The proof outline of the **while** loop of the algorithm in the style of Dijkstra [35] is given in Figure A.5. We make use of the heap definitions, and the property that results from the dynamic bottom up traversal that if $d[j] \leq size$, where $size$ is the number of leaves in the subtree rooted at j, then $d[2j], d[2j+1] \leq size/2$ (there are no more visited leaves than there are leaves).

At the beginning of phase W2, R_i is the value of $c[1]$, and U_i is $N - d[1]$. We define q as $\lceil R_i/U_i \rceil$. The top down traversal is executed synchronously by all surviving processors, with the processors writing identical values when writing concurrently. Therefore a sequential programming calculus for each processor can be used. We prove the following **while** loop invariant:

$$I : (a[j] < size) \wedge (a[j] \leq d[j])$$
$$\wedge((q-1)(size - a[j]) < c[j] \leq q(size - a[j]))$$
$$\wedge(1 \leq pn \leq c[j])$$

This invariant is established by the assignments to j, $size$, and $a[1]$, performed in the state where $d[1] < N$ by the main loop termination condition, $1 \leq pn \leq c[i]$ by the processor enumeration in phase W1, and $(q-1)(N - d[1]) < c[1] \leq q(N - d[1])$ by the properties of ceiling.

We also make use of the following abbreviation:

$$Q(j, s) : (q-1)(s - a[j]) < c[j] \leq q(s - a[j]).$$

It is straightforward to verify that after I is established, it is left invariant by the **while** loop. Some inference detail is ommited in the proof outline (Figure A.5) for readability. When a processor completes dynamic top-down traversal, the invariant I holds, and $size = 1$. The last assertion of the proof outline implies the following result:

$$(a[j] = 0) \wedge (q\text{–}1 < c[j] \leq q) \wedge (1 \leq pn \leq c[j])$$

```
{ d[1] < N  --by the main loop termination condition
  ∧1 ≤ pn ≤ c[j]  --by the processor enumeration of phase 1
  ∧(q − 1)(N − d[1]) < c[1] ≤ q(N − d[1])  --by the properties of ceiling  }
j,size,a[1] := 1,N,d[1];
{ I }  --invariant is initially established
while size ≠ 1 do
    j1 := 2*j; j2 := j1 + 1;
    if d[j1]+d[j2] = 0 → a[j1] := 0
    [] d[j1]+d[j2] ≠ 0 → a[j1] := a[j]*d[j1] div (d[j1]+d[j2])
    fi
    { I ∧ (a[j1] ≤ d[j1] ∧ a[j1] ≤ size/2) }
    a[j2] := a[j] − a[j1];
    { I1 : I ∧ (a[j1] ≤ d[j1] ∧ a[j1] ≤ size/2)
      ∧(a[j2] ≤ d[j2] ∧ a[j2] ≤ size/2) ∧ (a[j] = a[j1] + a[j2]) }
    c[j1] := c[j]*(size/2−a[j1]) div (size−a[j]);
    { I2 : I1 ∧ Q(j1, size/2) }
    c[j2] := c[j] − c[j1];
    { I3 : I1 ∧ Q(j1, size/2) ∧ Q(j2, size/2) ∧ c[j] = c[j1] + c[j2] }
    if pn ≤ c[j1] → { I3 ∧ pn ≤ c[j1] }{ I3 ∧ a[j1] ≠ size/2 } j := j1
    [] pn > c[j1] → { I3 ∧ pn > c[j1] }{ I3 ∧ a[j2] ≠ size/2 } j := j2; pn := pn − c[j1]
    fi ;
    { I^size_size/2 }  --this is I with size/2 replacing size
    size := size div 2;
    { I }  --invariant is preserved by each iteration
od ;
{ I ∧ size = 1 } { a[j] < 1 ∧ Q(j, 1) ∧ 1 ≤ pn ≤ c[j] }
```

Figure A.5: Proof outline of the phase W2 top down traversal

The safety property that each processor correctly constructs branches of the accounted tree using heap $a[1..2N-1]$ satisfying the constraints given in Section 3.2.1 follows directly from the proof outline by $a[1] = d[1]$ and for an interior node j, $a[j] = a[2j] + a[2j+1]$ with $a[j] \leq d[j]$ for all nodes j (the values of the a heap are computed once and not changed by the top down traversal). The $\Theta(\log N)$ time termination for all surviving processors follows from the initialization and assignments to *size*, and the **while** guard.

Thus, if a processor completes phase W2 then it would reach a leaf with $a[j] = 0$, that is an unaccounted leaf, and at most $q = \lceil R_i/U_i \rceil$ processors would reach the same leaf. □

A.6 Martel's Improved Lemma 3.4

Lemma 3.4bis. For any failure pattern with at least one surviving processor, algorithm W completes all remaining work. Its total number of block-steps V_1 is less than or equal to $U + P \log U / \log \log U$, where P is the initial number of processors and U is the initial number of unvisited elements.

Proof: Consider the ith iteration of the main *loop* of algorithm W, as we did in the analysis of the algorithm in Section 3.2.1.

At the beginning of the iteration, P_i is the overestimate of active processors, and U_i is the estimated remaining unvisited leaves. At the end of the i^{th} iteration (i.e., at the beginning of the $i + 1^{st}$ iteration, the corresponding values are P_{i+1} and U_{i+1}. From the analysis of algorithm W we know that $P_i \geq P_{i+1}$ and $U_i > U_{i+1}$. Let also $P_1 = P$ and $U_1 = U$.

If a processor begins, but does not complete an iteration of the *loop*, we are going to (over)charge the processor $\log U_1$ steps. Such charges, call them B_0, will amount to no more than $B_0 = O(P)$ block-steps (and thus no more than $O(P \log U)$ overall work steps) for a particular execution of the algorithm. Having taken care of this accounting, we are only going to account for the iterations that were completed by the participating processors in the following discussion.

We will treat the three $\log U$ time tree traversals performed by a single processor during each phase of the algorithm as a single *block-step* of cost $O(\log U)$. We will charge each processor for each such completed block step.

Let τ be the final iteration of the algorithm, i.e., $U_\tau \geq 0$ and the number of unvisited elements after the iteration τ is $U_{\tau+1} = 0$. We examine the following two major cases:

1. Consider *all* block steps in which $P_i < U_i$:

   By the balanced processor allocation of algorithm W, each leaf will be assigned no more than 1 processor, therefore the number of block steps B_1 accounted in this case will be no more than $B_1 \leq \sum_{i=1}^{\tau} (U_i - U_{i+1}) = U_1 - U_{t+1} = U - 0 = U$.

2. Now consider *all* block steps in which $P_i \geq U_i$, with the following two subcases :

   (2.a) Consider all block steps after which $U_{i+1} < \frac{U_i}{\log U / \log \log U}$:

This could occur no more than $O(\frac{\log U}{\log\log U})$ times since $U_{i+1} < U_1 = U$. No more than P processors complete such block steps, therefore the total number of blocks $B_{2.a}$ accounted by this sub-case is bounded by: $B_{2.a} = O(P\frac{\log U}{\log\log U})$.

(2.b) Finally consider all block steps such that $P_i \geq U_i$ and $U_{i+1} \geq \frac{U_i}{\log U/\log\log U}$: Consider a particular iteration i. By Lemma 3.2, at most $\lceil\frac{P_i}{U_i}\rceil$ but no less than $\lfloor\frac{P_i}{U_i}\rfloor$ processors were assigned to each of the U_i unvisited leaves. Therefore, the number of failed processors is at least

$$U_{i+1}\lfloor\frac{P_i}{U_i}\rfloor \geq \frac{U_i}{\log U/\log\log U}\cdot\frac{P_i}{2U_i} \geq \frac{P_i}{2\log U/\log\log U}.$$

This can happen no more than τ times. The number of processors completing step i is no more than $P_i(1-\frac{1}{2\frac{\log U}{\log\log U}})$. In general the number of processors completing j^{th} occurrence of case (2.b) will have no more than $P(1-\frac{1}{2\frac{\log U}{\log\log U}})^j$ where P is the initial number of processors.

Therefore the number of blocks $B_{2.b}$ accounted by this sub-case is bounded by:

$$B_{2.b} \leq \sum_{j=1}^{\tau} P(1-\frac{1}{2\frac{\log U}{\log\log U}})^j \leq P\sum_{j=1}^{\infty}(1-\frac{1}{2\frac{\log U}{\log\log U}})^j = P\frac{1}{1-(1-\frac{1}{2\frac{\log U}{\log\log U}})}$$
$$= P\cdot 2\frac{\log U}{\log\log U} = O(P\frac{\log U}{\log\log U}).$$

The total number of block steps B of all cases considered is:

$$B = B_0 + B_1 + B_{2.a} + B_{2.b} = O(U + P\frac{\log U}{\log\log U})\ .\ \square$$

Theorem 3.9 Algorithm W is a robust parallel algorithm for the *Write-All* problem with $S^* = O(N\log^2 N/\log\log N)$, where N is the input array size, and the initial number of processors P is between 1 and N.

Proof: When $1 \leq P \leq U = N$, each block-step is performed in $O(\log N)$ time with one array element at each leaf. Therefore

$$S = B\cdot O(\log N) = O(N\log N + N\frac{\log^2 N}{\log\log N}) = O(N\frac{\log^2 N}{\log\log N})\ .\ \square$$

Appendix B

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 [90] 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 "$\langle\langle PID \rangle\rangle_{\lfloor \log(k) \rfloor}$" 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 $\log N$.

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.

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.

```
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]; −−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 B.1: Algorithm X detailed pseudo-code.

Appendix C

Mathematical lemmas used for lower bounds

Lemma C.1 Given a sorted list of m ($m > 1$) nonnegative integers $a_1, a_2, \ldots, a_m$ then we have for all j ($1 \leq j < m$) that $(1 - \frac{j}{m}) \sum_{i=1}^{m} a_i \leq \sum_{i=j+1}^{m} a_i$.

Proof: We proceed by induction on m. Base case is trivial for $j = 1 < 2 = m$. Using inductive hypothesis $(1 - \frac{j}{k}) \sum_{i=1}^{k} a_i \leq \sum_{i=j+1}^{k} a_i$ we show that $(1 - \frac{j}{k+1}) \sum_{i=1}^{k+1} a_i \leq \sum_{i=j+1}^{k+1} a_i$ by extending the sorted list of k elements by the new element a_{k+1} in the following straightforward transformations:

$a_i \leq a_{k+1}$ (for any i, $1 \leq i \leq k$), and so $\sum_{i=1}^{k+1} a_i \leq (k+1) a_{k+1}$

$\frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq j a_{k+1}$ (for each j, $1 \leq j \leq k$)

$-j a_{k+1} + \frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq 0$

$k a_{k+1} - j a_{k+1} + \frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq k a_{k+1}$ (by adding $k a_{k+1}$ to both sides)

$(k - j) a_{k+1} + \frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq k a_{k+1}$

$(1 - \frac{j}{k}) \sum_{i=1}^{k} a_i + (1 - \frac{j}{k}) a_{k+1} + \frac{j}{k(k+1)} \sum_{i=1}^{k+1} a_i \leq a_{k+1} + \sum_{i=j+1}^{k} a_i$ (by dividing both sides by positive k and using the inductive hypothesis.)

$(1 - \frac{j}{k}) \sum_{i=1}^{k+1} a_i + j(\frac{1}{k} - \frac{1}{k+1}) \sum_{i=1}^{k+1} a_i \leq \sum_{i=j+1}^{k+1} a_i$ (after grouping terms), finally

$(1 - \frac{j}{k+1}) \sum_{i=1}^{k+1} a_i \leq \sum_{i=j+1}^{k+1} a_i$ (after simplifying the inequality). □

Lemma C.2 Given $G > 1$, $N > G$, and integer σ such that $\sigma < \frac{\log N}{\log G} - 1$ then the following inequality holds:

$$\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor/G\rfloor \ldots /G\rfloor}_{\sigma \ times} > 0 \qquad (*)$$

(where σ is the number of times that the expression in the left hand side of the inequality contains division by G).

Proof: To show $(*)$, it suffices to show that, after dropping one floor and strengthening the inequality: $(\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor/G\rfloor \ldots /G\rfloor}_{\sigma-1 \ times} /G) - 1 > 0$, or that $\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor/G\rfloor \ldots /G\rfloor}_{\sigma-1 \ times} > G$.

Applying such transformations for $\sigma - 1$ more steps, we get that it suffices to show: $N > G^{\sigma} + G^{\sigma-1} + \ldots + G$, or $N > \frac{G^{\sigma+1}-G}{G-1}$ using summation for geometric progressions.

We observe that $G^{\sigma+1} > \frac{G^{\sigma+1}-G}{G-1}$ thus it is enough to show that $N > G^{\sigma+1}$. After taking logarithms of both sides we get $\log N > (\sigma+1)\log G$, and so is suffices to have $\sigma < \frac{\log N}{\log G} - 1$. □

To achieve the results in Section 4, this lemma is used with $G = \log N$, i.e., $\sigma < \frac{\log N}{\log\log N} - 1$.

Lemma C.3 For $N \to \infty$: $(1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}} = 1 - \frac{1}{\log\log N} + O(\frac{1}{(\log\log N)^2})$.

Proof: The proof is done using standard techniques for manipulating asymptotics (e.g. Graham et al. [46]).

Let $A = (1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}}$, and $B = \ln(1 - \frac{1}{\log N})$, where ln is the natural logarithm. Using $\ln(1-z) = -z - \frac{z^2}{2} - \frac{z^3}{3} - \ldots$ expansion around 0, we get $B = -\frac{1}{\log N} - \frac{1}{2\log^2 N} - \frac{1}{3\log^3 N} - \ldots$ for $N \to \infty$.

From this we get:

$\ln A = -\frac{1}{\log\log N} - \frac{1}{2\log N\log\log N} - \frac{1}{3\log^2 N\log\log N} - lots$ (*lower order terms*), or that

$A = \exp(-\frac{1}{\log\log N} - \frac{1}{2\log N\log\log N} - \frac{1}{3\log^2 N\log\log N} - lots)$

Using $\exp(z) = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \ldots$ expansion around 0 and after multiplying the resulting series and gathering the terms, we get:

$A = 1 - \frac{1}{\log\log N} + \frac{1}{2(\log\log N)^2} - lots.$ □