Parallel Processing Letters
© World Scientific Publishing Company

FAILURE-SENSITIVE ANALYSIS OF PARALLEL ALGORITHMS WITH CONTROLLED MEMORY ACCESS CONCURRENCY[*]

CHRYSSIS GEORGIOU
Dept. of Computer Science, University of Cyprus,
75 Kallipoleos Str., CY-1678 Nicosia, Cyprus
chryssis@ucy.ac.cy

ALEXANDER RUSSELL and ALEXANDER A. SHVARTSMAN
Computer Science and Engineering, University of Connecticut,
371 Fairfield Rd., Unit 2155, Storrs, CT 06269, USA
{acr,aas}@cse.uconn.edu

Received (6/6/2003)
Revised (1/6/2005)
Communicated by (Paul Spirakis)

ABSTRACT
The abstract problem of using P failure-prone processors to cooperatively update all locations of an N-element shared array is called *Write-All*. Solutions to *Write-All* can be used iteratively to construct efficient simulations of PRAM algorithms on failure-prone PRAMs. Such use of *Write-All* in simulations is abstracted in terms of the *iterative Write-All* problem. The efficiency of the algorithmic solutions for *Write-All* and *iterative Write-All* is measured in terms of *work complexity* where all processing steps taken by the processors are counted. This paper considers determinitic solutions for the *Write-All* and *iterative Write-All* problems in the fail-stop synchronous CRCW PRAM model where *memory access concurrency* needs to be controlled. A deterministic algorithm of Kanellakis, Michailidis, and Shvartsman [16] efficiently solves the *Write-All* problem in this model, while controlling read and write memory access concurrency. However it was not shown how the number of processor failures f affects the work efficiency of the algorithm. The results herein give a new analysis of the algorithm [16] that obtain *failure-sensitive* work bounds, while retaining the known memory access concurrency bounds. Specifically, the new result expresses the work bound as a function of N, P *and* f. Another contribution in this paper is the new failure-sensitive analysis for *iterative Write-All* with controlled memory access concurrency. This result yields tighter bounds on work (vs. [16]) for simulations of PRAM algorithms on fail-stop PRAMs.

Keywords: Parallel algorithms, work complexity, fault-tolerance, algorithm simulations, memory access concurrency.

1. Introduction

The Parallel Random Access Machine, or PRAM, has served as the target model for numerous synchronous shared-memory parallel algorithms [8,19,15]. The PRAM model provides a convenient abstraction that combines the simplicity of the RAM model with the power of parallelism — this makes the PRAM easy to "program" using a high-level notation. There is ongoing research [1,28] developing hardware platforms that can be used to efficiently execute algorithms expressed in PRAM-like programming languages. However, PRAM makes assumptions that, given the current state of technology, make it difficult for it to be implemented as a scalable

*This research is supported by the NSF Grants 9988304 and 0311368, and NSF ITR Grant 0121277. The work of the second author is supported in part by the NSF CAREER Award 0093065. The work of the third author is supported in part by the NSF CAREER Award 9984774. The work of the first author was performed in part, while at the University of Connecticut.

parallel architecture. The model assumes that the processors are synchronous, that shared memory can be concurrently accessed by arbitrary number of processors, and that the processors are completely reliable. Several approaches have been developed in an attempt to deal with this by providing a sufficiently high-level programming model while weakening the PRAM assumptions or by using closely-related alternative models (e.g, [23,11,18,6,22,25,27,10]). Some approaches provide algorithmic simulations of PRAM algorithms on other platforms. It has been shown that solutions for a particular problem, called *Write-All*, can be used as iteratively in constructing such simulations (e.g., [7,22,26]). The *Write-All* problem [17] is defined as follows: *Given a N-element array and P processors, set each element of the array to* 1.

Write-All captures the essence of the computational progress that can be naturally accomplished in unit time by a PRAM where $P = N$. Here the storing of values in the shared array models constant-time computation that can be performed by the individual processors. The iterative use of *Write-All* in simulations of parallel algorithms on "imperfect" platforms led to the formulation of the *iterative Write-All* problem [13]: *Given a sequence of r shared arrays of size N and P initial processors, write the value* 1 *into all $r \cdot N$ locations, under the restriction that each location of the ith array is set to* 1 *before any location of the $(i+1)$st array is written.*

Obtaining efficient solutions for *Write-All* and *iterative Write-All* becomes very challenging in the presence of failures, or in the absence of synchrony, or without concurrent memory access; e.g., [4,18,3,9,20,24,5]. The efficiency of *Write-All* algorithm is assessed in terms of the *work* complexity that accounts for all steps taken by the processors during the computation [17]. Optimal *Write-All* solutions have work $O(N)$ and optimal *iterative Write-All* solutions have work $O(r \cdot N)$ for r iterations, while solutions having polylogarithmic (in N) multiplicative overhead are considered to be efficient.

Background. We consider deterministic synchronous systems, where processors are subject to stop-failures, and where memory access concurrency must be either controlled or eliminated. Note that for certain failure patterns with up to $P - 1$ failures, the work must be *quadratic* if processors are not allowed to access certain memory cells concurrently; cf. the *Write-One* problem and the $\Omega(P^2)$ lower bound for CREW (concurrent-read, exclusive-write) machines [18]. Thus in the presence of failures and in the absence of concurrency, parallel computation can be extremely inefficient. This is not surprising: redundancy is necessary for achieving fault-tolerance and concurrent memory access provides redundancy, e.g., when several processors write a value to the same shared memory location then the value is written correctly even if only one processor completes the write. If concurrency must be allowed to achieve efficiency, then it is interesting to understand whether concurrent memory access can be controlled in the presence of failures.

Consider a step of a parallel computation, where a particular location m is written by p processors. Then $p - 1$ of these writes are potentially "redundant", because a single write suffices. When considering the COMMON CRCW PRAM model where all writers write identical values when accessing the same memory location, then a single write indeed suffices. Thus in this work we measure "concurrency" as the number of redundant memory accesses, and we measure concurrency separately for reads and writes. Note that the EREW (exclusive-read, exclusive-write) model has memory access concurrency of 0, while a single step of a P-processor CRCW PRAM can have concurrency as high as $P-1$.

It was shown by Kanellakis, Michailidis and Shvartsman in [16] that it is indeed possible to construct efficient fault-tolerant algorithms such that when concurrency is required to tolerate failures, the number of concurrent accesses to memory locations increases gracefully as the number f of stop-failures increases. In particular it was shown that in these algorithms, at any time, at most one processor accesses any particular shared memory cell in the absence of failures, i.e., such algorithms can be executed on EREW (exclusive-read, exclusive-write) machines without any need for concurrent memory access when $f=0$.

The algorithms in [16] are quite involved and require a very careful analysis. The authors performed a thorough analysis of the concurrency required by the algorithms. However, the analysis of the work complexity is very conservative: work is assessed for the worst case of stop-failures in the range $0 \leq f < P$, as a function of P and N alone. That is, [16] did not show how the number of failures f affects the upper bound on work of these algorithms.

An impossibility result [17] established that, when $P = N$, no optimal solution is possible for *Write-All* for $f = \omega(P/\log\log P)$. The main algorithm in [16] achieves optimality when substantial processor "slackness" is assumed, i.e., when P is substantially smaller than N. Note that since the analysis in [16] is not sensitive to f, the optimal range of processors is necessarily conservative, because it is given for the worst case of failures without identifying f.

Contributions. We derive the first *failure-sensitive* bounds on work for deterministic *Write-All* and *iterative Write-All* algorithms in the natural setting where *memory access concurrency must be controlled*. The target model of computation is the CRCW PRAM where processors are subject to arbitrary patterns of stop-failures. We use our approach [13] for analyzing work-performing algorithms by separately assessing the costs of tolerating failures and the costs of achieving perfect load balancing. We give a new failure-sensitive analysis of algorithm KMS[a] from [16], and we refine its range of optimality. We then use algorithm KMS to establish new failure-sensitive bounds on work for the *iterative Write-All* problem, for synchronous shared-memory systems, while simultaneously bounding memory access concurrency. This result yields tighter bounds on work (vs. [16]) for simulations of PRAM algorithms on fail-stop PRAMs.

We let *Write-All*(N, P, f) stand for the *Write-All* problem for an array of size N, P processors ($P \leq N$), and up to f stop-failures ($0 \leq f < P$). We let r-*Write-All*(N, P, f) be the iterative problem of using P processors to solve r instances of N-size *Write-All* by "solving one instance at a time". Note that if W is the work of *Write-All*(N, P, f), then $O(r \cdot W)$ gives an immediate upper bound for r-*Write-All*(N, P, f), however we show that a much tighter upper bound can be derived.

Recall that by memory *access concurrency* we mean the total number of redundant memory accesses. We let $rc(N, P, f)$ stand for the worst case total read concurrency, and $wc(N, P, f)$ stand for the worst case total write concurrency, for a terminating computation in the presence of f failures. We now state our results.

[a]Kanellakis, Michailidis and Shvartsman call this algorithm $W^{opt}_{CR/W}$ in [16].

1. The *Write-All*(N, P, f) problem can be solved with write concurrency $wc(N, P, f) \le f$, and read concurrency $rc(N, P, f) \le 7f \log N$ [16]. We give a failure-sensitive analysis and show that this can be done with work W_1:

$$\begin{array}{lll}(a) & W_1 = O(N + P\log^2 N \log^2 P/\log\log P) & \text{when } f > P/\log P, \\ (b) & W_1 = O(N + P\log^2 N \log^2 P/\log(P/f)) & \text{when } f \le P/\log P.\end{array} \tag{1}$$

We also show that algorithm KMS is optimal if $P = O(N \log\log N/\log^4 N)$, when $f > P/\log P$, and if $P = O(N \log(N/f)/\log^4 N)$, when $f \le P/\log P$. The latter case improves the result in [16], where optimality is shown only for $P = O(N \log\log N/\log^4 N)$.

2. We show that the iterative r-*Write-All*(N, P, f) problem can be solved with write concurrency $wc(N, P, f) \le f$, and read concurrency $rc(N, P, f) \le 7f \log N$. Our failure-sensitive analysis shows that this can be done with work W_r:

$$\begin{array}{lll}(a) & W_r = O(r \cdot (N + P\log^2 N \log^2 P/\log\log P)) & \text{when } f > Pr/\log P, \\ (b) & W_r = O(r \cdot (N + P\log^2 N \log^2 P/\log(Pr/f))) & \text{when } f \le Pr/\log P.\end{array} \tag{2}$$

Note that our bounds for W_r in (2) are asymptotically better than those obtained by computing the product of r and the (non-iterated) *Write-All* bounds W_1 in (1).

3. Let A be any N-processor, r-time EREW PRAM algorithm. We show that A can be simulated on a P-processor CRCW PRAM subject to up to f stop-failures with write concurrency $wc(N, P, f) \le f$, read concurrency $rc(N, P, f) \le 7f \log N$, and with work W_r as in (2).

Finally we note that the constants hidden in the $O(\cdot)$ are not large, however the precise analysis requires restating the analysis in [16] in substantial detail.

Related work. *Write-All* algorithms can be used iteratively to simulate parallel algorithms formulated for synchronous failure-free processors, in deterministic and probabilistic settings, e.g., [22,24,25,26]. This commonly requires that (*i*) the individual processor steps are made idempotent (since they may have to be performed multiple times), and that (*ii*) a linear-size (in P) auxiliary memory is made available (to be used to store intermediate results). While the former can be solved with the help of an automated tool, e.g., a compiler, the latter requires sophisticated solutions because of the difficulty of (re)using the auxiliary memory due to "late writers" (slow processors that unknowingly write stale values to memory). Examples of randomized solutions addressing these problems include [23,2,20]. Another approach to simulations uses an optimistic approach, where the computation proceeds for several steps assuming that all tasks assigned to active processors are successfully completed, e.g., [21]. In some deterministic models optimal simulations are possible (cf. [26]), however randomized solutions are able to achieve optimality (with high probability) for broader ranges of models and algorithms. An example of a practical implementation is discussed in [7].

Document structure. The rest of the paper is structured as follows. In Section 2 we give models and definitions. In Section 3 we review algorithm KMS and in Section 4 we present its new failure-sensitive analysis. We also show the new analysis for *iterative Write-All* and the new improved analysis on PRAM simulations. We conclude in Section 5. A preliminary version of this paper appeared as [14].

2. Models and Definitions

In this section we define the model of computation, the *Write-All* problems, and the efficiency measures of memory access concurrency and work.

Parallel setting. We use as the basis the CRCW PRAM where all concurrently writing processors write the same value (COMMON CRCW). There are P initial processors with unique identifiers (PID) in the range $1, \dots, P$. Each processor knows its PID, P, and the input size N. *Shared memory* is accessible to all processors and each memory access takes unit time. Each processor also has a constant size local memory. Each memory cell can store $\Theta(\log \max\{N, P\})$ bits. The input is stored in N cells in shared memory and the rest of the shared memory is initially cleared (contains zeros).

Model of failures. We extend the basic parallel model with a failure model. We assume the *fail-stop* processor model, where a processor may stop at any moment during the computation and once stopped it does not restart. Shared memory writes are *atomic* with respect to failures: failures can occur before or after a write, but not during the write. We let an omniscient *adversary* impose failures on the system, and we use the term *failure pattern* to denote the set of the events, i.e., processor stop-failures, caused by the adversary. The only restriction on the adversary is that at least one processor must remain operational. For a failure pattern F, we define the *size* f of the failure pattern as $f = |F|$ (the number of failures). Our *failure model* is the set of all failure patterns F, such that $|F| < P$.

***Write-All* problems.** We define the *Write-All* problem as follows:

> *Write-All: Using P fail-stop processors write the value 1 into all locations of a shared array of size N.*

We let *Write-All*(N, P, f) stand for the *Write-All* problem, for a shared array of size N, P processors ($P \leq N$), and any pattern F of stop-failures such that $|F| \leq f < P$. We define the *iterative Write-All* problem as follows:

> *Iterative Write-All: Given a sequence of r shared arrays of size N each, write the value 1 into all $r \cdot N$ locations using P fail-stop processors, under the restriction that each location of the ith array is set to 1 before any location of the $(i+1)$st array is written.*

We let r-*Write-All*(N, P, f) denote the *iterative Write-All* problem, for a sequence of r shared arrays of size N each, P processors ($P \leq N$), and any fail-stop pattern F such that $|F| \leq f < P$.

Measures of efficiency. We are interested in studying the complexity of *Write-All* algorithms measured as *work* (or *available processor steps* [18]), and their read and write concurrency.

For a computation subject to a failure pattern F, denote by $P_i(F)$ the number of processors completing an instruction in step i of the computation.

Definition 1 *Given a problem and a P-processor algorithm that solves its instance of size N for a failure pattern F, with $|F| \leq f$, by time step $\tau(F)$, then the work complexity W of the algorithm is: $W = W_{N,P,f} = \max_{|F| \leq f} \left\{ \sum_{i=1}^{\tau(F)} P_i(F) \right\}$.*

We now define the read and write concurrency measures [16] that assess the worst case number of "redundant" reads and writes.

Definition 2 *Given a problem and a P-processor algorithm that solves its instance of size N for a failure pattern F, with $|F| \le f$, by time step $\tau(F)$, if at time i $(1 \le i \le \tau(F))$, $P_i^{\mathrm{r}}(F)$ processors complete reads from $N_i^{\mathrm{r}}(F)$ distinct locations and $P_i^{\mathrm{w}}(F)$ processors complete writes to $N_i^{\mathrm{w}}(F)$ distinct locations, then we define:*
(i) read concurrency rc as: $rc(N,P,f) = \max\limits_{|F|\le f} \left\{ \sum_{i=1}^{\tau(F)} \left(P_i^{\mathrm{r}}(F) - N_i^{\mathrm{r}}(F)\right) \right\}$ *and*
(ii) write concurrency wc as: $wc(N,P,f) = \max\limits_{|F|\le f} \left\{ \sum_{i=1}^{\tau(F)} \left(P_i^{\mathrm{w}}(F) - N_i^{\mathrm{w}}(F)\right) \right\}$.

3. Algorithm KMS

Algorithm KMS [16] solves the *Write-All*(N,P,f) problem, for any $f < P$. In this section we give a description of the algorithm (to avoid a complete restatement, we refer the reader to [16] for details). The algorithm consists of two layers, where the top layer provides the overall control structure for solving *Write-All* and the bottom layer is responsible for controlling memory access concurrency. The top layer control structure is described in Section 3.1. The bottom layer provides specific access routines for reading from, and writing to, the shared memory; this is presented, following [16], in Sections 3.2 and 3.3.

Algorithm KMS uses several data structures represented as binary trees. (1) The *progress tree* records the progress of the computation and it is used to balance processor loads in a divide-and-conquer fashion. (2) The *processor enumeration tree* is used to estimate the number of operational processors and to renumber the processor compactly. (3) The *processor priority tree* coordinates access to memory by determining which processors are allowed to read or write each shared location that has to be accessed concurrently by more than one processor. (4) The *broadcast tree* is used to disseminate values among readers and writers. The use of broadcast trees in conjunction with priority trees serves to bound read and write concurrency.

The readers familiar with the algorithm can proceed to Section 4.

3.1. Top Layer Control Structure

The top level algorithm (Figure 1) consists of the *main loop* that iterates through four phases until the *Write-All* problem is solved (this is based on algorithm W [17]). The algorithm uses two complete binary trees: the processor enumeration tree with P leaves, and the progress tree with H leaves $(1 \le H \le N)$, where a cluster of N/H elements of the *Write-All* array is associated with each leaf. The active processors synchronously execute the four phases as follows:

Phase 1, failure detection via processor enumeration. All processors traverse, bottom-up, the processor enumeration tree starting with the leaves associated with processor identifiers (PIDs) and finishing at the root. This parallel-prefix-like algorithm enumerates active processors and yields an overestimate of the total.

Phase 2, processor allocation. The processors traverse, top-down, the progress tree using a divide-and-conquer approach (based on processor enumeration and progress measurement) to allocate themselves to the unvisited leaves of the progress tree.

```
forall processors PID=1..$P$ parbegin
  Phase 3: Visit the leaves based on PID to perform work on the input data
  Phase 4: Traverse the progress tree bottom up to measure progress
  while the root of the progress tree is not $H$ do
    Phase 1: Traverse the enumeration tree bottom up to enumerate processors
    Phase 2: Traverse the progress tree top down to reschedule work
    Phase 3: Perform rescheduled work on the input data
    Phase 4: Traverse the progress tree bottom up to measure progress
  od
parend
```

Fig. 1. Top level control structure of algorithm KMS.

Phase 3, work phase. The processors work at the leaves they reached in *Phase 2*, where they write to the appropriate N/H elements of the input array.

Phase 4, progress measurement. The processors traverse, bottom-up, the progress tree and compute an underestimate of the progress for each subtree. They start from the leaves where they were at the end of *Phase 3*. Here the underestimate is computed using a version of the common logarithmic-time summation algorithm.

The bottom layer of algorithm KMS controls the concurrency of access to shared memory. We first describe the main data structure, then the algorithms.

3.2. Processor Priority Trees

Algorithm KMS controls read and write concurrency by organizing processors into a *processor priority tree* (PPT). This is a binary tree whose nodes are associated with processors based on a processor numbering. Say there are $p \leq P$ processors, numbered from 1 to p, that intend to write to a location T at the same time. The PPT has p nodes that are also numbered from 1 to p in a breadth-first left-to-right fashion. The processor i is associated with the node i. Thus all levels of the tree, except possibly for the last, are full and the leaves of the last level are packed as left as possible. *Priorities* are assigned to the processors according to the tree levels: the root has the highest priority and priorities decrease with each successive level.

Priorities determine when a processor can write to the memory location T. The processors with the same priority attempt to write to T concurrently but *only if higher priority processors have failed to do so*. So, if the value of T is changed by processors at a certain priority level, then no lower priority processors will write to T. To ensure this, processors at all priority levels need to decide whether the value of T is "new" or "old". If read concurrency were of no concern then all processors can simply read the value. In algorithm KMS, a *broadcast* routine is used to control read concurrency by propagating the value of T within each level of PPT.

The top layer of algorithm KMS has processors traversing the progress and enumeration trees in a bottom-up fashion. Here at each intermediate node of a tree two PPTs are combined into one as the processors that come up from the children of the node "meet" at the parent. This involves *compacting* and *merging* the PPTs. PPTs are compacted to eliminate "certifiably" faulty processors. Such processors are defined to be the processors of the higher priority than the processors that effected the write. The algorithm ensures that all processors in a PPT know the priority level of the successful writers, which allows the survivors to renumber themselves by subtracting from their indices the number of certifiably faulty processors. Then the two PPTs are merged: the processors of the left PPT are appended to the tree formed by the processors of the right one. This is done by adding the number of the processors in the right PPT to the processor numbers of the left PPT.

3.3. Dealing with Individual Reads and Writes

We now describe three algorithms used to control memory access concurrency for individual reads and writes.

Algorithm CR/W. The most general algorithm, called CR/W (Concurrent Read/Write), implements broadcast for processors within different levels of a PPT and allows processors to write to a shared location T only if processors at higher levels have not done so.

Communication between processors in a PPT takes place through a shared array, call it B, where the processors communicate based on their positions in the PPT. $B[k]$ stores values read by the kth processor of the PPT. Each processor on levels $0, \ldots, i-1$ is associated with exactly one processor on each of the levels i and lower. Specifically, the jth processor of the PPT broadcasts to the jth processor of each level below its own (in a left-to-right numbering within each level). The algorithm makes $\lfloor \log p \rfloor + 1$ iterations that correspond to the PPT levels. At iteration i, each processor of level i reads its B location. If this location has not been updated, then the processor reads T directly. Since each full PPT level has one more processor than all the levels above it combined (PPT is a binary tree), there may be at least one processor on each level that reads T directly since no processor at a higher level is assigned to it (for a full level, this processor is the rightmost one, or the root itself for level 0). In the absence of failures this is the only access to T. Concurrent accesses can occur only in the presence of failures, in which case the processors on the same level that fail to receive values from processors at higher levels concurrently read T. A processor reading T checks whether it contains the value to be written, then writes to it if it does not. Whenever processors update T they write the new value for T as well as the index of the level that effected the write. If a processor k accesses T and determines that T has the correct value, and if the failed processor ℓ that should have broadcast to k is at or below the level that effected the write, then k assumes the position of processor ℓ in the PPT. This "moves" failed processors toward the leaves. Failed processors are moved downwards only if they are not above the level that effects the write – processors above this level are eliminated by PPT compaction that takes place at the end of each run of CR/W.

Algorithms CR1 and CR2. Algorithm CR/W combines a read with a write. However, when the processors of a PPT need to read a common location but no write is involved, two simpler algorithms are used. Algorithm CR1 is similar to CR/W but includes no write step; it is simpler than CR/W in that the processors that are found to have failed are pushed toward the bottom of the PPT independent of their level. This is used for bottom-up traversals. Algorithm CR2 uses a simple top-down broadcast through the PPT. Starting with the root each processor broadcasts to its two children; if a processor fails then its two children read T directly. Thus the processors of level i broadcast only to processors of level $i+1$. Unlike CR1, no processor movement takes place. This is used for top-down traversals.

From the description of algorithms CR/W, CR1, and CR2 it follows that each takes time $O(\log P)$.

Using CR/W, CR1, and CR2 in algorithm KMS. We now describe how algorithm KMS integrates algorithms CR/W, CR1, CR2, and PPT merging and compaction within its four phases.

Phase 1: Processors begin this phase by forming single-processor PPTs. The objective is to write to each internal node of the enumeration tree the sum of the values stored at its two children. Algorithm CR/W is used to store the new value, the size of the PPT and the index of the level that completed the write. Then all PPTs are compacted. In order to merge PPTs the processors use algorithm CR1 to read the data stored at the enumeration tree node that is the sibling of the node they just updated. Then PPTs are merged. At this point the processors of the merged PPTs know the value they need to write at the next level of the enumeration tree. This value is the sum of the value written by CR/W and the value read by CR1. Hence one call to each of CR/W and CR1 is needed for each level of the enumeration tree.

Phase 2: This phase involves no concurrent writes. Processors traverse top-down the progress tree to allocate themselves to the unvisited leaves. The only global information needed at each level is the values stored at the two children of the current node of the progress tree. Two calls to CR2 are used to read these values, one for each child. Using this information the processors of a PPT compute locally whether they need to go left or right based on their identifiers. Here each PPT must be split in two. If a PPT has k processors of which k' need to go left and the remaining $k - k'$ need to go right, then by convention the first k' processors of the PPT form the PPT of the left child and the remaining $k - k'$ processors form the PPT of the right child. No compaction or merging is done in this phase.

Phase 3: Processors form PPTs based on the information they gathered during *Phase 2* and proceed to write 1 to the N/H locations that correspond to the leaf they reached. At this point, processors decide whether they need to use algorithm CR/W, followed by compaction for each of these writes. This is done locally by each processor: at the beginning of this phase, the processors have consistent information on the number of unvisited leaves, call it u, and the number of available processors, call it a (this is the information they used to allocate themselves at the leaves they reached by the end of *Phase 2*). When $u > a$, it is guaranteed (see [16]) that there is at most one processor per leaf, and therefore the processors do not use CR/W and compaction. Instead the processors go sequentially through the cluster of N/H elements at the leaf they reached and simply write to each element. When $u \leq a$, several processors may be allocated to the same leaf and the processors use algorithm CR/W followed by compaction to perform each write in the cluster. In any case, no merging is involved.

Phase 4: This phase initially uses the PPTs that resulted at the end of *Phase 3*. The task to be performed is similar to that of *Phase 1*. As before, algorithm CR/W is used for writing followed by compaction and one call to algorithm CR1, after which the PPTs are merged.

We now state previously known results [16] for algorithm KMS and for simulations using this algorithm.

Theorem 1 [16] *Algorithm* KMS *solves the Write-All(N, P, f) problem with work $W = O(N + P \log^2 N \log^2 P/ \log\log N)$, write concurrency $wc \le f$, and read concurrency $rc \le 7\, f \log N$.*

Theorem 2 [16] *Any N-processor, r-time* EREW PRAM *algorithm can be simulated on a fail-stop P-processor* CRCW PRAM *with work $W = O(r \cdot (N + P \log^2 P \log^2 N/ \log\log N))$, with write concurrency $wc \le f$, and the read concurrency $rc \le 7f \log N$, where f is the number processor stop-failures.*

These prior results do not show how the work depends on the number of processor stop-failures.

4. Analysis

We now give a new, failure-sensitive, analysis of algorithm KMS, and we obtain new failure-sensitive bounds on work of PRAM simulations with controlled memory access concurrency.

In the analysis we use the parameterized version of algorithm KMS with $P \le N$ and where the progress tree has $H = \max\{P, N/ \log N \log P\}$ leaves. The array elements are associated with the leaves of this tree, with N/H array elements per leaf. Henceforth we use KMS to denote this parameterized algorithm.

For algorithm KMS, we define U_i to be the number of unvisited leaves of the progress tree ($U_i \le H$), and P_i to be the number of non-faulty processors ($P_i \le P$), at the start of the i-th iteration of the main loop. We define σ_1 to be the time required for a processor to complete one iteration of the main loop when $P_i < U_i$. We define σ_2 to be the time required for a processor to complete one iteration of the main loop when $P_i \ge U_i$. We define a *block-step* to be the execution by one processor of the body of the main loop.

Lemma 1 *The work required by algorithm* KMS *to solve the Write-All(N, P, f) problem is $W = O(\sigma_1 \cdot H + \sigma_2 \cdot \frac{P \log P}{\log\log P})$.*

Proof: We consider two cases.

Case 1: Consider *all* iterations i in which $P_i < U_i$. In this case the number of block-steps is $O(H)$ since no more than one processor is assigned to each leaf of the progress tree. Then, using the definition of σ_1, the work of algorithm KMS in this case is $O(\sigma_1 \cdot H)$.

Case 2: We now account for *all* iterations in which $P_i \ge U_i$. In this case the number of block-steps is $O(P \frac{\log P}{\log\log P})$. Given the load-balancing properties of algorithm KMS, this follows directly from the case analysis of Theorem 3.1 [12], where Case 2 considers the work of perfect load-balancing iterative algorithms when $P_i > U_i$. (The simpler subcase of $P_i = U_i$ is dealt similarly.) Then, using the definition of σ_2, the work of algorithm KMS in this case is $O(\sigma_2 \cdot P \frac{\log P}{\log\log P})$.

Combining the two cases yields the result. □

Note that in the above lemma, work is not expressed as a function of f, the number of processor stop-failures. In the next lemma, we give work as a function of f, for $f \le P/ \log P$.

Lemma 2 *The work required by algorithm* KMS *to solve the Write-All(N, P, f) problem when $f \le \frac{P}{\log P}$ is $W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot \frac{P \log H}{\log(P/f)})$.*

Proof: Let U be the number of unvisited leaves of the progress tree (recall that the tree has H leaves with N/H array elements assigned to each leaf). Let Δf denote the number of processor stop-failures within a particular *iteration* of the algorithm. Δf is, in general, different for each iteration, though the sum of these for all iterations cannot exceed f. We set $b = b(P,f) = P/(2f)$, and we define $W(U,P,f)$, where $U \leq H$, to be the work required to solve *Write-All*$(U \cdot N/H, P, f)$. We show that for all U, P and f, $W(U,P,f)$ is no more than $\sigma_1(P+U) + 3\sigma_2 P + \sigma_2 P \log_{P/(2f)} U$ $(= O(\sigma_1 \cdot (U+P) + \sigma_2 \cdot P \log_{P/f} U))$. The proof proceeds by induction on U (following our approach in [13]).

Base Case: Observe that when $U = 1$ and $P \geq 1$ (hence $P \geq U$), $W(U,P,f) \leq \sigma_2 P \leq \sigma_1(P+U) + 3\sigma_2 P + \sigma_2 P \log_b U$, for all P and f, as desired.

Inductive Hypothesis: Assume that we have proved the result for all $U < \hat{U}$ and all P and f.

Inductive Step: Consider $U = \hat{U}$. We investigate two cases:

Case 1: $P < \hat{U}$. In this case each processor is assigned to a unique unvisited leaf (this follows from the load-balancing properties of algorithm KMS), hence

$$W(\hat{U},P,f) \leq \sigma_1 P + \max_{0 \leq \Delta f \leq f} W(\hat{U} - P + \Delta f, P - \Delta f, f - \Delta f).$$

As $P - \Delta f > 0$, $\hat{U} - P + \Delta f < \hat{U}$ and, by the induction hypothesis,

$$\begin{aligned} W(\hat{U},P,f) \leq \sigma_1 P + \max_{0 \leq \Delta f \leq f} [&\sigma_1(P - \Delta_f + \hat{U} - P + \Delta_f) + 3\sigma_2(P - \Delta f) \\ &+ \sigma_2(P - \Delta f)\log_{b(P-\Delta f, f-\Delta f)}(\hat{U} - P + \Delta f)]. \end{aligned}$$

Now, $b(P - \Delta f, f - \Delta f) \geq b(P,f)$, so that

$$W(\hat{U},P,f) \leq \sigma_1(P + \hat{U}) + 3\sigma_2 P + \sigma_2 P \log_{b(P,f)} \hat{U},$$

as desired.

Case 2: $P \geq \hat{U}$. In this case, by assumption we have

$$W(\hat{U},P,f) \leq \sigma_2 P + \max_{0 \leq \Delta f \leq f} W(\gamma\hat{U}, P - \Delta f, f - \Delta f),$$

where $\gamma = \gamma(\hat{U}, P, \Delta f)$ is the ratio of the number of the remaining unvisited leaves to $\hat{U}$ $(0 \leq \gamma < 1)$.

Let $\phi = \Delta f / P \leq f/P < 1$, the fraction of processors which fail during this iteration; then $\phi/2 < \gamma < 2\phi$.

$\Big($To see this, observe that $\frac{\phi P}{\lceil P/\hat{U} \rceil \hat{U}} = \frac{\phi P / \lceil P/\hat{U} \rceil}{\hat{U}} \leq \gamma \leq \frac{\phi P / \lfloor P/\hat{U} \rfloor}{\hat{U}} = \frac{\phi P}{\lfloor P/\hat{U} \rfloor \hat{U}}$. Let $P = c\hat{U}$, $c > 1$. Then $\frac{c}{\lceil c \rceil}\phi = \frac{\phi c \hat{U}}{\lceil c \rceil \hat{U}} \leq \gamma \leq \frac{\phi c \hat{U}}{\lfloor c \rfloor \hat{U}} = \frac{c}{\lfloor c \rfloor}\phi$. Now observe that $1 \leq \frac{c}{\lfloor c \rfloor} < 2$ and $1/2 < \frac{c}{\lceil c \rceil} \leq 1$, $\forall c > 1$, and hence, $\phi/2 < \gamma < 2\phi$, as desired.$\Big)$

Then,

$$W(\hat{U},P,f) \leq \sigma_2 P + \max_{\phi \in [0, f/P]} W(\gamma\hat{U}, (1-\phi)P, f - \phi P).$$

As $\gamma\hat{U} < \hat{U}$, we may apply the induction hypothesis:

$$\begin{aligned} W(\hat{U},P,f) \leq \sigma_2 P + \max_{\phi \in [0, f/P]} \Big[&\sigma_1(\gamma\hat{U} + (1-\phi)P) + 3\sigma_2(1-\phi)P \\ &+ \sigma_2(1-\phi)P\log_{b'}(\gamma\hat{U})\Big], \end{aligned}$$

where $b' = b(P - \phi P, f - \phi P)$. As above, $b' \geq b(P, f)$, so that

$$W(\hat{U}, P, f) \leq \sigma_2 P + \max_{\phi \in [0, f/P]} \Big[\sigma_1(\gamma \hat{U} + (1-\phi)P) + 3\sigma_2(1-\phi)P + \sigma_2(1-\phi)P \log_{b(P,f)}(\gamma \hat{U}) \Big].$$

To complete the proof, it suffices to show that for all $\phi \in [0, f/P]$,
$\sigma_1 \phi P + 2\sigma_2 P + \sigma_2 P \log_{b(P,f)} \hat{U} - (1-\phi)\sigma_2 P \log_{b(P,f)}(\gamma \hat{U}) \geq 3\sigma_2(1-\phi)P - \sigma_1 \hat{U}(1-\gamma)$.
Upper bounding $3\sigma_2(1-\phi)P - \sigma_1 \hat{U}(1-\gamma)$ with $3\sigma_2(1-\phi)P$, removing $\sigma_1 \phi P$ from the left hand side, and dividing through by $\sigma_2 P$, it is sufficient to show that

$$2 + \log_{b(P,f)} \hat{U} - (1-\phi)\log_{b(P,f)}(\gamma \hat{U}) \geq 3(1-\phi),$$

or, equivalently,

$$\log_{b(P,f)} \hat{U} - (1-\phi)\log_{b(P,f)}(\gamma \hat{U}) \geq 1 - 3\phi.$$

We now focus on the left hand side of the above equation:
$\log_{b(P,f)} \hat{U} - (1-\phi) \left[\log_{b(P,f)} \gamma + \log_{b(P,f)} \hat{U}\right] = \phi \log_{b(P,f)} \hat{U} + (1-\phi)\log_{b(P,f)} \gamma^{-1}$.
Since $f \leq P/\log P$, for any $P \geq 16$ we have that $P/(2f) > 2$. Observe that,

$$\phi \log_{b(P,f)} \hat{U} + (1-\phi)\log_{b(P,f)} \gamma^{-1} \geq (1-\phi)\log_{b(P,f)} \gamma^{-1}$$

since $\hat{U} \geq P/f > P/(2f)$. (Note that if $\hat{U} < P/f$, then all leaves are visited in this iteration.) Recall that $\gamma^{-1} \geq (2\phi)^{-1}$ and $\phi < f/P$. Therefore,

$$(1-\phi)\log_{b(P,f)} \gamma^{-1} \geq (1-\phi)\log_{b(P,f)}(2\phi)^{-1} \geq 1 - 3\phi.$$

Evidently,

$$W = O\left(\sigma_1 \cdot (U + P) + \sigma_2 \cdot \frac{P \log U}{\log(P/f)}\right) = O\left(\sigma_1 \cdot (H + P) + \sigma_2 \cdot \frac{P \log H}{\log(P/f)}\right),$$

as desired. □

We define the quantity $\Lambda_{r,P,f}$, that we use to simplify the presentation of the results in this section.

$$\Lambda_{r,P,f} = \begin{cases} \log(\frac{Pr}{f}) & \text{when } f \leq \frac{Pr}{\log P} \\ \log\log P & \text{when } f > \frac{Pr}{\log P} \end{cases}$$

Lemma 3 *Algorithm* KMS *solves the Write-All(N, P, f) problem for any stop-failure pattern using work* $W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot P \log N / \Lambda_{1,P,f})$.
Proof: We first record that $H < H + P$, $\log P \leq \log N$ and $\log H \leq \log N$. Then the result follows by combining Lemmas 1 and 2 with the definition of $\Lambda_{1,P,f}$ (it is $\log(P/f)$ when $f \leq P/\log P$ and $\log\log P$ when $f > P/\log P$). □

Lemma 4 *For algorithm* KMS, $\sigma_1 = O(\log N \log P)$ *and* $\sigma_2 = O(\log N \log^2 P)$.
Proof: We consider the following two cases.
Case 1: $P < \frac{N}{\log N \log P}$. Here the number of leaves in the progress tree is $H = N/\log N \log P$ and in *Phase 3* each processor writes to $N/H = \log N \log P$ array elements. The time required to traverse the enumeration and progress trees is $O(\log N \log P)$ and the execution of CR/W takes $O(\log P)$ time.

For the iteration i when $U_i \geq P_i$, algorithm CR/W is not used in *Phase 3* and therefore the time to update a leaf is $O(\log N \log P)$ (the number of elements). Therefore, $\sigma_1 = O(\log N \log P) + O(\log N \log P) = O(\log N \log P)$ (the time to reach a leaf plus the time to update a leaf).

For the iteration i when $U_i < P_i$, algorithm CR/W is used in *Phase 3*. In the worst case, all processors could be allocated to the same leaf (e.g., when there is only one unvisited leaf left) and hence, $\log P$ time must be spent at each element of the leaf. Since there are $\log N \log P$ elements per leaf the worst case time to update a leaf is $O(\log N \log^2 P)$. Hence, $\sigma_2 = O(\log N \log P) + O(\log N \log^2 P) = O(\log N \log^2 P)$.

Case 2: $\frac{N}{\log N \log P} \leq P \leq N$. Here the number of leaves in the progress tree is $H = P$ and in *Phase 3* each processor writes to $N/P = O(\log N \log P)$ array elements. Then the bounds on σ_1 and σ_2 are obtained similarly to Case 1. □

We now state and prove our main result for algorithm KMS.

Theorem 3 *Algorithm* KMS *solves the Write-All*(N, P, f) *problem with write concurrency* $wc \leq f$, *read concurrency* $rc \leq 7 f \log N$ *and work* $W = O(N + P \log^2 N \log^2 P/\Lambda_{1,P,f})$.

Proof: The bounds on wc and rc are obtained from Theorem 1 (see [16]). We now show the bounds on W, by considering the following two cases:

Case 1: $P < \frac{N}{\log N \log P}$. Here the number of leaves in the progress tree is $H = N/\log N \log P$. Combining Lemmas 3 and 4 we get $W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot P \log N/\Lambda_{1,P,f}) = O((\log N \log P) \cdot N/(\log N \log P) + (\log N \log^2 P) \cdot P \log N/\Lambda_{1,P,f}) = O(N + P \log^2 N \log^2 P/\Lambda_{1,P,f})$.

Case 2: $\frac{N}{\log N \log P} \leq P \leq N$. Here the number of leaves in the progress tree is $H = P$. Combining Lemmas 3 and 4 we have $W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot P \log N/\Lambda_{1,P,f}) = O((\log N \log P) \cdot P + (\log N \log^2 P) \cdot P \log N/\Lambda_{1,P,f}) = O(P \log^2 N \log^2 P/\Lambda_{1,P,f})$.

The result is obtained by combining Case 1 and Case 2. □

This analysis establishes the following processor ranges for which algorithm KMS becomes optimal.

Corollary 1 *Algorithms* KMS *is work-optimal if* $P = O(N \log(N/f)/\log^4 N)$, *when* $f \leq P/\log P$, *and if* $P = O(N \log\log N/\log^4 N))$, *when* $f > P/\log P$.

Theorem 1 teaches that algorithm KMS becomes optimal if $P = O(N \log\log N/\log^4 N)$, for all $f < P$. Corollary 1 shows that our failure-sensitive analysis extends the range of optimality of the algorithm when $f \leq P/\log P$.

We now obtain new failure-sensitive bounds for the *iterative Write-All* problem with controlled read and write memory access concurrency.

Theorem 4 *The r-Write-All*(N, P, f) *problem can be solved on* P *fail-stop processors with write concurrency* $wc \leq f$, *read concurrency* $rc \leq f \log N$ *and work* $W = O(r \cdot (N + P \log^2 N \log^2 P/\Lambda_{r,P,f}))$.

Proof: We solve *r-Write-All*(N, P, f) by running algorithm KMS r times, once for each *Write-All* instance. We enumerate the r instances of *Write-All* using numbers $1, \ldots, r$, and we refer to instance i as the *round* i. For round i, let P_i be the number of active processors at the beginning of the round and f_i be the number of crashes during the round. Note that $P_1 = P$, and that $P_i \leq P$.

We first establish the bounds on the memory access concurrency. Let wc_i and rc_i be the write and read memory access concurrency accrued in round i, respectively. Then, $wc = \sum_{i=1}^{r} wc_i$ and $rc = \sum_{i=1}^{r} rc_i$. Using Theorem 3 for each round, we have that $wc_i \leq f_i$ and $rc_i \leq f_i \log N$. Therefore, $wc = \sum_{i=1}^{r} wc_i \leq \sum_{i=1}^{r} f_i = f$, and $rc = \sum_{i=1}^{r} rc_i \leq \log N \sum_{i=1}^{r} f_i = f \log N$, as desired.

Observe that the choice of each f_i does not affect the bounds on the memory access concurrency. However, in order to establish the bounds on work we need to determine the values of the f_is that maximize the overall work of *r-Write-All*(N, P, f). We consider two cases:

Case 1: $f > \frac{Pr}{\log P}$. Consider round i. From Theorem 3 we have that the work for this round is $O(N + P_i \log^2 N \log^2 P_i / \log(P_i/f_i))$ when $f_i \leq P_i/\log P_i$ and $O\left(N + P_i \log^2 N \log^2 P_i / \log\log P_i\right)$ otherwise. However in this case, we can have $f_i = \Theta\left(P/\log P\right)$ for all rounds without "running out" of processors. Thus,

$$W = O\left(r \cdot \left(N + P \log^2 N \log^2 P / \log\log P\right)\right).$$

Case 2: $f \leq \frac{Pr}{\log P}$. First observe that any reasonable adversary would not fail-stop more that $P_i/\log P_i$ processors in round i, since it would not cause more work than $O(N + P_i \log^2 N \log^2 P_i / \log\log P_i)$ (which is achieved when $f_i \geq P_i/\log P_i$). Therefore, we consider $f_i \leq P_i/\log P_i$ for all rounds. Hence, the work in every round i (per Theorem 3) is $O\left(N + P_i \log^2 N \log^2 P_i / \log(P_i/f_i)\right) = O\left(N + P \log^2 N \log^2 P / \log(P/f_i)\right)$.

Let $W(N, P, f)$ be this one-round upper bound. As $f = \sum f_i$, an upper bound on *r-Write-All*(N, P, f) can be given by maximizing $\sum_i W(N, P_i, f_i)$ over all such failure patterns. As $W(\cdot, \cdot, \cdot)$ is monotone in P, we may assume that $P_i = P$ for the purposes of the upper bound. We show that this maximum is attained at $f_1 = f_2 = \ldots = f_r$. For simplicity, treat f_i as a continuous parameter and consider the factor in the single round work expression (given above) that depends on f_i: $k/\log(\frac{P}{f_i})$. (Here k is the constant hidden by the $O(\cdot)$ notation.)

The first derivative over f_i is $\frac{d}{df_i}\left(k/\log\left(\frac{P}{f_i}\right)\right) = k/f_i(\log P - \log f_i)^2$, and its second derivative is $\frac{d^2}{df_i^2}\left(k/\log\left(\frac{P}{f_i}\right)\right) = 2k/f_i^2(\log P - \log f_i)^3 - k/f_i^2(\log P - \log f_i)^2$. Observe that the second derivative is negative in the domain considered (assuming $P > 16$). Hence the first derivative is decreasing (with f_i). In this case, given any two f_i, f_j where $f_i > f_j$, the failure pattern obtained by replacing f_i with $f_i - \epsilon$ and f_j by $f_j + \epsilon$ (where $\epsilon < (f_i - f_j)/2$) results in increased work. This implies that the sum maximized when all f_is are equal, specifically when $f_i = f/r$.

As the above upper bound on the sum $\sum_i W(N, P_i, f_i)$ is valid over *all* f_i in this range, it holds in particular for the choices made by the adversary which must, of course, cause an integer number of faults in each round. Therefore,

$$W = O\left(r \cdot \left(N + P \log^2 N \log^2 P / \log(Pr/f)\right)\right).$$

The bound on work W follows by the two cases and the definition of $\Lambda_{r,P,f}$. $\square$

Theorem 4 enables us to obtain a tighter bound on work when algorithm KMS is iteratively used to obtain efficient PRAM simulations on fail-stop PRAMs (as opposed to the bound of Theorem 2).

Theorem 5 *Any N processor, r parallel time* EREW PRAM *algorithm can be simulated on a fail-stop P-processor* CRCW PRAM *with work $W = O(r \cdot (N + P \log^2 N \log^2 P / \Lambda_{r,P,f}))$ so that the write concurrency of the simulation is $wc \leq f$ and the read concurrency is $rc \leq 7f \log N$, f being the number of processor failures.*

Proof: The complexity of simulating a single parallel step of N ideal processors on P failure-prone processors does not exceed the complexity of solving a single *Write-All*(N, P, f) instance [22,26]. The result then follows from Theorem 4. $\square$

Note that this last result can be extended to other PRAM variants, such as CREW and CRCW, however in these cases the read and write concurrency of the simulation depends on the actual read and write concurrency of the specific algorithms.

5. Conclusion

In this paper we derive the first failure-sensitive bounds on work for the *Write-All* and *iterative Write-All* problems in the fail-stop synchronous CRCW PRAM model where memory access concurrency needs to be controlled. The failure-sensitive solution for the *iterative Write-All* problem also leads to a new and tighter bound on the work of PRAM simulations on fail-stop PRAMs. We obtain our results by giving a new analysis of the deterministic algorithm KMS [16]. Our future work in this area targets bounds on work and memory access concurrency in other models, such as PRAMs with stop-failures and restarts. Another promising direction includes the use of randomization that proved to be very effective in this general area, cf. the work of Kontogiannis et al. [23].

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, 1993.

[2] Y. Aumann and M.O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proc. of* 33rd *IEEE Symposium on Foundations of Computer Science*, pp. 147–156, 1992.

[3] R.J. Anderson and H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing*, 26(5):1277–1283, 1997.

[4] J. Buss, P.C. Kanellakis, P. Ragde, and A.A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.

[5] B. Chlebus, S. Dobrev, D. Kowalski, G. Malewicz, A. Shvartsman, and I. Vrto. Towards practical deterministic Write-All algorithms. In *Proc. of* 13th *ACM Symposium on Parallel Algorithms and Architectures*, pp. 271–280, 2001.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Torwards a realistic model of parallel computation. In *Proc. of* 4th *Symp. on Principles and Practice of Parallel Programming*, pp. 1–12, 1993.

[7] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstation: A fault-tolerant, high performance approach. In *Proc. of the* 15th *IEEE International Conference on Distributed Computer Systems*, pp. 467–474, 1995.

[8] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Reviews in Computer Science*, 3:233–283, 1988.

[9] J.F. Groote, W.H. Hesselink, S. Mauw, and R. Vermeulen. An algorithm for the asynchronous Write-All problem based on process collision. *Distributed Computing*, 14(2):75–81, 2001.

[10] P.B. Gibbons. A more practical PRAM model. In *Proc. of* 1st *ACM Symposium on Parallel Algorithms and Architectures*, pp. 158–168, 1989.

[11] P.B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, 196(1–2):3–29, 1998.

[12] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of distributed cooperation in the presence of failures. In *Proc. of the* 4th *International Conference on Principles of Distributed Systems*, pp. 245–264, 2000.

[13] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. *Distributed Computing*, 17(1):47–63, 2004.

[14] Ch. Georgiou, A. Russell, and A.A. Shvartsman, Failure-Sensitive Analysis of Parallel Algorithms with Controlled Memory Access Concurrency. In *Proc. of the 6th International Conference on Principles of Distributed Systems*, pp. 127–138, 2002.

[15] A.M. Gibbons and P. Spirakis, editors. *Lectures on Parallel Computation*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1993.

[16] P.C. Kanellakis, D. Michailidis, and A.A. Shvartsman. Controlling memory access concurrency in efficient fault-tolerant parallel algorithms. *Nordic Journal of Computing*, 2(2):146–180, 1995.

[17] P.C. Kanellakis and A.A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, 1992.

[18] P.C. Kanellakis and A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.

[19] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pp. 869–941, 1990.

[20] Z.M. Kedem, K.V. Palem, M.O. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proc. of* 24th *ACM Symposium on Theory of Computing*, pp. 306–318, 1992.

[21] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for dependable parallel computing. In *Proc. of* 23rd *ACM Symposium on Theory of Computing*, pp. 381–390, 1991.

[22] Z.M. Kedem, K.V. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. of* 22nd *ACM Symposium on Theory of Computing*, pp. 138–148, 1990.

[23] S. Kontogiannis, G. Pantziou, P. Spirakis, and M. Yung. Robust parallel computations through randomization. *Theory of Computing Systems*, 33(5/6):427–464, 2000.

[24] C. Martel, A. Park, and R. Subramonian. Work-optimal asynchronous algorithms for shared memory parallel computers. *SIAM J. on Computing*, 21(6):1070–1099, 1992.

[25] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proc. of* 31st *IEEE Symposium on Foundations of Computer Science*, pp. 590–599, 1990.

[26] A.A. Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters*, 39(2):59–66, 1991.

[27] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[28] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism. In *Proc. of* 10th *ACM Symposium on Parallel Algorithms and Architectures*, 1998.