Efficient parallel algorithms can be made robust

Paris C. Kanellakis* Alex A. Shvartsman†

Abstract

The efficient parallel algorithms proposed for many fundamental problems, such as list ranking, computing preorder numberings and other functions on trees, or integer sorting, are very sensitive to processor failures. The requirement of efficiency (commonly formalized using *Parallel-time* × *Processors* as a cost measure) has led to the design of highly tuned PRAM algorithms which, given the additional constraint of simple processor failures, unfortunately become inefficient or even incorrect. We propose a new notion of *robustness*, that combines efficiency with fault tolerance. For the common case of fail-stop errors, we develop a general (and easy to implement) technique to make robust many efficient parallel algorithms, e.g., algorithms for all the problems listed above. More specifically, *for any dynamic pattern of fail-stop errors with at least one surviving processor*, our method increases the original algorithm cost by at most a multiplicative factor *polylogarithmic* in the input size.

1 Introduction

An important issue for the full utilization of multiprocessor technology is: "the reliability problems of systems consisting of a large number of processors". Such "massively parallel" systems use many general-purpose, inexpensive processing elements to attain computation speed-ups comparable to or better than those achieved by expensive, specialized machines with a small number of fast processors. As the number of such inexpensive elements grows, one would expect to see an increased number of failures, because it would be too costly to make each element as reliable in hardware as a single (or few) processor machine. In response to this problem, *fault tolerance* is introduced in multiprocessors through *redundancy* in hardware and software.

In this paper we present a software approach to fault tolerance that provides an effective way of transforming existing parallel algorithms so that: *when processor fail-stop errors occur, the algorithm dynamically reconfigures its computation and can successfully proceed to completion, as long as there is at least one surviving processor.* Most importantly, the transformed parallel algorithm is *robust* in the sense that the efficiency of the original parallel algorithm is not significantly affected: *for any pattern of fail-stop errors, the original (Parallel-time × Processors) performance is increased, in the worst case, by a polylogarithmic, in the input size, multiplicative factor.*

Our contributions are: (1) defining the notion of robust parallel computation, and (2) showing its feasibility and wide applicability.

To understand the motivation for our work consider a number of fundamental computation tasks such as: manipulating integers (e.g., add or sort N integers) manipulating lists and trees (e.g., compute the rankings of the elements of a N-size list, and a preorder numbering, subtree sizes,..., of a N-size tree) and finding numerical solutions of equations using iterative methods. *Efficient parallel algorithms* exist for these problems. Namely, algorithms that attain close to linear speed-ups. That is, using P processors the parallel time achieved is $T(N)/P$, where $T(N)$ is the best known sequential bound for inputs of size N (within

*Address: *INRIA / Altaïr, BP 105, Rocquencourt 78153 Le Chesnay Cedex, FRANCE.* Currently on leave from Brown University. The research of this author was supported by NSF grant IRI-8617344, ONR grant N00014-83-K-0146 ARPA Order No. 4786, and an Alfred P. Sloan fellowship.

†Address: *Department of Computer Science, Brown University, PO Box 1910, Providence, RI 02912, USA* and *Digital Equipment Corporation, MLO21-3/E87, 146 Main Street, Maynard, MA 01754, USA.* The research of this author was supported by NSF grant IRI-8617344.

a multiplicative factor polylogarithmic in N) and P ranges over $1, \ldots, N$. These algorithms play an important role in realizing the promise of high speed-ups using "massive" parallelism.

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

The model of parallel computation that we use in this paper is: the synchronous, concurrent read, concurrent write (CRCW) PRAM of [FW 78], where the highest numbered processor write succeeds and the word size is $O(\log N)$, on inputs of size N. We investigate PRAM processors with *atomic* steps and *fail-stop* behavior, e.g., [SS 83]. (See Section 2). Consider a parallel algorithm that uses P processors and that, in τ parallel-time, completes its task on some input data I and in the presence of fail-stop errors F. Let there be $P_i(I,F)$ surviving processors at step i. The parallel-time processor product $\tau \times P$ is no longer that relevant. We use its natural generalization $S(I,F)$, which we call *the available processor steps* or S for short.

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

In order to deal with processor failures, it is necessary for the correct processors to detect the errors and reschedule the work of the failed processors. The main problem for efficient parallel algorithms is that the minimization of redundancy in the computation does not leave many resources for *error detection and load rescheduling*. It is fairly easy to see that naive processor error detection and reassignment strategies (e.g., use a master control or cluster the processors) are inadequate. A master control is sensitive to particular patterns of simple errors. Clustering can increase the performance, measured as worst case available processor steps (S defined above), by a linear or greater multiplicative factor. Let us illustrate this discussion with an example.

Example 1.1 One of the simplest tasks accomplished in constant time by a N processor PRAM is the task of: *given a zero-valued array of N elements, write value 1 into each array location.* This *Write-All* problem is trivially accomplished by the following (PRAM) program.

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

However, if as little as a single processor fails, then $x[i]=1$ ($i=1, \ldots, N$) is no longer guaranteed as a post-condition. Simple fixes are available that will make the above program more fault tolerant. For example, for a small number of failures $< k$, consider the clustering algorithm below. It performs well for dynamic failure patterns with few errors, but poorly if there are many failures. For example, if k is fixed and N variable then it cannot handle $N/2$ failures, and if k is allowed to grow to $N/2$ then S becomes quadratic in N.

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

□

Showing the existence of an algorithm for the *Write-All* problem of the previous example, for which $S = O(N \log^2 N)$, is the key to our technique. Such an algorithm also illustrates the notion of *robustness*. The original parallel-time processor product, N, is increased by at most a $c \log^2 N$ multiplicative factor, for any dynamic pattern of failures with at least one surviving processor. Note that we have no a-priori knowledge of how many, when, or which processors will fail.

Our robust solution of the *Write-All* problem is described in Section 3. It is based on a parallel *loop* through: (1) an error detecting phase, (2) a load rescheduling phase, (3) a work phase where assignment ($x[i]:=1$) is performed, and (4) a phase that estimates the work remaining and controls the parallel *loop*. The entire algorithm (W) is moderately involved, but fairly modular. Phases 1 and 4 involve bottom up traversal of two different heaps and phase 2 involves a top down traversal of these heaps. Algorithm W uses the ability to atomically write PRAM words of $O(\log N)$ bits and not only $O(1)$ bits. (In Section 6 we describe how to modify W to relax this assumption). Our solution is simple enough to capture certain engineering intuitions (e.g., the rescheduling involves divide-and-conquer) and to be easily implementable (e.g., we include a reasonably low level description of the code in Appendix A). Proving robustness is the subject of Section 4. The phases of the algorithm are such that reasoning about the failure patterns does not involve many cases and the algorithm analysis uses recurrences and inequalities.

In Section 5, we use the *Write-All* problem to derive robust solutions for a variety of problems. Most of the efficient algorithms in the literature can be made robust. There are a number of easy, yet interesting

consequences. The everpresent pointer-doubling algorithm can be implemented in a robust way and this leads to list ranking with $S = O(N \log^3 N)$. One can compute the tree functions described in [TV 84] with $S = O(N \log^3 N)$. Batcher's bitonic sort [B 68] can be adapted to give a robust PRAM algorithm for integer sorting with $S = O(N \log^4 N)$. Robust PRAM summation of N integers is possible with $S = O(N \log^2 N)$. For an application from numerical analysis: with the proposed approach we can increase the fault tolerance of *asynchronous iterative methods* [B 78].

In Section 6, we describe how to modify algorithm W in order to use only atomic writes of constant word size, without changing its performance. We conclude in Section 7 with some open questions.

Relationship to other work on fault tolerance:

Adding fault tolerance to algorithms is the subject of much current research in the (qualitatively different) setting of dynamic asynchronous network protocols, see [AAG 87], [AAPS 87], [A 88], [AS 88] for recent results and an overview of this area. Distributed controllers have been developed for resource allocation in network protocols, where the total number of messages sent is the resource controlled. For instance, the algorithms of [LGFG 86] (with a probabilistic setting) and of [AAPS 87] (with a deterministic setting) are among the most sophisticated in that area. The problem we address in this paper is, at an intuitive level, one of controlling resource allocation. The resource controlled is all available PRAM processor steps, and the reason we are forced to control it, is the requirement to complete the computation in the presence of faults. Note that unreliable PRAM processor steps must control all available PRAM processor steps. This introduces difficulties that recall the presence of network changes in [AAG 87], [A 88], [AS 88], i.e., dynamic changes of the computation medium.

At a high level, there are some analogies between our approach and the controller of [AAPS 87]. But our emphasis is on fault tolerant, efficient, parallel computation versus control of communication complexity in distributed networks. More specifically, in both cases, there are similar concerns of error detection (best performed through some approximate calculation) and load rescheduling (best performed through some divide-and-conquer according to a hierarchy). However, in each case, we have very different underlying models for computation, for faults and for the resources controlled. As a consequence, the actual algorithms as well as their analysis differ.

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

Our modeling of fault tolerance has some similarities with the design of "robust" sorting networks, as in [R 85], and in general with the design of reliable systems from unreliable components, as in [P 85] or [DPPU 86]. One distinguishing characteristic of our approach is investigating fault tolerance at the (PRAM) processor level as opposed to at the gate level [P 85, R 85]. Our notion of robustness differs from that of [R 85]. In the sorting network of [R 85], a linear number of operations are still critical.

Finally, the parallel setting with fail-stop processor errors is free from the limitations inherent in situations that require consensus ([PSL 80], see [F 83] for a good survey of the topic) such as the lower bounds of [FL 82], [FLP 85], [DDS 83]. This is because atomic broadcast can be simulated in a synchronous multiprocessor with shared memory and concurrent writes.

2 Robust parallel computation

We use the CRCW PRAM model [FW 78] where the highest numbered processor write succeeds and the word size is $O(\log N)$ on inputs of size N. Our basic technique can be applied in a fairly model independent fashion. So in describing PRAM algorithms we use *Pascal*-like notation with obvious constructs such as **parbegin** ... **parend**.

For our computation model we assume that: (1) We start with a set of P *initial* processors. Each of these initial processors knows its PID, a unique number in the range $1, \ldots, P$. (2) Each processor knows both the number of initial processors P and the input size N. It reads them from some common read-only store. (3) All processors execute the same instructions, from some common read-only store, on different data. Also, in the algorithms of this paper all concurrently writing processors write the same value. (4) We denote as **shared** the data-structures manipulated by more than one processor. All other data-structures are of constant size and are **local** to the processors. The *initial* values in all these data-structures are assumed to be 0.

For our failure model we assume that: (1) We only consider *fail-stop* behavior: *processors can fail between PRAM steps by stopping and not performing any further steps.* Fail-stop models are a reasonable approximation of what is desirable and achievable in practice [SS 83]. (2) We assume that individual PRAM steps are *atomic*: *if they execute then they execute fully.* This

is a nontrivial assumption, since words are of $O(\log N)$ size. We will reexamine it in Section 6. (3) We allow *any dynamic pattern* F of fail-stop errors provided one processor survives. Clearly 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 all about the algorithm and is totally unknown to the algorithm. (Note that the only limitations of the adversary are that: errors must be fail-stop and one processor survives).

Let a parallel algorithm complete its task, on some input data I and in the presence of fail-stop errors F, in parallel-time τ. If there are $P_i(I,F)$ surviving processors at step i then: $S(I,F) = \sum_{i=1}^{\tau} P_i(I,F)$. $S(I,F)$ is the number of all *available* PRAM *processor steps.*

Definition 2.1 Let $T(N)$ be the best sequential (RAM) time bound known for N-size instances of problem Π. We say that an algorithm for Π is a *robust parallel algorithm* if: for any input I of size N and for any number of initial processors P (where $1 \leq P \leq N$) and for any failure pattern F, this algorithm has $S(I,F) \leq cT(N)\log^{c'} N$, for some fixed constants c, c'.

For simplicity of presentation, in the rest of this abstract we assume that: P the initial number of processors is N, where N is the input size. Our results easily extend to any P in the range $1,\ldots,N$. All logarithms in this paper are base 2.

3 The Algorithm W

In this section we describe a robust parallel algorithm W for the *Write-All* problem from Example 1.1. Here and in the detailed description in the Appendices we assume that N is a power of 2. Nonpowers of 2 can be easily handled using conventional padding techniques. (Also recall that $P = N$).

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

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

The input is in shared array $x[1\ldots N]$, where the N elements of this array should be thought of as related to the leaves of the heaps. Element $x[i]$ is related to $c[i+N-1]$, $cs[i+N-1]$, $d[i+N-1]$, and to $a[i+N-1]$.

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

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

Control-flow: The algorithm consists of the parallel *loop* right below. This is performed, in a synchronous way, by all processors that have not stopped. The *loop* consists of four phases of steps, and the first time only part of it is executed (phases 3 and 4). Of course, processors can fail-stop at any time during the algorithm.

forall processors PID=1..N **parbegin**

(Phase 3:) Visit the leaves based on PID to perform work on the input data.
(Phase 4:) Traverse the d heap bottom up to measure progress.

while the root of the d heap is not N **do**

(Phase 1:) Traverse the c, cs heaps bottom up to count processors and give them pn's.
(Phase 2:) Traverse the d, a, c heaps top down to reschedule work.
(Phase 3:) Perform rescheduled work on the input data.
(Phase 4:) Traverse the d heap bottom up to measure progress.

od
parend

The *basic idea* of the *loop* is: "For *error detection* use bottom up, fast parallel summation to estimate the surviving processors and to estimate the progress they have made. For *load rescheduling* use a top down, divide-and-conquer strategy based on the estimate of progress made". This idea is realized as follows.

Phase 1: Each processor PID traverses heaps c and cs bottom up from x[PID] (i.e., from location PID+$N-1$). The $O(\log N)$ path of this traversal is the same (static) for all the loop iterations. As a processor performs this traversal it calculates an overestimate of the surviving processors. For this, it uses a standard $O(\log N)$ parallel-time version of a CRCW summation algorithm. Heap c holds the sums and heap cs timestamps (or step numbers) for the current *loop* iteration. This allows reusing c without having to initialize it each time. Also, during this traversal surviving processors calculate new processor numbers pn for themselves, based on the same sums. (Procedure Static Bottom Up Traversal in Appendix A).

Phase 2: All surviving processors now start at the root of the d heap. In $d[i]$ there is an underestimate

of the work already performed in the subtree defined by i. Now the processors traverse the d heap top down and get rescheduled dynamically according to the work remaining to be done in the children of i. Auxiliary heap a is used by the processors to compute paths to the leaves that were visited in the past and whose count is reflected in $d[1]$. We compute a from d during the traversal. The rescheduling of work is done using divide-and-conquer according to $N - a[2i]$ and $N - a[2i+1]$. This is accomplished by modifying pn and reusing heap c. (Procedure Dynamic Top Down Traversal in Appendix A).

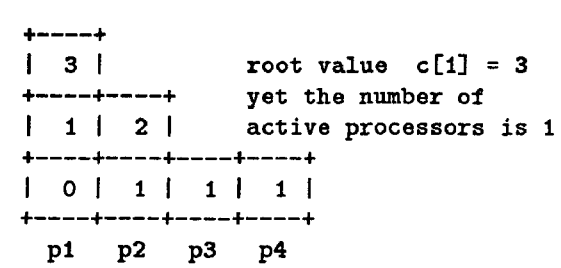
Phase 3: All processors are now at the leaves. Each processor tries to write 1 in the leaf it has been rescheduled to. To start the *loop* each processor PID tries to write in location x[PID]. (Procedure Main in Appendix A).

Phase 4: The processors record the progress made by traversing the d heap bottom up and using the standard summation method. The $O(\log N)$ path of this (dynamic) traversal can differ in each *loop* iteration, since processors start from the leaves where they were in phase 3. What is computed each time is an underestimate of the progress made. No timestamps are needed here because the progress recorded increases monotonically. (Procedure Dynamic Bottom Up Traversal in Appendix A).

Let us now examine phase 1 in some more detail. In the bottom up traversal processor PID writes a 1 in leaf $c[\text{PID}+N-1]$ of the tree c. If a processor has failed before it wrote 1 then that will not contribute to the overall count. If a processor failed after it wrote 1 then this number still can contribute to the overall sum, if its subtree neighbor survives. The same observation applies to counts at internal nodes, which are sums of the counts of the children nodes in tree c.

It is easy to show that: *phase 1 will always compute in $c[1]$ an overestimate of the number of processors, which are surviving at the time of its completion.* (See Lemma 4.1).

For example: given 4 processors, if processor 1 failed before the start of phase 1, and processor 3 failed right after writing 1 into its leaf $c[6]$, and processor 4 failed after calculating 2 the sum of its and processor 3's contribution in $c[3]$, then the heap will look as follows at the completion of the phase.



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

Finally, in phase 1 we must be able to reuse our heap several times. This presents a problem. For example, if a processor wrote 1 into its heap leaf and then failed then the 1 will remain there forever, thus preventing us from computing monotonically tighter estimates of the number of surviving processors. This is rectified by associating a step number with each node of the count heap, thus "time stamping" valid data. The count step is initially zero, and during each successive *loop* iteration, gets incremented by each surviving processor. Failed processors will not increment their step numbers, thus enabling the surviving processors to detect counts that are "out of date" and treat them as zeros. We need not worry about "time stamping" overflow, since we have words of $O(\log N)$ bits and in the worst case the *loop* iterates N times. (Because, every iteration writes one bit at least).

We now comment first on phases 3, 4 and then on phase 2. Phase 3 is where the work of the original nonrobust algorithm gets done. Phase 4 is a simple variant of phase 1, except for the fact that the path traversed bottom up is dynamically determined. One can easily show that: *the progress recorded in $d[1]$ by phase 4 increases monotonically and underestimates the actual progress.* (See Lemma 4.1).

In phase 2, it is essential to be able to *divide the remaining work evenly among the remaining processors.* In the next section, we will show that this is done with at most a small round-off error. (See Lemma 4.2). We use pn (dynamic processor number) and reuse heap c to partition the processors, between the left/right subtrees in heap d.

In this phase, we are going to guarantee that the remaining processors are divided evenly among the leaves that either have not been visited or whose visitation was not properly recorded in $d[1]$ (recall that $d[1]$ is an underestimate of the number of the leaves visited). This is accomplished within the top down traversal by disregarding "partial" progress recorded by the processors in the dynamic bottom up traversal of heap d. This partial progress can be detected, during the dynamic top down traversal, when a value at an internal node of d is less than the sum of the values of its two descendants. This could happen, if a processor had failed during the dynamic bottom up traversal of d. In algorithm W, we use an auxiliary heap a where the values of d are appropriately reduced to produce a correct summation heap. The leaves of heap a with value 0 are those leaves that have not contributed to the count in $d[1]$.

The values of heap a are nonnegative integers constrained (top down) as follows: $a[1]=d[1]$ and $a[2i]+a[2i+1]=a[i]$ and $a[2i] \leq d[2i]$ and $a[2i+1] \leq d[2i+1]$ $(1 \leq i \leq N-1)$. Clearly this does not define the values of a uniquely, and our top-down traversal in Appendix A implements one way of computing the values of a satisfying these constraints. Note that, for any $a[i]$, its value is computed based only on the values of heap d and the values of a along the unique path from $a[i]$ to the root $a[1]$, this allows us to compute the accounted values in parallel.

4 Analysis of Algorithm W

We now outline the proof of robustness for the algorithm W, described in Section 3.

In the dividing done during the dynamic top down traversal in W, we will allocate processors to tasks that have been completed, but not yet "accounted for" at the root $d[1]$. Formally, a leaf of d is *accounted* if it has value 1 and if the corresponding computed value in the leaf of heap a is also 1. In the algorithm W, the processors get allocated to the *unaccounted* leaves (leaves whose associated value in heap a is 0) in a balanced fashion.

We need some more terminology. Let us consider the i-th iteration of the *loop*. Define: (1) U_i to be the estimated remaining work, the value of $N - d[1]$ right before the iteration starts (i.e., right after phase 4 of the previous iteration). (2) P_i to be the real number of surviving processors, right before the iteration starts (i.e., right after phase 4 of the previous iteration). (3) R_i to be the estimated number of surviving processors, that is the value of $c[1]$ right after phase 1 of the iteration. Our algorithm produces R_i that is an overestimate of surviving processors P_{i+1}. However, at most P_i processors can be counted (an upper bound for R_i). The following can be shown by straightforward induction on the structure of the tree c (the proof is omited from this abstract).

Lemma 4.1 In algorithm W, for all loop-iterations i we have: $P_i \geq R_i \geq P_{i+1}$ and $U_i \geq U_{i+1}$, as long as at least one processor survives.

The next lemma is proven by establishing an invariant for phase 2 of the algorithm (the proof is easy and omited from this abstract).

Lemma 4.2 In phase 2 of each loop-iteration i of algorithm W: (1) processors are only allocated to unaccounted leaves, and (2) no leaf is allocated more than $\lceil R_i/U_i \rceil$ processors.

We will treat the three $\log N$ time tree traversals performed by a single processor during each phase of the algorithm as a single *block-step* of cost $O(\log N)$. We will charge each processor for each such block step, regardless of whether the processor actually completes the traversals or whether it fail-stops somewhere in-between. This coarseness will not distort our results; since we can have at most P processor failures it amounts to a one time overcharge of $O(P \log N)$.

Let us take a snapshot of the algorithm after completion of several loop-iterations. We are right before loop-iteration i. V_i stands for the total number of block-steps performed by the processors in trying to complete all remaining work (at most U_i). Now we present the central lemma:

Lemma 4.3 For any failure pattern with at least one surviving processor, and starting at each loop-iteration i, algorithm W completes all remaining work. Also, its total number of block-steps V_i is less than or equal to $P_i + U_i + P_i \log(U_i)$, where $1 \leq P_i, U_i \leq N$.

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

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

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

In both cases, by Lemma 4.2, We have that the (accounted) progress for iteration i is at least the number of surviving processors P_{i+1} divided by $\lceil R_i/U_i \rceil$. This is because each one of these processors returns to the root $d[1]$, reporting some progress, and at most $\lceil R_i/U_i \rceil$ processors report information about the same leaf.

Also, by Lemma 4.1, $P_i \geq R_i \geq P_{i+1}$ and we can assume that $kP_i = P_{i+1}$, for some k with $0 < k \leq 1$ (at least one processor survives). Thus, for both the above cases, we have:

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

$$\leq U_i(1 - \frac{k}{1 + U_i/P_i})$$

For case (1) it is easy to see that we will have at most one processor allocated to each unaccounted leaf so: $U_{i+1} \leq U_i - P_{i+1}$. For case (2) by the above inequality and $P_i > U_i$ we have $U_{i+1} \leq U_i(1 - k/2)$. Now we use the inductive hypothesis (but for iteration $i+1$) in both cases.

Case (1): The survival of at least one processor and $U_{i+1} \leq U_i - P_{i+1}$ imply that $U_{i+1} < U_i$. The total work (in block-steps) is at most $P_i + V_{i+1}$, where by the hypothesis $V_{i+1} \leq P_{i+1} + U_{i+1} + P_{i+1}\log(U_{i+1})$. Thus, it suffices to show that $P_{i+1} + U_{i+1} + P_{i+1}\log(U_{i+1})$ is less than or equal to $U_i + P_i \log(U_i)$. This is trivial given $U_{i+1} \leq U_i - P_{i+1}$ and Lemma 4.1.

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

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

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

Recall that all logarithms are base 2 and therefore $(\log(1/2) = -1)$. Since $U_i \geq 2$ ($U_i = 1$ was taken care of by base case) we have $\log U_i \geq 1$. Also, in this case $1 - U_i/2P_i \leq 1$. It thus suffices to show the inequality: $(*)$ $k\log(2-k) \leq (1-k)$, for $0 < k < 1$. Inequality $(*)$ is true by elementary calculus (it is tight only for $k = 1$). This completes the proof of the second subcase, of case (2) and of the Lemma. □

Theorem 4.1 Algorithm W is a robust parallel algorithm for the *Write-All* problem with $S = O(N \log^2 N)$, where N is the input array size.

This immediately follows from Definition 2.1 and Lemmas 4.2 and 4.3. In our analysis we show an upper bound of $O(N \log N)$ block-steps. It is possible to construct failure patterns that force the algorithm to take $\Omega(N \log\log N)$ (this example is due to Jeff Vitter) and $\Omega(N \log N/\log\log N)$ block-steps. Finally, note the difference between block-steps and loop-iterations; there can be at most N loop-iterations since $d[1]$ decreases by at least 1 each time.

5 Some Applications

The algorithm W for the *Write-All* problem can be used as a building block for transforming many efficient parallel algorithms into robust ones.

We can first extend the algorithm W to implement a robust *general parallel array assignment.* For example, consider computing and storing in an array $x[1 \ldots N]$ values of a function f whose values depend only on the processor numbers PID and the initial values of the array x. Also, assume f can be computed in $O(1)$ sequential time.

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

In order to adapt the algorithm W, we need to convert the parallel assignment to a form that is suitable for asynchronous fault-tolerant execution. This is accomplished using binary version numbers:

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

This approach is conceptually similar to that of [B 88] where a solution to the concurrent read/write register problem is given using two registers with single bit tags to implement a single register tolerant of fail-stop errors. Here, v is the current bit (modulo 2) version number (or tag), so that $x[v][1 \ldots N]$ is the array of current values. Function f will use only these values of x as its input. The values of f are stored in $x[v+1][1 \ldots N]$ creating the next generation of array x. After all the assignments are performed, the binary version number is incremented (modulo 2).

At this point, a simple transformation of the algorithm W will yield a robust algorithm for *general parallel array assignment.* In phase 3 the assignment of 1 to $x[i]$ is replaced with the assignment shown above. One important application of this technique produces a robust *pointer-doubling* operation that is a basic building block for many parallel algorithms. In a similar way, it is possible to systematically produce robust versions of many efficient parallel algorithms.

The transformation is almost automatic for a large number of efficient parallel algorithms, but not in all cases. The original algorithm has to have an iterative structure, that will be simulated by an iterated use of *Write-All.* Moreover, the computation performed must have a Church-Rosser like flavor. For example a sufficient condition is: "The efficient parallel algorithm manipulates an array x in an iterative loop. The new values in x computed by each iteration of the algorithm depend only on the old values of x. It does not matter if the operations on some data elements are not performed for some iterations". Such conditions are true for many numerical computations, such as in [B 78], to which we can add robustness (here the running-time also depends on a desired precision parameter ε). Let us describe some of its more practical consequences.

Proposition 5.1 There is a robust parallel algorithm for list ranking with $S = O(N \log^3 N)$, where N is the input list size.

Proposition 5.2 There is a robust parallel algorithm for computing the tree functions of [TV 84] with $S = O(N \log^3 N)$, where N is the input tree size.

Proposition 5.3 There is a robust parallel bitonic sort with $S = O(N \log^4 N)$, where N is the number of integers to sort.

6 On Constant Word Size

One observation is that algorithm W provides (by its definition) a robust parallel algorithm for N integer summation with $S = O(N \log^2 N)$. The interesting question, however is what happens with $O(1)$ bit words and arithmetic at the bit level. In this section, we adapt our method to $O(1)$ bit words.

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

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

The algorithm W can be modified so that: *there is preservation of the $O(N \log^2 N)$ available processor steps* (counting $\log N$ bit write cycles as one time unit) and *preservation of $O(N)$ word space use* (counting $\log N$ bits as one word).

This is accomplished by *simulating* $\log N$-size word atomic writes using a single bit *tag* and two $\log N$-size words. The two words are numbered 0 and 1, and the bit tag (initially 0) indicates which of the two words has valid contents. Thus each shared memory location is represented as:

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

Each read cycle of the shared memory now becomes:

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

The write cycle to the shared memory becomes:

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

Since the single bit tag is the last bit written during the write cycle, a failure anywhere during this high level write cycle will prevent the tag value to be updated, and so any subsequent read will be able to read the previous value stored. This approach is similar to that of [B 88], and it is somewhat simpler due to the fact that we are dealing with the *synchronous* model.

The algorithm W can be mechanically transformed using the macro read and write cycles above to a version that only requires single bit atomic writes. Clearly, the number of $\log N$-size words read or written by each macro cycle is $O(1)$ as before, and the shared memory requirements are within a factor of two of the original memory size. Therefore, performance of the modified algorithm has not changed asymptotically.

From the above discussion, it follows that robustness for *Write-All* can be achieved with constant size words. The same arguments apply to all the propositions on robustness in Section 5.

7 Conclusions

We have formally defined and demonstrated the feasibility and wide applicability of robust parallel computation. We close this paper with a number of open questions.

What are W's precise performance bounds and can it be improved? Are there general characterizations for classes of problems with robust parallel algorithms? What extensions can be made to our robustness definitions (e.g., other types of faults)? What are the precise analogs of robustness in network models of parallel computation (this could involve application of the work on distributed network controllers, e.g., [AAPS 87], to parallel computation)? What about randomized robust parallel computation (formalizations along the lines of [LGFG 86])?

Acknowledgements: We would like to thank François Bancilhon, Serge Abiteboul and Al Lathrop for their comments on a preliminary draft of this abstract as well as Serge Plotkin and Jeff Vitter for helpful discussions.

8 References

[A 88] B. Awerbuch, "On the effects of feedback in dynamic network protocols", *in Proc. of the 29th IEEE FOCS*, pp. 231-242, 1988.

[AAG 87] Y. Afek, B. Awerbuch, E. Gafni, "Applying static network protocols to dynamic networks", in *Proc. of the 28th IEEE FOCS*, pp. 358-370, 1987.

[AAPS 87] Y. Afek, B. Awerbuch, S. Plotkin, M. Saks, "Local management of a global resource in a communication network", *in Proc. of the 28th IEEE FOCS*, pp. 347-357, 1987.

[AS 88] B. Awerbuch, M. Sipser, "Dynamic networks are as fast as static networks", *in Proc. of the 29th IEEE FOCS*, pp. 206-219, 1988.

[B 68] K.E. Batcher, "Sorting networks and their applications", *in Proc. of the AFIPS Spring Joint Comp. Conf.*, vol. 32, pp. 307-314, 1968.

[B 78] G. Baudet, "Asynchronous iterative methods for multiprocessors", *JACM*, vol. 25, no. 2, pp. 226-244, 1978.

[B 88] B. Bloom, "Constructing two-writer atomic registers", *IEEE Trans. on Computers*, vol. 37, no. 12, pp. 1506-1514, 1988.

[DDS 83] D. Dolev, C. Dwork, L. Stockmeyer, "On the minimal synchronism needed for distributed consensus", *in Proc. of the 24th IEEE FOCS*, pp. 393–402, 1983.

[DPPU 86] C. Dwork, D. Peleg, N. Pippinger, E. Upfal, "Fault tolerance in networks of bounded degree", *in Proc. of the 18th ACM STOC*, pp. 370-379, 1986.

[F 83] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)", *Yale Univ. Tech. Rep.*, DCS/RR-273, 1983.

[FL 82] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency", *IPL*, vol. 14., no. 4, pp. 183-186, 1982.

[FLP 85] M. J. Fischer, N. A. Lynch, M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *JACM*, vol. 32, no. 2, pp. 374-382, 1985.

[FW 78] S. Fortune and J. Wyllie, "Parallelism in random access machines", *Proc. 10th ACM STOC*, pp. 114-118, 1978.

[LGFG 86] N.A. Lynch, N.D. Griffeth, M.J. Fischer, L.J. Guibas, "Probabilistic analysis of a network resource allocation algorithm", *Information and Control*, vol. 68, pp. 47-85, 1986.

[P 85] N. Pippinger, "On networks of noisy gates", *in Proc. of the 26th IEEE FOCS*, pp. 30-38, 1985.

[PSL 80] M. Pease, R. Shostak, L. Lamport, "Reaching agreement in the presence of faults", *JACM*, vol. 27, no. 2, pp. 228-234, 1980.

[R 85] L. Rudolph, "A robust sorting network", *IEEE Trans. on Comp.*, vol. c-34, no. 4, pp. 326-335, 1985.

[S 83] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems", *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222-238, 1983.

[TV 84] R. E. Tarjan, U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time", *in Proc. of the 25th IEEE FOCS*, pp. 12-22, 1984.

A Write-All Implementation

A.1 Main Procedure for Write-All

The *loop* consists of the four phases outlined in Section 3. Processor counting and enumeration is implemented as a static bootom up traversal in procedure S_BU(), work assignment is done in a dynamic top down traversal in procedure D_TD(), the work itself is a simple assignmnet "x[k]:=1", and the progress is measured in procedure via a dynamic bottom up traversal in D_BU().

```
forall processors PID=1..N
parbegin
    shared integer array
        x[1..N], --input array
        c[1..2N-1], --processor counts
        cs[1..2N-1], --count step numbers
        d[1..2N-1], --progress/done tree
        a[1..2N-1]; --accounted tree
    local integer
        pn, --enumerated processor no.
        k, --array element PID will be assigned to
        step; --time stamp

    step := 0; --initialize processor counting step
    k := PID; --initially work data item PID
    x[k] := 1; --visit leaf
    D_BU(k); --measure progress

    while d[1] ≠ N do

        S_BU(PID,step,pn); --enumerate proc-s
        D_TD(pn,k); --assign work
        x[k] := 1; --do work: visit leaf
        D_BU(k); --measure progress

    od
parend ;
```

A.2 Static Bottom Up Traversal

All processors traverse heap c to compute the overestimate of the number of processors in c[1], and each processor computes its processor number pn that is used in the work assignment phase. The heap cs is used to synchronize processor counting across multiple calls to S_BU().

```
procedure S_BU(value integer PID, --proc. id
               shared integer step, --time
               local integer pn) --proc. no.
    shared integer array
        c[1..2N-1], --processor counts
        cs[1..2N-1]; --count step numbers
    local integer j1, j2, --siblings indices
               t; --parent of j1 and j2

    step := step + 1; --new time stamp
    j1 := PID + (N-1); --heap-leaf init
    pn := 1; --assume this processor is no. 1
    c[j1] := 1;
    cs[j1] := step; --count the processor once

    --Traverse the tree from leaf to root
    for 1..log(N) do

        t := j1 div 2; --parent of j1 and j2

        if 2*t = j1
        then j2 := j1 + 1 --j1 came from left
        else j2 := j1 - 1 --j1 came from right
        fi ;

        if cs[j1] = cs[j2] --both sub-trees active?
        then c[t] := c[j1] + c[j2] --both active
            if j1 > j2 --j1 came from right
            then pn := pn + c[j2]
            fi
        else c[t] := c[j1] --all siblings failed
        fi ;
        cs[t] := step; --time stamp, and
        j1 := t --advance up the heap
    od
end ;
```

A.3 Dynamic Bottom Up Traversal

Heap d contains the underestimates for the number of leaves visited in each subtree, with d[1] containing the underestimate of the total number of leaves visited. This number is used in terminating the overall program (when d[1]=N).

```
procedure D_BU(value integer k  --current leaf
                )
    shared integer array
        d[1..2N-1];  --done/progress tree
    local integer
        i1, i2,  --siblings indices
        t;  --parent of i1 and i2

    i1 := k + (N-1);  --heap-leaf init.
    d[i1] := 1;  --done for good

    --Traverse the tree from leaf to root
    for 1..log(N) do

        t := i1 div 2;  --parent of i1 and i2

        --compute left/right indices
        if 2*t = i1
        then i2 := i1 + 1
        else i2 := i1 - 1
        fi ;

        d[t] := d[i1] + d[i2];  --update progress
        i1 := t  --advance to the predecessor
    od
end ;
```

A.4 Dynamic Top Down Traversal

This procedure implements load rescheduling of the remaining active processors. Heaps c and d are traversed top down. Heap *a* is used to construct paths to a set of the properly accounted leaves. Heap c is used to partition the remaining processors between the left and right tree branches, and heap d contains the progress information for the subtrees being traversed. Processors are allocated in proportion to the remaining work.

```
procedure D_TD(value integer pn  --enum. no.
               local integer k)  --data item
    shared integer array
        c[1..2N-1],  --procesor counts
        d[1..2N-1];  --progress/done tree
        a[1..2N-1];  --accounted tree
    local integer i, i1, i2;  --curr./left/right indices

    i := 1;  --start at the root
    size := N;  --whole tree is visible from the root
    a[1] := d[1];  --no. of all accounted nodes

    while size ≠ 1 do  --traverse from root to leaf

        i1 := 2*i; i2 := i1 + 1;  --left/right indices

        --compute accounted node values
        if d[i1]+d[i2] = 0
        then a[i1] := 0
        else a[i1] := a[i]*d[i1] div(d[i1]+d[i2])
        fi ;
        a[i2] := a[i] - a[i1];

        --processor alloc. to left/right sub-trees
        c[2*i] := c[i]*((size/2)-a[2*i]) div(size-a[i])
        c[2*i+1] := c[i] - c[2*i];

        --go left/right based on proc. no.
        if pn ≤ c[2*i]
        then i := 2*i  --go left
        else i := 2*i + 1  --go right
             pn := pn - c[2*i]
        fi ;

        size := size div2  --half of leaves visible
    od ;
    k := i - (N-1)  --assign processor based on i
end ;
```