Efficient parallelism vs reliable distribution: a trade-off for concurrent computations

Paris C. Kanellakis, Dimitrios Michailadis
and Alex A. Shvartsman

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-29
June 1994

Efficient parallelism vs reliable distribution: a trade-off for concurrent computations*

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

[1] Department of Computer Science, Brown University, Box 1910
Providence, RI 02912-1910, USA
[2] Digital Equipment Corporation, 30 Porter Road, LJ02/I11
Littleton, MA 01460-1446, USA

Abstract. Concurrent computations should combine efficiency with reliability, where efficiency is usually associated with parallel and reliability with distributed computing. Such a desirable combination is not always possible, because of an intuitive trade-off: efficiency requires removing redundancy from computations whereas reliability requires some redundancy. We survey a spectrum of algorithmic models (from fail-stop, synchronous to asynchronous and from approximate to exact computations) in which reliability is guaranteed with small trade-offs in efficiency. We illustrate a number of cases where optimal trade-offs are achievable. A basic property of all these models, which is of some interest in the study of concurrency, is that "true" read/write concurrency is necessary for fault tolerance. In particular, we outline (from [14]) how algorithms can be designed so that, in each execution, the total "true" concurrency used can be closely related to the faults that can be tolerated.

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. Here, we overview 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.

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 [15] we demonstrated that it is possible to combine efficiency and fault tolerance for

* Research supported in part by ONR grant N00014-91-J-1613 and by ONR contract N00014-91-J-4052 ARPA Order 8225.

many basic algorithms expressed as concurrent-read concurrent-write parallel (CRCW) random access machines (PRAMs [12]).

The [15] 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 [22, 34]. It was extended in [16] to include *processor restarts*, and in [36] to include *arbitrary static memory faults*, i.e., arbitrary memory initialization, and in [14] 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. Also, as shown in [15], 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 two key assumptions. The first provides the means of saving "correct" state (as the core of fault tolerance) and the second highlights the importance of "true" concurrency (as a source of redundancy for fault tolerance). We assume:

1. Processor faults do not affect memory.
2. Processors can read and write memory concurrently, CRCW.

These two assumptions are essential for deterministic algorithms with *dynamic faults*. *Static* or initial memory can be contaminated [36]. *Static* or initial processor faults can be handled with EREW algorithms [14]. For dynamic behavior there is a trade-off between concurrency and fault tolerance, which we overview here (see [14] for the full details).

A central algorithmic primitive in our work is the *Write-All* operation [15]. Iterated *Write-All* forms the basis for the algorithm simulation techniques of [22, 34] and for the memory initialization of [36]. 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 [22] which uses a certification bit to separate the various iterations of (Certified) *Write-All*. The *Write-All* completes 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 desirable. The first such solution was algorithm W of [15] 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 [21] for a different version of the algorithm and in [24] the basic argument was adapted to algorithm W. The strongest lower bound to date for *Write-All* was derived in [21], namely $\Omega(N + P \log N)$ work is required for $1 \leq P \leq N$.

Let us now describe the contents of this survey, with some pointers to the literature. We focus on upper bounds. In Section 2 we present a synthesis of parallel computation and fault models. This synthesis includes most of the models proposed to date (see [18]). It links the work on fail-stop no-restart errors, to fail-stop errors with restarts, where both detectable and undetectable restarts are considered.

The detectable restart case has been examined, using a slightly different formalism in [6, 16]. It represents an intermediate level on asynchrony for which we have many interesting algorithmic techniques.

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 [28].

For some important early work on asynchronous PRAMs we refer to [7, 8, 13, 21, 22, 25, 26, 27, 29]. In the last three years, randomized asynchronous computation has been examined in depth in [4, 5, 20]. 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 [30]. 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.

The work on fault-tolerant and efficient parallel shared memory models has also been applied to distributed message passing models; see [1, 9, 10].

In Section 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 6). In Section 4 we examine the problem of approximate *Write-All*, which has a work optimal solution. We also provide a randomized solution for this problem, which uses random choices to improve on the work bounds (which become linear). In Section 5 we overview the trade-off between concurrency and fault tolerance for *Write-All*. We apply the resulting concurrency minimization techniques to all functions computable in the fail-stop models, without or with restarts (Section 6). Throughout or exposition we present a number of open questions, which delimit the present state-of-the-art.

2 Fault-Tolerant Parallel Computation Models

In this section we detail a hierarchy of fail-stop models of parallel computation, define relevant complexity measures and characterize *robust* algorithms.

2.1 Fail-Stop PRAMs

The parallel random access machine (PRAM) of Fortune and Wyllie [12] combines the simplicity of a RAM with the power of parallelism, and a wealth of efficient algorithms exist for it; see surveys [11, 19] for the rationale behind this model and the fundamental algorithms. We build our models of fail-stop PRAMs as extensions of the PRAM model.

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. We assume they "know" the input size N, i.e., the $\log N$ bits describing it can be part of their finite state control. 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 [12]. 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 three 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].

Fail-stop failures: adversary causes processors to stop 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 stops and restarts, but a restarted processor does not necessarily "know" of the restart.

A major characteristic of these adversary models is that they are *dynamic and worst-case*. The adversaries 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.

In the case of *randomized algorithms* we assume that the adversary has full information about any random choices (when they are made), but cannot a priori predict what the random bits will be before they are chosen.

We formalize failures as follows. A failure pattern F is syntactically defined as a set of triples $<tag$, PID, $t>$ where tag is either **failure** indicating processor failure, or **restart** indicating a processor restart, PID is the processor identifier, and t is the time indicating when the processor stops or restarts. This time 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.

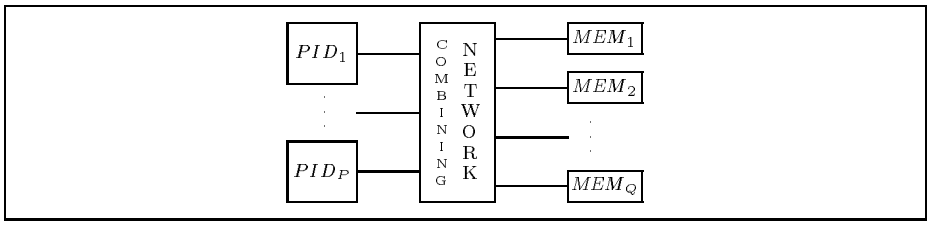

Fig. 1. An architecture for a fail-stop multiprocessor.

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 [6, 16] can be realized this way. 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.

The abstract model we are studying can be realized in the architecture in Fig. 1. There are P *fail-stop* processors [32]. There are Q shared memory cells. These semiconductor memories can be manufactured with built-in fault tolerance using replication and coding techniques [31]. Processors and memory are interconnected via a synchronous network [33]. A combining interconnection network well suited for implementing synchronous concurrent reads and writes is in [23] and can be made more reliable by employing redundancy [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 [15, 16]. 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 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. 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 processor steps measure S is used in turn to define the notion of algorithm *robustness* that combines fault tolerance and efficiency:

Definition 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 M 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.

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 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. 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.

Processor Issues. 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.

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.

Memory Issues. 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 algorithms that assume word atomicity can be mechanically compiled into algorithms that assume only the bit atomicity.

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 [12], 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, see [36].

Interconnect Issues. 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 [19]. However, a more severe difference exists between CRCW and CREW PRAMs (and thus also EREW PRAMs) when the processors are subject to failures.

The choice of CRCW (concurrent read, concurrent write) model used here is justified because of a lower bound [15] that shows that the CREW (concurrent read, exclusive write) model does not admit efficient fault-tolerant algorithms.

Theorem 5 [15]. *Given any deterministic or randomized N-processor* CREW PRAM *algorithm for the* Write-All *problem (where $P = N$), 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.*

However we would still like to control memory access concurrency. We define measures that gauge the concurrent memory accesses of a computation.

Definition 6. 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 from (write to) 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 from (writes to) a location, then ρ (ω) is 0, i.e., no concurrency is involved.

The concurrency measures ρ and ω are cumulative over a computation and a function of the individual computation (not only N, P, M as is S).

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.

3 Exact Write-All Algorithms

3.1 Dynamic Faults and Algorithm W

Algorithm W of [15] is an efficient fail-stop *Write-All* solution (Fig. 2) when the failures are dynamically determined by an on-line adversary. It uses full binary trees for processor enumeration, 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 and an enumeration of the 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. Each leaf of the progress tree is associated with a list L of input data that is sequentially processed (for optimality the size of L is $\log N$).

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 on L using a progress tree with $N/\log N$ leaves and $\log N$ input data associated with each of its leaves. 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 [15]. Martel [24] gave a tight analysis of algorithm W.

Theorem 7 [15, 24]. *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*.*

```
01 forall processors PID=1..P 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
```

Fig. 2. A high level view of algorithm W.

The above bound is tight for algorithm W. This upper bound was first shown in [21] for a different algorithm. The data structuring technique [21] might lead to even better bounds for *Write-All*. The best lower bound to date is $\Omega(N + P \log N)$.

Note that, in a "snapshot" model, where all memory can be read and locally processed in unit time but where writes are accounted as before, algorithm W is optimal in $\Theta(N + P \log N / \log \log N)$ [15, 16].

Open Problem A: Is there an optimal algorithm for *Write-All* in the fail-stop no restart model, e.g., with $\Omega(N + P \log N)$ work?

3.2 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 [16].

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 re-

spect to the phases after detected failures [16]. The work and the overhead ratio of the algorithm are as follows:

Theorem 8 [16]. *Algorithm V using $P \leq N$ processors subject to an arbitrary failure and restart pattern F of size M has work $S = O(N + P \log^2 N + M \log N)$ and overhead ratio $\sigma = O(\log^2 N)$.*

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.

3.3 Dynamic Faults, Undetected Restarts, Algorithms X and Y

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 $S = O(N \cdot P^{\log \frac{3}{2}}) = N \cdot P^{0.59}$ [6, 16]. 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
```

Fig. 3. A high level view of algorithm X.

The algorithm is given in Fig. 3. 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 9 [6, 16]. *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 [6]. 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.

A family of randomized *Write-All* algorithms was presented by Anderson and Woll [3]. The main technique in these algorithms is abstracted in Fig. 4. 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).

```
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 then
06       perform sequential work on the √N elements of the group
07       and mark the group as finished
08     fi
09   od
10 parend
```

Fig. 4. A high level view of the algorithm Y.

We propose the following way of determinizing the algorithm (see [17]): 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.

The most interesting open problems in this area arise in the model of dynamic faults, undetected restarts.

Open Problem B: What is the performance of deterministic algorithm Y? We conjecture that it uses $O(N \log N)$ work.

Open Problem C: The major open problem for the case of undetectable restarts is whether there is a robust *Write-All* solution, i.e., one with work $N \cdot \text{polylog}(N)$. Also, whether there is a solution with $\sigma = \text{polylog}(N)$.

3.4 Algorithms for Initial Faults

An assumption made by all of the above algorithms is that global memory is initially clear. It turns out that this assumption is not critical for the existence of efficient *Write-All* algorithms.

Algorithm Z of [36] is an efficient *Write-All* algorithm for the case that the global memory may be contaminated (i.e., set to arbitrary initial values). It works by combining a *bootstrap* approach with algorithm W. The algorithm makes a number of iterations.

At the beginning, all processors clear a small initial segment of memory. In subsequent iterations the memory cleared in the previous step is used to execute W in order to clear a bigger memory segment. By an appropriate choice of the relationship between the sizes of consecutive segments of clear memory the algorithm can be made to terminate in $O(\log N / \log\log N)$ iterations resulting in a *Write-All* solution that has work $S = O(N + P \log^3 N/(\log\log N)^2)$ *without any initialization assumptions.*

Looking at another variation of *Write-All*, [14] consider the case of *static* processor faults. This is the situation where all the processor faults occur at the beginning of the execution of the *Write-All* algorithm and no processors are allowed to fail once the algorithm has begun executing.

It turns out that this is a much simpler case that does not require any concurrent memory accesses. Algorithm E of [14] solves the *Write-All* problem with static faults optimally with work $S = O(N + P' \log P)$, where $P' \leq P$ is the number of live processors at the beginning of the execution. Like Z this algorithm makes no assumptions about the initial memory contents.

Although much is known about handling static faults, an interesting open problem remains.

Open Problem D: Prove the $\Omega(N + P \log N)$ lower bound on *Write-All* on a CRCW PRAM with initial faults only (the previous lower bounds make use of dynamic faults [21]).

4 Approximate Write-All Algorithms

So far we have required that all array locations be set to 1; this is the *exact Write-All* problem. In this section we relax the requirement that all locations

be set to 1 and instead require that only a fraction of them be written into. For any $0 < \varepsilon < \frac{1}{2}$ we define the *approximate Write-All* problem, denoted AWA(ε), as the problem of initializing at least $(1-\varepsilon)N$ array locations to 1.

We can interpret the exact *Write-All* as a computation of a function, where the result of an exact *Write-All* algorithm is a single vector value (with all locations set to 1), whereas the approximate *Write-All* would correspond to the computation of a relation, where the result could be one of several different vectors (with any $(1-\varepsilon)N$ locations set to 1).

4.1 Computing Approximate Write-All Optimally

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

Consider the majority relation $\mathcal{M}$: Given a binary array $x[1..N]$, $x \in \mathcal{M}$ when $|\{x[i] : x[i] = 1\}| > \frac{1}{2}N$. Dwork observed that the $\Omega(N \log N)$ lower bound of [21] on solving exact *Write-All* using N processors also applies to determining membership in $\mathcal{M}$ 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 the quantity ε such that $0 < \varepsilon < \frac{1}{2}$. As discussed above we would like to initialize at least $(1-\varepsilon)N$ array locations to 1. Surprisingly, algorithm W has the desired property:

Theorem 10 [18]. *Given any constant ε such that $0 < \varepsilon < \frac{1}{2}$, algorithm W solves the AWA(ε) problem with $S = O(N \log N)$ using N processors.*

If we choose $\varepsilon = 1/2^k$ (k a constant) and 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 11 [18]. *For each constant k, there is a robust AWA($\frac{1}{\log^k N}$) algorithm that has work $S = O(N \log N \log\log N)$.*

4.2 Linear Work Using Randomization

We close our exposition of AWA with an observation that illustrates the power of randomization (vs determinism). As shown in [21], deterministic *Write-All* solutions have a work lower bound of $\Omega(N \log N)$ when the number of active processors is N. This is true even for approximate *Write-All*. There is, however, a simple randomized *Write-All* algorithm for the approximate case that requires linear work, under a slightly weaker survivability assumption.

This simple probabilistic algorithm consists of having each processor pick independently at random one of the N locations and set it to 1. If the number of locations that must be written is $N' = \alpha N$ for some constant $0 < \alpha < 1$ this method results in a CRCW Monte-Carlo algorithm with linear expected work.

This algorithmic method has been used in [25, 27]. (For a definition of Monte-Carlo probabilistic algorithms see [19]). Let us first present an analysis without faults.

Lemma 12. *Let N be the size of an array initialized to 0 and let $P \leq N$ be the number of processors of a fault-free CRCW PRAM. If at each parallel step each processor sets to 1 an array location chosen independently uniformly at random, then the expected work required to set to 1 a number $N' < N$ of locations is $O(N \log \frac{N}{N-N'})$ and the expected number of parallel steps is $O(\frac{N}{P} \log \frac{N}{N-N'})$.*

Proof. Since the random choices are independent we consider these choices as if they occured sequentially. Let W_i be the random variable denoting the number of trials needed to go from i to $i+1$ array locations set to 1, $0 \leq i \leq N'-1$. Then the work of the algorithm is $W = W_0 + \cdots + W_{N'-1}$. Each of the W_i's is geometrically distributed with success probability $\frac{N-i}{N}$ so that $E(W_i) = \frac{N}{N-i}$. Then the expected work of the algorithm (expected number of choices needed) is

$$\begin{aligned} E(W) &= \sum_{i=0}^{N'-1} E(W_i) \\ &= \sum_{i=0}^{N'-1} \frac{N}{N-i} \\ &= N\,(H_N - H_{N-N'}) \\ &= N \ln \frac{N}{N-N'} + O(\frac{1}{N}) \\ &= O(N \log \frac{N}{N-N'}) \end{aligned}$$

where H_i is the ith harmonic number. Since the choices are independent P such choices can be made at each parallel step so that the expected number of parallel steps is $E(T) = \frac{E(W)}{P} = O(\frac{N}{P} \log \frac{N}{N-N'})$. □

Clearly, if we allow any number of processor failures, then an adversary can fail enough processors that no *Write-All* algorithm, not even randomized, can terminate in constant parallel time. If we make the additional assumption, however, that the number of surviving processors is linear in the size of the input, then the randomized algorithm given above requires only constant parallel steps.

Theorem 13. *The approximate Write-All problem of size N where the number of locations to be written is αN and the number of surviving processors is at least βN, for some constants $0 < \alpha, \beta < 1$ can be solved probabilistically on a CRCW PRAM with expected $O(N)$ work in expected $O(1)$ parallel steps.*

Proof. From the above theorem for $N' = \alpha N$ we obtain

$$E(W) = N \ln \frac{1}{1-\alpha} + O(\frac{1}{N}) = O(N)$$

and the expected number of parallel steps is

$$E(T) = \frac{1}{\beta} \ln \frac{1}{1-\alpha} + O(\frac{1}{N^2}) = O(1).$$

□

Note that the assumption that βN processors survive is only needed to obtain the bound on the expected number of parallel steps. The expected work does not depend on any survivability assumptions. To have a termination condition in the algorithm we stop the processors after a constant, e.g., $kE(T)$ number of rounds.

Open Problem E: The constant-time randomized algorithm described above is of the Monte-Carlo variety, i.e., it may fail to accomplish its task within the stated bounds. It is an open question whether there is a Las Vegas randomized algorithm (i.e., one that always succeeds) with similar performance bounds.

5 Minimizing Concurrency

Among the key lower bound results is the fact that no efficient fault-tolerant CREW PRAM *Write-All* algorithms exist [15] — 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.

5.1 Minimizing Concurrency: Processor Priority Trees

We gave a *Write-All* algorithm in [14] in which we bound the *total amount of concurrency* used in terms of the *number of dynamic processor faults* of the actual algorithm run. The algorithm is an extension of algorithm W and its key new ingredient is the organization of all processors that need to access a common memory location into a *processor priority tree* (PPT).

In the rest of our discussion we assume that T is a memory location that needs to be accessed by $p \leq P$ processors simultaneously. We concentrate here on concurrent writes and treat concurrent reads in the next section.

A PPT is a binary tree whose nodes are associated with processors based on a processor numbering. All levels of the tree but the last are full and the leaves of the last level are packed as left as possible. PPT nodes are numbered from 1 to p in a breadth-first left-to-right fashion where the parent of the ith node has index $\lfloor i/2 \rfloor$ and its left/right children have indices $2i$ and $2i+1$, respectively. Processors are also numbered from 1 to p and the ith processor is associated with the ith node. *Priorities* are assigned to the processors based on the level they are at. The root has the highest priority and priorities decrease according to the distance from the root. Processors at the same level of the tree have the same priority, which is lower than that of their parents.

Processor priorities are used to determine when a processor can write to the common memory location T. All the processors with the same priority attempt to write to T concurrently but *only if higher priority processors have failed to do so*. To accomplish this the processors of a PPT concurrently execute algorithm CW, shown in Fig. 5. Starting at the root (highest priority) and going down the tree one level at a time, the processors at each level first read T and then concurrently update it iff it contains an old value. This implies that if level i processors effect the write, all processors at higher levels must have failed.

```
01 forall processors ID = 1..p parbegin
02    --Processors write a new value to location T as follows
03   for i = 0 .. ⌊log p⌋ do --For each level i of processor priority tree
04     if 2^i ≤ ID < 2^(i+1) then --The processors at level i
05       READ location T
06       if T does not contain new value then
07         WRITE new value to T
08     fi fi
09   od
10 parend
```

Fig. 5. Pseudocode for the controlled write (CW) algorithm.

Whereas unrestricted concurrent writes by $p \leq P$ processors take only one step to update T, a PPT requires as many steps as there are levels in the tree, i.e., $O(\log p)$. The following lemma is instrumental in bounding the write concurency.

Lemma 14 [14]. *If algorithm* CW *is executed in the presence of a failure pattern F, then its write concurrency ω is no more than the number of failures $|F|$.*

A *Write-All* algorithm with controlled write concurrency can be obtained by incorporating PPTs into the four phases of algorithm W. This is done by replacing each concurrent write in these phases by an execution of CW. We allow $\lfloor \log P \rfloor + 1$ steps for each execution of CW, thus slowing down each iteration of the original algorithm W by a $O(\log P)$ factor. The resulting algorithm is known as $\mathrm{W_{CW}}$ [14].

PPTs are organized and maintained as follows. At the outset of the algorithm and at the beginning of each execution of phase W1 each processor forms a PPT by itself. As processors traverse bottom up the trees used by algorithm W they may encounter processors coming from the sibling node. In this case the two PPTs are *merged* to produce a larger PPT at the common parent of the two nodes. In order to be able to use Lemma 14 to bound the overall write concurrency of the algorithm, we must guarantee that a newly constructed PPT has no processors that have already been accounted for as dead in the above lemma. Such processors are those that are above the PPT level that effected the write. In order to accomplish this we *compact* PPTs before merging to eliminate these processors.

To compact and merge two PPTs, the processors of each PPT store in the memory location they are to update the size of the PPT, the index of the level that effected the write and a timestamp along with the new value to be stored in this location. A timestamp in this context is just a sequence number that need not exceed N (see [15]). Since there can be at most P processors, $O(\log N) + O(\log N) + O(\log P) + O(\log\log P) = O(\log N)$ bits of information need to be stored for the new value, the timestamp, the size of the tree and the level index.

After the $\log P$ steps of CW, the processors of each of the two PPTs to be merged read concurrently the information stored in the two memory locations they updated and compute the size of the resulting PPT, which is the sum of the two sizes read less the number of the certifiably dead processors above the levels that effected the writes. By convention, when two PPTs are merged the processors of the left PPT are added to the bottom of the right one. The timestamp is needed so that processors can check whether the values they read are current (i.e., there is some PPT at the sibling node and merging has to take place) or old. This process is illustrated in Fig. 6.

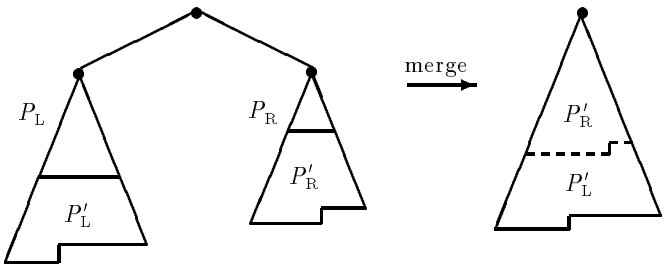

Fig. 6. Merging and compacting two PPTs.

Further details of algorithm $\mathrm{W_{CW}}$ are provided in [14]. The following theorem describes its performance.

Theorem 15 [14]. *Algorithm* $\mathrm{W_{CW}}$ *is a robust algorithm for the Write-All problem with* $\omega \leq |F|$ *and work* $S = O(N \log N \log P + P \log P \log^2 N / \log\log N)$, *where* $1 \leq P \leq N$.

5.2 Controlling Read/Write Concurrency

Algorithm CW can be extended to also handle concurrent reads. The basic idea here is to replace concurrent reads by broadcasts through each PPT. The broadcasts are performed in an elaborate way that allows us to control read concurrency without degrading CW's performance. It relies on the observation that each level of a PPT contains one more processor than all the levels above it. This allows us to use levels $0, \ldots, i-1$ to broadcast to level i in constant time.

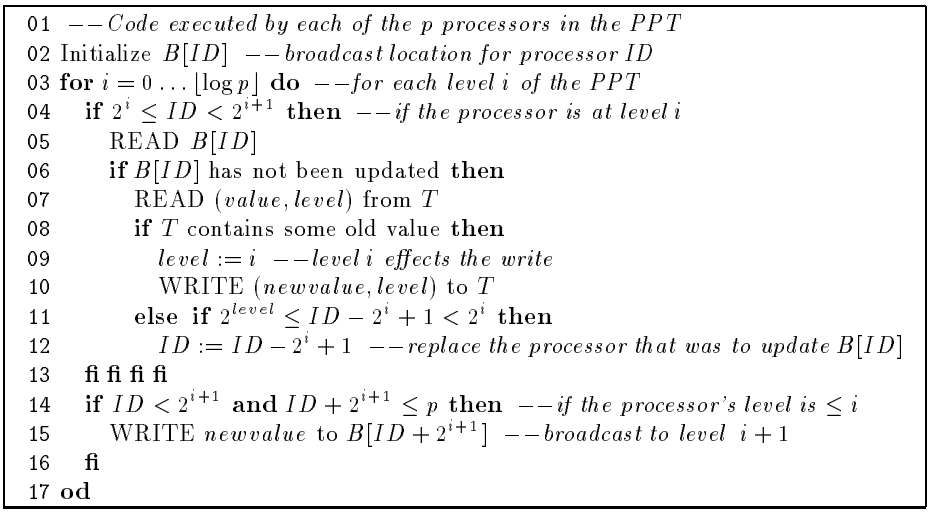

Fig. 7. High level view of algorithm CR/W for the controlled read/write. ID is the position of the processor in the PPT and *newvalue* is the value to be written.

Algorithm CR/W: A high level view of this algorithm, that we call CR/W, is given in Fig. 7. The broadcast takes place through a shared memory array (B in Fig. 7). Such an array is used by each PPT so that processors can communicate with each other based on their positions in the PPT; $B[k]$ stores values read by the kth processor of the PPT. This allows processors to communicate without requiring them to know each other's identity. These arrays can be stored in a segment of memory of size P for all the PPTs (see [14]).

Each processor on levels $0, \ldots, i-1$ is associated with exactly one processor on each of levels $i, i+1, \ldots$. Specifically, the jth processor of the PPT is responsible for broadcasting to the jth (in a left-to-right numbering) processor of each level below its own. The algorithm proceeds in $\lfloor \log p \rfloor + 1$ iterations that correspond to the PPT levels. At iteration i, each processor of level i reads its B location (line 5). If this location has not been updated, then the processor reads T directly (lines 6–7).

Since a PPT level has potentially more live processors than all the levels above it combined, there is in general at least one processor on each level that reads T since no processor at a higher level is assigned to it. If a level is full, this processor is the rightmost one (the root of the PPT for level 0). As long as there are no failures this is the only direct access to T. Concurrent accesses can occur only in the presence of failures. In such a case several processors on the same level may fail to receive values from processors at higher levels, in which case they will access T directly incurring concurrent reads.

A processor that needs to read T directly first checks whether it contains the value to be written (line 8) and then writes to it (line 10) if it does not. As can be seen from the figure, whenever processors update T they write not only the

new value for T but also the index of the level that effected the write.

If a processor P_k that accesses T directly verifies that T has the correct value and the failed processor that should have broadcast to P_k is at or below the level that effected the write, then P_k assumes the position of the failed processor in the PPT (lines 11–12). This effectively moves failed processors towards the leaves of the PPT and plays an important role in establishing the bound on the read concurrency. Failed processors are moved towards the leaves only if they are not above the level that effects the write since processors above this level will be eliminated by compaction as was done in the previous section.

Algorithms CR1 and CR2: In addition to CR/W we use two simpler algorithms when all the processors of a PPT need to read a common memory location but no write is involved.

The *first* of them, referred to as CR1, is similar to CR/W but without the write step (lines 8–10). It is also simpler than CR/W in that processors that are found to have failed are pushed towards the bottom of the PPT irrespective of the level they are at. This algorithm is used for bottom-up traversals during W.

The *second* algorithm, referred to as CR2, uses a simple top-down broadcast through the PPT. Starting with the root each processor broadcasts to its two children; if a processor fails then its two children read T directly. Thus the processors of level i are responsible to broadcast only to processors of level $i+1$. No processor movement takes place. This algorithm is used for top-down traversals during W.

Clearly, all three of CR/W, CR1, and CR2 require $O(\log P)$ time for PPTs of at most P processors.

Algorithm $\mathrm{W}_{\mathrm{CR/W}}$**:** Incorporating algorithms CR/W, CR1, and CR2 along with the techniques of PPT compaction and merging into W results in a *Write-All* algorithm that controls both read and write concurrency. The details of this algorithm, known as $\mathrm{W}_{\mathrm{CR/W}}$, can be found in [14]. The following theorem provides bounds on the performance of $\mathrm{W}_{\mathrm{CR/W}}$.

Theorem 16 [14]. *Algorithm* $\mathrm{W}_{\mathrm{CR/W}}$ *is a robust algorithm for the Write-All problem with* $S = O(N \log N \log P + P \log P \log^2 N / \log\log N)$*, read concurrency* $\rho \leq 7\,|F| \log N$*, and write concurrency* $\omega \leq |F|$*, where* $1 \leq P \leq N$*.*

By taking advantage of parallel slackness and by clustering the input data into groups of size $\log N \log P$, we can obtain a modification of algorithm $\mathrm{W}_{\mathrm{CR/W}}$ that has a non trivial processor optimality range. The modified algorithm, known as $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$, is described in [14]. Its performance is characterized by the following theorem:

Theorem 17 [14]. *Algorithm* $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ *is a robust algorithm for the Write-All problem 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 algorithm can be further extended to handle arbitrary initial memory contents [14]. It is also possible to reduce the maximum *per step* memory access

concurrency by polylogarithmic factors by utilizing a general pipelining technique. Finally, [14] established a lower bound showing that no robust *Write-All* algorithm exists with write concurrency $\omega \leq |F|^{\varepsilon}$ for $0 \leq \varepsilon < 1$.

6 General Computations with Minimal Concurrency

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

6.1 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 the *Write-All* problem is a more difficult (vector) task than computing scalar boolean functions such as multiple input OR and AND. One of the known lower bounds applies to the model with *memory snapshots*, i.e., processors can read and process the entire shared memory in unit time [15]. For the snapshot model there is a sharp separation between *Write-All* and boolean functions. Clearly, any boolean function can be computed in constant time and $O(N)$ work in the snapshot model, while we have a lower bound result for any *Write-All* solution in the snapshot model requiring work $\Omega(N \frac{\log N}{\log \log N})$.

6.2 General Parallel Assignment

Consider computing and storing in an array $x[1..N]$ the values of a vector function f that depend on PIDs and the initial values of the array x. Assume each of the N scalar components of f can be computed in $O(1)$ sequential time. This is the *general parallel assignment* problem.

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

In [15] a general technique was shown for making this operation robust using the same work as required by *Write-All*. 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 modified *general parallel assignment* replacing the trivial "$x[i] = 1$" assignment, will yield a robust N-processor algorithm:

Theorem 18. *The asymptotic work complexities of solving the general parallel assignment problem and the Write-All problem are equal.*

6.3 Any PRAM Step

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 [22, 34] 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. 8. 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 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 [22, 34] 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 $O(\tau \cdot S_w(N, P))$.

By using algorithm $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ we obtain efficient PRAM simulations on fail-stop PRAMs whose concurrency depends on the number of failures. In particular, we obtain the following:

Theorem 19 [14]. *Any N processor, τ parallel time EREW PRAM algorithm can be simulated on a fail-stop P-processor CRCW PRAM using work $O(\tau \cdot (N + \frac{P \log^2 P \log^2 N}{\log \log N}))$ so that the write concurrency of the simulation is $\omega \leq |F|$ and the read concurrency is $\rho \leq 7|F| \log N$, for any fail-stop failure pattern F and $1 \leq P \leq N$.*

The work bound of this theorem also holds for simulations of non-EREW PRAM algorithms but the concurrency bounds depend on the concurrency of the simulated algorithm. If ω_0 and ρ_0 are the write and read concurrencies of the simulated algorithm, respectively, then for the simulation we have $\omega \leq |F| + \omega_0$ and $\rho \leq 7|F| \log N + \rho_0 O(\log N)$. Alternatively, we can maintain the same concurrency bounds at the expense of increasing the work by a logarithmic factor

by first converting the original algorithm into an equivalent EREW algorithm [19].

When the full range of simulating processors is used ($P = N$) optimality is not achievable. In this case customized transformations of parallel algorithms (such as prefix and list ranking algorithms [25, 35]) may improve on the oblivious simulations.

```
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
```

Fig. 8. Simulations using *Write-All* primitive.

Fail-stop faults with detectable restarts: For the model with detectable restarts we have the following simulation theorem:

Theorem 20. *Any N processor, τ parallel time EREW PRAM algorithm can be simulated on a fail-stop P-processor CRCW PRAM with detectable restarts using work $O(\tau \cdot (N \log P + P \log P \log^2 N) + |F| \log N \log P)$ and overhead ratio $\sigma = O(\log^2 N \log P)$ so that the write concurrency of the simulation is $\omega \leq |F|$ and the read concurrency is $\rho \leq 7|F| \log N$, for any failure/restart pattern F and $1 \leq P \leq N$.*

Proof. To obtain this result we use algorithm $\mathrm{W_{CR/W}}$ with $\log N$ elements clustered at each leaf of the progress tree and then apply the analysis of algorithm V [16]. This algorithm differs from $\mathrm{W_{CR/W}}$ for the no-restarts model in that a failed processor upon restarting waits until the beginning of the next W1 phase. This gives rise to the $|F| \log N \log P$ term in the work bound. □

Note that the above algorithm is not optimal and the work may be unbounded since F may be arbitrarily large.

Open Problem F: It is open whether there exist robust simulations that achieve both limited concurrency and bounded work for the detectable restart model.

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. Fortu-

nately, 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 21. *Any N-processor, $\log^{O(1)} N$-time, Q-memory* PRAM *alg. can be determin. executed on a fail-stop P processor* CRCW PRAM *($P \leq N$) with undetectable restarts, and using shared memory $Q \cdot \log^{O(1)} N$. Each N-processor PRAM step is executed in the presence of any pattern F of* failures *and* undetected restarts *with $S = O(N^{1+\varepsilon})$, for any positive ε.*

Open Problem G: It is open whether there exist (robust) PRAM simulations with limited concurrency for the undetectable restart model.

The fail-stop model with undetectable restarts is clearly the most challenging model, both from a technical and from a practical point of view. The simulation result above is considerably weaker than what is achievable for the other models. This is an area where the application of randomized techniques (e.g., [4, 5, 20]) is most promising.

References

1. M. Ajtai, J. Aspnes, C. Dwork, O. Waarts, "The Competitive Analysis of Wait-Free Algorithms and its Application to the Cooperative Collect Problem", *to appear in PODC*, 1994.
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. 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.
7. 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.
8. R. Cole and O. Zajicek, "The Expected Advantage of Asynchrony," in *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, pp. 85-94, 1990.

9. R. DePrisco, A. Mayer, M. Yung, "Time-Optimal Message-Efficient Work Performance in the Presence of Faults," *to appear in PODC*, 1994.
10. 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.
11. D. Eppstein and Z. Galil, "Parallel Techniques for Combinatorial Computation", *Annual Computer Science Review*, 3 (1988), pp. 233-83.
12. S. Fortune and J. Wyllie, "Parallelism in Random Access Machines", *Proc. the 10th ACM Symposium on Theory of Computing*, pp. 114-118, 1978.
13. P. Gibbons, "A More Practical PRAM Model," in *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 158-168, 1989.
14. P. C. Kanellakis, D. Michailidis, A. A. Shvartsman, "Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms", *7th Intl Workshop on Distributed Algorithms*, pp. 99-114, 1993. An extended version appears as TR CS-94-23, Brown University.
15. 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.
16. 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.
17. P. C. Kanellakis and A. A. Shvartsman, "Robust Computing with Fail-Stop Processors", in *Proc. of the Second Annual ONR Workshop on Ultradependable Multicomputers*, Office of Naval Research, pp. 55-60, 1991.
18. P. C. Kanellakis and A. A. Shvartsman, "Fault Tolerance and Efficiency in Massively Parallel Algorithms", in *Foundations of Dependable Computing*, vol. II, chapter 2.2, G. Koob, C. Lau (editors), Kluwer, 1994 (to appear).
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. C. Martel, personal communication, March, 1991.
25. 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
26. 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).

27. 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.
28. J. Naor, R.M. Roth, "Constructions of Permutation Arrays for Ceratin Scheduling Cost Measures", manuscript, 1993.
29. N. Nishimura, "Asynchronous Shared Memory Parallel Computation," in *Proc. 3rd ACM Symp. on Parallel Algor. and Architect.*, pp. 76-84, 1990.
30. 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.
31. D. B. Sarrazin and M. Malek, "Fault-Tolerant Semiconductor Memories", *IEEE Computer*, vol. 17, no. 8, pp. 49-56, 1984.
32. 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.
33. J. T. Schwartz, "Ultracomputers", *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 4, pp. 484-521, 1980.
34. A. A. Shvartsman, "Achieving Optimal CRCW PRAM Fault Tolerance", *Information Processing Letters*, vol. 39, no. 2, pp. 59-66, 1991.
35. A. A. Shvartsman, *Fault-Tolerant and Efficient Parallel Computation*, Ph.D. dissertation, Brown University, Tech. Rep. CS-92-23, 1992.
36. 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.

This article was processed using the LaTeX macro package with LLNCS style