ANALYSIS OF INSTANT AND TOTAL MEMORY ACCESS CONCURRENCY IN ROBUST PARALLEL ALGORITHMS

CHRYSSIS GEORGIOU, ALEXANDER RUSSELL, AND ALEXANDER SHVARTSMAN

Abstract. Algorithms in synchronous parallel models of computation with processor crashes can be made both efficient and fault-tolerant. The basis for fault-tolerance in such settings is the ability of multiple processors to concurrently read from and write to shared memory. Concurrent memory access provides redundancy that is necessary for combining fault-tolerance and efficiency. The model considered here is the PRAM (parallel random access machine) where processors can crash. When considering failure-free PRAMs, there exists a sharp separation between CRCW (concurrent access) and EREW (exclusive access) PRAMs—it is known that EREW PRAM algorithms can incur up to a logarithmic degradation in efficiency as compared to the CRCW PRAMs. Given that efficiency can be completely lost when EREW processors can crash, it is interesting to investigate the extent to which concurrent access is necessary for guaranteeing both fault-tolerance and efficiency. The efficiency of algorithms in this setting is measured in terms of work and memory access concurrency. It has also been shown that solutions for a specific problem can be used to construct fault-tolerant versions of all efficient algorithms for EREW PRAMs. This problem is called Write-All: “using \( P \leq N \) processors, write 1’s into \( N \) locations.” The focus of this paper is on the analysis of algorithms that control the maximum instant memory access concurrency, i.e., maximum concurrency that can be incurred during any single step, and total concurrency, i.e., the sum of instant concurrency over all steps of a computation. This paper gives a new failure-sensitive analysis of an algorithmic approach proposed by Kanellakis et al. The details of the complete algorithm are described, and the analysis of its work, and instant and total concurrency is shown. The analysis is extended for the iterative use of the algorithm, leading to the results dealing with simulations of arbitrary failure-free EREW PRAMs on crash-prone synchronous multiprocessors.

1. Introduction

The Parallel Random Access Machine, or PRAM [10], has served as the target model for numerous synchronous shared-memory parallel algorithms [9, 16, 21]. 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. However, PRAM makes assumptions that, given the current state of technology, make it difficult to be implemented as a scalable massively-parallel architecture (cf. [1, 28]). The model assumes that the processors are globally synchronized, that shared memory can be concurrently accessed by arbitrary number of processors, and that the processors are completely reliable. Several approaches have been developed to deal with this by providing a sufficiently convenient programming model while weakening the PRAM assumptions (e.g., [7, 13, 23, 20, 25, 27, 12]). Some approaches propose simulations of PRAM algorithms on other platforms. It has been shown that solutions for a particular problem, called Write-All, can be used iteratively in constructing such simulations (e.g., [8, 20, 26]). The Write-All problem [22] 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 accomplished in unit time by a PRAM where \( P = N \). Here the storing of the 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 [14]:

*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.*

The efficiency of Write-All algorithms is assessed in terms of the work complexity that accounts for all steps taken by the processors during the computation [22]. 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.

Obtaining efficient solutions for Write-All problems becomes challenging in the presence of failures, in the absence of synchrony, or without concurrent memory access, e.g. [3, 4, 5, 11, 23, 18, 24]. Algorithms in synchronous models with processor crashes can be made both efficient and fault-tolerant [22]. The basis for
fault-tolerance in such settings is the ability of multiple processors to read from, and write to, shared memory concurrently.

The target model considered here is the CRCW PRAM where processors can crash. For failure-free PRAMs, there exists a sharp separation between CRCW and EREW PRAMs—it is known that EREW PRAM algorithms can incur up to a logarithmic degradation in efficiency as compared to the CRCW PRAMs [6]. On the other hand, efficiency can be completely lost when EREW processors can crash—the lower bound for the Write-All in this case is quadratic [22]. Thus concurrency is necessary for both efficiency and fault-tolerance, and it is interesting to investigate the bounds on concurrency in fault-tolerant parallel algorithms.

Here we are concerned with measuring work and memory access concurrency of fault-tolerant parallel algorithms. Consider a step of a parallel computation, where location $m$ is written by $P$ processors. Then $P - 1$ of these writes are potentially “redundant”, because a single write suffices. Thus we measure “concurrency” as the number of redundant memory accesses, and we measure concurrency for both 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 $P - 1$.

The focus of this paper is on the analysis of algorithms that control instant memory access concurrency, i.e., maximum concurrency that can be incurred during any single step, and total concurrency, i.e., the sum of instant concurrency over all steps of a computation.

Prior work. It was shown by Kanellakis, Michailidis and Shvartsman in [17] how to construct efficient fault-tolerant algorithms such that the total 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 $f = 0$.

Georgiou, Russell, and Shvartsman [15] obtained failure-sensitive bounds on work for the algorithm in [17], while retaining the known total memory access
concurrency bounds. Specifically, they express the work bound as a function of \( N \), \( P \), and \( f \). They also give a failure-sensitive analysis for iterative Write-All with controlled total memory access concurrency. This result yields tighter bounds on work (vs. [17]) for simulations of PRAM algorithms on fail-stop PRAMs. The author do not consider the instant access concurrency.

The work of Kanellakis, Michailidis, and Shvartsman [17] additionally proposed an approach based on pipelining that enables for the instant memory access concurrency to be controlled. They sketched the work analysis for such algorithms, however it was not shown how the number of processor failures \( f \) affects the work efficiency of the algorithms.

**Contributions.** This paper gives a new failure-sensitive analysis of an algorithmic technique proposed by Kanellakis, Michailidis, and Shvartsman [17] for the Write-All problem. The details of the complete algorithm are described, and the analysis of its work, and instant and total concurrency is shown. Results are also given for the iterative use of the algorithm, leading to results dealing with algorithms simulations. The target model of computation is the CRCW PRAM where processors are subject to arbitrary patterns of stop-failures.

The algorithm, called algorithm \( \tilde{\text{KMS}} \), is based on algorithm KMS, combined with a pipelining technique as proposed in [17]. Algorithm \( \tilde{\text{KMS}} \) solves the Write-All problem by having processors traversing data structures of logarithmic depth. The work analysis of the algorithm uses our approach [14, 15] for analyzing work-performing algorithms by separately assessing the costs of tolerating failures and the costs of achieving perfect load balancing. To control the total concurrency we use data structures also of logarithmic depth, introduced in [17]. To control the instant concurrency we use the pipelining technique proposed in [17], where processors are divided into \( V \) waves, and traverse the data structures one wave at a time. By iteratively using algorithm \( \tilde{\text{KMS}} \) we obtain new results for the iterative Write-All problem that lead to results dealing with simulations of arbitrary failure-free EREW PRAMs on crash-prone synchronous multiprocessors.

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 Write-All problem of using \( P \) processors
to solve \( r \) instances of \( N \)-size Write-All by “solving one instance at a time”. Recall that by memory access concurrency we mean the total number of redundant memory accesses. We let \( \hat{rc}(N, P, f) \) stand for the instant read concurrency, and \( \hat{wc}(N, P, f) \) for the instant write concurrency, that is, the maximum read and write concurrency that can be incurred during any single step of a terminating computation in the presence of \( f \) failures. Similarly, we let \( rc(N, P, f) \) stand for the worst case total read concurrency, and \( wc(N, P, f) \) for the worst case total write concurrency, for a terminating computation in the presence of \( f \) failures. Finally, we let \( V \) denote the number of processor waves used in algorithm \( \widetilde{KMS} \).

We now state our results in detail.

1. Algorithm \( \widetilde{KMS} \) with \( V \) waves solves the Write-All\((N, P, f)\) problem with total write concurrency \( wc(N, P, f) \leq f \), total read concurrency \( rc(N, P, f) \leq 7f \log N \), instant read and write concurrency \( \hat{rc}(N, P, f), \hat{wc}(N, P, f) \leq P/V \) and with work \( W_1 \) as follows.

\[
W_1 = \begin{cases} 
O(N(1 + V/ \log N) + P(\log N + V) \log N \log^2 P / \log \log P), & \text{when } f > P/ \log P \\
O(N(1 + V/ \log N) + P(\log N + V) \log N \log^2 P / \log(P/f)), & \text{when } f \leq P/ \log P 
\end{cases}
\]

For \( V = \log N \), the bounds on work and total concurrency are the same as in [15], but the instant concurrency is reduced from \( O(P) \) to \( O(P/ \log N) \).

For \( V = \log^k N \), \( k \geq 1 \), the bounds on total and instant concurrency are the same as the bounds (stated without a proof) in [17], but the bounds on work are improved and made failure-sensitive.

2. By iteratively using algorithm \( \widetilde{KMS} \) with \( V = \log^k N \), \( k \geq 1 \), we show that the iterative \( r \)-Write-All\((N, P, f)\) problem can be solved with total write concurrency \( wc(N, P, f) \leq f \), total read concurrency \( rc(N, P, f) \leq 7f \log N \), instant access
concurrency \( \hat{rc}(N, P, f) \), \( \hat{wc}(N, P, f) \) ≤ \( P/V \), and with work \( W_r \) as follows.

\[
W_r = \begin{cases} 
O(r \cdot (N \log^{k-1} N + P \log^{k+1} N \log^2 P / \log \log P)), & \text{when } f > Pr / \log P \\
O(r \cdot (N \log^{k-1} N + P \log^{k+1} N \log^2 P / \log(Pr/f))), & \text{when } f \leq Pr / \log P 
\end{cases}
\]

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

For \( k = 1 \), the above analysis for iterative Write-All obtains the same bounds on work and total concurrency as the bounds in [15], but it also obtains bounds on instant concurrency, whereas no such bounds are given in [15].

3. Using the above and the fact [20, 26] that the complexity of simulating a single parallel step of \( N \) fault-free processors on \( P \) failure-prone processors (for \( P \leq N \)) does not exceed the complexity of solving a single Write-All \( (N, P, f) \) instance, we obtain the following simulation result. Let \( A \) be any \( N \)-processor, \( r \)-time EREW PRAM algorithm. We show that \( A \) can be simulated on a \( P \)-processor CRCW PRAM with up to \( f < P \) stop-failures with

- total write concurrency \( wc(N, P, f) \) ≤ \( f \),
- total read concurrency \( rc(N, P, f) \) ≤ \( 7f \log N \),
- instant read and write concurrency \( \hat{rc}, \hat{wc} \) ≤ \( P / \log^k N \), and
- work \( W_r \) as in (2),

for parameters \( V = \log^k N \) and \( k \geq 1 \).

**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. In Section 4 we describe algorithm \( \tilde{KMS} \), and in Section 5 we present its analysis. We also show the analysis for the iterative Write-All and the analysis for PRAM simulations. We conclude in Section 6.

SIU 2004


2. Models and Definitions

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

**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 \( 1, \ldots, P \). 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 assume the fail-stop processor model, where a processor may stop at any moment during the computation and once stopped it takes no further actions. 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 its size \( |F| \) to be the number of processor stop-failures. Our failure model is the set of all failure patterns \( F \), such that \( |F| < P \). Note that the number of failures \( f \) is never known to the processors.

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

\[
\text{Write-All: Using } P \text{ fail-stop processors write the value 1 into all locations of a shared array of size } N.
\]

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 $i$th 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 measure algorithm complexity as work (or available processor steps [23]), and instant and total memory access 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:

$$W = W(N, P, f) = \max_{|F| \leq f} \left\{ \sum_{1 \leq i \leq \tau(F)} P_i(F) \right\}.$$

We are interested in robust algorithms (cf. [22]) that achieve fault-tolerance while degrading the optimal work by at most a polylogarithmic factor. For Write-All, an algorithm is robust if its work is $W = O(N \log^{O(1)} N)$.

We now define the concurrency measures that assess the number of “redundant” reads and writes (the total memory access concurrency is as in [17]).

**Definition 2.** 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 $\tau(F)$, if at time $i$ ($1 \leq i \leq \tau(F)$), $P^r_i(F)$ processors complete reads from $N^r_i(F)$ distinct locations and $P^w_i(F)$ processors complete writes to $N^w_i(F)$ distinct locations, then we define:

(i) **instant read concurrency** $\hat{rc}$ as:

$$\hat{rc} = \hat{rc}(N, P, f) = \max_{|F| \leq f, 1 \leq i \leq \tau(F)} \left\{ P^r_i(F) - N^r_i(F) \right\},$$

(ii) **total read concurrency** $rc$ as:

$$rc = rc(N, P, f) = \max_{|F| \leq f} \left\{ \sum_{1 \leq i \leq \tau(F)} \left( P^r_i(F) - N^r_i(F) \right) \right\}.$$
(iii) **instant write concurrency** \( \widehat{wc} \) as:

\[
\widehat{wc} = \widehat{wc}(N, P, f) = \max_{|F| \leq f, 1 \leq i \leq \tau(F)} \left\{ \frac{P^w_i(F)}{N^w_i(F)} \right\},
\]

(iv) **total write concurrency** \( wc \) as:

\[
wc = wc(N, P, f) = \max_{|F| \leq f} \left\{ \sum_{1 \leq i \leq \tau(F)} \left( \frac{P^w_i(F)}{N^w_i(F)} \right) \right\}.
\]

3. Algorithm KMS

Algorithm KMS [17] 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 [17] 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 given in Figure 3.1 and 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 [17], in Sections 3.2 and 3.3.

Algorithm KMS uses several data structures represented as binary trees.

- The **progress tree** records the progress of the computation and it is used to balance processor loads in a divide-and-conquer fashion.
- The **processor enumeration tree** is used to estimate the number of operational processors and to renumber the processor compactly.
- 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.
- 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 [17] can proceed to Section 4.
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

Figure 3.1: Top level control structure of algorithm KMS.

3.1. Top Layer Control Structure

The top level algorithm (Figure 3.1) consists of the main loop that iterates through four phases until the Write-All problem is solved (this is based on algorithm W [22]). The algorithm uses two complete binary trees: the processor enumeration tree with \( P \) leaves, and the progress tree with \( H \) leaves \((1 \leq H \leq N)\), 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.

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

SIU 2004
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 $k$th 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 $j$th processor of the PPT broadcasts to the $j$th processor of each level below its own (in a left-to-right numbering within each level). The algorithm makes $\lceil \log p \rceil + 1$ 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 $\ell$ that should have broadcast to $k$ is at or below the level that effected the write, then $k$ assumes the position of processor $\ell$ 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 [17]) 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.

SIU 2004
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, we established in [15], for algorithm KMS and for simulations using this algorithm.

**Theorem 1.** [15] Algorithm KMS solves the Write-All$(N, P, f)$ problem with total write concurrency $wc \leq f$, total read concurrency $rc \leq 7f \log N$ and work
\[ W = O(N + P \log^2 N \log^2 P / \log(P/f)) \] when $f \leq P/\log P$, and
\[ W = O(N + P \log^2 N \log^2 P / \log \log P) \] when $P/\log P < f < P$.

**Theorem 2.** [15] Any $N$-processor, $r$-time EREW PRAM algorithm can be simulated on a fail-stop $P$-processor CRCW PRAM with total write concurrency $wc \leq f$, total read concurrency $rc \leq 7f \log N$, and with work
\[ W = O(r \cdot (N + P \log^2 P \log^2 N / \log(P/f)) \] when $f \leq P/\log P$, and work
\[ W = O(r \cdot (N + P \log^2 P \log^2 N / \log \log P)) \] otherwise.

Observe that these results, only demonstrate how the work and the total memory access concurrency is bounded; they do not show how the instant memory access concurrency is bounded. The only obvious observation is that concurrency is bounded by the maximum number of processors, $rc, wc \leq P$. In fact, when the number of failures $f$ is high, e.g., linear, $f = \Theta(P)$, then instant memory access concurrency can be $\Theta(P)$.

4. Algorithm $\tilde{\text{KMS}}$

Given the high instant memory access concurrency of algorithm KMS, it was proposed in [17] to employ a pipelining technique to control both the total and instant memory access concurrency. In this section we make this proposal concrete and present the resultant algorithm that we call $\tilde{\text{KMS}}$ (pronounced “KMS wave”).

Recall that in algorithm KMS, in Phases 1, 2, and 4, processors traverse trees of logarithmic depth, with only one level of the tree being used at any given step.
The pipelining solution is to divide the processors into waves and have them move through the trees separately by introducing suitable time delays between the waves. This is the basis of algorithm $\tilde{\text{KMS}}$, and we detail it next.

Algorithm $\tilde{\text{KMS}}$ uses the data structures of the parametrized algorithm $\text{KMS}$, with $P \leq N$, and where the progress tree has $H = \max\{P, N/\log N \log \log P\}$ leaves. The array elements are associated with the leaves of this tree, with $N/H$ array elements per leaf. We divide the processors into $V$ synchronous waves each of $P/V$ processors.

Phases 1 to 4 are as in algorithm $\text{KMS}$, with the exception that we send the processors through the appropriate trees (in Phases 1, 2, and 4), or the array of elements (in Phase 3), one wave at a time. Recall that in Phases 1, 2, and 4, the traversal of each level of a tree takes $O(\log P)$ time due to the use of PPTs. To ensure that waves do not overlap we send the waves in intervals of $\Theta(\log P)$ steps — the constant hidden in $\Theta(\cdot)$ can be computed at “compile time” by carefully counting the number of low-level operations taken by algorithms $\text{CW/R}$, $\text{CR1}$, and $\text{CR2}$. This is also sufficient for Phase 3 (some optimizations are discussed later). Given that each wave has at most $P/V$ processors and that waves do not overlap, it follows that at any given step, there can be at most $P/V$ concurrent memory accesses, i.e., $\hat{r}c, \hat{w}c \leq P/V$.

It is important to note that the values computed at the nodes of the trees at each phase are correct only after the last wave finishes. Hence, we require that waves that finish earlier must wait for the last wave of processors to arrive and then begin the next phase. In the next section we show how this affects the efficiency of the algorithm, depending on the choice of $V$.

We now discuss how processors are divided into waves. For Phases 2, 3 and 4, the order in which each wave traverses the progress tree or the elements in the array is not material because the computation is monotone (this can be seen by carefully considering the phase algorithms). However, Phase 1 needs extra care, since in this phase the prefix sums are computed and a processor must not reach a node of the processor enumeration tree before any processor with smaller PID. To ensure the correct ordering of waves, the processors are assigned to waves in a left to right fashion, that is, the processors with PIDs from 1 to $\lceil P/V \rceil$ are in the
first wave, the processors with PIDs from \(\lceil P/V \rceil + 1\) to \(2\lceil P/V \rceil\) are in the second wave, and so on. This allocation is used in all phases.

We conclude with the description of optimizations that can be applied to Phase 3. Recall from the description of Phase 3 in previous section, that the choice of whether to use algorithm CR/W depends on the relation between \(u\) and \(a\), the common knowledge that processors have at the beginning of the phase about the number of unvisited leaves and available processors respectively. In a similar manner, processors decide locally whether to use waves or not. In particular, when \(u \geq a\), each leaf (array location) has at most one processor allocated to it, and in this case CW/R is not used and waves are not necessary. When \(u < a\), CW/R is used, but waves must be used only in the case where \(\lceil a/u \rceil > P/V\) to assure that in any step memory access concurrency is not greater than \(P/V\).

For the wave-based approach with \(V = \log^k N, k \geq 1\), Kanellakis et al. [17] stated the following (without a proof).

**Theorem 3.** [17] The Write-All\((N, P, f)\) problem can be solved with work 
\[W = O(N \log^{k-1} N + P \log^2 P \log^{k+1} N/ \log \log N),\]
total write concurrency \(wc \leq f\), total read concurrency \(rc \leq 7 f \log N\), and instant access concurrency \(\hat{wc}, \hat{rc} \leq P/\log^k N\).

The bound on work in Theorem 3 is not failure-sensitive, and it is not clear how to extend this analysis to iterative Write-All in a failure-sensitive way. On the other hand, the bounds given in [15] (Theorems 1 and 2) are failure-sensitive, however no bounds on instant memory access concurrency are given.

5. Analysis

In this section we analyze algorithm \(\tilde{\text{KMS}}\) and obtain failure-sensitive bounds on work for this controlled memory-access algorithm. Our analysis identifies the trade-offs, governed by the choice of the parameter \(V\), between work-efficiency and the total and instant memory access concurrency.
We also give a failure-sensitive analysis of the iterative Write-All problem with bounded total and instant memory access concurrency. This yields new failure-sensitive bounds on work of PRAM simulations with controlled total and instant memory access concurrency.

For algorithm \( \tilde{\mathrm{KMS}} \), we define \( U_i \) to be the number of unvisited leaves of the progress tree \( (U_i \leq H) \), and \( P_i \) to be the number of non-faulty processors \( (P_i \leq P) \), at the start of the \( i \)-th iteration of the main loop. We define \( \sigma_1 \) to be the time required for a processor to complete one iteration of the main loop when \( P_i < U_i \). We define \( \sigma_2 \) to be the time required for a processor to complete one iteration of the main loop when \( P_i \geq U_i \). We define a block-step to be the execution by one processor of the body of the main loop.

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

\[
\Lambda_{r,P,f} = \begin{cases} 
\log(\frac{P_r}{f}) & \text{when } f \leq \frac{P_r}{\log P}, \\
\log \log P & \text{when } f > \frac{P_r}{\log P}.
\end{cases}
\]

The main difference between algorithms \( \mathrm{KMS} \) and \( \tilde{\mathrm{KMS}} \) is the use of waves in the latter. Note that the number of block-steps is the same in both algorithms. In algorithm \( \tilde{\mathrm{KMS}} \) the processors traverse the trees as in algorithm \( \mathrm{KMS} \); the only difference is that in \( \tilde{\mathrm{KMS}} \) the processors do so in smaller groups (waves) at a time. (One can also view algorithm \( \mathrm{KMS} \) as a special case of algorithm \( \tilde{\mathrm{KMS}} \) where all processors belong to a single wave.)

Using the same analysis as in Lemma 5 in [15], we have:

**Lemma 4.** Algorithm \( \tilde{\mathrm{KMS}} \) solves the Write-All\((N, P, f)\) problem for any stop-failure pattern using work \( W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot P \log N/\Lambda_{1,P,f}) \).

We now compute the bounds on \( \sigma_1 \) and \( \sigma_2 \). As expected, these depend on \( V \), the number of waves.

**Lemma 5.** For algorithm \( \tilde{\mathrm{KMS}} \), \( \sigma_1 = O((\log N + V) \log P) \) and \( \sigma_2 = O((\log N + V) \log^2 P) \).

**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 \( H = \frac{N}{\log N \log P} \) and in Phase 3 each processor writes to \( \frac{N}{H} = \log N \log P \) array elements. The time required for each processor wave to traverse the enumeration and progress trees is \( O(\log N \log P) \) and the execution of CR/W takes \( O(\log P) \) time. The waves are sent in intervals of \( \Theta(\log P) \) steps and the trees have depth \( O(\log N) \). Since there can only be \( \log N \) active waves during any traversal, every phase will be “slowed down” by a factor of \( V/\log N \) (as compared to algorithm KMS), since the processors in the first wave must wait the processors in the last wave to finish, before they move to the next phase. Note that the number of steps from the time that the first wave finishes traversing a tree to the time the last wave finishes traversing a tree is \( O(V \log P) \) — there are \( V \) waves and each level of the tree takes \( O(\log P) \) time due to the use of PPTs.

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

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, hence \( \log P \) time must be spent at each element of the leaf and waves must be used. Since there are \( \log N \log P \) elements per leaf the worst case time to update a leaf for each processor wave is \( O(\log N \log^2 P) \). The number of steps that the first wave must wait until the last wave finishes is \( O(V \log P) \). This is because, by the time the first wave finishes, the last one is \( V \) elements “behind”, and each element takes \( \log P \) time (using algorithm CR/W). Hence, \( \sigma_2 = O(\log N \log^2 P) + O(V \log P) = O((\log N + V) \log^2 P) \).

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 \( \frac{N}{P} = O(\log N \log P) \) array elements. Then the bounds on \( \sigma_1 \) and \( \sigma_2 \) are obtained similarly to Case 1. \( \square \)

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

**Theorem 6.** Algorithm \( \tilde{\text{KMS}} \) solves the Write-All\((N, P, f)\) problem with work
\[
W = O(N(1 + V/\log N) + P(\log N + V) \log N \log^2 P/\Lambda_{1, P, f}),
\]
total write
concurrency \( wc \leq f \), total read concurrency \( rc \leq 7f \log N \), and instant read and write concurrency \( \tilde{r}c, \tilde{w}c \leq P/V \).

**Proof:** The bounds on \( wc \) and \( rc \) are obtained from Theorem 1 (see [17] and [15]). The bound on the instant concurrency follows from the fact that each wave has at most \( P/V \) processors and that waves do not overlap.

We now show the bounds on work. The idea is to combine the results of Lemmas 4 and 5. We consider two cases.

**Case 1:** \( P < \frac{N}{\log N \log P} \). Here the number of leaves in the progress tree is \( H = \frac{N}{\log N \log P} \). Combining Lemmas 4 and 5 we get \( W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot P \log N/\Lambda_{1,P,f}) = O(((\log N + V) \log P) \cdot \frac{N}{\log N} + \frac{N + V}{\log N}) \). The bounds on work are obtained by combining Case 1 and Case 2.

**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 4 and 5 we have \( W = O(\sigma_1 \cdot (H + P) + \sigma_2 \cdot P \log N/\Lambda_{1,P,f}) = O(((\log N + V) \log P) \cdot P + ((\log N + V) \log^2 P) \cdot P \log N/\Lambda_{1,P,f}) = O(P \log N + V) \log N \log^2 P/\Lambda_{1,P,f}) \).

The bounds on work are obtained by combining Case 1 and Case 2.

Theorem 6 characterizes in terms of \( V \), the number of waves, the trade-off between the work efficiency and the instant concurrency.

For \( V = \log N \), algorithms \( \text{KMS} \) and \( \tilde{\text{KMS}} \) have the same bounds on work and total access concurrency, while algorithm \( \tilde{\text{KMS}} \) additionally reduces the instant memory access concurrency from \( O(P) \) to \( O(P/\log N) \).

**Corollary 7.** Algorithm \( \tilde{\text{KMS}} \) with \( V = \log N \) solves the Write-All(\( N, P, f \)) problem with work \( W = O(N + P \log^2 N \log^2 P/\Lambda_{1,P,f}) \), total write concurrency \( wc \leq f \), total read concurrency \( rc \leq 7f \log N \), and instant read and write concurrency \( \tilde{r}c, \tilde{w}c \leq P/\log N \).

For the choice of \( V \) in Corollary 7, the time required for a processor to complete an iteration of the main loop (the bounds on \( \sigma_1 \) and \( \sigma_2 \)) are asymptotically the same for both algorithms \( \text{KMS} \) and \( \tilde{\text{KMS}} \). In other words, the cost of traversing a tree subsumes the delay introduced by the use of waves. This result is similar
Choosing $V = \log^k N$, $k \geq 1$, yields the following.

**Corollary 8.** Algorithm $\tilde{KMS}$ with $V = \log^k N$, for any $k \geq 1$, solves the Write-All$(N, P, f)$ problem with work $W = O(N \log^{k-1} N + P \log^{k+1} N \log^2 P / \Lambda_{1,P,f})$, total write concurrency $wc \leq f$, total read concurrency $rc \leq 7f \log N$, and instant read and write concurrency $\hat{rc}, \hat{wc} \leq P / \log^k N$.

This result subsumes [17] by giving a failure-sensitive bound on work.

By iteratively using algorithm $\tilde{KMS}$ with $V = \log^k N$ we obtain the following result for the iterative Write-All problem.

**Theorem 9.** The $r$-Write-All$(N, P, f)$ problem can be solved on $P$ fail-stop processors with work $W = O(r \cdot (N \log^{k-1} N + P \log^{k+1} N \log^2 P / \Lambda_{r,P,f}))$, total write concurrency $wc \leq f$, total read concurrency $rc \leq f \log N$, and instant read and write concurrency $\hat{rc}, \hat{wc} \leq P / \log^k N$, for $k \geq 1$.

**Proof:** The bounds on work and total memory access concurrency are obtained using the same analysis as that in Theorem 2 (see [15]), and by iteratively using algorithm $\tilde{KMS}$ to solve each Write-All instance (instead of $KMS$). The bound on the instant memory access concurrency follows from Theorem 6: for each Write-All instance no more than $P / \log^k N$ concurrent accesses may occur at any time for any memory location. $\square$

Using Theorem 9 we obtain a bound on the instant memory access concurrency, with failure-sensitive bounds on work, when algorithm $KMS$ is iteratively used in PRAM simulations on fail-stop PRAMS [20, 26]. By contrast, Theorem 2 gives no bound on the instant memory access concurrency.

**Theorem 10.** Any $N$ processor, $r$ parallel time EREW PRAM algorithm can be simulated on a fail-stop $P$-processor CRCW PRAM with work $W = O(r \cdot (N \log^{k-1} + P \log^{k+1} N \log^2 P / \Lambda_{r,P,f}))$, total write concurrency $wc \leq f$, total read concurrency is $rc \leq 7f \log N$, and instant read and write concurrency $\hat{rc}, \hat{wc} \leq P / \log^k N$, for $k \geq 1$.

SIU 2004
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 \([20, 26]\). The result then follows from Theorem 9.

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. An alternative is first to convert the simulated algorithm into an equivalent EREW algorithm using a standard PRAM conversion technique \([21]\) at the expense of increasing its time (and work) complexity by a logarithmic factor. Then, the simulation obtains the same concurrency bounds as in Theorem 10.

6. Conclusion

In this paper we consider the Write-All and iterative Write-All problems in the fail-stop CRCW PRAM model where both the instant and total memory access concurrency need to be bounded. We detail and analyze a robust algorithm, called algorithm \( \tilde{\text{KMS}} \), based on algorithm KMS and the pipelining technique proposed in \([17]\). Using our technique \([14, 15]\) for analyzing work-performing algorithms by separately assessing the costs of tolerating failures and the costs of achieving perfect load balancing, we obtain failure-sensitive bounds on work for Write-All and iterative Write-All. The bounds on memory access concurrency capture trade-offs between the total concurrency and fault-tolerance, and between the instant concurrency and work efficiency. Our failure-sensitive solution for the iterative Write-All problem also leads to failure-sensitive bounds on the work of PRAM simulations on fail-stop PRAMS with bounded instant and total concurrency. 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.

Acknowledgments. 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.
References


SIU 2004


Authors addresses:

Chryssis Georgiou, University of Cyprus, Cyprus,  
Email: chryssis@ucy.ac.cy

Alexander Russell, University of Connecticut, USA,  
Email: acr@cse.uconn.edu

SIU 2004
Alexander A. Shvartsman, University of Connecticut and MIT CSAIL, USA,

Email: aas@cse.uconn.edu and alex@theory.csail.mit.edu

SIU 2004