Efficient Parallel Algorithms Can Be Made Robust

Paris C. Kanellakis
Alex A. Shvartsman

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-89-35
October 24, 1989

Efficient parallel algorithms can be made robust *

Paris C. Kanellakis† Alex A. Shvartsman‡

October 24, 1989

Abstract

The efficient parallel algorithms proposed for many fundamental problems, such as list ranking, computing preorder numberings on trees and 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 on* a CRCW PRAM *with at least one surviving processor*, our method increases the original algorithm cost by at most a *polylog* multiplicative factor. We also show that at least a *log/loglog* multiplicative overhead will be incurred for certain patterns of failures by any algorithm that implements robust solutions to the problems listed.

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 processor machine. In response to this problem, *fault tolerance* is introduced in multiprocessors through *redundancy* in hardware and software.

In this paper we present an algorithmic 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

*A preliminary version of this work appears as part of the Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC'89), Edmonton 1989.

†Address: *Department of Computer Science, Brown University, PO Box 1910, Providence, RI 02912, USA.* 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.

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* $\times$ *Processors)* complexity is increased, in the worst case, by a multiplicative factor polylogarithmic in the input size.

Our contributions consist of: (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 compute a preorder numbering or subtree sizes, ..., of a N-size tree). Efficient parallel algorithms exist for these problems. Specifically, these problems can be solved by algorithms that attain a near linear speed-up. 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 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, so that every processor is fully utilized doing something essential for resolving the input task. Thus, parallel algorithm *efficiency* implies a *minimization of redundancy* in the computation that leaves very little room for fault tolerance. It is interesting to note that most of the known efficient parallel algorithms do not terminate correctly or become quite inefficient if they are perturbed by simple processor errors. These perturbations are of course outside the original setting, but are 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]. The convention for determining which processor or processors succeed, when concurrently writing to shared memory, is immaterial in our algorithms. but the ability to read and write concurrently is a requirement. We investigate PRAM processors with *atomic* steps and *fail-stop* behavior, e.g., [SS 83]. We only require concurrent reading and writing of $O(1)$ bit words. The redundancy provided by concurrent reads and writes is essential to our model (see Section 2 for more details).

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 of a master control or clustering of processors) are inadequate. A master control strategy is sensitive to particular patterns of simple errors. Clustering can degrade 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, even a single processor failure will prevent the establishing of $x[i]$=1 (i=1, ..., N) as the postcondition. 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 to PID+k do
        if i≤N then x[i] := 1 else x[i−N] := 1 fi
    od
parend
```

□

Showing the existence of an N-processor CRCW PRAM algorithm for the *Write-All* problem of Example 1.1, for which $S = O(N \log^2 N)$, is the key to our technique. This solution also illustrates the notion of *robustness.* The original parallel-time $\times$ processors 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 knowledge of how many, when, or which processors will fail.

Our robust solution to 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 assignments ($x[i]$:=1) are 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 of the PRAM to atomically write words of $O(\log N)$ bits. (This is only for convenience of presentation, in Section 7 we show how to modify W to relax this assumption and require only that the writes of $O(1)$ bits are atomic.)

This solution is simple enough to capture certain engineering intuitions (e.g., the rescheduling involves divide-and-conquer) and to be easily implementable (e.g., we include detailed description of the code in Appendix A). Proving robustness is the subject of Section 4. The phases of the algorithm are such that reasoning about the failure patterns involves few cases and the algorithm analysis uses recurrences and inequalities. By introducing a "slack" in the number of processors ($1 \leq P \leq N/\log^2 N$, where N is input size) we show that *Write-All* can be solved optimally with $S = O(N)$.

However, this "slack" is necessary. In Section 5 we give lower bound results. We prove that given any N-processor algorithm that implements a robust solution to the *Write-All* problem, it is possible to construct an adversary failure pattern such that $S = \Omega(N \log N/\log\log N)$. We also show that concurrent reads and writes are necessary. For example, *Write-All* has no robust CREW solution.

In Section 6, we use the *Write-All* problem to derive robust solutions for a variety of problems. There are a number of easy, yet interesting consequences. The everpresent pointer-doubling operation can be implemented in a robust way which leads to a robust list ranking algorithm with $S = O((N \log^2 N + P \log^3 N)/\log\log N)$ for N-sized lists and P initial processors ($P \leq N$). One can robustly compute the tree functions described in [TV 84]. Batcher's bitonic sort [B 68] can be adapted to give a robust PRAM algorithm for integer sorting. The same applies to all integer operations with efficient PRAM implementations. It has been recently shown that *all* efficient PRAM algorithms with P initial processors and input size N can be made robust [KKPS 89] and optimally so [Shv], for $P \leq N/\log^2 N$.

In Section 7, 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 8 with a discussion and some open questions.

Relationship to other work on fault tolerance:

Adding fault tolerance to algorithms is the subject of significant 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 the underlying models for computation, faults and the resources controlled are very different. 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 the investigation of 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 is 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] with word size $O(\log N)$ 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 parallel constructs such as **forall/parbegin/parend**. For our computation model we assume that:

1. There are P *initial* processors with unique identifiers (PID) in the range $1, \ldots, P$. Each processor has access to its PID, the initial number of processors P and the input size N.
2. In the algorithms of this paper all concurrently writing processors write the same value making our approach independent of CRCW conventions for concurrent write.
3. 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.

We use the following failure model:

1. 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 everything about the algorithm and is totally unknown to the algorithm.
2. We only consider *fail-stop* behavior: processors fail by stopping and not performing any further actions. Fail-stop models are a reasonable approximation of what is desirable and achievable in practice [SS 83].
3. We assume that the *shared memory writes* of the individual PRAM steps are *atomic* with respect to failures: failures can occur before or after a shared write, but not during the write. This is a nontrivial assumption, since words are of $O(\log N)$ size. (This assumption is reexamined in Section 7 and relaxed to only $O(1)$ atomic writes.)

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 a parallel (PRAM) 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 c, c'.

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 Appendix A we assume that N is a power of 2. Nonpowers of 2 can be handled using conventional padding techniques. All logarithms are base 2.

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

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 left child $h[2i]$ and right child $h[2i + 1]$.

The heaps are $c[1 \ldots 2N - 1]$ (for processor counting and allocation), $cs[1 \ldots 2N - 1]$ (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 initially 0.

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

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

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

Control-flow: The algorithm consists of the parallel *loop* given 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

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

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 processors perform this traversal they calculate an overestimate of the surviving processors. This is done using a standard $O(\log N)$ parallel-time version of a CRCW summation algorithm. Heap c holds the sums and heap cs the timestamps (or step numbers) for the current *loop* iteration. This allows reusing c without having to initialize it each time. Also, during this traversal surviving processors calculate new processor numbers pn for themselves, based on the same sums. This is Static Bottom Up Traversal procedure 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 subtrees of i. Auxiliary heap a is used by the processors to compute paths to the unvisited leaves or the leaves that may have been visited in the past, but whose count is not 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. This is procedure Dynamic Top Down Traversal in Appendix A.

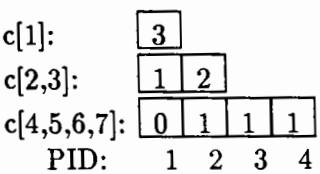
Phase 3: All processors are now at the leaves. Each processor tries to write 1 in the leaf it has been rescheduled to. To start the *loop* each processor PID tries to write in location x[PID]. This is procedure Main in Appendix A.

Phase 4: The processors record the progress made by traversing the d heap bottom up and using the standard summation method. The $O(\log N)$ paths (dynamically) traversed by processors 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. This is procedure Dynamic Bottom Up Traversal in Appendix A.

Let us now examine phase 1 in more detail. In the bottom up traversal processor PID writes a 1 in leaf c[PID+$N-1$] of the tree c. If a processor fails *before* it writes 1 then its action will not contribute to the overall count. If a processor fails *after* it writes 1 then this number can still contribute to the overall sum if a 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.

Example 3.1 Consider algorithm W for $N = 4$. There are 4 processors with PIDs 1, 2, 3, and 4, and the tree c represented as the heap c[1..7].

If processor 1 failed prior to the start of phase 1, processor 3 failed right after writing 1 into its leaf c[6], and processor 4 failed after calculating $c[3] = 2$ as the sum of its ($c[4] = 1$) and processor 3's ($c[3] = 1$) contributions, then the heap will look like this after the completion of the phase.



Observe that the root value $c[1] = 3$ yet the actual number of active processors is 1. □

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).

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 had written 1 into its heap leaf and then failed then the value 1 will remain there for the duration of the computation, thus preventing us from computing monotonically tighter estimates of the number of surviving processors. This is corrected by associating a step number with each node of the count heap c and storing it in heap cs, thus time stamping valid data. The count step is initially zero, and during each successive *loop* iteration, gets incremented by each surviving processor. Failed processors will not increment their step numbers, thus enabling the surviving processors to detect counts that are out-of-date and treat them as 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 (Lemma 4.3).

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.

We 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 "incomplete" progress caused by failures and recorded by the processors in the dynamic bottom up traversal of heap d. This incomplete progress can be detected during phase 2 when a value of an internal node in heap d is less than the sum of the values of its two descendants. We use an auxiliary heap a where the values of d are appropriately reduced to produce a correct summation tree. The leaves of heap a with value 0 (when computed in a top down traversal) are those leaves that have not contributed to the count in $d[1]$ (Lemma 4.2).

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 dynamic 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 the root $a[1]$ to $a[i]$, this allows us to compute the accounted values in parallel.

Phase 3 is where the work of the original non-robust 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 4.3). This guarantees that the algorithm terminates after at most N iterations, since $d[1] = N$ is the guard that controls the main *loop*.

4 Analysis of Algorithm W

We now outline the proof of robustness for the algorithm W, described in Section 3. Lemma 4.1 shows that in each loop iteration, the algorithm computes tight (over)estimates of the remaining processors. In Lemma 4.2 we prove that processors are only allocated to leaves that were not verifiably visited by processors, and that all such leaves are allocated a balanced number of processors. Lemma 4.3 assures monotonic progress of the computation, and thus its termination. In Lemma 4.4 we develop an upper bound on the work performed by the processors prior to the algorithm termination. Lemmas 4.1 and 4.3 are proved using simple inductions on the structure of the heaps

used by the algorithm. Lemma 4.2 involves a detailed but straight forward proof outline in the style of [D 76]. Lemma 4.4 is the central lemma of this section and its proof consists of a relatively involved induction on the size of the estimated work remaining at some step of the algorithm.

We first introduce some terminology. Let us consider the i-th iteration of the *loop* ($1 \leq i \leq N$). Note that the first iteration consists only of phases 3 and 4. 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 (U_1 is N). (2) P_i to be the real number of surviving processors, right before the iteration starts, i.e. right after phase 4 of the previous iteration (P_1 is P). (3) R_i to be the estimated number of surviving processors, that is the value of $c[1]$ right after phase 1 of the i-th iteration (R_1 is P). The following is shown by straightforward induction on the structure of the tree c.

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

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

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

Basis: For all subtrees of height 0 rooted at t, $c[t] \leq s(t)$, because some processors may have stop-failed after the initiation of phase 1, but before the initialization of the leaves of c tree.

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

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

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

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

In the dividing done during the dynamic top down traversal in W, we will allocate processors to tasks that either have not been completed, or have been completed, but not yet "accounted for" at the root $d[1]$. 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 in a balanced fashion to the *unaccounted* leaves, i.e. the leaves whose associated computed value in heap a is 0. The next lemma shows that the processors are divided evenly among the unaccounted leaves.

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.

Proof: This lemma is proven by defining an invariant for the phase 2 loop of the algorithm (see Appendix A.4 for procedure D_TD()). The algorithm of D_TD() is given in this proof using guarded commands to facilitate its partial correctness proof in the style of [D 76]. We, of course, cannot prove termination due to possible processor failures. We make use of the heap definitions, and the property that results from the dynamic bottom up traversal that if $d[j] \leq size$, where $size$ is the number of leaves in the subtree rooted at j, then $d[2j], d[2j+1] \leq size/2$ (the number of visited leaves is bounded by the number of leaves in the subtrees).

At the beginning of phase 2, R_i is the value $c[1]$, and U_i is the value $N - d[1]$. We define q as $\lceil R_i/U_i \rceil$. Because the top down traversal is executed synchronously by all surviving processors, and because all processors write identical values when utilizing concurrent write, we can use a sequential programming calculus for each processor participating in phase 2. We use the following invariant for the DO/**while** loop:

$$I : a[j] < size \wedge a[j] \leq d[j] \wedge (q-1)(size - a[j]) < c[j] \leq q(size - a[j]) \wedge 1 \leq pn \leq c[j]$$

This invariant is established by the assignments to j, $size$, and $a[1]$, performed in the state where $d[1] < N$ by the main loop termination condition, $1 \leq pn \leq c[i]$ by the processor enumeration in phase 1, and $(q-1)(N - d[1]) < c[1] \leq q(N - d[1])$ by the properties of ceiling.

We will also make use of the following abbreviation: $Q(j, s) : (q-1)(s - a[j]) < c[j] \leq q(s - a[j])$.

It is straightforward to verify that after I is established, it is left invariant by the DO/**while** loop. Some inference detail is ommitted in the proof outline below for readability.

```
{ d[1] < N  --by the main loop termination condition
  ∧1 ≤ pn ≤ c[j]  --by the processor enumeration of phase 1
  ∧(q − 1)(N − d[1]) < c[1] ≤ q(N − d[1])  --by the properties of ceiling  }
j,size,a[1] := 1,N,d[1];
{ I }
do size ≠ 1 →
    j1 := 2*j; j2 := j1 + 1;
    if d[j1]+d[j2] = 0 →  a[j1] := 0
    [] d[j1]+d[j2] ≠ 0 →  a[j1] := a[j]*d[j1] div(d[j1]+d[j2])
    fi
    { I ∧ (a[j1] ≤ d[j1] ∧ a[j1] ≤ size/2) }
    a[j2] := a[j] − a[j1];
    { I1 : I ∧ (a[j1] ≤ d[j1] ∧ a[j1] ≤ size/2)
      ∧(a[j2] ≤ d[j2] ∧ a[j2] ≤ size/2) ∧ (a[j] = a[j1] + a[j2]) }
    c[j1] := c[j]*(size/2−a[j1]) div(size−a[j]);
    { I2 : I1 ∧ Q(j1, size/2) }
    c[j2] := c[j] − c[j1];
    { I3 : I1 ∧ Q(j1, size/2) ∧ Q(j2, size/2) ∧ c[j] = c[j1] + c[j2] }
    if pn ≤ c[j1] →  { I3 ∧ pn ≤ c[j1] }{ I3 ∧ a[j1] ≠ size/2 } j := j1
    [] pn > c[j1] →  { I3 ∧ pn > c[j1] }{ I3 ∧ a[j2] ≠ size/2 } j := j2; pn := pn − c[j1]
    fi ;
    { I^size_size/2 }  --this is I with size/2 replacing size
    size := size div2;
    { I }
od ;
{ I ∧ size = 1 } { a[j] < 1 ∧ Q(j, 1) ∧ 1 ≤ pn ≤ c[j] }
```

When a processor completes dynamic top-down traversal, the invariant I holds, and $size = 1$. The assertion on the last of the outline above implies the following result:

$a[j] = 0 \wedge q{-}1 < c[j] \leq q \wedge 1 \leq pn \leq c[j]$

The safety property that each processor correctly constructs branches of the accounted tree using heap $a[1..2N-1]$ satisfying the constraints given in Section 3 follows directly from the proof outline by $a[1] = d[1]$ and for an interior node j, $a[j] = a[2j] + a[2j+1]$ with $a[j] \leq d[j]$ for all nodes j (the values of the a heap are computed once and not changed by the top down traversal).

Thus, if a processor completes phase 2 then it would reach a leaf with $a[j] = 0$, that is an unaccounted leaf, and at most $q = \lceil R_i/U_i \rceil$ processors would reach the same leaf. □

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

Lemma 4.3 In algorithm W, for all loop-iterations i we have: $U_i > U_{i+1}$, as long as at least one processor survives.

Proof: To prove this, we define $a_i[1..2N\text{-}1]$ and $d_i[1..2N\text{-}1]$ to be the values of trees a and d after the completion of iteration i. $d_0[1..2N\text{-}1]$ are the initial zero values. We first show that if an iteration $i+1$ is started with tree d satisfying $d_i[t] \leq d_i[2t] + d_i[2t+1]$ $(1 \leq t < N)$, then after the termination of loop-iteration $i+1$, tree d will satisfy $d_{i+1}[t] \leq d_{i+1}[2t] + d_{i+1}[2t+1]$ $(1 \leq t < N)$, and along a path completely traversed from leaf to root by a processor in phase 4: $a_i[t] < d_{i+1}[t]$ $(1 \leq t < 2N$, t along the path traversed).

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

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

Lemma 4.4 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: 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 Lemmas 4.1 and 4.3.

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

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

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

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 $S = O(N\log N + P\log^2 N)$, where N is the input array size, and the initial number of processors P is between 1 and N.

Proof: This immediately follows from Definition 2.1 and Lemmas 4.1-4.4. Note that although we assumed N processors in the algorithm W, we only used the fact that $P \leq N$ in the lemmas. In fact, as indicated in Section 3, we can accommodate $P < N$ processors by considering that $N - P$ processors failed prior to the beginning of the algorithm. This contributes a single charge of $O(N - P)$ to the cost, and does not distort the asymptotic result that consists of the product

of the total block-steps V_1 performed by the algorithm since the first iteration of the algorithm (inclusive) times the per-block-step cost of $O(\log N)$:

$$S = V_1 \times O(\log N) = (P_1 + U_1 + P_1 \log U_1) \times O(\log N) \quad \text{(using Lemma 4.4)}$$
$$= (P + N + P \log N) \times O(\log N) \quad \text{(using } P_1 = P \text{ and } U_1 = N\text{)}$$
$$= O(P \log N + N \log N + P \log^2 N) = O(N \log N + P \log^2 N) \quad . \square$$

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

We first observe that each block-step takes $\Theta(\log N)$ time and therefore each processor can be asked to perform $\Theta(\log N)$ processing steps in phase 3 without affecting the asymptotic complexity. To take advantage of this, we parametrize algorithm W as follows:

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

Thus, the cost of a single block-step C_B is $O(\log H + G) = O(\log H + N/H)$. By Lemma 4.4 the algorithm will verifiably visit all leaves of the progress heap after spending $V_1 = P_1 + U_1 + P_1 \log U_1 = P + H + P \log H$ block-steps. Therefore $S = V_1 \times C_B$, and so:

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

We would like to achieve $S = O(N)$. While the exact solution is involved, we observe that the following values for parameters G, H and P produce the desired result:

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

This results in the following "optimality" corollary:

Corollary 4.1 Parametrized algorithm W is a robust parallel algorithm that solves the *Write-All* problem of size N with $S = O(N)$, when $P \leq N/(\log^2 N - \log N \log\log N)$.

In the next section we show that no optimal N-processor algorithm exists for *Write-All.*

5 Lower Bounds

In this section we are going to show that for any algorithm that implements an N-processor robust solution to the *Write-All* problem a *failure pattern* can be constructed that will cause the algorithm to perform a *superlinear* number of processing steps. We also show that concurrent writes are necessary for the existence of robust algorithms. Concurrent writes are the source of redundancy in our apporach.

Appendix B contains some simple mathematical lemmas that are used in the theorem below.

Theorem 5.1 Given any N-processor CRCW PRAM algorithm that solves the *Write-All* problem, the adversary can force fail-stop errors that result in $\Omega(N\frac{\log N}{\log\log N})$ steps being performed by the algorithm.

Proof: We are going to present a strategy for the adversary that results in this worst case behavior. Let X be the best possible algorithm that implements a robust solution for the *Write-All* problem. Let $P_0 = N$ be the initial number of processors, and $U_0 = N$ be the initial number of unvisited array elements. The strategy of the adversary is outlined below. Step numbers refer to the PRAM steps (not to be confused with block-steps or loop-iterations used in Section 4).

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

To estimate the number of surviving processors and to express this mathematically, we will be using Lemma B.1 with the following definitions:

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

$P_1 \geq (1 - U_1/U_0)P_0$ or, after substituting for U_1 and using the properties of *floor*:

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

Step 2: The adversary again chooses among the U_1 remaining unvisited array elements $U_2 = \lfloor U_1/\log U_0 \rfloor$ elements with the least number of processors assigned to them. Using Lemma B.1 again in a similar way:

$$P_2 \geq (1 - \frac{\lfloor U_1/\log U_0 \rfloor}{U_1})P_1 = (1 - \frac{\lfloor \lfloor U_0/\log U_0 \rfloor/\log U_0 \rfloor}{\lfloor U_0/\log U_0 \rfloor})P_1$$

$$\geq (1 - \frac{\lfloor U_0/\log U_0 \rfloor/\log U_0}{\lfloor U_0/\log U_0 \rfloor})P_1 = (1 - \frac{1}{\log U_0})P_1 \geq (1 - \frac{1}{\log U_0})^2 P_0$$

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

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

This process is repeated for as long as there are any unvisited array elements. Let ρ be the step at which the last unvisited element is finally visited. Let us use Lemma B.2 with $G = \log N$ and

σ the largest integer such that $\sigma < \log N / \log\log N - 1$. Then $U_\sigma = \lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor/G\rfloor \ldots /G\rfloor}_{\sigma \text{ times}} > 0$, and since $U_\rho = 0$, ρ must be greater than σ because $U_\rho = 0$, and so:

$$\rho \geq \frac{\log U_0}{\log\log U_0} - 1 = \frac{\log N}{\log\log N} - 1 > \sigma$$

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

$$S = \sum_{i=0}^{\rho} P_i \geq \sum_{i=0}^{\rho} (1 - \frac{1}{\log N})^i P_0$$

Using the summation of a geometric progression we get:

$$S \geq P_0 \frac{1 - (1 - \frac{1}{\log N})^\rho}{1 - (1 - \frac{1}{\log N})} = P_0 \log N (1 - (1 - \frac{1}{\log N})^\rho) \geq P_0 \log N (1 - (1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}})$$

Using the result of Lemma B.3: $(1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}} = 1 - \frac{1}{\log\log N} + O(\frac{1}{(\log\log N)^2})$, we obtain the following lower bound on the number of PRAM processor steps:

$$S \geq P_0 \log N (1 - 1 + \frac{1}{\log\log N} + O(\frac{1}{(\log\log N)^2})) = P_0 \frac{\log N}{\log\log N} + O(P_0 \frac{\log N}{(\log\log N)^2})$$

And so: $S = \Omega(N \frac{\log N}{\log\log N})$. □

This result can immediately be utilized to produce a processor failure pattern for algorithm W that results in $S = \Omega(N \log^2 N / \log\log N)$. This is accomplished when the adversary fail-stops processors according to the strategy in the proof above, except that instead of PRAM steps, the adversary uses *block-steps*, and the processors are stopped only during the phase 3 where the operations on the actual data take place. This corresponds to the fixed per block-step *processor survival* coefficient k defined in Lemma 4.4 being equal to $1 - 1/\log N$.

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

We now show a difference between CRCW and CREW (concurrent read exclusive write) PRAMs.

Theorem 5.2 Given any N-processor CREW PRAM algorithm that solves the *Write-All* problem, the adversary can force fail-stop errors that result in $\Omega(N^2)$ steps being performed by the algorithm.

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

Let Y be the most efficient asymptotically CREW algorithm that solves the *Write-One* problem. Such an algorithm is no more efficient asymptotically than the algorithm below that utilizes an *oracle* to predict the best selection of a processor that is chosen to exclusively write the value 1:

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

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

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

We close this section with a comment on the knowledge of the adversary used in the proofs. The adversary fails processors based solely on their intent to write into the array to be initialized by the algorithm. The adversary uses no knowledge about the methods that the algorithms use in selecting the array elements to write into. Therefore the worst case behavior caused by such adversary will also apply to the cases where the algorithm uses a stronger PRAM model (such as *ideal* PRAM with arbitrarily powerful instruction set [BH 89]) or the cases where the algorithms makes probabilistic decisions (such as using a coin toss).

6 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. Actually, it is permissible for f to be computable in $O(\log N)$ sequential time without affecting the complexity of each block-step of algorithm W, and robust algorithms can take advantage of this property.

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

In order to adapt 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 N-processor algorithm for *general parallel array assignment* with $S = O(N \log N + P \log^2 N)$, $1 \leq P \leq N$. In phase 3 the assignment of 1 to $x[i]$ is replaced with the assignment shown above. Consecutive use of the *Write-All* technique in a single algorithm requires that the heaps initially contain zeros. This can be easily accomplished by utilizing three identical sets of the heaps. One is used in the actual algorithm, another set of heaps is used for zeroing the other two in a *Write-All* style, and the third set is saved for the next use of the zeroing clean-up step. This step does not affect the asymptotic efficiency, and we will assume that the heaps are appropriately cleared in the rest of this section. 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.

Proposition 6.1 There is a robust list ranking parallel algorithm with $S = O(\frac{N \log^2 N}{\log \log N} + \frac{P \log^3 N}{\log \log N})$, where N is the input list size, for the initial number of processors $P : 1 \leq P \leq N$.

Proof: The robust list ranking algorithm can be implemented by using a variation of the algorithm W and *robust parallel assignment* as its building blocks. We will first produce a naive version, and then apply an optimization. Below is a high level algorithm description.

Top level view of the algorithm:

forall processors PID=1..N **parbegin**
    for 1..log(N) **do**
        Perform a single stage during which we double each of the leaves' pointers at least once (each stage is a variation of algorithm W)
    od
parend

Here is a detailed view of a single stage, where phases 1 through 4 refer to the phases defined for algorithm W:

begin
    Phase 3: Each processor doubles its leaf's pointer
    Phase 4: Perform bottom up traversal to compute an underestimate of the number of leaves whose pointers were doubled at least once in phase 3
    while the underestimate of the visited leaves is not N **do**
        Phase 1: Perform bottom up traversal to enumerate remaining processors
        Phase 2: Perform top down traversal to reschedule work
        Phase 3: Each processor doubles its leaf's pointers (DoublePointer block)
        Phase 4: Perform bottom up traversal to measure progress made
    od
end

In phase 3 we use a *robust parallel assignment* approach to double pointers and update distances/ranks. The list nodes are stored in an array E[1..N] whose elements are defined in terms of the following records:

```
record
    bit integer v;  --current valid version number, index for P and D
    bit integer Q;  --boolean: was the end of list reached?
    integer D[0..1];  --distance traversed by the node so far
    integer P[0..1];  --node reached after traversing D links
end
```

Using the above definition, the pointer doubling operation for node k that is suitable for our robust algorithm becomes:

```
begin DoublePointers  --phase 3 robust assignment block
    local bit integer s, t;  --placeholders for tags
    local integer j;  --placeholder for index that k points to

    if not E[k].Q  --If end of list not reached
    then s := E[k].v  --Current valid tag for element k
        j := E[k].P[s]  --List element pointed to by k
        t := E[j].v  --Valid tag for element j
        E[k].D[s+1] := E[k].D[s] + E[j].D[t]  --New distance
        E[k].P[s+1] := E[j].P[t]  --Double pointer
        E[k].Q := E[j].Q  --[k] is done if P[k] is done
        E[k].v := s + 1  --One more step is done
    fi
end
```

The algorithm will perform the top level **for** loop $\log N$ times. Each phase will take at most $O(N \log N + P \log^2 N)$ processor steps (same as algorithm W). The overall number of steps is thus $S = O(N \log^2 N + P \log^3 N)$.

We now note, that the algorithm can be optimized by performing the pointer doubling operation up to $\log N$ times in phase 3 without affecting the complexity of the approach in algorithm W. This is because phases 1, 2 and 4 already take $\Theta(\log N)$ time. So, if in phase 3, we execute the DoublePointers block $\log N$ times, then the outmost **for** loop need only iterate $\log N / \log\log N$ times. With this modification, the final algorithm is characterized by $S = O(\frac{N \log^2 N}{\log\log N} + \frac{P \log^3 N}{\log\log N})$. □

Proposition 6.2 There is a robust parallel algorithm for computing the tree functions of [TV 84] with $S = O(\frac{N \log^2 N}{\log\log N} + \frac{P \log^3 N}{\log\log N})$, where N is the input tree size, for the initial number of processors $P : 1 \leq P \leq N$.

Proof: The technique of [TV 84] is used to number the vertices of a spanning tree of a graph in preorder in $O(\log n)$ time using $O(n + m)$ processors. This gives $O((n + m) \log n)$ total processing steps. This technique can be used to number the vertices in postorder, or compute other functions on trees.

The technique uses spanning tree adjacency lists to produce an Euler circuit that is used in determining the preorder numbering for the vertices.

The original algorithm is based on pointer doubling and parallel array assignments as basic building blocks. There are three pointer doubling/list ranking sub-algorithms and several constant time parallel array initializations.

Our robust technique is naturally applied to the constituent components of the original algorithm: algorithm W is used for simple assignments and other constant time operations, and the

robust pointer doubling algorithm of Proposition 6.1 is used for each of the three pointer doubling loops. Therefore, as in the case of list ranking, our robust version will use up to $P = O(n+m)$ processors with the size of the input N being $n+m$. Thus results in the overall $S = O(\frac{N \log^2 N}{\log \log N} + \frac{P \log^3 N}{\log \log N})$ number of steps. □

The algorithms discussed in the section so far are optimized for the worst case behavior in the presence of arbitrary fail-stop error patterns. The algorithms obtained using our technique incur a $O(\log^2 N)$ *multiplicative overhead* relative to the source algorithm. This multiplicative overhead remains even if there are no processor failures. However the robust list ranking algorithm can be easily modified to yield $S = O(N \log N)$ in the absence of failures making its asymptotic performance identical to the original pointer doubling algorithm. This is accomplished by preceding the algorithm with $\log N$ pointer doubling operations and a phase 4 bottom-up traversal. Thus we only accrue $O(N \log N)$ *additive* overhead in the absence of failures. Therefore, for the algorithms that are dominated by pointer doubling with *Parallel-time* × *Processors* cost measure of $\Theta(N \log N)$ (e.g. [TV 84]), there is no asymptotic degradation in the absence of failures. This optimization can also be applied to the sorting techniques of next proposition to reduce the overall multiplicative cost to just $O(\log N)$ over the $O(N \log^2 N)$ cost associated with sorting networks.

Proposition 6.3 There is a robust parallel bitonic sort with $S = O(N \log^3 N + P \log^4 N)$, where N is the number of integers to sort, for the initial number of processors $P : 1 \leq P \leq N$.

Proof: Bitonic sorting networks (also odd-even network, and other variations of such sorting techniques) can be presented using PRAM paradigm as two synchronous nested loops, each executed $\log N$ times by N processors to sort N integers in $O(\log^2 N)$ time. The exchange-shuffle operation is represented algorithmically as a fixed number of parallel array assignments.

This structure immediately suggests a robust approach using the framework developed above. The robust algorithm will also consist of two nested loops. The operation within the loops is again the adaptation of algorithm W to perform the exchange-shuffle operation.

Each processing stage within the inner while is a single application of algorithm W with $S_w = O(N \log^2 N)$. The two nested loops are executed $\log^2 N$ times. Thus $S = S_w \log^2 N = O(\log^4 N)$. □

7 On Constant Word Size

One observation is that algorithm W (by its definition) can be used to provide a robust parallel algorithm for an 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.

Proposition 7.1 Algorithm W can be adapted to provide a robust solution to the *Write-All* problem using $O(1)$ bit atomic writes so that there is preservation of: (1) $S = O(N \log N + P \log^2 N)$ used processor steps (counting $\log N$ bit write cycles as one time unit), and (2) $O(N)$ word space use (counting $\log N$ bits as one word) using P processors for $1 \leq P \leq N$.

Proof: We are going to show how to accomplish this by *simulating* word-atomic writes using bit-atomic writes.

This is done by implementing $\log N$-size words using 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. □

Since robustness for *Write-All* can be achieved with constant size words, same arguments apply to all the propositions on robustness in Section 6.

8 Discussion and open questions

We have formally defined and demonstrated the feasibility and wide applicability of robust parallel computation.

In Section 6 we showed how to generally apply the technique developed here to various parallel algorithms. It turns out that this technique can be applied to arbitrary PRAM algorithms. PRAM algorithms consist of synchronous parallel instruction steps. Each step executed by a processor is made of the following three sub-steps: (1) reading from shared memory to local memory, (2) performing some computation using local memory, and (3) writing from local memory to shared memory. Processors may be executing distinct programs. It has recently been proven in [KKPS 89] and [Shv] that each PRAM step can be efficiently simulated using the *Write-All* technique by storing the programs, local processor memory and registers in shared memory. When an optimal solution to *Write-All* in Corollary 4.1 is used as the basis of such simulation, this yields an *optimal* fault-tolerant execution of a parallel algorithm

We close this paper with some open questions. What are W's precise performance bounds? What extensions can be made to our robustness definitions, e.g., other types of faults? For example, it easy to modify algorithm W to work correctly and terminate in the presence of fail-stop failures followed by processor recoveries, however the algorithm efficiency can be severly affected by such failures.

What are the precise analogs of robustness in network models of parallel computation? For example, this could involve application of the work on distributed network controllers such as [AAPS 87]. What about randomized robust parallel computation? Could a formalization along the lines of [LGFG 86] be used?

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.

9 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.

[**BH 87**] P. Beame and J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM", *Journal of the ACM*, vol. 36, no. 2, pp. 643-670, 1989.

[**D 76**] E. W. Dijkstra, *The Discipline of Programming*, Prentice-Hall, 1976.

[**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.

[GKP 89] R. Graham, D. Knuth and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, 1989.

[KKPS 89] Z. Kedem, S. Khuler, K. Palem, P. Spirakis, unpublished manuscript, 1989.

[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.

[SS 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.

[Shv] A. A. Shvartsman, "Fault tolerant parallel computation", forthcoming Ph.D. thesis, Brown University.

[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.

Appendix A

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 bottom 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 assignment "x[k]:=1", and the progress is measured in procedure via a dynamic bottom up traversal in D_BU(). (Parameter passing is by reference in all cases.)

```
forall processors PID=1..N
parbegin
    shared integer array
        x[1..N], --input array
        c[1..2N-1], --processor counts
        cs[1..2N-1], --count step numbers
        d[1..2N-1], --progress/done tree
        a[1..2N-1]; --accounted tree
    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(integer PID,  --processor id
               integer step,  --timestamp
               integer pn)  --processor number
    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(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(integer pn  --enumerated processor number
               integer k)  --data item
    shared integer array
        c[1..2N-1],  --processor counts
        d[1..2N-1];  --progress/done tree
        a[1..2N-1];  --accounted tree
    local integer j, j1, j2;  --current/left/right indices
    j := 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
        j1 := 2*j; j2 := j1 + 1;  --left/right indices
        --compute accounted node values
        if d[j1]+d[j2] = 0
        then a[j1] := 0
        else a[j1] := a[j]*d[j1] div(d[j1]+d[j2])
        fi ;
        a[j2] := a[j]-a[j1];
        --processor alloc. to left/right sub-trees
        c[2*j] := c[j]*((size/2)-a[2*j]) div(size-a[j])
        c[2*j+1] := c[j]-c[2*j];
        --go left/right based on proc. no.
        if pn ≤ c[2*j]
        then j := 2*j  --go left
        else j := 2*j + 1  --go right
             pn := pn-c[2*j]
        fi ;
        size := size div2  --half of leaves visible
    od ;
    k := j - (N-1)  --assign processor based on j
end ;
```

Appendix B

Lemma B.1 Given a sorted list of m ($m > 1$) nonnegative integers $a_1, a_2, \ldots, a_m$ then we have that for all j ($1 \leq j < m$):

$$(1 - \frac{j}{m}) \sum_{i=1}^{m} a_i \leq \sum_{i=j+1}^{m} a_i.$$

Proof: We proceed by induction on m. Base case is trivial for $j = 1 < 2 = m$. Using inductive hypothesis $(1-\frac{j}{k}) \sum_{i=1}^{k} a_i \leq \sum_{i=j+1}^{k} a_i$ we show that $(1-\frac{j}{k+1}) \sum_{i=1}^{k+1} a_i \leq \sum_{i=j+1}^{k+1} a_i$ by extending the sorted list of k elements by the new element a_{k+1} in the following straightforward transformations:

$a_i \leq a_{k+1}$ for any i, $1 \leq i \leq k$

$\sum_{i=1}^{k+1} a_i \leq (k+1) a_{k+1}$

$\frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq j a_{k+1}$ for each j, $1 \leq j \leq k$

$-j a_{k+1} + \frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq 0$

$k a_{k+1} - j a_{k+1} + \frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq k a_{k+1}$ by adding $k a_{k+1}$ to both sides

$(k-j) a_{k+1} + \frac{j}{k+1} \sum_{i=1}^{k+1} a_i \leq k a_{k+1}$

$(1 - \frac{j}{k}) \sum_{i=1}^{k} a_i + (1 - \frac{j}{k}) a_{k+1} + \frac{j}{k(k+1)} \sum_{i=1}^{k+1} a_i \leq a_{k+1} + \sum_{i=j+1}^{k} a_i$ by dividing both sides by positive k and using the inductive hypohtesis.

$(1 - \frac{j}{k}) \sum_{i=1}^{k+1} a_i + j(\frac{1}{k} - \frac{1}{k+1}) \sum_{i=1}^{k+1} a_i \leq \sum_{i=j+1}^{k+1} a_i$ after grouping terms, finally

$(1 - \frac{j}{k+1}) \sum_{i=1}^{k+1} a_i \leq \sum_{i=j+1}^{k+1} a_i$ after simplifying the inequality.

□

Lemma B.2 Given $G > 1, N > G$, and integer σ such that $\sigma < \frac{\log N}{\log G} - 1$ then the following inequality holds:

$$\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor /G\rfloor \ldots /G\rfloor}_{\sigma \text{ times}} > 0 \qquad (*)$$

(where σ is the number of times that the expression in the left hand side of the inequality contains division by G)

Proof: To show $(*)$, it suffices to show that, after dropping one floor and strengthening the inequality: $(\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor /G\rfloor \ldots /G\rfloor}_{\sigma-1 \text{ times}} /G) - 1 > 0$, or that $\lfloor \ldots \lfloor \lfloor N \underbrace{/G\rfloor /G\rfloor \ldots /G\rfloor}_{\sigma-1 \text{ times}} > G$.

Applying such transformations for $\sigma - 1$ more steps, we get that it suffices to show: $N > G^{\sigma} + G^{\sigma-1} + \ldots + G$, or $N > \frac{G^{\sigma+1}-G}{G-1}$ using summation for geometric progressions.

We observe that $G^{\sigma+1} > \frac{G^{\sigma+1}-G}{G-1}$ thus it is enough to show that $N > G^{\sigma+1}$. After taking logarithms of both sides we get $\log N > (\sigma+1) \log G$, and so is suffices to have $\sigma < \frac{\log N}{\log G} - 1$.

□

To achieve the results in Section 5, we will make use of this lemma with $G = \log N$ resulting in $\sigma < \frac{\log N}{\log \log N} - 1$.

Lemma B.3 For $N \to \infty$:

$$(1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}} = 1 - \frac{1}{\log\log N} + O(\frac{1}{(\log\log N)^2}).$$

Proof: The proof is done using standard techniques for manipulating asymptotics (e.g. [GKP 89]).

Let $A = (1 - \frac{1}{\log N})^{\frac{\log N}{\log\log N}}$, and $B = \log(1 - \frac{1}{\log N})$. Using $\log(1 - z) = -z - \frac{z^2}{2} - \frac{z^3}{3} - \ldots$ expansion around 0, we get $B = -\frac{1}{\log N} - \frac{1}{2\log^2 N} - \frac{1}{3\log^3 N} - \ldots$ for $N \to \infty$.

From this we get:

$\log A = \frac{1}{\log\log N} - \frac{1}{2\log N\log\log N} - \frac{1}{3\log^2 N\log\log N} - lots$ *(lower order terms)*, or that

$A = \exp(-\frac{1}{\log\log N} - \frac{1}{2\log N\log\log N} - \frac{1}{3\log^2 N\log\log N} - lots)$

Using $\exp(z) = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \ldots$ expansion around 0 and after multiplying the resulting series and gathering the terms, we get:

$A = 1 - \frac{1}{\log\log N} + \frac{1}{2(\log\log N)^2} - lots.$

□