Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms

Paris C. Kanellakis Dimitrios Michailidis
and Alex A. Shvartsman

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-23
May 1994

Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms*

Paris C. Kanellakis† Dimitrios Michailidis† Alex A. Shvartsman‡

May 9, 1994

Abstract. The CRCW PRAM under dynamic fail-stop (no restart) processor behavior is a fault-prone multiprocessor model for which it is possible to both guarantee reliability and preserve efficiency. To handle dynamic faults some redundancy is necessary in the form of many processors concurrently performing a common read or write task. In this paper we show how to significantly decrease this concurrency by bounding it in terms of the number of actual processor faults. We describe a low concurrency, efficient and fault-tolerant algorithm for the *Write-All* primitive: "using $\leq N$ processors, write 1's into N locations". This primitive can serve as the basis for efficient fault-tolerant simulations of algorithms written for fault-free PRAMs on fault-prone PRAMs. For any dynamic failure pattern F, our algorithm has total write concurrency $\leq |F|$ and total read concurrency $\leq 7\,|F|\log N$, where $|F|$ is the number of processor faults (for example, there is no concurrency in a run without failures); note that, previous algorithms used $\Omega(N \log N)$ concurrency even in the absence of faults. We also describe a technique for limiting the per step concurrency and present an optimal fault-tolerant EREW PRAM algorithm for Write-All, when all processor faults are initial.

1. Introduction

1.1. Motivation and Background

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.

Such combinations of fault and computation models are interesting because they illustrate good trade-offs between reliability and efficiency. To see that there is a trade-off consider that reliability usually requires *adding redundancy* to the computation in order to

*Research supported by ONR grant N00014-91-J-1613. A preliminary version appeared in [17].

†Department of Computer Science, Brown University, Box 1910, Providence, RI 02912-1910, USA.

‡Digital Equipment Corporation, 30 Porter Road, LJ02/I11, Littleton, MA 01460-1446, USA.

detect errors and reassign resources, whereas gaining efficiency by massively parallel computing requires *removing redundancy* from the computation to fully utilize each processor.

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 [18] 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). The [18] fault model allows *any pattern of dynamic fail-stop no restart processor errors, as long as one processor remains alive.* Interestingly, our technique can be extended to *all* CRCW PRAMs. In [20] it is shown how to simulate any fault-free PRAM on a CRCW PRAM that executes in a fail-stop no-restart environment with comparable efficiency and in [31] it is shown how the simulation can be made optimal using processor slackness. The fault model was extended in [32] to include *arbitrary static memory faults*, i.e., arbitrary memory initialization. The CRCW PRAM computation model is a widely used abstraction of real massively parallel machines. As shown in [18], 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.

All the above-mentioned algorithmic work makes two somewhat unrealistic, but critical, assumptions. It is assumed that: (1) processors can read and write memory concurrently, and (2) processor faults are fail-stop without restarts. This paper is an investigation of assumption (1) about the model of computation, while still making assumption (2) about the fault model. Specifically, the redundancy which is essential for the above algorithmic techniques seems to be the result of more than one processor concurrently performing a common read or write task. Our contribution here is to show how to control and minimize this redundancy.

Our analysis is deterministic with worst-case failure-inducing adversaries. For this we still assume (2) above, namely that the dynamic processor errors do not affect shared memory and processors once stopped do not resume computation. More general processor asynchrony has been examined in [2, 3, 4, 7, 8, 9, 11, 14, 20, 21, 22, 25, 26, 27]. Many of these analyses involve average processor behavior and use randomization. Some (e.g., [2, 7, 22]) deal with deterministic asynchronous computation (i.e., removing assumption (2) without using randomness). In this case the strongest lower bounds are in [22] and the tightest upper bounds are in [2].

A key primitive in much of the above mentioned work is the Write-All operation of [18]. The main technical insight in this paper is an efficient, deterministic fault-tolerant Write-All algorithm that significantly reduces memory access concurrency.

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. In the fail-stop PRAM model the P processors share a global clock and are fail-stop no restart.

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 [18] 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 shown in [22] for a variant of algorithm W and in [24] the same bound was shown for algorithm W). Algorithm W was extended to handle arbitrary initial memory in [32] and served as a building block for transforming fault-free PRAMs into fault-prone PRAMs of comparable efficiency in [20, 31].

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 [20] 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 algorithm W. This is also the case for the algorithms presented here.

In Section 2.1 we present our fault and computation models. In Section 2.2 we review our measures of efficiency and introduce a measure of concurrency. To make the exposition self contained we present in Section 2.3 an outline of the main algorithmic ideas of algorithm W and sketch how it is utilized in PRAM simulation and memory clearing.

Memory access concurrency is the main topic of this paper. In [18] it is shown that read and write concurrency are necessary for deterministic efficient solutions under dynamic processor faults. All Write-All solutions that have efficient work for any dynamic processor failures require concurrent-read concurrent-write (CRCW) PRAMs. Moreover, all such solutions to date make very liberal use of concurrent reads and writes.

1.2. Summary of Results

In Section 3 we present our main contribution, that *we can bound the total amount of memory access concurrency in terms of the number of dynamic processor faults of the actual run of the algorithm.*

When there are no faults our new algorithm executes as an EREW PRAM and when there are faults the algorithm differs from an EREW PRAM in an amount of concurrency related to the number of faults. The algorithm is based on a conservative policy towards concurrency: concurrent reads or writes occur only when the presence of failures can be inferred from what processors know (i.e., when failures are detected) and these concurrent operations happen in proportion to the failures detected.

We describe an efficient fault-tolerant CRCW algorithm for the Write-All problem such that, for any failure pattern F in a run of the algorithm the total write concurrency is $\leq |F|$ and the total read concurrency is $\leq 7\,|F|\log N$, where $|F|$ is the number of failures. The algorithm is based on algorithm W and makes use of processor identifiers to construct mergeable processor priority trees, which control concurrent access to memory. We present it in two steps, first controlled write concurrency (Section 3.1) and then controlled read concurrency (Section 3.2). The work of the Write-All algorithm described in Section 3.2 is $O(N\log N\log P + P\log P\log^2 N/\log\log N)$, where $1 \leq P \leq N$.

In Section 3.3 we utilize parallel slackness to obtain a Write-All algorithm that has optimal work for a non-trivial range of processors. The algorithm has work $O(N + \frac{P\log^2 P\log^2 N}{\log\log N})$

and is optimal for $P \log^2 P \leq N \log\log N / \log^2 N$. We extend this algorithm to handle arbitrary initial memory contents in Section 3.4. In Section 3.5 we demonstrate that there is no efficient and fault-tolerant algorithm whose total write concurrency is bounded by $|F|^\varepsilon$ for $0 \leq \varepsilon < 1$. In Section 3.6 we describe how this algorithm can be used as a building block for efficient simulations of arbitrary PRAMs. This is a dynamic version of Brent's lemma [6].

In Section 4 we examine the worst-case concurrency per step. This is a different measure than the overall concurrency of Section 3. We present a variation of algorithm W whose *maximum read/write concurrency per step* is $N/\text{polylog}(N)$ instead of N. The point here is decreasing concurrency per step by polylog amounts using a general pipelining technique of synchronizing processors into waves. This idea can be combined with those of Section 3.

The case of static faults is examined in Section 5. Using a variant of the parallel prefix operation, we give a $\Theta(N + P' \log P)$-work EREW algorithm for Write-All, where P' is the number of live processors and $P - P'$ is the number of initial faults. For $P = N$, this solution is optimal (based on a result of [5]. For general simulations of PRAM algorithms our solution introduces a $\Theta(P' \log P)$ additive overhead. It also handles static memory faults at no additional overhead. In summary, the static case has a simple optimal solution which eliminates memory access concurrency.

We conclude with some open problems in Section 6.

2. The Model and Base Algorithms

2.1. Fail-Stop PRAM

The parallel random access machine (PRAM) of Fortune and Wyllie [13] combines the simplicity of RAM with the power of parallelism, and a wealth of efficient algorithms exist for it; see surveys [12, 19] for the rationale behind this model and the fundamental algorithms. The main features of this model are as follows:

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

2. The memory accessible to all processors is denoted as *shared*, each processor also has a constant size local memory denoted as *private*. Each memory cell can store $\Theta(\log \max\{N, P\})$ bits on inputs of size N.

3. The input is stored in N cells in shared memory and the rest of the shared memory is cleared (i.e., contains zeroes). The processors know the input size N.

In the study of fail-stop PRAMs, this base model is extended with a failure model:

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

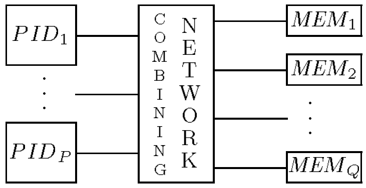

Fig. 1. An architecture for a fail-stop multiprocessor.

5. We consider *fail-stop* (no restart) errors: processors fail by stopping and not performing any further actions; their private memory is lost, but shared memory is not affected by processor failures.

6. *Shared memory writes* are *atomic* with respect to failures: failures can occur before or after a shared write of $\Theta(\log \max\{N, P\})$-bit words, but not during the write. (This non-trivial assumption is made only for simplicity of presentation; algorithms that make it can be converted to use only single bit atomic writes [18].) We also handle *arbitrary initialization of the shared memory*, which is assumed clear in the fault-free PRAM model. (This extends 3 above.)

The abstract model that we are studying can be realized in the architecture of Fig. 1. Fault-tolerant technologies, such as those in surveys [10, 15, 16], all contribute towards concrete realizations of its components.

a. There are P *fail-stop* processors (see [29]), each with a unique address and some local memory.

b. There are Q shared memory cells, the input of size $N \leq Q$ is stored in shared memory. These semiconductor memories can be manufactured with built-in fault tolerance using replication and coding techniques without appreciably degrading performance [28].

c. Processors and memory are interconnected via a synchronous network (e.g., as in the Ultracomputer [30]). A combining interconnection network that is well suited for implementing synchronous concurrent reads and writes is studied in [23] (the combining properties are used in their simplest form only to implement concurrent access to memory). The network can be made more reliable by employing redundancy [1].

2.2. Measures of Efficiency

The complexity measure used throughout this work is the *available processor steps* of [18]. It generalizes the fault-free *Parallel-time* $\times$ *Processors* product and accounts for all steps performed by the active processors.

Definition 2.1. Consider a computation with P initial processors that terminates in parallel time τ after completing its task on some input data I of size $|I| = N$, and in the presence of some fail-stop error pattern F of size $|F| < P$. If $P_i(I, F) \leq P$ is the number of processors completing an instruction at time i, we define the *available processor steps of this computation* as

$$S_{I,F,P} = \sum_{i=1}^{\tau} P_i(I, F),$$

and the *available processor steps S of an algorithm* as

$$S = S(N, P) = \max_{I,F}\{S_{I,F,P} : |I| = N, |F| < P\}.$$

□

The notion of algorithm *robustness* combines fault tolerance and efficiency:

Definition 2.2. 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 $|I| = N$ and for any number of initial processors P $(1 \leq P \leq N)$ and for any failure pattern F of size $|F| < P$, this algorithm completes its task and it has $S = S(N, P) \leq c\, T(N) \log^{c'} N$, for fixed c, c'. □

We now introduce new measures to gauge the concurrent memory accesses of a computation.

Definition 2.3. Consider a computation with P initial processors that terminates in parallel time τ after completing its task on some input data I of size $|I| = N$ in the presence of fail-stop error pattern F of size $|F| < P$. If at time i $(1 \leq i \leq \tau)$, P_i^{R} processors complete reads from N_i^{R} shared memory locations and P_i^{W} processors complete writes to N_i^{W} locations, then we define:

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

Remark 1. For concurrent reads from (writes to) a particular memory location, the read (write) concurrency for that location is simply the number of readers (writers) minus one. For example, if only one processor reads from (writes to) to a location, then the concurrency is 0, i.e., no concurrency is involved. This leads to an equivalent definition of ρ and ω: For a given computation, let R_{ij} be the number of reads from location i during step j, and W_{ij} be the number of writes to location i during step j. Then ρ for the computation is $\sum_{j=1}^{\tau} \sum_{i:R_{ij}>0}(R_{ij}-1)$, and ω is $\sum_{j=1}^{\tau} \sum_{i:W_{ij}>0}(W_{ij}-1)$. It is easy to see that this definition of ρ and ω is equivalent to Definition 2.3 above.

Remark 2. Note that ρ and ω are defined for a computation, or actual run, of an algorithm. They are cumulative measures over all the steps of the computation. S is the maximum over all computations where $|I| = N$ and $|F| < P$. Our bounds for S are worst-case over all inputs and failure patterns, but our bounds for ρ and ω depend on $|F|$ and $|I| = N$.

```
forall processors PID=1..N parbegin
   Phase W3: Visit the leaves based on PID to perform work on the input data
   Phase W4: Traverse the progress tree bottom up to measure progress
   while the root of the progress tree is not N do
      Phase W1: Traverse the processor enumeration tree bottom up to enumerate processors
      Phase W2: Traverse the progress tree top down to reschedule work
      Phase W3: Perform rescheduled work on the input data
      Phase W4: Traverse the progress tree bottom up to measure progress
   od
parend
```

Fig. 2. A high level view of algorithm W for $P = N$.

Remark 3. ρ and ω can be used to characterize the type of PRAM that an execution of an algorithm requires. An execution that has $\rho = \omega = 0$ can be carried out on an EREW PRAM while a CREW PRAM suffices for an execution with $\omega = 0$. This does not imply that the algorithm itself is an EREW or CREW algorithm. However, an algorithm is EREW if its *worst case* ρ and ω are 0 and it is CREW if its *worst case* ω is 0.

2.3. Background Algorithms

2.3.1. Algorithm W

Algorithm W of [18] is a robust Write-All solution. It uses two complete binary trees as data structures: the *processor enumeration* and the *progress* trees. A high level view of algorithm W is in Fig. 2. It is an iterative algorithm in which all active processors synchronously execute the following four phases:

- *Phase* W1 — *processor enumeration.* All processors traverse bottom-up the processor enumeration tree always starting at the same leaf, i.e, this traversal is static. This phase utilizes a version of the standard parallel prefix algorithm. The result is an overestimate of the number of live processors in every subtree and a numbering of the live processors.

- *Phase* W2 — *processor allocation.* The processors traverse top-down the progress tree using a divide-and-conquer approach (based on processor enumeration and progress measurement) to allocate themselves to the unvisited leaves of the progress tree.

- *Phase* W3 — *work phase.* The processors work at the leaves they reached in W2.

- *Phase* W4 — *progress measurement.* The processors traverse bottom-up the progress tree and compute an underestimate of the progress of the algorithm for each subtree. They start from the leaves they were at in phase W3. A version of the standard parallel summation algorithm is employed to compute underestimates of the progress for each subtree.

```
forall processors PID=1..P parbegin  --P fail-stop processors simulate N fault-prone ones
  The PRAM program for N processors is stored in shared memory (read-only)
  Shared memory has two generations: current and future; the rest of shared memory is clear
  Initialize N simulated instruction counters to start at the first instruction
  while there is a simulated processor that has not halted do
   --Perform tentative computation: Fetch next instruction; Copy registers to scratchpad;
     Do read cycle using "current" memory; Perform the compute cycle;
     Do write cycle into the "future" memory; Compute next instruction address
   --Reconcile memory and registers: Copy "future" memory and registers to "current"
  od
parend
```

Fig. 3. Simulations using Write-All primitive and two generations of shared memory.

The following lemma of [24] provides a bound on the total number of block steps executed by all processors (where a block step is an execution by one processor of the body of the while-loop in Fig. 2). It will be used in our analyses in Section 3.

Lemma 2.1 ([24]). *Let U be the number of unvisited leaves of the progress tree and P be the number of processors at the beginning of algorithm* W. *Let U_i and P_i be the number of unvisited leaves of the progress tree and processors, respectively, at the start of the ith iteration of the while loop. Then the total number of block steps of all iterations for which $P_i < U_i$ is at most U and the total number of block steps of iterations for which $P_i \geq U_i$ is at most $O(P \frac{\log U}{\log \log U})$. The overall number of block steps is $O(U + P \frac{\log U}{\log \log U})$.* □

Algorithm W can be parameterized by clustering elements of the input array and assigning one cluster to each leaf of the progress tree. We will use the notation W[s] to specify that the input elements form clusters of size s; in this case the progrees tree has N/s leaves, one for each cluster. When using P processors such that $P > \frac{N}{s}$, it is sufficient for each processor to take its PID modulo $\frac{N}{s}$ to assure a uniform initial assignment of at least $\lfloor P/\frac{N}{s} \rfloor$ and no more than $\lceil P/\frac{N}{s} \rceil$ processors to a work element.

Algorithm W performs best when the cluster size s is chosen to be $\log N$, resulting in a progress tree of $N/\log N$ leaves. A complete description of the algorithm can be found in [18]. Using the above lemma for $U = N/\log N$, a tight bound can be obtained for algorithm W[$\log N$]:

Theorem 2.2 ([18, 24]). *Algorithm* W[$\log N$] *is a robust parallel algorithm for the Write-All problem 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.* □

2.3.2. PRAM Simulations

The original motivation for studying the Write-All problem was that it intuitively captures the essential nature of a single synchronous PRAM step. This intuition was made concrete when it was shown in [20, 31] how to use solutions for the Write-All problem in implementing general PRAM simulations.

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

Fig. 4. A high level view of algorithm Z.

A high level view of this approach is given in Fig. 3. The simulations are implemented by robustly executing each of the cycles of the PRAM step: instruction fetch, read cycle, compute cycle, write cycle, 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 Write-All style, e.g., using algorithm W (for details, see [20, 31]).

Using such algorithm simulation techniques it was shown in [20, 31] that if $S_w(N, P)$ denotes the available steps of solving a Write-All instance of size N using P processors, and if a linear in N 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 available steps of the fault-tolerant simulation will be $O(\tau \cdot S_w(N, P))$.

2.3.3. Static Initial Memory Errors

The Write-All algorithms and simulations, e.g., [18, 20, 22, 31], or the algorithms that can serve as Write-All solutions, e.g., the algorithms in [8, 26], 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 "using up" the clear memory, and concurrently or subsequently clearing segments of memory needed for future iterations. An efficient Write-All solution that requires no clear shared memory has recently been defined using a *bootstrap* approach [32].

The bootstrapping proceeds in stages: 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 $i-1$. If $N_{i+1} > N_i$ and $N_0 \geq 1$, then the required N memory location will be cleared in at most N stages.

The efficiency of the resulting algorithm depends on the particular Write-All solution(s) used and the parameters N_i and G_i. In [32], a solution for Write-All (algorithm Z, see Fig. 4) is presented that for any failure pattern F ($|F| < P$) has work $O(N + P \log^3 N/(\log\log N)^2)$ *without any initialization assumptions.*

This solution uses algorithm W[$\log N$] as the underlying Write-All algorithm. In the early iterations, the number P of processors may be greater than the number N_i of leaves of the enumeration tree. In this case processors are assigned to the leaves by taking their PIDs mod N_i, as described in Section 2.3.1.

3. Controlling Memory Access Concurrency

Algorithm W is a robust fail-stop Write-All solution, however it makes liberal use of concurrent memory accesses *even when there are no faults*. When $P = N$, in phase W4 (Fig. 2, line 03), the algorithm has $\omega = \sum_{t=1}^{\log N+1} \left(N - \frac{N}{2^{t-1}}\right) = \Omega(N \log N)$ in the absence of failures. The same problem is shared by all robust deterministic Write-All algorithms that have been proposed to date.

As shown in [18], concurrent memory writes cannot be eliminated altogether. Any Write-All algorithm that does not use concurrent writes, e.g. CREW PRAM, has worst case work $\Omega(N^2)$ and thus is not robust.

3.1. Algorithm W with Controlled Write Concurrency

In this section we present a version of algorithm W that controls the overall number of concurrent accesses. For ease of presentation we will first develop methods to control the write concurrency while allowing unlimited concurrent reads. In the next section we will extend these techniques to control the read concurrency as well.

Assume there are $p \le P$ processors that want to write to a common memory location T. Our technique for controlling the write concurrency is based on organizing the p processors into a *processor priority tree* (PPT).

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. Thus these trees are structurally similar to heaps. 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. Note that these processor numbers need not be the same as their permanent PIDs. *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.

It is clear that whereas unrestricted concurrent writes by $p \le P$ processors take only one step to update T, a PPT requires as many steps as there are levels in the tree, i.e., $O(\log p)$. Provided there is at least one live processor in the tree the correct value will eventually be written. Although CW introduces several idle processor cycles, it allows us to bound the number of concurrent writes incurred during an update of T.

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

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

Lemma 3.1. *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 $|F|$.*

Proof. By the description of CW above it is clear that location T is updated by the processors of at most one level, say i. In this case all processors on levels $0, \ldots, i-1$ must have failed. Since level j has 2^j processors the total number of processors on these levels is $\sum_{j=0}^{i-1} 2^j = 2^i - 1$ and hence $|F| \geq 2^i - 1$. On the other hand, level i has at most 2^i live processors and hence $\omega \leq 2^i - 1$. The lemma follows by combining the two inequalities. □

We can now modify algorithm W by incorporating PPTs into its four phases. This is done by replacing each concurrent write in these phases by an execution of CW. Since in the worst case all P processors can form a single PPT, 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. (Although this will be sufficient for our analysis, in practice we could use the overestimate of surviving processors computed in phase W1 to find the height of the largest possible PPT for the current iteration and use this to determine how much time to spend in CW.) 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 (if there is one). 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 3.1 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 during 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 [18]). 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

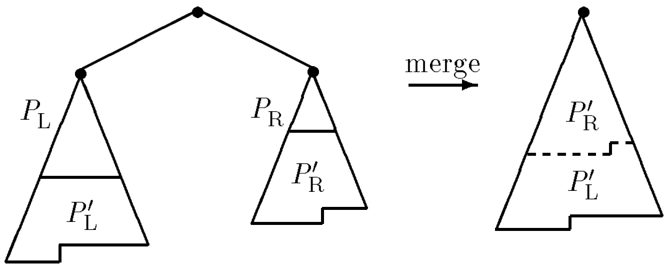

Fig. 6. Merging and compacting two PPTs. There are P_L and P_R processors in the left and right PPTs, respectively. At the end of algorithm CW, up to P'_L and P'_R processors remained in the PPTs. The resulting PPT will contain the surviving $P'_L + P'_R$ processors.

and the level index.

After the $\log P$ steps of CW, the processors of each of the two PPTs that are to be merged concurrently read 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.

Using the size and the level index each processor can locally compute its position in the merged tree. If s_1 and s_2 are the sizes of the left and right PPTs respectively, and l_1 and l_2 are the corresponding level indices, the merged tree will have size $s_1 + s_2 - 2^{l_1} - 2^{l_2} + 2$ and the first $s_2 - 2^{l_2} + 1$ nodes will be occupied by the processors from the right PPT. This process is illustrated in Fig. 6.

We refer to the algorithm that results from W by incorporating CW and PPT compaction and merging as algorithm $\mathrm{W_{CW}}$. In more detail, the four phases of algorithm W need to be modified as follows:

W1: At the beginning of this phase each processor forms a PPT by itself. Processors traverse the enumeration tree bottom up storing at each internal node of the tree a value based on the values stored at its two children. In addition, in the original algorithm W a timestamp is written to distinguish new from old values. The required modifications entail using algorithm CW for writing to the nodes of the enumeration tree and dispensing with the timestamp; the latter is because all values written by CW are already timestamped in order to facilitate PPT merging as discussed above. After each call of CW all PPTs are compacted and merged.

W2: This phase can be implemented without concurrent writes and the only modification is that processors need to keep track of the size of the PPT they belong to and their position within this PPT since this information is needed in the subsequent phases. As processors traverse the progress tree top-down they move to the left or right child of the current node in proportion to the number of unvisited leaves in the left and right subtrees, as in the original algorithm W. This causes each PPT to be split in two at each step. If a PPT has k processors of which k' need to go left and the remaining $k - k'$ need to go right, then by convention the first k' processors of the PPT are allocated to the PPT that moves to the left child and the remaining $k - k'$ processors form the PPT that moves to the right child.

W3: Processors use the information they gathered during phase W2 to organize themselves into PPTs and then proceed to write 1 to the input elements associated with the leaf they reached at the end of W2. Each such write uses algorithm CW and is followed by compaction to eliminate the dead processors. There is no PPT merging in this phase. As a result there is no need for timestamping either.

W4: In this phase processors traverse the progress tree bottom up writing at each node the sum of the values stored at its two children. This write is accomplished by calling CW and is followed by compaction and merging. The PPTs used at the beginning of this phase are those left at the end of W3.

For the case where each leaf of the progress tree is associated with one element of the input array we can obtain the following bound on the performance of $\mathrm{W_{CW}}$:

Lemma 3.2. *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* $1 \leq P \leq N$.

Proof. The correctness of the algorithm follows from the correctness of algorithms W and CW and from the fact that we just replace each write step in W by a call to CW.

To show the bound on S we invoke Lemma 2.1 according to which the total number of block steps of algorithm W is at most $U + P \log U / \log\log U$, where U is the initial number of unvisited leaves. The number of available processor steps is $(U + P \log U / \log\log U)(t_t + t_l)$, where t_t is the time needed to traverse the processor enumeration and progress trees in phases W1, W2, and W4, and t_l is the time spent at the leaves during phase W3.

In our case, $t_t = O(\log N \log P)$ since each execution of CW takes $O(\log P)$ time and the height of the trees traversed is $O(\log N)$, and $t_l = O(\log P)$, the time of a single execution of CW. Thus, with $U = N$

$$\begin{aligned} S &= (N + P \log N / \log\log N)(O(\log N \log P) + O(\log P)) \\ &= O(N \log N \log P + P \log P \log^2 N / \log\log N). \end{aligned}$$

To show the bound on ω, we first note that contributions to ω occur only during the executions of CW since there are no other concurrent writes. If there were m executions of CW, let ω_i denote the contribution of the ith such execution to ω, $1 \leq i \leq m$. Let $F_i \subseteq F$ be the failure pattern that consists of the failures of the processors above the PPT level that effected the write or of all the processors of the PPT if all failed during the ith execution. From the proof of Lemma 3.1, $\omega_i \leq |F_i|$. Due to the compaction of PPTs all processors named in F_i are eliminated at the end of CW and hence can cause at most one concurrent write (contribute at most once to ω). By summing the contributions from all executions of CW we conclude that the overall write concurrency is $\omega = \sum_{i=1}^{m} \omega_i \leq \sum_{i=1}^{m} |F_i| \leq |F|$.

Note that failed processors that are below the level that completed the write during CW are not removed by compaction but they do not cause any concurrent writes. □

3.2. Algorithm W with Controlled Read/Write Concurrency

In this subsection we present extensions of algorithm CW that control read concurrency in addition to write concurrency. Our approach is based on replacing concurrent reads by broadcasts through the PPTs. We first describe a simple scheme and then give a more elaborate and efficient method.

As before assume there are $p \leq P$ processors that need to write to a common memory location T. Algorithm CW allows all the processors of a PPT level to concurrently read T to check if it has been updated. Our first scheme for controlling read concurrency replaces this concurrent read by a broadcast through the processors of the PPT level. Specifically, we view the processors of each level as forming a heap-like binary tree where the leftmost processor is the root, the second and third processors are the root's children and so on.

The broadcast through each such tree proceeds in a top-down fashion by having each processor broadcast the value it read to its two children, starting with the root. If a processor does not receive a value because its parent has failed, then it reads T directly. If the sibling processor is alive it will also read T thus incurring a concurrent read. The rest of the algorithm is as in CW. The tree formed by the processors of the ith PPT level has $i+1$ levels of its own so $i+1$ steps are required for the ith PPT level ($i \leq \lceil \log p \rceil$). This means that algorithm CW modified as described here requires $\max\{i+1\} \log P = O(\log^2 P)$ steps.

Simple as this method may be it is not satisfactory as it does not provide good bounds on the read concurrency. The problem stems from the fact that a failed processor may remain in a PPT for several steps and cause concurrent reads during each of these steps.

We next present a more elaborate scheme that addresses this problem. The advantages of this algorithm over the one described above are

- it uses several PPT levels at a time to broadcast instead of just one, and
- it tries to move failed processors towards the leaves of the PPT.

Even though a failed processor can still remain in a PPT for several steps, its movement towards the leaves means that it can cause concurrent reads for only a small number of steps, as we will show below. This algorithm will allow us to control read concurrency without degrading the performance of algorithm $\mathrm{W_{CW}}$. 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.

Algorithm CR/W: A high level view of this algorithm, that we call CR/W, is given in Fig. 7. Communication 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

01 $--$*Code executed by each of the p processors in the PPT*

02 Initialize $B[ID]$ $--$*broadcast location for processor ID*

03 **for** $i = 0 \ldots \lfloor \log p \rfloor$ **do** $--$*for each level i of the PPT*

04   **if** $2^i \leq ID < 2^{i+1}$ **then** $--$*if the processor is at level i*

05   READ $B[ID]$

06   **if** $B[ID]$ has not been updated **then** $--$*note that this is always true if* $ID = 2^{i+1} - 1$

07   READ $(value, level)$ from T

08   **if** T contains some old value **then**

09         $level := i$ $--$*level i effects the write*

10   WRITE $(newvalue, level)$ to T

11   **else if** $2^{level} \leq ID - 2^i + 1 < 2^i$ **then** $--$*unless it is above the level that effects the write*

12         $ID := ID - 2^i + 1$ $--$*take the place of the processor that was to update* $B[ID]$

13   **fi fi fi fi**

14   **if** $ID < 2^{i+1}$ **and** $ID + 2^{i+1} \leq p$ **then** $--$*if the processor's level is* $\leq i$

15   WRITE *newvalue* to $B[ID + 2^{i+1}]$ $--$*broadcast to appropriate processor at level* $i+1$

16   **fi**

17 **od**

Fig. 7. High level view of algorithm CR/W for the controlled read/write. This code is executed by each of the $p \leq P$ processors in the PPT. Here ID is the position of the processor in the PPT and *newvalue* is the value to be written. Processors read and write pairs that include T's value as well as the index of the PPT level that effects the write.

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. For all the PPTs, we can store these B arrays in a segment of memory of size P; we discuss an implementation later on in this section.

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 (for simplicity we leave out the timestamps and the size of the tree).

```
01 --Code executed by each of the p processors in the PPT
02 Initialize B[ID] --broadcast location for processor ID
03 for i = 0 ... ⌊log p⌋ do --for each level i of the PPT
04   if 2^i ≤ ID < 2^{i+1} then --if the processor is at level i
05     READ B[ID]
06     if B[ID] has not been updated then --note that this is always true if ID = 2^{i+1} − 1
07       READ T
08       if ID ≠ 2^{i+1} − 1 then --unless this is the rightmost processor
09         ID := ID − 2^i + 1 --take the place of the processor that was to update B[ID]
10   fi fi fi
11   if ID < 2^{i+1} and ID + 2^{i+1} ≤ p then --if the processor's level is ≤ i
12     WRITE value read to B[ID + 2^{i+1}] --broadcast to appropriate processor at level i + 1
13   fi
14 od
```

Fig. 8. High level view of algorithm CR1 for the controlled read. The code is executed by each of the $p \leq P$ processors in the PPT. Here ID is the position of the processor in the PPT, which may change during the execution of the algorithm.

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: Algorithm CR/W combines a read with a write (i.e., it incorporates the functionality of the old CW). 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 is similar to CR/W but without the write step. This algorithm, referred to as CR1, is presented in Fig 8. It is simpler than CR/W since processors that are found to have failed are pushed towards the bottom of the PPT independent of the level they are at. This algorithm will be 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 $i + 1$. Unlike CR1 no processor movement takes place. This algorithm will be used for top-down traversals during W.

It is easy to see from the description of the algorithms that all three of CR/W, CR1, and CR2 require $O(\log P)$ time for PPTs of at most P processors.

Algorithm $\mathrm{W}_{\mathrm{CR/W}}$**:** We are now ready to present a Write-All algorithm that controls both read and write concurrency. It utilizes algorithms CR/W, CR1, and CR2 for memory accesses as well as the techniques of PPT merging and compaction introduced in the previous section. As with W_{CW}, we augment the values written by CR/W with timestamps to facilitate PPT merging and also write the PPT size to assist with compaction (for simplicity, these two additional pieces of information are not shown in Fig. 7). The resulting algorithm will be referred to as $W_{CR/W}$. In more detail, the following modifications must be made to the four phases of the original algorithm W:

W1: Processors begin this phase by forming single-processor PPTs. The objective is for processors to write to each internal node of the enumeration tree the sum of the values stored at its two children. For this they use algorithm CR/W to store the new value, the size of the PPT, the index of the level that completed the write, and a timestamp. As with W_{CW} the timestamps written in algorithm W are not required.

Upon completion of CR/W all PPTs are compacted. In order to merge PPTs the processors of each need to read the data stored at the enumeration tree node that is the sibling of the node they just updated. For this, algorithm CR1 is used and after it finishes PPTs are merged. At this point the processors of the merged PPTs know the value they need to write at the next level of the enumeration tree. This value is just the sum of the value written by CR/W and the value read by CR1. Therefore one call to each of CR/W and CR1 is needed for each level of the enumeration tree.

W2: This phase involves no concurrent writes and can be implemented using mostly local computations. Surviving processors traverse top-down the progress tree to allocate themselves to the unvisited leaves. The only global information needed at each level is the values stored at the two children of the current node of the progress tree. These values are read using two calls to CR2, one for each child. Using this information the processors of a PPT compute locally whether they need to go left or right. No compaction or merging is done in this phase. As PPTs move downwards they are split as described in phase W2 of algorithm W_{CW}.

W3: Processors organize themselves into PPTs based on the information they gathered during the previous phase and proceed to write 1 to the locations that correspond to the leaf they reached. Each of these writes uses algorithm CR/W and is followed by compaction. No merging is involved.

W4: This phase initially uses the PPTs that resulted at the end of W3. The task to be performed is similar to that of W1. As before CR/W is used for writing followed by compaction and then one call to CR1, after which PPTs are merged.

Before embarking on the analysis of this algorithm, we discuss briefly how to accomodate the B arrays of all the PPTs in $O(P)$ space. We use an array $\mathcal{B}$ of length P and implement the B array of each PPT as a contiguous subarray of $\mathcal{B}$. Each PPT maintains the starting location of this subarray, whose length is just the size of the PPT.

Initially each PPT has a single processor and processor P_i is allocated location $\mathcal{B}[i]$. When two PPTs are merged we need to combine their $\mathcal{B}$ subarrays. If the left PPT uses a subarray of size s_l that starts at $\mathcal{B}[t_l]$ and the corresponding parameters for the right PPT are s_r and $\mathcal{B}[t_r]$, then the merged PPT will use a subarray of size $s_l + s_r$ starting at $\mathcal{B}[t_l]$. The only information that needs to be communicated between the two PPTs is that the right PPT must know t_l. This is accomplished by having the processors of the left PPT write t_l along with the new value for T, the timestamp, the PPT size and the level index. Note that the subarrays used by the two PPTs that are merged may not be contiguous in $\mathcal{B}$ but in this case the space between the two subarrays is unused (i.e., there may be a gap between them) since the two PPTs are siblings.

When PPTs are split during W2 their $\mathcal{B}$ subarrays are also split. If a PPT of size s that uses a subarray starting at $\mathcal{B}[t]$ is split into a left PPT of size s' and a right PPT of size $s - s'$, then the subarray for the left PPT will start at $\mathcal{B}[t]$ and the one for the right PPT will start at $\mathcal{B}[t + s']$. Only local computations are involved here.

Analysis of algorithm $\mathrm{W}_{\mathrm{CR/W}}$**:** Although we can guarantee a write concurrency of at most $|F|$ for algorithm $\mathrm{W}_{\mathrm{CR/W}}$, the read concurrency is slightly more. This is because PPT compaction does not eliminate all the failed processors from a PPT but only those above the level that completed the write. Failed processors below this level remain in the PPT and although they do not cause any concurrent writes they may cause concurrent reads.

In order to account for the concurrent reads incurred we associate a potential with each processor. Our intention is to make the overall potential an upper bound on ρ. Below we explain how this is done.

Let Φ_i denote the potential of the the ith processor, $1 \leq i \leq P$, let $\Phi_{PPT} = \sum_{P_i \in \mathrm{PPT}} \Phi_i$ denote the potential of a PPT, and let $\Phi = \sum_{1 \leq i \leq P} \Phi_i$ be the overall potential. We use $\Delta\Phi_i$ to denote the change in the potential of the ith processor during an operation or sequence of operations. Initially $\Phi_i = 0$, $1 \leq i \leq P$.

During each operation we increase the potential of a failed processor by an amount that is an upper bound on the number of concurrent reads that can potentially be incurred during this operation due to the failure of this processor. As a result Φ will be an upper bound on the total number of concurrent reads incurred during an execution of the algorithm, i.e., $\rho \leq \Phi$. The following lemma describes the required increase in the potential of a failed processor during an execution of each of the algorithms CR/W, CR1 and CR2.

Lemma 3.3. *Let l_1 be the PPT level of a processor at the beginning of an execution of any of the algorithms CR1, CR2, or CR/W and let l_2 be the PPT level of the same processor at the end of this execution. If this processor is dead at the end of this execution, then its potential should be increased by (i) $l_2 - l_1$ during CR1, (ii) 2 during CR2, and (iii) $l_2 - l_1$ if it is at or below the level that completes the write and* $\log P - l_1$ *if it is above this level during CR/W.*

Proof. We need to show that the proposed increases in the potential of a failed processor are upper bounds on the number of concurrent reads that can be incurred due to its failure. (i) From the description of the algorithm it is clear that concurrent reads are the result of processors discovering that processors at higher PPT levels have failed and could not broadcast as a result. In this case the failed processors move to lower PPT levels as the processors that detect the failures take their position in the PPT. Since a failed processor

moves by at least one level for each concurrent read it causes, the number of concurrent reads caused by a failed processor that moves from level l_1 to level l_2 can be at most $l_2 - l_1$.
(ii) Since in CR2 a processor is responsible only for broadcasting to its two children, a failed processor can cause at most two concurrent reads. This happens if both of its children are alive and other processors at the same PPT level are also accessing the same location as a result of their parents also having failed.
(iii) Let L be the index of the PPT level that completed the write. If a processor is above L then no processor will take its position in the PPT. Since the PPT can have up to $\log P$ levels this processor can cause a concurrent read at each of the levels below its own resulting in $\log P - l_1$ concurrent reads. If the processor is at or below L then the situation is similar to that of case (i) and by the same argument the processor will incur $l_2 - l_1$ concurrent reads. $\square$

Because of PPT compaction it is possible for some failed processor to actually move to a higher PPT level after an execution of CR/W (note that compactions occur only after CR/W). To account for this we charge each processor that is eliminated by compaction for $\log P$ concurrent reads. As the following lemma shows, this one time charge is sufficient to account for all the concurrent reads that can be caused by failed processors that move to higher levels following a compaction.

Lemma 3.4. *If each processor that is eliminated by compacting a PPT is charged* $\log P$ *and each processor that moves to a higher level as a result of the compaction is credited for the number of levels it moves, then the potential* Φ_{PPT} *of the PPT does not increase and neither does* Φ.

Proof. Let L be the PPT level that completes the write. As a result of compaction $2^L - 1$ processors will be eliminated. Increasing the potential of each such processor by $\log P$ increases the potential of the PPT by $(2^L - 1)\log P$.

The absence of these $2^L - 1$ processors from the compacted PPT means that some processors at levels $\geq L$ move up. Since $2^L - 1$ processors are eliminated, the first $2^L - 1$ processors of each level $\geq L$ will move up while all other processors will move left within their levels but will not change level. We decrease the potential of each processor that moves up by the number of levels it moves.

The $2^L - 1$ processors of level L that will move may move by as many as L levels, so the decrease in their potential is at most L (this is a pessimistic estimate). The processors of the other levels that will move may move by just one level, so their potential is decreased by 1.

The total decrease in potential for all processors that move to higher levels is at most

$$(2^L - 1)L + \sum_{i=L+1}^{\log P} (2^L - 1) = (2^L - 1)\log P,$$

equal to the increase in potential. Thus potential redistribution does not increase Φ_{PPT} (since we are taking a pessimistic approach, Φ_{PPT} may actually decrease due to this redistribution). As a consequence Φ does not increase either. $\square$

The remaining operation involved in $W_{CR/W}$, PPT merging, can never cause a processor to move to a higher level. As a result we neither increase nor decrease processor potentials during this operation, so we ignore merging in the rest of our analysis.

During phases W1 and W4 the main operation at each level of the enumeration or progress tree entails one call to each of CR/W, PPT compaction, and CR1 in this order. Phase W3 is similar but without CR1 (phase W2 is quite different). The following lemma provides a bound on the change of potential during a sequence of such operations.

Lemma 3.5. *Let s be the length of a sequence of steps each of which consists of a call to CR/W, PPT compaction, and CR1 in this order and let P_i be a processor that is dead at the end of this sequence but is not eliminated by compaction during the sequence. If P_i is at PPT level l_0 at the beginning of the sequence and at level l_j after the jth step of the sequence ($1 \le j \le s$), then $\Delta\Phi_i = l_s - l_0$ over the entire sequence.*

Proof. Consider the jth step of the sequence. P_i is at level l_{j-1} at the beginning of this step and at level l_j at the end of it. If P_i is at level $l'_j \ge l_{j-1}$ after the call to CR/W and at level $l''_j \le l'_j$ after compaction, then Φ_i increases by $l'_j - l_{j-1} + l_j - l''_j$ (Lemma 3.3) and decreases by $l'_j - l''_j$ (Lemma 3.4). Thus $\Delta\Phi_i = l_j - l_{j-1}$ for the jth step and for the entire sequence we obtain

$$\Delta\Phi_i = \sum_{j=1}^{s}(l_j - l_{j-1}) = l_s - l_0.$$

This shows that the change in potential depends only on the starting and ending PPT levels and not on the length of the sequence. □

As in the case of algorithm W, we may cluster several elements of the input array and assign them to a single leaf of the progress tree. We use the notation $W_{CR/W}[s]$ to indicate that input elements are organized into clusters of size s. For the case $s = 1$ we use the shorthand $W_{CR/W}$ for $W_{CR/W}[1]$. The following theorem provides bounds on the performance of this special case:

Theorem 3.6. *Algorithm $W_{CR/W}$ 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 $\rho \le 7\,|F| \log N$, and write concurrency $\omega \le |F|$, where $1 \le P \le N$.*

Proof. The bounds on S and ω are obtained as in Lemma 3.2. To show the bound on ρ we consider the change $\Delta\Phi_i$ of the potential of a failed processor throughout a block step (an iteration of the while loop). It suffices to focus on a single block step since PPTs are rebuilt at the beginning of phase W1 and dead processors are eliminated. As a result, a failed processor can cause concurrent reads for at most one block step.

Suppose that a processor is eliminated by compaction when it is at level l of a PPT. It is clear that the processor causes the maximum number of concurrent reads when this elimination takes place in phase W4 while the processor has failed since phase W1.

When it is eliminated its potential will increase by $2 \log P - l$, where $\log P$ is due to the potential redistribution as described in Lemma 3.4, and $\log P - l$ is due to the concurrent reads it may cause in the last execution of CR/W right before the compaction that eliminates it (Lemma 3.3, (iii)).

The concurrent reads it causes until its elimination are accounted for as follows. In phase W1 the sequence CR/W, PPT compaction, CR1, is executed once for each level of the enumeration tree. If the processor is at PPT level l_s at the beginning of the phase and at level l_f at the end, then by Lemma 3.5 the change in potential is $l_f - l_s \leq \log P$ since the height of the PPT is at most $\log P$.

Since the PPTs that exist at the end of W3 are the same as those used at the beginning of W4, as far as accounting goes these two phases can be considered as being one. Consider the sequence of calls to CR/W, compaction, and CR1 upto the call to CR/W that immediately precedes the compaction that eliminates this processor. The processor is at level l at the end of this sequence. If it is at level l' at the beginning of W3, then by Lemma 3.5 the change in potential is $l - l' \leq l$.

Finally, the change in potential for phase W2 is 2 for each execution of CR2 and there are two such executions for each level of the progress tree. Thus the change in potential is 4 for each level of the progress tree and the total change in potential for phase W2 is $4 \log N$.

Combining the above we have that $\Delta\Phi_i$ for an entire block step is bounded by

$$\log P + 4 \log N + l + 2 \log P - l \leq 7 \log N.$$

Thus the amortized number of concurrent reads charged to a failed processor is at most $7 \log N$. Since the potential of alive processors is non-positive we obtain the following bound for ρ

$$\rho \leq \Phi = \sum_{1 \leq i \leq P} \Phi_i \leq \sum_{P_i \text{ failed}} \Phi_i \leq 7|F| \log N.$$

□

3.3. An Optimal Algorithm

As discussed in Section 2.3.1, algorithm W becomes optimal if input elements are clustered into groups of size $\log N$, each group associated with one leaf of the progress tree, and a number of processors smaller than N is deployed. In a similar vein, algorithm $\mathrm{W}_{\mathrm{CR/W}}$ performs best if clusters of size larger than one are used and fewer than N processors are employed.

Since the enumeration and progress trees traversed have height $O(\log N)$ and algorithm CR/W requires $O(\log P)$ time, phases W1, W2, and W4 require $O(\log N \log P)$ time. Thus, we cluster the input elements into groups of size $\log N \log P$ and have a progress tree with $N/\log N \log P$ leaves. The resulting algorithm $\mathrm{W}_{\mathrm{CR/W}}[\log N \log P]$ turns out to be non-optimal, so a further modification is needed.

To obtain optimality we make use of the perfect allocation property of algorithm W (see [18]). This property guarantees that available processors are allocated evenly to the unvisited leaves in phase W2. Let U_i and P_i be the numbers of unvisited leaves and available processors, respectively, at the beginning of the ith iteration of the while loop of algorithm

$\mathrm{W}_{\mathrm{CR/W}}$. The balanced allocation property implies that if $U_i \geq P_i$, then at most one processor will be allocated to each unvisited leaf.

This leads to a modification of algorithm $\mathrm{W}_{\mathrm{CR/W}}[\log N \log P]$. The difference in this algorithm is that processors examine the overestimate of remaining processors P_i and the overestimate of the number of unvisited leaves U_i at the beginning of each iteration. If $U_i \geq P_i$ we are guaranteed that there will be at most one processor per leaf at the beginning of phase W3 so that there is no need to use CR/W and compaction. Instead processors go sequentially through the $\log N \log P$ elements of the leaf they reached spending $O(1)$ time for each element. If, on the other hand, $U_i < P_i$ several processors may be allocated to the same leaf and algorithm CR/W needs to be used. The latter situation occurs infrequently enough that the algorithm becomes optimal for a certain range of processors, as we show below.

We refer to algorithm $\mathrm{W}_{\mathrm{CR/W}}[\log N \log P]$ with the above modification as algorithm $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$. The following theorem provides bounds on its performance.

Theorem 3.7. *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* $1 \leq P \leq N$.

Proof. Let P_i be the (overestimate of the) number of live processors and U_i be the (overestimate of the) number of unvisited leaves at the begining of the ith iteration. We divide the iterations into two cases:

Case 1: The number of unvisited leaves is $U_i \geq P_i$. In this case phase W3 requires $O(\log N \log P)$ time, the same as the time to traverse the enumeration and progress trees. By Lemma 2.1 the number of block steps for all such iterations is at most U_0, where U_0 is the initial number of leaves. In this case $U_0 = N/\log N \log P$ and the total work for all such iterations is

$$S_1 = \frac{N}{\log N \log P}\, O(\log N \log P) = O(N).$$

Case 2: The number of unvisited leaves is $U_i < P_i$. In this case all the processors could be allocated to the same leaf. Thus, in the worst case, $\log P$ time must be spent at each element of the leaf and since there are $\log N \log P$ elements per leaf the worst case time to update a leaf is $O(\log N \log^2 P)$. By Lemma 2.1 the number of block steps for all such iterations is at most $O(P\frac{\log N}{\log\log N})$. Hence the total work of all these iterations is

$$\begin{aligned} S_2 &= O(P\frac{\log N}{\log\log N})\,(O(\log N \log P) + O(\log N \log^2 P)) \\ &= O(P\frac{\log^2 N \log^2 P}{\log\log N}). \end{aligned}$$

In addition, we must account for the work performed by processors that fail in the middle of an iteration. There are at most $O(P)$ such block steps each costing at most

$O(\log N \log^2 P)$ so that the total work in this case is $S_0 = O(P \log N \log^2 P)$. Combining the three cases, we have

$$S = S_0 + S_1 + S_2 = O(N + P\frac{\log^2 N \log^2 P}{\log \log N}).$$

The algorithm is optimal if the second term is $O(N)$.

The bound on ω is obtained as before by noticing that whenever a PPT has more than one processor, the tree is compacted after the write to eliminate any processors that have caused concurrent writes. The bound on ρ is obtained as in Theorem 3.6. □

3.4. Handling Unitialized Memory

The algorithms considered so far require that memory be initially clear. Algorithm Z of [32] extends algorithm W to handle unitialized memory (see Section 2.3.3). It is possible to incorporate CR/W into Z to obtain a Write-All algorithm with limited concurrency that dispenses with the requirement that memory be clear.

A problem that arises if we try to combine the two algorithms directly is that we can no longer guarantee good read and write concurrency. This stems from the fact that to achieve these bounds the PPTs that are built at the beginning of phase W1 must contain no processors that are known to have failed. This is easy to do in the algorithms considered so far since each PPT starts out with at most one processor. In algorithm Z, however, there can be several processors that are assigned to the same leaf of the enumeration tree and thus belong to the same PPT at the begining of W1; this happens whenever $P > N_i$. The assignment of processors to such leaves is made based on the processor PIDs and since this does not take into account failed processors, the resulting PPTs may have dead processors that have already been accounted for in ω.

To work around this we will use an enumeration tree with P leaves for all iterations of Z, so that each PPT will have at most one processor at the beginning of W1. To initialize this tree we note that phase W1 is a static phase in the sense that a given processor traverses the same path through the enumeration tree every time W1 is executed. In particular, since processors can not restart once they have failed, if a node of the enumeration tree is accessed at all, it must be accessed the first time W1 is executed. The consequence of this is that we do not need to clear the entire enumeration tree but only that portion that can be accessed during the first execution of W1. To achieve this, processors make an initial bottom-up traversal of the enumeration tree similar to the traversal of W1 (i.e., each processor traverses the same path it would traverse in W1) and clear the locations that they would access in W1. This process is carried out as follows.

Processors start at the leaves of the enumeration tree, with the ith processor at the ith leaf, so that each processor forms a PPT by itself. PPTs then traverse the tree bottom-up taking the same paths as in phase W1. The operation performed by the PPTs at each node they reach consists of clearing the node they are at using CR/W, PPT compaction, clearing the sibling node using CR/W, PPT compaction, reading the contents of the node they are at using CR1, and finally PPT merging. The reason for clearing the sibling node is that during phase W1 processors need to access the node they are at and its sibling and we need to make certain that this sibling will not contain garbage data. The reason for the remaining operations is to assist with merging. Essentially, this initialization phase is like W1 but processors clear the sibling nodes instead of summing. The cost of this initialization is $O(\log N \log P)$ since the enumeration tree is of height $O(\log N)$ and the time spent at each level is $O(\log P)$.

The algorithm that results from Z by using a P-leaf enumeration tree initialized as described above and utilizing algorithm $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ to clear progressively larger areas of memory is called $\mathrm{Z}_{\mathrm{CR/W}}$ and has the performance bounds given in the following theorem:

Theorem 3.8. *Algorithm $\mathrm{Z}_{\mathrm{CR/W}}$ with contaminated memory of size $O(P)$ is a robust Write-All algorithm with $S = O(N + P \log^2 P \log^3 N/(\log\log N)^2)$, read concurrency $\rho \leq 7\,|F| \log N$, and write concurrency $\omega \leq |F|$, where $1 \leq P \leq N$.*

Proof. Since $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ uses clusters of size $\log N \log P$ we choose $G_i = \log N \log P, i \geq 0$ as the parameters for Z (see Fig. 4). Thus initially processors clear a memory area of size $G_0 = \log N \log P$ using algorithm CR/W; this costs $P \log N \log^2 P$. Following this there are $\log N/(\log(\log N \log P)) - 1 = O(\log N/ \log\log N)$ applications of algorithm $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ where the ith application clears a memory area of size $N_i = (\log N \log P)^{i+1}$. This analysis is very similar to the analysis of [32] and the work bound follows by the same method; the only difference is the different value for the G_i's and that the work of each iteration is given by Theorem 3.7. The bounds on ρ and ω are obtained as in the previous theorems in this section. □

3.5. A Lower Bound

Having shown a robust Write-All algorithm with $\omega \leq |F|$ it is interesting to consider whether there is a robust algorithm with even smaller ω. Here we show that this is not the case for a large class of functions of $|F|$. Specifically, we show that there is no robust algorithm for the Write-All problem with concurrency $\omega \leq |F|^{\varepsilon}$ for $0 \leq \varepsilon < 1$.

For this we consider a simpler *Write-One* problem [18] which involves P processors and a single memory location. The objective is to write 1 into the memory location subject to faults determined by an on-line adversary. The adversary will try to maximize the work while the algorithm will try to minimize it by appropriately scheduling the available processors subject to the constraint that only up to $|F|^{\varepsilon}$ concurrent writes are permitted.

We allow algorithms that can sense when the location has been written by employing "oracles" thus having no need to certify the write. Algorithms without such power can not perform better than oracle algorithms and hence the lower bound applies to them as well. This is because given a non-oracle algorithm A there is an oracle algorithm B that mimics A up to the point where A writes to memory. At this point B stops while A has to continue in order to certify the write thus incurring additional work.

Since not all processors can write concurrently the algorithm will need to form "waves" of processors. The adversary will fail each wave (but the last) right before the processors in the wave write to the memory. This way the adversary prolongs the conclusion of the process until the very last step while charging each processor as much as possible. As the following lemma shows, the algorithm's best answer to this strategy is to use a greedy approach, that is assign to each wave as many processors as possible within the constraints.

Lemma 3.9. *The work of Write-One algorithms is minimized when processors are allocated greedily.*

Proof. Let $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_k$ be the processor waves and let S be the total work of the algorithm. Assuming that the original schedule is not greedy, let $\mathcal{W}_j$, $1 \leq j < k$ be the first wave that contains fewer processors than are allowed by the $|F|^\varepsilon$ constraint. By assigning a processor from $\mathcal{W}_{j'}$ to $\mathcal{W}_j$ for some $j < j' \leq k$ we obtain a schedule with work $S' = S - (j' - j) < S$ and hence the original non-greedy schedule is not optimal. □

We can now show the following:

Lemma 3.10. *There is no robust algorithm for the Write-One problem with write concurrency $\omega \leq |F|^\varepsilon$, for $0 \leq \varepsilon < 1$.*

Proof. Since $|F| \leq P$, where P is the number of initial processors, it suffices to show that the bound holds for the broader class of *Write-One* algorithms that allow upto P^ε concurrent writes per step. By Lemma 3.9 P^ε processors should be allocated to each wave and hence there will be $P/P^\varepsilon = P^{1-\varepsilon}$ steps (for simplicity we assume that P is a multiple of P^ε). Then the work of the algorithm is $\sum_{i=1}^{P^{1-\varepsilon}} (P - iP^\varepsilon) + P^\varepsilon > \frac{P^{2-\varepsilon}}{2} - \frac{P}{2} = \Omega(P^{2-\varepsilon})$. □

We can now show that:

Theorem 3.11. *There is no robust algorithm for the Write-All problem with write concurrency $\omega \leq |F|^\varepsilon$, for $0 \leq \varepsilon < 1$.*

Proof. Consider a Write-All algorithm for instances of size N with P processors. The adversary picks one of the array locations and fails only processors that attempt to write to it according to the strategy outlined earlier in this section. According to Lemma 3.10 the work of the algorithm will be $\Omega(P^{2-\varepsilon})$. □

3.6. PRAM Simulations Subject to Dynamic Faults

As discussed in Section 2.3.2, solutions to the Write-All problem can serve as building blocks for efficent simulations of parallel algorithms on fail-stop PRAMs. The algorithms presented above likewise can be employed in fault-tolerant simulations.

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

Theorem 3.12. *Any N processor, τ parallel time EREW PRAM algorithm can be simulated on a fail-stop P-processor CRCW PRAM with 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 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].

Similar results can be derived for contaminated initial memory using algorithm Z and Theorem 3.4 to initialize an appropriate amount of auxiliary shared memory required by the simulation. There is an additive expense in work of $O(N + P \log^2 P \log^3 N/(\log \log N)^2)$.

Finally, if the faulty processors are restarted in the simulations so that they are re-synchronized at the beginning of the individual simulated PRAM steps then the simulations results apply as well.

4. Algorithm W with Bounded Maximum Access

For the algorithms developed in the previous section, the read and write concurrencies are bounded, but during some time steps the read and write concurrency might still be as high as $\Theta(P)$ for large $|F|$. In this section, we are going to employ a pipelining technique to limit the maximum per step concurrency of algorithm W.

As a first approach to limiting concurrency we note that phases W1, W2, and W4 of algorithm W involve traversal of trees of logarithmic depth with only one level of the trees being used at any given time. A solution that naturally suggests itself is to divide the processors into waves and have them move through the trees separately by introducing a time delay between the waves. If we choose the number of waves carefully it is possible to reduce the number of concurrent accesses without degrading the asymptotic work of the algorithm. In particular, since the trees are of height $\log N$ we can divide the processors into $\log N$ waves each of $P/\log N$ processors. During each of the phases W1, W2, and W4 we send the processors through the appropriate trees one wave at a time thus reducing the number of concurrent accesses to at most $P/\log N$ for any memory location. Since the values computed at the nodes of the heaps during each phase will be correct only after the last wave of processors finishes, waves that finish earlier must wait for the last wave to arrive before beginning the next phase. This introduces a $O(\log N)$ *additive* overhead per phase doubling the running time of each phase but does not affect the asymptotic efficiency of the algorithm.

For phases W2 and W4, it is not significant how we divide the processors into waves. However, phase W1 requires that processors be assigned to waves in a left to right fashion, i.e., the leftmost $\log N$ processors form the first wave, the next $\log N$ processors are in the second wave and so on. This is because phase W1 computes prefix sums and a processor must not reach a node of the processor enumeration tree before any processor with smaller PID.

This pipelining idea can be generalized by allowing a greater number of processor waves. In particular, for any $k \geq 1$ we can partition the processors into $\log^k N$ waves each of $P/\log^k N$ processors. Since there can only be $\log N$ active waves at any time, each phase will be slowed down by a factor of $\log^{k-1} N$. This yields the following:

Theorem 4.1. *Algorithm* $W[\log N]$ *with waves is a robust algorithm for the Write-All problem with* $S = O(N \log^{k-1} N + P \log^{k+1} N/\log\log N)$ *for any* $k \geq 1$ *and with at most* $P/\log^k N$ *concurrent accesses at any time for any memory location.*

Remark 4. For $k = 1$ there is no asymptotic degradation in the efficiency of the algorithm.

Even though the pipelining of processor waves reduces the number of concurrent accesses for each memory location, the algorithm still makes a lot of concurrent accesses overall. For example, for $k = 1$ and $P = N$ the algorithm makes $\Omega(\log N(\frac{N}{\log N} - 1)) = \Omega(N)$ concurrent writes when no processors fail since each wave makes $\frac{N}{\log N} - 1$ concurrent writes at the root and there are $\log N$ such waves.

The use of processor waves can be combined with the PPTs of Section 3.2 to yield an algorithm that uses *waves of priority trees* to control both the per step and the overall concurrency:

Theorem 4.2. *Algorithm* $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ *with pipelined PPTs is a robust Write-All algorithm with* $S = O(N \log^{k-1} N + P \log^2 P \log^{k+1} N/\log\log N)$ *for any* $k \geq 1$*, read concurrency* $\rho \leq 7\,|F| \log N$*, write concurrency* $\omega \leq |F|$ *and with at most* $P/\log^k N$ *concurrent accesses at any time for any memory location.*

5. Write-All with Static Faults

The failure model we have been considering so far allows arbitrary dynamic faults. As we have seen this model is too strong to allow efficient solutions without concurrent memory accesses. In this section we consider the alternative, weaker, model of static (initial) faults in which failures can only occur prior to the start of an algorithm. We will show that simple and efficient solutions are possible in this model without requiring concurrent accesses and without any need to initialize the shared memory.

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. We do not know P' a priori or which of the processors are faulty.

Our EREW algorithm E consists of two phases E1 and E2. In phase E1, processors enumerate themselves and compute the total number of live processors. With this information 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 (Fig. 9).

The enumeration phase E1 is similar to phase W1 of algorithm W that is based on the parallel prefix computation. However W1 cannot be used because of its concurrent accesses and the standard parallel prefix algorithm is not adequate since it is oblivious and requires that all processors be available. We present a simple non-oblivious EREW enumeration algorithm that does not assume a known number of processors and that produces correct results for any pattern of static failures.

A detailed description of phase E1 is given in Fig. 10. It uses a full binary tree of height $\log P$ for enumeration, denoted T in the figure. The processors begin with initial rank 1 at the leaves and move up toward the root. The algorithm ensures that at each interior node of the tree, at most one processor moves up. If two processors reach sibling nodes within the tree, one of them has to stop at the current level to avoid concurrent accesses at the parent. By convention, the processor that is allowed to continue is the one coming from the right.

Each processor P_i that continues needs to be able to communicate with the processors that have stopped at lower levels of the subtree rooted at the current location of P_i so that it can broadcast rank updates to them. To this end, the stopped processors in P_i's subtree form a tree whose root is the processor most recently stopped because it encountered P_i. Note that each processor in this tree stopped because it encountered either P_i or some other processor in the tree. P_i can broadcast a value to the processors in this tree by passing the value to the root of the tree and then having each processor pass the value to its children.

To do this, the algorithm of Fig. 10 assumes that each node of T has two fields: *val*, which is used for broadcasting, and *right_child*, which points to the root of the right subtree of stopped processors. We also use *left_child* to point to the left subtree. Note that *left_child* is a private variable, whereas *right_child* is a node field; this is because a processor P_i determines its left subtree by itself while its right subtree is determined by the processor that stops P_i.

As a processor moves up, it checks whether there is any processor at the sibling node by using a "probing" scheme. The processor writes 0 to its sibling node, then writes its current rank at its own node and then reads the sibling node (lines 7–9). If it is 0, then there is no processor at the sibling node, otherwise the value read is the number of processors in the subtree rooted at the sibling. As in the standard parallel prefix the processor that comes from the right adds this value to its rank (line 15). In our case, the right processor also initiates the broadcasting of this value to the processors in its tree of stopped processors by writing the value to the root of this tree (line 16). The trees of the two processors that meet are merged by creating a new tree with its root being the processor that came from

forall processors PID=1..P **parbegin**
    Phase E1: Use non-oblivious parallel prefix to compute $rank_{PID}$ and P'
    Phase E2: Set $x[(rank_{PID} - 1) \cdot \frac{N}{P'} \ldots (rank_{PID} \cdot \frac{N}{P'}) - 1]$ to 1
parend

Fig. 9. A high level view of algorithm E.

```
01 current_rank := 1  --each processor starts with initial rank 1
02 left_child := nil  --no left children yet
03 stopped := false  --initially all processors can move up
04 pos := P + PID − 2  --initial position in the tree
05 for i := 1..log P do
06   if stopped = false then  --processor can still move
07     T[sibling(pos)].val := 0 --probe sibling; sibling() returns the location of the sibling node
08     T[pos].val := current_rank
09     rank_sib := T[sibling(pos)].val
10     if rank_sib = 0 then  --noone there
11       pos := ⌊pos/2⌋  --move to parent node
12     else if current node is a left child then
13       stopped := true; T[pos].val := 0  --let the sibling processor continue
14     else  --current node is a right child; processor continues
15       current_rank := current_rank + rank_sib
16       T[left_child].val := rank_sib  --propagate rank update
17       T[sibling(pos)].right_child := left_child --modify trees
18       left_child := T[sibling(pos)]
19       pos := ⌊pos/2⌋  --move to parent node
20     fi
21   else  --processor has stopped; just propagate rank updates
22     if T[pos].val > 0 then  --there's a rank update to propagate
23       current_rank := current_rank + T[pos].val  --update own rank
24       T[T[pos].right_child].val := T[pos].val --pass the update on to the right child, if there is one
25       T[pos].val := 0  --get ready for the next iteration
26   fi fi
27 od
28 --complete propagating rank updates and broadcast the number of live processors
29 for i := 1..log P do
30   if stopped = false then  --this is the processor that reached the root
31     T[left_child].val := −current_rank
32   else  --stopped processors just propagate the values they receive
33     if T[pos].val > 0 then  --there's a rank update to propagate
34       current_rank := current_rank + T[pos].val  --update own rank
35       T[T[pos].right_child].val := T[pos].val --pass the update on to the right child, if there is one
36       T[pos].val := 0  --get ready for the next iteration
37     else if T[pos].val < 0 then  --this is the number of alive processors
38       T[left_child].val := T[T[pos].right_child].val := T[pos].val --propagate to both children
39       alive := −T[pos].val  --alive is the number of alive processors
40   fi fi
41 od
```

Fig. 10. A detailed description of phase E1.

the left, its left subtree the tree of the left processor and its right subtree the tree of the right processor (lines 17–18). An illustration of this process is provided in Fig. 11 when $P = 8$.

When the rightmost live processor gets to the root its rank is the number of live processors and this value is propagated through the tree down to the other processors (lines 29–41). In the algorithm of Fig. 10 we broadcast this value as a negative number to distinguish it

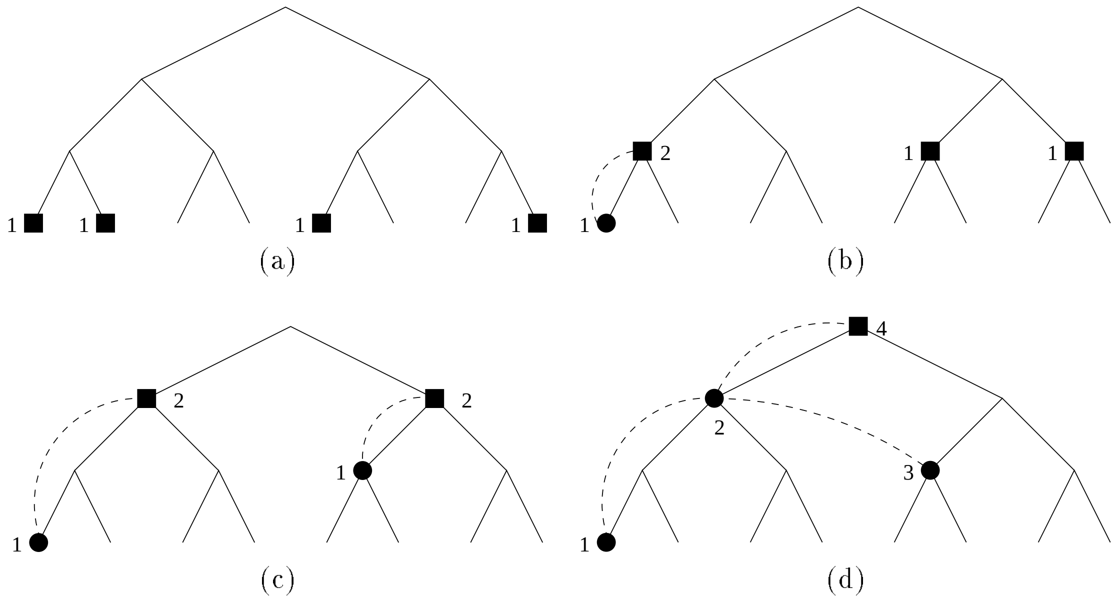

Fig. 11. An illustration of phase E1 with 8 processors of which 4 have failed. Circles indicate processors that have stopped their ascend because of encounters with other live processors and squares indicate processors that continue their upward movement. The numbers beside the processors indicate their ranks at each stage of the algorithm.

from the rank updates which are positive. The whole process takes $2 \log P$ steps.

The algorithm can be made to work in place by collapsing the levels of the tree into a single linear array. In addition, because of the probing scheme, the initial values of the tree are irrelevant and so the algorithm works even in the presence of initial memory faults.

In phase E2, knowing the number of live processors and their ranks, processors partition the input and each sets to 1 the entries in its own part.

Theorem 5.1. *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 [5] it can be shown that this algorithm is optimal.

Algorithm E can be used for simulations of PRAM algorithms on fail-stop PRAMs that are subject to static initial processor and memory faults. Simulations are much simpler for this case as compared to the dynamic failures case.

In particular, only the enumeration phase E1 of algorithm E is needed. At the beginning of the simulation we use the enumeration algorithm to determine the number P' of live processors. We then use Brent's Lemma [6] to simulate the P processors of the original algorithm on the P' available processors. The time overhead of the simulation in this case is just $O(\log P)$ for the enumeration and the work overhead is $O(P' \log P)$. Both overheads are *additive*. Thus we have the following:

Theorem 5.2. *Any P-processor, τ parallel time PRAM algorithm can be simulated on a fail-stop 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. The simulating PRAM is of the same type as the PRAM for which the simulated algorithm is written and the read and write concurrencies of the simulation are no more than those of the simulated algorithm.*

Note that when $P \cdot \tau = \Omega(P' \log P)$, the simulation is optimal.

6. Conclusions and Open Problems

We have shown that it is possible to obtain robust solutions to the Write-All problem that make concurrent accesses only subsequent to processor failures and then in a controlled fashion. We have further demonstrated that we cannot reduce the write concurrency much further.

An open question is whether we can improve the read concurrency by eliminating the logarithmic factor from ρ.

A second open problem is whether there is a faster algorithm than W for the Write-All and whether the methods presented here for controlling memory accesses can be applied to it.

In [22] an $\Omega(N \log N)$ lower bound was shown for the Write-All problem but no known deterministic algorithm attains it. A third open problem is whether the lower bound of [22] applies to the static case. It is interesting to consider whether there is a CRCW algorithm for the static Write-All that requires $o(N \log N)$ work.

References

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

[2] R. Anderson and H. Woll, Wait-Free Parallel Algorithms for the Union-Find Problem, in: *Proc. 23rd ACM STOC* (1991) 370–380.

[3] Y. Aumann and M. O. Rabin, Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation, in: *Proc. 34th IEEE FOCS* (1992) 147–156.

[4] Y. Aumann, Z. M. Kedem, K. V. Palem and M. O. Rabin, Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs, in: *Proc. 35th IEEE FOCS* (1993) 271–280.

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

[6] R. P. Brent, The parallel evaluation of general arithmetic expressions, *J. ACM* **21** (1974) 201–206.

[7] J. Buss, P.C. Kanellakis, P. Ragde and A. A. Shvartsman, Parallel Algorithms with Processor Failures and Delays, Brown University TR CS-91-54, 1991. (Prel. version appears as P. C. Kanellakis and A. A. Shvartsman, Efficient Parallel Algorithms On Restartable Fail-Stop Processors, in: *Proc. 10th ACM PODC* (1991) 23–36.)

[8] R. Cole and O. Zajicek, The APRAM: Incorporating Asynchrony into the PRAM Model, in: *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures* (1989) 170–178.

[9] R. Cole and O. Zajicek, The Expected Advantage of Asynchrony, in: *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures* (1990) 85–94.

[10] F. Cristian, Understanding Fault-Tolerant Distributed Systems, *Communications of the ACM* **3** 2 (1991) 56–78.

[11] C. Dwork, J. Halpern and O. Waarts, Performing work efficiently in the presence of faults, *Proc. 11th ACM PODC* (1992) 91–102.

[12] D. Eppstein and Z. Galil, Parallel Techniques for Combinatorial Computation, *Annual Computer Science Review* **3** (1988) 233–283.

[13] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, in: *Proc. 10th ACM STOC* (1978) 114–118.

[14] P. Gibbons, A More Practical PRAM Model, in: *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures* (1989) 158–168.

[15] *Fault-Tolerant Computing*, IEEE Computer special issue **17** 8 (1984).

[16] *Fault-Tolerant Systems*, IEEE Computer special issue **23** 7 (1990).

[17] P. C. Kanellakis, D. Michailidis and A. A. Shvartsman, Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms, *Proc. 7th WDAG*, Lecture Notes in Computer Science vol. 725 (1993) 99–114.

[18] P. C. Kanellakis and A. A. Shvartsman, Efficient Parallel Algorithms Can Be Made Robust, *Distributed Computing* **5** (1992) 201–217. (Prel. version in *Proc. 8th ACM PODC* (1989) 138–148.)

[19] R. M. Karp and V. Ramachandran, A Survey of Parallel Algorithms for Shared-Memory Machines, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science* (North-Holland, 1990) 869–941.

[20] Z. M. Kedem, K. V. Palem and P. Spirakis, Efficient Robust Parallel Computations, in: *Proc. 22nd ACM STOC* (1990) 138–148.

[21] Z. M. Kedem, K. V. Palem, M. O. Rabin and A. Raghunathan, Efficient Program Transformations for Resilient Parallel Computation via Randomization, in: *Proc. 24th ACM STOC* (1992) 306–318.

[22] Z. M. Kedem, K. V. Palem, A. Raghunathan and P. Spirakis, Combining Tentative and Definite Executions for Dependable Parallel Computing, in: *Proc. 23rd ACM STOC* (1991) 381–390.

[23] C. P. Kruskal, L. Rudolph, M. Snir, Efficient Synchronization on Multiprocessors with Shared Memory, *ACM TOPLAS* **10** (1988) pp. 579–601.

[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* **21** (1992) 1070–1099.

[26] C. Martel, R. Subramonian and A. Park, Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs, in: *Proc. 32nd IEEE FOCS* (1990) 590–599.

[27] N. Nishimura, Asynchronous Shared Memory Parallel Computation, in: *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures* (1990) 76–84.

[28] D. B. Sarrazin and M. Malek, Fault-Tolerant Semiconductor Memories, *IEEE Computer* **17** 8 (1984) 49–56.

[29] R. D. Schlichting and F. B. Schneider, Fail-Stop Processors: an Approach to Designing Fault-tolerant Computing Systems, *ACM Trans. on Comp. Sys.* **1** (1983) 222–238.

[30] J. T. Schwartz, Ultracomputers, *ACM TOPLAS* **2** (1980) 484–521.

[31] A. A. Shvartsman, Achieving Optimal CRCW PRAM Fault-Tolerance, *Information Processing Letters* **39** (1991) 59–66.

[32] A. A. Shvartsman, An Efficient Write-All Algorithm for Fail-Stop PRAM without Initialized Memory, *Information Processing Letters* **44** (1992) 223–231.