Efficient Robust Parallel Computations* (Extended Abstract)

Zvi M. Kedem † Krishna V. Palem ‡ Paul G. Spirakis §

Abstract

A parallel computing system becomes increasingly prone to failure as the number of processing elements in it increases. In this paper, we describe a completely general strategy that takes an *arbitrary* step of an ideal CRCW PRAM and *automatically* translates it to run efficiently and robustly on a PRAM in which processors are prone to failure. The strategy relies on efficient robust algorithms for solving a core problem, the *Certified Write-All Problem*. This problem characterizes the core of robustness, because, as we show, its complexity is equal to that of any general strategy for realizing robustness in the model. We analyze the expected parallel time and work of various algorithms for solving this problem. Our results are a non-trivial generalization of Brent's Lemma. We consider the case where the number of the available processors decreases *dynamically* over time, whereas Brent's Lemma is only applicable in the case where the processor availability pattern is *static*.

*This research was partially supported by the Office of Naval Research under contract number N00014-85-K-0046, by the National Science Foundation under grant number CCR-89-6949, and by the EEC ESPRIT Basic Research Actions Project ALCOM (No 3075).

†Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, (212) 998-3101, kedem@nyu.edu.

‡IBM Research Division, T.J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, (914) 789-7846, kpalem@ibm.com.

§Computer Technology Institute, Patras University, Patras, P. O. Box 1122, 26110 Patras, +30 (61) 225-073, spirakis@graptvx1.bitnet

1 Introduction

With hardware becoming cheaper, it is expected that "massively parallel systems" with large numbers of processors will both increase the speed of computations and decrease their cost. Unfortunately, as the number of processing elements grows, certain difficulties need to be addressed, among them *processor faults* and *asynchrony*. Clearly, the larger the number of processors, the greater the probability of some processors failing or going out of synchrony. These two problems are related, but in this paper we restrict ourselves to processor faults.

Much of the recent emphasis in algorithm design for PRAMs for a variety of basic problems such as list ranking [TV84,CV86], sorting [Co86], forest matching [KP88], pattern matching [KLP89] and others, has emphasized efficiency and optimal speedup. These extremely efficient (ideally optimal speedup) parallel algorithms have very little "slack" in that every step of the algorithm is essential. Therefore, quite often, they do not terminate correctly when perturbed by a few processor failures.

Because of this, it is important to study the design of *robust* parallel algorithms, and even more important, *robust execution of arbitrary parallel programs*. Obviously it is important to ensure that the overhead incurred in realizing robustness is not excessive. In this paper we present a general methodology for *implementing ideal CRCW PRAMs robustly on faulty CRCW PRAMs*. We adopt the approach of

"graceful degradation," so that if the processors fail-stop during the computation, as long as at least one processor remains operational, the PRAM program, *independent of its semantics*, can continue executing correctly. In addition, we want to minimize the performance penalty incurred in such robust implementations.

Specifically, we present a technique that takes an *arbitrary* step of an ideal CRCW PRAM designed to run on U "virtual" processors, and execute it on a CRCW PRAM of the same type with $P \leq U$ processors that are prone to failure[1]. The number of virtual processors U, that is the parallelism width of the ideal PRAM program being translated, can vary from step to step. By modeling these failures probabilistically, we analyze the expected *work* and *parallel time* complexities of deterministic schemes for realizing these robust implementations. We also analyze a probabilistic algorithm for solving the same problem.

Kanellakis and Shvartsman [KS89] were the first to formalize this notion of robustness, in the formal context of synchronous parallel computation. They developed a failure model (to be described later), which we use here. In [KS89], they considered the *Write-All* problem, and described its deterministic robust implementation. In the Write-All problem, given an array $x[1..U]$ initialized to 0, set $x[i] := 1$ for all i. (We will also refer to such writing of 1 as "marking.") Their algorithm, henceforth referred to as the *KS-algorithm*, iteratively estimates the amount of remaining unwritten locations and remaining live processors, and reschedules the processors to other locations for writing 1's. (It is easy to see that their algorithm for the Write-All problem can be used for computing associative functions, such as *max*.)

Using the Write-All algorithm, Kanellakis and Shvartsman, designed robust algorithms for fundamental problems such as list ranking. In [KS89] they also analyze the deterministic worst case complexity of the KS-algorithm. They observed various specific complexity improvements in the case where, for the number of initial processors P, $P < U$. Specifically, they parameterized the KS-algorithm to achieve work of $O(U \log U + P \log^2 U)$. In [KS89] they observe that this algorithm achieves robustness optimally (work $O(U)$) provided $P \leq U/\log^2 U - \log U \log\log U)$. Work improvement for the case where $P = U/\log U$ was observed independently by Khuller [Kh89].

[1] That is we execute an ideal Common/Arbitrary/Priority CRCW PRAM program respectively on a Common/Arbitrary/Priority faulty CRCW PRAM.

Subsequently to Kanellakis and Shvartsman, Martel, Park, and Subramonian [MPS89] described a probabilistic algorithm for computing the maximum of U elements assuming asynchronous computations, in which the processors can fail. Their algorithm (to which we refer to as a *Multiple Coupon Collector Algorithm* or *MCC-algorithm* for short) can be immediately used for robust computation of the maximum as well as for solving the Certified Write-All problem. The Certified Write-All problem is the Write-All problem with the additional requirement that a global bit is set after all the locations have been written[2]. A variety of an asynchronous parallel model (APRAM) was described by Cole and Zajicek [CZ89], who presented algorithms for summation, graph connectivity, and other problems on this model. Many researchers have studied closely related issues, including robustness and fault tolerance in an asynchronous setting [Aw88,AAG87,AS88,DPPU86,Pi85,SS83].

Previous efforts at "adapting" ideal PRAM programs to cope with imperfections, such as processor faults or asynchrony, have focussed on redesigning algorithms for specific well-known problems such as those listed above. We depart completely from this approach, by providing a general strategy for simulating arbitrary PRAM steps on PRAMs with faults.

Specifically, we show that the complexities of solving the Certified Write-All problem robustly, and implementing a step of an ideal CRCW PRAM robustly are identical up to small constant multiplicative time and small constant per processor additive space overheads. We do this constructively, by showing (in sections 4 and 5) how to use an arbitrary robust Certified Write-All algorithm twice to implement an arbitrary step of the ideal PRAM. In effect we provide a *two-phase idempotent execution strategy* (*TIES* for short) that uses *any* robust algorithm for solving the Certified Write-All problem[3], to automatically yield robust implementations of arbitrary ideal PRAM steps. (Recently [Sh89], Shvartsman obtained a technique, similar to our TIES strategy, with the following two restrictions. He can implement a parallel algorithm robustly provided that its work is

[2] The KS-algorithm, as well as the algorithm due to Martel et al. provide such certification implicitly.

[3] Actually, *any* algorithm that can verify that all the the given U instructions have been "touched" (or executed), will suffice. The Certified Write-All problem is a special case of this more general *Certified Touch-All problem*. Note for example that the computation of conjunction, maximum, or summation could have been used also.

within a polylog factor of that of the best sequential algorithm and its local memory requirements are within a polylog factor of the problem size.)

Since we show (in section 5) the Certified Write-All problem to be the core step in robust implementations of PRAM algorithms, solving it efficiently is critical to realizing robust implementations with low overhead. Towards this end, we also introduce a new and extremely simple algorithm based on pointer-doubling (the *PS-algorithm* where PS stands for pointer shortcutting) for solving the Certified Write-All problem. We analyze the expected *work* as well as the expected parallel *time* taken by this simple PS-algorithm algorithm, as well by the KS-algorithm. We show that the expected behavior of the PS-algorithm is better than that of the KS-algorithm; detailed statements of these results are in section 3. We also show that the expected *work* done by a *randomized* parallel version of a Coupon Collector scheme (originally introduced in [MPS89]) is less than that of the deterministic KS and PS algorithms.

2 Model of failures and some conventions

Although our results hold with nonessential modifications for different varieties of CRCW PRAMs, for the purpose of this paper, we assume the Common variant of the CRCW PRAM [FW78]. When dealing with probabilistic algorithms, we assume that the individual processors of the PRAM are supplied with independent random number generators.

Consider a parallel algorithm A that starts at time 0 with P available processors and in parallel time τ completes its computation on input data I_A. The *availability pattern* Π_A indicates which processors are available or operational for the algorithm A at each parallel time step. Let P_i be the number of processors available at time i $(0 \leq i \leq \tau)$. Then,

Definition 1 *The work performed in the execution of algorithm A on input I_A, with availability pattern Π is $W(A, I_A, \Pi) = \sum_{i=0}^{\tau} P_i$.*[4]

[4] This notion of work generalizes the traditional processor-time product definition, for the case when there are no processor failures. It was originally introduced as *available processor steps* in [KS89]. A similar definition of work is introduced subsequently in [MPS89], although they define it in the slightly different but essentially equivalent model of asynchronous PRAMs.

As in the case of Kanellakis and Shvartsman [KS89], we will only be interested in the *fail-stop* availability patterns. In such patterns, processors either stay available or *fail* and once they fail, they stay failed for the remaining steps of the computation.

An availability pattern Π is called *oblivious* if it is determined before the start of the execution of the algorithm. It is called *Byzantine* if it is created (by an adversary), adaptively, during the computation. In such a case, the adversary determines the set of available processors at the next time step by examining the history of the computation (which includes random choices previously done) up to that time.

The average case analysis in this paper deals with *random failures* where each processor may fail with a fixed probability $q < 1$ in a consecutive sequence of time steps, referred to here as an *epoch*; the length of each epoch may be a function of U. In this paper, unless stated otherwise, the epoch length is $\Theta(\log U)$. If all the processors in the PRAM fail, this condition can presumably be detected externally. For convenience of presentation (and also to adhere closer to the model in [KS89]), we assume that at least one processor survives, no matter what are the probabilistic failure assumptions. This, of course, is nonessential.

3 Summary of Results

We now summarize our main contributions. All our expected case results listed below hold with an extremely high probability of at least $1 - U^{-\gamma}$ for some $\gamma > 1$ controllable by the implementor.

3.1 The two-phase execution strategy

1. We introduce a two-phase idempotent execution strategy that, in conjunction with *any* robust algorithm for the Certified Write-All problem can be used to simulate an arbitrary step of a PRAM robustly. This TIES strategy can be implemented using a constant amount of additional space per live processor.

3.2 The KS-algorithm

2. Average case analysis of the KS-algorithm for the Certified Write-All problem. We show that:

(a) Its expected work is $O((P+U)\log U)$, where U is the array size and P is the initial number of processors.

(b) When $P = U$ and the processor failure rate q is small, its expected parallel time is $O(\log P \log U)$.

(c) When $P \leq \lambda U$ for some constant $\lambda \in (0,1)$, its expected parallel time is $\Theta((U + \log P)\log U)$.

(d) When $P = \lambda U/\phi(U)$ for some $\lambda \in (0,1)$ and an increasing function $\phi(U) \leq \log U$ and $q < 1$ where q is the decay rate for an epoch of length $\Theta(\phi(U)\log U)$, its expected parallel time is $\Omega(U^{\epsilon})$ for some $\epsilon > 0$.

Note that the KS-algorithm has an interesting threshold in its expected parallel time, which degrades very rapidly from $O(\log P \log U)$ (case (b) above) to *essentially sequential behavior* (case (c)) even when the number of initial processors P is $O(U)$. In the above analysis as well as in the average case analysis of the PS-algorithm below, we assume a random pattern of failures as stated in section 2, where the failures can be Byzantine within each epoch.

3.3 The PS-algorithm

3. We present a new and simple PS-algorithm for the Certified Write-All problem which is based on pointer doubling. We show that:

   (a) When $P = U$, its expected work is $O(U \log U)$.

   (b) When $P = U$, its expected parallel time is is $O(\log U)$.

   (c) When $\lambda U \leq P \leq U$ there exists $q^* \in (0,1)$ such that if $q < q^*$, its expected parallel time is $\Theta(\log U)$.

   (d) When $P = \lambda U/\phi(U)$ for some constant λ and an increasing function $\phi(U) \leq \log U$ then there exists $q^* \in (0,1)$ such that if $q < q*$ is the failure rate for an epoch of length $\Theta(\phi(U)\log U)$, its expected parallel time is $O(\phi(U)\log U)$.

   The PS-algorithm does better in its expected parallel time behavior than the KS-algorithm.

3.4 A related result

4. We analyze a probabilistic algorithm for the Certified Write-All problem. In this analysis, we assume worst-case oblivious availability patterns of processors. We can prove that this *Adapted Coupon Collector Algorithm* (or *ACC-algorithm* sketched in section 8.2) does $O(U \log\log U)$ work provided $P \leq U/\log U$. This algorithm is a trivial variant of the one studied by Martel et al. [MPS89] for computing associative functions in parallel and asynchronous situations that are fault-prone. Therefore, its expected work improves on that of the two deterministic algorithms described above provided $P \leq U/\log U$.

4 Two-phase idempotent execution strategy

4.1 Introductory remarks

A naive attempt to emulate an ideal PRAM on a faulty PRAM based on a simple redistribution of the live processors, will run into difficulties. Consider for instance an ideal program written for 2 processors whose task is to exchange the values of the variables $v[1]$ and $v[2]$. p_1 and p_2 in parallel read $v[1]$ and $v[2]$ respectively and write the value they read into the other location. If, say p_1 fails, the value of $v[1]$ is irretrievably lost. We therefore will need to use temporary storage for the values to be written, so that the inputs for the step remain available as long as necessary to simulate the step of the PRAM. There are however, additional subtle points to be taken care of.

As stated, we will use a single execution of any Certified Write-All algorithm as the basic building block in our strategy to simulate a *single step* of the ideal PRAM on the faulty PRAM. In general, such algorithm assumes the the existence of some data structure, *suitably initialized*. The purpose of that structure is to monitor the progress of the algorithm, e.g., to estimate the number of live processors, the work done so far, etc. Thus at the end of the execution of the algorithm, in general, the data structure has a different value. If we now wish to run the algorithm again to simulate another step, we again need a clean, initialized copy of the data structure. Providing a new data structure for each step of the ideal PRAM program is clearly impractical, as this will require a very large number of initialized data

structures.

The obvious approach, is to to reinitialize the original data structure. Here, again, the simplistic approach will not be sufficient, as we have to do it in a faulty PRAM. Assume that, for instance, that the initialized data structure has 0s in all positions. Then, initializing the data structure is a complementary problem to Certified Write-All. To do this, we need another data structure, properly initialized, etc., and we have got ourselves into a circular argument. These are the types of problems we need to consider in our robust implementations.

4.2 Model of the ideal PRAM

Without loss of generality, our ideal CRCW PRAM is a simple modification of the RAM as described in [AHU74]. To save space, we omit many details. The PRAM has U "virtual" processors: $p_1, \ldots, p_U$ with *no local* registers; all memory is *global* and *shared*. The "application memory" accessible by the processors is a shared vector $M[1..last]$ of some length. Without loss of generality, the processors execute a single program whose instructions are listed in a vector $INST[1..I]$. The instructions are as those in [AHU74], with appropriate modifications. Thus, a typical instruction might be: $M[j_1] := M[j_1]+M[j_2]$. There is also a vector $PC[1..U]$, initialized to 1, containing the program counters of the processors. In a single step of the PRAM, each processor p_i executes the following

Internal Program:

1. read $PC[i]$;
2. **if** $PC[i] = 0$ **then**
3. halt
4. **else**
5. **begin**
6. read $INST[PC[i]]$;
7. decode the instruction;
8. execute the instruction;
9. write the new value of $PC[i]$
10. **end.**

4.3 The strategy

We assume the existence of some robust algorithm for the Certified Write-All problem, which given a vector $x[1..U]$ initialized to 0 and an availability pattern Π, sets $x[i] := 1$ for $i = 1, \ldots, U$ in τ steps. It uses some auxiliary data structure AUX of size $O(U)$, appropriately initialized, *all* of whose locations are accessed during the execution of the algorithm. Furthermore, at the end of its execution, a bit variable $DONE$ is set to $TRUE$.

To save on space, our description will be somewhat informal, and we will not utilize the resources most efficiently.

Our (faulty) CRCW PRAM, will contain structures required both by the original application problem and by the robust interpreter. Thus we will have:

- M, $INST$, and PC of the original application program.
- Two versions of x, AUX, and $DONE$, referred to as $x.old$, $x.new$, $AUX.old$, $AUX.new$, $DONE.old$, $DONE.new$.
- Three new vectors $LOC[1..U]$, $VAL[1..U]$, and $NEXT[1..U]$.
- The interpreter (based on the given Certified Write-All algorithm), its program counters etc.

We now sketch, very briefly, the simulation of one step of the ideal PRAM while omitting many important technical details. By informal induction we assume:

- The values of M, $INST$, PC in the faulty PRAM are the same as those of M, $INST$, PC in the ideal PRAM.
- The value of $AUX.old$ at the beginning of the simulation is the same as that of AUX at the beginning of the execution of the robust algorithm for Certified Write-All, $x.old$ is initialized to 0s, and $DONE.old$ is $FALSE$.

The interpreter proceeds in two phases, each based on a single execution of the robust Certified Write-All algorithm. By the algorithm, the (live) processors are synchronized between the phases. The behavior of the interpreter is reminiscent of the deferred write approach in the *redo/no-undo* recovery protocol in database operating systems. Phase 1, in effect, creates the deferred writes, phase 2 installs them. It is important that the phases are idempotent, as processors may fail after doing some work, and the various structures may become inconsistent. We describe the two phases (not in complete detail) in turn.

4.3.1 Phase 1

The first phase of the interpreter is a modification of the Certified Write-All algorithm. However, instead of addressing x, AUX, and $DONE$ it addresses $x.old$, $AUX.old$, and $DONE.old$. Furthermore, in order to prepare clean structure for the next phase: before it accesses some location of $x.old$, it initializes (to 0) the corresponding location of $x.new$; before it accesses some location of $AUX.old$, it initializes the corresponding location of $AUX.new$; before it accesses $DONE.old$, it initializes (to $FALSE$) $DONE.new$. There are other book-keeping activities to be done, resetting the program counters of the interpreter at the end of the phase, etc.

We now describe the modifications done to the Certified Write-All algorithm to allow the simulation of the application program. Run the underlying robust Certified Write-All algorithm. Just before the point where p_k executes $x[i] := 1$, simulate the execution of $INST[PC.old[i]]$. However, if $INST[PC.old[i]]$ assigns v to $M[l]$, *instead* execute $LOC[i] := l$, $VAL[i] := v$; if the instruction does not write, set $LOC[i] := 0$. $PC[i]$ is not overwritten, its new value is stored in $NEXT[i]$.

4.3.2 Phase 2

This phase starts after $DONE.old$ becomes $TRUE$. The second phase too is a modification of the Certified Write-All algorithm. However instead of addressing x, AUX, and $DONE$ it addresses $x.new$, $AUX.new$, and $DONE.new$. Furthermore, in order to prepare clean structure for the next phase: before it accesses some location of $x.new$, it initializes (to 0) the corresponding location of $x.old$; before it accesses some $AUX.new$, it initializes the corresponding location of $AUX.old$; before it accesses $DONE.new$, it initializes (to $FALSE$) $DONE.old$.

Again, using the robust Certified Write-All algorithm, values computed in phase are copied into correct locations: if $LOC[i] \neq 0$ then $M[LOC[i]] := VAL[i]$. Furthermore, $PC[i] := NEXT[i]$ is executed This is done for the index i before the instruction $x.new[i] := 1$ is executed.

5 TIES as a generalization of Brent's Lemma

To understand the full generality of TIES, consider a parallel computation that takes Π steps. Let the available number of processors on step i be Π_i. In Brent's case[B74], this sequence of available processors is determined completely and deterministically at the *beginning* of the computation (and was generally a constant). Therefore, we can apply his theorem at any step i immediately, by dividing up the U_i wide parallel step into pieces of size (U_i/Π_i), and distributing them among the available processors[5]. (Of course, even in this simple case, a "two-phase" strategy is needed for applying Brent's Lemma to PRAMs, first by using temporary storage to store the values being computed, and then copying them back into the original positions. For a trivial example, consider cyclic shift on an array of length U_i, similar to the example in section 4.1.)

The situation in this paper constitutes a non-trivial generalization of this problem since the availability patterns change with time, and we do not know exactly how these changes occur. Therefore, processors need to be dynamically rescheduled to ensure that all the instructions have been executed and in addition, this condition needs to be detected (certification)[6]. Furthermore, since this process of *repeatedly* rescheduling and certification is done on the same faulty PRAM, this algorithm for the Certified Write-All problem has to be robustly implemented as well. The TIES strategy does this as characterized in the theorem below.

Theorem 1 *Assume that we are given a robust Certified Write-All algorithm A for writing 1s in an array $x[1..U]$. Let it be stored in κ locations of the PRAM M and use auxiliary storage of size $\sigma(U)$, all of which locations are accessed during the execution. Furthermore, for any failure pattern π, the execution time is some $\tau(\pi)$.*

Then, given an arbitrary ideal PRAM program using U (virtual) processors running in T steps, and failure patterns $\pi_1, \pi_2, \ldots, \pi_t$ (π_i is the failure in step i), it is possible to execute the program robustly on M in time $\sum_{i=1}^{T} \tau(\pi_i)$ using $O(\kappa+\sigma(U)+U)$ additional storage.

[5] Without loss of generality, we assume that U_i is a multiple of Π_i.

[6] If M is an asynchronous PRAM rather than one prone to processor failures, then certification is sufficient and processor rescheduling is not necessary

6 Average-case analysis of the KS-algorithm

For the description of this algorithm, please see [KS89].

Let $j_0, j_1, \ldots$ be the time instants at which processors actually write into the array x; these steps will be referred to as *writing epochs* or *epochs*. We use U_j to denote the number of unwritten locations at the beginning of step j. Our availability patterns Π are *random* in the sense that each processor, which is operational at the end of writing epoch j_x, may fail with fixed probability $q < 1$ between the epochs j_x and j_{x+1}. Π is byzantine otherwise. Let $P_0 = P$ be the number of processors initially available, and let $U_0 = U$. We will first prove the following useful technical theorem.

Theorem 2 *There are constants $\epsilon_1, \epsilon_2 \in (0,1)$, and $a > 1$ that depend on q, such that, as long as $P_{j_x} \geq a \log P$, the inequality $\epsilon_1 \leq P_{j_{x+1}}/P_{j_x} \leq \epsilon_2$ holds for all such epochs j_x with probability at least $1 - P^{-\gamma}$ for some $\gamma > 2$.*

Proof Given an epoch j_x such that the number of operational processors in the beginning of j_x is $P_{j_x} \geq a \log P$, and any $\beta \in (0,1)$, we have from Chernoff bounds [Ch52] that if E_x is the event "the number of failed processors is in the interval $(1 \pm \beta)qP_{j_x}$," then $\text{Prob}\{E_x\} \geq 1 - \exp(-\frac{\beta^2}{2} qP_{j_x})$. Choose any $a > \frac{6}{\beta^2 q}$ and let $\gamma = \frac{a\beta^2 q}{2} - 1$ ($\gamma > 2$). Then, $\text{Prob}\{E_x\} \geq 1 - P^{-(\gamma+1)}$.

Now, let E be the event that E_x holds for all epochs j_x such that $P_{j_x} \geq a \log P$. Then, if $\bar{E}$ is the event "there is a j_x such that E is violated," we get $\text{Prob}\{\bar{E}\} \leq \sum \text{Prob}\{\bar{E}_x\}$ over all j_x for which $P_{j_x} \geq a \log P$, i.e. $\text{Prob}\{\bar{E}\} \leq U_0 P^{-(\gamma+1)} \leq P^{-\gamma}$. Hence, $\text{Prob}\{E\} \geq 1 - P^{-\gamma}$. This completes the proof of the Theorem with constants $\epsilon_1 = 1-(1+\beta)q$, $\epsilon_2 = 1 - (1-\beta)q$. □

For convenient analysis, we will view the execution of algorithm KS as being split into two phases. The first phase contains all epochs j_x such that $P_{j_x} \geq a \log P$ and the second phase is the rest of the algorithm's execution.

6.1 Analysis of the first phase

For the first phase we have:

Lemma 3 *Conditioned on event E, the number of epochs of the first phase is $O(\log P)$.*

Proof We have $P_{j_{x+1}} \leq \epsilon_2 P_{j_x}$ for all the epochs of the first phase. If the number of such epochs is e_1, then $P_{j_{e_1+1}} < a \log P$, i.e. $P\epsilon_2^{(e_1+1)} \leq a \log P$, i.e. $e_1 = O(\log P)$. □

Corollary 4 *Conditioned on event E, the amount of work for the first phase is $W_1 = O(P \log U)$.*

Corollary 5 *Conditioned on event E, the number of parallel steps of algorithm KS for the first phase is $T_1 = O(\log P \log U)$.*

6.2 Analysis of the second phase

To analyze the second phase of algorithm KS we need the following result of [KS89]:

Fact 1 *Let P_i and U_i respectively denote the number of operating processors and the number of unwritten positions at the beginning of epoch i. Then, the work from this epoch is $O((P_i + U_i + P_i \log U_i) \log U)$.*

By using this fact and Theorem 2, we get:

Lemma 6 *Conditioned on event E, the work in the second phase of algorithm KS is $W_2 = O((a \log P + U + a \log P \log U) \log U)$.*

Having estimated the work during the second phase, we now analyze its expected parallel time. To do this, we need to calculate the number of unwritten positions $U_{j_{e_1}}$ at the end of the first phase. Again, conditioning on the event E we get that $U_{j_{e_1}} \geq U - \frac{P}{1-\epsilon_2}$. We distinguish two cases depending on the value of P:

6.2.1 Case 1: $P = U$ and $q < 1/3$

Lemma 7 *Conditioned on event E, for any epoch j_x of the first phase $U_{j_x} \leq P_{j_x}$.*

Proof Conditioned on event E, we have $\frac{P_{j_1}}{P_{j_0}} \geq \epsilon_1$ (where $P_{j_0} = P_0$). Let $r_1 = \frac{P_{j_1}}{P_{j_0}}$. By an inequality of [KS89] we have $U_{j_1} \leq U_{j_0}(1 - \frac{r_1}{2})$. Thus, $U_{j_1} \leq U(1 - \frac{\epsilon_1}{2})$. Also, $P_{j_1} \geq \epsilon_1 P \geq \epsilon_1 U$. Thus, $U_{j_1} \leq P_{j_1}$ when $1 - \frac{\epsilon_1}{2} \geq \epsilon_1$, i.e. when $\epsilon_1 \geq 2/3$, i.e. when $q < 1/3$. By repeating the above argument (for $q < 1/3$) the Lemma is proved. □

By Lemma 7 and given the event E, we get $U_{j_{e_1}} = O(\log P)$, for which we have:

Lemma 8 *If $U \leq P$ and $q < 1/3$, then with probability at least $1 - P^{-\gamma}$ for some $\gamma > 2$, the number of parallel steps of the second phase of algorithm KS is $T_2 = O(\log P \log U)$ and hence its parallel time is $O(\log P \log U)$ with the same probability.*

6.3 $P < U$

We consider two cases:

1. $\frac{P}{1-\epsilon_2} \leq \lambda U$, $\lambda \in (0,1)$
   Then, given that E holds, we get $U_{j_{e_1}} \geq (1-\lambda)U$ and $P_{j_{e_1}+1} < a \log P$. In this case, the number of epochs of the second phase is $\Theta(U)$ since the processors are very few and the number of unwritten positions is very large. Thus

   Lemma 9 *If there is a constant $\lambda \in (0,1)$ such that $P \leq \lambda(1-\epsilon_2)U$ (ϵ_2 as in Theorem 2) then, with probability at least $1 - P^{-\gamma}$ for some $\gamma > 2$, the parallel time of the KS-algorithm is $\Theta(U \log U)$.*

2. $P = \lambda U/\phi(U)$, $\lambda \in (0,1)$, increasing $\phi(U) \leq \log U$
   Here q is the decay rate in an epoch of length $\Theta(\phi(U) \log U)$. Then,

   Lemma 10 *If there is a constant $\lambda \in (0,1)$ and an increasing function $\phi(U) \leq \log U$ such that $P = \lambda U/\phi(U)$, then for every failure rate $q < 1$ on an epoch of length $\Theta(\phi(U) \log U)$, with probability at least $1 - U^{-\gamma}$ for some $\gamma > 2$, the parallel time of the KS-algorithm is $T = \Omega(U^{\epsilon})$ for some $\epsilon > 0$.*

6.4 The expected work and parallel time of the KS-algorithm

From Corollary 5 and Lemma 6, the claimed result 2(a) follows. From Corollary 4, Lemma 7, and Lemma 8, the claimed result 2(b) follows. From Lemma 9, the claimed result 2(c) follows. From Lemma 10 the claimed result 2(d) follows.

7 The PS-algorithm and its average case analysis

7.1 The algorithm

We propose a simple algorithm for Certified Write-All with certification that exhibits stable expected parallel time behavior, and has low expected work complexity. The PS-algorithm is based on a trivial modification of the well-known straightforward pointer-doubling algorithm. Our auxiliary data structure AUX is an array $c[1..U]$. Initially, $c[i] = 0$ for all i. Furthermore, *DONE* is initialized to *FALSE*. It will be convenient, to define addition in the set $\{1, \ldots, n\}$ with 1 following n. We denote such addition by $\hat{+}$, thus $a \hat{+} b = (a + b - 1 \bmod n) + 1$.

Algorithm Pointer-Shortcutting:

- *Processor assignment:* For each k $(1 \leq k \leq P)$ processor k is assigned to array position $i_k = (k-1)\frac{U}{P} + 1$.
- Each processor k executes the following loop:

```
1.  while not DONE do
2.        begin
3.        if c[i_k ^+ c[i_k]] = 0 then
4.          begin
5.          x[i_k ^+ c[i_k]] := 1;
6.          c[i_k] := c[i_k] + 1
7.          end
8.        else c[i_k] := c[i_k] + c[i_k ^+ c[i_k]];
9.        if c[i_k] ≥ n then
10.         DONE := TRUE
11.       end.
```

Lemma 11 *(Correctness) If at least one processor survives, all of $x[1..U]$ will be set to 1.* □

7.2 Average case analysis

In the sequel we will assume that the initial number of processors is $P \geq U/\log U$. We will assume an availability pattern Π that is *random*. As before, we split the parallel time into consecutive epochs. Each processor which is operational at the beginning of an epoch has a fixed probability $q < 1$ of failing at some step during it, independently of other processors. Notice that the availability pattern assumed here is equivalent to that assumed in the average case analysis of the KS-algorithm in the sense that the processor "decay rate" is constant, measured over an epoch. Patterns with more frequent failure (faster decay rates) than Π are unacceptable because they tend to exhaust the available processors before any useful work can be done. We analyze the complexity of the PS-algorithm by considering several cases.

Assuming that the choice of the step in it fails is random, we can assume that for each live processor the probability f of failing in the next step is equal to q divided by epoch length. In fact, this value of f is a nonessential approximation.

7.2.1 $P = U$

To give a more presentable analysis, we look at a variant of the PS algorithm. Let $c[1..2U]$ be a vector initialized by $c[k] = k+1$ for $k < 2U$ and $c[2U] = 2U$. There are $2U$ processors, one per location and each processor k executes the loop:

1. **while** $c[k] < 2U$ **do**
2. $\quad c[k] := c[c[k]]$
3. **end.**

For the purpose of the analysis, it is therefore enough to estimate the time when for at least one k, $1 \leq k \leq U$, $c[k] = 2U$, so that a consecutive segment of length U at least has been "shortcut."

Observe that any stage of the algorithm, for each k there is a path starting with k and ending with $2U$, defined by the sequence: $k, c[k], c[c[k]], \ldots$. The algorithm starts with a path $k, k+1, k+2, \ldots, 2U$ for each k. (In fact, we have a single path $1, 2, \ldots, 2U$, which we choose to consider as $2U$ overlapping paths.) During the execution, because of shortcutting, the paths may split. In effect, the algorithm attempts to perform tree path compression starting with a tree consisting of a single long path from the leaf (1) to the root ($2U$).

Let $L_j[k]$ be the path from k to $2U$ after j steps of the algorithm. Let $l_j[k]$ denote its length and $a_j[k]$ the number of live processors on it. Of course, $l_j[k] \leq l_{j-1}[k]$. Without loss of generality, we assume that the processors fail between PRAM steps, and not in the middle of a step. The numbers above refer to values immediately after the step was executed. Also, without loss of generality, we can assume that the two processors at locations $2U-1$ and $2U$ are dead before the start of the execution, and that once $c[k] = 2U$, the processor p_k dies, for each k.

Lemma 12 *Let $k < 2U$, and let $c_1[k], c_2[k], \ldots, c_z[k]$ be a sequence such that $a_j[k] \geq c_j[k] l_j[k]$ for $j = 1, 2, \ldots, k$. Then $l_z[k] \leq l_0 / \prod_{j=0}^{z}(1 + c_j[k])$.*

Proof We first show that for any j, $l_j[k] + a_j[k] = l_{j-1}[k]$. Let i be a node in $L_j[k]$, which therefore was also in $L_{j-1}[k]$. If i was dead during step j, then $c[i]$ did not change during step j. If i was alive during step j, then before step j, $c[i] = l$ for some l, and after step j, $c[i] = c[l]$; thus l was removed from $L_{j-1}[k]$ while creating $L_j[k]$. The equation follows from considering all nodes in $L_j[k]$, and the lemma follows immediately from the equation. □

To prove the desired complexity bound, it is essentially enough to show that for at least one k, such that $1 \leq k \leq U$, $l_j[k]$ becomes 2 after $O(\log U)$ steps. The proof will consider two phases for each path. The first phase when the path is still at least $\Omega(\log U)$ long, the second phase, when it is shorter. (Of course, all derivations will be done under high probability assumptions.)

Lemma 13 *There exist constants $\zeta, \eta > 0$ such that $l_{\zeta \log U}[k] \leq \eta \log U$ with probability at least $1 - U^{-\gamma}$ for some $\gamma > 2$, for $k = 1, 2, \ldots, U$.*

Proof We will prove the claim for all values of k, so we fix some k, and do not list it in the variables. Thus we write L_j for $L_j[k]$, etc.

Consider any step $j \geq 1$. Let π_j be the probability that a random node on L_j is alive. It is possible to show that $\pi_j = (1 - q/\log U)^j \pi_0$. Technically, $\pi_0 \geq 1 - 2/U$, but we will write $\pi_0 = 1$. Then $\pi_j = (1 - q/\log U)^j$. Let $c_j = \pi_j/2$. Define the event E_j by "$a_j \geq c_j l_j$." Then from Chernoff bounds, $\text{Prob}\{E_j\} \geq 1 - \exp(-\pi_j l_j/8)$.

Using elementary analysis it easy to show that there exists ζ such that $\prod_{j=1}^{\zeta \log U}(1 + c_j) \geq 2U/\log U$, and therefore from Lemma 12, $l_{\zeta \log U} \leq \log U$. We now have to show that for some $\eta > 0$, $l_{\zeta \log U} \leq \eta \log U$ holds with high probability.

Let $\eta = 24/\exp(-q\zeta)$. Consider the event E defined by: "as long as $j \leq \zeta \log U$ and $l_j \geq \eta \log U$: $a_j \geq c_j l_j$." $\text{Prob}\{\bar{E}\} \leq \sum_{j=1}^{\zeta \log U} \exp(-\pi_j l_j/8) \leq \zeta \log U \cdot \exp(-(1-q/\log U)^{\zeta \log U} \eta \log U/8) \leq \zeta \log U \cdot U^{-\exp(-q\zeta)\eta/8} = \zeta \log U \cdot U^{-3}$. □

We have shown that $l_{\zeta \log U}[k] \leq \eta \log U$ with probability $1 - U^{-\gamma}$ at least. We observe that for a constant fraction of locations k the processor p_k will stay alive for $(\zeta + \eta) \log U$ steps. Then as for such k, $l_{j+1}[k] < l_j[k]$ as long as $l_j[k] > 2$, the algorithm will terminate in at most $(\zeta + \eta) \log U$ steps.

Theorem 14 *The expected parallel time of the PS-algorithm, with U processors initially is $O(\log U)$. The probability that the actual parallel time exceeds $\Theta(\log U)$ is less than $U^{-\gamma}$, for some $\gamma > 2$.*

7.2.2 $P < U$

We modify the algorithm so that the processors are placed $\approx U/P$ locations apart.

1. $\lambda \leq P < U$, $\lambda \in (0,1)$

   Lemma 15 *If there is a constant $\lambda \in (0,1)$ such that $\lambda U \leq P < U$, then there exists $q^* \in (0,1)$ such that if $q < q^*$ (and $f = q/\log U$), then with some probability at least $1 - P^{\gamma}$ for some $\gamma > 2$, the parallel time of the PS-algorithm is $O(\log U)$.*

2. $P = \lambda U/\phi(U)$, $\lambda \in (0,1)$, increasing $\phi(U) \leq \log U$

   Here q is the decay rate in an epoch of length $\Theta(\phi(U)\log U)$. Then,

   Lemma 16 *If there is a constant $\lambda \in (0,1)$ and an increasing function $\phi(U) \leq \log U$ such that $P = \lambda U/\phi(U)$, then there exists $q^* \in (0,1)$ such that for every failure rate $q < q^*$ on an epoch of length $\Theta(\phi(U)\log U)$ ($f = q/\phi(U)\log U$), with probability at least $1 - U^{-\gamma}$ for some $\gamma > 2$ the parallel time of the PS-algorithm is $T = O(\phi(U)\log U)$.*

   Proof (Sketch.) Informally, we consider the algorithm to have two phases. During the first phase, the (live processors) are attempting to "hook-up," during the second phase the standard PS-algorithm for the case $P = U$ is running. Of course, there is no such rigid division; some processors may be attempting to hook-up, while other may already be shortcutting.

   Let α be a constant to be fixed later. The probability of a particular processor staying alive for $\alpha\phi(U)\log U$ steps is $s = (1 - q/\phi(U)\log U)^{\alpha\phi(U)\log U} \geq \exp(-q\alpha)$. The probability that a consecutive sequence of $\alpha\lambda\log U$ processors has died by step $\alpha\phi(U)\log U$ is $\leq U^{\alpha\lambda\log(1-s)}$. Thus the probability that at time $\alpha\phi(U)\log U$ there exist two live processors separated by more than $\alpha\phi(U)\log U$ cells occupied by dead processors or empty is $\leq U^{1+\alpha\lambda\log(1-s)}$. This holds uniformly throughout the array. (We ignore some small initial and final segments of the array.) Then it is easy to fix α such that $1 + \alpha\lambda\log(1-s) < -2$. □

7.3 The expected work and parallel time of the PS-algorithm

From Theorem 14, the claimed results 3(a) and 3(b) follow. From Lemma 15, the claimed result 3(c) follows. From Lemma 16, the claimed result 3(d) follows.

8 The benefit of randomization

Both the MCC algorithm and its extension, the ACC algorithm are essentially those described by Martel et al. in [MPS89]. We consider them here since the ACC algorithm does less expected work than any of the other algorithms considered in the earlier sections for the Certified Write-All problem, and it is extremely simple and clean to implement.

8.1 The MCC-algorithm

Informally, the MCC algorithm views the locations in the array x of the Certified Write-All problem as the U leaves of a binary tree that is $\log U$ deep. The MCC-algorithm proceeds as follows. Initially all tree nodes are *unmarked*. We start with $P = U$ processors. Each live processor selects a tree node at random. If the node v is a leaf or if the children of the node are marked, then node v is also marked. This step is repeated by all the live processors until the root is marked. Note that marking the root is the same as certifying that all the locations of x have been written. A simple variant of the analysis in [MPS89] shows that this algorithm does $O(U\log U)$ expected work.

8.2 The ACC-algorithm

We start with $P \leq U/\log U$ processors and divide the array $x[1..U]$ to be marked into P subarrays each of size U/P. (Without loss of generality, we will assume that U and P are powers of two.) Each of these subarrays is now treated as as a single "chunk" and is associated with the leaf of a full binary tree of $(2P-1)$ nodes. The ACC-algorithm involves running the MCC-algorithm on the new tree, where now marking a leaf of the tree implies setting $x[i] := 1$ for all positions of the corresponding subarray.

Surprisingly, this simple modification to the MCC-algorithm only does an expected work of $O(U\log\log U)$ with as many as $U/\log U$ processors.

Theorem 17 *The expected work done by the ACC-algorithm to solve the Certified Write-All problem with $P \leq U/\log U$ processors is $O(U \log\log U)$.*

To the best of our knowledge, at the time of writing this paper, this is the strongest provably correct bound on the expected work done by the ACC-algorithm.

9 Acknowledgements

We are grateful to Samir Khuller for pointing out an omission in the first version of the TIES, for bringing reference [MPS89] to our attention, and for several helpful discussions. We are also grateful to Paris Kanellakis for bringing the issue of robustness in synchronous parallel computing to our attention.

10 References

[Aw88] B. Awerbuch, "On the effects of feedback in dynamic network protocols," *Proc. 29th IEEE FOCS*, pp. 231-242, 1988.

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni, "Applying static network protocols to dynamic networks," *Proc. 28th IEEE FOCS*, pp. 358-370, 1987.

[AAPS87] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks, "Local management of a global resource in a communication network," *Proc. 28th IEEE FOCS*, pp. 347-357, 1987.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman, "The design and analysis of computer algorithms," Addison-Wesley, 1974.

[AS88] B. Awerbuch and M. Sipser, "Dynamic networks are as fast as static networks," *Proc. 29th IEEE FOCS*, pp. 206-219, 1988.

[B74] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *JACM*, vol 21, no. 3, pp. 201-206, 1974.

[Ch52] H. Chernoff, "A measure of asymptotic efficiency for test of a hypothesis based on the sum of observations," *Annals of Math. Stat.*, vol. 23, pp. 493-509, 1952.

[Co86] R. Cole, "Parallel merge sort," *Proc. 27th IEEE FOCS*, pp. 511-516, 1986.

[CV86] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with application to list, tree, and graph problems," *Proc. 27th IEEE FOCS*, pp. 468-491, 1986.

[CZ89] R. Cole and O. Zajicek, "The APRAM: incorporating asynchrony into the PRAM model," *Proc. 89'SPAA*, pp. 170-178, 1989.

[DPPU86] C. Dwork, D. Peleg, N. Pippinger, and E. Upfal, "Fault tolerance in networks of bounded degree," *Proc. 18th ACM STOC*, pp. 370-379, 1986.

[FW78] S. Fortune and J. Wyllie, "Parallelism in random access machines," *Proc. 10th ACM STOC*, pp. 114-118, 1978.

[Kh89] S. Khuller, private communication, June 1989.

[KLP89] Z. Kedem, G. Landau, and K. Palem, "Optimal parallel suffix-prefix matching algorithm and applications," *Proc. 89'SPAA*, pp. 388-391, 1989.

[KP88] Z. Kedem and K. Palem, " Optimal parallel algorithms for forest and term matching," *to appear in Theoretical Computer Science.*

[KS89] P. Kanellakis and A. Shvartsman, "Efficient parallel algorithms can be made robust," *Tech. Rep. CS-89-35*, Brown Univ., pp. 1-28, October 24, 1989. (Initial version appeared in *Proc. 8th ACM PODC*, pp. 211-222, 1989.)

[MPS89] C. Martel, A. Park, and R. Subramonian, "Fast asynchronous algorithms for shared memory parallel computers," *Tech. Rep. CSE-89-8*, Univ. of California – Davis, pp. 1-17, July 25, 1989.

[Pi85] N. Pippinger, "On networks of noisy gates," *Proc. 26th IEEE FOCS*, pp. 30-38, 1985.

[SS83] R. Schlichting and F. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222-238, 1983.

[Sh89] A. Shvartsman, "Achieving optimal CRCW fault-tolerance," *Tech. Rep. CS-89-49*, Brown Univ., pp. 1-8, December 22, 1989,

[TV84] R. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *Proc. 25th IEEE FOCS*, pp. 12-22, 1984.