Non-Adaptive Algorithms For The Write-All Problem

by

Yoav Freund

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Non-Adaptive Algorithms For The Write-All Problem

Yoav Freund
Master of Science
Graduate Department of Computer Science
University of Toronto
2010

In the *Write-All* problem, processes are required to write 1 to each cell of a shared-memory bit-array, initially containing 0's. An array of size n represents the tasks to be completed in one step of an n-process parallel machine. The value of each cell indicates whether the corresponding task has been completed.

We focus on a class of asynchronous non-adaptive algorithms that solve *Write-All*. In the model we consider, each process examines the array in a fixed order and writes 1 to every 0-valued cell it encounters. In this model, when a number of processes read a 0 from a cell, they all end up writing to that cell. Our goal is to find orderings that minimize such redundancy. In this paper, we introduce our model, present and analyze algorithms that minimize redundancy, and prove lower bounds on the work of algorithms in our model.

Acknowledgements

I would like to thank my supervisor, Faith Ellen, for the excellent guidance, the insightful discussions, and the invaluable on-going feedback. The past year and a half of joint work has taught me so much, and I am grateful for it.

I would also like to thank the second reader of my thesis, Eric Mendelsohn, for his time and effort.

Last but not least, I would like to take this opportunity to thank my family and friends for supporting me in every step I take.

Contents

Chapter 1

Introduction

In this chapter, we will introduce the *Write-All* problem, and survey past work related to it.

1.1 The Write-All Problem

The *Write-All* problem can be described as follows:

Given a shared-memory bit array, X, of size n, initialized to contain only 0's, write 1 to all of X's cells.

A common variant of the problem is *Certified Write-All*, where we need to write 1 to a binary flag, *Done*, after writing 1 to all of X's cells. Both variants of the problem appear in the literature, and are equally hard.

The n array cells represent n tasks to be completed in one step by an n-process synchronous system. The value 1 is written to a cell to indicate that the corresponding task has been completed. When all cells have been set to 1 we know that the step is completed.

One popular parallel computing model is the PRAM, as defined in [4], a fault-free synchronous shared-memory model. This PRAM model is good for developing and analyzing parallel algorithms, but it fails to describe the behaviour of realistic systems that are not necessarily synchronous and are subject to process failures. A solution to *Write-All* can be used to simulate this ideal model on more realistic models.

Shvartsman [13] used a solution to the *Write-All* problem as a synchronization primitive to simulate a fault-free synchronous system on a fault-prone synchronous system. Martel et al. [12] did the same on a faulty asynchronous system. They applied a variant of *Certified Write-All* to simulate a synchronous program step by step. In their simulation, the cells of X and the flag *Done* contain integer values instead of bits. A processes writes s to $X[i]$ to indicate that it completed task i of step s, and the value of *Done* is set to s to indicate that the s'th simulated step was completed.

Another possible application of Write-All is a parallel search. Consider searching a large data set, stored in n different remote file servers. In order for a process to search the data in a file server, the process must first copy the data to its local storage. The process then searches the data and saves the results in shared-memory. Once all n servers have been searched, we can combine the results from all n servers. A solution to Write-All can be used to perform the parallel search. The i'th cell of the array X indicates whether a search over the i'th server has been completed. When all of X's cells are set to 1, we know that all servers have been searched.

In the example above, writing 1 to a cell is associated with searching a remote file server. Searching a remote file server involves copying the data from the remote file server to a local storage, which can be extremely expensive. Therefore, when evaluating a Write-All solution for this scenario, a good complexity measure would be the total number of write operations performed. In the earlier example, writing 1 to a cell was associated with performing a single low-level instruction. In that case, a more suitable complexity measure would be the total number of steps performed by the Write-All solution.

Generally, we use multi-process parallel systems to get wait-free solutions that are faster than the best known sequential solution. A *wait-free* solution is guaranteed to be correct as long as at least one process survives. The problem is that as the number of processes increases, more processes may take redundant steps. For example, consider an arbitrary algorithm that solves *Write-All* on an asynchronous system with p processes and an array X of length 1. Since each process must be able to complete the work in case all other processes fail, an adversary can allow each process to run and then pause it just as it is about to write 1 to X's cell. The adversary's next step is to let all p processes

write, causing the algorithm to perform $\Omega(p)$ steps. This simple example shows us how increasing the number of processes increases the number of redundant steps.

When solving *Write-All* on a multi-process system, processes must coordinate their work in order to reduce redundancy. On the one hand, coordinating between the processes, while maintaining efficiency, becomes increasingly more challenging as the number of processes grows. On the other hand, the whole point of parallel systems is to distribute work among many processes in order to get faster solutions. Researchers study this trade-off with the goal of finding fault-tolerant optimal solutions on various parallel models with as many processes as possible.

1.2 Related Work

The *Write-All* problem was introduced in 1989 by Shvartsman & Kanellakis and they were responsible for most of the early results related to it. In their paper [7], they studied the problem on p-process synchronous CRCW shared-memory models with process failures (processes may crash at any point of the execution as long as at least one survives). They presented algorithm W, the best deterministic synchronous algorithm known to date, which performs $O(n\log(n) + p\log^2(n))$ work.

Algorithm W is an iterative algorithm that uses two binary trees stored in shared memory to keep track of the computation as it progresses. The *progress tree* is used for keeping track of the work done. The n leaves of the tree are the cells of the input array X, and each internal node contains a count of the 1-valued leaves in its subtree.
The *process enumeration tree* is used for keeping track of the available processes. The p leaves of the process enumeration tree correspond to the p processes in the system. Each node in the tree contains a *count* of active processes in its subtree. Each node also stores a *timestamp* which allows us to detect "old" counts without re-initializing the tree between iterations.

Each iteration of the algorithm works as follows: processes are allocated to cells of the input array, they do the work (i.e., write 1 to a cell) and update the trees.
The algorithm terminates when the root of the progress tree contains the value n, indi-

cating that all n cells have been set.

First, we will describe how the processes update the trees, and then describe the way process allocation works.

The progress tree is updated as follows: After a process does the work at a leaf it walks up the tree towards the root and updates the values of the visited nodes. (The new value of each node is the sum of its children's values.) Internal nodes of the progress tree contain underestimates of the work done because processes may fail on their way up the tree.
The process enumeration tree is updated in a similar way. Each process starts from its corresponding leaf and makes its way up to the root, updating the timestamp and setting the count of each node to the sum of its children's counts (treating "old counts" as 0). The counts in the process enumeration tree are overestimates because a process may fail on its way up but other processes might "carry its count" up the tree.
In addition to updating the tree, each process computes an estimated rank, that is, a unique integer in the range $[1, c]$, where c is the count at the root of tree after the update.

The process allocation works as follows: Initially processes are implicitly allocated to cells based on their identifiers. (Each process knows its unique integer identifier in the range $[1, p]$.) In later iterations, the algorithm uses the information in the progress tree to allocate processes to cells in proportion to the estimated remaining work in each subtree. To understand how the allocation works, consider the following example:
Suppose the values stored at the left and right children of the progress tree's root are $\frac{n}{4}$ and $\frac{n}{8}$, respectively, and suppose we start with c processes at the root. Since there might be incomplete work in both subtrees, we want to determine how many of the c processes should go left and how many should go right. The root's children have $\frac{n}{2}$ leaves in their subtrees, so we compute the following ratio $(\frac{n}{2} - \frac{n}{4})/(\frac{n}{2} - \frac{n}{8}) = 2/3$, and allocate $\frac{2c}{5}$ processes to the left child and $\frac{3c}{5}$ to the right child.

The allocation is done in parallel: Each process starts at the root of the progress tree, assuming there are c active processes (where c is the count at the root of process enumeration tree). It computes the ratio as described above, and determines whether to go left or right according to its estimated rank (computed when the process enumeration tree

was last updated). Each process repeats this procedure until it reaches a leaf. When all active processes reach a leaf, the allocation is determined.

In addition to algorithm W, Shvartsman and Kanellakis used an adversary argument to obtain an $\Omega(n \log(n)/\log(\log(n)))$ work bound for any n-process fault-tolerant CRCW synchronous algorithm (deterministic or randomized) that solves Write-All.

Kedem et al. [8] modified algorithm W slightly to get an n-process algorithm that performs $O(\frac{n \log^2(n)}{\log(\log(n))})$ work, when $p \leq n$. In their version they associate each leaf of the progress tree with a block of $\log_2(n)$ cells of X. This modification does not affect the asymptotic work performed by the algorithm because each process already performs $\Omega(\log(n))$ work in each iteration when updating the trees. They also improved the work lower bound, showing that any synchronous solution to *Write-All* requires $\Omega(n \log(n))$ work.

A number of researchers have continued to study algorithm W, presenting different analyses and adapting the algorithm to other synchronous shared-memory models. Georgiou, Russell and Shvartsman [5] analyzed the performance of algorithm W as a function of n, p and f, where f is the number of process failures that occur during an execution. Kedem, Palem and Spirakis [9] performed an average case analysis of algorithm W. Buss et al. [2] adapted the algorithm to work on systems with faulty restartable processes.

In [2], Buss et al. considered two related models: an asynchronous model and a synchronous fail-restart model. In the fail-restart model, processes that fail (i.e., crash) can be restarted, in which case, their program counters are reset and the information that was stored in their local memories is lost. Buss et al. showed an $n - p + \Omega(p \log(p))$ work bound for any algorithm that solves *Write-All* on the synchronous fail-restart model. Their adversarial approach works as follows: First, the adversary allows the processes to set $n - p$ of the array cells. It then restarts all processes and determines which process is about to write to which cell. A simple averaging argument shows that the adversary can stop at most $\lceil \frac{p}{2} \rceil$ processes and cause at least half of the unset cells to remain unset. The adversary applies this strategy $\log_2(p)$ times to get their lower bound.

In the same paper, Buss et al. considered a model where processes can take a snapshot

and process the entire shared memory at unit cost. In this model, they presented an algorithm that matches their lower bound. This result shows that any improvement to their lower bound must use the fact that a process cannot "learn everything" about the input array between its write operations.

The work on synchronous models left a small gap between the $\Omega(n\log(n))$ and the $O(\frac{n\log^2(n)}{\log(\log(n))})$ work bounds, and the *Write-All* problem is considered to be essentially solved on synchronous models. A complete and more detailed survey can be found in [6].

When working on an asynchronous model, we think of an execution as an interleaving of process instructions. Any interleaving is possible as long as each process executes its instructions in order. In this model, arbitrary delays can occur between the interleaved instructions.

The asynchronous model is related to the synchronous fail-restart model since every execution on the synchronous fail-restart model has an equivalent execution on the asynchronous model (process down-time in the fail-restart model is process delay in the asynchronous model).
Buss et al. [2] used this observation to show that their $\Omega(n\log(n))$ work bound for the synchronous fail-restart model applies to the asynchronous model as well.

Martel et al. [12] presented and analyzed a randomized asynchronous algorithm that performs expected work $O(n)$ when $p \leq \frac{n}{\log(n)\log^*(n)}$. In their algorithm, they divide the array X into blocks of size $\log(n)$, and associate each block with a different leaf of a binary tree. The nodes in the tree contain either 0 or 1, indicating whether the work in a subtree is done. Each process examines the root of the binary tree, if the value at the root is 1 the process halts, otherwise it continues by randomly selecting a tree node. If the selected node is a 0-valued leaf the process does the associated work (i.e., write 1 to the $\log(n)$ cells of the associated block), and then writes 1 to the leaf. If the process selected an internal node, it examines the values of the node's children and if they are both 1 it writes 1 to the node.

Buss et al. [2] presented a deterministic asynchronous algorithm that performs $O(np^{\log(\frac{3}{2})})$

$\approx O(np^{0.59})$ work when $p \leq n$. In this algorithm, the array cells are treated as the leaves of a binary tree. Each internal node also contains a bit that is initially 0. Each process is assigned to a leaf and makes its way towards the root of the tree, writing 1 to every 0-valued node it encounters. When visiting an internal node, a process will recurse down to its 0-valued children before setting the node's value to 1.

In the same paper, Buss et al. considered the *Write-All* problem on systems with a small number of processes. They presented a simple 2-process algorithm that performs at most $n + 2$ reads and $n + 1$ writes. In that algorithm, each process starts from one end of the array and makes its way towards the other end, writing 1 to every 0-valued cell it encounters. When a process reads 1 from a cell, it knows that the whole array is set, and it exits. They present a 3-process algorithm that uses a similar approach, and results in $n + O(\log(n))$ read/write operations. In this algorithm, process P_0 starts from the left end of the array and makes its way to the right, process P_1 starts from the right making its way left, and process P_2 starts from the middle and makes its way towards the ends (alternating between left and right). If processes P_0 and P_1 collide, then the array is all set and the processes exit. When process P_0 collides with P_2, say on cell x, then the only cells that may still contain 0 are the rightmost x cells of the array. In this case, process P_0 "jumps" to cell $n-x$, and process P_2 "jumps" to cell $n-\frac{x}{2}$. The case where P_1 and P_2 collide is symmetric. Every time a collision occurs the number of unset cells is at most half of what it was when the previous collision occurred (or when the execution started). The only time a cell is written to by more than one process is in the case of a collision. Therefore, we get the $n + O(\log(n))$ bound on the number of read/write operations. We will re-visit this algorithm in chapter 3.

Anderson and Woll [1] developed the best deterministic asynchronous algorithm known to date. They started by introducing a randomized, asynchronous, p-process algorithm for input array of size $n = p^2$. The algorithm works as follows:

- We randomly select p permutations, $\pi_1, \ldots, \pi_p$, over $\{1 \ldots, p\}$.
- X is partitioned into p blocks of size p, denoted $B_1, \ldots, B_p$. Each block B_i has a corresponding *completion bit*, b_i, initially set to 0.
- For each k, process P_k examines the blocks in the order specified by π_k (i.e.,

$B_{\pi_k(1)}, \ldots, B_{\pi_k(p)}$). When examining B_i, a process reads b_i. If the value read is 0, then the process writes 1 to all cells in B_i and then sets b_i to 1.

Notice that if process P_i examines block B before it examines B', and process P_j examines B' before B, then it is impossible for both processes to write to both blocks. Anderson and Woll used this idea and presented a probabilistic argument showing that a "good" set of permutations is chosen with high probability, resulting in $O(p^2 \log(p)) = O(n \log(n))$ work.

This algorithm is presented for $p = \sqrt{n}$, but obviously works for all values of $p \leq \sqrt{n}$ ("missing" processes can be considered to be crashed). To get an algorithm that works for larger values of p, Anderson & Woll extended their approach by using two hierarchies of blocks.

In this case $n = p^{\frac{3}{2}}$, the array is partitioned into $\sqrt{p}$ "big blocks", and each big block is partitioned into $\sqrt{p}$ "small blocks" of size $\sqrt{p}$. The algorithm randomly selects $\pi_1, \ldots, \pi_{\sqrt{p}}$, permutations over $\{1, \ldots, \sqrt{p}\}$, and each process uses one permutation for the ordering of the "big blocks" and one for the ordering of the "small blocks" within each big block. (Each process uses a different ordered pair of permutations (π_i, π_j).) This algorithm performs $O(p^{3/2} \log^2(p)) = O(n \log^2(n))$ work (with high probability).

Anderson and Woll extended their approach to work for all values of $p \leq n$. This extension resulted in a deterministic, yet non-constructive, algorithm called AW^T.

In AW^T, they used a tree of nested blocks. They chose a constant q that parameterizes the algorithm, and used a q-ary tree of blocks. The tree has height $\log_q(n)$, and the input array cells are the leaves of the tree. The algorithm uses $\pi_1, \ldots, \pi_q$, q permutations over q, as follows: Each process, P_i, uses a different q-tuple of permutations $(\pi_{i_1}, \ldots, \pi_{i_q})$. Each process traverses the tree skipping sub-trees whose completion bit is set. P_i examines the children at level l according to the order specified by π_{i_l}.

They analyzed this algorithm and showed that, for every $\epsilon > 0$, there exists a constant q such that the algorithm performs $O(n^{1+\epsilon})$ work. Finding a "good" set of permutations, $\pi_1, \ldots, \pi_q$, is done using brute force (its existence is guaranteed) and is considered a part

of the pre-processing. The work required to find these permutations is not counted as part of the work of the algorithm.

The analysis of algorithm AW^T and the resulting work bound are based on the existence of a "good set of permutations". Anderson and Woll defined what a good set of permutations is, but they did not attempt to construct such a set deterministically.

An alternative approach to the *Write-All* problem is to look for algorithms that perform $O(n)$ work using the largest possible number of processes, p. The $\Omega(p \log(p))$ work bound, mentioned earlier, implies that no such solution exists when $p \in \omega(n/\log(n))$. Malewicz [11] presented an asynchronous algorithm that performs $O(n)$ work when p is $O((n/\log(n))^{1/4})$. Kowalski and Shvartsman [10] improved on that result, showing such an algorithm for $p \in O(n^{\frac{1}{2+\epsilon}})$.

1.3 Statement of Results

We will introduce a class of algorithms that are similar to ones presented by Anderson and Woll. The performance of our algorithms depends on a chosen sequence of permutations. These algorithms are non-adaptive in the sense that the permutations are chosen at the beginning of an execution and do not change during the execution of the algorithm.

In chapter 2, we introduce our non-adaptive algorithms. In these algorithms, each process examines the cells of X in some fixed order, writing 1 to every 0-valued cell it encounters. We represent an algorithm as a sequence of permutations, one permutation for each process, that determines the order in which processes examine the cells of X. We use the total number of write operations as a work measure, which makes our algorithms suitable for applications of *Write-All* where performing a single task is expensive. We formally define the model, introduce convenient notation, and complete the chapter by stating and proving a number of properties of our algorithms.

In chapter 3, we begin by presenting the 2-process algorithm, mentioned in [2], in our model. We formally prove that this algorithm is optimal in our model. We continue by

incrementally developing 3-process algorithms and analyzing their performance. Later in the chapter, we extend these algorithms to systems with 4 processes. We show a lower bound on the work of any 3-process algorithm (conforming to our model), and extend this bound to systems with an arbitrary number of processes.

In chapter 4, we summarize our results, relate them to past work and discuss the open questions that arise from our work.

Chapter 2

The Non-Adaptive Model

In this chapter, we formally define our non-adaptive model.

2.1 Notation and Conventions

Before we introduce our model, we would like to specify some of the notation and conventions we will use.

For any positive integer n, we use $[n]$ to denote the set $\{1, \ldots, n\}$.

We represent permutations using sequences. A permutation $\pi : [|S|] \to S$ is represented using the following sequence:

$$\sigma = \langle \pi(1), \cdots, \pi(|S|) \rangle$$

Since there is a one to one correspondence between permutations and their sequence representations, we use the two interchangeably and we talk about sequence operations being applied to permutations. For example:

- *Reversal.* Let π be a permutation and let σ be its sequence representation. The reversal of π, denoted as π^R, is the permutation represented by σ^R.

- *Concatenation.* Let π_1 and π_2 be permutations over disjoint sets, S_1 and S_2, and let σ_1 and σ_2 be their respective sequence representations. The concatenation of π_1 and π_2, denoted as π_1, π_2, is the permutation represented by σ_1, σ_2 (the concatenation

of σ_1 and σ_2).

The following notation makes it convenient to discuss the ordering of elements in a permutation (or its sequence representation).

Definition 2.1. *Let π be a permutation over a set S. For any two elements $x, y \in S$, we use the notation $x <_\pi y$ to mean x appears to the left of y in the sequence representation of π.*

Similarly, we use the notation $x \leq_\pi y$ to say that either $x = y$ or $x <_\pi y$.

2.2 Model of Computation

We consider an asynchronous shared-memory model of computation:

- We have p processes, denoted $P_1, \ldots, P_p$.
- The processes communicate using read and write operations on the shared-memory array, $X[1, \ldots, n]$.
- An execution in this model is a sequence of read and write operations performed by the processes. In our analysis, we assume that the execution is determined by an adversary.
- Any number of processes may crash at any point of the execution, as long as at least one process survives.

2.3 Non-Adaptive Algorithms

In general, a non-adaptive algorithm is an algorithm that performs that same computation regardless of what happens in an execution. In our context, a *non-adaptive algorithm* is an algorithm where each process traverses the array X in a (fixed) pre-specified order, and writes 1 to every 0-valued cell it encounters.

Our non-adaptive algorithms are non-uniform. That is, we define different algorithms for different values of n (where n is the length of the array X). For a non-adaptive algorithm A, we use the notation $|A|$ to denote the length of the array X. We represent a

p-process non-adaptive algorithm, A, as a sequence of p permutations, $\langle \pi_1, \ldots, \pi_p \rangle$, over $[|A|]$. The permutation π_k specifies the order in which P_k traverses the array. The following pseudo code, for P_k, should make our definition of a non-adaptive algorithm clear:

```
FOR i = π_k(1), ..., π_k(n):
   IF X[i] == 0 THEN X[i] = 1
```

Since each process examines all cells, we know that, as long as at least one process survives, a non-adaptive algorithm solves the *Write-All* problem.

Example 1: Consider the following execution of an algorithm with $n = 1$ and $p = 2$:

P_1 reads 0 from $X[1]$
P_2 reads 0 from $X[1]$
P_1 writes 1 to $X[1]$
P_2 writes 1 to $X[1]$

In this example, P_2 writes to $X[1]$ after it has already been set to 1, performing redundant work. This is the simplest example of how our algorithm may perform redundant work. In general, a number of processes may write to the same cell if they all read it before it is set to 1. Our goal is to find algorithms that minimize the total number of write operations performed.

Given $\pi_1, \ldots, \pi_p$, we let an adversary choose an execution of the algorithm. An *execution* is an interleaving of the read/write operations that satisfies the following two conditions: (1) For all $1 \leq k \leq p$: P_k reads the cells in the order π_k. (2) A process writes to a cell if and only if it read 0 from that cell in its preceding operation.

Definition 2.2. *For a given execution E, we define its associated permutation, denoted α_E, to be the permutation over $[n]$ that describes the order in which the array cells get set (i.e., change their value from 0 to 1).*

In other words, during the execution E, $X[\alpha_E(1)]$ is the first cell to get set, $X[\alpha_E(2)]$ is the second, ... , and $X[\alpha_E(n)]$ is the last cell to get set. The next lemma follows directly

from the definition of α_E.

Lemma 2.1. *If P_k writes to $X[x]$ during execution E, then*

$$y <_{\pi_k} x \;\Rightarrow\; y <_{\alpha_E} x.$$

Proof. If P_k writes to $X[x]$, then P_k must have read $X[x]$ when its value was 0. If $y <_{\pi_k} x$, then P_k examined $X[y]$ before reading $X[x]$. Therefore, when P_k reads $X[x]$ the cell $X[y]$ is already set, and, by definition, we have $y <_{\alpha_E} x$. □

Each execution has exactly one associated permutation, but multiple executions can have the same associated permutation. To see that, consider the following two executions, with $n = 1$ and $p = 2$:

Execution E_1	Execution E_2
P_1 reads 0 from $X[1]$	P_1 reads 0 from $X[1]$
P_2 reads 0 from $X[1]$	P_1 writes 1 to $X[1]$
P_1 writes 1 to $X[1]$	P_2 reads 1 from $X[1]$
P_2 writes 1 to $X[1]$	

E_1 and E_2 are two different executions with the same associated permutation ($\alpha_{E_1} = \alpha_{E_2} = \langle 1 \rangle$). Notice that E_1 results in two writes and E_2 results in one.

Definition 2.3. *We define the associated permutation set of a non-adaptive algorithm $A = \langle \pi_1, \ldots, \pi_p \rangle$, denoted PS_A, as:*

$$PS_A = \{\alpha_E \mid E \text{ is an execution of } A\}$$

Our next step is to associate a unique execution with each associated permutation.

Given an algorithm A and an arbitrary permutation $\alpha \in PS_A$ we would like to find an execution of A, whose associated permutation is α, that maximizes the total number of writes. We construct such an execution greedily as follows:

Each process, P_k, reads a 0 from the cell $X[\pi_k(1)]$.
FOR $i = 1, \ldots, n$:
  For each P_k, if the last cell read by P_k is $X[\alpha(i)]$, then
  P_k writes 1 to $X[\alpha(i)]$ and continues reading the cells of X
  until it reads a 0.

We denote this execution as $E(A, \alpha)$. The next lemma is used to show that $E(A, \alpha)$ maximizes the number of write operations among all executions whose associated permutation is α.

Lemma 2.2. *Consider any non-adaptive algorithm A and any $\alpha \in PS_A$. Suppose that*

$$y <_{\pi_k} x \;\Rightarrow\; y <_{\alpha} x, \quad \text{for all } y \in [n].$$

Then P_k writes to $X[x]$ in execution $E(A, \alpha)$.

Proof. By construction, write operations to $X[\alpha(i)]$ in $E(A, \alpha)$ must occur during the i'th iteration, for all $i \in [n]$.

Let P_k be an arbitrary process, and let $i \in [n]$ be arbitrary. Suppose that, for all $y \in [n]$, $y <_{\pi_k} \alpha(i) \Rightarrow y <_{\alpha} \alpha(i)$. Consider the beginning of iteration i of $E(A, \alpha)$, and let x be the index of the last cell read by P_k. P_k read a 0 from $X[x]$, otherwise it would have kept on reading. By the definition of a non-adaptive algorithm, we know that after P_k reads a 0 from $X[x]$, P_k writes 1 to $X[x]$.

If $x <_{\pi_k} \alpha(i)$, then, by our assumption, $x <_{\alpha} \alpha(i)$. In other words, $x = \alpha(j)$, for some $j < i$. Since all write operations to $X[\alpha(j)]$ occur during iteration j, we get that, during iteration j, P_k writes 1 to $X[x]$ and then continues reading the array. This implies that, at the beginning of iteration i, $X[x]$ is not the last cell read by P_k, which is a contradiction.

If $\alpha(i) <_{\pi_k} x$, then , by definition of a non-adaptive algorithm, before P_k reads x, P_k reads $X[\alpha(i)]$ and if it is 0, writes 1 there. This means that, at the beginning of iteration i, the cell $\alpha(i)$ has already been set. This is a contradiction, since write operations to $X[\alpha(i)]$ can only occur during iteration i.

We conclude that $x = \alpha(i)$, which implies that P_k writes to $\alpha(i)$ during iteration i. □

Corollary 1. *$E(A, \alpha)$ maximizes the total number of writes out of all executions of A whose associated permutation is α.*

Proof. Let E' be an arbitrary execution of A with $\alpha_{E'} = \alpha$. By lemma 2.1, if P_k writes to x during E', then every $y <_{\pi_k} x$ satisfies $y <_\alpha x$. Therefore, by lemma 2.2, P_k writes to x during $E(A, \alpha)$. This implies that $E(A, \alpha)$ results in at least as many writes as E'. □

Given a non-adaptive algorithm A and an associated permutation α, we let $W(A, \alpha)$ denote the total number of writes performed during the execution $E(A, \alpha)$.

Definition 2.4. *We define the work of a non-adaptive algorithm A, denoted $W(A)$, to be $\max_{\alpha \in PS_A} \{W(A, \alpha)\}$.*
A is an optimal non-adaptive algorithm, if $W(A) \leq W(A')$, for every non-adaptive algorithm A' with $|A| = |A'|$.

2.4 Properties of Non-Adaptive Algorithms

In chapter 3, we prove upper and lower bounds on the work of a number of non-adaptive algorithms. The next two lemmas will be used in those proofs. Lemma 2.3 tells us that if two processes examine two cells in a different order, then it is impossible for both processes to write to both cells. Lemma 2.4 tells us that an adversary can make all processes write to the last remaining unset cell.

Lemma 2.3. *For any non-adaptive algorithm, if both P_i and P_j write to cells x and y in some execution, then:*

$$x <_{\pi_i} y \quad \Leftrightarrow \quad x <_{\pi_j} y$$

Proof. Suppose P_i and P_j both write to cells x and y in some execution. To obtain a contradiction, assume, without loss of generality, that $x <_{\pi_i} y$ and $y <_{\pi_j} x$. Then the following operations must happen in the following order during the execution:

(1) P_i reads 0 from x and sets it to 1
(2) P_i reads 0 from y (the value must be 0)
(3) P_j writes 1 to y
(4) P_j examines x

When P_j examines cell x, the cell is already set, and P_j will not write to it. This is a contradiction. □

Lemma 2.4. *For any non-adaptive algorithm, A, and any $\alpha \in PS_A$, all processes write to $\alpha(n)$ during the execution $E(A, \alpha)$.*

Proof. This is a direct consequence of lemma 2.2, since, for every $y \in [n]$, if $y \neq \alpha[n]$ then $y <_\alpha \alpha(n)$. □

Re-ordering the sequence $\pi_1, \ldots, \pi_p$ does not change the amount of work performed by an algorithm.

Let A be an arbitrary non-adaptive algorithm with $|A| = n$, and let $\phi : [n] \to [n]$ be an arbitrary bijection.

Lemma 2.5. *If $A' = \langle \pi_{\phi(1)}, \ldots, \pi_{\phi(p)} \rangle$, then $W(A) = W(A')$.*

Proof. For every execution of A there is an equivalent execution of A'. When P_k performs an operation in an execution of A, $P_{\phi(k)}$ performs the same operation in the equivalent execution of A'. □

Similarly, applying the same permutation to each of the permutations $\pi_1, \ldots, \pi_p$ does not change the work performed by an algorithm.

Lemma 2.6. *If $\phi(A) = \langle \phi(\pi_1), \ldots, \phi(\pi_p) \rangle$, then $W(A) = W(\phi(A))$*

Proof. For every execution of A there is an equivalent execution of $\phi(A)$. When a process reads/writes from/to cell x in an execution of A, the same process performs the same operation on cell $\phi(x)$ in the equivalent execution of $\phi(A)$. □

Lemma 2.6 allows us to consider an arbitrary non-adaptive algorithm and assume, without loss of generality, that π_1 is the identity permutation. We will use it, and lemma 2.5, in our lower bound proofs in chapter 3.

For any (non-empty) set $S \subseteq [n]$, we can talk about $A|S$, the *partial algorithm* of A restricted to S. $A|S$ is represented as $\langle \pi'_1, \ldots, \pi'_p \rangle$, where each π'_k is a permutation over S satisfying $x <_{\pi'_k} y$ iff $x <_{\pi_k} y$.

Given a partial algorithm $A|S$ and a bijection $\phi' : S \to R \subseteq [n]$, we represent the partial algorithm $\phi'(A|S)$ as $\langle \phi'(\pi'_1), \ldots, \phi'(\pi'_p) \rangle$, where each π'_k is a permutation over S satisfying $x <_{\pi'_k} y$ iff $x <_{\pi_k} y$.

For any subset $S \subseteq [n]$ and any bijection $\phi : S \to [|S|]$, we can transform a partial algorithm $A|S$ into an algorithm $\phi(A|S)$. We define the work of a partial algorithm, $A|S$, denoted $W(A|S)$, to be $W(\phi(A|S))$, where ϕ is some bijection from S to $[|S|]$. For every execution, E, of $\phi(A|S)$, there is an execution, E', of A that results in at least as much work. Given E, it is easy to construct an execution E' such that, if a process reads/writes from/to cell x in E, the same process performs the same operation on cell $\phi^{-1}(x)$ in E' (where ϕ^{-1} denotes the inverse of ϕ). The work performed during E' may exceed the work performed during E, because cells in $[n] - S$ need to be set as well. Therefore, by definition, $W(A|S) \leq W(A)$.

In the chapter 3 we present some of our analyses in terms of *redundant writes*. The number of redundant writes performed by an algorithm is the total number of writes performed minus n. An alternative way of computing the number of redundant writes is

$$\sum_{x=1}^{n} (\#\text{ of processes writing to cell } x - 1).$$

We can also compute an upper bound on the number of redundant writes as follows:

$$\sum_{i,j \in [n], i \neq j} \#\text{ of cells written to by both } P_i \text{ and } P_j$$

We will use these counting methods in chapter 3, when proving upper bounds on the work of non-adaptive algorithms.

For every execution, E, of a partial algorithm $A|S$, there is an equivalent execution, E', of A, that results in at least as many redundant writes. The next lemma formalizes this idea.

Lemma 2.7. *For any non-adaptive algorithm, A, and any partial algorithm $A|S$:*

$$W(A|S) - |S| \leq W(A) - |A|$$

Proof. Given an execution, E, of $A|S$, we construct an execution, E', of A that performs at least as many redundant writes. For every $x \in S$, if P_k writes to x during E, we make P_k write to x in E'. For every $x \notin S$, we make the processes examine x as soon as possible after all $y \in S$ such that $y <_{\pi_k} x$ have been examined. Since the number of redundant writes performed during E' equals $\sum_{x=1}^{n}$ (# of processes writing to cell $x-1$), it is clear that E' results in at least as many redundant writes as E. □

Example: Let A be the following 2-process algorithm:

$$\pi_1: \quad \langle 2,3,4,5,1 \rangle$$
$$\pi_2: \quad \langle 5,3,2,1,4 \rangle$$

Let $S = \{2,4\}$, the partial algorithm $A|S$ is:

$$\pi_1: \quad \langle 2,4 \rangle$$
$$\pi_2: \quad \langle 2,4 \rangle$$

Let E be the the following execution of $A|S$, that results in 2 redundant writes:

P_1 and P_2 read 0 from $X[2]$ (in some order)
P_1 and P_2 write 1 to $X[2]$ (in some order)
P_1 and P_2 read 0 from $X[4]$ (in some order)
P_1 and P_2 write 1 to $X[4]$ (in some order)

We will construct an execution of A that results in the same number of redundant writes as E.

We start by examining cells not in S (as many cells as we can):
P_2 reads 0 from $X[5]$ and writes 1 to it
P_2 reads 0 from $X[3]$ and writes 1 to it
We continue as in E:
P_1 and P_2 read 0 from $X[2]$ (in some order)
P_1 and P_2 write 1 to $X[2]$ (in some order)
We continue by examining cells not in S:
P_1 reads 1 from $X[3]$
P_2 reads 0 from $X[1]$ and writes 1 to it

We continue as in E, and then examine remaining cells that are not in S:

P_1 and P_2 read 0 from $X[4]$ (in some order)

P_1 and P_2 write 1 to $X[4]$ (in some order)

P_1 reads 1 from $X[5]$

P_1 reads 1 from $X[1]$

Chapter 3

Algorithms and Lower Bounds

In this chapter, we present and formally analyze non-adaptive algorithms for systems with 2, 3 or 4 processes. We also present a number of lower bounds on the work of non-adaptive algorithms.

3.1 2-Process Optimal Algorithm

We start by presenting the trivial 2-process algorithm (described in [2]) in our model. For any positive integer n, let $A_2(n)$ be the following algorithm:

$$\pi_1: \quad \langle 1, \ldots, n \rangle$$
$$\pi_2: \quad \langle n, \ldots, 1 \rangle$$

Lemma 3.1. $W(A_2(n)) \leq n + 1$

Proof. Let E be an arbitrary execution of A_2. Let $x, y \in [n]$ be two distinct values, and assume that, during E, both P_1 and P_2 write to both cells, x and y. By Lemma 2.3, $x <_{\pi_1} y \Leftrightarrow x <_{\pi_2} y$, which is a contradiction. Therefore, at most one cell gets written to by both P_1 and P_2, and the total number of writes is at most $n + 1$. □

Lemma 3.2. $W(A) \geq n+1$, *for any 2-process non-adaptive algorithm, A, with* $|A| = n$.

Proof. Let A be an arbitrary 2-process non-adaptive algorithm, and let E be an arbitrary execution of A. Each of the n cells gets written to by at least one process, and, by Lemma 2.4, both processes write to $\alpha_E(n)$ (the last cell to get set). Therefore, E results in at least $n + 1$ writes. □

Corollary 2. *Algorithm $A_2(n)$ is a work-optimal 2-process non-adaptive algorithm.*

Proof. This is a direct consequence of the two previous lemmas. □

3.2 3-Process Algorithms and Lower Bounds

In this section, we construct non-adaptive algorithms and present lower bounds for systems with 3 processes. We develop our algorithms incrementally, and analyze their performance by bounding the number of redundant writes.

The first idea that comes to mind is to represent the 3-process algorithm presented by Buss et al. [2] in our non-adaptive model. In that algorithm, two processes start from opposite ends of the array, and a third process starts in the middle and alternately goes left and right. This approach leads us to consider the following algorithm:

$$\begin{aligned}
\pi_1&: \quad \langle 1, \ldots, n \rangle \\
\pi_2&: \quad \langle n, \ldots, 1 \rangle \\
\pi_3&: \quad \langle \lfloor \tfrac{n}{2} \rfloor + 1, \lfloor \tfrac{n}{2} \rfloor, \lfloor \tfrac{n}{2} \rfloor + 2, \lfloor \tfrac{n}{2} \rfloor - 1, \ldots \rangle
\end{aligned}$$

The problem is that the original algorithm is adaptive - when processes "collide" they compute the block of possibly unset cells, "jump" to that block, and work on it recursively. It turns out that, in our non-adaptive model, the algorithm above does not perform so well. To see why that is the case, consider the following example for $n = 8$:

$$\begin{aligned}
\pi_1&: \quad \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle \\
\pi_2&: \quad \langle 8, 7, 6, 5, 4, 3, 2, 1 \rangle \\
\pi_3&: \quad \langle 5, 4, 6, 3, 7, 2, 8, 1 \rangle
\end{aligned}$$

The execution E_α, where $\alpha = \langle 1, \ldots, 8 \rangle$, looks as follows:

P_1 `reads 0 from` $X[1]$.
P_2 `reads 0 from` $X[8]$.
P_3 `reads 0 from` $X[5]$.
P_1 `writes to` $X[1]$.
P_1 `reads 0 and writes to the cells` $X[2]$ `,`$X[3]$ `and` $X[4]$.
P_1 `reads 0 from` $X[5]$.

P_1 and P_3 write to $X[5]$.
P_3 reads 1 from $X[4]$
P_1 and P_3 read 0 from $X[6]$.
P_1 and P_3 write to $X[6]$.
P_3 reads 1 from $X[3]$
P_1 and P_3 read 0 from $X[7]$.
P_1 and P_3 write to $X[7]$.
P_3 reads 1 from $X[2]$
P_1 and P_3 read 0 from $X[8]$.
P_1, P_2 and P_3 write to $X[8]$.
P_2 reads 1 from $X[7], X[6], \ldots, X[1]$.
P_3 reads 1 from $X[8]$.

Notice that, in the execution above, the processes perform redundant writes to $X[5]$, $X[6]$, $X[7]$ and $X[8]$. In general, when applying this approach to get an algorithm of size n, an adversary can cause $n/2$ redundant writes. Later in this chapter, we will present a 3-process non-adaptive algorithm that performs $O(\sqrt[3]{n})$ redundant. We will also show that any 3-process non-adaptive algorithm performs $\Omega(\sqrt[3]{n})$ redundant writes.

3.2.1 Algorithm $A_3(n)$

We start by defining an algorithm for values of n such that $n = 1 + \cdots + k$, for some positive integer k. We partition the set $[n]$ into k sub-sequences $B_1, \ldots, B_k$, with $|B_i| = i$, for all i. We refer to these sub-sequences as *blocks*, and define the algorithm $A_3(n)$ using these blocks:

$$\begin{aligned} \pi_1&: \quad \langle B_1, \ldots, B_k \rangle \\ \pi_2&: \quad \langle B_k^R, \ldots, B_1^R \rangle \\ \pi_3&: \quad \langle B_k, \ldots, B_1 \rangle \end{aligned}$$

We can partition $[n]$ into $B_1, \ldots, B_k$ in many different ways. Lemma 2.6 tells us that the chosen partition does not affect the work of the resulting algorithm. We make the convention, in all of our algorithms, to use the partition that makes $\pi_1 = \langle 1, \ldots, n \rangle$. Here is an example of $A_3(10)$:

$$\begin{aligned}
\pi_1&: \quad \langle 1, \quad 2,3, \quad 4,5,6, \quad 7,8,9,10\rangle \\
\pi_2&: \quad \langle 10,9,8,7, \quad 6,5,4, \quad 3,2, \quad 1\rangle \\
\pi_3&: \quad \langle 7,8,9,10, \quad 4,5,6, \quad 2,3, \quad 1\rangle
\end{aligned}$$

Lemma 3.3. *For any positive integer k, if $n = 1 + \cdots + k$, then $W(A_3(n)) \leq n + k + 1$.*

Proof. Let α be an arbitrary permutation in $PS_{A_3(n)}$. In order to show that $W(A_3(n)) \leq n + k + 1$, it suffices to show that the execution E_α results in at most $k + 1$ redundant writes. To see why that is true, we make the following observations about $E(A_3(n), \alpha)$:

1. At most one cell gets written to by both P_1 and P_2.
   Notice that $\pi_1 = \pi_2^R$. Therefore, for any distinct $x, y \in [n]$, $x <_{\pi_1} y \Leftrightarrow y <_{\pi_2} x$. Then, by Lemma 2.3, we cannot have both P_1 and P_2 writing to both cells x and y.

2. Cells that are written to by both P_1 and P_3 belong to the same block.
   For any $x, y \in [n]$ from two different blocks, $x <_{\pi_1} y \Leftrightarrow y <_{\pi_3} x$. Therefore, by Lemma 2.3, we cannot have both P_1 and P_3 writing to both cells x and y.

3. At most one value in each block gets written to by both P_2 and P_3.
   For any distinct $x, y \in [n]$ from the same block, $x <_{\pi_2} y \Leftrightarrow y <_{\pi_3} x$. Therefore, by Lemma 2.3, we cannot have both P_2 and P_3 writing to both cells x and y.

By observation 1, P_1 and P_2 cause at most one redundant write. Lemma 2.4 implies that P_1 and P_2 make their redundant write to the cell $\alpha(n)$, which gets written to by all three processes.

Let B_t be the block containing $\alpha(n)$. By Lemma 2.4, both P_1 and P_3 write to $\alpha(n)$. Therefore, by observation 2, P_1 and P_3 cause at most $t - 1$ additional redundant writes.

For all $j < t$, P_3 cannot write to B_j, because P_1 completes examining B_j before P_3 starts examining it. Therefore, P_2 and P_3 cannot cause redundant writes in blocks $B_1, \ldots, B_{t-1}$. By observation 3, P_2 and P_3 cause at most 1 redundant write in each block B_j, for $j \geq t$.

Therefore, the total number of redundant writes is at most $1+(t-1)+(k-t+1) = k+1$. □

So far, $A_3(n)$ is only defined for values of n such that $n = 1 + \cdots + k$, for some k. To define $A_3(n)$ for all values of n, let k be the maximum integer such that $1 + \cdots + k \leq n$, and write n as $(1 + \cdots + k) + r$, where $0 \leq r \leq k$. Once again, we partition $[n]$ into k blocks, only this time we satisfy:

$$|B_i| = \begin{cases} i+1 & , 1 \leq i \leq r \\ i & , r < i \leq k \end{cases}$$

We define π_1, π_2 and π_3 using the blocks $B_1, \ldots, B_k$ exactly as we did before. For example, if $n = 12$, we have $k = 4$ and $r = 2$. We can use the following partition:

$$B_1 = \langle 1, 2 \rangle \ , \ B_2 = \langle 3, 4, 5 \rangle \ , \ B_3 = \langle 6, 7, 8 \rangle \ , \ B_4 = \langle 9, 10, 11, 12 \rangle$$

From this partition we get the following algorithm:

$$\begin{aligned} \pi_1&: \quad \langle 1, 2, \quad 3, 4, 5, \quad 6, 7, 8, \quad 9, 10, 11, 12 \rangle \\ \pi_2&: \quad \langle 12, 11, 10, 9, \quad 8, 7, 6, \quad 5, 4, 3, \quad 2, 1 \rangle \\ \pi_3&: \quad \langle 9, 10, 11, 12, \quad 6, 7, 8, \quad 3, 4, 5, \quad 1, 2 \rangle \end{aligned}$$

The next lemma extends Lemma 3.3, and shows an upper bound on the work of $A_3(n)$, for all values of n.

Lemma 3.4. *For any positive integer n, if k is the largest integer such that $1+\cdots+k \leq n$ then:*

$$W(A_3(n)) = \begin{cases} n+k+1 & , \textit{if } n = 1 + \cdots + k \\ n+k+2 & , \textit{otherwise} \end{cases}$$

Proof. If $n = 1 + \cdots + k$, Lemma 3.3 gives us the desired work bound. Otherwise, $1 + \cdots + k < n$, and there is (at least one) block satisfying $|B_i| = i + 1$. The observations made in the proof of Lemma 3.3 hold in this case as well. Observation 2 implies that P_1 and P_3 might cause one more redundant write. The rest of the proof remains the same, and we get that the total number of redundant writes is at most $(k - t + 1) + (t + 1) = k + 2$. □

The next lemma shows us that $A_3(n)$ is the best algorithm possible when $\pi_1 = \pi_2^R$.

3.2.2 $n + \Omega(\sqrt{n})$ work bound, special case $\pi_1 = \pi_2^R$

Lemma 3.5. *Let A be an arbitrary 3-process non-adaptive algorithm with $|A| = n$, and let k be the largest integer such that $1 + \cdots + k \leq n$. If two of A's permutations are the reversal of one another, then:*

$$W(A) \geq \begin{cases} n + k + 1 & , \textit{if } 1 + \cdots + k = n \\ n + k + 2 & , \textit{otherwise} \end{cases}$$

Proof. We start by assuming that $\pi_1 = \langle 1, \ldots, n \rangle$ and $\pi_2 = \langle n, \ldots, 1 \rangle$. Lemmas 2.5 and 2.6 tell us that we can make this assumption without loss of generality. The rest of the proof is based on the proof of the Erdös-Szekeres theorem [3].

Intuition: For any $x \in [n]$, let S be a monotonically increasing sub-sequence of π_3 ending at x and let S' be a monotonically decreasing sub-sequence of π_3 ending at x. Since $\pi_1 = \langle 1, \ldots, n \rangle$ and $\pi_2 = \langle n, \ldots, 1 \rangle$, there is an execution of A where both P_1 and P_3 write to all elements of S and both P_2 and P_3 write to all elements of S'. This execution results in (at least) $|S| + |S'|$ redundant writes. Notice that all three processes write to cell x.

For each $x \in [n]$, we define the following quantities:

- $a(x)$ - The length of a longest monotonically increasing sub-sequence of π_3 ending at x (including x itself).
- $b(x)$ - The length of a longest monotonically decreasing sub-sequence of π_3 ending at x (including x itself).
- $c(x) \equiv a(x) + b(x)$

Formalizing our intuition, we observe that, for each $x \in [n]$, there exists $\alpha \in PS_A$ such that $\alpha(n) = x$, and in $E(A, \alpha)$, P_1 and P_3 cause at least $a(x)$ redundant writes, and P_2 and P_3 cause at least $b(x)$ redundant writes, for a total of at least $c(x)$ redundant writes. Next, we will prove a lower bound on $\max\{c(x) \mid x \in [n]\}$.

Observation: For all $x, y \in [n]$, if $a(x) = a(y)$ and $b(x) = b(y)$, then $x = y$.
We prove the contrapositive. Let x and y be two distinct number in $[n]$, and assume,

without loss of generality, that $x <_{\pi_3} y$. If $x < y$, then $a(x) < a(y)$. This is because we can append y to any increasing sub-sequence of π_3 ending at x and get a longer increasing sub-sequence of π_3. Otherwise, $x > y$, in which case $b(x) < b(y)$. This is because we can append y to any decreasing sub-sequence of π_3 ending at x and get a longer decreasing sub-sequence.

Let $C_i = \{x \in [n] \ : \ c(x) = i\}$, and let M be the largest integer such that $C_M \neq \emptyset$. Since $a(x), b(x) \geq 1$, it follows that $c(x) \geq 2$ for all $x \in [n]$. Thus $n = |C_2| + \cdots + |C_M|$. Next, we will bound the size of each C_i.

Let $x, y \in C_i$ be arbitrary. By definition, $c(x) = c(y) = i$. If $a(x) = a(y)$ then $b(x) = c(x) - a(x) = c(y) - a(y) = b(y)$, which implies that $x = y$. Since $b(x) \geq 1$, it follows that $a(x) = c(x) - b(x) \leq c(x) - 1 = i - 1$ for all $x \in C_i$. Therefore, $|C_i| \leq i - 1$, for all $i \geq 2$.

Since $n = |C_2| + \cdots + |C_M|$, we get a relation between M and k:

$$1 + \cdots + k \leq n = |C_2| + \cdots + |C_M| \leq 1 + \cdots + (M - 1)$$

This relation implies that:

$$M \geq \begin{cases} k + 1 & \text{, if } 1 + \cdots + k = n \\ k + 2 & \text{, otherwise} \end{cases}$$

By definition, there exists $x \in [n]$ such that $c(x) = M$. Therefore, there is an execution of A that results in at least M redundant writes. The inequality above gives us the desired lower bound on the work of A.

□

Lemmas 3.4 and 3.5 imply that $A_3(n)$ is optimal if one permutation is the reversal of the other, causing approximately $\sqrt{2n}$ redundant writes. The question is whether $A_3(n)$ is always optimal. To answer this question, consider the following algorithm:

$$\pi_1: \quad \langle 1, 4, 2, 3 \rangle$$
$$\pi_2: \quad \langle 2, 4, 3, 1 \rangle$$
$$\pi_3: \quad \langle 3, 4, 1, 2 \rangle$$

It is not hard to verify that the algorithm above results in at most 3 redundant writes, while $A_3(4)$ results in 4 redundant writes. This implies that $A_3(n)$ is not always optimal.

We will improve $A_3(n)$ to get a construction that results in $\Theta(\sqrt[3]{n})$ redundant writes. We will also prove that every 3-process non-adaptive algorithm performs $\Omega(\sqrt[3]{n})$ redundant writes.

3.2.3 Algorithms $A'_3(n)$ and $A''_3(n)$

To improve A_3, we partition $[n]$ into 2 sets of k blocks, $B_1, \ldots, B_k$ and $C_1, \ldots, C_k$. We restrict the block sizes to be $|B_i| = i + 1$ and $|C_i| = i$, for all i, and define the algorithm $A'_3(n)$ as follows:

$$\pi_1: \quad \langle B_1, C_1, \ldots, B_k, C_k \rangle$$
$$\pi_2: \quad \langle B_k^R, C_k^R, \ldots, B_1^R, C_1^R \rangle$$
$$\pi_3: \quad \langle C_k, B_k, \ldots, C_1, B_1 \rangle$$

Lemma 3.6. *For any positive integer k, if $n = k^2 + 2k$, then $W(A'_3(n)) = n + k + 2$.*

Proof. Similarly to the proof of Lemma 3.3, we let $\alpha \in PS_{A'_3(n)}$ be arbitrary, and consider the execution $E(A'_3(n), \alpha)$. We make the following observations:

1. At most two cells get written to by both P_1 and P_2: at most one from B_i and at most one from C_i, for some i.

2. Cells that are written to by both P_1 and P_3 belong to the same block.

3. For each i, at most one cell from $B_i \cup C_i$ gets written to by both P_2 and P_3.

Let t be the block-index where P_1 and P_3 cause redundant writes. Let t' be the block-index where P_1 and P_2 cause redundant writes.

P_3 cannot write to any block whose index is less than t, because such a block is examined by P_1 before P_3 starts examining it. Therefore, by observation 3, P_2 and P_3 can cause at most $k - t + 1$ redundant writes.

Next, we show that $t = t'$. P_1 cannot write to any block whose index is greater than t, because such a block is examined by P_3 before P_1 starts examining it. Since P_1 writes to a block whose index is t', we conclude that $t' \leq t$. P_1 cannot write to any block whose index is greater than t', because such a block is examined by P_2 before P_1 starts examining it. Since P_1 writes to a block whose index is t, we conclude that $t \leq t'$.

Case 1: P_1 and P_3 cause a redundant write in B_t. By observation 2, P_1 and P_3 cause at most $|B_t| = t + 1$ redundant writes. Also, P_1 cannot write to C_t, because this block is examined by P_3 before P_1 starts examining it. Since $t = t'$, P_1 and P_2 cannot cause a redundant write in $C_{t'}$. By observation 1, we get that P_1 and P_2 cause at most 1 redundant write.

Case 2: P_1 and P_3 cause a redundant write in C_t. By observation 2, P_1 and P_3 cause at most $|C_t| = t$ redundant writes. By observation 1, P_1 and P_2 cause at most 2 redundant writes.

In both cases the total number of redundant writes performed by $A'_3(n)$ is at most $k + 2$. Recall that, $\alpha(n)$ is written to by all three process, but only two of these writes are redundant.

□

By modifying $A_3(n)$ to $A'_3(n)$, we slightly more than doubled the value of n at the cost of a single redundant write. To improve the algorithm even further, instead of using two sets, we will use $l > 2$ sets of k blocks. This way, we will increase n by a factor of l at a cost of l additional redundant writes.

We let B_i^j denote the i'th block of the j'th block-set. i is the *block-index* and j is the

block-set. We choose a partition such that $|B_i^j| = i + (j-1)$, for all $1 \leq i \leq k$ and $1 \leq j \leq l$, and define $A_3''(n)$ as follows:

$$\begin{array}{lllll} \pi_1\colon & \langle B_1^l, \ldots, B_1^1 & , \ldots, & B_k^l, \ldots, B_k^1 \rangle \\ \pi_2\colon & \langle (B_k^l)^R, \ldots, (B_k^1)^R & , \ldots, & (B_1^l)^R, \ldots, (B_1^1)^R \rangle \\ \pi_3\colon & \langle B_k^1, \ldots, B_k^l & , \ldots, & B_1^1, \ldots, B_1^l \rangle \end{array}$$

The construction above looks slightly complicated because of the multiple indices, but it is a natural extension of $A_3'(n)$ (where we used letters B_i for B_i^2 and C_i for B_i^1 instead of introducing the additional index for the block-set). We can express n as function of k and l as follows:

$$\begin{aligned} n &= \sum_{i=1}^{k}\sum_{j=1}^{l} |B_i^j| \\ &= \sum_{i=1}^{k}\sum_{j=1}^{l} (i+j-1) \\ &= l\sum_{i=1}^{k} i + k\sum_{j=1}^{l} j - kl \\ &= \frac{l(k^2+k)}{2} + \frac{k(l^2+l)}{2} - kl \\ &= \frac{lk^2+kl^2}{2} \end{aligned}$$

Next, we adapt the proof of Lemma 3.6, to get an upper bound on the work of $A_3''(n)$.

Lemma 3.7. *For any positive integers k and l, if $n = \frac{lk^2+kl^2}{2}$, then $W(A_3''(n)) = n+k+l$.*

Proof. As in the previous proofs, we let $\alpha \in PS_{A_3''(n)}$ be an arbitrary permutation, and consider the execution $E(A_3''(n), \alpha)$. We modify the observations from the previous proof:

1. At most one cell from each block-set gets written to by both P_1 and P_2. All cells written to by both P_1 and P_2 belong to blocks with the same block-index.
2. Cells that are written to by both P_1 and P_3 belong to the same block.
3. For each i, at most one cell from blocks whose index is i gets written to by both P_2 and P_3.

Let B_t^s be the block where P_1 and P_3 cause a redundant write. Let t' be the block-index where P_1 and P_2 cause a redundant write.

As in the proof of Lemma 3.6, we get that P_2 and P_3 cause at most $k - t + 1$ redundant writes, and that $t = t'$.

By observation 3, P_1 and P_3 cause at most $|B_t^s| = t + (s - 1)$ redundant writes.

For each $j < s$, P_1 cannot write to B_t^j, because this block is examined by P_3 before P_1 starts examining it. Therefore, P_1 and P_2 cannot cause a redundant write to blocks whose block-set is smaller than s. By observation 1, we get that P_1 and P_2 cause at most $l - (s - 1)$ redundant writes.

Therefore, the total number of redundant writes performed by $A_3''(n)$ is at most $k+l$. Once again, one of the writes to $\alpha(n)$ does not count as redundant. Therefore, $W(A_3''(n)) \leq n + k + l$.

Following our argument, it is easy to see how an adversary can force $k + l$ redundant writes. For example, let $B = B_1^l$. An adversary can make both P_1 and P_3 write to all the cells of B, causing l redundant writes. The adversary can also make both P_2 and P_3 write the right-most cell of B_i^l, for all $1 \leq i \leq k$, causing additional k redundant writes. Therefore, we conclude that $W(A_3''(n)) \geq n + k + l$. □

Our next goal is to find a good value for l. We observe that, by choosing $l \in \Theta(k)$, we can increase n to be $\Theta(k^3)$. Lemma 3.7 tells us that, for such a choice of l, $A_3''(n)$ performs at most $k + l \in \Theta(k) = \Theta(\sqrt[3]{n})$ redundant writes. We can also see that, if we choose $l \in \omega(k)$, then the maximum number of redundant writes performed by A_3'' is $\omega(\sqrt[3]{n})$.

We write $l = ck$, where $c > 0$ is some constant, and we would like to find the best value for c. To do that, we write n as a function of k and c:

$$n = \frac{lk^2 + kl^2}{2} == \frac{c + c^2}{2} k^3$$

We rearrange the equality in order to express k as a function of n and c:

$$k = \sqrt[3]{\frac{2}{c^2+c}n} = \sqrt[3]{2n}c^{-1/3}(c+1)^{-1/3}$$

We can now express the maximum number of redundant writes performed by $A_3''(n)$ as function of n and c:

$$k + l = k + kc = k(1+c) = \sqrt[3]{2n}c^{-1/3}(c+1)^{2/3}$$

Finally, to find the best value for l we consider the function

$$f(c) = \sqrt[3]{2n}c^{-1/3}(c+1)^{2/3}.$$

We compute the derivative

$$f'(c) = \sqrt[3]{2n}(\frac{-1}{3}c^{-4/3}(c+1)^{2/3} + \frac{2}{3}c^{-1/3}(c+1)^{-1/3}).$$

We solve $f'(c) = 0$ and get that $f(c)$ has a local minimum at $c = 1$.

Corollary 3. *For any positive integer k, if $n = k^3$, then $W(A_3''(n)) = n + 2k$.*

Proof. This is a direct consequence of Lemma 3.7 with $l = k$. □

So far we have defined $A_3''(n)$ for values of n such that $n = k^3$, for some k. For other values of n we construct $A_3''(n)$ as follows:

- Let $n' > n$ be the the smallest integer such that $n' = k^3$, for some k.
- Define $A_3''(n)$ to be $A_3''(n')|[n]$, that is, $A_3''(n')$ restricted to the values $\{1, \ldots, n\}$.

By Lemma 3.7, $A_3''(n')$ performs at most $2k$ redundant writes. Therefore, by Lemma 2.7, $A_3''(n)$ performs at most $2k$ redundant writes. Since $n > (k-1)^3$, we get that $k \in O(\sqrt[3]{n})$. This tells us that $W(A_3''(n)) = n + O(\sqrt[3]{n})$.

Corollary 4. *For any positive integer n, $W(A_3''(n)) = n + O(\sqrt[3]{n})$.*

3.2.4 $n + \Omega(\sqrt[3]{n})$ work bound

We will show that every 3-process non-adaptive algorithm performs $\Omega(\sqrt[3]{n})$ redundant work, implying that $A_3''(n)$ is asymptotically optimal. We will use the Erdös-Szekeres Theorem [3]:

Theorem 3.1. *For any positive integers r and s, a sequence of length $n > (r-1)(s-1)$ contains either a monotonically increasing sequence of length r, or a monotonically decreasing sequence of length s.*

We are now ready to prove our lower bound.

Lemma 3.8. *For any 3-process non-adaptive algorithm A, with $|A| = n$:*

$$W(A) = n + \Omega(\sqrt[3]{n}).$$

Proof. Let $A = \langle \pi_1, \pi_2, \pi_3 \rangle$ be an arbitrary non-adaptive algorithm with $|A| = n$, and assume, without loss of generality, that $\pi_1 = \langle 1, \ldots, n \rangle$. Let d_2 and d_3 be the lengths of a longest monotonically decreasing sub-sequence of π_2 and π_3, respectively. Let $d = \max\{d_2, d_3\}$, and assume, without loss of generality, that $d = d_2$.

Case 1: $d \geq \lfloor 2^{-1/3} n^{2/3} \rfloor$. Let $S \subseteq [n]$ be a set of size d whose elements form a monotonically decreasing sub-sequence of π_2. The partial algorithm $A|S$ has $\pi_1 = \pi_2^R$, and Lemma 3.5 gives us a lower bound on the work of $A|S$. The maximum number of redundant writes performed by A is greater or equal to the maximum number of redundant writes performed by $A|S$. Therefore, we get that A performs $\Omega(\sqrt{d}) \subseteq \Omega(\sqrt[3]{n})$ redundant writes, and $W(A) = n + \Omega(\sqrt[3]{n})$.

Case 2: $d < \lfloor 2^{-1/3} n^{2/3} \rfloor$. We apply Theorem 3.1 with $s = d + 1$, and $r = \lfloor \sqrt[3]{2n} \rfloor$. Notice that:

$$(r-1)(s-1) < r \cdot d < \sqrt[3]{2n} \cdot 2^{-1/3} n^{2/3} = n.$$

By definition of d, π_2 does not contain a decreasing sequence of length $s = d + 1$. Therefore, π_2 must contain an increasing sequence of length r. An adversary can causes both P_1 and P_2 to write to each of these r cells. Since $r \in \Omega(\sqrt[3]{n})$, we get that $W(A) = n + \Omega(\sqrt[3]{n})$.

□

3.3 4-Process Algorithms

In this section, we extend the 3-process algorithms from the previous section, and construct 4-process algorithms. As in the previous section, we divide the set $[n]$ into blocks, and concatenate the blocks in various ways to define our algorithms. The techniques used in this section are very similar to the ones used in the previous section.

We start by defining algorithm $A_4(n)$, which is a natural extension of $A_3(n)$. We partition $[n]$ into k blocks, $B_1, \ldots, B_k$, such that $|B_i| = i$, for all i. We define $A_4(n)$ as follows:

$$\begin{aligned}
\pi_1&: \quad \langle B_1, \ldots, B_k \rangle \\
\pi_2&: \quad \langle B_k^R, \ldots, B_1^R \rangle \\
\pi_3&: \quad \langle B_k, \ldots, B_1 \rangle \\
\pi_4&: \quad \langle B_1^R, \ldots, B_k^R \rangle
\end{aligned}$$

Lemma 3.9. *For any positive integer k, if $n = 1 + \cdots + k$, then $W(A_4(n)) = n + O(\sqrt{n})$.*

Proof. π_1, π_2 and π_3 are the same as in $A_3(n)$, and Lemma 3.3 tells us that P_1,P_2 and P_3 cause at most $k+1$ redundant writes. In order to get an upper bound on the work of $A_4(n)$, we need to bound the number of redundant writes involving P_4. The following observations, similar to the ones made in the proof of Lemma 3.3, tell us that, in any execution of $A_4(n)$, P_4 can be involved in at most $O(\sqrt{n})$ additional redundant writes:

1. At most one cell gets written to by both P_4 and P_3.
2. Cells that are written to by both P_4 and P_2 belong to the same block.
3. At most one cell from each block gets written to by both P_4 and P_1.

□

We modify $A_3'(n)$ to get a 4-process algorithm in a similar way. We partition $[n]$ into two sets of blocks, $B_1, \ldots, B_k$ and $C_1, \ldots, C_k$, such that $|B_i| = |C_i| = i$, for all i.

Notice that the B-blocks were bigger in $A'_3(n)$, which improved that algorithm a little: It increased the value of n by k without causing additional redundant writes. This improvement does not work for 4 processes. We define $A'_4(n)$ as follows:

$$\begin{aligned}
\pi_1&: \quad \langle B_1, C_1, \ldots, B_k, C_k \rangle \\
\pi_2&: \quad \langle B_k^R, C_k^R, \ldots, B_1^R, C_1^R \rangle \\
\pi_3&: \quad \langle C_k, B_k, \ldots, C_1, B_1 \rangle \\
\pi_4&: \quad \langle C_1^R, B_1^R, \ldots, C_k^R, B_k^R \rangle
\end{aligned}$$

Lemma 3.10. *For any positive integer k, if $n = k^2 + k$, then $W(A'_4(n)) \leq n + 3k + 4$.*

Proof. Lemma 3.6 tells us that P_1, P_2 and P_3 cause at most $k + 2$ redundant writes (decrementing the size of the B-blocks does not affect this result). To get a bound the redundant writes involving P_4, we make the following observations:

1. At most two cells get written to by both P_4 and P_3 - At most one from B_i and at most one from C_i, for some i.
2. Cells that are written to by both P_4 and P_2 belong to the same block.
3. For each i, at most one cell from blocks whose index is i gets written to by both P_4 and P_1.

We consider an arbitrary execution, and let t be the block-index where P_4 and P_2 cause a redundant write.

P_4 cannot write to any block whose index is greater than t, because such a block is examined by P_2 before P_4 starts examining it. Therefore, by observation 3, P_4 and P_1 can cause at most t redundant writes.

By observation 1, P_3 and P_4 cause at most 2 redundant writes. By observation 2, P_2 and P_4 cause at most t redundant writes. By summing the counts, we get that P_4 is involved in at most $2t + 2 \leq 2k + 2$ redundant writes. Therefore, the total number of redundant writes performed by $A'_4(n)$ cannot exceed $3k + 4$.

□

Naturally, our next step is to modify $A_3''(n)$ to get a 4-process algorithm of size $\Theta(k^3)$ that performs $O(k)$ redundant writes, for some k. We partition $[n]$ into l sets of k blocks each. As in the previous section, B_i^j denotes the i'th block of the j'th set. This time, our partition satisfies $|B_i^j| = i$, for all $1 \leq i \leq k$, $1 \leq j \leq l$. We define $A_4''(n)$ as follows:

$$\begin{array}{lllll}
\pi_1\colon & \langle B_1^l, \ldots, B_1^1 & , \ldots, & B_k^l, \ldots, B_k^1 \rangle \\
\pi_2\colon & \langle (B_k^l)^R, \ldots, (B_k^1)^R & , \ldots, & (B_1^l)^R, \ldots, (B_1^1)^R \rangle \\
\pi_3\colon & \langle B_k^1, \ldots, B_k^l & , \ldots, & B_1^1, \ldots, B_1^l \rangle \\
\pi_4\colon & \langle (B_1^1)^R, \ldots, (B_1^l)^R & , \ldots, & (B_k^1)^R, \ldots, (B_k^l)^R \rangle
\end{array}$$

Lemma 3.11. *For any positive integers k and l, if $n = l(1 + \cdots + k)$, then $W(A_4''(n)) = n + O(\sqrt[3]{n})$.*

Proof. Lemma 3.7 tells us that P_1, P_2 and P_3 cause at most $k + l$ redundant writes. The following observations imply that P_4 is involved in at most $2k + l$ additional redundant writes:

1. At most one cell from each block set gets written to by both P_4 and P_3.
2. Cells that are written to by both P_4 and P_2 belong to the same block.
3. For each i, at most one cell from blocks whose index is i gets written to by both P_4 and P_1.

Observation 1 implies that P_4 and P_3 cause at most l redundant writes. Observation 2 implies that P_4 and P_2 cause at most k redundant writes. Observation 3 implies that P_4 and P_1 cause at most k redundant writes. Therefore, P_4 is involved in at most $2k + l$ redundant writes, and the total number of redundant writes performed by $A_4''(n)$ cannot exceed $3k + 2l$.

Any choice of $l \in \Theta(k)$ will give us the desired upper bound. Let $l = k$, and notice that $n = l(1 + \cdots + k) = \frac{k^3+k^2}{2} \in \Theta(k^3)$. Since $A_4''(n)$ cannot perform more than $3k + 2l = 5k$ redundant writes, we get that $W(A_4''(n)) \leq n + 5k = n + O(\sqrt[3]{n})$.

□

By choosing $l = k$, we have defined $A_4''(n)$ for values of n such that $n = \frac{k^3+k^2}{2}$, for some k. For other values of n, we start with $A_4''(n')$, where n' is the smallest integer greater than n, such that $n' = \frac{k^3+k^2}{2}$, for some k. We define $A_4''(n)$ to be $A_4''(n')|[n]$, that is, $A_4''(n')$ restricted to the values $\{1, \ldots, n\}$. Lemma 3.12 tells us that $A_4''(n')$ performs $O(\sqrt[3]{n'})$ redundant writes. By Lemma 2.7, the restricted algorithm, $A_4''(n)$, performs $O(\sqrt[3]{n'})$ redundant writes as well. Since $O(\sqrt[3]{n'}) = O(\sqrt[3]{n})$, we get that $W(A''4(n)) = n + O(\sqrt[3]{n})$.

Corollary 5. *For any positive integer n, $W(A_4''(n)) = n + O(\sqrt[3]{n})$.*

3.4 $n + \Omega(\sqrt[3]{n})$ Work Bound

By Lemma 3.8, in any execution of a 4-process algorithm, three of the four processes cause $\Omega(\sqrt[3]{n})$ redundant writes. In fact, we can apply the same bound to any p-process non-adaptive algorithm, where $p \geq 3$. We state this result formally in the next lemma.

Lemma 3.12. *Let A be an arbitrary p-process non-adaptive algorithm with $|A| = n$. If $p \geq 3$, then:*

$$W(A) = n + \Omega(\sqrt[3]{n})$$

Proof. The case $p = 3$ was proven in Lemma 3.8. If $p > 3$, an adversary can pause all processes except for P_1, P_2 and P_3. Lemma 3.8 implies that P_1, P_2 and P_3 cause $\Omega(\sqrt[3]{n})$ redundant writes. Therefore, by definition, $W(A) = n + \Omega(\sqrt[3]{n})$. □

Chapter 4

Conclusions

In this chapter, we list our contributions and relate them to previous work on the *Write-All* problem. We will also discuss a number of open questions directly related to our work.

4.1 Contributions

Our non-adaptive model relates to past work on the *Write-All* problem. As mentioned in section 1.2, Anderson and Woll [1] studied the same model, where processes use a fixed sequence of permutations to determine the order in which they examine the shared-memory. The main difference between our work and theirs is the complexity measure, which makes our algorithms more suitable for applications of *Write-All* where performing a task is an expensive operation. In our work, we consider only a small number of processes, similar to the approach presented in [2]. The main difference between our work and the work presented in that paper is that our algorithms are non-adaptive. We believe that our model helps understanding the combinatorial nature of the *Write-All* problem, and offers another view of the problem that was not fully explored.

Our constructions, described in detail in chapter 3, are non-trivial. We find it fairly surprising that non-adaptive algorithms on systems with 3 and 4 processes perform $O(\sqrt[3]{n})$ redundant work, and that this bound is asymptotically optimal.

The $n + \Omega(\sqrt[3]{n})$ work bound for non-adaptive algorithms with at least 3 processes gives us a good idea of the limitation of this model.

Our work relates to another result from [2]. The authors of that paper considered a model where processes take a snapshot and process all of the shared-memory at unit cost. We can view this model as "the most adaptive" model possible, where processes learn everything there is to know about the state of the shared memory, and adapt accordingly between write operations. Their model is a little too strong, and ours is a little too restrictive. A realistic model should take into account that processes can learn a limited amount of information between write operations, and allow processes to use this information to adapt. We believe that our constructions may be useful in developing a solution to *Write-All* on such models, where processes cannot learn enough information to "perfectly adapt" to the current state of the shared-memory.

4.2 Future Work

We constructed non-adaptive algorithms for systems with three and four processes. It is not obvious how to extend our approach to work for systems with 5 or more processes. Our $n+\Omega(\sqrt[3]{n})$ work bound applies to non-adaptive algorithms with an arbitrary number of processes (greater than 2). The question is, can we construct p-process non-adaptive algorithms that perform $n + O(\sqrt[3]{n})$ work, for $p > 4$? If not, can we show a stronger lower bound for systems with more than 4 processes?

Our lower and upper bounds for systems with 3 or 4 processes match asymptotically, but not exactly. For example, as mentioned in chapter 3, the lower bound is $n+\sqrt[3]{2n}+O(1)$, and the upper bound is $n+2\sqrt[3]{n}+O(1)$. It would be interesting to see either a 3-process algorithm that improves on $A_3''(n)$, or a proof of a stronger lower bound. We believe that such an improvement could be extended to improve our results for systems with 4 processes as well.

In general, improving current solutions to the *Write-All* on asynchronous models is still considered an open problem. We hope that researchers working on such improvements will benefit from our work, and use our results as a building block in their solutions.

Bibliography

[1] Richard J. Anderson and Heather Woll. Algorithms for the certified write-all problem. *SIAM J. Comput.*, 26(5):1277–1283, 1997.

[2] Jonathan F. Buss, Paris C. Kanellakis, Prabhakar L. Radge, and Alex A. Shvartsman. Parallel algorithms with processor failures and delays. Technical Report CS-91-54, 1991.

[3] P. Erdös and G. Szekeres. A combinatorial problem in geometry. *Compositio Math. 2*, pages 463–470, 1935.

[4] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, New York, NY, USA, 1978. ACM.

[5] Chryssis Georgiou, Alexander Russell, and Alex A. Shvartsman. The complexity of synchronous iterative do-all with crashes. *Distrib. Comput.*, 17(1):47–63, 2004.

[6] Chryssis Georgiou and Alex A. Shvartsman. *Do-All Computing in Distributed Systems*. Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 2007.

[7] Paris C. Kanellakis and Alex A. Shvartsman. Efficient parallel algorithms can be made robust. *Distrib. Comput.*, 5(4):201–217, 1992.

[8] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *STOC '91: Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 381–390, New York, NY, USA, 1991. ACM.

[9] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *STOC '90: Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, pages 138–148, New York, NY, USA, 1990. ACM.

[10] D.R. Kowalski and A.A. Shvartsman. Writing-all deterministically and optimally using a nontrivial number of asynchronous processors. *ACM Transactions on Algorithms (TALG)*, 4(3):33, 2008.

[11] Grzegorz Malewicz. A work-optimal deterministic algorithm for the asynchronous certified write-all problem. In *PODC '03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pages 255–264, New York, NY, USA, 2003. ACM.

[12] C. Martel, R. Subramonian, and A. Part. Asynchronous prams are (almost) as good as synchronous prams. In *SFCS '90: Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 590–599 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[13] Alex A. Shvartsman. Achieving optimal crcw pram fault-tolerance. Technical report, Providence, RI, USA, 1989.