Fault-Tolerance and Efficiency in Massively Parallel Algorithms

Chapter · January 1994

DOI: 10.1007/978-0-585-27316-7_5

CITATIONS
5

READS
18

2 authors, including:

Alexander A. Shvartsman
Augusta University
245 PUBLICATIONS **2,866** CITATIONS

SEE PROFILE

SECTION 2.2

Fault-Tolerance and Efficiency in Massively Parallel Algorithms

Paris C. Kanellakis[1] and Alex A. Shvartsman[2]

Abstract

We present an overview of massively parallel deterministic algorithms which combine high fault-tolerance and efficiency. This desirable combination (called *robustness* here) is nontrivial, since increasing efficiency implies removing redundancy whereas increasing fault-tolerance requires adding redundancy to computations. We study a spectrum of algorithmic models for which significant robustness is achievable, from static fault, synchronous computation to dynamic fault, asynchronous computation. In addition to various kinds of fail-stop processors, these models handle arbitrarily initialized memory and restricted memory concurrency. We survey both deterministic upper bounds for the basic *Write-All* primitive, as well as lower bounds on its efficiency, and we identify some of the key open questions. We also generalize the robust computing of functions to relations; this new approach can model approximate computations. We show how to compute approximate *Write-All* optimally. Finally, we synthesize the state-of-the-art in a complexity classification, which extends with fault-tolerance the traditional classification of efficient parallel algorithms.

2.2.1 Introduction

A basic problem of massively parallel computing is that the unreliability of inexpensive processors and their interconnection may eliminate any potential efficiency advantage of parallelism. Our research is an investigation of fault models and parallel computation models under which it is possible to achieve algorithmic efficiency (i.e., speed-ups close to linear in the number of processors) despite the presence of faults. We would like to note that these models can also

[1] *Computer Science Department, Brown University, PO Box 1910, Providence, RI 02912, USA*. Electronic mail: pck@cs.brown.edu. This research was supported by ONR grant N00014-91-J-1613.

[2] *Digital Equipment Corporation, Digital Consulting Technology Office, 30 Porter Road, Littleton, MA 01460, USA*. Electronic mail: alex@hydra.enet.dec.com.

Gary M. Koob and Clifford G. Lau, Eds., Foundations of Dependable Computing, Chapter 2.2, Kluwer Academic Publishers (The Springer International Series in Engineering and Computer Science), September 30, 1994, ISBN 0792394852

be used to explore common properties of a broad spectrum of fault-free models, from synchronous parallel to asynchronous distributed computing. Here, our presentation focuses on deterministic algorithms and complexity, as opposed to algorithms that use randomization.

There is an intuitive trade-off between reliability and efficiency because reliability usually requires *introducing redundancy* in the computation in order to detect errors and reassign resources, whereas gaining efficiency by massively parallel computing requires *removing redundancy* from the computation to fully utilize each processor. Thus, even allowing for some abstraction in the model of parallel computation, it is not obvious that there are any non-trivial fault models that allow near-linear speed-ups. So it was somewhat surprising when in [17] we demonstrated that it is possible to combine efficiency and fault-tolerance for many basic algorithms expressed as concurrent-read concurrent-write parallel random access machines (CRCW PRAMs [14]).

The [17] fault model allows *any pattern of dynamic fail-stop no restart processor errors, as long as one processor remains alive*. The fault model was applied to all CRCW PRAMs in [23, 40]. It was extended in [18] to include *processor restarts*, and in [42] to include *arbitrary static memory faults*, i.e., arbitrary memory initialization, and in [16] to include *restricted memory access* patterns through controlled memory access. Concurrency of reads and writes is an essential feature that accounts for the necessary redundancy so it can be restricted but not eliminated – see [17, 16] for an in-depth discussion of this issue. Also, as shown in [17], it suffices to consider COMMON CRCW PRAMs (all concurrent writes are identical) in which the atomically written words need only contain a constant number of bits.

The work we survey makes three key assumptions. Namely that:

1. Failure-inducing adversaries are worst-case for each model and algorithms for coping with them are deterministic.
2. Processors can read and write memory concurrently – except that initial faults can be handled without memory access concurrency.
3. Processor faults do not affect memory – except that initial memory can be contaminated.

A central algorithmic primitive in our work is the *Write-All* operation of [17]. Iterated *Write-All* is the basis for the algorithm simulation techniques of [23, 40] and for the memory initialization of [42]. Therefore, improved *Write-All* solutions lead to improved simulations and memory clearing techniques.

The *Write-All* problem is: *using P processors write 1s into all locations of an array of size N, where $P \leq N$.* When $P = N$ this operation captures the

computational progress that can be naturally accomplished in one time unit by a PRAM. We say that *Write-All completes at the global clock tick at which all the processors that have not fail-stopped share the knowledge that 1's have been written into all N array locations*. Requiring completion of a *Write-All* algorithm is critical if one wishes to iterate it, as pointed out in [23] which uses a certification bit to separate the various iterations of (Certified) *Write-All*. Note that the *Write-All* completes when all processors halt in all algorithms presented here.

Under dynamic failures, efficient deterministic solutions to *Write-All*, i.e., increasing the fault-free $O(N)$ work by small polylog(N) factors, are non-obvious. The first such solution was algorithm W of [17] which has (to date) the best worst-case work bound $O(N + P \log^2 N / \log \log N)$ for $1 \leq P \leq N$. This bound was first shown in [22] for a different algorithm and in [29] the basic argument was adapted to algorithm W.

Let us now describe the contents of this survey, with some pointers to the literature, as well as our new contributions. In Section 2.2.2 we present a synthesis of parallel computation and fault models. This synthesis is *new* and includes most of the models proposed to date. It links the work on fail-stop no-restart errors, to fail-stop errors with restarts (both detectable and undetectable restarts).

The detectable restart case has been examined, using a slightly different formalism in [8, 18]. The undetectable restart case is equivalent to the most general general model of asynchrony that has received a fair amount of attention in the literature. An elegant deterministic solution for *Write-All* in this case appeared in [3]. The proof in [3] is existential, because it uses a counting argument. It has recently been made constructive in [33].

For some important early work on asynchronous PRAMs we refer to [9, 10, 15, 22, 23, 30, 32, 34]. In the last three years, randomized asynchronous computation has been examined in depth in [4, 5, 21]. These analyses involve randomness in a central way. They are mostly about off-line or *oblivious* adversaries, which cause faults during the computation but pick the times of these faults before the computation. Although, we will not survey this interesting subject here we would like to point-out that one very promising direction involves combining techniques of randomized asynchronous computation with randomized information dispersal [36]. The work on fault-tolerant and efficient parallel shared memory models has also been applied to distributed message passing models; for example see [1, 11, 12].

In Section 2.2.3 we examine an array of algorithms for the *Write-All* problem. These employ a variety of deterministic techniques and are extensible to

the computation of other functions (see Section 2.2.4). In particular, in Section 2.2.4, we provide *new* bounds for fault-tolerant and efficient computation of parallel prefixes. In Section 2.2.5 we introduce the problem of approximate *Write-All* by computing relations instead of functions. One *new* contribution that we make is to solve approximate *Write-All* optimally. In Section 2.2.6 we survey the state-of-the-art in lower bounds. In Section 2.2.7 we present a *new* complexity classification for fault-tolerant algorithms. We close with a discussion of randomized vs deterministic techniques for fault-tolerant and efficient parallel computation (see Section 2.2.8).

2.2.2 Fault-tolerant parallel computation models

In the first subsection we detail a hierarchy of fail-stop models of parallel computation. We then explain the cost measures of available processor steps and overhead ratio, which we use to characterize *robust* algorithms. The final three subsections contain comments on variations of the processor, memory, and network interconnect parts of our models.

2.2.2.1 Fail-Stop PRAMs

The parallel random access machine (PRAM) of Fortune and Wyllie [14] combines the simplicity of a RAM with the power of parallelism, and a wealth of efficient algorithms exist for it; see surveys [13, 20] for the rationale behind this model and the fundamental algorithms. We build our models of fail-stop PRAM's as extensions of PRAM's.

1. There are Q *shared* memory cells, and the input of size $N \leq Q$ is stored in the first N cells. Except for the cells holding the input, all other memory is cleared, i.e., contains zeroes. Each memory cell can store $\Theta(\log N)$ bits. All processors can access shared memory. For convenience we assume they "know" the input size N, i.e., the $\log N$ bits describing it can be part of their finite state control. For convenience we assume that each processor also has a constant size *private* memory, that only it can access.
2. There are $P \leq N$ initial processors with unique identifiers (PIDs) in the range $1, \ldots, P$. Each processor "knows" its PID and the value of P, i.e., these can be part of its finite state control.
3. The processors that are active all execute synchronously as in the standard PRAM model [14]. Although processors proceed in synchrony and an observer outside the PRAM can associate a "global time" with every event, the processors do not have access to "global time", i.e., processors

can try to keep local clocks by counting their steps and communicating through shared memory but the PRAM does not provide a "global clock".

4. Processors stop without affecting memory. They may also restart, depending on the power of a *fault-inducing adversary*.

In the study of fail-stop PRAMs, we consider four main types of failure-inducing adversaries. These form a hierarchy, based on their power. Note that, each adversary is more powerful than the preceding ones and that the last case can be used to simulate *fully asynchronous* processors [3].

Initial faults: adversary causes processor failures only prior to the start of the computation.

Fail-stop failures: adversary causes stop failures of the processors during the computation; there are no restarts.

Fail-stop failures, detectable restarts: adversary causes stop failures; subsequently to a failure, the adversary might restart a processor and a restarted processor "knows" of the restart.

Fail-stop failures, undetectable restarts: adversary causes stop failures and restarts; a restarted processor does not necessarily "know" of the restart.

Except for the initial failures case, the adversaries are dynamic. A major characteristic of these adversary models is that they are worst-case. These have full information about the structure and the dynamic behavior of the algorithms whose execution they interfere with, while being completely unknown to the algorithms.

Remark on (un)detectable restarts: One way of realizing detectable restarts is by modifying the finite state control of the PRAM. Each instruction can have two parts, a *green* and a *red* part. The green part gets executed under normal conditions. If a processor fails then all memory remains intact, but in the subsequent restart the next instruction red part is executed instead of the green part. For example, the model used in [8, 18] can be realized this way, instead of using "update cycles". The undetectable restarts adversary can also be realized in a similar way by making the algorithm weaker. For undetectable restarts algorithms have to have identical red and green parts. For example, the fully asynchronous model of [3] can be realized this way. □

We formalize failures as follows. A failure pattern F is syntactically defined as 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. This time

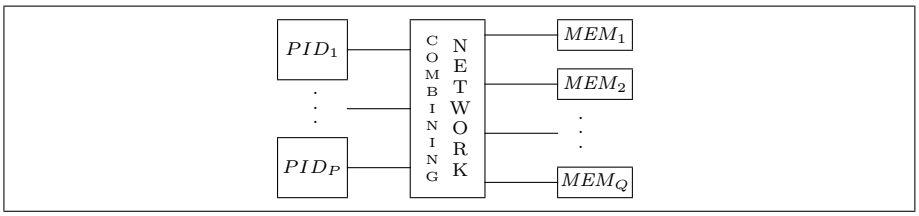

Figure 2.1: An architecture for a fail-stop multiprocessor.

is a global time, that could be assigned by an observer (or adversary) outside the machine. The *size* of the failure pattern F is defined as the cardinality $|F|$, where $|F| \leq M$ for some parameter M.

The abstract model that we are studying can be realized in the architecture in Fig. 2.1. This architecture is more abstract than, e.g., an implementation in terms of hypercubes, but it is simpler to program in. Moreover, various fault-tolerant technologies all contribute towards concrete realizations of its components. There are P *fail-stop* processors [38]. There are Q shared memory cells. These semiconductor memories can be manufactured with built-in fault tolerance using replication and coding techniques [37]. Processors and memory are interconnected via a synchronous network [39]). A combining interconnection network well suited for implementing synchronous concurrent reads and writes is in [24] and can be made more reliable by employing redundancy [2].

2.2.2.2 Measures of Efficiency

We use a generalization of the standard *Parallel-time* $\times$ *Processors* product to measure work of an algorithm when the number of processors performing work fluctuates due to failures or delays [17, 18]. In the measure we account for the *available processor steps* and we do not charge for time steps during which a processor was unavailable due to a failure.

Definition 2.2.1 Consider a parallel computation with P initial processors that terminates in time τ after completing its task on some input data I of size N and in the presence of the 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: $S(I, F, P) = \sum_{i=1}^{\tau} P_i(I, F)$. □

Definition 2.2.2 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 $|F| \leq M$ uses *available processor steps* $S = S_{N,M,P} = \max_{I,F}\{S(I,F,P)\}$. □

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

Definition 2.2.3 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 at most M with at least one surviving processor ($M < N$ for fail-stop model), the algorithm completes its task with $S = S_{N,M,P} \leq c\ T(N) \log^{c'} N$, for fixed c, c'. □

For arbitrary 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 the failure pattern F. The ultimate performance goal is 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: Consider a *Write-All* solution, where it takes a processor one instruction to recover from a failure. If an adversary has a failure pattern F with $|F| = \Omega(N^{1+\varepsilon})$ for $\varepsilon > 0$, then work will be $\Omega(N^{1+\varepsilon})$ regardless of how efficient the algorithm is otherwise.

This illustrates the need for a measure of efficiency that is sensitive to both the size of the input N, and the size of the failure pattern $|F| \leq M$. We thus also introduce the *overhead ratio* σ that amortizes work of the essential work and failures:

Definition 2.2.4 A P-processor PRAM algorithm on any input data I of size $|I| = N$ and in the presence of any pattern F of failures and restarts of size $|F| \leq M$ has *overhead ratio* $\sigma = \sigma_{N,M,P} = \max_{I,F}\left\{\frac{S(I,F,P)}{|I|+|F|}\right\}$. □

When $M = O(P)$ as in the case of the stop failures without restarts, S properly describes the algorithm efficiency, and $\sigma = O(\frac{S_{N,M,P}}{N})$. When F can be large relative to N and P with restarts enabled, σ better reflects the efficiency of fault-tolerant algorithms. We can generalize the definition of σ in Def. 2.2.4 in terms of the ratio $\frac{S(I,F,P)}{T(I)+|F|}$, where $T(I)$ is the time complexity of the best known sequential solution for a particular problem.

2.2.2.3 Processor issues: survivability

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 passing 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.

The fail-stop model with undetectable restarts and dynamic adversaries is the most general fault model we deal with. It can be viewed as a model of parallel computation with arbitrary asynchrony.

Remark on stronger survivability assumption: The default assumption we make is that throughout the computation one processor is fault-free. This assumption can be made stronger, i.e., a constant fraction of the processors are fault-free. We always list the stronger assumption explicitly when used (e.g., in the complexity classification). □

Remark on weaker survivability assumption and restarts: For the models with restarts one can use the weaker survivability assumption that at each global clock tick one processor step executes. In [18] this was stated using "update cycles", but it can be stated using our green-red instruction implementation – remark on (un)detectable restarts.□

2.2.2.4 Memory issues: words vs bits and initialization

In our models we assume that $\log N$-bit word parallel writes are performed atomically in unit time. The algorithms in such models can be modified so that this restriction is relaxed.

The sufficient definition of atomicity is: (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.

The algorithms that assume word atomicity can be mechanically compiled into algorithms that assume only the bit atomicity as stated above.

A much more important assumption in many *Write-All* solutions was 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.*

While this is consistent with definitions of PRAM such as [14], it is nevertheless a requirement that fault-tolerant systems ought to be able to do without. Interestingly there is an efficient deterministic procedure that solves the *Write-All* problem even when the shared memory is *contaminated*, i.e., contains arbitrary values.

2.2.2.5 Interconnect issues: concurrency vs redundancy

The choice of CRCW (concurrent read, concurrent write) model used here is justified because of a lower bound [17] that shows that the CREW (concurrent read, exclusive write) model does not admit fault-tolerant efficient algorithms. However we still would like control memory access concurrency. We define measures that gauge the concurrent memory accesses of a computation.

Definition 2.2.5 Consider a parallel computation with P initial processors that terminates in time τ after completing its task on some input data I of size N in the presence of fail-stop error pattern F. If at time i $(1 \leq i \leq \tau)$, P_i^{R} processors perform reads from N_i^{R} shared memory locations and P_i^{W} processors perform writes to N_i^{W} locations, then we define:

(i) the *read concurrency* ρ as: $\rho = \rho_{I,F,P} = \sum_{i=1}^{\tau} \left(P_i^{\mathrm{R}} - N_i^{\mathrm{R}}\right)$, and

(ii) the *write concurrency* ω as: $\omega = \omega_{I,F,P} = \sum_{i=1}^{\tau} \left(P_i^{\mathrm{W}} - N_i^{\mathrm{W}}\right)$. □

For a single read to (write from) a particular memory location, the read (write) concurrency ρ (ω) for that location is simply the number of readers (writers) minus one. For example, if only one processor reads to (writes from) a location, then ρ (ω) is 0, i.e., no concurrency is involved. Also note that the concurrency measures ρ and ω are cumulative over a computation.

For the algorithms in the EREW model, $\rho = \omega = 0$, while for the CREW model, $\omega = 0$. Thus our measures capture one of the key distinctions among the EREW, CREW and CRCW memory access disciplines.

2.2.3 Robust parallel assignment and Write-All

2.2.3.1 Write-All and initial faults

We first consider the weak model of initial (static) faults in which failures can only occur prior to the start of an algorithm. We assume that the size of the *Write-All* instances is N and that we have P processors, $P' \leq P$ of which are alive at the beginning of the algorithm. Our EREW algorithm E (Fig. 2.2)

```
01 forall processors PID=1..P parbegin
02    Phase E1: Use non-oblivious parallel prefix to compute rank_PID and P'
03    Phase E2: Set x[(rank_PID − 1) * N/P' ... (rank_PID * N/P') − 1] to 1
04 parend
```

Figure 2.2: A high level view of algorithm E.

consists of phases E1 and E2. In phase E1, processors enumerate themselves and compute the total number of live processors. The details of this non-oblivious counting are in [16]. In phase E2, the processors partition the input array so that each processor is responsible for setting to 1 all the entries in its partition.

Theorem 2.2.1 The *Write-All* problem with initial processor and memory faults can be solved in place with $S = O(N + P' \log P)$ on an EREW PRAM, where $1 \leq P \leq N$ and $P - P'$ is the number of initial faults.

With the result of [7] it can be shown that this algorithm is optimal, without memory access concurrency.

2.2.3.2 Dynamic faults and algorithm W

A more sophisticated approach is necessary to obtain an efficient parallel algorithm when the failures are dynamically determined by an on-line adversary. Algorithm W of [17] is an efficient fail-stop *Write-All* solution (Fig. 2.3). It uses full binary trees for processor counting, processor allocation, and progress measurement. Active processors synchronously iterate through the following four phases:

W1: *Processor enumeration.* All the processors traverse bottom-up the processor enumeration tree. A version of parallel prefix algorithm is used resulting in an overestimate of the number of live processors.

W2: *Processor allocation.* All the processors traverse the progress measurement tree top-down using a divide-and-conquer approach based on processor enumeration and are allocated to un-written input cells.

W3: *Work phase.* Processors work at the leaves reached in phase W2.

W4: *Progress measurement.* All the processors traverse bottom-up the progress tree using a version of parallel prefix and compute an underestimate of the progress of the algorithm.

Algorithm W achieves optimality when parameterized using a progress tree with $N/\log N$ leaves and $\log N$ input data associated with each of its leaves.

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

Figure 2.3: A high level view of algorithm W.

By optimality we mean that for a range of processors the work is $O(N)$. A complete description of the algorithm can be found in [17]. Martel [29] gave a tight analysis of algorithm W.

Theorem 2.2.2 [17, 29] Algorithm W is a robust parallel *Write-All* algorithm with $S = O(N + P \log^2 N / \log \log N)$, where N is the input array size and the initial number of processors P is between 1 and N.

Note that the above bound is tight for algorithm W. This upper bound was first shown in [22] for a different algorithm. The data structuring technique [22] might lead to even better bounds for *Write-All*.

2.2.3.3 Dynamic faults, detected restarts, and algorithm V

Algorithm W has efficient work when subjected to arbitrary failure patterns without restarts and it can be extended to handle restarts. However, since accurate processor enumeration is impossible if processors can be restarted at any time, the work of the algorithm becomes inefficient even for some simple adversaries. On the other hand, the second phase of algorithm W does implement efficient top-down divide-and-conquer processor assignment in $O(\log N)$ time when permanent processor PIDs are used. Therefore we produce a modified version of algorithm W, that we call V. To avoid a restatement of the details, the reader is referred to [18].

V uses the optimized algorithm W data structures for progress estimation and processor allocation. The processors iterate through the following three phases based on the phases W2, W3 and W4 of algorithm W:

V1: Processors are allocated as in the phase W2, but using the permanent PIDs. This assures load balancing in $O(\log N)$ time.

V2: Processors perform work, as in the phase W3, at the leaves they reached in phase V1 (there are $\log N$ array elements per leaf).

V3: Processors continue from the phase V2 progress tree leaves and update the progress tree bottom up as in phase W4 in $O(\log N)$ time.

The model assumes re-synchronization on the instruction level, and a wrap-around counter based on the PRAM clock implements synchronization with respect to the phases after detected failures [18]. The work and the overhead ratio of the algorithm are as follows:

Theorem 2.2.3 [18] Algorithm V using $P \leq N$ processors subject to an arbitrary failure and restart pattern F of size M has the work $S = O(N + P \log^2 N + M \log N)$, and its overhead ratio is: $\sigma = O(\log^2 N)$.

Algorithm V achieves optimality for a non-trivial set of parameters:

Corollary 2.2.4 Algorithm V with $P \leq N/\log^2 N$ processors subject to an arbitrary failure and restart pattern of size $M \leq N/\log N$ has $S = O(N)$.

One problem with the above approach is that there could be a large number of restarts and a large amount of work. Algorithm V can be combined with algorithm X of the next section or with the asymptotically better algorithm of [3] to provide better bounds on work.

2.2.3.4 Dynamic faults, undetected restarts, and algorithm X

When the failures cannot be detected, it is still possible to achieve sub-quadratic upper bound for any dynamic failure/restart pattern. We present *Write-All* algorithm X with $S = O(N \cdot P^{\log \frac{3}{2}}) = N \cdot P^{0.59}$. This simple algorithm can be improved to $S = O(N \cdot P^{\epsilon})$ using the method in [3]. We present X for its simplicity and in the next section a (possible) deterministic version of [3].

Algorithm X utilizes a progress tree of size N that is traversed by the processors independently, not in synchronized phases. This reflects the local nature of the processor assignment as opposed to the global assignments used in algorithms V and W. Each processor searches for work in the smallest subtree that has work that needs to be done. It performs the work, and moves to the next subtree.

```
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 subtree rooted at current node u is done then move one level up
05          elseif u is a leaf then perform the work at the leaf
06          elseif u is an interior tree node then
07              Let u_L and u_R be the left and right children of u respectively
08              if the subtrees rooted at u_L and u_R are done then update u
09              elseif only one is done then go to the one that is not done
10              else move to u_L or u_R according to PID bit values
11          fi fi
12      od
13  parend
```

Figure 2.4: A high level view of the algorithm X.

The algorithm is given in Fig. 2.4. Initially the P processors are assigned to the leaves of the progress tree (line 02). The *loop* (lines 03-12) consists of a multi-way decision (lines 04-11). If the current node u 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 08). If a single subtree is not done, the processor moves down appropriately (line 09). For the final case (line 10), the processors move down when neither child is done. Here the processor PID 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 PID, bit 0 will send the processor to the left, and bit 1 to the right.

The performance of algorithm X is characterized as follows:

Theorem 2.2.5 Algorithm X with P processors solves the *Write-All* problem of size N ($P \leq N$) in the fail-stop restartable model with 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.

The algorithm views undetected restarts as delays, and it can be used in the asynchronous model where it has the same work [8]. Algorithm X could also be useful for the case without restarts, even though its worst-case performance without restarts is no better than algorithm W.

Open Problem: A major open problem for the model with undetectable restarts is whether there is robust *Write-All* solution, i.e., where the work is $Npolylog(N)$. Also, whether there is a solution with $\sigma = polylog(N)$.

```
01   forall processors PID = 1..√N parbegin
02     Divide the N array elements into √N work groups of √N elements
03     Each processor 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 fi
09     od
10   parend
```

Figure 2.5: A high level view of the algorithm Y.

2.2.3.5 Dynamic faults, undetected restarts, and algorithm Y

A family of randomized *Write-All* algorithms was presented by Anderson and Woll [3]. The main technique in these algorithms is abstracted in Fig. 2.5. The basic algorithm in [3] 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 (see [19]): Given $P = \sqrt{N}$, we choose the smallest prime m such that $P < m$. Primes are sufficiently dense, so that there is at least one prime between P and $2P$, 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. Each row of this table is a permutation and this structure is a group. Processor with PID i uses the ith permutation as its schedule.

This 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.

Conjecture: We conjecture that the worst case work of this deterministic algorithm is no worse than the expected work of the randomized algorithm. Experimental analysis supports the conjecture. Formal analysis can be reduced to the open problem below that contains an interesting group-theoretic aspect of the multi-processor scheduling problem [41]. In order to show that the worst case work of Y is $O(N \log N)$, 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 $O(m \log m)$.

```
01 forall processors PID=1..P parbegin  --P processors clear N locations
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 Write-All solution with data structures of size N_i and G_{i+1} elements
06     at the leaves to clear memory of size N_{i+1} = N_i · G_{i+1}; i := i + 1
07   od
08 parend
```

Figure 2.6: A high level view of algorithm Z.

2.2.3.6 Bootstrapping and algorithm Z

The *Write-All* algorithms and simulations (e.g., [17, 22, 23, 40]) or the algorithms that can serve as *Write-All* solutions (e.g., the algorithms in [9, 32]) 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, these algorithms perform their computation by "consuming" the clear memory, and concurrently or subsequently clearing segments of memory needed for future iterations. We define an efficient *Write-All* solution that requires no clear shared memory [42].

The solution uses a *bootstrap* approach: In stage 1 all P processors clear an initial segment of N_0 locations in the auxiliary memory. In stage i the P processors clear $N_{i+1} = N_i \cdot G_{i+1}$ memory locations using N_i memory locations that were cleared in stage $i-1$.

Using algorithm W and tuning the parameters N_i and G_i we obtain a solution (algorithm Z, see Fig. 2.6) that for any failure pattern F ($|F| < P$) has work $O(N + P\frac{\log^3 N}{(\log\log N)^2})$ *without any initialization assumption.*

A similar algorithm that *inverts* the bootstrap procedure can be used to clear the contaminated shared memory if the output must contain only the results of the intended computation. The complexity of algorithm Z^{-1} is identical to the complexity of algorithm Z. For algorithm simulation and for transformed algorithms, the complexity cost is *additive* in both cases.

2.2.3.7 Minimizing concurrency: processor priority trees

Among the key lower bound results is the fact that no efficient fault-tolerant CREW PRAM *Write-All* algorithms exist [17] – if the adversary is dynamic then any P-processor solution for the *Write-All* problem of size N will have (deterministic) work $\Omega(N \cdot P)$. Thus memory access concurrency is necessary to combine efficiency and fault-tolerance. However, while most known solutions

for the *Write-All* problem indeed make heavy use of concurrency, the goal of minimizing concurrent access to shared memory is attainable.

We gave a *Write-All* algorithm in [16] in which we bound the *total amount of concurrency* used in terms of the *number of dynamic processor faults* of the actual run of the algorithm.

When there are no faults our algorithm executes as an EREW PRAM and when there are faults the algorithm differs from EREW in the amount of concurrency proportional to the number of faults. The algorithm is based on a conservative policy: concurrent reads or writes occur only when the presence of failures can be inferred and then concurrency is allowed in proportion to the failures detected.

The robust CRCW algorithm $W_{CR/W}$ in [16] is based on algorithm W and it uses processor identifiers to construct *mergeable processor priority trees* (PPT), which control concurrent access to memory. During the execution, the PPTs are *compacted* and *merged* to remove faulty processors and to determine when concurrent access to memory is warranted.

By taking advantage of parallel slackness and by clustering the input data into groups of size $\log N \log P$, we obtain an algorithm that has a range of optimality and that controls its memory access concurrency:

Theorem 2.2.6 Algorithm $\mathrm{W_{CR/W}}$ of [16] with input clustering is a robust *Write-All* algorithm with $S = O(N + P\frac{\log^2 N \log^2 P}{\log\log N})$, write concurrency $\omega \leq |F|$, and read concurrency $\rho \leq 7\,|F| \log N$, where $1 \leq P \leq N$.

The basic algorithm can be extended to handle arbitrary initial memory contents [16] and also to use some pipelining. Finally, [16] shows that there is no robust algorithm whose total write concurrency is bounded by $|F|^{\varepsilon}$ for $0 \leq \varepsilon < 1$.

2.2.4 Computing functions robustly

In this section we will work our way from the simplest to the most complicated functions with robust solutions.

2.2.4.1 Constants, booleans and Write-All

Solving a *Write-All* problem of size N can be viewed as computing a constant vector function. Constant scalar functions are the simplest possible functions (e.g., simpler than boolean OR and AND). At the same time, it appears

that *Write-All* problem is a more difficult (vector) task than computing scalar boolean functions such as multiple input OR and AND. In the lower bounds discussion we consider a model with *memory snapshots*, i.e., processors can read and process the entire shared memory in unit time. For the snapshot model there is a sharp separation between *Write-All* and boolean functions. Clearly any boolean can be computed in constant time in the snapshot model, while we have a lower bound result for any *Write-All* solution in the snapshot model requiring work $\Omega(N \frac{\log N}{\log \log N})$.

Solving a *Write-All* problem is no more difficult than computing any other vector function, e.g., parallel prefix. In the next subsection we also show that the best (as of this writing) *Write-All* solution can be used to derive a robust parallel prefix algorithm that has the same work complexity.

2.2.4.2 Parallel prefix and Write-All

Solutions for the *Write-All* problem can be used as building blocks for custom transformations of efficient parallel algorithms into robust algorithms [17]. Transformations are of interest because in some cases it is possible to improve on the work of oblivious simulation such as [23, 32, 40]. These improvements are most significant for fast algorithms when a full range of processors is used, i.e., when N processors are used to simulate N processors, because in this case parallel slack cannot be taken advantage of.

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

Theorem 2.2.7 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 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. Efficient parallel algorithms and circuits for computing prefix sums were given by Ladner and Fischer in [26], 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 [6, 27], 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 with algorithm W (the most efficient as of this writing *Write-All* solution) whose work is $S_w = O(N\frac{log^2 N}{\log\log N})$, the work of the simulation will be $O(S_w \cdot \log N)$.

We can extend Theorem 2.2.7 to show a robust prefix algorithm whose work is the same as that of algorithm W: $O(S_w) = O(N\frac{\log^2 N}{\log\log N})$. In the fail-stop model we have the following result that uses as the basis an iterative version of the recursive algorithm of [26]:

Theorem 2.2.8 Parallel prefix for N values can be computed using N fail-stop processors using $O(N)$ clear memory with $S = O(N\frac{\log^2 N}{\log\log N})$.

2.2.4.3 List ranking

Another important improvement for the fail-stop case is for the *pointer doubling* operation that is used in many parallel algorithms. The robust algorithm is implemented using a variation of algorithm W and the standard pointer doubling algorithm. We associate each list element with a progress tree leaf. In the work phase of algorithm W we double pointers and update distances. The $\log N$ pointer doubling operations in the work phase make $\log N/\log\log N$ overall iterations sufficient with each iteration performing the same work S_w as algorithm W.

Theorem 2.2.9 There is a robust list ranking algorithm for the fail-stop model with $S^* = O(\frac{\log N}{\log\log N} S_w(P, N))$, where N is the input list size and $S_w(N, P)$ is the complexity of algorithm W for the initial number of processors $P : 1 \leq P \leq N$.

This improvement can be used with several algorithms based on pointer doubling, e.g., algorithms for computing the tree functions of Tarjan and Vishkin [43]. Note also that by preceding the algorithm with $\log N$ pointer doubling operations with $O(N \log N)$ *additive* overhead, we obtain a solution that has no asymptotic degradation in the absence of failures.

2.2.4.4 General Parallel Assignment

Consider computing and storing in an array $x[1..N]$ the values of a function f that depend on PIDs and the initial values of the array x. Assume f can be computed in $O(1)$ sequential time. This is the *general parallel assignment* problem. In [17] it was shown that this quite general operation is equivalent to one *Write-All*.

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

We modify the assignment so that it remains correct when processors fail and when multiple attempts are made to execute the assignment (assuming the surviving processors can be reassigned to the tasks of faulty processors). 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, bit v is the current version number or tag (**mod** 2), 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 any *Write-All* algorithm, with the *general parallel assignment* replacing the trivial "$x[i] = 1$" assignment, will yield a robust N-processor algorithm:

Theorem 2.2.10 The asymptotic work complexities of solving the *general parallel assignment* problem and the *Write-All* problem are equal.

2.2.4.5 Any PRAM steps

The original motivation for studying the *Write-All* problem was that it captured the essence of a single PRAM step computation. It was shown in [23, 40] how to use the *Write-All* paradigm in implementing general PRAM simulations. The generality of this result is somewhat surprising.

Fail-stop faults: An approach to such simulations is given in Fig. 2.7. The simulations are implemented by robustly executing each of the cycles of the PRAM step: instruction fetch, read, compute, and write cycles, and next instruction address computation. This is done using two generations of shared

```
01 forall processors PID=1..P parbegin --Simulate N fault-prone processors
02   The PRAM program for N processors is in shared memory (read-only)
03   Shared memory has two generations: current and future;
04   Initialize N simulated instruction counters to start at the first instruction
05   while there is a simulated processor that has not halted do
06    --Tentative computation: Fetch instruction; Copy registers to scratchpad
07       Perform read cycle using current memory
08       Perform the compute cycle using scratchpad
09       Perform write cycle into future memory
10       Compute next instruction address
11    --Reconcile memory and registers: Copy future locations to current
12   od
13 parend
```

Figure 2.7: Simulations using *Write-All* primitive.

memory, "current" and "future", and by executing each of these cycles in the *general parallel assignment* style, e.g., using algorithm W.

Using such techniques it was shown in [23, 40] that if $S_w(N, P)$ is the efficiency of solving a *Write-All* instance of size N using P processors, and if a linear amount of clear memory is available, then any N-processor PRAM step can be deterministically simulated using P fail-stop processors and work $S_w(N, P)$. If the *Parallel-time* $\times$ *Processors* of an original N-processor algorithm is $\tau \cdot N$, then the work of the fault-tolerant simulation will be $O(\tau \cdot S_w(N, P))$.

The simulation in the fail-stop model is optimal for a wide range of processors [40]. The following theorem might have some practical significance, given the constant overhead.

Theorem 2.2.11 Any N-processor PRAM algorithm can be optimally simulated (with constant overhead) on a fail-stop P-processor CRCW PRAM, when $P \leq N/\log^2 N$. 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 PRAMs of the same type.

When the full range of simulating processors is used ($N = P$) optimality is not achievable. In this cases, our customized parallel prefix and list ranking algorithms improve on the oblivious simulations.

Initial faults: Algorithm E can be used for simulations of EREW PRAM algorithms on fail-stop EREW PRAMs [16]. Simulations are much simpler for this case as compared to the dynamic failures case. The computational overhead of such simulations is *additive*. This simulation is optimal when $P \cdot \tau = \Omega(P' \log P)$.

Theorem 2.2.12 Any P-processor, τ parallel time EREW PRAM algorithm can be robustly simulated on a fail-stop EREW PRAM that is subject to static initial processor and memory faults. The work of the simulation is $P \cdot \tau + O(P' \log P)$, where P' is the number of live processors.

Fail-stop faults with detectable restarts: There is broad range of parameters for 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.

Theorem 2.2.13 Any N-processor PRAM algorithm can be executed on a fail-stop P-processor CRCW PRAM with detectable restarts, 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: $S = O(\min\{N + P\log^2 N + M \log N, \quad N \cdot P^{\log \frac{3}{2}}\})$, and 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 PRAMs of the same type.

Fail-stop faults with undetectable restarts: When the failures are undetectable, deterministic simulations become difficult due to the possibility of processors delayed due to failures writing stale values to shared memory. Fortunately, for fast polylogarithmic time parallel algorithms we can solve this problem by using polylogarithmically more memory. We simply provide as many "future" generations of memory as there are PRAM steps to simulate. Processor registers are stored in shared memory along with each generation of shared memory.

Prior to starting a parallel step simulation, a processor uses binary search to find the newest simulated step. When reading, a processor linearly searches past generations of memory to find the latest written value. In the result below we use the existential algorithm [3].

Theorem 2.2.14 Any N-processor, $\log^{O(1)} N$-time, M-memory PRAM algorithm can be deterministically executed on a fail-stop P-processor CRCW PRAM ($P \leq N$) with undetectable restarts, and using shared memory $M \cdot \log^{O(1)} N$. Each N-processor PRAM step is executed in the presence of any pattern F of *failures* and *undetected restarts* with $S = O(N^{\varepsilon})$.

2.2.5 Computing relations and approximate Write-All

Here we show that computing some relations robustly is easier than computing functions robustly.

Consider the majority relation $\mathcal{M}$: Given an array $x[1..N]$, $x \in \mathcal{M}$ when $|\{x[i] : x[i] = 1\}| > \frac{1}{2}N$. C. Dwork observed that the $\Omega(N \log N)$ lower bound [22] on solving *Write-All* using N processors also applies to producing a member of $\mathcal{M}$ in the presence of failures. It turns out that $O(N \log N)$ work is also sufficient to compute a member of the majority relation.

Let's parameterize the majority problem in terms of the *approximate Write-All* problem by using a quantity ε such that $0 < \varepsilon < \frac{1}{2}$, thus we would like to initialize at least $(1-\varepsilon)N$ array locations to 1. We call this problem the $AWA(\varepsilon)$. Surprisingly, algorithm W has the desired property:

Theorem 2.2.15 Given any constant ε such that $0 < \varepsilon < \frac{1}{2}$, algorithm W solves the $AWA(\varepsilon)$ problem with $S = O(N \log N)$ using N processors.

If we choose $\varepsilon = 1/2^k$ ($k = const$) and then iterate this *Write-All* algorithm $\log \log N$ times, the number of unvisited leaves will be $N\varepsilon^{(\log \log N)} = N(\log N)^{\log \varepsilon} = N(\log N)^{-k} = N/\log^k N$. Thus we can get even closer to solving the *Write-All* problem:

Theorem 2.2.16 For each $k = const$, there is a robust $AWA(\frac{1}{\log^k N})$ algorithm that has work $S = O(N \log N \log \log N)$.

2.2.6 Lower bounds

The strongest known lower bound for *Write-All* was derived by Kedem, Palem, Ragunathan and Spirakis in [22].

Theorem 2.2.17 [22] Given any P-processor CRCW PRAM algorithm for the *Write-All* problem of size N, an adversary can force fail-stop (no restart) errors that result in $N + \Omega(P \log N)$ (where $P \leq N$) steps being performed.

Recently, Martel and Subramonian [31] have extended the Kedem et al. deterministic lower bound [22] to randomized algorithms against oblivious adversaries. It is open whether this lower bound applies to the static fault case.

It was shown in [17] that no optimal solutions for the *Write-All* problem exist that use the range of processor $1 \leq P \leq N$ even when the processors can take *instant memory snapshots*, i.e., processors can read and locally process the entire shared memory at unit cost. The lower bound below applies to fail-stop, deterministic or randomized, PRAMs and it is the *strongest* possible bound under the memory snapshots assumption, i.e., there is a matching upper bound.

Theorem 2.2.18 [17] Given any N-processor CRCW PRAM algorithm for the *Write-All* problem of size N, an adversary can force fail-stop errors that result in $\Omega(N \frac{\log N}{\log \log N})$ steps being performed, even if the processors can read and locally process all shared memory at unit cost.

When restarts are introduced, we show the following result that also is the *strongest* possible result under the snapshot assumption [8]:

Theorem 2.2.19 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 $N + \Omega(P \log P)$ work steps.

The next result shows that CRCW is necessary to achieve efficient solutions to the *Write-All* problem. 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 [20]. However a more severe difference exists between CRCW and CREW PRAMs (and thus also EREW PRAMs) when the processors are subject to failures.

Theorem 2.2.20 Given any deterministic or randomized N-processor CREW PRAM algorithm for the *Write-All* problem, the adversary can force fail-stop errors that result in $\Omega(N^2)$ steps being performed, even if the processors can read and locally process all shared memory at unit cost.

For the CREW PRAMs, Martel and Subramonian [31] show a randomized algorithm with expected work of only $O(N \log N)$ for $P = N$.

2.2.7 A Complexity classification

2.2.7.1 Efficient parallel computation

Many efficient parallel algorithms can be used to show problem membership in the class $\mathcal{NC}$ (of polylog time and polynomial number of processors [35]). The inverse is not necessarily true. This is because the algorithms in $\mathcal{NC}$ allow for polynomial inefficiency in work [25] – the algorithms are fast (polylogarithmic time), but the computational agent can be large (polynomial) relative to the size of a problem [35].

A characterization of parallel algorithm efficiency that takes into account both the parallel time and the size of the computational resource is defined by

Vitter and Simmons [44] and expanded on by Kruskal et al. [25]. The complexity classes in [25] are defined with respect to the time complexity $T(N)$ of the best sequential algorithm for a problem of size N – this is analogous to the definition of *robustness*. Each class is characterized in terms of *parallel time* $\tau(N)$ and, *parallel work* $\tau(N) \cdot P(N)$. We give these class definitions below, but instead of failure-free work, we use the overhead ratio σ that for the failure-free case is simply $\tau(N) \cdot P(N)/T(N)$:

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

ENC:	if $\tau(N) = \log^{O(1)}(T(N))$	and $\sigma = O(1)$.
EP:	if $\tau(N) \leq T(N)^{\varepsilon}$ (const $\varepsilon < 1$)	and $\sigma = O(1)$.
ANC:	if $\tau(N) = \log^{O(1)}(T(N))$	and $\sigma = \log^{O(1)}(T(N))$.
AP:	if $\tau(N) \leq T(N)^{\varepsilon}$ (const $\varepsilon < 1$)	and $\sigma = \log^{O(1)}(T(N))$.
SNC:	if $\tau(N) = \log^{O(1)}(T(N))$	and $\sigma = T(N)^{O(1)}$.
SP:	if $\tau(N) \leq T(N)^{\varepsilon}$ (const $\varepsilon < 1$)	and $\sigma = T(N)^{O(1)}$.

2.2.7.2 Closures under failures

We now define criteria for evaluating whether algorithm transformation preserves the efficiency of the algorithms for each of the classes above.

To use time complexity in comparisons, we need to introduce a measure of time for the fault-tolerant algorithms. In a fault-prone environment, a time metric is meaningful provided that a significant number of processors still are active. Here we use the worst case time provided a linear number of processors are active during the computation. This is our weak survivability assumption. Without this assumption, all one can conclude about the running time is that it is no better than the time of the best sequential algorithm, since the number of active processors might become quite small.

We assuming P is a polynomial in N (note that until now we generally assumed $P \leq N$). Then $\log P = O(\log N)$. We now state the definition:

Definition 2.2.6 Let $\mathcal{C}_{\tau,w}$ be a class with parallel time in the complexity class τ and parallel work in the complexity class w. We say that $\mathcal{C}_{\tau,w}$ is closed with respect to a fault-tolerant transformation ϕ if for any algorithm A in $\mathcal{C}_{\tau,w}$: (1) overhead σ of $\phi(A)$ is such that $\sigma \cdot \tau \cdot P$ is in w, and (2) when the number of active processors at any point of the computation is at least cP for constant $c > 0$, then the running time t is in τ. □

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

Table 2.1: Closure under the fail-stop transformation ξ.

In the fail-stop model without restarts, given any algorithm A, let $\xi(A)$ be the fault-tolerant algorithm that can be constructed as either a simulation or a transformation.

Using, for example, algorithm W as the basis for transforming non-fault-tolerant algorithms, we have the following:

(1) The multiplicative overhead in work is $O(\log N^2/\log\log N)$, and so the worst case overhead σ is $O(\log N^2/\log\log N) = \log^{O(1)} N$ and the worst case work of the fault-tolerant version $\xi(A)$ is $\sigma \cdot \tau(N) \cdot P$.

(2) Algorithm W terminates in $S_w/cP = O(\log^2 N/\log\log N)$ 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 $O(\tau(N)\log^2 N/\log\log N)$.

The resulting closure properties of the classes in [25] under our fail-stop transformation is summarized in Table 2.1.

In the fail-stop model with detectable restarts, for any algorithm A, let $\rho(A)$ be the fault-tolerant algorithm constructed using any of our techniques. In this model we provide existential closure properties by taking advantage of the existential result by Anderson and Woll [3], who showed that for every $\varepsilon > 0$, there exists a deterministic algorithm for P processors that simulates P instructions with $O(P^{1+\varepsilon})$ work. Given the algorithm [3], we interleave it with algorithm V, for example, so that the overhead σ of the combined algorithm is $O(\log^2 N)$. Table 2.2 gives the closure properties under the restartable fail-stop transformation. Note that due to the lower bounds for the *Write-All* problem, the entries that are marked "No" mean non-closure, while the "Unknown" result means that closure is not achieved with the known results.

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

Table 2.2: Closure under the restartable fail-stop transformation ρ.

2.2.8 Discussion: on randomization and approximation

We have presented an overview of the theory of efficient and fault-tolerant parallel algorithms. Our focus has been deterministic algorithms, partly because our work has concentrated on this topic, but also because many deterministic techniques exist for the problems of interest.

We close our exposition with an observation (by D. Michailidis) that illustrates the power of randomization (vs determinism). As we described above deterministic write-all solutions require logarithmic time. This is true even for approximate write-all. However:

Theorem 2.2.21 The approximate *Write-All* problem (AWA) of size N where the number of locations to be written is $N' = \alpha N$ and the number of surviving processors is at least βN, for some constants $0 < \alpha, \beta < 1$ can be solved probabilistically (error is Monte Carlo) on a CRCW PRAM with $O(N)$ expected work in $O(1)$ parallel steps.

Randomization is an important algorithmic tool which has had extensive and fruitful application to fault-tolerance, e.g., [36]. Probabilistic techniques have played a key role in the analysis of asynchronous parallel computing – see for example, [4, 5, 9, 10, 15, 22, 23, 21, 30, 32, 34]. Note however, that it is often hard to compare the analytical bounds of deterministic vs randomized algorithms, since much of the randomized analysis is done using an oblivious adversary assumption.

Randomized algorithms often achieve better practical performance than deterministic ones, even when their analytical bounds are similar. Future developments in asynchronous parallel computation will employ randomization as well as the array of deterministic techniques surveyed here.

Bibliography

[1] M. Ajtai, J. Aspnes, C. Dwork, O. Waarts, "The Competitive Analysis of Wait-Free Algorithms and its Application to the Cooperative Collect Problem", manuscript 1993.

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

[4] Y. Aumann and M.O. Rabin, "Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation", in *Proc. of the 33rd IEEE Symposium on Foundations of Computer Science*, pp. 147-156, 1992.

[5] Y. Aumann, Z.M. Kedem, K.V. Palem, M.O. Rabin, "Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs", in *Proc. of the 34th IEEE Symposium on Foundations of Computer Science*, pp. 271-280, 1993.

[6] 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.

[7] P. Beame, M. Kik and M. Kutylowski, "Information broadcasting by Exclusive Read PRAMs", manuscript 1992.

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

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

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

[11] R. DePrisco, A. Mayer, M. Young, "Time-Optimal Message-Optimal Work performance in the Presence of Faults" manuscript, 1994.

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

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

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

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

[16] P. C. Kanellakis, D. Michailidis, A. A. Shvartsman, "Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms", *7th Int-l Workshop on Distributed Algorithms*, pp. 99-114, 1993.

[17] P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms Can Be Made Robust", *Distributed Computing*, vol. 5, no. 4, pp. 201-217, 1992; prelim. vers. in *Proc. of the 8th ACM PODC*, pp. 211-222, 1989.

[18] 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.

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

[20] 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.

[21] Z. M. Kedem, K. V. Palem, M. O. Rabin, A. Raghunathan, "Efficient Program Transformations for Resilient Parallel Computation via Randomization," in *Proc. 24th ACM Symp. on Theory of Comp.*, pp. 306-318, 1992.

[22] 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.

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

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

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

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

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

[28] 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.

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

[30] C. Martel, A. Park, and R. Subramonian, "Work-optimal Asynchronous Algorithms for Shared Memory Parallel Computers," *SIAM Journal on Computing*, vol. 21, pp. 1070-1099, 1992

[31] C. Martel and R. Subramonian, "On the Complexity of Certified Write-All Algorithms", to appear in *Journal of Algorithms* (a prel. version in the *Proc. of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, December 1992).

[32] 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.

[33] J. Naor, R.M. Roth, "Constructions of Permutation Arrays for Ceratin Scheduling Cost Measures", manuscript, 1993.

[34] N. Nishimura, "Asynchronous Shared Memory Parallel Computation," in *Proc. 3rd ACM Symp. on Parallel Algor. and Architect.*, pp. 76-84, 1990.

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

[36] 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.

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

[38] 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.

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

[40] A. A. Shvartsman, "Achieving Optimal CRCW PRAM Fault-Tolerance", *Information Processing Letters*, vol. 39, no. 2, pp. 59-66, 1991.

[41] A. A. Shvartsman, "Fault-Tolerant and Efficient Parallel Computation", PhD dissertation, Brown University, Tech. Rep. CS-92-23, 1992.

[42] A. A. Shvartsman, "Efficient Write-All Algorithm for Fail-Stop PRAM Without Initialized Memory", *Information Processing Letters*, vol. 44, no. 6, pp. 223-231, 1992.

[43] 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.

[44] J. S. Vitter, R. A. Simmons, "New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems for $\mathcal{P}$," *IEEE Trans. Comput.*, vol. 35, no. 5, 1986.