DISTRIBUTED SCHEDULING
FOR DISCONNECTED COOPERATION

Grzegorz Malewicz, Ph.D.

University of Connecticut, 2003

Grzegorz Malewicz––University of Connecticut, 2003

The dissertation studies how distributed devices that are disconnected for long and unknown periods can efficiently perform a set of tasks. Given n distributed devices that must perform t independent tasks, known to each device, the goal is to schedule work of the devices locally, in the absence of communication, so that when communication is established between some devices at some later point of time, the devices that connect have performed few tasks redundantly beyond necessity. The dissertation gives a lower bound on redundant work, and randomized and deterministic schedules, that allow devices to avoid doing redundant work provably well. The lower bound shows how the wasted work increases as the devices progress in their work. When each disconnected device randomly selects its next task, from among the tasks remaining to be done, then the amount of work duplicated by any devices that reconnect is close to the lower bound in a precise sense. In order to derandomize the construction of schedules, techniques from design theory, linear algebra, and graph theory are used. The topics developed within the dissertation are related to the theory of latin squares and coding theory. For example the lower bound shown in the dissertation generalizes the Second Johnson Bound. The dissertation also studies scheduling problems for shared memory systems. It shows a method for creating near-optimal instances of an algorithm of Anderson and Woll. The dissertation also shows a work-optimal deterministic algorithm for the asynchronous Certified Write-All problem.

DISTRIBUTED SCHEDULING

FOR DISCONNECTED COOPERATION

Grzegorz Malewicz

M.S., Computer Science, University of Warsaw, 1998
B.A., Applied Mathematics, University of Warsaw, 1998
B.A., Computer Science, University of Warsaw, 1996

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2003

APPROVAL PAGE

Doctor of Philosophy Dissertation

DISTRIBUTED SCHEDULING

FOR DISCONNECTED COOPERATION

Presented by

Grzegorz Malewicz, M.S., B.A.

Major Advisor ____
Alex A. Shvartsman

Associate Advisor ______________________________________
Bogdan S. Chlebus

Associate Advisor ______________________________________
Arnold L. Rosenberg

Associate Advisor ______________________________________
Alexander C. Russell

University of Connecticut

2003

ACKNOWLEDGEMENTS

This work is affectionately dedicated to my parents, Aldona and Mieczysław, for their love and for passing onto me the family traditions for learning. Your values, dedication, and sacrifices have been guiding me and helping me for many years. I am deeply thankful to you my beloved parents.

I am thankful to my first tutors in science and computing. Halina Płoskonka was my primary school physics teacher, who was coaching me for a physics competition of Kielce district, Poland. It was because of her enthusiasm and engaging exposition of concepts that I discovered the pleasure of formulating questions and answering them. I received my first informal lessons in computer science from my elder friends: Artur Michalczyk and Mirosław Domagała. Their insightful comments provided bases for my first major program written in the assembly language of Z80.

I greatly benefited from studying computer science and applied mathematics at the University of Warsaw. I am especially indebted to my linear algebra teacher, Cezary Bowszyc, my algorithms teachers: Krzysztof Diks and Adam Malinowski, and my topology teacher, Sławomir Nowak, for being meticulous, expressing sense of humor during classes, and crafting the structure of exercises well. My first tutor, Andrzej Szałas, provided valuable guidance for me throughout my studies. My Masters Thesis advisor, Władyslaw Skarbek, led me through an exciting project, which was my first research project. The critical feedback of my advisor improved the quality of my research.

I am deeply thankful to my Ph.D. advisor, Alex Shvartsman, who guided my doctoral work in computer science at the University of Connecticut. He gave me experiences in academic and professional life, enabled my diverse development, and provided critical comments. Interaction with him substantially improved the quality of my work. He has made a profound impact on my career, for which I would like to thank him.

I am also thankful to other members of my doctoral committee. The opportunity to work with Alexander Russell enriched my portfolio of research techniques, and his enthusiasm about research and teaching served as a model for me. Bogdan Chlebus from the University of Warsaw and Arnold Rosenberg from the University of Massachusetts, Amherst provided comments on summarizing the work of others and brought to my attention the significance of motivating theoretical work through applications.

I would like to thank Michael Merritt for giving me a wonderful opportunity to spend a summer doing research in his group at the AT&T Shannon Lab, and for his generous support and encouragement. During this visit, my mentor Amotz Bar-Noy taught me a methodology that made my research more effective. I had fruitful discussions with David Johnson, Jeff Lagarias, and S. Muthukrishnan that added to my research sophistication.

I am thankful to my host, Charles Leiserson, at the Massachusetts Institute of Technology for inviting me to join his research group for a year, and showing me how to write better proofs. I am also thankful to Piotr Indyk, Bradley Kuszmaul, and Larry Rudolph for reading my writings and commenting on them.

Several people helped in my study of Design Theory. I am thankful to Shmuel Zaks who pointed out a basic relationship between disconnected cooperation and design theory, and to Charles J. Colburn, Donald Preece, Neil J.A. Sloan, and Madhu Sudan who gave me directions for further reading.

Many other people have helped me in my research endeavors since the time I entered the Ph.D. program. Reda Ammar, Tushar Chandra, Burkhard Englert, Dariusz Kowalski, Shay Kutten, Ibrahim Matta, Marios Mavronicolas, Mikołaj Rotkiewicz, Eugene Spiegel, and T.C. Ting discussed projects with me, provided insightful comments and ideas, and encouraged me to do research.

I learned a lot about developing software during my two visits to product groups at Microsoft Corporation. I am thankful to Levon Esibov, Eron Kelly, Stuart Kwan, Derek LaSalle, and Nagender Vedula for teaching me about, and enriching my experience in, software development, management, marketing, and delivering presentations.

There are other things beyond books and office. My karate sensei, Małgorzata Kosiba, taught me about maintaining a balance between physical and intellectual development, for which I would like to thank her. I am also thankful to companions of my travels: Wiesiek Bartkowski, Przemek Ferdek, Karol Giemza, Krzysiek Jasek, Paweł Kot, Jarek Smulski, and Piotrek Zabłudowski, and many friends from the Student Mountaineering Club at the University of Warsaw for assisting in the organization of several of my expeditions to explore the beauty of the world.

CREDITS

This thesis is based on the following publications.

The paper [100] of Malewicz, Russell, and Shvartsman, that appeared in the Proceedings of the 14th International Symposium on Distributed Computing: defined the h-waste measure (appearing in Section 3); showed a lower bound on 2-waste (appearing in Section 4); showed a construction of asymptotically scalable randomized schedules (appearing in Section 5.1); and showed a deterministic construction of schedules based on designs.

The paper [101] of Malewicz, Russell, and Shvartsman, that appeared in the Proceedings of the 8th International Colloquium on Structural Information and Communication Complexity: refined and simplified the design-theoretic deterministic construction that uses a vector space of dimension 3 over a finite field, and showed asymptotic scalability of the constriction when $n \leq t$ (appearing in Section 6.1); showed a graph-theoretic method of constructing schedule that improves initial growth of waste (appearing in Section 6.4.1); showed a recursive method for constructing scheduled that improves initial growth of waste (appearing in Section 6.4.2); and studied h-waste of randomized schedules (appearing in Section 5.2).

The paper [104] of Malewicz, Russell, and Shvartsman, that appeared in the Proceedings of the IEEE International Symposium on Network Computing and Applications: showed a deterministic asymptotically scalable construction of schedules when $n = t^{2-\epsilon}$ based on a vector space of dimension higher than 3 over a finite field (appearing in Section 6.3); and showed a scheduling method based on polynomials (appearing in Section 6.2).

The paper [103] of Malewicz, that appears in the Proceedings of the 11th Annual European Symposium on Algorithms, shows a method for creating near-optimal instances of a Certified Write-All algorithm of Anderson and Woll (appearing in Section 7.3).

The paper [102] of Malewicz, that appears in the Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing, shows a deterministic work-optimal algorithm for the asynchronous Certified Write All problem (appearing in Section 7.4).

The paper [27] of Chlebus, Dobrev, Kowalski, Malewicz, Shvartsman, and Vrto, that appeared in the Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures, presents a conjecture and supporting analysis indicating that it may be possible to efficiently construct a list of q permutations on $[q]$ such that the list has contention $O(q \log^2 q)$ (appearing in Section 7.5).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This dissertation studies a cooperation problem where a system of distributed asynchronous devices, that can be disconnected for long and unknown periods, must efficiently perform a set of tasks.

1.1 Motivation

Distributed systems have many compelling advantages over centralized systems, such as ability to mask failures, potential for high availability, and improved performance. However, building correct and efficient distributed systems is often challenging because of the possible unpredictable changes in the system that can occur during the lifetime of the system. The goal of understanding the advantages and fundamental challenges of distributed systems have been the subject of research over the last decades [45, 46, 96, 54, 5, 135].

One of the instances of distributed systems are grid computing infrastructures [22, 57, 68]. A grid infrastructure leverages the computing power of massive numbers of independent computing devices, to tackle large computational problems. Grids are used to forecast weather and design drugs. Key to the efficient operation of many grid infrastructures is the ability to perform a collection of tasks in a distributed setting. It is often difficult to perform the tasks quickly, because of changes in communication and computing media, that may occur during the operation of the grid.

Failures are common in current communication networks. Communication topology may change over time (e.g., due to link failures), messages may be delayed (e.g., due to congestion), communication may be costly (e.g., due to competition for resources), or communication may be intentionally disabled (e.g., due to the need to maintain radio silence or save energy). Consequently, some devices may not be able to communicate with others for prolonged periods, and thus may need to make decisions using limited and possibly obsolete information about the state of other devices. Dealing with obsolete information makes it challenging to efficiently utilize computing resources [9, 12, 20, 43, 52, 64, 80, 110, 111, 120, 121]. A consequence of local decision making is that devices may unknowingly choose to perform the same tasks, thus increasing the number of redundantly performed tasks and postponing the moment by which the devices compute all tasks.

Variability in the computing medium is also common. Devices may work at varying paces (e.g., due to time-sharing) or can quit computation (e.g., due to a failure). Asynchrony can increase the execution time of a computation. Suppose that a correctly working device has performed a task, and slows down before communicating the progress to other devices. Other devices will not know if the task has been performed by the device, and may (unknowingly) redo the task, again increasing the number of redundantly performed tasks and postponing the moment by which the devices compute all tasks.

Performing a computation quickly on asynchronous devices under variable communication is challenging. As long as devices are connected, they can share their knowledge to estimate which tasks remain to be done, and agree what to work on next, so as not to duplicate the execution of any of the remaining tasks. However, when devices become disconnected, they no longer can exchange information, and they must make local decisions about which tasks to work on next, if at all. Each device should indeed continue computing tasks during a period of disconnection (as opposed to idle), so as to reduce the set of tasks remaining to be done to the extent possible given the spare CPU cycles available to the device, and hence potentially reduce the set of tasks remaining to be done, when results of tasks are exchanged during a future connection. At the same time, one would like to ensure that disconnected devices work on the tasks that they somehow know that other devices have not been working on. Doing so allows to maximize the computational progress when devices connect. The challenge here is to allow devices to always work, but at the same time keep the number of redundantly performed tasks low, in order to possibly accelerate the moment by which the devices compute all tasks.

1.2 Problem statement

This thesis studies the problem of scheduling work of disconnected asynchronous devices with the goal of reducing the number of redundantly performed tasks. There are t independent and idempotent tasks, and n asynchronous processors. A task is *idempotent* if the execution of the task yields the same result when it is performed more than once. Tasks are independent if the result of execution of any task does not depend on the order in which other tasks are executed. The processors have unique identifiers from the set $[n] = \{1, \ldots, n\}$, and the tasks have unique identifiers from the set $[t] = \{1, \ldots, t\}$. Initially each processor knows all the tasks, the value of t, and the value of n. At the beginning of computation communication is not available, but at some *a priori* unknown point of time some group of *a priori* unknown processors rendezvous. Processors compute at all times during the period of disconnection, and our goal is to control the number of redundantly performed tasks in the group.

We consider the following framework for solving the above problem. A *schedule* σ is a permutation on $[t]$. Schedules $\sigma_1, \ldots, \sigma_n$ immediately give rise to a strategy for n isolated processors who must complete t tasks until communication between some pair (or group) of processors is established: the processor i

proceeds to complete the tasks in the order prescribed by σ_i. Suppose now that some h of these processors, say $q_1, \ldots, q_h$, should rendezvous at a time when the i-th processor in this group, q_i, has completed a_i tasks. Ideally, the processors would have completed disjoint sets of tasks, so that the total number of tasks completed is $\sum_i a_i$. As this too much to hope for in general, it is natural to attempt to bound the gap between $\sum_i a_i$ and the actual number of distinct tasks completed. This gap we call *waste.* (For fixed schedules $\sigma_1, \ldots, \sigma_n$, waste is a function of the $a_1, \ldots, a_h$.)

Let us consider an example of computation involving a single rendezvous of some two processors. In the worst case they rendezvous after performing all tasks individually. In this case no savings in the combined number of tasks performed by the processors is realized. Suppose that they rendezvous after having performed $t/2$ tasks each. In the best case, the two processors performed mutually-exclusive subsets of tasks and they learn the complete set of results as a consequence of the rendezvous (during which state exchange occurs). In particular if these two processors know that they will be able to rendezvous in the future, they could schedule their work as follows: one processor performs the tasks in the order $1, 2, \ldots, t$, the other in the order $t, t-1, \ldots, 1$. No matter when they happen to rendezvous and how much progress they will have made by then, the number of wasted tasks they both perform is minimized. In our setting processors do not know *a priori* what group will be able to rendezvous. Thus our goal is to produce task execution schedules for all processors, such that upon the first rendezvous of any group of processors, the number of tasks performed redundantly is minimized.

1.3 Summary of results

The dissertation studies how n distributed devices that are disconnected for long and unknown periods can efficiently perform a set of t independent tasks that are known to each device. The thesis investigates local scheduling decisions that require no communication between devices. We show fundamental limitations of local scheduling and develop randomized and deterministic scheduling techniques that offer nearly tight bounds on waste.

We show a lower bound on waste, that says that worst-case number of redundantly performed tasks must grow as processors progress in their work. Specifically, for any n schedules for t tasks, at least $a^2/(t-b+a) \cdot \left(1 - \frac{t-b}{a(n-1)}\right)$ redundant tasks must be performed by at least two processors when they establish communication after one of them has performed a tasks and another b tasks, $a \leq b$. For example, when $n = t$, then there are two processors such that when they rendezvous after having performed a tasks each, then at least $\frac{a^2-a}{t-1}$ tasks have been performed redundantly by the two processors. Thus in this situation, waste is bounded from below by a quadratic function of a. Naturally, the lower bound also applies to any group of $h \geq 2$ processors that rendezvous.

It is not surprising that when each disconnected device randomly selects its next task, from among the tasks remaining to be done, then this device will avoid duplicating the work of other devices quite well. Specifically, we study waste of a system of n schedules obtained when each schedule is selected uniformly (and independently) at random among all permutations of $[t]$. We show that with probability at least $1 - \frac{1}{nt}$, any two processors that rendezvous, after having performed sufficient number of a and b tasks respectively (i.e., $7\sqrt{t}\ln(2nt) \leq a, b \leq t$), have performed at most $\frac{ab}{t} + \Delta(a,b)$ redundant tasks, where $\Delta(a,b)$ is a lower order summand equal to $11\sqrt{\frac{ab}{t}\ln(2nt)}$. This compares favorably with the lower bound when $a = b$. We also bound waste of random schedules when a group of two or *more* processors rendezvous. We show that with probability at least $1 - \frac{1}{n}$, waste incurred by any h processors that establish communication after having performed a tasks each, is within a lower order summand of $(2h+1)\sqrt{a \ln n}$ from the expected waste of $\sum_{s=2}^{h}(-1)^s \binom{h}{s} \frac{a^s}{t^{s-1}}$.

Main contributions of this thesis include methods for derandomizing the construction of schedules using intersection properties of certain subspaces of a multidimensional vector space over a finite field. We give constructions for different relations between n and t. When $t \geq n$, for a prime power q, $n = q^2 + q + 1$, such that n divides t, we show how to construct n schedules so that when any two processors rendezvous after having performed sufficient number of a tasks each (i.e., when $a \geq \frac{t}{n}\left(1 + q^{7/4}\right)$), then waste is at most $1 + 22 \cdot n^{-1/4}$ times the corresponding lower bound. Each schedule can be produced in time $O(t)$. This construction has the property that waste can grow linearly when progress a is small compared to t, which is a disadvantage when t is much larger than n.

We alleviate this issue with a recursive construction that offers tighter bounds on waste for small progress a. Next, we consider the case when there are more processors than tasks, $n = t^{1-x}$, for some $0 < x < 1$. This models the end-game where there are relatively few tasks remaining to be performed, compared to the number of processors available to perform the computation. We construct $n = t \cdot \frac{q^{m-1}-1}{q^2-1}$ schedules for $t = \frac{q^m-1}{q-1}$ tasks. (Note that n is about $t^{2-\epsilon(m)}$, where the $\epsilon(m)$ tends to 0 when m tends to infinity.) When progress a is large enough (i.e., when $a > \frac{q^{m-2}-1}{q-1}$), waste is at most $1 + 64 \cdot t^{-1/(3m-3)}$ times the lower bound. Each schedule can be produced in time $O(mt)$. We also study the super-saturated case when there are substantially more processors than tasks, $n = t^u$, for a constant $u \geq 2$. Specifically, when $t = q^2$ and $n = q^d$, for $d \geq 2$, we present a construction of n schedules based on polynomials such that waste is bounded by $(d-1)\sqrt{t} + 4\frac{ab}{t}$, when progress of any two processors a and b is at most $t/4$. Finally, we give a construction of schedules with a bound on waste for groups of size $h \geq 2$ when $n = t^{1-x}$. Specifically, when $n = \frac{q^{m-1}-1}{q-1}$ and $t = \frac{q^m-1}{q-1}$, we show how to construct a system of schedules so that any group of h processors that rendezvous after having performed a tasks each, wastes at most $\binom{h}{2}\frac{a^2}{t}\left(1 + c_1 \cdot t^{-1/(m-1)} + c_2 \cdot h^{-1}\right)$ tasks, where c_1 and c_2 are absolute constants.

We show how our distributed computing problem is linked to several mathematical theories. Our distributed computing problem is related to the sphere packing problem. Our solutions give rise to a construction of latin squares with specific uniformity properties. Our lower bound generalizes the Second Johnson Bound from coding theory.

In addition to results on scheduling work of disconnected devices, the thesis also shows results on scheduling work of parallel processors in a shared memory system. It demonstrates how to create near-optimal instances of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll [4]. In this algorithm n processors update n memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations, where q can be chosen from a wide range of values. The thesis shows that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work. Finally, the thesis presents a deterministic asynchronous algorithm for the Certified Write-All problem. The algorithm has optimal *work* complexity $O(n)$ for a nontrivial number of processors $p \leq (n/\log n)^{1/4}$. In contrast, all deterministic algorithms known to date require superlinear in n work when $p = n^{1/r}$, for any fixed $r \geq 1$. This work-optimal algorithm generalizes the collision principle of the algorithm T [24].

1.4 Overview of related work

The efficiency of algorithms for computing tasks depends on how well the loads are balanced among the participating processors, and on the ability of the processors to disseminate information about the progress of the computation.

Load balancing to perform tasks with processor crashes was studied by Dwork, Halpern and Waarts [51]. References to a large body of work on this topic can be found in the monograph by Kanellakis and Shvartsman [85].

Load balancing to perform tasks with communication failures was studied by Dolev, Segala and Shvartsman [49], who formulate a distributed computing problem of performing a collection of n tasks by n processors in partitionable networks. They give a competitive analysis showing that the termination time of any on-line task scheduling algorithm is greater than the termination time of an off-line task scheduling algorithm by at least a factor linear in n. They also present a work-efficient load-balancing algorithm for networks subject to arbitrary fragmentations. Finally they introduce the definition of pairwise waste and give a deterministic scheduling strategy for up to $n^{1/3}$ tasks that minimizes it. This work was followed by Georgiou and Shvartsman [66] who improved the fragmentation-tolerant algorithm of [49], and who analyze the work and message efficiency of their algorithm using a DAG that models network fragmentation and merges. The problem of waste minimization introduced in [49] was later studied in depth by Malewicz, Russell, and Shvartsman [100, 101]. The authors generalize the notion of pairwise waste of [49] to h-wise waste, for any $h \geq 2$. This work shows that random schedules perform well in terms of expected waste and shows how to effectively generate deterministic schedules with low worst-case waste. (Randomized scheduling techniques were previously considered in a variety of similar task-performing settings, e.g., by Chlebus and Kowalski

[29], Anderson and Woll [4], and by Martel and Subramonian [108].) Georgiou, Russell and Shvartsman [65] observed that it is impossible to achieve low work for the basic task-performing problem of [49] in partitionable networks subject to fragmentations and merges. They pursue competitive analysis of work and use the algorithmic approach of [66] together with random task scheduling. The authors show that their randomized algorithm has optimal competitive ratio of $1 + \mathbf{cw}/e$, where $\mathbf{cw}$ is the *computation width*, defined as a number associated with the DAG that describes the network reconfigurations.

There are several results developed in the context of design theory that provide partial solutions to our scheduling problem. The result on pairwise waste of Dolev, Segala and Shvartsman [49] can be improved using techniques from design theory (see e.g., Hughes and Piper [77] for an introduction to design theory). Specifically the n blocks of a u-$(t, k, 1)$ packing design (see Mills and Mullin [109] and Stinson [133] for an introduction to packing designs) overlap by at most $u - 1$, and a block can be used to schedule the first k out of t tasks of a processor. For example, there is a known construction that allows to schedule about $k = t^{1/2}$ tasks with overlap of 1. The problem of controlling the number of redundantly performed tasks for arbitrary progress of processors could be solved using multiply-nested packing designs. Unlike existing approaches presented by Morgan, Preece, and Rees [112] and Preece [123], for our purpose the packings must have the same number of blocks, but the parameter u should be different across the nested packings. It appears that multiply nested packings that yield almost tight bounds on waste are not known. In addition, the Second Johnson bound [81] yields a lower bound on waste for processors that connect after performing the same number of tasks. Our results extend this lower bound to the case when progress of processors may be different. The concept of nesting has been studied in coding theory in a different context by Dodis and Halevi [47]

Another line of research shows that quality of load balancing correlates with the extent of communication available to the devices. Papadimitriou and Yannakakis [120] study how limited patterns of communication affect load-balancing. They consider a problem where there are 3 agents, each of which has a job of a size drawn uniformly at random from $[0, 1]$, and this distribution of job sizes is known to every agent. Any agent A can learn the sizes of jobs of some other agents as given by a directed graph of three nodes. Based on this information each agent has to decide to which of the two servers its job will be sent for processing. Each server has capacity 1, and it may happen that when two or more agents decide to send their jobs to the same server, the server will be overloaded. The goal is to devise cooperative strategies for agents that will minimize the chances of overloading any server. The authors present several strategies for agents for this purpose. They show that adding an edge to a graph can improve load balancing. This problem is similar to our scheduling problem. Sending a job to server number $x \in \{0, 1\}$ resembles doing task number x in our problem. The goal to avoid overloading servers resembles avoiding overlaps between tasks. The problem differs from our problem because we are interested in sequencing the execution of tasks, where the number of tasks can be arbitrary $t \geq 1$. The problem of Papadimitriou and Yannakakis has been studied by: Deng and Papadimitriou [43], Irani and Rabani [80], Brightwell, Ott and Winkler [20], and Georgiades, Mavronicolas and Spirakis [64]. Similar results demonstrating that the availability of communication can affect the effectiveness of load-balancing were given by Papadimitriou and Yannakakis [121], Awerbuch and Azar [9], Bar-tal, Byers and Raz [12], Eager, Lazowska and Zahorjan [52], Mitzenmacher [111], and Mirchandaney, Towsley and Stankovic [110].

1.5 Thesis organization

The remainder of this thesis is organized as follows. In Section 2, we give an account of research results related to the problem. In Section 3, we detail our cooperation problem, mathematical tools used in the solutions, and links between our problem and other mathematical problems. In Section 4, we show a lower bound on the number of redundantly performed tasks for processors that connect. In Section 5 and Section 6, we give randomized and deterministic constructions that control the number of redundantly perfomed tasks quite well. Finally, in Section 7, we present results on scheduling for shared memory systems.

Chapter 2

Related work

In this chapter we present several problems that are related to the scheduling problem studied in this dissertation. We begin with the review of distributed platforms for performing large numbers of tasks (Section 2.1). Next we review results of studies of an abstract version of the load balancing problem with fault-prone processors (Section 2.2). We also report on load balancing techniques when processors are fault-free, but the communication medium may be faulty (Section 2.3). Next we report on distributed cooperation techniques when only partial information about the problem is available to the distributed devices (Section 2.3). Then we report on work about producing schedules to that avoid redundancy (Section 2.5). Next we present mathematical tools used to derive and analyze the properties of the schedules developed in the dissertation (Section 2.6). Finally we review some facts from design theory and several applications of design theory in computer science (Section 2.7).

2.1 Grid computing

A grid [57, 68] is a software platform that allows users to share storage, communication, and computation resources that are heterogeneous and distributed, but at the same time allow each user to control which resources, and to what extent, are shared with other users. A grid allows users to request a set of resources that satisfy prescribed requirements (such as the number of computers, time and duration of their required availability, their operating system, quality of network that connects the requested computers), and perform some computation using the resources. There are many implementations of grids (see [22, 69] for examples). A grid called High Throughput Computing Grid [1, 23, 41, 59, 63, 79, 94, 97, 98, 119, 122, 127, 126, 128, 138, 139, 94] executes large numbers of independent, idempotent, and similar tasks using distributed, heterogeneous computers (that are often underutilized [115]). We review some of the instances of these grids.

The authors of the SETI@home project [94] aim at finding evidence of extraterrestrial life. They develop a computing platform where distributed computers cooperate on a computation to detect patterns in radio signals received from the cosmos. It is assumed that an extraterrestrial civilization would broadcast a radio signal at the frequency of the hydrogen line. Due to the Doppler effect, the unknown velocity of the emitter at the time of emission and unknown relative velocity of the receiver at the time of receiving, one needs to probe frequencies in the neighborhood of the hydrogen line in order not to overlook a potential signal. A telescope in Puerto Rico repeatedly scans about 25% of the sky and records a range of radio frequencies centered at the hydrogen line. This data is delivered to a server at the University of California, Berkeley. The data is decomposed into tasks each described using about 300KB. Each task needs to be processed using the FFT to detect spikes of frequencies significantly above the noise level. Each task can be performed independently. The processing of one task takes about 10 hours on a typical personal computer. During the time when tasks are processed, new signals are received, and so new tasks continue being created at the server. The solution employs a client-server architecture. The server can: (1) register any volunteer and send there a client program that can process any task, (2) send a task to a client program that contacted the server, and (3) receive the result of processing of a task from a client program. In this solution any client computer processes at most one task at any time. The server stores a database of about 2 million volunteers, half a million of which actively participate in the processing, and keeps track of which volunteer processed which task. A task is sometimes processed more than once (by different volunteers) to verify the results using a (hopefully) independent volunteer.

In the XPulsar@home project [139] the authors study a mathematical model of a pulsar and their goal is to calculate the distribution of photons emitted from the pulsar. For this purpose they implement a Monte Carlo method. In this method, initial locations and velocities of photons are chosen according to some random process and the trajectories of these photons are then traced to arrive at the distribution of photons. Each tracing can be done independently from one another. The authors implement this method by creating a server program for collecting results, and a client program for tracing trajectories. The client program is uploaded to hundreds of computers. Then any client program repeatedly chooses the initial location and velocity, traces the photon, and reports the results of tracing to a central server.

The goal of the Intel CURE project [79] is to evaluate the curing potential of hundreds of millions of molecules, in order to design new drugs. Evaluations can be done independently from one another. The authors propose a distributed architecture similar to that of SETI@home, where volunteers download a client program that repeatedly requests a task from a server, performs evaluation for about 1 day, and returns results to the server. A similar method is used by the FightAIDS@Home project [119]. Here the problem requires testing if the trial drug molecules can be *docked* at the target protein.

In the RSA Factoring by Web project [128], distributed machines use a factoring method called Number Field Sieve [23]. For the instance of the factoring problem considered by the authors, one of the stages of the method called sieving would normally take 8.9 years on a single personal computer. However, this stage can be decomposed into independent and idempotent tasks, and executed in parallel on distributed machines in 1 CPU month.

2.2 Load balancing with processor failures

Several studies consider an abstract load-balancing problem in a distributed environment, where computing medium is prone to failures.

Dwork, Halpern and Waarts [51] were the first to formulate and study the abstract problem of performing t independent, idempotent, and similar tasks in a distributed environment consisting of n failure-prone processors. We call this problem DO-ALL. In this problem each processor has access to a global clock. Processors communicate by sending messages. Any processor can fail (i.e., stop computing) at any time, as long as there is at least one operational processor at any time. The goal of these processors is to perform the t tasks so that the total number of tasks performed (counting multiplicities) is small. These tasks are initially known to every processor. Following this work, there has been substantial research on how to efficiently perform the tasks subject to processors failures. These results were given by De Prisco, Mayer and Yung [44], Galil, Mayer and Yung [60], Chlebus and Kowalski [29], and Chlebus, Kowalski and Lingas [30]. There were also extensions of this model to scenarios where processors can restart. A processor that restarts begins computation in a predefined initial state without the knowledge of its prior state (i.e., it does not have access to stable storage). These results were give by Chlebus, De Prisco and Shvartsman [26].

The DO-ALL problem was originally considered in the shared memory model, where it is called WRITE-ALL. In this model processors can exchange information (or communicate) by writing to and reading from shared memory. The WRITE-ALL problem was first introduced and studied by Kanellakis and Shvartsman [84, 85]. Other work includes Kedem, Palem, Raghunathan and Spirakis [90], Buss, Kanellakis, Ragde and Shvartsman [24], Anderson and Woll [4], Groote, Hesselink, Mauw and Vermeulen [70], and Chlebus, Dobrev, Kowalski, Malewicz, Shvartsman and Vrto [27]. These studies on DO-ALL and WRITE-ALL present various load-balancing techniques for sequencing the work for the computing devices that are able to communicate by some means. A common challenge is to design effective failover techniques. When a processor performs some tasks and fails before communicating to other processors that it has performed these tasks, the other processors need to perform the tasks, which increases the total number of tasks that need to be performed (counting multiplicities), and degrades performance.

2.3 Load balancing with communication failures

In some scenarios, processors that do not participate in the exchange of information can perform computation. This happens, for example, in the setting of partitionable networks [49]. This setting is also called a message passing model with communication failures. In this setting the communication topology of the network may change during the computation, causing regroupings of processors. Communication links that fail can "fragment" (or partition) the system into connected components. When links recover, processors that were disconnected may "merge" to form larger components. Such failures occur, for example, in the context of mobile computing. Indeed, an intrinsic feature of mobile computing [25, 78, 83, 92, 130, 136] is that communication topology changes over time, and some devices may not be able to communicate with others for prolonged periods of time. In such a setting, a processor that does not participate in the exchange of information may still do useful work. If a processor becomes isolated before all tasks have been performed, it will never know that tasks have been performed nor know their results, unless it executes all the remaining tasks on its own. This setting motivates the consideration of a variation of the DO-ALL problem called OMNI-DO.

The OMNI-DO problem was defined by Dolev, Segala and Shvartsman [49] as follows. There are n processors that do not fail. There are t independent and idempotent tasks initially known to each processor. The goal is that each processor learns the results of all tasks. A processor learns the result of a task either by executing the tasks or by receiving the result from some other processor. The requirement that each processor knows the results of all tasks at the end of computation, is an extension to the DO-ALL problem. The efficiency of OMNI-DO algorithms is measured by summing up how many times each task has been performed by processors (a measure introduced in [51]). Two solutions to the OMNI-DO problem have been studied. Each of them uses a message passing model of computation augmented with a group communication service

[11, 17, 18, 31, 37, 48, 53, 74, 113, 125] that notifies processors about the current membership of components and provides a multicast service for delivering messages to all processors in the component. The OMNI-DO algorithms studied by Dolev, Segala and Shvartsman [49] are developed for an asynchronous system where processors within a group work in rounds controlled by a group communication service. Processors start computing as a single component. Later on, component(s) may fragment into smaller components. Georgiou and Shvartsman [66] extend these results to the setting where components may undergo fragmentations and merges during the computation. They offer work-efficient and message-efficient algorithms. A common feature of these solutions is that processors do load-balancing inside connected components. Sequencing the tasks for execution inside such a component does not take into account possible future merges with other components.

2.4 Load balancing using partial information about distributed systems

Distributed devices may not know the current state of other devices in a distributed system due to delays in transmitting information. Therefore, it is interesting to study how to make load balancing decisions using only partial information about the instance of a load balancing problem but without compromising quality by much. Specifically, one could look for load balancing algorithms that are competitive compared to the best load balancing algorithm that has complete information about the instance of the problem. In this section we review some work on this topic that indicates that the more the devices know, the better the decisions can be.

The work of Papadimitriou and Yannakakis [120] (mentioned in Section 1.4), who study how communication topology influences the effectiveness of load assignment, was continued by other researchers.

Deng and Papadimitriou [43] study competitive strategies for the problem of [120]. Each agent knows the communication graph and they are to cooperatively assign jobs to two servers so as to minimize makespan. The authors show optimal competitive strategies for each graph (the competitive ratio ranges from 1 to 2). Then they show that for any r, it is NP-hard to find an assignment of jobs of n agents to m servers, that achieves competitive ratio r. Finally, they consider a model with no communication at all, in which each of the n agents has some number of jobs of some sizes, and must decide to which of the n servers to send the jobs for processing, without coordinating with other agents. The authors show a $2\sqrt{n}$-competitive deterministic algorithm, and a lower bound of $\sqrt{n}$ on the ratio for deterministic algorithms. They present a randomized algorithm that has ratio $O(\log n/\log\log n)$.

Irani and Rabani [80] generalize the work of [43] to an arbitrary communication graph with an arbitrary number of nodes (agents). Each agent has some jobs of some known size, and the goal is to distribute the jobs across other agents (maybe even agents from which information was not available, e.g., because there is no edge in the graph) so as to balance the load. The authors show that for a given graph G, the competitive ratio for deterministic algorithms is at least the square root of the size of a maximum independent set, and that there is a deterministic algorithm for distributing load that is competitive by at most twice the square root of the size of minimum clique cover (the algorithm uses answers to some NP-hard problems). The authors show that among r-regular graphs the best graph (up to constant factor of 2) is a graph with cliques of size $r+1$ with no communication between cliques, which yields $2\sqrt{n/(r+1)}$ bound on competitive ratio. The authors show that randomization can beat the lower bound, and extend the result of [43], to show that for any graph, a randomized algorithm has $O(\log f/\log\log f)$ ratio, where f is the size of minimum clique cover. When graph is r-regular, then the competitive ratio is $O(\log\frac{n}{r}/\log\log\frac{n}{r})$. When r not too large compared to n, the authors show a lower bound of $\Omega(\log\frac{n}{r}/\log\log\frac{n}{r})$, which demonstrates than the competitive ratio is optimal up to a constant for these values of r. The authors also study a problem of routing a permutation on some fixed network where sources only know destination of some other sources as given by a communication graph, with the goal of controlling network congestion.

Brightwell, Ott, and Winkler [20] study load assignment when agents that have some load need to assign it to the server without communicating with other agents. There are n agents and each agent i has a job of size X_i which is a random variable, and the variables are pairwise independent. There is a server of capacity T. They would like to decide if an agent i should or should not assign its job to the server based on the value of the variable X_i only, so as to minimize the expected square of the distance between T and the total load assigned to the server by all the clients (intuitively we want to make it close to the target T without communication). The authors consider strategies where each agent i has a measurable subset $S_i \subseteq \mathbb{R}$, and assigns load iff $X_i \in S_i$, and show that it is enough to consider subsets that are real intervals that start or end at zero. The authors show the existence of an optimal solution and develop an optimality condition (from which it is in general difficult to find what the intervals should be!). The authors characterize some special cases, including the case when the variables are all uniform on $[0, 1]$. The authors study two other models: one when each agent may assign a fraction of its job to the server (and not assign the remaining fraction), and the other when there is more than one server, each of which has its own threshold and weight specifying the relative importance of achieving an assignment close to the threshold for the server. The authors characterize some properties of optimal solutions.

Georgiades, Mavronicolas, and Spirakis [64] study a similar load assignment problem. They consider arbitrary number of agents n where there is no communication between processors whatsoever, and arbitrary computable load assignment strategies. There are two servers that process jobs, and the servers have some given capacity that is not necessarily 1. They study two families of load assignment strategies: (a) strategies that cannot see the size of jobs before making a decision about which server to send a job to for processing, and (b) strategies that can make decisions based on the size of the job. They completely settle these cases, by showing that their strategies minimize the chances of overloading any server.

The paper of Papadimitriou and Yannakakis [121] studies how distributed agents can solve a specific optimization problem with incomplete information about the problem. The problem is a linear program with a matrix A with nonnegative entries, and decision variables $x_1, \ldots, x_n$ which must be nonnegative, while we are interested in maximizing their sum under the condition that $Ax \leq \mathbf{1}$ (i.e., the cross product of each row of the matrix and the vector $\langle x_1, \ldots, x_n \rangle$ can be at most 1). There is an interesting special case of the linear programming problem, that encodes load balancing with limited information, across servers with bounded capacity. Suppose that there are n agents and m servers, and that each agent i wants to assign a job of size 1 to each of servers in the set S_i. We assume that for any server there are at most d agents that want to assign a job to the server (any server is in at most d of these sets). Each agent is allowed to assign to a server a fraction of the job of size 1 destined for the server. Each server has capacity 1 and this capacity cannot be exceeded by the assignments of agents. Each agent has incomplete information about where other agents want to assign their jobs. Specifically, the agent i knows, for each server j (in S_i), which other agents also want to send the job to the server j. Agents must decide what fraction of jobs to send to each server, given this limited information about other agents intents. The authors show a d competitive algorithm that is optimal.

The authors study two variants of the linear programming problem, which differ by what type of information is available to the agents, and give competitive algorithms.

The paper by Awerbuch and Azar [9] studies solutions to linear programming where agents can accumulate some information about the parts of the problem known to other agents. There are n agents and n severs. Each agent a wants to assign a job, or its fraction, to a server from some fixed subset $P(a)$. Each job of an agent a have size s_a, and a server s has capacity c_s. The agent a can only communicate with servers in $P(a)$. It is assumed that the minimum capacity of any server is order of $\log n$ factor greater than the maximum size of any job. The goal is to assign fractions of jobs so that the total "throughput", i.e., sum of the sizes of jobs assigned to servers, is as large as possible. The authors give a $O(\log n)$ competitive algorithm where agents communicate with servers in $O(\log n)$ rounds (since in each round an agent a may only contact servers in $P(a)$, if $P(a)$ and $P(a')$ intersect, then agent a can reveal its part of the matrix to the agent a'). An agent initially assigns small fraction of its jobs to servers, and then iteratively learns about current load on the servers, and increases the assignments. The authors also study generalizations of the problem.

Bartal, Byers, and Raz [12] study how to improve the quality of an answer to a linear programming problem at the expense of time during which agents gain more knowledge about the matrix. The authors assume: specific communication topology between agents, that each agent initially knows only a part of the matrix, and that the computation progresses in rounds during which any agent can send a fixed size message to any of its neighbors sharing, for example, the knowledge of the matrix. The authors show a distributed algorithm that achieves $1+\epsilon$ competitiveness using polylogarithmic number of rounds of communication between agents.

Consequences of load balancing with stale information have been studied by several authors. Eager, Lazowska and Zahorjan [52] show that in some cases gathering too much information may be useless. Mitzenmacher [111] demonstrates that using stale information can actually be counterproductive. Mirchandaney, Towsley and Stankovic [110] show that long message delays can encourage devices to balance load across small neighborhood of devices.

2.5 Sequencing to avoid redundancy

There exist results on producing sequences of numbers with certain intersection properties. Our scheduling problem is related to these results.

Gasieniec, Pelc, and Peleg [62] study a variation of the wakeup problem [55]. In broadcast systems, such as Ethernet or radio networks, processors share a communication medium. The medium delivers a message only when one processor broadcasts. In such systems, it is desirable to establish a pattern of access to the shared medium that avoids collisions. This problem can be abstracted as the wakeup problem, where some processors are waken up by the adversary at some points of time, and the goal is that processors succeed in waking up all other processors by means of broadcasting. This should be done as quickly as possible, counting from the moment when the first processor is woken up. This can be achieved by producing, for each processor, a sequence saying when it is to be silent, and when it is to broadcast, so as to avoid collisions. The authors consider four cases, that are combinations of global/local clock ticking at the same pace, and known/unknown number of processors participating in the protocol. The authors give deterministic and randomized constructions of sequences with guarantees on the wakeup time, and study lower bounds.

Rosenberg [127] studies a problem where a server is to assign tasks to clients. There is an infinite number of tasks, indexed with natural numbers. Clients register at the server, and each client obtains a unique identifier. Later clients repeatedly request tasks from the server, compute them, and return their results to the server. The goal of the author is to find a function f, that for a given an identifier i of a client and the number x of tasks already sent to this client, yields the index $f(i,x)$ of the task that should be sent to the client next. This function should be easy to compute. Also its inverse should be easy to compute, so as to make it easier for the server to decide which client did which task. The tasks should be assigned is such a way, that no task is assigned to more than one client, and each task is eventually assigned to some client (i.e., f is a bijection), and that tasks are assigned in the order of their indices. The author studies functions in the form $f(i,x) = a(i) \cdot x + b_i$ parameterized with a “slope” function $a(i)$. The author is interested in constructing functions, each with specific growth of $a(i)$, so as to assign tasks to clients in the approximate order of task indexes given clients with specific computational speeds. The author presents a few functions f that achieve these goals.

2.6 Mathematical tools

We report some properties of vector spaces over finite fields, that will be used in Chapter 6. Let GF(q) denote the finite field with q elements, where q is a prime power, V be a vector space of dimension m over the field. We cite a counting lemma that lets us find the number of certain subspaces of V ([77], Lemma 1.6). Let W be a w-dimensional subspace of V, $w \geq 0$, and let $h \geq 0$. We let the number of $(w+h)$-dimensional subspaces of V containing W be denoted by $\Delta(q, m, w, h)$.

Lemma 2.6.1 ([77]). *(i)* $\Delta(q, m, w, h) = \prod_{i=0}^{h-1} \frac{q^{m-w}-q^i}{q^h-q^i}$, *(ii)* $\Delta(q, m, 0, d) = \Delta(q, m, 0, m-d)$*, and (iii)* $\Delta(q, m, w, h) = \Delta(q, m-w, 0, h)$.

We use several properties of *inner product spaces* (see, e.g., [86]). Recall that an inner product space is a structure consisting of V and $\langle \cdot, \cdot \rangle$, where V is a vector space over a field K, and $\langle \cdot, \cdot \rangle : V \times V \rightarrow K$ is an inner product, i.e., for all $\alpha, \beta, \gamma \in V$ and $a, b \in K$, we have $\langle \alpha, \beta \rangle = \langle \beta, \alpha \rangle$ and $\langle a\alpha + b\beta, \gamma \rangle = a\langle \alpha, \gamma \rangle + b\langle \beta, \gamma \rangle$. For any two vectors $\alpha, \beta \in V$, we say that α is orthogonal to β (we write $\alpha \perp \beta$), if $\langle \alpha, \beta \rangle = 0$. For any linear subspace U of V, an *orthogonal complement* of U, denoted as $U^{\perp}$, is defined to be a linear space that contains all vectors orthogonal to every vector in U, i.e., $U^{\perp} = \{\alpha \in V \mid \forall \beta \in U,\ \alpha \perp \beta\}$. For a subspace U and a vector α, we write $\alpha \perp U$ if $\alpha \in U^{\perp}$. For two subspaces U and W, we write $U \perp W$ if and only if $\forall \alpha \in U, \alpha \perp W$. We focus here on finite dimensional spaces; in this case an inner product space is called *non-singular* if the matrix $A = (a_{i,j})$, $a_{i,j} = \langle \alpha_i, \alpha_j \rangle$, $1 \leq i, j \leq m$, is non-singular (i.e., $det(A) \neq 0$), where $\alpha_1, \ldots, \alpha_m$ is a base of V.

Lemma 2.6.2. *Let U be a d-dimensional subspace of m-dimensional non-singular inner product space V. The space $U^{\perp}$ is unique and has dimension $m-d$, $(U^{\perp})^{\perp} = U$. Let K be any subspace of V. Then $D \subseteq K$ if and only if $K^{\perp} \subseteq D^{\perp}$.*

Our analysis in Chapter 5 uses tools for bounding tails of distributions. We first record some standard Chernoff bounds. See [2], for example, for discussion and proofs.

Proposition 2.6.3 ([2]). *Let X_i be a family of t independent random variables for which*

$$X_i = \begin{cases} 1-p & \textit{with probability}\, p, \\ -p & \textit{with probability}\, 1-p. \end{cases}$$

Then for each $a > 0$,

$$\mathsf{Pr}\left[\sum_i X_i > a\right] \leq e^{a-(a+tp)\ln(1+\frac{a}{tp})}, \tag{1}$$

$$\mathsf{Pr}\left[\sum_i X_i > a\right] \leq e^{-\frac{a^2}{2tp}+\frac{a^3}{2(tp)^2}}, \textit{ and} \tag{2}$$

$$\mathsf{Pr}\left[\sum_i X_i < -a\right] \leq e^{-\frac{a^2}{2tp}}. \tag{3}$$

(Inequality (2) follows from (1) by recalling that $\ln(1+x) \geq x^2 - \frac{1}{2}x^2$*, for all* $x \geq 0$ *.)*

Another technique for bounding tails of distributions uses Azuma's inequality. Recall that a martingale is a sequence $X_1, X_2, \ldots, X_n$ of real valued random variables for which $\mathbf{E}[X_{i+1} \,|\, X_i] = X_i$. A martingale is said to have *Lipschitz constant* L, if for each i, $|X_i - X_{i-1}| \leq L$. (See [2, §7] for a general discussion of discrete martingales and a proof of Azuma's inequality.)

Theorem 2.6.4 (Azuma's Inequality, [75, 10]). *Let* $X_0, X_1, \ldots, X_k$ *be a martingale with Lipschitz constant* 1 *and assume that* X_0 *is a constant random variable* $X_0 = c$*. Then* $\mathsf{Pr}\left[X_k - X_0 > \lambda\sqrt{k}\right] \leq e^{-\frac{\lambda^2}{2}}$.

Observe that Azuma's inequality does not require that the X_i be independent.

2.7 Design theory

Set systems with prescribed intersection properties have been the object of intense study by both the design theory community and the extremal set theory community (see, e.g., [77, 16, 21] for a surveys). We review a notion of a packing. A u-(t, k, λ) *packing design* [109, 133] is a family of subsets $\mathcal{S} = (S_1, \ldots, S_n)$ of the set $[t]$ with the property that each $|S_i| = k$ and any set of u elements of $[t]$ is a subset of at most λ of the S_i. (The subsets S_i are typically referred to as *blocks*.) A packing design is called a *design* if it has a more regular structure;

namely, we require that subsets are distinct, and that any subset of u elements of $[t]$ is included in exactly λ of the S_i's.

We will apply the following design-theoretic construction several times in the sequel. Treating $\mathrm{GF}(q)^m$, $m \geq 3$, as a vector space over $\mathrm{GF}(q)$, the design will be given by the lattice of linear subspaces of $\mathrm{GF}(q)^m$. There are $n = \Delta(q, m, 0, m-2)$ distinct $(m-2)$-dimensional subspaces of $GF(q)^m$ denoted $A_1, \ldots, A_n$, and $t = \Delta(q, m, 0, m-1)$ distinct subspaces of dimension $m-1$ denoted $B_1, \ldots, B_t$. Any subspace of dimension $m-1$ contains $k = \Delta(q, m-1, 0, m-2)$ subspaces of dimension $m-2$. Note the intersection of any two different subspaces of dimension $m-1$ is a subspace of dimension $m-2$ (by the theorem of the dimension of the sum of two subspaces). The 2-$(n, k, 1)$ design consists of t sets $L_u^m = \{i \mid A_i \subset B_u\}$. Note that each subspace of dimension $m-2$ is contained in exactly $h = \Delta(q, m, m-2, 1)$ subspaces of dimension $m-1$.

Theorem 2.7.1 ([77]). *Let $n = \frac{(q^m-1)(q^{m-1}-1)}{(q^2-1)(q-1)}$, $t = \frac{q^m-1}{q-1}$, where q is a prime power. Then the sets $L_1^m, \ldots, L_n^m$ possess the following properties: each L_u^m is a subset of $[t]$, has cardinality $\frac{q^{m-1}-1}{q-1}$, for each $u \neq v$, $|L_u^m \cap L_v^m| = 1$, and any element $i \in [t]$ appears in exactly $q+1$ distinct sets.*

When $m = 3$, the sets $L_1^m, \ldots, L_n^m$ form blocks of a 2-$(q^2+q+1, q+1, 1)$ design, and we omit the superscript m.

Theorem 2.7.2. *Let $n = q^2+q+1$, where q is a prime power. Then the sets $L_1, \ldots, L_n$ possess the following properties: each L_u has cardinality $q+1$, for each*

$u \neq v$, $|L_u \cap L_v| = 1$, and any element $i \in [n]$ appears in exactly $q+1$ distinct sets. We note also that if q is prime, the first element of each set can be calculated in $O(\log n)$ time; each subsequent element can be calculated in $O(1)$ time.

(We assume throughout that addition or multiplication of two $\log(\max\{n, t\})$-bit numbers can be performed in $O(1)$ time.)

Design theory [77] has already been applied in numerous contexts in computer science. Two surveys by Colbourn and van Oorschot [33], and by Colbourn, Dinitz and Stinson [32] are excellent introductions to these applications. Here we review other related problems.

Frankl, Rödl and Wilson [58] show certain uniformity property of some matrices. Specifically they show that in Hadamard matrices, small submatrices have about the same number of zeros and ones in them. Our constructions of schedules yield matrices that have duplicate elements "evenly" distributed across the matrices.

Mullender and Vitányi [114] introduce a problem called distributed match making and show a relationship of this problem to distributed control issues such as name server, mutual exclusion (see e.g., [82] for recent work in this area), and replicated data management. Solutions to this problem involve producing sets with certain intersection properties. The authors use a multidimensional construction similar to the one presented in Section 6.3.

Karp and Wigderson [88] show a parallel algorithm for the maximal independent set problem. They first present a solution that uses random sets, and then use objects from design theory to derandomize their solution. Specifically, they substitute these random sets with deterministically constructed sets, and argue about the quality of the resulting solution. This is similar in spirit to the derandomization technique that we present in Chapter 6.

Shamir and Schuster [131] show a construction of a network using vector spaces over finite fields. Nodes are associated with subspaces of dimension about half of the dimension of the space, and edges reflect containment between the subspaces. This is similar to the technique of building a bipartite graph presented in Section 6.4.1. The number of nodes in the network is about square root of the number of nodes allowed by the Moore bound for the diameter and the degree of nodes in the network constructed. The authors give a randomized method for routing that does the routing in the order of diameter of the network parallel steps, with high probability.

Chapter 3

Definitions, problem statement, and relationships to other problems

This chapter defines the model of distributed system and states the disconnected cooperation problem studied in the dissertation. Then relationships between the problem and other problems encountered in design and coding theories are given.

3.1 Schedules and waste

We consider the abstract setting where n asynchronous processors must perform t abstract tasks. We model tasks as distinct numbers from $[t]$ (recall that $[t] = \{1, \ldots, t\}$), and processors as distinct numbers from $[n]$. We assume that there is a (deterministic) function $f : [t] \rightarrow R$, where R is the set of *results*, and say that a processor *performs* (or executes) a task i, when it evaluates the function f at a point i. (Tasks modeled this way are called independent and idempotent, because the execution of any task yields the same result when it is performed more than once, and the result of execution of any task does not depend on the order in which other tasks are executed.) We assume that each processor can perform addition or multiplication of two $\log(\max\{n, t\})$-bit numbers in $O(1)$ time.

At the beginning of a computation, communication is not available, but at some *a priori* unknown point of time, a group of *a priori* unknown processors rendezvous. Processors compute at all times during the period of disconnection, in between communications.

We focus on the scheduling problem discussed above, abstracted as follows. An (n, t)-*schedule* is a tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$ of n permutations of the set $[t]$. When $n = 1$, it is elided and we simply write t-*schedule.*

An (n, t)-schedule immediately gives rise to a strategy for n isolated processors who must complete t tasks until communication between some pair (or group) is established: each processor i simply proceeds to complete the tasks in the order prescribed by σ_i. Suppose now that k of these processors, say $q_1, \ldots, q_k \in [n]$, happen to rendezvous at a time when the ith processor in this group, q_i, has completed a_i tasks. We call a_i the *progress* of processor i at the instant of the rendezvous. Ideally, the processors would have completed disjoint sets of tasks, so that the total number of tasks completed is $\sum_i a_i$. As this is too much to hope for in general, it is natural to attempt to bound the gap between $\sum_i a_i$ and the actual number of distinct tasks computed. This gap we call *waste*:

Definition 3.1.1. *If $S = \langle \sigma_1, \ldots, \sigma_n \rangle$ is a (n, t)-schedule and $\langle a_1, \ldots, a_k \rangle \in \mathbb{N}^k$, the* waste *function for S is*

$$\mathcal{W}_S(a_1, \ldots, a_k) = \max_{\langle q_1, \ldots, q_k \rangle} \left(\sum_i^k a_i - \left| \bigcup_i^k \sigma_{q_i}([a_i]) \right| \right),$$

this maximum taken over all k tuples $\langle q_1, \ldots, q_k \rangle$ of distinct elements of $[n]$.

Here (and throughout), if $\phi : X \to Y$ is a function and $S \subset X$, we let $\phi(S) = \{\phi(x) \mid x \in S\}$. For example $\sigma_i([a])$ is the set of the first a tasks performed by processor i. For a specific sequence of numbers $a_1, \ldots, a_k$, $\mathcal{W}_S(a_1, \ldots, a_k)$ captures the worst-case number of redundant tasks performed by any collection of k processors, when the ith process has performed the first a_i tasks of its schedule. One immediate observation is that bounds on *pairwise* waste $\mathcal{W}_S(\cdot, \cdot)$ can be naturally extended to bounds on *k-wise* waste $\mathcal{W}_S(\underbrace{\cdot, \ldots, \cdot}_{k})$: specifically, note that if S is an (n, t)-schedule then

$$\mathcal{W}_S(a_1, \ldots, a_k) \leq \sum_{i<j} \mathcal{W}_S(a_i, a_j)$$

just by considering the first two terms of the standard inclusion-exclusion rule. Equality in this relationship is and nearly attained by randomized schedules (see Section 5.2). We shall content ourselves to focus the investigation on pairwise waste – the function $\mathcal{W}_S(a, b)$.

3.2 Relationships to other problems

We observe that a specific *sequence* of packing designs can be used to construct schedules with limited waste. Let u-$(t, k, 1)$ be a packing design with n blocks $B_1, \ldots, B_n$. Observe that if 2 blocks had u or more elements in common, then the set of these u elements would be on 2 blocks, which is impossible by the definition of u-$(t, k, 1)$ packing. Thus the cardinality of the intersection of any 2 distinct blocks is at most $u - 1$. Using this remark, we can construct an (n, t)-schedule S with waste bounded by $u - 1$ for progress k, just by taking any sequence of n permutations on $[t]$ for which $\sigma_i([k]) = B_i$. Unfortunately, this construction offers satisfactory control of 2-waste only for the specific pair (k, k). Furthermore, considering that the construction only determines the *sets* $\sigma_i([k])$ and $\sigma_i([t] \setminus [k])$, the ordering of these can be conspiratorially arranged to yield poor bounds on waste for values of progress other than k. Our goal is to construct schedules with satisfactory control on waste for all pairs (a, a).

We can construct an (n, t)-schedule $S = (\sigma_1, \ldots, \sigma_n)$ with waste controlled at a finer granularity of values of progress, from a *sequence* of packing designs that have specific containment properties. For a given n and t, and $k_1 \leq \ldots \leq k_t$, let

P_u be a u-$(t, k_u, 1)$ packing design with n blocks $B_{u,1}, \ldots, B_{u,n}$. Let, for any i, the corresponding blocks of packing designs form a monotonically nondecreasing sequence, i.e., $\emptyset = B_{0,i} \subseteq B_{1,i} \subseteq B_{2,i} \ldots \subseteq B_{t,i} \subseteq [t]$. Such sequence of packings is called *multiply nested* because blocks of one packing are nested, or contained, in the corresponding blocks of the next packing in the sequence. (Nested designs were introduced by Preece [123]; see also [112] for a recent survey.) A permutation σ_i is obtained by assigning the elements in consecutive "deltas" in arbitrary order to consecutive arguments i.e., $\sigma_i([|B_{u,i}|+1, |B_{u+1,i}|]) = B_{u+1,i} \setminus B_{u,i}$, for $0 \leq u < t$, and $\sigma_i([|B_{t,i}| + 1, t]) = [t] \setminus B_{t,i}$. (There is also a dual construction for packing designs that are called *regular* [14].)

Theorem 3.2.1. *Any (n, t)-schedule S constructed as above from t nested packing designs has waste bounded by $\mathcal{W}_S(k_u, k_u) < u$, for any u, $1 \leq u \leq t$.*

This theorem teaches us that we should seek a packing design that, for any u, maximizes k_u. Fundamental limitations on how large k_u can be are known. These bounds are derived from the so called packing number. The *packing number* $D_\lambda(t, k, u)$ is the maximum number of blocks n in any u-(t, k, λ) packing. Johnson [81] gave an upper bound on the packing number.

Theorem 3.2.2 (Second Johnson bound, [81]). *The packing number $D_1(t, k, u)$ is bounded by*

$$D_1(t, k, u) \leq \frac{t(k + 1 - u)}{k^2 - (u - 1)t}$$

This bound lets us immediately derive a lower bound on waste.

Corollary 3.2.3.

$$\mathcal{W}_S(k,k) \geq \frac{n}{t(n-1)}k^2 - \frac{k+t+1}{n-1} \ .$$

Proof. Let us take any (n,t)-schedule $D = (\delta_1, \ldots, \delta_n)$ and suppose that its waste is $\mathcal{W}_D(k,k) = u-1$. The n sets $\delta_i([k])$ form n blocks of a u-$(t,k,1)$ packing design. From the Second Johnson bound we get that $n \leq \frac{t(k+1-u)}{k^2-(u-1)t}$, and so $u \geq \frac{n}{t(n-1)}k^2 - \frac{k+t-n}{n-1}$. Hence $\mathcal{W}_S(k,k) \geq \frac{n}{t(n-1)}k^2 - \frac{k+t-n}{n-1} - 1$, and the result follows. □

Our lower bound, presented in Chapter 4, generalizes the second Johnson Bound [81] for the case when two processors execute *different* number of tasks prior to their rendezvous. We note that one can trivially obtain a lower bound on $\mathcal{W}_S(a,b)$ when $a<b$, just by taking the bound for $k=a$ in the Corollary 3.2.3. However, this bound is weaker than the one we derive in Chapter 4.

It appears that there is no existing construction of multiply nested packings that, for arbitrary large n and t, yields an (n,t)-schedule that has waste that compares favorably with the lower bound.

Our problem can be reformulated in the language of the classical coding theory problem called the sphere packing problem (see, e.g., [38]). Consider the metric space M_k of all subsets of $[t]$ of cardinality k, with distance between two sets defined as the number of elements that appear in only one of the two sets. The problem of minimizing the worst case intersection between sets of cardinality k can be reformulated as selecting n points from the metric space, so as to maximize minimum distance between the selected points. The difference between our problem and coding theory is that coding theory studies the Hamming distance, while we need a solution for the sphere packing problem with the different distance that we defined in this paragraph. An additional difference is that in order to control intersection for arbitrary progress of processors, we would need the sets of points selected for $M_1, M_2, \ldots, M_t$ to have a specific nesting property: we obtain the point number i selected for M_k to be equal to the point number i selected for M_{k-1} union some element of $[t]$ (i.e., we do not have complete freedom in selecting points in M_k).

Chapter 4

A lower bound on waste

Coordination of a distributed computation in the presence of arbitrary changes of communication is a futile task. In the extreme case where all processors are isolated from the beginning of computation and remain isolated, the number of tasks performed by any algorithm is $t \cdot n$, which can be achieved by a trivial oblivious algorithm, where each processor performs all tasks. However, as we pointed out earlier, it is possible to schedule the work of a pair of processors so that each can perform up to $t/2$ tasks without a single task being performed redundantly. This suggests that when communication is established early, i.e., when each processor has performed only a part of t tasks, a more interesting lower bound may be obtained. Such lower bound will be shown in this chapter.

4.1 Lower bound on waste

If we insist that among the n total processors, any two processors, having executed the same number of tasks t', where $t' < t$, perform *no* redundant work, then it must be the case that $t' \leq \lfloor t/n \rfloor$. In particular, if $n = t$, then the pairwise waste jumps to one if any processor executes more than one task.

The next natural question is: how many tasks can processors complete before the lower bound on pairwise redundant work is 2? In general, if any two processors perform t_1 and t_2 tasks respectively, what is the lower bound on pairwise redundant work? In this section we answer these questions. The answers contain both good and bad news: given a fixed t, the lower bound on pairwise redundant work starts growing slowly for small t_1 and t_2, then grows quadratically in the schedule length as t_1 and t_2 approach t.

We begin with a short lemma that has geometric interpretation (see [129]).

Lemma 4.1.1. *For any integers a, b, t, and n such that $1 \leq a \leq b \leq t$, $2 \leq n$, let integer variables x_i, y_i, $1 \leq i \leq t$, satisfy the constraints $0 \leq x_i \leq y_i \leq n$, $\sum_{i=1}^{t} x_i = na$, and $\sum_{i=1}^{t} y_i = nb$. Then the sum $\sum_{i=1}^{t} x_i y_i$ is bounded from below by*

$$\sum_{i=1}^{t} x_i y_i \geq \frac{(na)^2}{t - b + a} \ .$$

Proof. The proof of the lemma is by induction. We first show that the lemma is true for $1 \leq a = b \leq t$, and then show that the lemma is true for $1 \leq a < b \leq t$ given that it is true for a, $b - 1$, and $t - 1$. Observe that applying this rule inductively exhausts all possible choices for a, b and t, which will complete the inductive proof.

For the base case let $1 \leq a = b \leq t$. Then $\sum_{i=1}^{t} x_i = \sum_{i=1}^{t} y_i$. But $x_i \leq y_i$, and so $x_i = y_i$. Thus $\sum_{i=1}^{t} x_i y_i = \sum_{i=1}^{t} x_i^2$. By the Cauchy-Schwarz's inequality, this sum can be bounded from below by

$$\sum_{i=1}^{t} x_i y_i \geq \frac{1}{t}\left(\sum_{i=1}^{t} x_i\right)^2 = \frac{(na)^2}{t-b+a} \ .$$

For the inductive step we pick $1 \leq a < b \leq t$ and show that the lemma holds given that it holds for a, $b-1$, $t-1$. Take any x_i, y_i, $1 \leq i \leq t$ such that $\sum_{i=1}^{t} x_i = an$, $\sum_{i=1}^{t} y_i = bn$, $1 \leq a < b \leq t$, $0 \leq x_i \leq y_i \leq n$. We apply two transformations to the sequences of variables x_i and y_i. Each transformation never increases the sum $\sum_{i=1}^{t} x_i y_i$. Then we will apply the inductive hypothesis to the sequences after transformations, to bound the sum from below.

For the first transformation we do the following. (It is convenient to draw a plot of a function $f(i) = y_i$ above the function $g(i) = x_i$, each bounded from above by $h(i) = n$.) After possibly renumbering, let x_t be a smallest x_i i.e., for all i, $x_i \geq x_t$. Since $b > a$, we have that $\sum_{i=1}^{t} y_i - \sum_{i=1}^{t} x_i \geq n$, and so $\sum_{i=1}^{t-1} y_i - \sum_{i=1}^{t-1} x_i \geq n - y_t$. Let $u = n - y_t$. Thus we can increase y_t by u, and decrease some of the y_i, $i \in [t-1]$, cumulatively by u without violating the $x_i \leq y_i$ constraint. We denote the resulting values of variables by $\bar{y}_i$. Since x_t is minimal, the sum $\sum_{i=1}^{t} x_i \bar{y}_i$ for the resulting sequences can only be smaller compared to $\sum_{i=1}^{t} x_i y_i$. Notice that the resulting sequence has $\bar{y}_t = n$. We now transform the sequences again. Since $b > a$, we have that $\sum_{i=1}^{t-1} \bar{y}_i = (b-1)n \geq an = \sum_{i=1}^{t} x_i$, and so we can increase some variables x_i, $1 \leq i \leq t-1$, cumulatively by x_t, without increasing any $\bar{y}_i$, $1 \leq i \leq t-1$, and without violating the $x_i \leq \bar{y}_i$ constraint. Moreover, let us decrease x_t to 0. We denote the resulting values by $\bar{x}_i$. The impact of the second transformation on the value of the sum $\sum_{i=1}^{t} x_i \bar{y}_i$ is as follows. Setting $\bar{x}_t$ to 0 reduces the sum by nx_t (because $\bar{y}_t = n$), while increasing the value of an x_i by 1 can increase the sum by at most n, hence cumulatively by at most nx_t. Consequently, $\sum_{i=1}^{t} x_i \bar{y}_i \geq \sum_{i=1}^{t-1} \bar{x}_i \bar{y}_i$.

Finally notice that after the two transformations, we have $\sum_{i=1}^{t-1} \bar{x}_i = an$, and $\sum_{i=1}^{t-1} \bar{y}_i = (b-1)n$, so we can use the inductive hypothesis to bound the transformed sum from below by $\sum_{i=1}^{t} x_i y_i \geq \frac{(na)^2}{t-b+a}$. The result thus follows. □

Now we proceed to the lower bound, which generalizes the second Johnson Bound [81] for the case when two processors execute *different* number of tasks prior to their rendezvous.

Theorem 4.1.2. *Let $\mathcal{P} = \langle \pi_1, \ldots, \pi_n \rangle$ be an (n,t)- schedule and let $0 \leq a \leq b \leq t$. Then*

$$\mathcal{W}_{\mathcal{P}}(a,b) \geq \frac{na^2}{(n-1)(t-b+a)} - \frac{a}{n-1}.$$

Proof. We obtain the lower bound by computing the expected waste of a pair of t-schedules selected at random from $\mathcal{P}$. Let $\lambda = \mathcal{W}_{\mathcal{P}}(a,b)$. Consider selection of i and j independently at random in the set $[n]$. We focus on the expected value of the random variable

$$|\pi_i([a]) \cap \pi_j([b])|.$$

There are a total of n^2 pairs for i and j; if $i \neq j$ then the cardinality of the intersection is bounded above by λ. If $i = j$ then this cardinality is obviously a. Hence

$$\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] \leq \frac{n(n-1)\lambda + n \cdot a}{n^2} = \frac{\lambda(n-1) + a}{n}. \tag{4}$$

Consider now the t random variables X_τ, indexed by $\tau \in [t]$, defined as follows: $X_\tau = 1$ if $\tau \in \pi_i([a]) \cap \pi_j([b])$, and 0 otherwise. Then $\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] = \mathbf{E}[\sum_{\tau \in [t]} X_\tau]$, and by linearity of expectation,

$$\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] = \sum_{\tau \in [t]} \mathbf{E}[X_\tau] = \sum_{\tau \in [t]} \mathsf{Pr}\left[\tau \in \pi_i([a])\right] \cdot \mathsf{Pr}\left[\tau \in \pi_j([b])\right],$$

since i and j are independent.

Now we introduce the function $x^m(\tau)$, equal to the number of prefixes of schedules of length m to which τ belongs, i.e., $x^m(\tau) = |\{i : \tau \in \pi_i([m])\}|$. Then

$$\begin{aligned}\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] &= \sum_{\tau \in [t]} \mathsf{Pr}\left[\tau \in \pi_i([a])\right] \cdot \mathsf{Pr}\left[\tau \in \pi_j([b])\right] = \sum_{\tau \in [t]} \frac{x^a(\tau)x^b(\tau)}{n^2} \\ &= \frac{1}{n^2} \sum_{\tau \in [t]} x^a(\tau)x^b(\tau)\ .\end{aligned} \tag{5}$$

Noting that $\sum x^a(\tau) = an$ and $\sum x^b(\tau) = bn$, we apply Lemma 4.1.1 to the last expression in (5) above and combine this with the bound of (4):

$$\frac{1}{n^2} \cdot \frac{(na)^2}{(t-b+a)} \leq \frac{1}{n^2} \sum_{\tau \in [t]} x^a(\tau)x^b(\tau) \leq \mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] \leq \frac{(n-1)\lambda + a}{n}$$

whence

$$\lambda \geq \frac{a}{n-1}\left(\frac{na}{t-b+a} - 1\right),$$

as desired. □

For example, when processors perform the same number of tasks $a = b$ and $n = t$, then the worst case number of redundant tasks for any pair is at least $\frac{a^2-a}{t-1}$. This means that (for $n = t$) if a exceeds $\sqrt{t} + 1$, then the number of redundant task is at least 2.

Corollary 4.1.3. *For $t = n$, if $a > \sqrt{n - 3/4} + \frac{1}{2}$ then any n-processor schedule of length a for t tasks has worst case pairwise waste at least* 2*.*

Chapter 5

Random schedules

This chapter studies waste of randomized schedules. When the processors are endowed with a reasonable source of randomness, a natural candidate scheduling algorithm is one where processors select tasks by choosing them uniformly among all tasks they have not yet completed. This give rise to an (n, t)-schedule $R = \langle \rho_1, \ldots, \rho_n \rangle$, where permutations ρ_i are selected independently, and uniformly at random from $S_{[t]}$, the collection of all permutations of the set $[t]$. We demonstrate that the (n, t)-schedule R has waste that is close to the lower bound.

5.1 Pairwise waste for random schedules

Our objective is to show that random schedules R have controlled pairwise waste with high probability. This amounts to bounding, for each pair i, j and each pair of numbers a, b, the overlap $|\rho_i([a]) \cap \rho_j([b])|$. Observe that when these ρ_i are selected at random, the expected size of this intersection is ab/t. By showing that the actual waste is very likely to be close to this expected value, one can conclude that waste is bounded for *all* sufficiently long prefixes.

The next technical lemma demonstrates that the cardinality of intersection of two random sets is close to the expected cardinality with high probability.

Claim 5.1.1. *Let $n, t \geq 1$, and $a, b \geq 7\sqrt{t}\ln(2nt)$. Let $A \subseteq [t]$ and $B \subseteq [t]$ be chosen randomly, A among all sets of size a, and B among all sets of size b. Then*

$$\mathsf{Pr}\left[|A \cap B| > ab/t + \Delta\right] \leq \frac{1}{2n^3t^3} \ ,$$

where $\Delta = 11\sqrt{\frac{ab}{t}\ln(2nt)}$.

Proof. Observe that $\mathbf{E}[\ |A \cap B|\ ] = ab/t$. We need to see that this intersection is close to the expected value with high probability.

Fixing for the moment the set A, for each $a \in A$, let X_a be the indicator variable for the event $a \in B$. These X_a are not independent, which slightly complicates the analysis. To begin, we approximate the selection of the set B by the selection of a set $\hat{B}$, which will contain each element $x \in [t]$ independently with probability $(1+\delta)b/t$, where δ is a parameter we shall set presently. Observe that, $\mathbf{E}\left[\left|\hat{B}\right|\right] = (1+\delta)b$. We have now two goals:

- control the intersection of $\hat{B}$ and A,
- correct for the distortion induced by the approximation of B by $\hat{B}$.

We first concentrate on the second goal. Let π be a permutation of $[t]$ chosen uniformly at random (and independently of A and $\hat{B}$). When $\hat{B}$ has cardinality at least b, let $\hat{B}_\pi$ denote the set consisting of the first b elements of $\hat{B}$ with the order induced by π. Then define the map $\phi : 2^{[t]} \times S_t \to 2^{[t]} \cup \{\perp\}$ so that

$$\phi(\hat{B}, \pi) = \begin{cases} \hat{B}_\pi & \text{if } \left|\hat{B}\right| \geq b, \\ \perp & \text{otherwise.} \end{cases}$$

The range of ϕ is $\{S \subseteq [t] \mid |S| = b\} \cup \{\perp\}$, where $\phi^{-1}(\perp)$ is exactly the collection of sets not large enough to afford a subset of size b. Now consider the random variable $\phi(\hat{B}, \pi)$, and observe that the distribution on $2^{[t]}$ induced by conditioning on $\phi(\hat{B}, \pi) \neq \perp$ is precisely the distribution of B. Let $\hat{\epsilon}$ be the probability that $\phi(\hat{B}, \pi) = \perp$. Recall that for any two events E_1 and E_2, $\mathsf{Pr}\left[E_1 \mid E_2\right] \leq \mathsf{Pr}\left[E_1\right] + \mathsf{Pr}\left[\ \overline{E_2}\ \right]$. Observe that when $\phi(\hat{B}, \pi) \neq \perp$, $\hat{B}_\pi \subseteq \hat{B}$. Hence for any sets $A, B \subseteq [t]$ and any η,

$$\begin{aligned} \mathsf{Pr}\left[|B \cap A| > \eta\right] &= \mathsf{Pr}\left[\left|\hat{B}_\pi \cap A\right| > \eta \,\middle|\, \phi(\hat{B}, \pi) \neq \perp\right] \\ &\leq \mathsf{Pr}\left[\left|\hat{B} \cap A\right| > \eta \,\middle|\, \phi(\hat{B}, \pi) \neq \perp\right] \leq \mathsf{Pr}\left[\left|\hat{B} \cap A\right| > \eta\right] + \hat{\epsilon}. \end{aligned} \tag{6}$$

To bound $\hat{\epsilon}$, observe that $\hat{\epsilon} = \mathsf{Pr}\left[\left|\hat{B}\right| < b\right] \leq e^{-\frac{\delta^2 b^2}{2tp}} = e^{-\frac{\delta^2 b}{2(1+\delta)}}$, by the Chernoff bound (3). We now select $\delta = 2\sqrt{\ln(4n^3t^3)/b}$ and recall that $b \geq 7\sqrt{t}\ln(2nt) \geq$

$4\ln(4n^3t^3)$. As a result, $\delta \leq 1$, and so $\frac{\delta^2}{1+\delta} \geq \frac{\delta^2}{2}$, and we can further bound $\hat{\epsilon}$ as

$$\hat{\epsilon} \leq e^{-\frac{\delta^2 b}{4}} = \frac{1}{4n^3t^3} \quad . \tag{7}$$

We now focus on the first goal, and show how to control intersection between $\hat{B}$ and A. Applying the Chernoff bound (2) yields

$$\mathsf{Pr}\left[\left|A \cap \hat{B}\right| > (1+\delta)ab/t + \lambda\right] \leq e^{-\frac{t\cdot\lambda^2}{2(1+\delta)ab} + \frac{t^2\cdot\lambda^3}{2(1+\delta)^2a^2b^2}} \quad .$$

We now instantiate this bound for $\lambda = z\sqrt{(1+\delta)ab/t}$, $z = 2\sqrt{\ln(4n^3t^3)}$. Recall that $a, b \geq 7\sqrt{t}\ln(2nt) \geq 4\sqrt{t\ln(4n^3t^3)}$, $\delta \geq 0$, and hence

$$\begin{aligned}\mathsf{Pr}\left[\left|A \cap \hat{B}\right| > (1+\delta)ab/t + \lambda\right] &\leq e^{-\frac{z^2}{2} + \frac{z^3}{2}\sqrt{\frac{t}{(1+\delta)ab}}} \leq e^{-\frac{z^2}{2} + \frac{z^3}{2}\sqrt{\frac{t}{16t\ln(4n^3t^3)}}} \\ &= e^{-\frac{z^2}{2} + \frac{z^2}{4}} = \frac{1}{4n^3t^3} \quad .\end{aligned}$$

Observe that $\frac{ab}{t\sqrt{b}} \leq \sqrt{\frac{ab}{t}}$, because their ratio is equal to $\sqrt{\frac{a}{t}} \leq 1$. Hence we can bound

$$\begin{aligned}(1+\delta)ab/t + \lambda &\leq ab/t + \delta\frac{ab}{t} + z\sqrt{\frac{2ab}{t}} \leq ab/t + 3z\sqrt{\frac{ab}{t}} \\ &\leq ab/t + 6\sqrt{\frac{ab}{t}\ln(2nt)^3} \leq ab/t + 11\sqrt{\frac{ab}{t}\ln(2nt)}\end{aligned}$$

Let $\Delta = 11\sqrt{\frac{ab}{t}\ln(2nt)}$. Using the above bounds we have that

$$\mathsf{Pr}\left[\left|A \cap \hat{B}\right| > ab/t + \Delta\right] \leq \frac{1}{4n^3t^3} \quad . \tag{8}$$

Taking $\eta = ab/t + \Delta$, and combining equations (6), (7), and (8) yields the lemma.

□

The above lemma allows us to control the size of intersection between $\rho_i([a])$ and $\rho_j([b])$ for fixed a, b, i and j. Next we show a bound on waste for *any* two distinct schedules from R and *any* large enough progress.

Theorem 5.1.2. *Let R be a random (n,t)-schedule as constructed previously. Then with probability at least $1-\frac{1}{nt}$,*

$$\forall a,b \textit{ such that } 7\sqrt{t}\ln(2nt) \leq a,b \leq t \ : \quad \mathcal{W}_R(a,b) \leq \frac{ab}{t} + \Delta(a,b) \ ,$$

for $\Delta = 11\sqrt{\frac{ab}{t}\ln(2nt)}$

Proof. Let $\mathcal{B}$ be the (bad) event that the condition from the statement of the theorem fails, and let $\mathcal{B}_{i,j}^{a,b}$ be the event that $|\rho_i([a]) \cap \rho_j([b])| > ab/t + \Delta(a,b)$.Then $\mathcal{B} = \bigvee_{1\leq i,j\leq n,\ i\neq j,\ t_0\leq a,b\leq t} \mathcal{B}_{i,j}^{a,b}$, where $t_0 = 7\sqrt{t}\ln(2nt)$. By Lemma 5.1.1, $\mathsf{Pr}\left[\mathcal{B}_{i,j}^{a,b}\right] \leq \frac{1}{2n^3t^3}$. Hence $\mathsf{Pr}\left[\mathcal{B}\right] \leq n^2t^2 \times \max_{i,j,a,b} \mathsf{Pr}\left[\mathcal{B}_{i,j}^{a,b}\right] \leq \frac{1}{2nt} < \frac{1}{nt}$. □

5.2 Bounds for a group of arbitrary size

In this section we analyze k-waste for the (n,t)-schedule R. The expected value of k-waste can be computed by inclusion-exclusion to be $\sum_{s=2}^{k}(-1)^s\binom{k}{s}\frac{a^s}{t^{s-1}}$. We show that, with high probability, k-waste is close to this expectation. The proof proceeds by considering the martingale which exposes the ith element of all schedules at step i. The theorem then follows by noting that the expected value can change by at most k during a single exposure, and by applying Azuma's inequality.

For convenience of the analysis before we present the proof we define I-waste, denoted by $\mathcal{W}_{R,I}(a_1,\ldots,a_k)$, that measures the number of redundant tasks performed by a given subset I of k processors. Observe that k-waste is equal to maximum I-waste, where I ranges across all subsets of $[n]$ of cardinality k.

An application of the inclusion-exclusion principle shows, that for a set $I \subset [n]$ of cardinality h, the expected value of $\mathcal{W}_{R,I}(a,\ldots,a)$ denoted by $E(k,a)$, is equal to ka minus the expected number of tasks executed by at least one of the processors, and so $E(k,a) = \mathbf{E}[\mathcal{W}_{R,I}(a,\ldots,a)] = ka - t\left(1-\left(1-\frac{a}{t}\right)^k\right) = \sum_{s=2}^{k}(-1)^s\binom{k}{s}\frac{a^s}{t^{s-1}}$. Our goal is to show that R is quite likely to have k-waste close to this expected value.

Theorem 5.2.1. *Let R be a random (n,t)-schedule as constructed previously, and let $a \leq t$. Then with probability at least $1-1/n$,*

$$\mathcal{W}_R(\underbrace{a,\ldots,a}_{k}) \leq \sum_{s=2}^{k}(-1)^s\binom{k}{s}\frac{a^s}{t^{s-1}} + \Delta_{a,k}\ ,$$

where $\Delta_{a,k} = (2k+1)\sqrt{a\ln n}$.

Proof. Let $\emptyset = C_0 \subsetneq C_1 \subsetneq \cdots \subsetneq C_{ka} = I \times [a]$ be a sequence of subsets of $I \times [a]$, and let $\Pi_l : C_l \to [t]$ be the map such that $\Pi_l(i,x) = \rho_i(x)$. The range $\Pi_l(C_l)$ contains the distinct tasks that occur among some l tasks that have been executed by the k processors before the rendezvous. Now, let $Z_l = \mathbf{E}\left[\mathcal{W}_{R,I}(a,\ldots,a)\,|\,\Pi_l\right]$ be the expected value of I-waste when it is known what the values of elements $\rho_i(x)$ for all $(i,x) \in C_l$ are. (See [50] for a discussion of conditional expectation.) Observe now that $Z_0 = \mathbf{E}\left[\mathcal{W}_{R,I}(a,\ldots,a)\right]$, $Z_{ka} = \mathcal{W}_{R,I}(a\ldots,a)$, and an easy argument shows that this martingale has Lipschitz constant 1. Applying Azuma's inequality with $\lambda = \sqrt{2\ln(n\binom{n}{k})}$ guarantees that for any set I of processors of size k

$$\mathsf{Pr}\left[\mathcal{W}_{R,I}(a,\ldots,a) - E(k,a) > (2k+1)\sqrt{a\ln n}\right] \leq \frac{1}{n\binom{n}{k}}\ .$$

Let, for a subset $I \subset [n]$ of cardinality k, $\mathcal{B}_I$ be the bad event that $\mathcal{W}_{R,I}(a,\ldots,a) > E(k,a) + (2k+1)\sqrt{a \ln n}$. Then

$$\mathsf{Pr}\left[\mathcal{W}_{R,I}(a,\ldots,a) > E(k,a) + (2k+1)\sqrt{a\ln n}\right] \leq \mathsf{Pr}\left[\exists I, \mathcal{B}_I\right] \leq \sum_{\substack{I \subset [n] \\ |I|=k}} \mathsf{Pr}\left[\mathcal{B}_I\right] \leq \frac{1}{n}\,.$$

□

In the above, the quantity $\sum_{s=2}^{k}(-1)^s\binom{k}{s}$ is exactly the expected k-waste manifested by k random subsets of size a. Observe that when k is small (i.e., when $k2^{\sqrt{k}}a = o(t)$), this expectation is asymptotically $(1+o(1))\binom{k}{2}a^2$. In particular, this occurs when $k = O(1)$ and $a = o(t)$. Observe, also, that the above theorem begins to be interesting when $\Delta_{a,k} \ll \mathbf{E}\left[\mathcal{W}_R(a,\ldots,a)\right]$. For constant k, this occurs when $a = \omega(t^{2/3}\ln^{1/3} n)$.

Chapter 6

Derandomization via finite geometries

We now present a method for derandomizing the schedules described in the previous chapter, using the designs discussed in Chapter 3. The main idea of the construction is that a processor executes tasks contained in some selected blocks of a design, block by block. Since the intersection of any two different blocks is small, one can hope to ensure gradual buildup of the number of redundantly performed tasks, as processors progress through their schedules. There are two cruxes of the construction. The first one is to ensure that blocks selected for any two processors are "mostly" different, because if many are the same, then they may be conspiratorially arranged so that the same blocks are executed by the processors at the very beginning of processors' work, and consequently high waste may be incurred during initial work, thus waste could significantly diverge from the lower bound. The second crux is to ensure that blocks selected for a processor do not intersect too much among themselves, because if this were the case, then, for a given block, the number of *extra* tasks that the block contains and that have not yet been performed by the processor may decrease rapidly. Conspiratorially, it could be the case that the redundant tasks occur precisely among the extra tasks, and so again waste could diverge substantially from the lower bound.

6.1 Basic construction ($n \leq t$)

In this section we present a special case of our recursive derandomization technique to illustrate the main ideas of the technique. We construct an (n, n)-schedule, by arranging tasks from the sets $L_1, \ldots, L_n$ constructed in Theorem 2.7.2 in a recursive fashion. (Recall that while each pair of these sets intersect by 1 element, each set has cardinality only roughly $\sqrt{n}$.)

Let us demonstrate the second crux. Take any element u of $[n]$. By Theorem 2.7.2, there are exactly $q+1$ sets $L_{t_u^1}, \ldots, L_{t_u^{q+1}}$ that contain u. This fact suggests the following construction of an (n, n)-schedule $P = \langle \pi_1, \ldots, \pi_n \rangle$. Let the processor u execute tasks in the order: first task u, then tasks from $L_{t_u^1}$ arranged in some order except for task u, then tasks from $L_{t_u^2}$ except for task u, and so on. Specifically permutation π_u is chosen so that,

$$\pi_u(1) = u,$$

$$\pi_u([q(i-1)+2, qi+1]) = L_{t_u^i} \setminus \{u\}, \qquad 1 \leq i \leq q+1\ .$$

Note now that since each L_i has cardinality $q+1$, the total number of distinct elements in $\pi_u([n])$ is evidently $(q+1)q+1 = n$. Thus each element of $[n]$

must appear exactly once in each $\pi_u([n])$, and so any π_u is indeed a permutation. Observe that the above selection of blocks of a design ensures that each block adds at least q extra tasks to the schedule.

Let us now present the first crux. We need to ensure that almost all sets selected for two different processors $u \neq u'$ are different. Recall that any two sets selected for the processor u intersect at u. If it were the case that some two distinct sets selected for u were also selected for u', then the intersection of these sets should be equal to u', and so u would be equal to u', a contradiction. This ensures that among all sets selected for u and u' at most two of them are the same.

When we combine the above two observations, one on the number of extra tasks that each set adds, and the other stating that sets are mostly different, we obtain a lemma saying that the growth of waste is gradual.

Lemma 6.1.1. *Let q be a prime power, $n = q^2+q+1$. Let $a = 1+iq$, $b = 1+jq$, $0 \leq i, j \leq q+1$. Then*

$$\mathcal{W}_P(a,b) \leq \begin{cases} 0, & \text{if } \; i+j=0, \\ 1, & \text{if } \; i=0, j\geq 1 \text{ or } i \geq 1, j=0, \\ q+ij, & \text{if } \; i\cdot j \geq 1. \end{cases}$$

Note that when $t = \kappa n$, for some $\kappa \in \mathbb{N}$, the above construction can be trivially applied by placing the t tasks into n chunks of size κ. In this case, of course, when a single overlap occurred in the original construction, this penalty is amplified by κ. We denote the resulting (n,t)-schedule by $V = \langle \chi_1, \ldots, \chi_n \rangle$.

The bound stated in the preceding lemma limits waste for specific values of progress of processors. A possible issue is that waste diverges relatively much in between these values. This, however, is not the case, because the bounds give control over waste for dense enough values of progress. We can show that for fixed $x \geq 1$, when $t = n^x$ is large enough, waste of the (n,t)-schedule V compares favourably with the lower bound when a is at least $t^{1-1/(8x)+\epsilon}$.

Lemma 6.1.2. *Let $n = q^2+q+1$, $t = \kappa n$, $\kappa \in \mathbb{N}$, and let V be (n,t)-schedule as constructed previously. Then*

$$\mathcal{W}_V(a,a) \leq \left(1 + 22 \cdot n^{-1/4}\right) LB(a,a) \ ,$$

for any a such that $\frac{t}{n}(1+q^{7/4}) \leq a \leq t$.

Proof. Take any a such that $\frac{t}{n}(1+q^{7/4}) \leq a \leq t$. It can be written as $a = \frac{t}{n} + iq\frac{t}{n} + k,\ \ 0 \leq i \leq q+1,\ \ 0 \leq k < q\frac{t}{n}$. By Lemma 6.1.1, waste is bounded by $\mathrm{UB} = \mathcal{W}_V(a,a) \leq \frac{t}{n}\left(q + (i+1)^2\right)$. Using Theorem 4.1.2, we further bound waste by

$$\begin{aligned}\mathcal{W}_V(a,a) \geq \mathrm{LB} &= \mathrm{LB}(t/n(1+iq), t/n(1+iq)) \\ &= \frac{n}{t(n-1)}\frac{t^2}{n^2}(1+iq)^2 - \frac{t(1+iq)}{n(n-1)} = \frac{t}{n}\cdot\frac{i^2q+i}{q+1}.\end{aligned}$$

Let us consider the ratio of the upper bound to the lower bound.

$$\frac{\mathrm{UB}}{\mathrm{LB}} \leq \frac{(q+1)(q+i^2+2i+1)}{i(iq+1)}$$

Observe that by assumption $i \geq q^{3/4}$, so $q \leq \frac{i^2}{\sqrt{q}}$. Observe also that $2i + 1 \leq 4q$. From this we get:

$$\begin{aligned} \frac{\text{UB}}{\text{LB}} &\leq \frac{(q+1)(\frac{i^2}{\sqrt{q}} + i^2 + 4q)}{i^2 q} \leq \frac{(q+1)(q^{-1/2} + 1 + 4q^{-1/2})i^2}{i^2 q} \\ &\leq \frac{q + 5q^{1/2} + 1 + 5q^{-1/2}}{q}. \end{aligned}$$

Hence UB/LB $\leq 1 + 5q^{-1/2} + q^{-1} + 5q^{-3/2}$, and the result follows. □

We mention that the construction of P can be done on-line. For each schedule the first element can be calculated in $O(1)$ time. For the remaining $q(q+1)$ elements, at the beginning of every sequence of q elements, we need to invert at most two elements in GF(q). This can be done using the extended Euclidean algorithm, and can be done in $O(\log n)$ time when q is prime. Other elements of the schedule can be found in $O(1)$ time.

6.2 Generic recursive construction ($n = t^u$)

This section illustrates a more general technique for producing (n,t)-schedules recursively, from set systems with controlled intersection properties. This technique offers weaker bound on waste than the one presented earlier, because when constructing a schedule, the selected sets may add fewer and fewer extra tasks to the schedule.

The construction is based on polynomials over finite fields and has two interesting features. Firstly, it yields a (t^u, t)-schedule when the number of processors $n = t^u$ is polynomially larger than the number of tasks t. Secondly, processor do not need to know the value of n in order to construct their schedules in isolation. In order to get good bounds on waste, it is enough to ensure that processors have identifiers that are distinct and are compactly packed among the smallest natural numbers.

Assume that there is a prime power q, so that $t = q^2$, and that $n = q^d$, for some fixed $d \geq 2$. We construct sets $\kappa_1^{(d)}, \ldots, \kappa_{q^d}^{(d)}$, where each $\kappa_i^{(d)}$ has cardinality q. Let $f_1, \ldots, f_{q^d}$ be an enumeration of all polynomials of degree at most $d-1$ over $\mathrm{GF}(q) = \{e_1, \ldots, e_q\}$, and canonically associate these polynomials with $[n]$, the processor identifiers. By canonically associating the set $[t]$ of tasks with $\mathrm{GF}(q)^2$, we define the set $\kappa_i^{(d)}$ to be a "plot" of the polynomial; i.e., $\kappa_i^{(d)} = \{(e_1, f_i(e_1)), \ldots, (e_q, f_i(e_q))\}$. Recalling that two distinct polynomials of degree $d-1$ can agree at no more than $d-1$ points, yields the following lemma (this property is also used by Reed-Solomon codes).

Lemma 6.2.1. *Let q be a prime power, $d \geq 2$, $n = q^d$, and let $\kappa_1^{(d)}, \ldots, \kappa_n^{(d)}$ be subsets of $[q^2]$ of cardinality q as constructed above. For any $i \neq j$, $|\kappa_i^{(d)} \cap \kappa_j^{(d)}| \leq d-1$.*

The construction of a (q^d, q^2)-schedule $J = \langle \iota_1, \ldots, \iota_{q^d} \rangle$ uses the sets $\kappa_1^{(2)}, \ldots, \kappa_{q^2}^{(2)}$, to recursively "elongate" sets $\kappa_1^{(d)}, \ldots, \kappa_{q^d}^{(d)}$. The t-schedule ι_i is constructed as follows. Let $\kappa_i^{(d)} = \langle y_1, \ldots, y_q \rangle$. We build a list K inductively. We start with an empty list K, and append a set $\kappa_{y_1}^{(2)}$ (called segment number 1), in arbitrary order, to K. Suppose that we have appended segment number $i < q$, to K. We take the set $\kappa^{(2)}_{y_{i+1}}$, remove from it all elements that already appear in K, and append the result (called segment number $i+1$), in arbitrary order, to K. We proceed in this fashion until segment number q has been appended. Then we append all elements that do not appear in K, at the end of K, in arbitrary order. The resulting list defines the permutation ι_i; i.e., x is the j-th element on the list, if and only if $\iota_i(j) = x$. In the following theorem we show that waste for J is close to the lower bound, for lengths linear in t.

Theorem 6.2.2. *Let q be a prime power, $t = q^2$, $n = q^d$, $d \geq 2$, and let $a, b \leq \frac{t}{4}$. Then the waste for the (n,t)-schedule J is bounded by:*

$$\mathcal{W}_J(a,b) \leq (d-1)\sqrt{t} + 4\frac{ab}{t} \ .$$

Proof. Take a schedule ι_i as constructed in the paragraph preceding the statement of the theorem, and consider element number a in the schedule. Since each two sets $\kappa^{(2)}_i$ and $\kappa^{(2)}_j$, $i \neq j$, overlap by at most 1, and any $\kappa^{(2)}_i$ contains q elements, each of the first $\lceil \frac{q}{2} \rceil$ segments added in the inductive process adds at least $\lceil \frac{q}{2} \rceil$ new elements to K. Hence these segments contain at least $\lceil \frac{q}{2} \rceil^2$ elements. Since $a \leq \frac{q^2}{4} \leq \lceil \frac{q}{2} \rceil^2$, element number a must belong to one of the first $\lceil \frac{q}{2} \rceil$ inductively added segments, and the segment's number is at most $\frac{a}{\lceil \frac{q}{2} \rceil} \leq \frac{2a}{q}$.

Consider any two schedules ι_i and ι_j, and elements number a and b from the schedules respectively. We estimate the overlap between $\iota_i([a])$ and $\iota_j([b])$. Consider the pairs of segments (I, J), where I is a segment included in $\iota_i([a])$ and J is a segment included in $\iota_j([b])$. By Lemma 6.2.1, only $d-1$ pairs may have segments where $I \subseteq J$ or $J \subseteq I$. This results in at most $(d-1)q$ overlap for the $d-1$ pairs. For the remaining at most $\frac{4}{q^2}ab-(d-1)$ pairs, the overlap is at most 1. □

The complexity of the construction of any of the n schedules is $O(t)$ time and $O(t)$ space (recall that d is a fixed constant).

6.3 Generalization of basic construction ($n = t^{2-x}$)

We turn back to the study of schedules obtained from designs. We have seen in Section 6.1, that the regular structure of blocks of certain designs enabled us to control waste quite well. This section generalizes this construction, and allows to schedule tasks for $n = t^{2-x}$ processors, for some $0 < x < 1$. In the previous construction, both a schedule and a task were associated with subspaces of dimension 1 of the vector space of dimension 3 used to construct a design. In the construction presented in this section, a task is still associated with a subspace of dimension 1, but a schedule is associated with a subspace of dimension $m-2$, and as m grows, n approaches t^2. The appropriate arrangement of blocks of specific designs allows us to construct a family of schedules with waste that compares favorably with the lower bound.

For any $m \geq 3$, and q prime power, we take the vector space $GF(q)^m$, $n = \Delta(q,m,0,m-2)$, and $t = \Delta(q,m,0,m-1)$, and construct (n,t)-schedule $N^m = \langle \nu_1, \ldots, \nu_n \rangle$ as follows. For any u, by Theorem 2.7.1, there are exactly $q+1$ distinct sets $L^m_{t^1_u}, \ldots, L^m_{t^{q+1}_u}$ that contain u. Recall that A_u is a subspace of dimension $m-2$ that is contained in any of the subspaces $B_{t^1_u}, \ldots, B_{t^{q+1}_u}$; that A_u contains $e = \Delta(q, m-2, 0, 1)$ subspaces of dimension 1; that any B_i contains $r = \Delta(q, m-1, 0, 1)$ subspaces of dimension 1; and that $\ell_1, \ldots, \ell_t$ are all the subspaces of $GF(q)^m$ of dimension 1. Permutation ν_u is picked so that

$$\nu_u([e]) = \{j \mid \ell_j \subseteq A_u\} \ ,$$

$$\nu_u([e + (r-e)i + 1, e + (r-e)(i+1)]) = \{j \mid \ell_j \subseteq B_{t^i_u} \setminus A_u\}, \qquad 0 \le i \le q \ .$$

By the same argument as in Section 6.1, we know that ν_u is indeed a permutation, and that for any two $u \neq u'$, among the sets $L^m_{t^1_u}, \ldots, L^m_{t^{q+1}_u}$ and $L^m_{t^1_{u'}}, \ldots, L^m_{t^{q+1}_{u'}}$, at most two of them are the same. This yields the following bound on waste that generalizes Lemma 6.1.1.

Lemma 6.3.1. *Let $m \ge 3$, q prime power, and N^m be the (n,t)-schedule constructed in the preceding paragraphs. Let $a = e + (r-e)i$, $b = e + (r-e)j$, $1 \le i, j \le q+1$. Then*

$$\mathcal{W}_{N^m}(a,b) \le \frac{q^{m-1}-1}{q-1} + (ij-1)\frac{q^{m-2}-1}{q-1} \ .$$

The lemma bounds waste for sufficiently dense values of a, so that we can show that for any fixed $m \ge 3$, large enough t, when a is at least $t^{1-2/(m-1)+\epsilon}$, waste compares favorably with the lower bound.

Theorem 6.3.2. *For any a such that $\frac{q^{m-2}-1}{q-1} < a \le t$,*

$$\mathcal{W}_{N^m}(a,a) \le \left(1 + 64 \cdot t^{-\frac{1}{3(m-1)}}\right) LB(a,a).$$

Proof. Let us take a that satisfies the condition from the statement of the theorem. Then there exists an i, such that $\underline{a} := \frac{q^{m-2}-1}{q-1} + (i-1)q^{m-2} < a \leq \frac{q^{m-2}-1}{q-1} + iq^{m-2} =: \overline{a}$. The waste incurred by any two processors that have executed a tasks each, must be at least $LB = LB(\underline{a}, \underline{a})$. Since $m \geq 3$ and $i \geq 1$, we can bound LB from below by $LB \geq i^2 q^{m-3}\frac{1-q^{-2}}{1+2q^{-3}}$. The waste of any two processors that have executed a tasks each, can be at most $UB = \mathcal{W}_{N^m}(\overline{a}, \overline{a})$, and we can bound the waste using Lemma 6.3.1 by $UB \leq \frac{q^{m-1}-1}{q-1} + (i^2-1)\frac{q^{m-2}-1}{q-1}$. Since $i \geq q^{-2/3}$, we can further bound UB from above by $UB \leq i^2 \cdot q^{m-3}\frac{1+q^{-1/3}}{1-q^{-1}}$. Therefore, we can bound the ratio of UB to LB by $\frac{UB}{LB} \leq \frac{(1+q^{-1/3})(1+2q^{-3})}{(1-q^{-1})(1-q^{-2})}$. Since $q^{-1} \leq 1/2$, we have that $1/(1-q^{-1}) \leq 1 + 2q^{-1}$, and $1/(1-q^{-2}) \leq 1 + 2q^{-2}$. In addition, since $t \leq 2q^{m-1}$, $q^{-1/3} \leq 2t^{-1/(3(m-1))}$. Hence the result follows. □

6.3.1 Efficient incremental construction of schedules

In this section we show how to efficiently construct schedules from the (n,t)-schedule $N^m = \langle \nu_1, \ldots, \nu_n \rangle$. An important feature of our construction, for the applications that we consider, is that processors pay for the construction in small chunks as they build their schedules, and each processor can perform the construction without coordination with other processors.

Recall that according to the method described in the previous section, a schedule ν_i is constructed by taking the $(m-2)$-dimensional subspace A_i, and by considering all $(m-1)$-dimensional subspaces $B_1, \ldots, B_{q+1}$ that contain A_i. Then elements of the schedule ν_i were obtained by listing indices of 1-dimensional subspaces contained in each B_j, $1 \leq j \leq q+1$, excluding repeated indices. Using this simple approach would mean the we must use $\Theta(m^2)$ memory to describe $(m-2)$-dimensional subspaces, this would also influence the construction time. In order to avoid this high cost, we use a dual approach that is more efficient.

Let $\langle \cdot, \cdot \rangle$ be the normal inner product given by $\langle (x_1, \ldots, x_m), (y_1, \ldots, y_m) \rangle = \sum_j x_j y_j$, where additions and multiplications are performed in $GF(q)$. By Lemma 2.6.2, there is a unique 2-dimensional subspace P orthogonal to A_i. By the same lemma, an $(m-1)$-dimensional subspace K contains $D = A_i$ if and only if the 1-dimensional subspace $K^\perp$ is contained in the 2-dimensional subspace $D^\perp$. Similarly, a 1-dimensional subspace W is contained in a $(m-1)$-dimensional subspace K if and only if $K^\perp \subseteq W^\perp$. But $K^\perp \subseteq W^\perp$ is equivalent to $K^\perp$ being orthogonal to W.

These observations lead to the following construction of ν_i. Let $P_1, \ldots, P_n$ be all the subspaces of $GF(q)^m$ of dimension 2. We take a 2-dimensional subspace $P_i = \text{span}(\alpha, \beta)$, and all distinct 1-dimensional subspaces $\text{span}(\gamma_1), \ldots, \text{span}(\gamma_{q+1})$ contained in P_i. The first segment of ν_i is obtained in two phases. We first take the e indices of all 1-dimensional subspaces $\text{span}(\gamma)$ orthogonal to P_i, i.e., $\langle \gamma, \alpha \rangle = 0$ and $\langle \gamma, \beta \rangle = 0$. Then we take the $r - e$ indices of all 1-dimensional subspaces $\text{span}(\gamma)$ orthogonal to $\text{span}(\gamma_1)$ but not orthogonal to P_i, i.e., $\langle \gamma, \gamma_1 \rangle = 0$, but $\langle \gamma, \alpha \rangle \neq 0$ or $\langle \gamma, \beta \rangle \neq 0$. Next we append elements from the remaining segments.

A segment number j, $2 \leq j \leq k$, is obtained by taking indices of all 1-dimensional subspaces span(γ) such that $\langle \gamma, \gamma_j \rangle = 0$, but $\langle \gamma, \alpha \rangle \neq 0$ or $\langle \gamma, \beta \rangle \neq 0$.

In order to realize this construction, we must be able to efficiently carry out several procedures. We have associated the tasks with 1-dimensional subspaces and the schedules with 2-dimensional subspaces. We require a representation of these spaces so that we can

(i) efficiently compute a distinct 2-dimensional subspace associated with an index $i \in [n]$,

(ii) efficiently enumerate all distinct 1-dimensional subspaces contained in a particular 2-dimensional subspace,

(iii) efficiently enumerate all distinct 1-dimensional subspaces orthogonal to a given 1- or 2-dimensional subspace, and

(iv) efficiently compute a distinct index in $[t]$ associated with a given 1-dimensional subspace.

Such a representation, called Row Reduced Echelon Form, will be discuss in the remainder of this section. It gives rise to the algorithm described in Figure 1. The algorithm constructs any schedule from N^m efficiently, incrementally, and without coordination with other processors. An important feature of the algorithm is that the cost of construction is almost evenly distributed during construction. Each of the $q + 1$ segments of at most t/q consecutive elements of a schedule costs $O(mt/q)$ time to construct, except for the first segment which costs $O(mt/q + q \log q)$). Thus, processors pay in small chunks while constructing schedules. Each element can be found using $O(m+q)$ space. The performance of the algorithm is described by the following theorem:

Theorem 6.3.3. *Let q be prime and $m \geq 3$. The t elements of each ν_i in N^m ($1 \leq i \leq n$) can be calculated incrementally. The first $\frac{q^{m-1}-1}{q-1}$ elements are determined in time $O(mq^{m-2} + q \log q)$. All elements from any of the remaining q segments of consecutive q^{m-2} elements are determined in time $O(mq^{m-2})$. Each element can be found using $O(m+q)$ space, and the entire construction can be done in $O(t \cdot m)$ time and $O(m+q)$ space.*

First we discuss a representation of the distinct 2-dimensional subspaces of V called Row Reduced Echelon Form. For each pair $1 \leq i_1 < i_2 \leq m$, and a vector $x = (x_1, \ldots, x_u) \in GF(q)^u$, $u = 2(m-2) + 3 - (i_1 + i_2)$, we select the pair $\alpha_1, \alpha_2 \in GF(q)^m$, so that for each $j = 1, 2$,

$$\alpha_j = (\overbrace{0, \ldots, 0}^{i_j - 1}, 1, x^j_{i_j+1}, \ldots, x^j_m),$$

where

- $x^1_h = x_{h-i_1}$, $i_1 + 1 \leq h \leq i_2 - 1$,

- $x^1_{i_2} = 0$,

- $x^1_h = x_{h-i_1-1}$, $i_2 + 1 \leq h \leq m$,

- $x^2_h = x_{h-i_2-1+m-i_1}$, $i_2 + 1 \leq h \leq m$.

```
int next( in-out $\alpha, \beta, x \in (\mathbb{Z}_q)^m$ , $y \in (\mathbb{Z}_q)^{m-1}$,
        $z \in (\mathbb{Z}_q)^{m-2}$, $w \in \mathbb{Z}_q$, $case \in [3]$)
  while $true$ do
      if $case = 3$ then
        advance( $y, m-1, carry$ )
        if $carry = true$ then
          $x$ =normalize( $\alpha + w\beta$ )
          $w = w + 1$
          $y = (1, 0, \ldots, 0)$
        $v$ = normalize( $\lambda(x, y)$ )
      if $case = 2$ then
        $w = 0$ ;  $x = \beta$
        $y = (1, 0, \ldots, 0)$
        $v$ = normalize( $\lambda(x, y)$ )
        $case = 3$
      if $case = 1$ then
        $v$ =normalize( $\mu(\alpha, \beta, z)$ )
        advance( $z, m-2, carry$ )
        if $carry = true$ then
          $case = 2$
        return( getIndexFrom1dim($v$) )
      if $\langle v, \alpha \rangle \neq 0 \vee \langle v, \beta \rangle \neq 0$ then
        return( getIndexFrom1dim($v$) )
```

```
advance( in-out $z \in (\mathbb{Z}_q)^a$;
            in $a \in [m]$;
            out $carry \in \{true, false\}$ )
    for $j = 1$ to $a$
            if $z[j] \neq 0$ break
    for $b = j + 1$ to $a$
            $z[b] = z[b] + 1$
            if $z[b] \neq 0$ break
    $carry = false$
    if $j = a$ then
            $carry = true$
    else if $b = a \wedge z[b] = 0$ then
            $z[j] = 0$
            $z[j+1] = 1$
            for $b = j + 2$ to $a$
              $z[b] = 0$
initW( in $i \in [n]$; out $\alpha, \beta \in (\mathbb{Z}_q)^m$;
            out $z \in (\mathbb{Z}_q)^{m-2}$;
            out $case \in [3]$)
    $(\alpha, \beta)$ = get2dimFromIndex( $i$ )
    $z = (1, 0, \ldots, 0)$
    $h = 1$
    $case = 1$
```

Figure 1: Algorithm for incremental construction of schedule ν_i from the (n, t)-schedule $N^m = \langle \nu_1, \ldots, \nu_n \rangle$.

(Note that these α_1, α_2 depend on i_1, i_2, and x.) Hence for each $j = 1, 2$, there are $i_j - 1 + 2 - j$ coordinates of α_j that are set to 0, one other coordinate is set to 1, and the remaining $m - (i_j + 2 - j)$ are copies of distinct coordinates of vector x. We let $S(i_1, i_2, x)$ be the list of vectors α_1, α_2 as defined above. Observe that for fixed i_1 and i_2, vector x can be chosen in $q^{2(m-2)+3-(i_1+i_2)}$ unique ways. It is easy to see that these vectors $S(i_1, i_2, x)$ are linearly independent. Therefore, span$(S(i_1, i_2, x))$ is a 2-dimensional subspace of $GF(q)^m$. It is also easy to observe that any 2-dimensional subspace $D = \text{span}(\beta_1, \beta_2)$ is equal to span$(S(i_1, i_2, x))$, for some $1 \leq i_1 < i_2 \leq m$, and $x \in GF(q)^u$, $u = 2(m-2) + 3 - (i_1 + i_2)$.

For convenience, we canonically associate elements of $GF(q)^u$ with elements of $[q^u]$, and we treat them interchangeably in the remaining part of the section. The following lemma asserts that the function S defines distinct 2-dimensional subspaces of $GF(q)^m$

Lemma 6.3.4. *Let $1 \leq i_1 < i_2 \leq m$, $1 \leq i'_1 < i'_2 \leq m$, $x \in [q^{2(m-2)+3-(i_1+i_2)}]$, and $x' \in [q^{2(m-2)+3-(i'_1+i'_2)}]$. If $(i_1, i_2, x) \neq (i'_1, i'_2, x')$ then span$(S(i_1, i_2, x)) \neq$ span$(S(i'_1, i'_2, x'))$.*

Proof. Let us take $(i_1, i_2, x) \neq (i'_1, i'_2, x')$. We consider two cases, one of which must occur.

Case 1: Suppose that $(i_1, i_2) \neq (i'_1, i'_2)$. Since the indices are ordered, the cardinality of the set $\{i_1, i_2, i'_1, i'_2\}$ is at least 3, and so is the dimension of $\mathrm{span}(S(i_1, i_2, x)) + \mathrm{span}(S(i'_1, i'_2, x'))$. Therefore, the subspaces $\mathrm{span}(S(i_1, i_2, x))$ and $\mathrm{span}(S(i'_1, i'_2, x'))$ are distinct.

Case 2: Suppose that $(i_1, i_2) = (i'_1, i'_2)$, but $x \neq x'$. Then there is $1 \leq h \leq 2$, such that the vector α_h is not equal to vector α'_h. But then $\alpha_h - \alpha'_h$ is a nonzero vector with the first nonzero coordinate that is not in the set $\{i_1, i_2\}$, and so it is linearly independent from vectors α_1, α_2. Hence the subspaces $\mathrm{span}(S(i_1, i_2, x))$ and $\mathrm{span}(S(i'_1, i'_2, x'))$ are distinct. □

Now we show that we can quickly compute a distinct 2-dimensional subspace given an index i from the set $[n] = [\Delta(q, m, 0, 2)]$. In the following lemma we introduce a linear ordering on the two dimensional subspaces of $GF(q)^m$.

Lemma 6.3.5. *For any $i \in [n]$, $n = \Delta(q, m, 0, 2)$, vectors that span a distinct 2-dimensional subspace P_i of $GF(q)^m$, q prime, can be found in $O(m)$ time and $O(m)$ space.*

Proof. We introduce a linear order on the set of 2-dimensional subspaces of $GF(q)^m$ in a recursive fashion.

We order the 2-dimensional subspaces in groups from group 1 to $m-1$ as follows. Group number j consists all subspaces for $i_1 = j$, $i_2 > i_1$, and $x \in [q^{2(m-2)+3-(i_1+i_2)}]$. We can count the number s_j of subspaces in a group j using Lemma 6.3.4, and observing that the sum forms a geometric progression:

$$s_j = q^{2(m-2)+3} \sum_{x=3+2(j-1)}^{m+j} q^{-x} = q^{2(m-j-1)} \frac{q^{-m+j} - 1}{q^{-1} - 1}.$$

We can count the number a_j of subspaces in groups 1 through j as:

$$a_j = s_1 + \ldots + s_j = \frac{(1+q)(q^{m-j-1} - q^{m-1}) - q^{2m-2j-1} + q^{2m-1}}{(q^2-1)(q-1)}.$$

Hence, for a given i, we can find the group j to which i belongs, and the offset $\hat{j}$ within the group in $O(\log^2 m)$ time and $O(1)$ space, using binary search.

We order the subspaces that belong to a group $j < m$ in subgroups, from subgroup $j+1$ to subgroup m, as follows. Subgroup number h consists of all 2-dimensional subspaces for $i_1 = j$, $i_2 = h$, and $x \in [q^{2(m-2)+3-(i_1+i_2)}]$. We can count the number b_g of subspaces in subgroups $j+1$ through g as:

$$b_g = q^{2(m-2)+3} \sum_{x=2j+1}^{j+g} q^{-x} = \frac{q^{2m-j-g-1} - q^{2(m-j-1)+1}}{1-q}.$$

Hence, for a given j and $\hat{j}$, we can find the subgroup g to which $\hat{j}$ belongs, and the offset $\hat{g}$ within the subgroup, in $O(\log^2 m)$ time and $O(1)$ space, using binary search.

We order the subspaces that belong to a subgroup as follows. Observe that $\hat{g} \in [q^{2(m-2)+3-(i_1+i_2)}]$, $i_1 = j$, $i_2 = g$. We represent $\hat{g}$ as a string of length $2(m-2)+3-(i_1+i_2)$ with symbols from $GF(q)$ using "q-ary" expansion. Since q is prime, this can be done in $O(m)$ time and $O(m)$ space. From this string we obtain α_1 and α_2 by placing zeros, ones, and the elements of the string at the corresponding coordinates of the two vectors, according to the definition of function S. This can be done in $O(m)$ time and $O(m)$ space.

Finally, observe that the order introduced above is linear. □

In the following lemma, we show how to find all distinct 1-dimensional subspaces contained in a given 2-dimensional subspace.

Lemma 6.3.6. *Let $W = span(\alpha, \beta)$ be a 2-dimensional subspace of a vector space $GF(q)^m$. Then the $q+1$ spaces $span(\beta)$, $span(\alpha + x\beta)$, $x \in GF(q)$, are distinct, and are all 1-dimensional subspaces of W.*

In our construction of (n, t)-schedule N^m we need to find all distinct 1-dimensional subspaces that are orthogonal to a given 2-dimensional subspace, or 1-dimensional subspace of $GF(q)^m$. We now discuss the construction and its complexity.

Let span(x, y) be a 2-dimensional subspace of a vector space $GF(q)^m$ with

$$x = (0, \ldots, 0, 1, x_{i+1}, \ldots, x_m),$$

$$y = (0, \ldots, 0, 1, y_{j+1}, \ldots, y_m),$$

where $i < j \leq m$ and $x_j = 0$. Let $Z \subset GF(q)^{m-1}$ be the set

$$\{(0, \ldots, 0, 1, z_{h+1}, \ldots, z_{m-2}) \mid h \leq m-2\}.$$

For $z \in Z$ we define $\mu(x, y, z) \in GF(q)^m$ to be

$$\mu(x, y, z) = (z_1, \ldots, z_{i-1}, -\xi_{ij}^{(1)} - \upsilon_{jm}^{(2)} x_j + \xi_{jm}^{(2)}, z_i, \ldots, z_{j-2}, -\upsilon_{jm}^{(2)}, z_{j-1}, \ldots, z_{m-2}),$$

where

$$\begin{aligned} \xi_{ij}^{(1)} &= \sum_{a=i}^{j-2} z_a x_{a+1}, \\ \xi_{jm}^{(2)} &= \sum_{a=j-1}^{m-2} z_a x_{a+2}, \text{ and} \\ \upsilon_{jm}^{(2)} &= \sum_{a=j-1}^{m-2} z_a y_{a+2}. \end{aligned}$$

Lemma 6.3.7. *The set $P = \{span(\mu(x, y, z)) \mid z \in Z\}$ contains all distinct 1-dimensional subspaces of $GF(q)^m$ that are orthogonal to span(x, y).*

Corollary 6.3.8. *For $GF(q)^m$ and q prime, the vector $\mu(x, y, z)$ can be found in $O(m)$ time and $O(m)$ space.*

Let span$(\hat{x})$ be a 1-dimensional subspace of a vector space $GF(q)^m$, $\hat{x} = (0, \ldots, 0, 1, x_{i+1}, \ldots, x_m)$, $1 \leq i \leq m$. Let $Y \subset \mathrm{GF}(q)^{m-1}$ be the set

$$\{(0, \ldots, 0, 1, y_{j+1}, \ldots, y_{m-1}) \mid 1 \leq j \leq m-1\}.$$

For $\hat{y} \in Y$, we define $\lambda(\hat{x}, \hat{y}) \in GF(q)^m$ to be

$$\lambda(\hat{x}, \hat{y}) = (y_1, \ldots, y_{i-1}, -x_{i+1}y_i - \ldots - x_m y_{m-1}, y_i, \ldots, y_{m-1}) \ .$$

Lemma 6.3.9. *The set $P = \{span(\lambda(\hat{x}, \hat{y})) \mid \hat{y} \in Y\}$ contains all distinct 1-dimensional subspaces of $GF(q)^m$ that are orthogonal to span($\hat{x}$).*

Corollary 6.3.10. *For $GF(q)^m$ and q prime, the vector $\lambda(\hat{x}, \hat{y})$ can be found in $O(m)$ time and $O(m)$ space.*

Proof of Theorem 6.3.3. For convenience of the analysis let us cluster the elements of a schedule ν_i in $k + 1 = q + 2$ groups. The group number zero consists of the first $e = \frac{q^{m-2}-1}{q-1}$ elements. The remaining k groups consist of $r - e = q^{m-2}$ consecutive elements. Each time an element number 1 through $q-1$ of a schedule is constructed, we invert a consecutive element of $\mathbb{Z}_q$, in $O(\log q)$ time and $O(1)$ space for each element, and store the result in a table of size $O(q)$.

First we judge the complexity of determining elements from group number zero. Finding the very first element of this group can be done by obtaining the two dimensional subspace span(α, β) corresponding to the index i. This, by Lemma 6.3.5, can be done in $O(m)$ time and $O(m)$ space. Then we need to find a vector $\mu(\alpha, \beta, z)$ that spans a 1-dimensional subspace orthogonal to span(α, β), which, by Corollary 6.3.8, takes $O(m)$ time and $O(m)$ space. Next we need to compute a distinct index $j \in [t]$ associated with the space spanned by the vector. Since q is prime, this can be done in $O(m + \log q)$ time and $O(m)$ space (as it may require normalizing i.e., multiplying a vector by the inverse of its first nonzero coordinate). In order to find the elements 2 through e of the group we need to: find vectors $\mu(\alpha, \beta, z)$ for the remaining admissible values of z as stated in Lemma 6.3.7, normalize the resulting vectors, and compute distinct indices associated with them. Since we need to invert at most $q-1$ elements of $\mathbb{Z}_q$, the total complexity of determining elements from group number zero is $O(mq^{m-3} + q \log q)$ time. Each element can be found in $O(m)$ space.

Let us consider the complexity of finding elements from a group number $j > 0$. According to the description that appears earlier in the section, we do it by finding all 1-dimensional subspaces orthogonal to a given 1-dimensional subspace $\text{span}(\gamma_j)$, but not orthogonal to a given 2-dimensional subspace $\text{span}(\alpha, \beta)$. We can find γ_j in $O(m)$ time and space using Lemma 6.3.6. The vector γ_j may need to be normalized in $O(m)$ space and $O(m + \log q)$ time, when inverses of elements from $\mathbb{Z}_q$ are not available, but only in $O(m)$ time, when they are available. There are $r = \frac{q^{m-1}-1}{q-1}$ 1-dimensional subspaces orthogonal to a given 1-dimensional subspace, and we can find each, by Corollary 6.3.10, in $O(mq^{m-2})$ time and $O(m)$ space. However, only $r - e = q^{m-2}$ of them are not orthogonal to $\text{span}(\alpha, \beta)$, and we can test each in $O(m)$ time and $O(m)$ space. Each one dimensional subspace that passes the test needs to be converted to its unique index in $[t]$ in $O(m)$ time and space, and this may require prior normalization, in $O(m)$ space and $O(m + \log q)$ time, when inverses of elements from $\mathbb{Z}_q$ are not available, but only $O(m)$ time, when they are available. Thus, the complexity of finding all the elements from the group is $O(mq^{m-2} + q \log q)$ time, when inverses are not available, and $O(mq^{m-2})$ time, when they are. Each element can be found in $O(m)$ space.

Finally observe that the first segment contains group number zero and one, and so the total time sufficient to construct its elements is $O(mq^{m-2} + q \log q)$. Since the first segment contains at least $q-1$ elements, after they are constructed, the inverses are available in the table, and so constructing any of the remaining segments takes $O(mq^{m-1})$ time. □

If space is of concern, we can use a different version of the algorithm. We find the inverse of an element of $\mathbb{Z}_q$ every time it is needed, in $O(\log q)$ time and $O(1)$ space. For such an algorithm, the total construction time is $O(t(m + \log q))$ and $O(m)$ space.

6.4 Controlling waste for short prefixes

The deterministic schedules developed so far have the property that when progress is small waste may grow linearly (this is expressed by the existence of additive constants: q in Lemma 6.1.1, $(d-1)\sqrt{t}$ in Theorem 6.2.2, and $\frac{q^{m-1}-1}{q-1}$ in Lemma 6.3.1). This property may be undesired in some case. If $t \gg n$ then processors may waste relatively much work beyond necessity, when they rendezvous early in their computation. Moreover, redundant work gets amplified, as more processors rendezvous, in which case an undesired constant fraction of their total work may be wasted. This section deals with the issues by presenting several constructions for improving prefix waste. We note that our objective can be trivially attained at the expense of later higher waste just by each processor selecting a distinct block of a design to schedule its initial work. This, however, results in possibly increasing by two the number of blocks that some two schedules share, and can cause about half of the tasks to be wasted, when some two processors rendezvous after performing tasks from three blocks each. Our constructions presented next are more subtle, and have better control over the growth of waste.

6.4.1 A method when $t \geq n$

One disadvantage of the (n, n)-schedule P is that the sets of $q + 1$ tasks executed first by some two processors may be the same, and so $q + 1$ waste may be incurred when a prefix of length $q + 1$ is executed. To postpone this increase, one would like to rearrange the sets $L_{t_u^1}, \ldots, L_{t_u^{q+1}}$ used to construct π_u, so that the sets of the first $q + 1$ tasks are different for different schedules of P. This can be accomplished by finding a permutation $\alpha : [n] \to [n]$ such that u is contained in the set $L_{\alpha(u)}$.

Consider the bipartite graph (U, V, E), where $U = V = [n]$ and $n = q^2 + q + 1$. The set U can be placed in one-to-one correspondence with the one dimensional subspaces of $\mathrm{GF}(q)^3$, and V with the two dimensional subspaces. An edge is placed between $u \in U$ and $v \in V$, if u is a subset of v. Based on Theorem 2.7.2, the graph is $(q + 1)$-regular. By Hall's theorem (see e.g., [72]), there is always a perfect matching in a d-regular bipartite graph, and note that such a matching yields a permutation α with the desired properties. In particular, if the edge (u, v) appears in the perfect matching, then we put $\alpha(u) = v$. This matching can be found using the Hopcroft-Karp algorithm [76] that runs in time $O(\sqrt{|U| + |V|} \cdot |E|) = O(n^2)$.

We use α to construct (n, n)-schedule $G = \langle \gamma_1, \dots, \gamma_n \rangle$, such that the sets $\gamma_i([q+1])$ intersect by exactly one. For any $1 \le u \le n$, let $L_{\alpha(u)}, L_{t_u^2}, \dots, L_{t_u^{q+1}}$ be the $q+1$ sets that contain u. Permutation γ_u is chosen so that

$$\gamma_u(1) = u,$$

$$\gamma_u([q+1]) = L_{\alpha(u)},$$

$$\gamma_u([2 + (i-1)q, 1 + iq]) = L_{t_u^i} \setminus \{u\}, \qquad 2 \le i \le q+1$$

Theorem 6.4.1. *Let q be a prime power, and $n = q^2 + q + 1$. Let $a = 1 + iq$, $b = 1 + jq$, $0 \le i, j \le q+1$. Then:*

$$\mathcal{W}_G(a, b) \le \begin{cases} 0, & \text{if } \; i + j = 0, \\ 1, & \text{if } \; i = 0, j \ge 1 \text{ or } i \ge 1, j = 0, \\ 1, & \text{if } \; i \cdot j = 1, \\ q + ij, & \text{if } \; i \cdot j > 1. \end{cases}$$

Proof. Take any two distinct t-schedules γ_u and $\gamma_{u'}$, $u \ne u'$. When $i = j = 1$, observe that by the construction of G, $\alpha(u) \ne \alpha(u')$, and so $L_{\alpha(u)}$ intersect by one with $L_{\alpha(u')}$. The other cases follow the proof of Lemma 6.1.1. □

Observe that this construction is time-optimal, as it produces n^2 elements and runs in $O(n^2)$ time. However, the algorithm requires $O(n^2)$ time to construct even a single permutation. (This construction can be further refined, and by $(q+1)$ coloring of the graph, we can produce a latin square [42] rows of which yield an (n, n)-schedule with the same bound on waste as in Theorem 6.4.1. The construction of latin square can be done in $O(n^2 \log n)$ time using the algorithm of Cole and Hopcroft [36].)

We could use the (n, n)-schedule G to produce an (n, t)- schedule, for $t = \kappa n$, in the same fashion as we produced V i.e., by grouping tasks into t/n size chunks. However, this is not a good idea if we want to control waste when $t \gg n$ and progress is small. We know that first segments of schedules from G overlap by 1. Hence in the resulting (n, t)-schedule, the set of the first $(q+1) \cdot \frac{t}{n}$ elements of any t-schedule would intersect by $\frac{t}{n}$ elements with the set of the first $(q+1) \cdot \frac{t}{n}$ elements of some other t-schedule. In the worst case, an undesired linear growth of waste could be incurred when progress is between t/n and $2t/n$. One way to alleviate the problem is to compute tasks from the chunks in a different order. Observe that when at most d elements in a segment of G are wasted, then $q - d$ are not wasted. This suggests the following sequencing of tasks within a segment. We compute the first tasks from all the chunks, then the second tasks, and so on, up to t/n-th tasks from the chunks. We call the resulting (n, t)-schedule H.

Lemma 6.4.2. *Let $n = q^2+q+1$, q prime power, $t = \kappa n$, $\kappa \in \mathbb{N}$, and let $0 \le a \le (q+1)t/n$. Waste for the (n, t)-schedule H is bounded by $\frac{a}{q+1} \le \mathcal{W}_H(a, a) \le \frac{a+q}{q+1}$.*

6.4.2 A recursive method when $t \geq n^{3/2}$

When the number of tasks t is sufficiently larger than the number of processors n, then H does not offer good control over waste when progress is $a = O(tn^{-1/2})$. Waste grows linearly and diverges increasingly from the lower bound, as the ratio t/n grows. Therefore, it is desired to construct an (n,t)-schedule that would provide tighter control over waste when t/n is large. The method presented below reschedules elements of H in the prefixes of length $t/\sqrt{n}$, and the resulting schedules are relatively close to the lower bound in this range. The method can be applied recursively to tighten the bound on waste in the first segments. We discuss only the first recursive step in detail.

For the construction of (n,t)-schedule $M = \langle \mu_1, \ldots, \mu_n \rangle$, let us assume that $n = q^2 + q + 1$, q prime power, $t = \kappa n$, $\kappa = (q+1)\bar{\kappa}$, $q + 1 = \bar{q}^2 + \bar{q} + 1$, $\bar{q}$ prime power, $\kappa, \bar{\kappa} \in \mathbb{N}$. (These technical assumptions about the existence of primes can be relaxed in exchange for a lower order increase of waste using the fact [71] that, for any $\epsilon > 0$ and any large enough x, there is always a prime number between x and $x + x^{11/20+\epsilon}$.) Let $G = \langle \gamma_1, \ldots, \gamma_n \rangle$ be the (n,n)-schedule constructed in Theorem 6.4.1, and let α be the permutation used in the construction of G. A schedule μ_i is constructed as follows. For clarity of presentation, we construct a matrix K with κ rows and $q+1$ columns, and the processor will perform tasks row by row: the entry $k_{x,y}$ in row x and column y of the matrix will be performed as task number $(x-1)(q+1)+y$ of the schedule μ_i i.e., $\mu_i((a-1)(q+1)+b) = k_{a,b}$.

The entries in the column y will be exactly the tasks from the chunk $\gamma_i(y)$ (i.e., tasks $\{\gamma_i(y)\kappa, \ldots, (\gamma_i(y)+1)\kappa - 1\}$) sequenced so as to ensure that this chunk is performed in different order by any two processors that have this chunk in one of the columns of its matrix. Note that we know which these processors are. Let us recall that, by Theorem 2.7.2, the task $\gamma_i(y)$ occurs in exactly $q+1$ different sets $L_{i_1}, \ldots, L_{i_{q+1}}$, $i_1 < \ldots < i_{q+1}$. By the construction of G, the set L_{i_j} is equal to the set of the first $q+1$ task in the t-schedule $\gamma_{\alpha^{-1}(i_j)}$. Thus the matrix K constructed for the schedule μ_z will have the chunk in one of the columns of K, if and only if $z = \alpha^{-1}(i_j)$, for some j, $1 \le j \le q+1$. Let j be such that $i = \alpha^{-1}(i_j)$. We order the chunk in column y of schedule μ_i according to the sequence given by the schedule χ_j of the $(q+1, \kappa)$-schedule V.

The theorem below gives a bound on waste of the (n,t)-schedule constructed above, for coarse-grained values of progress up to $(q+1)t/n$:

Theorem 6.4.3. *Let $n = q^2 + q + 1$, q prime power, $t = \kappa n$, $\kappa = (q+1)\bar{\kappa}$, $q + 1 = \bar{q}^2 + \bar{q} + 1$, $\bar{q}$ prime power, $\kappa, \bar{\kappa} \in \mathbb{N}$. Let $a = i(q+1)$, $b = j(q+1)$, $0 \le i, j \le t/n$. Let $i = \frac{t}{n(q+1)}\left(1 + \hat{i}\bar{q}\right)$, and $j = \frac{t}{n(q+1)}\left(1 + \hat{j}\bar{q}\right)$, $0 \le \hat{i}, \hat{j} \le \bar{q}+1$. Waste of the (n,t)-schedule M is bounded by:*

$$\mathcal{W}_M(a,b) \le \begin{cases} 0, & \text{if } \hat{i} + \hat{j} = 0, \\ \frac{t}{n(q+1)}, & \text{if } \hat{i} = 0, \hat{j} \ge 1 \text{ or } \hat{i} \ge 1, \hat{j} = 0, \\ \left(\bar{q} + \hat{i}\hat{j}\right)\frac{t}{n(q+1)}, & \text{if } \hat{i} \cdot \hat{j} \ge 1. \end{cases}$$

```
local static $s_i, g_1, \ldots, g_{q+1}$

nextTaskM( in $i \in [n]$ ;
           in-out  $matrix \in \{0,1\}$,
           $x \in [\frac{t}{n}]$,   $y \in [q+1]$  )
  if $matrix = 1$ then
    if $x = 1 \wedge y = 1$ then
      $s_i = \alpha(i)$
      $g_1 =$firstTaskG($i$)
      $j =$ findJ($g_1, s_i$)
      $\tau =$ firstTaskV($j$)
    else if $x = 1 \ \wedge \ y > 1$ then
      $g_y =$nextTaskG($i$)
      $j =$ findJ($g_y, s_i$)
      $\tau =$ firstTaskV($j$)
    else if $1 < x \leq t/n$ then
      $j =$ findJ($g_y, s_i$)
      $\tau =$ nextTaskV($j$)
    $m = (g_y - 1)t/n + \tau$
    if $x = t/n \ \wedge \ y = q+1$ then
      $matrix = 0;\ y = q+2;\ x = 1$
    else if $y = q+1$ then
      $y = 1;\ \ x = x+1$
    else
      $y = y+1$
  else $\{matrix = 0\}$
    if $x = 1$ then
      $g_y =$ nextTaskG($i$)
    $m = (g_y - 1)t/n + x$
    if $x = t/n$ then
      $x = 1;\ y = y+1$
    else
      $x = x+1$
  return($m$)

initializeM( in-out $matrix \in \{0,1\}$)
             $x \in [\frac{t}{n}]$,   $y \in [q+1]$  )
  $matrix = 1;\ x = 1;\ y = 1$
```

Figure 2: Algorithm for incremental construction of schedule μ_i from the (n,t)-schedule $M = \langle \mu_1, \ldots, \mu_n \rangle$.

Proof. Pick any two schedules μ_c and $\mu_{c'}$, $c \neq c'$, and let K and K' be the matrices constructed for the schedules as above. By the construction of M, after having executed task a and b of some two schedules from M, at most task number i and j respectively of some two columns have been executed. By the construction, for any two prefixes of length $(q+1)t/n$, only t/n tasks are the same. These tasks are located in a column of the matrix K and a column of the matrix K'. Note that by the construction of the matrices, tasks from each of the two columns were executed according to different schedules from the $(q+1,\kappa)$- schedule V. Hence we can use Lemma 6.1.1 to bound waste incurred. □

The bound on waste, which is presented only for specific values of progress a, is sufficient to show that waste is relatively close to the lower bound for all a that are sufficiently large, as shown in the following theorem.

Theorem 6.4.4. *For any a such that $\frac{t}{n}(1+\bar{q}^{5/3}) \leq a \leq \frac{t}{n}(q+1)$, waste of the (n,t)-schedule M is bounded by*

$$\mathcal{W}_M(a,a) \leq \left(1 + c \cdot n^{-1/12}\right) \cdot LB(a,a) \quad ,$$

where c is an absolute constant (independent of n, t, q, and $\bar{q}$).

Proof. We use the matrix K constructed above. Pick any a satisfying the condition from the statement of the theorem. The element number a belongs to a column of K and a row i. This column can be divided into consecutive chunks of size $\frac{t\bar{q}}{n(q+1)}$, and there exists $\hat{\imath}$ such that

$$\underline{i} := \frac{t}{n(q+1)}\left(1+(\hat{\imath}-1)\bar{q}\right) \;<\; i \;\leq\; \frac{t}{n(q+1)}\left(1+\hat{\imath}\bar{q}\right) =: \bar{\imath}$$

(the formulas define $\underline{i}$ and $\bar{\imath}$). Let $\bar{a} = \bar{\imath}(q+1)$, and $\underline{a} = \underline{i}(q+1)$. By the construction, we have $\underline{a} \;<\; a \;\leq\; \bar{a}$, and by the selection of a, $0 \leq \hat{\imath} \leq \bar{q}+1$. Of course waste for a is at most the value of waste for $\bar{a}$, and we can use Theorem 6.4.3 to bound it by $\mathrm{UB} = \mathcal{W}_M(a,a) \leq \mathcal{W}_M(\bar{a},\bar{a}) \leq \left(\bar{q}+\hat{\imath}^2\right)\frac{t}{n(q+1)}$. Of course the waste for a cannot be smaller than the value of a lower bound on waste for $\underline{a}$, and we can use lower bound Theorem 4.1.2 to bound it by

$$\mathrm{LB} \geq \mathrm{LB}(\underline{a},\underline{a}) \geq \frac{n}{t(n-1)}\underline{a}^2 - \frac{1}{n-1}\underline{a} = \frac{t}{n(n-1)}\left((\hat{\imath}-1)^2\bar{q}^2 + (\hat{\imath}-1)\bar{q}\right).$$

Consider the ratio of the upper bound to the lower bound. Using the above bounds we can bound the ratio by

$$\frac{\mathrm{UB}}{\mathrm{LB}} \leq \frac{(n-1)(\bar{q}+\hat{i}^2)}{(q+1)\bar{q}^2\left(\frac{\hat{i}-1}{\bar{q}}+(\hat{i}-1)^2\right)} \leq \frac{(n-1)}{(q+1)\bar{q}^2}\cdot\frac{\bar{q}+\hat{i}^2}{(\hat{i}-1)^2}$$
$$=\left(1+\frac{1}{\bar{q}}\right)\left(1+\frac{2}{\hat{i}-1}+\frac{\bar{q}+1}{(\hat{i}-1)^2}\right).$$

By the selection of a, $\bar{q}^{2/3} \leq \hat{i}$, and by assumption, $1 \leq \frac{1}{2}\bar{q}^{2/3}$. Thus we can bound the ratio by $\frac{\mathrm{UB}}{\mathrm{LB}} < \left(1+\frac{1}{\bar{q}}\right)\left(1+\frac{2}{\frac{1}{2}\bar{q}^{2/3}}+\frac{\bar{q}+1}{(\frac{1}{2}\bar{q}^{2/3})^2}\right) = 1+O\left(\frac{1}{\bar{q}^{1/3}}\right)$, and the result follows. □

The above construction dictates how to sequence the first $(q+1)\kappa$ tasks of any schedule μ_i. The remaining $t-(q+1)\kappa$ tasks are scheduled by following the sequence established by χ_i, $\mu_i(a) = \chi_i(a)$, $(q+1)\kappa < a \leq t$. Thus Theorem 6.1.2 also holds for the (n,t)-schedule M.

The construction of the (n,t)-schedule M is summarized in Figure 2. The initial cost of $O(n^2)$ (in order to find the permutation α) of constructing a schedule is fully amortized if we can do the construction off-line for all schedules (the permutation can be calculated once for all schedules). Note that if $t \geq n^2$, then the construction is amortized even if done in isolation.

The first step of recursion described above, can be used to have even better control over initial waste, by applying this step more times. We used $(q+1,\kappa)$-schedule V in the construction of (n,t)-schedule M, to control waste in the tail of the prefix. However, we could use $(q+1,\kappa)$- schedule M instead. This recursive process can be iterated as long as t is sufficiently large compared to n, and n is sufficiently large, and desired primes exist. We can use Theorem 6.4.4 to relate waste of the resulting system to the lower bound when progress is small.

6.4.3 A method when $n = t^{1-x}$

This section presents another technique for reducing the initial growth of waste. Recall that any two t-schedules share at most one set L_i^m. If we knew which set is shared between some two schedules, then we could reorder the sets so that the processors execute the task from this set last. The difficulty is that this shared set may be different across different schedules, and the reordering can be tricky. We observe that we can find many schedules that all share the same set which is the key idea of our construction.

We construct an (n,t)-schedule $S^m = \langle \sigma_1, \ldots, \sigma_n \rangle$, for $n = \frac{q^{m-1}-1}{q-1}$ and $t = \frac{q^m-1}{q-1}$, by selecting specific schedules from W^m, and reordering their tasks. Recall that for any two $u \neq u'$, among the $2(q+1)$ sets from Theorem 2.7.1 that contain either u or u', only two sets may be the same. Let us fix a set, say L_1^m. Naturally, for each element u in L_1^m, among the $q+1$ sets $L_{t_u^1}^m, \ldots, L_{t_u^{q+1}}^m$ that contain u there is L_1^m. So if we select from W^m only the schedules constructed for elements in L_1^m, then we know precisely which set is shared between the schedules. This set is L_1^m. This suggests the following construction of S^m. Let, for any $u \in L_1^m$, the blocks $L_{t_u^1}^m, \ldots, L_{t_u^q}^m, L_1^m$ be all the $q+1$ blocks that contain u. Recall that A_u is a subspace of dimension $m-2$ that is contained in any of the subspaces $B_{t_u^1}, \ldots, B_{t_u^q}, B_1$. We associate points from L_1^m with numbers from $[n]$ through a bijection id. The permutation $\sigma_{id(u)}$ is taken so that

$$\sigma_{id(u)}([e]) = \{j \mid \ell_j \subseteq A_u\},$$

$$\sigma_{id(u)}([e+(r-e)i+1, e+(r-e)(i+1)]) = \{j \mid \ell_j \subseteq B_{t^i_u} \setminus A_u\}, \qquad 0 \le i < q \ ,$$

$$\sigma_{id(u)}([e+(r-e)q+1, t]) = \{j \mid \ell_j \subseteq B_1 \setminus A_u\} \ .$$

Lemma 6.4.5. *Let $n = \frac{q^{m-1}-1}{q-1}$, $t = \frac{q^m-1}{q-1}$, $m \ge 3$, q prime power, and S^m be the (n,t)-schedule defined above. Let $k \le n$, $0 \le i_1, \ldots, i_k \le q$, $i = i_1 + \ldots + i_k$. Then*

$$\mathcal{W}_{S^m}(e+(r-e)i_1, \ldots, e+(r-e)i_k) \le e\left(\binom{i}{2} - \sum_{j=1}^{k}\binom{i_j}{2}\right).$$

Proof. Since $i_1, \ldots, i_h \le q$, all sets in the prefixes are associated with distinct $(m-1)$-dimensional subspaces. This adds at most $\binom{i}{2}e$ to the overlap. But for any t-schedule, sets do not add any overlap, and so we overestimated waste by $\left(\binom{i_1}{2} + \ldots + \binom{i_k}{2}\right) e$. □

The above lemma gives control over waste for coarse-grained values of progress. Although one can carry out analysis similar to that in Theorem 6.1.2, we content ourselves with a coarse-grained bound that explicitly exhibits the $\binom{h}{2}\frac{a^2}{t}$ leading factor.

Theorem 6.4.6. *If $a = e + (r-e)i$, $0 \le i \le q$, then k-waste, $2 \le k \le r$, is bounded by*

$$\mathcal{W}_{Q^m}(\underbrace{a, \ldots, a}_{k}) \le \binom{k}{2}\frac{a^2}{t} \cdot \left(1 + c_1 \cdot t^{-1/(m-1)} + c_2 \cdot k^{-1}\right) \ ,$$

where c_1 and c_2 are absolute constants.

Proof. We show the result by applying Lemma 6.4.5 and using a sequence of upper bounds presented below.

Since $i = \frac{(a-e)k}{r-e}$, we can bound $\binom{i}{2}$ from above by

$$\binom{i}{2} \leq \frac{a^2}{(r-e)^2} \cdot \frac{k^2}{2} \leq \frac{a^2}{(r-e)^2} \cdot \frac{k(k-1)(1+\Theta(k^{-1}))}{2}$$
$$= \binom{k}{2} \frac{a^2}{(r-e)^2}(1+\Theta(k^{-1})) \ .$$

Recall that $t = r + q(r-e)$, and so $r - e = \frac{t}{q}\left(1 - \frac{r}{t}\right)$. Hence we can rewrite the bound as:

$$\binom{i}{2} \leq \binom{k}{2} \frac{a^2}{t} \cdot \frac{1}{\frac{t}{q^2}\left(1-\frac{r}{t}\right)^2}(1+\Theta(k^{-1})) \ .$$

Since $t = q^{m-1}(1+\Theta(q^{-1}))$, and $e = q^{m-3}(1+\Theta(q^{-1}))$, we can write

$$\frac{e \cdot q^2}{t} = \frac{1+\Theta(q^{-1})}{1+\Theta(q^{-1})} = 1 + \Theta(q^{-1}) \ .$$

Since $r = q^{m-2}(1+\Theta(q^{-1}))$, we can write

$$\frac{1}{(1-\frac{r}{t})^2} = \frac{1}{(1-\frac{1}{q}(1+\Theta(q^{-1}))^2} \leq \frac{1}{1-\frac{2}{q}(1+\Theta(q^{-1}))} = 1 + \Theta(q^{-1}) \ .$$

Next observe that $(1+\Theta(k^{-1}))\cdot(1+\Theta(q^{-1}))\cdot(1+\Theta(q^{-1})) = 1+\Theta(q^{-1})+\Theta(k^{-1})$.

The result now follows from Lemma 6.4.5 with $i = \frac{ak}{r-e}$, the above bounds, and by noticing that the inequalities turn Θ into O. □

Chapter 7

Scheduling for shared memory systems

This chapter shows results on scheduling for shared memory systems. We study algorithms for the Certified Write-All problem. Such algorithms can be used to simulate a synchronous parallel machine on an asynchronous machine. The efficiency of an algorithm is measured by *work* that is equal to the worst-case total number of instructions executed by the algorithm. First we show how to create near-optimal instances of a Certified Write-All algorithm that was introduced by Anderson and Woll [4] (Section 7.3). Then we present a work-optimal deterministic asynchronous algorithm for the Certified Write-All problem, that solves a question posed by Martel et al. [105] in 1992 (Section 7.4). Finally, we present a conjecture about an efficient construction of schedules used by the algorithm of Anderson and Woll (Section 7.5).

7.1 Introduction

Many existing parallel systems are asynchronous. However, writing correct parallel programs on an asynchronous shared memory system is often difficult, for example because of data races, which are difficult to detect in general [15, 117]. When the instructions of a parallel program are written with the intention of being executed on a system that is synchronous, then it is easier for a programmer to write correct programs, because it is easier to reason about synchronous parallel programs than asynchronous ones. Therefore, in order to improve productivity in parallel computing, one could offer programmers the illusion that their programs run on a parallel system that is synchronous, while in fact the programs would be simulated on an asynchronous system.

Simulations of a parallel system that is synchronous on a system that is asynchronous have been studied for over a decade [6, 7, 8, 13, 28, 40, 61, 84, 87, 89, 90, 91, 105, 107, 132, 134]. Simplifying considerably, such simulations assume that there is a system with p asynchronous processors, and the system is to simulate a program written for n synchronous processors. The simulations use three main ideas: idempotence, load balancing, and synchronization. Specifically, the execution of the program is divided into a sequence of phases. A phase executes an instruction of each of the n synchronous programs. The simulation executes a phase in two stages. First the n instructions are executed and the results are saved to a scratch memory. Only then cells of the scratch memory are copied back to desired cells of the main memory. This ensures that the result of the phase is the same even if multiple processors execute the same instruction in a phase, which may happen due to asynchrony. The p processors run a load balancing algorithm to ensure that the n instructions of the phase are executed quickly despite possibly varying speeds of the p processors. In addition, the p processors should be synchronized at every stage, so as to ensure that the simulated program proceeds in lock-step. Such simulation implements the PRAM model [56] on an asynchronous system.

One challenge in realizing the simulations is the problem of "late writers" i.e., when a slow processor clobbers the memory of a simulation with a value from an old phase. This problem has been addressed in various ways: by replication of variables [89]; by a combination of hashing, replication, and error correction [7]; by approximate detection of who is late, and replication of variables [8]; by using instructions that execute relatively fast [105]; by versioning of variables using extra atomic primitives [106]; or by restricting a class of computations that can be simulated [105].

Another challenge is the development of efficient load-balancing and synchronization algorithms. This challenge is abstracted as the Certified Write-All (CWA) problem. In this problem, introduced in a slightly different form by Kanellakis and Shvartsman [84], there are p processors, an array w with n cells and a flag f, all initially 0, and the processors must set the n cells of w to 1, and then set f to 1. One efficiency criterion for the simulation is to reduce the wasteful use of computing resources. This use can be abstracted as the *work* complexity (or work for short) that is equal to the worst-case total number of instructions executed by the simulation. A simulation uses an algorithm that solves the CWA problem. Therefore, it is desirable to develop low-work algorithms that solve the CWA problem.

When creating a simulation of a given parallel program for n processors, one may have a choice of the number p of simulating processors. On the one hand, when a CWA algorithm for $p \gg n$ is used in a simulation, the simulation may be faster as compared to the simulation that uses an algorithm for $p \ll n$ processors, simply because of higher parallelism which means that more processors are available to perform the simulation. On the other hand, however, processors that access shared memory may create hotspots, which may cause delays, and as a result an algorithm for $p \gg n$ may run slower than an algorithm for $p \ll n$ (memory contention is disregarded in the model studied in this thesis). The actual speed of a simulation may depend on system parameters, and so it is interesting to study CWA algorithms for different relationships between p and n.

Deterministic algorithms that solve the CWA problem on an asynchronous system can be used to create simulations that have bounded worst-case overhead. Thus several deterministic algorithms have been studied [4, 24, 27, 70, 85, 116]. The best to date deterministic algorithm that solves the CWA problem on an asynchronous system for the case when $p = n$ was introduced by Anderson and Woll [4]. This algorithm is called AWT, and it generalizes the algorithm X of

Buss et al. [24]. The AWT algorithm uses a list of q permutations on $\{1, \ldots, q\}$. Anderson and Woll showed that for any $\epsilon > 0$, there is a $q \in \mathbb{N}$, a list of q permutations with desired *contention* (a value associated with a list of permutations, see Section 7.3.1 for a formal definition), and a constant c_q, such that for any $h > 0$, the algorithm for $p = q^h$ processors and $n = p$ cells that uses the list, has work at most $c_q \cdot n^{1+\epsilon}$. Note that this upper bound includes a multiplicative constant factor that is a function of q. The result that an $O(n^{1+\epsilon})$ work algorithm can be found is very interesting from theoretical standpoint. However, a different search objective will occur when a simulation is developed for a specific parallel system.

A specific parallel system will have a fixed number p of processors. It is possible to create many instances of the AWT algorithm for these p processors and $n = p$ cells, that differ by the number q of permutations used to create an instance. It is possible that the work of these different instances is different. If this is indeed the case, then it is interesting to find an instance with the lowest work, so as to create a relatively more efficient simulation on this parallel system.

Section 7.3 studies a method for creating near-optimal instances of the AWT algorithm of Anderson and Woll. We show that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work. Specifically, we show a tight (up to an absolute constant) lower bound on work of the AWT algorithm instantiated with a list of q permutations (appearing in Lemma 7.3.5). This lower bound generalizes the Lemma 5.20 of Anderson and Woll by exposing a constant that depends on q and on the contention of the list. We then combine our lower bound with a lower bound on the contention of permutations given by Lovász [99] and Knuth [93], to show that for any $\epsilon > 0$, the work of any instance must be at least $n^{1+(1-\epsilon)\sqrt{2\ln\ln n/\ln n}}$, for any large enough n (appearing in Theorem 7.3.9). The resulting bound is nearly optimal, as demonstrated by our method for creating instances of the AWT algorithm. We show that for any $\epsilon > 0$ and for any m that is large enough, when $q = \lceil e^{\sqrt{1/2\ln m\ln\ln m}}\rceil$, and $h = \lceil\sqrt{2\ln m/\ln\ln m}\rceil$, then there exists an instance of the AWT algorithm for $p = q^h$ processors and $n = p$ cells that has work at most $n^{1+(1+\epsilon)\sqrt{2\ln\ln n/\ln n}}$ (appearing in Theorem 7.3.10). We also prove that there is a penalty if one selects a q that is too far away from $e^{\sqrt{1/2\ln n\ln\ln n}}$. For any fixed $r \geq 2$, and any large enough n, work is at least $n^{1+r/3\cdot\sqrt{2\ln\ln n/\ln n}}$, whenever the AWT algorithm is instantiated with q permutations, such that $16 \leq q \leq e^{\sqrt{1/2\ln n\ln\ln n}/(r\cdot\ln\ln n)}$ or $e^{r\cdot\sqrt{1/2\ln n\ln\ln n}} \leq q \leq n$ (appearing in Proposition 7.3.11).

This chapter also studies algorithms for the CWA problem for the case when $p \ll n$. Fixing $r \geq 1$, when $p = n^{1/r}$ all known to date deterministic algorithms for the asynchronous CWA problem have work $\omega(n)$. Specifically, when $r = 1$ the first asynchronous CWA algorithm, called X, was developed by Buss et al. [24]. This algorithm was later generalized by Anderson and Woll [4], and the generalized algorithm, called AWT, has work $\Omega(n^{1+\sqrt{\ln\ln n/\ln n}/2})$. (The authors showed that there exists a deterministic algorithm with work $O(n^{1+\epsilon})$, for any $\epsilon > 0$. Such algorithm can be created in time that is independent of n or t, but is exponential in $1/\epsilon$.) When $r \geq 2$ a naive algorithm where each processor writes to every cell of the array w has work $\Omega(n^{1+1/r})$. The best known algorithm for $r \geq 2$ is due to Anderon and Woll [4]. This algorithm, called AW, has work $\Omega(n \log n)$ (the authors showed an upper bound of $O(n \log n)$ on work of their algorithm), which can be shown using a lower bound of Lovász [99] and Knuth [93] (this algorithm can be instantiated using results of Naor and Roth [116], Kanellakis and Shvartsman [85], and Chlebus et al. [27], with the same asymptotic lower bound on work). The elegant algorithm of Groote et al. [70] has work $\Omega(n^{1+1/(2r2^r \ln 2)})$ (the authors showed that their algorithm has work complexity $O(np^{\log_2(1+1/x)})$, where $x = n^{1/\log_2 p}$) and it uses a Test-And-Set instruction [73]. Buss et al. [24] give a lower bound of $n+\Omega(p \log(n/p))$ on work of asynchronous CWA algorithms that use Test-And-Set.

An interesting deterministic algorithm, called T, for 3 processors is due to Buss et al. [24]. In this algorithm two processors start from the two opposite tips of the array w, each works towards the opposite tip. The third processor starts from the middle of the array and "expands" by setting to 1 further and further cells on each side of the starting cell. When a "collision" between two processors occurs, the two processors "jump" to repeat the pattern of work recursively in a different part of the array. The algorithm has work $O(n)$, and at most $n+O(\log n)$ cells of the array are set to 1. The problem of generalizing this algorithm to more than 3 processors was posed by Buss et al. In a recent paper Groote et al. [70]

say: “Algorithm T does not appear to be generalizable to larger numbers of processes.”

Section 7.4 presents a work-optimal deterministic algorithm for the asynchronous Certified Write-All problem for a nontrivial number of processors. Our algorithm has work complexity of $O(n+p^4 \log n)$ (appearing in Theorem 7.4.15). The algorithm has optimal work for a nontrivial number of processors $p \leq (n/\log n)^{1/4}$. In contrast, all known to date deterministic algorithms require as much as $\omega(n)$ work when $p = n^{1/r}$, for any fixed $r \geq 1$. The p processors combined set, at most, $n+4p^3 \log n$ cells of w to 1. The processors use $O(n+p^4 \log n)$ memory cells for coordinating their work (shown in Theorem 7.4.16). Our algorithm generalizes the collision principle used by the algorithm T. Namely, each processor has a collection of intervals of w and iteratively selects an interval to work on. The processor proceeds from one tip (or end) of the interval towards the other tip. When processors collide, they exchange appropriate information and schedule their future work accordingly. Our algorithm uses Test-And-Set instructions to detect collisions. Finally, we show an $\Omega(n+p\log p)$ lower bound on work of any deterministic algorithm that solves the Certified Write-All problem and that uses Test-And-Set (appearing in Theorem 7.4.17). When $p = n$, our lower bound improves the lower bound Buss et al. [24].

7.2 Model and definitions

We consider a shared memory system where processors can work at arbitrarily varying paces. Our formal definition is based on the Atomic Asynchronous Parallel System as presented by [8] (cf. [34, 35, 39, 67, 95, 105, 118, 137]).

The system consists of p processors, each of which has a dedicated local memory and a unique identifier from the set $\{1, \ldots, p\}$, and every processor has access to shared memory. Each processor has a discrete local clock ranging over $\mathbb{N}$. A processor executes exactly one *basic action* at any tick of the local clock unless the processor has halted. The basic actions that a processor can execute include: a $Halt$ action that stops the operation of the processor, an operation on a constant number of cells from the local memory [1] , and a transfer between the local memory and shared memory. The possible transfers are: reading a single cell of shared memory into a cell of the local memory; writing from a cell of the local memory to a cell of shared memory; and performing a Test-And-Set (TAS) action that checks if the value stored at a cell of shared memory is equal to the value stored at a cell of the local memory, if so the action sets the cell of shared memory to the value of a (possibly different) cell of the local memory, but in any case returns the result of the test.

An execution of an algorithm progresses according to the following model of asynchrony. Local time of processor i is mapped to global time through a

[1] An operation can be formally defined as an arbitrary function $f : \mathbb{N}^s \times [\log n]^s \rightarrow \mathbb{N}^s \times [\log n]^s$ that takes the addresses of s cells of the local memory and values stored there, and returns the addresses of s cells of the local memory and the new values that should be stored at these cells.

strictly increasing function $T_i : \mathbb{N} \to \mathbb{R}$. We assume that no local clock ticks of two processors are mapped to the same instant of global time i.e., if $T_i(x) = T_j(y)$, then $i = j$ and $x = y$. When mappings $T_1, \ldots, T_p$ have been fixed, each processor executes basic actions dictated by its algorithm. The processors take turns according to the total order prescribed by the mappings. Any processor i does not execute basic actions after the tick when the processor executed the $Halt$ action, if the processor executed the action. The execution of any basic action is instantaneous, and so the resulting memory updates are atomic.

We adopt the following definition of the Certified Write-All (CWA) problem: given an array $w[0, \ldots, n-1]$ with n cells and a flag f, all located in $n+1$ cells of shared memory and all initially 0, set the n cells of w to 1 and then set f to 1. An algorithm *solves* the CWA problem for p processors and n cells, if for all allowed choices for mappings, each processor halts after a finite number of local clock ticks, and, whenever a processor halts, the n cells and the flag f have been set to 1.

The *work* complexity of a deterministic algorithm that solves the CWA problem for p processors and n cells measures the maximum total number of basic actions executed by the processors. Consider any mappings $T_1, \ldots, T_p$ allowed by our model of asynchrony. Let h_i be the first local clock tick when processor i executes the $Halt$ action or ∞ if it does not execute the action. Then the total number of basic actions executed by the processors is $\sum_{i=1}^{p} h_i$. The work of the algorithm is defined as the maximum value of the sum across allowed choices for mappings (Work is a function of n and p.)

Definition 7.2.1. *Work of a deterministic algorithm A for p processors and n cells that solves the Certified Write-All problem is defined as*

$$work(A,p,n) = \max_{T_1,\ldots,T_p} \sum_{i=1}^{p} h_i(T_1,\ldots,T_p) \ ,$$

where the maximum is taken over T_i's that are strictly increasing functions from $\mathbb{N}$ to $\mathbb{R}$ such that no two functions map numbers to the same number from $\mathbb{R}$; and where $h_i(T_1,\ldots,T_p)$ is a number from $\mathbb{N}$ that is the first tick of the local clock of processor i during which the processor executes the Halt basic action, when local time of processors have been mapped to global time using the maps $T_1,\ldots,T_p$.

Note that in this model, there is a trivial Write-All algorithm for $n = p$ where the first basic action that a processor i, $0 \leq i \leq n-1$, executes is an assignment of 1 to cell i of the array w (because the model ensures that each processor will eventually perform a basic action). This takes $O(n)$ work in total. However, in general no processor can certify and halt right after performing its first basic action, as the processor cannot ensure that always each of the n cells has been set to 1 (because other processors may be delayed).

7.3 A method for creating near-optimal instances of a Certified Write-All algorithm

This section shows how to create near-optimal instances of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll [4]. In this algorithm n processors update n memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations, where q can be chosen from a wide range of values. This section shows that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work.

7.3.1 Preliminaries

For a permutation ρ on $[q] = \{1, \ldots, q\}$, $\rho(v)$ is a *left-to-right maximum* [93] if it is larger than all of its predecessors; i.e., $\rho(v) > \rho(1), \rho(v) > \rho(2), \ldots, \rho(v) > \rho(v-1)$. The *contention* [4] of ρ with respect to a permutation α on $[q]$, denoted as $Cont(\rho, \alpha)$, is defined as the number of left-to-right maxima in the permutation $\alpha^{-1}\rho$ that is a composition of α^{-1} with ρ. For a list $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ of q permutations on $[q]$ and a permutation α on $[q]$, the contention of R_q with respect to α is defined as $Cont(R_q, \alpha) = \sum_{v=1}^{q} Cont(\rho_v, \alpha)$. The contention of the list of permutations R_q is defined as $Cont(R_q) = max_{\alpha \ on \ [q]} Cont(R_q, \alpha)$.

Lovász [99] (see Exercise 3.17.(a)) and Knuth [93] showed that the expectation of the number of left-to-right maxima in a random permutation on $[q]$ is H_q (H_q is the qth harmonic number). This immediately implies the following lower bound on contention of a list of q permutations on $[q]$.

Lemma 7.3.1. *[99, 93] For any list R_q of q permutations on $[q]$, $Cont(R_q) \geq qH_q > q \ln q$.*

Anderson and Woll [4] showed that for any q there is a list of q permutations with contention at most $3qH_q$. Since $H_q / \ln q$ tends to 1, as q tends to infinity, the following lemma holds.

Lemma 7.3.2. *[4] For any q that is large enough, there exists a list of q permutations on $[q]$ with contention at most $4 \cdot q \ln q$.*

We describe the algorithm AWT of Anderson and Woll [4] that solves the CWA problem when $p = n$. There are $p = q^h$ processors, $h \geq 1$, and the array w that has $n = p$ cells. The identifier of a processor is represented by a distinct string of length h over the alphabet $[q]$. The algorithm is instantiated with a list of q permutations $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ on $[q]$, and we write AWT(R_q) when we refer to the instance of algorithm AWT for a given list of permutations R_q. This list is available to every processor (in its local memory). Processors have access to a shared q-ary tree called *progress tree*. Each node of the tree is labeled with a string over alphabet $[q]$. Specifically, a string $s \in [q]^*$ that labels a node identifies the path from the root to the node (e.g., the root is labeled with the empty string λ, the leftmost child of the root is labeled with the string 1). For convenience, we say node s, when we mean the node labeled with a string s. Each node s of the tree, apart from the root, contains a *completion bit*, denoted by

```
AWT($R_q$)
01    $Traverse(h, \lambda)$
02    set $f$ to 1 and Halt

$Traverse(i, s)$
01    if $i = 0$ then
02        $w[val(s)] := 1$
03    else
04        $j := q_i$
05        for $v := 1$ to $q$
06            $a := \rho_j(v)$
07            if $b_{s \cdot a} = 0$ then
08                $Traverse(i - 1, s \cdot a)$
09                $b_{s \cdot a} := 1$
```

Figure 3: The instance AWT(R_q) of an algorithm of Anderson and Woll, as executed by a processor with identifier $\langle q_1 \dots q_h \rangle$. The algorithm uses a list of q permutations $R_q = \langle \rho_1, \dots, \rho_q \rangle$.

b_s, initially set to 0. Any leaf node s is canonically assigned a distinct number $val(s) \in \{0, \dots, n-1\}$.

The algorithm, shown in Figure 3, starts by each processor calling procedure AWT(R_q). Each processor traverses the q-ary progress tree by calling a recursive procedure $Traverse(h, \lambda)$. When a processor visits a node that is the root of a subtree of height i (the root of the progress tree has height h) the processor takes the ith letter j of its identifier (line 04) and attempts to visit the children in the order established by the permutation ρ_j. The visit to a child $a \in [q]$ *succeeds* only if the completion bit $b_{s \cdot a}$ for this child is still 0 at the time of the attempt (line 07). In such case, the processor recursively traverses the child subtree (line 08), and later sets to one the completion bit of the child node (line 09). When a processor visits a leaf s, the processor performs an assignment of 1 to the cell $val(s)$ of the array w. After a processor has finished the recursive traversal of the progress tree, the processor sets f to 1 and halts.

We give two technical lemmas that will be used to solve recursive equations in the following section.

Lemma 7.3.3. *Let h and q be integers, $h \geq 1$, $q \geq 2$, and $k_1 + \ldots + k_q = c > 0$. Consider a recursive equation $W(0, r) = r$, and $W(i, r) = r \cdot q + \sum_{v=1}^{q} W(i - 1, k_v \cdot r/q)$, when $i > 0$. Then for any r, $W(h, r) = r\left(q \cdot \frac{(c/q)^h - 1}{c/q - 1} + (c/q)^h\right)$.*

Proof. First we observe that W is right-linear i.e., that $W(i, z \cdot r) = z \cdot W(i, r)$, for any real z, which can be shown by induction on i. We use right-linearity to solve the recursive equation as follows:

$$\begin{aligned} W(h, r) &= r \cdot q + \sum_{v=1}^{q} k_v/q \cdot W(h - 1, r) = r \cdot q + c/q \cdot W(h - 1, r) \\ &= r \cdot q \sum_{j=0}^{h-1} (c/q)^j + r(c/q)^h = r \cdot q \frac{(c/q)^h - 1}{c/q - 1} + r(c/q)^h \ . \end{aligned}$$

□

Lemma 7.3.4. *Let h and q be integers, $h \geq 1$, $q \geq 2$, and for any string $s \in [q]^*$, $k_1^s + \ldots + k_q^s = c > 0$. Consider a recursive equation $V(s, r) = 3r$, for any string s of length h, and $V(s, r) = 7rq + \sum_{v=1}^{q} V(s \cdot v, k_v^s \cdot r/q)$, for any string s of length less than h. Then for the empty string λ and any r, $V(\lambda, r) = r\left(7q \cdot \frac{(c/q)^h - 1}{c/q - 1} + 3(c/q)^h\right)$.*

Proof. First we observe that V is right-linear i.e., that $V(s, z \cdot r) = z \cdot V(s, r)$, which can be shown by backward induction on the length of s, starting from length h. We also observe that the value of $V(s, r)$ is the same across all strings of the same length i.e., for any s and s' of the same length $V(s, r) = V(s', r)$.

This can also be shown by backward induction, where the inductive step is

$$\begin{aligned} V(s,r) &= 7rq + \sum_{v=1}^{q} k_v^s/q \cdot V(sv,r) = 7rq + \sum_{v=1}^{q} k_v^s/q \cdot V(s1,r) \\ &= 7rq + c/q \sum_{v=1}^{q} \cdot V(s'1,r) = V(s',r) \ . \end{aligned}$$

We use these two properties to solve the recursive equation as follows:

$$\begin{aligned} V(\lambda,r) &= 7r \cdot q + \sum_{v=1}^{q} k_v^\lambda/q \cdot V(v,r) = 7r \cdot q + c/q \cdot V(1,r) \\ &= 7r \cdot q \sum_{j=0}^{h-1} (c/q)^j + 3r(c/q)^h = 7r \cdot q \frac{(c/q)^h - 1}{c/q - 1} + 3r(c/q)^h \ . \end{aligned}$$

□

7.3.2 Near-optimal instances of AWT

This section presents a method for creating near-optimal instances of the AWT algorithm. The main idea of this section is that for fixed number p of processors and $n = p$ cells of the array w, work of an instance of the AWT algorithm depends on the number of permutations used by the instance, along with their contention. This observation has several consequences. It turns out (not surprisingly) that work increases when contention increases, and conversely it becomes the lowest when contention is the lowest. Here a lower bound on contention of permutations given by Lovász [99] and Knuth [93] is very useful, because we can bound work of any instance from below, by an expression in which the value of contention of the list used in the instance is replaced with the value of the lower bound on contention. Then we study how the resulting lower bound on work depends on the number q of permutations on $[q]$ used by the instance.

It turns out that there is a single value for q, where the bound attains the global minimum. Consequently, we obtain a lower bound on work that, for fixed n, is independent of both the number of permutations used and their contention. Our bound is near-optimal. We show that if we instantiate the AWT algorithm with about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ permutations that have small enough contention, then work of the instance nearly matches the lower bound. Such permutations exist as shown by Anderson and Woll [4]. We also show that when we instantiate the AWT algorithm with much fewer or much more permutations, then work of the instance must be significantly greater than the work that can be achieved. Details of the overview follow.

We will present a tight bound on work of any instance of the AWT algorithm. Our lower bound generalizes the Lemma 5.20 of Anderson and Woll [3]. The bound has an explicit constant which was hidden in the analysis given in the Lemma 5.20. The constant will play a paramount role in the analysis presented in the reminder of the section.

Lemma 7.3.5. *Work W of the AWT algorithm for $p = q^h$ processors, $h \geq 1$, $q \geq 2$, and $n = p$ cells, instantiated with a list $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ of q permutations on $[q]$, is bounded by $\frac{c}{84} \cdot n^{1+\log_q \frac{Cont(R_q)}{q}} \leq W \leq c \cdot n^{1+\log_q \frac{Cont(R_q)}{q}}$, where $c = \frac{28q^2}{Cont(R_q)}$.*

Proof. The idea of the lemma is to carefully account for work spent on traversing the progress tree, and spent on writing to the array w. The lower bound will be shown by designing an execution during which the processors will traverse the progress tree in a specific, regular manner. This regularity will allow us to conveniently bound work inside a subtree from below by work done at the root of the subtree and work done by quite large number of processors that traverse the child subtrees in a regular manner. A similar recursive argument will be used to derive the upper bound.

Recall that our model assumes that no two basic actions are executed at the same time. In the proof below, however, we will say that some instructions are executed at the same instant. This is done for convenience of description only, and a simple transformation allows us to prove the lemma in our model. Indeed, the concurrent instructions will be either: reads from the same shared location, or writes of the same value to the same shared location, or operations on disjoint subsets of memory. Therefore, the concurrent instructions may be sequenced in arbitrary order in a close enough neighborhood of the instant, to comply with the assumptions of our model. Such transformation guarantees that the lemma holds in our model.

Consider any execution of the algorithm. We say that the execution is *regular at a node s* (recall that s is a string from $[q]^*$) iff the following three conditions hold:

(i) the r processors that ever visit the node during the execution, visit the node at the same time,

(ii) at that time, the completion bit of any node of the subtree of height i rooted at the node s is equal to 0,

(iii) if a processor visits the node s, and x is the suffix of length $h-i$ of the identifier of the processor, then the q^i processors that have x as a suffix of their identifiers, also visit the node during the execution.

We define $W(i,r)$ to be the largest number of basic actions that r processors perform inside a subtree of height i, from the moment when they visit a node s that is the root of the subtree until the moment when each of the visitors finishes traversing the subtree, maximized across the executions that are regular at s and during which exactly r processors visit s (if there is no such execution, we put $-\infty$). Note that the value of $W(i,r)$ is well-defined, as it is independent of the choice of a subtree of height i (any pattern of traversals that maximizes the number of basic actions performed inside a subtree, can be applied to any other subtree of the same height), and of the choice of the r visitors (suffixes of length $h-i$ do not affect traversal within the subtree). There exists an execution that is regular at the root of the progress tree, and so the value of $W(h,n)$ bounds work of AWT(R_q) from below.

We will show a recursive formula that bounds $W(i,r)$ from below. We do it by designing an execution recursively. The execution will be regular at every node of the progress tree. We start by letting the q^h processors visit the root at the same time. For the recursive step, assume that the execution is regular at a node s that is the root of a subtree of height i, and that exactly r processors visit the node. We first consider the case when s is an internal node i.e., when $i > 0$. Based on the i-th letter of its identifier, each processor picks a permutation that gives the order in which completion bits of the child nodes will be read by the processor. Due to regularity, the r processors can be partitioned into q collections of equal cardinality, such that for any collection j, each processor in the collection checks the completion bits in the order given by ρ_j. Let for any collection, the processors in the collection check the bits of the children of the node in lock step (the collection behaves as a single "virtual" processor). Then, by Lemma 2.1 of Anderson and Woll [4], there is a pattern of delays so that every processor in some $k_v \geq 1$ collections succeeds in visiting the child $s \cdot v$ of the node at the same time. Thus the execution is regular at any child node. The lemma also guarantees that $k_1 + \ldots + k_q = Cont(R_q)$, and that these $k_1, \ldots, k_q$ do not depend on the choice of the node s. Since each processor checks q completion bits of the q children of the node, the processor executes at least q basic actions while traversing the node. Therefore, $W(i,r) \geq rq + \sum_{v=1}^{q} W(i-1, k_v \cdot r/q)$, for $i > 0$. Finally, suppose that s is a leaf i.e., that $i = 0$. Then we let the r processors work in lock step, and so $W(0,r) \geq r$.

We can bound the value of $W(h,n)$ using Lemma 7.3.3, the fact that $h = \log_q n$, and that for any positive real a, $a^{\log_q n} = n^{\log_q a}$, as follows

$$\begin{aligned}
W(h,n) &\geq n \cdot (Cont(R_q)/q)^h \left(q \cdot \frac{1-(q/Cont(R_q))^h}{Cont(R_q)/q - 1} + 1 \right) \\
&= n^{1+\log_q(Cont(R_q)/q)} \left(q^2/Cont(R_q) \cdot \frac{1-(q/Cont(R_q))^h}{1 - q/Cont(R_q)} + 1 \right) \\
&> q^2/Cont(R_q) \cdot n^{1+\log_q(Cont(R_q)/q)} \left(1 - (q/Cont(R_q))^h \right) \\
&\geq 1/3 \cdot q^2/Cont(R_q) \cdot n^{1+\log_q(Cont(R_q)/q)} \ ,
\end{aligned}$$

where the last inequality holds because for all $q \geq 2$, $q/Cont(R_q) \leq 2/3$, and $h \geq 1$.

The argument for proving an upper bound will be similar to the above argument for proving the lower bound. The main conceptual difference is that processors may write completion bits in different order for different internal nodes of the progress tree. Therefore, while the coefficients $k_1, \ldots, k_q$ were the same for each node during the analysis above, in the analysis of the upper bound presented now, each internal node s will have its own coefficients $k_1^s, \ldots, k_q^s$ that may be different for different nodes. To see this, take any execution and consider the root node $s = \lambda$. The r processors that visit the node satisfy condition (iii) and so they can be divided into q collections, each of cardinality r/q, where any processor in a collection j reads the completion bits in the order given by the permutation ρ_j. During the execution, the completion bits of the child nodes are set to 1 in some sequence, and let α_s be this sequence (α_s is a permutation on $[q]$). The argument of Anderson and Woll ensures that any processor from a collection j can only visit a child $s \cdot v$, $1 \leq v \leq q$, when $\left(\alpha_s^{-1}\rho_j\right)(v)$ is a left-to-right maximum of the permutation $\alpha_s^{-1}\rho_j$. Let k_v^s be the number of permutations ρ_j such that $\left(\alpha_s^{-1}\rho_j\right)(v)$ is a left-to-right maximum. By the definition of contention, $k_1^s + \ldots + k_q^s \leq Cont(R_q)$. Work could only be increased if every processor from the collection j indeed visited every child node that corresponds to left-to-right maximum of $\alpha_s^{-1}\rho_j$ (a processor from the collection may not visit every child if another processor from the collection is quite fast and manages to set the completion bits to 1 before its peers check the bits). But if every processor from every collection visited every child node admissible by the sequence α_s, then the $k_v^s \cdot r/q$ processors that would visit the child node $s \cdot v$ would satisfy condition (iii), and so could be divided into q collections of equal cardinality, and every processor from such collection would check completion bits according to a distinct permutation. Therefore, we could repeat the argument described for the root recursively for the child nodes. As a result, we obtain a recursive formula $V(s,r) \leq 3r$, when s is a string of length h, and $V(s,r) \leq 7rq + \sum_{v=1}^{q} V(s \cdot v, k_v^s \cdot r/q)$, when s is a string of length less than h, while for all s, $k_1^s + \ldots + k_q^s \leq Cont(R_q)$. We can bound work of the given execution from above by $V(\lambda, n)$. The result now follows by applying Lemma 7.3.4 and observing that $7q \cdot \frac{1-(q/Cont(R_q))^h}{1-q/Cont(R_q)} + 3$ can be bounded from above by $\frac{28q^2}{Cont(R_q)}$, because $q/Cont(R_q) \leq 2/3$ and $3 \leq 3q^2/Cont(R_q)$. □

How does the bound from the preceding lemma depend on contention of the list R_q? We should answer this question so that when we instantiate the AWT algorithm, we know whether to choose permutations with low contention or perhaps with high contention. The answer to the question may be not so clear at first, because for any given q, when we take a list R_q with lower contention, then although the exponent of n is lower, but the constant c is higher. In the lemma below we study this tradeoff, and demonstrate that it is indeed of advantage to choose lists of permutations with as small contention as possible.

Lemma 7.3.6. *The function $c \mapsto q^2/c \cdot n^{\log_q c}$, where $c > 0$ and $n \geq q \geq 2$, is a non-decreasing function of c.*

Proof. We consider a derivative

$$\begin{aligned}\frac{\partial}{\partial c}\left(q^2/c \cdot n^{\log_q c}\right) &= -q^2/c^2 \cdot n^{\log_q c} + q^2/c \cdot (\ln n)/(c \ln q) \cdot n^{\log_q c} \\ &= \left(-1 + \log_q n\right) q^2/c^2 \cdot n^{\log_q c} \ .\end{aligned}$$

Recall that $n \geq q \geq 2$, and so $\log_q n \geq 1$. Thus the derivative is non-negative. □

This lemma, simple as it is, is actually quite useful. In several parts of the section we use a list of permutations, for which we only know an upper bound or a lower bound on contention. The lemma allows us to bound work respectively from above or from below, even though we do not actually know the exact value of contention of the list.

We would like to find out how the lower bound on work depends on the choice of q. The subsequent argument shows that careful choice of the value of q is essential, in order to guarantee low work. We begin with two technical lemmas, the second of which bounds from below the value of a function occurring in Lemma 7.3.5.

The next lemma shows that an expression that is a function of x must vanish inside a “slim” interval.

Lemma 7.3.7. *Let $\epsilon > 0$ be any fixed constant. Then for any large enough n, the expression $x^2{-}x{+}(1-\ln x)\cdot\ln n$ is negative when $x{=}x_1{=}\sqrt{1/2\ln n\ln\ln n}$, and positive when $x{=}x_2{=}\sqrt{(1+\epsilon)/2\ln n\ln\ln n}$.*

Proof. The key idea of the proof is that x^2 creates in the expression a highest order summand with factor either $1/2$ or $(1+\epsilon)/2$ depending on which of the two values of x we take, while $\ln x$ creates a summand of the same order with factor $1/2$ independent of the value of x. As a result, for the first value of x, the former “is less positive” than the later “is negative”, while when x has the other value, then the former “is more positive” than the later “is negative”. This intuition is made precise next.

We will show that the expression is negative when $x = \sqrt{1/2\ln n\ln\ln n}$. We split the expression into two parts: $x^2 - x$ and $(1-\ln x)\ln n$, and compare their values. Obviously, $x^2 - x < 1/2\ln n\ln\ln n$. We rewrite the second part as $(1-\ln x)\ln n = -1/2\ln n\ln\ln n(1/2\ln(1/2)-1+1/2\ln\ln\ln n)\ln n$, and observe that, for large enough n, the triple logarithm $\ln\ln\ln n$ is large enough so that any negative constant summand in the parenthesis becomes balanced by a positive summand, and so the expression inside parenthesis becomes positive. As a result, for large enough n, the second part is smaller or equal to $-1/2\ln n\ln\ln n$. This means that the sum of the first part and the second part is negative, as desired.

We will show that the expression is positive when $x = \sqrt{(1+\epsilon)/2 \ln n \ln\ln n}$. Since the summand x of the first part is of lower order than the summand x^2, we know that for large enough n, $x^2 - x > (1+\epsilon/2)/2 \ln n \ln\ln n$ (recall that $\epsilon > 0$ is a fixed constant). Now a key observation is that the multiplier of the highest order summand $\ln n \ln\ln n$ of the second part is equal to $1/2$, and not $(1+\epsilon)/2$, because it is obtained by taking a logarithm of a square root. Thus the second part $(1-\ln x)\ln n = -1/2 \ln n \ln\ln n - (1/2\ln((1+\epsilon)/2) - 1 + 1/2 \ln\ln\ln n)\ln n$ is more than $-(1+\epsilon/2)/2 \ln n \ln\ln n$, for large enough n. This completes the proof. □

Lemma 7.3.8. *Let $\epsilon > 0$ be any fixed constant. Then for any large enough n, the value of the function $f : [\ln 3, \ln n] \to \mathbb{R}$, defined as $f(x) = e^x/x \cdot n^{\ln x/x}$, is bounded from below by $f(x) \geq n^{(1-\epsilon)\sqrt{2\cdot\ln\ln n/\ln n}}$.*

Proof. We shall show the lemma by reasoning about the derivative of f. We will see that it contains two parts: one that is strictly convex, and the other that is strictly concave. This will allow us to conveniently reason about the sign of the derivative, and where the derivative vanishes. As a result, we will ensure that there is only one local minimum of f in the interior of the domain. An additional argument will ascertain that the values of f at the boundary are larger than the minimum value attained in the interior.

Let us investigate where the derivative

$$\frac{\partial f}{\partial x} = e^x n^{\ln x/x}/x^3 \cdot \left(x^2 - x + (1-\ln x)\ln n\right)$$

vanishes. It happens only for such x, for which the parabola $x \mapsto x^2 - x$ "overlaps" the logarithmic plot $x \mapsto \ln n \ln x - \ln n$. We notice that the parabola is strictly convex, while the logarithmic plot is strictly concave. Therefore, we conclude that one of the three cases must happen: plots do not overlap, plots overlap at a single point, or plots overlap at exactly two distinct points. We shall see that the later must occur for any large enough n.

We will see that the plots overlap at exactly two points. Note that when $x = \ln 3$, then the value of the logarithmic plot is negative, while the value of the parabola is positive. Hence the parabola is "above" the logarithmic plot at the point $x = \ln 3$ of the domain. Similarly, it is "above" the logarithmic plot at the point $x = \ln n$, because for this x, the highest order summand for the parabola is $\ln^2 n$, while it is only $\ln n \ln \ln n$ for the logarithmic plot. Finally, we observe that when $x = \sqrt{\ln n}$, then the plots are "swapped": the logarithmic plot is "above" the parabola, because for this x the highest order summand for the parabola is $\ln n$, while the highest order summand for the logarithmic plot is as much as $1/2 \ln n \ln \ln n$. Therefore, for any large enough n, the plots must cross at exactly two points in the interior of the domain.

Now we are ready to evaluate the monotonicity of f. By inspecting the sign of the derivative, we conclude that f increases from $x = \ln 3$ until the first point, then it decreases until the second point, and then it increases again until $x = \ln n$. This holds for any large enough n.

This pattern of monotonicity allows us to bound from below the value of f in the interior of the domain. The function f attains a local minimum at the second point, and Lemma 7.3.7 teaches us that this point is in the range between $x_1 = \sqrt{1/2 \ln n \ln\ln n}$ and $x_2 = \sqrt{(1+\epsilon)/2 \ln n \ln\ln n}$. For large enough n, we can bound the value of the local minimum from below by $f_1 = e^{x_1}/x_2 \cdot n^{\ln x_1/x_2}$. We can further weaken this bound as

$$f_1 = n^{-\ln x_2/\ln n + \ln x_1/x_2 + x_1/\ln n} \geq n^{-\ln x_2/\ln n + 1/2 \ln\ln n/x_2 + \sqrt{1/2 \ln\ln n/\ln n}}$$

$$\geq n^{(1-\epsilon)\sqrt{2 \cdot \ln\ln n/\ln n}} ,$$

where the first inequality holds because for large enough n, $\ln(1/2 \ln\ln n)$ is positive, while the second inequality holds because for large enough n, $\ln x_2 \leq \ln\ln n$, and $1/\sqrt{1+\epsilon} \geq 1-\epsilon$, and for large enough n, $\sqrt{1/2 \ln\ln n/\ln n} - \ln\ln n/\ln n$ is larger than $\sqrt{1/(2+2\epsilon) \ln\ln n/\ln n}$.

Finally, we note that the values attained by f at the boundary are strictly larger then the value attained at the second point. Indeed, $f(\ln n)$ is strictly grater, because the function strictly increases from the second point towards $\ln n$. In addition, $f(\ln 3)$ is strictly grater because it is at least $n^{1.08}$, while the value attained at the second point is bounded from above by n raised to a power that tends to 0 as n tends to ∞ (in fact it suffices to see that the exponent of n in the bound on f_1 above, tends to 0 as n tends to ∞).

This completes the argument showing a lower bound on f. □

The following two theorems show that we can construct an instance of AWT that has the exponent for n arbitrarily close to the exponent that is required, provided that we choose the value of q carefully enough.

Theorem 7.3.9. *Let $\epsilon > 0$ be any fixed constant. Then for any n that is large enough, any instance of the AWT algorithm for $p = n$ processors and n cells has work at least $n^{1+(1-\epsilon)\sqrt{2\ln\ln n/\ln n}}$.*

Proof. This theorem is proven by combining the results shown in the preceding lemmas. Take any AWT algorithm for n cells and $p = n$ processors instantiated with a list R_q of q permutations on $[q]$. By Lemma 7.3.5, work of the instance is bounded from below by the expression $q^2/(3Cont(R_q)) \cdot n^{1+\log_q(Cont(R_q)/q)}$. By Lemma 7.3.6, we know that this expression does not increase when we replace $Cont(R_q)$ with a number that is smaller or equal to $Cont(R_q)$. Indeed, this is what we will do. By Lemma 7.3.1, we know that the value of $Cont(R_q)$ is bounded from below by $q \ln q$. Hence work of the AWT is at least $n/3 \cdot q/\ln q \cdot n^{\ln\ln q/\ln q}$.

Now we would like have a bound on this expression that does not depend on q. This bound should be fairly tight so that we can later find an instance of the AWT algorithm that has work close to the bound. Let us make a substitution $q = e^x$. We can use Lemma 7.3.8 with $\epsilon/2$ to bound the expression from below as desired, for large enough n, when q is in the range from 3 to n. What remains to be checked is how large work must be when the AWT algorithm is instantiated with just two permutations (i.e., when $q = 2$). In this case we know what contention of any list of two permutations is at least 3, and so work is bounded from below by n raised to a fixed power strictly greater than 1. Thus the lower bound holds for large enough n. $\square$

Theorem 7.3.10. *Let $\epsilon > 0$ be any fixed constant. Then for any large enough m, when $q = \lceil e^{\sqrt{1/2 \ln m \ln\ln m}} \rceil$, and $h = \lceil \sqrt{2 \ln m / \ln\ln m} \rceil$, there exists an instance of the AWT algorithm for $p = n = q^h$ processors and n cells that has work at most $n^{1+(1+\epsilon)\sqrt{2\ln\ln n/\ln n}}$.*

Proof. Had it not been for the ceilings in the definitions of q and h, the result would have been immediate. The main problem that we are facing is that taking ceiling could make q too far away from the best possible choice for q, and so work of the resulting algorithm could be too large compared to the lower bound on work of Theorem 7.3.9. However, this cannot happen, as we will see shortly. Intuitively, this is because h and q are quite small, and so q^h is close to the q^h with ceilings dropped. This intuition is formally shown next.

We construct a specific instance of the AWT algorithm and bound its work from above. By Lemma 7.3.2, for any q that is large enough, there exists a list of q permutations on $[q]$ with contention at most $4q \ln q$. Thus, by Lemma 7.3.5 and Lemma 7.3.6, work W of the AWT algorithm instantiated with this list is at most $W \leq 28q/(4\ln q) \cdot n^{1+\ln(4\ln q)/\ln q}$, for any m that is large enough. We now apply a series of algebraic manipulations to bound this expression from above, for large enough m. Specifically, W can be bounded as

$$\begin{aligned} W &\leq 28 \cdot n^{1+\ln(4\ln q)/\ln q+\ln q/\ln n} \\ &\leq n^{1+\ln(8\sqrt{1/2\ln m \ln\ln m})/\sqrt{1/2\ln m\ln\ln m}+\left(1+\ln 28+\sqrt{1/2\ln m\ln\ln m}\right)/\ln m} \\ &\leq n^{1+\left(1+\sqrt{\ln\ln\ln m}/\ln\ln m\right)\sqrt{2\ln\ln m/\ln m}}\ , \end{aligned}$$

where the second inequality holds because for large enough m, $q \leq e^{1+\sqrt{1/2\ln m\ln\ln m}} \leq e^{2\cdot\sqrt{1/2\ln m\ln\ln m}}$, and because $m \leq n$, while the third inequality holds because $\ln(8\sqrt{1/2\ln\ln m})/\sqrt{1/2\ln m\ln\ln m}+(1+\ln 28)/\ln m \leq \ln\ln\ln m/\sqrt{1/2\ln m\ln\ln m}$.

Our goal now is to replace every occurrence of m above with n. Since $m \leq n$, we can easily do the substitution in the enumerator. However, we also have an expression $1/\ln m$, where such substitution could decrease the exponent. In order to alleviate this problem, we will show that n and m are close to each other. Let $\bar{q}$ and $\bar{h}$ be equal to q and h respectively except for the ceilings dropped i.e., $\bar{q} = e^{\sqrt{1/2\ln m\ln\ln m}}$, and $\bar{h} = \sqrt{2\ln m/\ln\ln m}$. We can trivially bound n from above by m^3, because $\bar{q}^2 \geq q$, for large enough m. However, any upper bound of m raised to a power bounded away from 1 is not satisfactory, as it will boost the constant 1 in the $(1+\sqrt{\ln\ln\ln m}/\ln\ln m)$ factor in the expression above. Therefore, we need a tighter upper bound, and we will develop it now. We first note that $\left((\bar{q}+1)/\bar{q}\right)^{\bar{h}} \leq e^{\bar{h}/\bar{q}} \leq m^{\bar{h}/\ln m} \leq m^{\sqrt{2/(\ln m\ln\ln m)}}$, and that for large enough m, $\bar{q}+1 \leq \bar{q}^2 = m^{2\ln\bar{q}/\ln m} \leq m^{\sqrt{2\ln\ln m/\ln m}}$. Using the fact that, for large enough m, $\ln(m^3)\ln\ln(m^3) \leq 4\ln m\ln\ln m$, and $n \leq m^3$, we obtain two bounds: $1/(\ln m\ln\ln m) \leq 4/(\ln n\ln\ln n)$ and $1/\ln m \leq 3/\ln n$. These two bounds can be applied to replace m with n in the former two bounds, and we experience a slight weakening of the former bounds:$\left(\left(\bar{q}+1\right)/\bar{q}\right)^{\bar{h}} \leq m^{\sqrt{8/(\ln n \ln\ln n)}}$, and $\bar{q}+1 \leq m^{\sqrt{6 \ln\ln n/\ln n}}$. We combine the latest two bounds to show that n is bounded from above by m raised to a power that tends to 1, as n tends to infinity. Specifically, $n \leq (\bar{q}+1)^{\bar{h}+1} = \bar{q}^{\bar{h}}((\bar{q}+1)/\bar{q})^{\bar{h}} \cdot (\bar{q}+1) \leq m^{1+8\sqrt{\ln\ln n/\ln n}}$. This ensures that $1/\ln m \leq (1+8\sqrt{\ln\ln n/\ln n})/\ln n$, for any large enough m. This bound allows us to replace m with n in the expression above, while maintaining the $(1+o(1))$ multiplicative factor. Thus the result follows. □

The above two theorems teach us that when q is selected carefully, we can create an instance of the AWT algorithm that is nearly optimal. A natural question that one immediately asks is: what if q is *not* selected well enough? Lemma 7.3.5 and Lemma 7.3.6 teach us that lower bound on work of an instance of the AWT algorithm depends on the number q of permutations on $[q]$ used by the instance. On one extreme, if q is a constant that is at least 2, then work must be at least n to some exponent that is greater than 1 and that is bounded away from 1. On the other extreme, if $q = n$, then work must be at least n^2. In the "middle", when q is about $e^{\sqrt{1/2 \ln n \ln\ln n}}$, then the lower bound is the weakest, and we can almost attain it as shown in the preceding two theorems. Suppose that we chose the value of q slightly away from the value $e^{\sqrt{1/2 \ln n \ln\ln n}}$. By how much must work be increased as compared to the lowest possible value of work? Although one can carry out a more precise analysis of the growth of a lower bound as a function of q, we will be contented with the following result, which already establishes a gap between the work possible to attain when q is chosen well, and the work required when q is not chosen well.

Proposition 7.3.11. *Let $r \geq 2$ be any fixed constant. For any large enough n, if the AWT algorithm is instantiated with q permutations on $[q]$, such that $16 \leq q \leq e^{\sqrt{1/2 \ln n \ln\ln n}/(r \cdot \ln\ln n)}$ or $e^{r \cdot \sqrt{1/2 \ln n \ln\ln n}} \leq q \leq n$, then its work is at least $n^{1+r/3 \cdot \sqrt{2 \ln\ln n / \ln n}}$.*

Proof. By Lemma 7.3.5, Lemma 7.3.6, and Lemma 7.3.1, work of the AWT algorithm instantiated with any list of q permutations on $[q]$ is at least $q/\ln q \cdot n^{1+\ln\ln q/\ln q - \ln 3/\ln n}$. Suppose that q falls into the first interval. By taking logarithm of both sides of the inequality $q \leq e^{\sqrt{1/2 \ln n \ln\ln n}/(r \cdot \ln\ln n)}$, we obtain an inequality $r \cdot \sqrt{2 \ln\ln n / \ln n} \leq 1/\ln q$. Recall that $q \geq 16$, and so $\ln\ln q \geq 1$. Therefore, we can multiply the right-hand side of the former inequality by $\ln\ln q$ without violating the inequality, and obtain $r/2 \cdot \sqrt{2 \ln\ln n / \ln n} \leq \ln\ln q / \ln q$, as desired. Now suppose that q falls into the second interval. We obtain the desirable bound by observing that then $q \geq n^{r \cdot \sqrt{1/2 \ln n \ln\ln n}/\ln n} = n^{r/2 \cdot \sqrt{2 \ln\ln n/\ln n}}$. □

7.4 A work-optimal deterministic algorithm for the asynchronous Certified Write-All problem

This section presents a deterministic asynchronous algorithm for the Certified Write-All problem. The algorithm has work complexity of $O(n + p^4 \log n)$, which is optimal for a nontrivial number of processors $p \leq (n/\log n)^{1/4}$.

7.4.1 Collision algorithm

The algorithm (see Figure 4) generalizes the collision principle of algorithm T [24]. The main algorithmic approaches of our algorithm are: to ensure that any processor often works on a relatively large interval of unset cells of the array w according to a sequence that enables rapid detection of two processors setting to 1 the same cell of the array; and, when the same cell has been set to 1 more than once, to ensure effective mechanism of reassigning work to processors. Briefly speaking, all processors share an array tab with n cells used for coordination of their work. Each processor maintains a list of intervals of the array tab (an interval is a subset of consecutive cells of the array). A processor takes an interval from the list and keeps setting cell $w[x]$ to 1 and marking the cell $tab[x]$ with some special information, while working from a tip (or end) of this interval towards the opposite tip. Later, the processor removes some intervals or their parts from the list, possibly based on information obtained from other processors. This process is repeated as long as there is an interval on the list. When the list becomes empty, the processor sets the flag f to 1 and halts.

There are several challenges that we solve to ensure that our algorithm avoids doing redundant work. It may happen that two processors "collide" at the same cell while working in the opposite or the same directions. When they work in the opposite directions then it could happen that they "cross" each other and duplicate the work that the other processor already did. When they work in the same direction then they may keep on working “side-by-side” and again duplicate the work that the other processor is doing. Another potential problem is that even if we are able to detect collisions, a processor that collides must decide upon a cell of the table where the processor will resume its work from. Ideally, the processor should chose to work from a tip of an interval, so that this tip is “far away” from any cell that any other processor is currently working on. This is desirable because it would help to ensure that when the next collision of this processor occurs, substantial number of distinct new cells of the array w have been set to 1.

Intuitively, our algorithm solves these challenges as follows. The processors coordinate their work on intervals using atomic Test-And-Set (TAS) instructions. This ensures that whenever a processor does a successful TAS to a cell, no other processor can succeed. As a result, a colliding processor sets at most one cell of w to 1 before it detects a collision with another processor and has an opportunity to reassign its own future work. The choice of a relatively long interval located in a rather unassigned part of the table is intuitively done by a processor always working on an interval that is at least as long as half of the length of a longest interval on the list. In addition, we ensure that a colliding processor obtains knowledge from the other processor about which cells of w remain to be set to 1, and this allows us to guarantee that when a processor often collides it must substantially reduce the amount of work remaining to be done even though it has *not* actually recently set to 1 any distinct cells of w.

The following two sections present the details of this intuitive explanation. The first section makes some observations about the flow of information between colliding processors. We use these results in the second section to prove a bound on work for our algorithm.

7.4.2 Collections of intervals, their transformations, and preserved properties

This section defines several properties of collections of intervals (see Figure 5 for illustrations), and shows that certain transformations of the collections preserve these properties. All intervals in this and subsequent sections are over the set of integers $\{0, 1, \ldots, n-1\}$.

Definition 7.4.1. *Let $U_0, \ldots, U_g$ be collections of intervals over $\{0, \ldots, n-1\}$. We say that the collections are* **regular** *iff the following three properties are satisfied:*

(i) any of the collections is composed of mutually disjoint nonempty intervals, each interval has length that is a power of two (length can be 1), and the lengths of any two intervals from the same collection differ by at most factor of 2,

(ii) for any two intervals I and J, each from a different collection, either $I \subseteq J$ or $J \subseteq I$ or $I \cap J = \emptyset$,

(iii) if $I \subseteq J$ are two intervals from different collections, then J can be partitioned into a power of 2 intervals of the same length and I is among the partitions.

We say that the collections are **monotonic** *iff the following property is satisfied:*

(iv) for any j, $0 \leq j < g$, if interval I belongs to U_{j+1}, then there is interval J that belongs to U_j such that $J \supseteq I$.

We now show a simple fact that halving any interval in the last collection preserves monotonicity.

Lemma 7.4.1. *Let $U_0, \ldots, U_g$ be monotonic collections of intervals, and let U be equal to U_g except for some interval of length 2 or more replaced by its two halves. Then the collections $U_0, \ldots, U_g, U$ are monotonic.*

Proof. We consider two cases. If $j < g$ then the existence of an interval J in U_j that contains any given interval I contained in U_{j+1} is ensured because collections $U_0, \ldots, U_g$ are monotonic. For the second case, let $j = g$ and I be an interval from U. If I is one of the two halves of the interval J from U_g that was halved, then this interval J (strictly) contains I. If I is none of the halves then U_g contains I. □

The following lemma states that halving a relatively long interval in any collection preserves regularity.

Lemma 7.4.2. *Let $U_0, \ldots, U_g$ be regular collections of intervals and let U be equal to U_i, $0 \leq i \leq g$, except for some longest interval H from U_i which has length 2 or more replaced by its two halves. Then the collections $U_0, \ldots, U_g, U$ are regular.*

Proof. Since H is an interval from U_i that has the maximum length across all intervals in U_i and its length is at least 2, then the collection U contains intervals that differ by at most factor of two. Thus collections $U_0, \ldots, U_g, U$ satisfy property (i).

To see that properties (ii) and (iii) are also satisfied let us take any interval I from the collection U and any interval J of from some collection U_j, $0 \leq j \leq g$. If I is not any of the halves of H then I must be an interval from U_i and so the properties hold by the assumption that $U_0, \ldots, U_g$ are regular. Suppose that I is one of the halves of H. By property (ii) that holds for the collections $U_0, \ldots, U_g$, either $H \cap J = \emptyset$, or $H \subseteq J$, or $J \subset H$. Let us consider these three cases in turn. Firstly, suppose that $H \cap J = \emptyset$. Then property (ii) holds for any of the halves and J, because their intersection with J is empty, and property (iii) is void for any of the halves and J. Secondly, suppose that $H \subseteq J$. Obviously property (ii) holds for each of the halves and J, because any half is included in J. By property (iii), J can be partitioned into a power of two intervals one of which is H. But then J can be partitioned into twice as many intervals two of which are the halves of H. Hence property (iii) holds for J and each of the halves. Thirdly, suppose that $J \subset H$. By assumption, H can be partitioned into two or more intervals one of which is J. Then J is a subset (not necessarily strict) of one of the halves and J has empty intersection with the other half. As a result, property (ii) holds for J and each half. In addition, the half that includes J can be partitioned into twice fewer than H can (perhaps just one) intervals one of which is J. Thus property (iii) holds for J and each of the halves. $\square$

Taking a specific intersection of two collections preserves regularity and monotonicity, as shown below.

Lemma 7.4.3. *Let $k_1, \ldots, k_p \geq 0$, the collections $U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ be regular, and the collections $U_h^0, \ldots, U_h^{k_h}$ be monotonic for any h, $1 \leq h \leq p$. Let $i \neq j$ be two numbers from $\{1, \ldots, p\}$, $0 \leq m \leq k_j$, and $U = U_i^{k_i}$ and $U' = U_j^m$. Let 2^k be the maximum length of an interval in U and $2^{k'}$ be the maximum length of an interval in U'. Let V be the collection*

$$V = \begin{cases} U' \cap U := \left\{\ H \mid H \neq \emptyset \ \wedge\ J \in U' \ \wedge\ H = J \cap \bigcup_{I \in U} I\ \right\} & \text{if } k > k' \\ \\ U \cap U' := \left\{\ H \mid H \neq \emptyset \ \wedge\ I \in U \ \wedge\ H = I \cap \bigcup_{J \in U'} J\ \right\} & \text{if } k \leq k'. \end{cases}$$

Then the number of intervals in V is at most the maximum of the number of intervals in U and in U'. If $k > k'$ then V contains some complete intervals from U', and when $k < k'$ some complete intervals from U. If $k = k'$ then V contains some complete intervals from U and a single half for each of some other intervals of length 2^k from U (V never contains two halves of any interval from U). The collections $V, U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, V$ are monotonic.

Proof. We shall study the content of V based on the relationship between the length of a longest interval in U and the length of a longest interval in U'. In preparation for the case analysis we record what the lengths of intervals in these collections may be. Since collections are regular, by property (i), we indeed know that the length of a longest interval in U is 2^k, and the length of a longest interval in U' is $2^{k'}$, for some integers $k, k' \geq 0$. We also know that intervals in U have length 2^k or 2^{k-1}, and the intervals in U' have length $2^{k'}$ or $2^{k'-1}$.

For the first case, suppose that $k > k'$, and let us investigate common parts between U' and U. Let I be an interval from U, and J from U'. The interval I cannot be shorter than 2^{k-1}, and the interval J cannot be longer than 2^{k-1}. Therefore, J is too short to be a strict superset of I. Hence, by property (ii), any interval J from U' is either a subset of some interval from U or does not intersect with any interval from U. Consequently, the set $U' \cap U$ contains only some complete nonempty intervals from U'. The length of a longest interval in $U \cap U'$ is reduced by the factor of 2 or more compared to the length of a longest interval in U. Since removing an interval from a collection does not invalidate the properties, the collections V,$U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular and the collections $U_i^0, \ldots, U_i^{k_i}, V$ are monotonic.

The second case is when $k < k'$. Here we can carry out similar analysis as is the above paragraph. The set $U \cap U'$ contains only some complete nonempty intervals from U, and so regularity and monotonicity hold, however we do not guarantee that the length of a longest interval is reduced.

The final case is when $k = k'$. Let us again investigate the result of the operation $U \cap U'$. Take any interval I from U, and let us see what part of this interval is contained in $U \cap U'$, if any. By property (ii), for any interval J from U' we have: either $I \subseteq J$, or $I \supset J$, or $I \cap J = \emptyset$. If the first subcase occurs, then we are guaranteed that complete I is contained in $U \cap U'$. Suppose that the first subcase does not happen, and so for all J, either $I \supset J$ or $I \cap J = \emptyset$ (but never $I \subseteq J$). The result now depends on how many distinct J there are that satisfy $I \supset J$. Suppose that $I \supset J$. Our assumption about the length of intervals ensures that the length of such J is 2^{k-1} and of I is 2^k. By property (i), we can have either zero, or one, or two intervals in U' that are strict subsets of I. In the former situation all intervals J have empty intersection with I, and so I is not contained in $U \cap U'$. In the later situation the two intervals combined must yield I, and so complete I is contained in $U \cap U'$. The discussion presented so far in this paragraph implies that regularity and monotonicity trivially hold because the intervals contained in $U \cap U'$ are complete intervals from U. In the middle situation, by property (iii), the interval I is partitioned into two halves: J and $I \setminus J$, and so only the half J is contained in $U \cap U'$, but not the other half. An argument similar to that in Lemma 7.4.1 shows that monotonicity is preserved, and similar to that in Lemma 7.4.2 shows that regularity is preserved. □

The operation of intersection defined in the above lemma yields a certain reduction of size or length of the intervals in the resulting collection V compared to the given collection U.

Corollary 7.4.4. *If $V \neq U$ then either the length of a longest interval in V is smaller or equal to half of the length of a longest interval in U, or the intervals in V combined have fewer elements than the intervals in U combined by least half of the length of a longest interval in U.*

Proof. The analysis again proceeds by considering the relationship between the lengths of longest intervals in U and U'. We consider three mutually exclusive cases. Suppose that $k > k'$. Then, by Lemma 7.4.3, the length of a longest interval in V is equal or shorter than half of the length of a longest interval in U. If $k < k'$ then V contains some complete intervals from U, and since $U \neq V$, one of them must be missing in V. This missing interval has length at least half of the length of a longest interval in U. If $k = k'$ then V contains complete intervals from U or their halves. Again, since $U \neq V$, either one interval is missing or its half. If a half is missing, then this half must contain at least 2^{k-1} elements. $\square$

From now until the end of Section 7.4.2, let us fix the values of the collections $U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ as well as the values of variables i, j, and m. Let V be the collection defined in Lemma 7.4.3 for the fixed values of the collections and the variables. We let I be a fixed interval from the collection $U_i^{k_i}$, and J a fixed interval from the collection U_j^m, such that these two intervals have nonempty intersection. This intersection contains a cell where a processor collides with other processor. The lemmas and corollaries in the remainder of Section 7.4.2 show transformations on collections that preserve regularity and monotonicity, and lead to certain reductions in the number of elements contained in the intervals of the resulting collections. The proofs are similar to the proofs of Lemma 7.4.3 and Corollary 7.4.4 (see Figure 6 for illustrations).

Lemma 7.4.5. *Let D be a possibly empty prefix of I and D' a possibly empty suffix of J, or vice versa D be a suffix and D' a prefix, such that $D \cap D' = \emptyset$ and $D \cup D'$ is a nonempty interval. Let W be the collection*

$$W = V \setminus (D \cup D') := \{ \ H \mid H \neq \emptyset \ \wedge \ K \in V \ \wedge \ H = K \setminus (D \cup D') \ \} \ .$$

Then the number of intervals in W is at most the number of intervals in V. If $k \neq k'$ then the collection W contains some complete intervals from V. If $k = k'$ then W contains some complete intervals from V and a single half for each of some other intervals from V. The collections $W, U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, V, W$ are monotonic.

Proof. To prove the lemma, we take any interval K from V and argue about what the result of subtracting $D \cup D'$ from K is. We arrange the argument in 3 cases by the relationship between the length of a longest interval in U and in U'.

For the first case suppose that $k > k'$. Then, by Lemma 7.4.3, the collection V contains only some complete intervals from U'. Inspecting the possible lengths of intervals from U and U' reveals that it cannot happen that an interval from U is a strict subset of some interval from U', and so, by property (ii), $J \subseteq I$ (recall that we assume that $I \cap J \neq \emptyset$). Similarly, for any K from V, $K \subseteq I$ or $K \cap I = \emptyset$. Note that the interval $D \cup D'$ starts at a tip of I, runs through the entire J, and ends at the opposite tip of J. Thus the three sets $(D \cup D') \setminus J$, J, and $I \setminus (D \cup D')$ are disjoint possibly empty intervals whose union is I. If K does not intersect with I, then $K \setminus (D \cup D') = K$. Assume now that K is a subset of I. Since J and K are intervals from U' we have, by property (i), that either $J = K$ or $J \cap K = \emptyset$. In the first subcase $K \setminus (D \cup D') = \emptyset$, while in the second subcase K either belongs to $(D \cup D') \setminus J$ or to $I \setminus (D \cup D')$, and so $K \setminus (D \cup D')$ is equal to $\emptyset$ or K respectively. Thus W is equal to V, except for possibly some complete intervals removed, and so desired regularity and monotonicity hold.

A symmetric case is when $k < k'$. Now collection V contains only some complete intervals from U, and it must be that $I \subseteq J$, and that for any K from V, $K \subseteq J$ or $K \cap J = \emptyset$. As above, if K does not intersect with J, then $K \setminus (D \cup D')$ is equal to K, while if $K \subseteq J$, then $K \setminus (D \cup D')$ is either empty or equal to K, and so desired regularity and monotonicity hold.

Finally, consider the case when $k = k'$. Since $I \cap J \neq \emptyset$, by property (ii), we have three subcases $J \subset I$, $I \subset J$, $I = J$. We consider them in turn. For the first subcase suppose that $J \subset I$. Since, by property (i), the length of I and J can be either 2^k or 2^{k-1}, the length of I is 2^k and, by property (iii), J is a half of I and has length 2^{k-1}. Thus $D \cup D'$ is either equal to I or equal to a half of I. Take any K from V. By Lemma 7.4.3, V contains complete intervals from U or halves of some other intervals form U but never two halves of the same interval from U. If K is an interval from U, then, by property (i), K is either equal to I or has empty intersection with I. If $K \cap I = \emptyset$ then $K \setminus (D \cup D') = K$. If $K = I$

then $K \setminus (D \cup D')$ is equal to either a half of K or an empty set depending on whether $D \cup D'$ is the other half of K or not. If K is a half of an interval from U, then either $K \cap I = \emptyset$ or K is a half of I. If K is a half of I, then $K \setminus (D \cup D')$ is either K or an empty set. Thus in the first subcase the set $K \setminus (D \cup D')$ has length either 2^k or 2^{k-1} or 0 and is equal to K, or a half of K, or the empty set. Hence desired regularity and monotonicity hold. For the second subcase suppose now that $I \subset J$. Then J has length 2^k, I is its half of length 2^{k-1}, and $D \cup D'$ is equal to J or I. Take any K from V. Since K is an interval from U or its half and has length 2^k or 2^{k-1}, K is too short to be a strict superset of J, and, by property (ii), $K \subseteq J$ or $K \cap J = \emptyset$. When the later is true, $K \setminus (D \cup D') = K$. Let $K \subseteq J$. If the length of K is 2^k, then $K = J$ and $K \setminus (D \cup D')$ is equal to $\emptyset$ or a half of K. If the length of K is 2^{k-1}, then $K \setminus (D \cup D')$ is equal to K or $\emptyset$. Thus in the second subcase the set $K \setminus (D \cup D')$ has length either 2^k or 2^{k-1} or 0. Hence desired regularity and monotonicity hold. Finally, consider the last subcase when $I = J$. Then the set $K \setminus (D \cup D')$ is either empty or equal to K, and so it has length either 2^k, or 2^{k-1}, or 0, and so desired regularity and monotonicity hold. □

Corollary 7.4.6. *If $U = V$ then the intervals in W combined have fewer elements than the intervals in U combined by least half the length of a longest interval in U.*

Proof. If $U = V$ then the I used in the statement of Lemma 7.4.5 belongs to V. Consequently, one of the sets K from the statement of Lemma 7.4.5 is equal to I. We now follow the last three paragraphs of the proof of Lemma 7.4.5 to see what the difference is between V and W. It cannot be that $k > k'$ because then $U \neq V$. If $k < k'$ then the set $D \cup D'$ used in the statement of Lemma 7.4.5 contains I, and so the collection W does not contain the interval I, which has length at least half of the length of the longest interval is U. If $k = k'$ then: when $J \subset I$ then I has the length of a longest interval in U and the set $D \cup D'$ is at least a half of I, which is removed; while when $I \subset J$ then $D \cup D'$ contains I and so I is removed; finally when $I = J$ then I is removed. □

Lemma 7.4.7. *Let $D \neq \emptyset$ be a prefix of I and $D' \neq \emptyset$ a prefix of J such that x is the smallest element in D' and $x - 1$ is the largest element in D, or vice versa $D \neq \emptyset$ be a suffix of I and $D' \neq \emptyset$ a suffix of J such that x is the largest element in D' and $x + 1$ is the smallest element in D. Let Q be the collection*

$$Q = V \setminus D := \{\ H \mid H \neq \emptyset\ \wedge\ K \in V\ \wedge\ H = K \setminus D\ \}\ .$$

Then the number of intervals in Q is at most the number of intervals in V. If $k \neq k'$ then Q contains some complete intervals from V. If $k = k'$ then Q contains some complete intervals from V and a single half for each of some other intervals from V. The collections Q, $U_1^0, \ldots, U_1^{k_1}$, ……, $U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections U_i^0, …, $U_i^{k_i}, V$, Q are monotonic.

Proof. We begin by showing that there are just two cases to consider. Since $I \cap J \neq \emptyset$, by property (ii), we know that either $I \subseteq J$ or $J \subset I$. Suppose that $I \subseteq J$. Then, when D' is a prefix of J, x must be the smallest element in J, and so $x - 1$ does not belong to J and so cannot belong to I either, while we know that x belongs to I, or when D' is a suffix of J, x is the largest element in J and so $x + 1$ does not belong to J and so cannot belong to I either. Hence the assumption that $I \subseteq J$ leads to a contradiction. Consequently it must be that $J \subset I$, and so either $k = k'$ or $k > k'$.

For the first case, suppose that $k > k'$. Then V contains only some complete intervals from U'. Take any K from V. By property (ii), we have one of the three subcases: $K \subseteq I$, $K \supset I$, $K \cap I = \emptyset$. The interval K is too short to be a strict superset of I, so the middle subcase cannot happen. If the last subcase happens, then $K \setminus D = K$. Let us now focus on the first subcase when $K \subseteq I$. Notice that the sets D, J, and $I \setminus (D \cup J)$ are disjoint intervals and their union is I. By property (i), either $K = J$ or $K \cap J = \emptyset$. If $K = J$ then $K \setminus D = K$, while when K does not intersect with J, then $K \setminus D$ is either empty or equal to K. Consequently the collection Q contains only some complete intervals from V and so desired regularity and monotonicity hold.

Finally, consider the second case when $k = k'$. Take any K from V. The collection V contains some complete intervals from U or halves of some other intervals from U but never two halves of any interval from U. Hence, by property (i), either $K \cap I = \emptyset$, or $K = I$, or K is a half of I but then there is no other interval in V that is equal to the other half of I. In the first subcase $K \setminus D = K$, and so let us focus on the two remaining subcases. Note that the length of J can be either 2^k or 2^{k-1}, but since $J \subset I$, then the length of J must be 2^{k-1}, and J must be a half of I. As a result, D and J are the two halves of I. So if $K = I$ then $K \setminus D$ is equal to J, a half of K. If K is a half of I, then $K \setminus D$ is either K or empty. Again regularity and monotonicity hold for Q. □

Corollary 7.4.8. *If $U = V$ then the intervals in Q combined have fewer elements than the intervals in U combined by at least half the length of a longest interval in U.*

Proof. As explained in Corollary 7.4.6, we have that $k \leq k'$ and I is in V, and so $I = K$ for some K from the statement of the Lemma 7.4.7. It cannot be that $k < k'$ because this is disallowed by the proof of Lemma 7.4.7. Thus the only possible relationship between k and k' is that $k = k'$. In this case I has length 2^k and a half of I is removed. □

Lemma 7.4.9. *Let x be the smallest element in I, $D' \neq \emptyset$ a prefix of J such that x is the largest element in D', or vice versa x be the largest element in I, $D' \neq \emptyset$ a suffix of J such that x is the smallest element in D'. Let R be the collection*

$$R = V \setminus (D' \setminus \{x\}) := \{\ H \mid H \neq \emptyset\ \wedge\ K \in V\ \wedge\ H = K \setminus (D' \setminus \{x\})\ \}\ .$$

Then the number of intervals in R is at most the number of intervals in V. The collection R contains some complete intervals from V. The collections R,

$U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, V,$ R are monotonic.

Proof. The argument is similar to that of Lemma 7.4.5: we take an interval K from V and argue about what part of the interval is in R. We start with an observation that when D' contains just one element x, then the result is trivial because $R = V$. Assume that D' contains at least two elements. But then J contains an element that is not in I and so, by property (ii), $I \subset J$. Consequently, we have just two cases to consider $k = k'$ and $k < k'$. We will study them in turn.

For the first case suppose that $k = k'$. Take any K from V. The collection V contains some complete intervals from U or halves of some other intervals from U. Hence, by property (i), either $K \cap I = \emptyset$, or $K = I$, or K is a half of I. Note that the length of K can be either 2^k or 2^{k-1} and the length of I is 2^{k-1}, so K cannot be a half of I. As a result, the last subcase does not happen and we have either $K \cap I = \emptyset$, or $K = I$. If $K = I$ then $K \setminus (D' \setminus \{x\}) = K$, while if $K \cap I = \emptyset$ then $K \setminus (D' \setminus \{x\})$ can be either K or empty. Again regularity and monotonicity hold for R.

Finally, consider the second case when $k < k'$. This case is very similar to the case when $k > k'$ in the proof of Lemma 7.4.7. We shall show it here for completeness. In this case V contains only some complete intervals from U. Take any K from V. By property (ii), we have one of the three mutually exclusive subcases: $K \subseteq J$, $K \supset J$, $K \cap J = \emptyset$. The interval K is too short to be a strict superset of J, so the middle subcase cannot happen. If the last subcase happens, then $K \setminus (D' \setminus \{x\}) = K$. Let us now focus on the first subcase when $K \subseteq J$. Notice that the sets $D' \setminus \{x\}$, I, and $J \setminus ((D' \setminus \{x\}) \cup I)$ are disjoint intervals and their union is J. By property (i), either $K = I$ or $K \cap I = \emptyset$. If $K = I$ then $K \setminus (D' \setminus \{x\}) = K$, while when K does not intersect with I, then $K \setminus (D' \setminus \{x\})$ is either empty or equal to K. Consequently, the collection R contains only some complete intervals from V, and so desired regularity and monotonicity hold. □

Corollary 7.4.10. *If $U = V$ and $R \neq V$ then the intervals in R combined have fewer elements than the intervals in U combined by least half the length of a longest interval in U.*

Proof. Since $U = V$ then each interval K from the statement of the Lemma 7.4.9 has length at least half of the length of a longest interval in U. But $R \neq V$ and so at least one interval or its part must have been removed. Lemma 7.4.9 shows that either a complete interval is removed or not a part of it at all. Thus at least one K is missing in R compared to V. □

7.4.3 Analysis of the algorithm

This section presents an analysis of the generalized collision algorithm given in Figure 4. We assume that n and p are powers of 2. Without loss of generality, we assume that TAS can transfer $O(p)$ cells between local and shared memory (this assumption can be easily relaxed to comply with our model by using pointer representation of the tuples that are TAS into cells of the array *tab*; see proof of Theorem 7.4.16 for details). For any processor i, we let $U_i^0, U_i^1, U_i^2, \ldots$ be the values of the local variable U of the processor i consecutively recorded (by an external observer) in line `05` as the processor iterates through the while loop (lines `04` to `28`). If a processor i never reaches line `05` of an iteration number k, then we define $U_i^{k-1} = \emptyset$ (this happens when the processor has halted during prior iterations). We say that a processor *is working on an interval*, when it is executing its first TAS in the for loop (lines `06` to `10`), or any instruction during this loop until the processor has executed the last TAS in the for loop. For a fixed execution, processor, and iteration of the while loop, we let U^z denote the value of U right before the processor executes line number z of the iteration, and U_z the value of U right after line number z. It will be clear form the context which execution, processor, and iteration U^z or U_z refer to.

The analysis starts with three lemmas and a corollary that reduce the analysis of properties of the algorithm to the analysis of properties of collections of intervals. The first lemma shows that the collections U of intervals recorded as the algorithms unfolds have specific structure.

Lemma 7.4.11. *Consider any moment (of the global clock) during an execution of the algorithm, and let $k_1, \ldots, k_p \geq 0$ be the numbers of the last TAS in the for loops that corresponding processors have executed by then. Then each processor i either will begin work on an interval from $U_i^{k_i}$, or is working on an interval from $U_i^{k_i}$, or will halt without doing any more TAS, or has halted. The intervals*

$U_1^0, \ldots, U_1^{k_1}$, ..., $U_p^0, \ldots, U_p^{k_p}$ are regular, and the intervals $U_h^0, \ldots, U_h^{k_h}$ are monotonic, for any $1 \leq h \leq p$.

Proof. The proof is by induction on the moments in the execution where the values of k_i's change. We shall see that the lemma holds from the moment when processors begin execution until the first time a last TAS in the for loop is executed by any processor i. Then we will consider two moments: one right before a last TAS is executed by some processor i in the for loop, and one right after the TAS. This TAS causes the value of k_i to increase by one. We will argue that if the hypothesis holds before the TAS then it holds after the TAS until the earliest moment before the next last TAS is executed by some processor.

Let us consider the base case. The lemma is true right before the last TAS is executed first by any processor. The first line of the code of procedure COLLIDE for processor i sets U to $\{[0, n-1]\}$, and the value of U is not changed at least until the processor reaches line `11` of the first iteration of the while loop. Thus $U_i^0 = \{[0, n-1]\}$, for any $1 \leq i \leq p$. At the moment right before a last TAS is executed for the first time by any processor, each processor i is either working, or will work on an interval from U_i^0. Note that collections $U_1^0, \ldots, U_p^0$ are regular as each contains the same single interval of length that is a power of two. Trivially, the collection U_i^0 is monotonic for any $1 \leq i \leq p$ because any single collection is always monotonic.

For the inductive step, suppose that $k_1, \ldots, k_p \geq 0$ are the numbers of last TAS in the for loops that corresponding processors have executed by then, and that the lemma is true right before the last TAS in the for loop is executed by a processor i during iteration number $k_i + 1$. We have two cases: either the processor succeeds on the TAS or fails.

Case 1: If the processor i succeeds on the last TAS, then it means that the processor has successfully TAS to all cells in the interval I that it has been working on. Notice that the processor never reads any memory cell that could be written to by a different processor between now and the moment when the processor reaches line `05` again in the next iteration number $k_i + 2$ of the while loop, if the processor ever reaches this line again. Therefore, the value of $U_i^{k_i+1}$ is already determined. Since the processor has been working on an interval I from $U = U_i^{k_i}$, then, in line `11`, the processor removes the interval from the collection. Let V denote the resulting collection of intervals. Clearly, the collections $U_i^0, \ldots, U_i^{k_i}, V$ are monotonic and the collections V, $U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular.

When processor i reaches line `21`, the collections U^{21}, $U_1^0, \ldots, U_1^{k_1}$, $\ldots\ldots$, $U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, U^{21}$ are monotonic. By property (i), if U^{21} contains an interval of length 1, then U^{21} contains intervals of length 1 or 2 only. Hence a possible removal of intervals of length 1 preserves regularity and monotonicity. If U^{22} is empty then processor i halts and $U_i^{k_i+1} = \emptyset$. Suppose that U^{22} is not empty. The processor will begin work on an interval from a collection as explained next. When not all tips of intervals in U^{22} are marked, then $U_i^{k_i+1} = U^{22}$ and the processor will begin work on an interval with an unmarked tip from the collection. Alternatively, the processor will begin work on an interval when all tips of intervals in U^{22} are marked. Then at least one interval in U^{22} has length 2 or more. Let R be a collection equal to U^{22} except for a longest interval replaced by its two halves (line 26). By Lemma 7.4.2, the collections R, $U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular, and, by Lemma 7.4.1, the collections $U_i^0, \ldots, U_i^{k_i}, R$ are monotonic. Then $U_i^{k_i+1} = R$. Note that at any moment until the next time a last TAS is executed by some processor, each processor $j \neq i$: will begin work on an interval from $U_j^{k_j}$, or is working on an interval from $U_j^{k_j}$, or will halt without doing any more TAS, or has halted; while the processor i : will begin work on an interval from $U_i^{k_i+1}$, or is working on an interval from $U_i^{k_i+1}$, or will halt without doing any more TAS, or has halted.

The argument presented so far ensures that the inductive step holds in the case when the processor i succeeds on the last TAS.

Case 2: Suppose that the processor i fails on a TAS to a cell x that belongs to the interval I that the processor has been working on. Then the processor reaches line 13 and, if the processor was working on I from left to right, the set D contains a prefix of I without x, or, if the processor was working from right to left, a suffix of I without x. Suppose that it was a processor j that has done a successful TAS to the cell x. This must have happened when the processor j was working on an interval J either from its left tip towards its right tip or vice versa, and so the set D' read by the processor i in line 08 of iteration $k_i + 1$ is a prefix of J that contains x, or vice versa it is a suffix of J that contains x. Note that $i \neq j$ because if a processor does a successful TAS to a cell, then it never again even attempts to do a TAS to this cell. Thus the collection U'^{13} is equal to U_j^m, for some $0 \leq m \leq k_j$.

We now consider the changes to U that can happen from line `13` to line `20`, and see how these changes affect regularity and monotonicity.

After the processor i has executed lines `13` to `15`, the desired properties hold. Indeed, by Lemma 7.4.3, the collections U^{16}, $U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, U^{16}$ are monotonic.

The execution of processor i now depends on whether i collided with j when working in the opposite direction or the same direction.

First, suppose that the processor j was working on its interval J in the opposite direction than the processor i. Then i will execute line `16`. As a result, by Lemma 7.4.5, the collections U_{16}, $U_1^0, \ldots, U_1^{k_1}, \ldots\ldots, U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, U_{16}$ are monotonic, and so the collection U^{21} is equal to U_{16}.

Second, assume that the processor j was working on J in the same direction as processor i has been on I. We now study two subcases depending on the success of i in TAS to any cell in I.

Subcase number one is when processor i has managed to successfully TAS to at least one cell in I. Then $D \neq \emptyset$ and so the processor will execute line `18`. We now argue that specific relationships must hold between D, D', and x. If i and j worked to the right, then x cannot be the second or later to the right element of J, because then the first element of J would be successfully TAS by j and so

i would have failed on a TAS to an element different than x as ensured by the fact that i has been working on consecutive elements from the interval I and that $D \neq \emptyset$. So x must be the first element of J, and so x is the smallest element of D' and $x-1$ is the largest element of D. Similarly, if i and j were working to the left, then x is the largest element of D' and $x+1$ is the smallest element of D. These relationships ensure that when i has executed line 18, by Lemma 7.4.7, collections U_{18}, $U_1^0, \ldots, U_1^{k_1}$, ……, $U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, U_{18}$ are monotonic, and so the collection U^{21} is equal to U_{18}.

Subcase number two is when processor i failed on its first TAS to a cell in I. Then $D = \emptyset$ and so the processor i will execute line 19. Since i has failed on its first TAS, then, when i works to the right, x must be the smallest element in I and x the largest element in D', or, when i works to the left, then x is the largest element in I and x the smallest element in D'. As a result, after i has executed line 19, by Lemma 7.4.9, the collections U_{19}, $U_1^0, \ldots, U_1^{k_1}$, ……, $U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, U_{19}$ are monotonic, and so the collection U^{21} is equal to U_{19}.

Let us summarize how the changes to U done by processor i from line 13 to line 20 can affect regularity and monotonicity right before i reaches line 21. By the above argument, when the processor i fails on the TAS, we know that collections U^{21}, $U_1^0, \ldots, U_1^{k_1}$, ……, $U_p^0, \ldots, U_p^{k_p}$ are regular, and the collections $U_i^0, \ldots, U_i^{k_i}, U^{21}$ are monotonic. We can carry out the same analysis as in the case of a successful last TAS described before in Case 1, to show that any processor either has halted, will halt, is working, or will begin work, at any moment until the next time some k_g is increased, and that the collections are regular and monotonic as desired.

This completes the inductive step, and the proof. □

Corollary 7.4.12. *Consider any moment during an execution of the algorithm and let $k_1, \ldots, k_p \geq 0$ be the numbers of last TAS in the for loops that corresponding processors have executed by then. Then the cells $\{0, \ldots, n-1\} \setminus U_i^{k_i}$ of the array w have been set to 1, for any i. When processor halts then each cell of w has been set to 1.*

Proof. The claim that any cell $x \in \{0, \ldots, n-1\} \setminus U_i^{k_i}$ has been set to 1 can be shown in a straightforward inductive way similar to that in Lemma 7.4.11. For the second part observe that a processor halts only when its U is empty. □

The next lemma shows that any processor has at most $p/2$ intervals in its collection U at any time during execution of the algorithm. The key observation that allows to prove the lemma is that no processor can split intervals into halves so many times that the number of intervals becomes large. This is achieved by realizing that if a processor has too many intervals then it stops splitting them, as the processor must collide with some other processor that has done a part of what the former processor believes that remains to be done.

Lemma 7.4.13. *Consider any moment during an execution of the algorithm and let $k_1, \ldots, k_p \geq 0$ be the numbers of last TAS in the for loops that corresponding* *processors have executed by then. Then for each processor i, the collection $U_i^{k_i}$ has at most $p/2$ intervals. None of the collections can have p marked tips.*

Proof. The proof is by induction on the moments in the execution when the values of k_i's change as in Lemma 7.4.11.

For the base case, we observe that the lemma is true right before the last TAS is executed first by any processor, because $U_1^0 = \ldots = U_p^0$ are collections each containing just one interval $[0, n-1]$ with no marked tips.

For the inductive step, suppose that $k_1, \ldots, k_p \geq 0$ are the numbers of last TAS in the for loop that corresponding processors have executed by then, and that each of the collections $U_1^{k_1}, \ldots, U_p^{k_p}$ has at most $p/2$ intervals right before a last TAS in the for loop is executed by a processor i for the $(k_i + 1)$-th time during the iteration number $k_i + 1$ of the while loop. Right before the TAS, the value of the variable U that the processor i has is equal to $U_i^{k_i}$. If the TAS is successful, then when the processor reaches line `21` then the number of intervals in U^{21} is one less than in $U_i^{k_i}$, and so it is at most $p/2$. If the TAS failed then the processor must have collided with a different processor j and so the value of the variable U'^{13} of processor i is equal to U_j^m, for some $0 \leq m \leq k_j$. By the inductive hypothesis the number of intervals in U'^{13} is at most $p/2$. After the processor i has executed lines `13` to `20`, the number of intervals in U can be at most the maximum of the number of intervals in $U_i^{k_i}$ and U_j^m, because of Lemma 7.4.3, Lemma 7.4.5, Lemma 7.4.7, and Lemma 7.4.9. And so U^{21} has at most $p/2$ intervals.

We now evaluate U until the processor reaches line 05 again, if ever. We begin the evaluation with three simple cases. Recall that removing entire intervals from a collection preserves regularity and monotonicity. If U_{21} is empty, then the result is trivial because $U_i^{k_i+1}$ is empty. Second, if the U_{21} has an interval with an unmarked tip, then processor i does not execute lines 24 to 26 and so $U_i^{k_i+1}$ is equal to U_{21}. Third, if U_{21} is not empty, all tips of all intervals in U_{21} are marked, and U_{21} has strictly fewer than $p/2$ intervals, then the processor i executes lines 24 to 26. But then the interval $[a, b]$ selected in line 24 has length at least 2 and after its split there are exactly two intervals in U^{26} that have unmarked tips. Consequently, the $U_i^{k_i+1}$ in the third case has one more interval compared to U_{21}, and so the number of intervals is bounded by $p/2$. In all these three cases the result follows.

The final case in the inductive step is when the collection U_{21} has exactly $p/2$ intervals and all their tips are marked. We show that this cannot happen. Let the collection U_{21} be composed of the intervals $I_1, \ldots, I_{p/2}$. Since all intervals of length 1 with marked tips were removed in line 21, each of the $p/2$ intervals has length at least 2. Thus the intervals have exactly p distinct tips and the tips are marked. Note that if a processor i had done a successful TAS to a tip of its interval during the for loop in a k-th iteration of the while loop, $k \leq k_i + 1$, then this tip (not necessarily entire interval) would have been removed from its collection before the processor i reaches line 22 of the while loop in iteration number k, and the subsequent collection U_i^{k+1}, if any, does not include the tip, and, by the monotonicity property (iv), no subsequent collections until $U_i^{k_i}$ can contain the tip. Therefore, when the processor i reaches line 22 of the current iteration number k_i+1, all p distinct tips of the intervals $I_1, \ldots, I_{p/2}$ have been successfully TAS by processors other than i. But then, by the pigeonhole principle, there is a processor j other than i such that the processor j had successfully TAS to two of the p tips before the processor i did the last TAS in the for loop of iteration number k_i+1 of the while loop. Let x be the tip TAS by j first according to the total order established by the TAS instructions, and y the other tip (TAS later). Let $j_x \leq j_y$ be the iterations numbers of the while loop of processor j during which the processor did a successful TAS to the two tips x and y respectively. We consider two cases depending on whether x and y were TAS in the same or different iteration of the while loop.

For the first case, suppose that x was TAS in an earlier iteration i.e., that $j_x < j_y$. Then processor j must have removed the cell x from its set U by the end of the iteration j_x and so all subsequent collections starting from $U_j^{j_x}$ until $U_j^{k_j}$ inclusive do not contain any interval that contains x. During an iteration number $i_y \leq k_i+1$, the processor i fails on TAS to the cell y and executes lines 13 to 15. At that time no interval of the collection U' of processor i contains an interval with element x. Hence by monotonicity, the interval U_{21} of the iteration number k_i+1 of processor i does not contain any interval that contains x. This is a desired contradiction because we assumed that x is a marked tip of an interval in the collection U_{21}.

Finally, assume that x and y were TAS during the same iteration of the while loop i.e., that $j_x = j_y$. Then, when i reaches line `13` during iteration i_y, the interval D' contains two distinct elements x and y. It cannot be the case that i managed to successfully TAS to a cell in the for loop while working in the same direction as j did when j was doing TAS to y. Hence i executes either line `16` or line `19` in which case at least one x or y is removed from U, together with some interval or its part, before i reaches line `22` of iteration i_y. Again, this leads to a contradiction.

The inductive step is completed, which proves the lemma. □

During an execution a processor performs some number of iterations of the while loop. The next lemma shows an upper bound on this number. The lemma is proven by noticing that the sequence $U_i^0, U_i^1, U_i^2, \ldots$ cannot have too long subsequences of equal collections because the number of marked tips would increase, and that when two subsequent collections are different then the processor makes substantial progress on its work. This means that the successive collections quickly become "slimmer and slimmer", and eventually become empty.

Lemma 7.4.14. *In any execution of the algorithm any processor i performs at most $p^2 + p\,(2p+1)\log n$ iterations of the while loop.*

Proof. Let us consider any execution, any processor i, and the sequence of collections $U_i^0, U_i^1, U_i^2, \ldots$ that the processor recorded. We begin with an observation about the structure of the sequence. Firstly, if two consecutive collections are equal, then there is one tip that is unmarked in the previous collection and it is marked in the subsequent collection, while all already marked tips remain marked. Indeed, suppose that $U_i^k = U_i^{k+1}$. Note that when a processor is working on an interval then the tip from which the processor has started the work is unmarked at the time when the work begins. Since the two collections are equal, during iteration $k+1$ of the while loop of processor i, the value of the collection U stays intact from the moment right before line `11` until right after line `28`. In order for this value to be unchanged, the processor must have failed on its first TAS to a cell x in the for loop in the iteration, and the failure must have been because a different processor had successfully TASed to x while it had been working on an interval in the same direction as i was (otherwise the value of U would change). As a result, a previously unmarked tip x of an interval from U becomes marked in line `20`. Note that no tip of any interval in a collection is ever unmarked by a processor, unless a part of the interval that contains x is removed from a collection. This leads to an observation that we can bound the length of a sequence of consecutive equal collections. Suppose that for some k and $c \geq 0$ we have $U_i^k = U_i^{k+1} = \ldots = U_i^{k+c}$. By Lemma 7.4.13, the collection U_i^k has at most $p/2$ intervals and so at most p unmarked tips. Because each subsequent collection in the sequence has one less unmarked tip and, by Lemma 7.4.13, no collection can have p marked tips, c can be at most $p-1$.

We now look into what must happen when two consecutive collections are different. Take any k and let c be the largest number so that $U_i^k = U_i^{k+1} = \ldots =$

U_i^{k+c}, and suppose that a collection U_i^{k+c+1} is recorded by the processor i in the execution. Let us now investigate what the difference is between the collections U_i^{k+c} and U_i^{k+c+1}. We want to show that this difference is "big". The value of U^{11} in iteration $k+c+1$ is equal to U_i^{k+c}, while the value of U_{28} is equal to U_i^{k+c+1}. We consider all the ways in which the execution of the processor i may proceed from line `11` until line `28` of iteration $k+c+1$.

If the processor succeeded on the TAS, then it executes line `11` where an interval is removed from U. By property (i), this interval has length that is equal to at least half the length of a longest interval in U_i^{k+c}. Due to monotonicity of subsequent operations in the iteration, the intervals in U_i^{k+c+1} combined have fewer elements than the intervals in U_i^{k+c} combined by least half the length of a longest interval in U_i^{k+c}.

Suppose on the other hand that the processor failed on the TAS, and so it executes line `13`. If the execution of lines `13` to `15` leads to a change in the value of U, then, by Corollary 7.4.4, either the length of a longest interval in U_{15} is smaller or equal to half the length of a longest interval in U_i^{k+c}, or the intervals in U_{15} combined have fewer elements compared to U_i^{k+c} by at least half of the length of a longest interval in U_i^{k+c}. Again, due to monotonicity, this relative difference carries over to the difference between U_i^{k+c} and U_i^{k+c+1}.

Processor i could fail on the TAS and the execution of lines `13` to `15` does not change U. Then $U^{16} = U_i^{k+c}$. Let $j \neq i$ be the processor that successfully TAS to the cell x while working on an interval. Now we have two cases: either the processor j (with which i collided) worked in the same direction as i did or they worked in different directions.

If the colliding processors i and j worked in different directions, then, by Corollary 7.4.6, the intervals in U_{16} combined have fewer elements compared to U_i^{k+c} by at least half of the length of a longest interval in U_i^{k+c}. Again, due to monotonicity, the intervals U_i^{k+c+1} can only have even fewer elements. Suppose that i and j worked in the same direction. Then we have that either the lines 18 to 19 changed the value of U or not. When $U_{19} \neq U^{18}$ then, by Corollary 7.4.8 and Corollary 7.4.10, the intervals in U^{20} combined have fewer elements compared to U_i^{k+c} by at least half of the length of a longest interval in U_i^{k+c}. If $U_{19} = U^{18}$ then we have that $U^{21} = U_i^{k+c}$. Consequently, if any interval of length 1 is removed in line 21, then, by property (i), this interval is at least as long as half of the length of a longest interval in U_i^{k+c}. Finally, assume that no interval of length 1 is removed. Since U_i^{k+c+1} is recorded, then i does not halt in line 22, and so $U^{23} = U_i^{k+c}$. Then it cannot be the case that there is an unmarked tip, because we assumed that $U_i^{k+c} \neq U_i^{k+c+1}$. Hence a longest interval in U^{23} gets split, and so U_i^{k+c+1} is equal to U_i^{k+c} except for a longest interval split into two halves.

Let us sum up the above study of the execution of the processor i from line 11 to line 28. When the collection U_i^{k+c+1} is different than the collection U_i^{k+c} then: either the length of a longest interval in U_i^{k+c+1} is smaller or equal to half the length of a longest interval in U_i^{k+c}, or U_i^{k+c+1} is equal to U_i^{k+c} except for a longest interval being split into two halves, or the intervals in U_i^{k+c+1} combined have fewer elements compared to the intervals in U_i^{k+c} at least by half of the length of a longest interval in U_i^{k+c}.

We now count how many times each of these three events can happen until U becomes empty (in which case processor i must halt). Reduction of length by the factor of 2 or more can happen at most $\log n$ times, the longest interval can be split into two halves at most $p \log n$ times (because the number of intervals in a collection is bounded by $p/2$), and at least a half of a longest interval can be removed at most $p\,(1 + \log n)$ times. As a result $U_i^{p(\log n + p \log n + p(1+\log n))} = \emptyset$, and the result follows. □

We are now ready to prove the main result of this section.

Theorem 7.4.15. *The algorithm solves the Certified Write-All problem and has work of $O(n + p^4 \log n)$; the combined number of cells of w that the processors set to* 1 *is at most $n + 4p^3 \log n$.*

Proof. Consider any execution of the algorithm. By Lemma 7.4.13 each processor performs a bounded number of iterations, and, by Corollary 7.4.12, when a processor stops all cells of w have been set to 1. Hence the algorithm solves the CWA problem.

We now argue about the work complexity of the algorithm. Let us fix a processor and divide each of its iterations of the while loop into two parts. The first part contains the instructions starting from the first TAS of the for loop until but not including the last TAS of the for loop, while the second part contains all other instructions in the iteration (the second part has two discontinuous sections of instructions). Note that all TAS in the first part are successful and so the combined work of all processors on the first parts is $O(n)$ (using a pointer representation of the tuples) and at most n cells of the array w are set to 1 by the p processors during the first parts. For any processor, the number of second parts is equal to the number of iterations, which is bounded, by Lemma 7.4.14, by $p^2 + p\,(2p+1)\log n$. During each second part, at most one cell of w can be set to 1, and so the p processors combined may set to 1 at most $p^3 + p^2\,(2p+1)\log n$ cells of w in their second parts. Recall that, by Lemma 7.4.13, the number of intervals in any collection during the execution is at most $p/2$ and so any second part takes $O(p)$ instructions to execute. Thus the combined work that processors performed on the second parts is $O(p^4 \log n)$. This completes the proof. □

Theorem 7.4.16. *The combined size of shared memory used by the processors is $O(n + p^4 \log n)$.*

Proof. Note that instead of a tuple being TAS to locations of the array *tab*, each processor can store a pointer to a tuple during any TAS operation. This tuple can contain D and a pointer to U. Each processor generates as most $O(p^2 \log n)$ distinct U until it halts, and each U has size $O(p)$. □

7.4.4 Lower bound

We show a lower bound on work of any deterministic algorithm that solves the CWA problem using Test-And-Set. When $p = n$, the bound is stronger than the one presented in the Theorem 3.5 of Buss et al. [24]. Our lower bound generalizes techniques shown by Buss et al. in Theorem 3.1 and Theorem 3.5. The main idea of the proof of our lower bound is that even if processors use the Test-And-Set synchronization primitive, whenever a cell of the array w is 0, one of the relatively fast processors has to eventually attempt to set the cell to 1, or a fast processor performs arbitrary large amount of work. In the former case, adversary can therefore stop processors from setting cells to 1, until large enough number of processors try to set a cell to 1, and then the adversary can let only this large number of processors set the cell to 1, thus ensuring small progress, but high work.

Theorem 7.4.17. *For any* $3 \leq p \leq n$*, work of any deterministic algorithm that solves the Certified Write-All problem and that uses Test-And-Set must be* $\Omega(n + p \log p)$*.*

Proof. We shall design an execution of any deterministic algorithm that solves the CWA problem, where processors will perform at least $c(n + p \ln p)$ basic actions, where c will be an absolute constant that does not depend on n nor on p.

We schedule work of processors arbitrarily until there are exactly p cells of the array w that remain to be set to 1. The rest of the execution will be divided into phases numbered from p down to 1, so that at the beginning of phase u, exactly u cells of w remain to be set to 1. We now define the execution inside a phase u. The phase will consist of rounds. In any round we sequentially execute: a basic action of processor 1, then of processor 2, and so on until we execute a basic action of processor p, unless a processor attempts to write 1 (possibly using Test-And-Set) to a cell of the array w that is still 0, in which case we stop the processor before the attempt, if it is not on hold already (we can detect if a processor will or will not write 1 next, because the algorithm is deterministic). We schedule work of processors in rounds in the way just described, and eventually we either: (i) reach a round $k_u < n + p \ln p$ at the end of which each processor is on hold and about to write 1 to a cell of the array w that is still 0, or (ii) there are at least $n + p \ln p$ rounds in the phase. In the second case the result is trivial, because a processor has performed $n + p \ln p$ basic actions in the phase, and so let us assume that this case does not happen during any execution. At the end of round $k_u < n + p \ln p$, there is a cell c_u of the array w to which $\lceil p/u \rceil$ processors are about to write 1, and this cell is still 0. We let these processors perform the write, one after another, while all processors that are about to write to a cell other than c_u are still kept on hold. The execution of the phase so far results in at least p/u basic actions being executed, and exactly one cell of the array w being changed from 0 to 1 (cells of shared memory other than w could have been modified as well during the phase). At this moment, the phase u is over. We proceed scheduling work of processors like this, phase after phase, until all p phases are over. When p phases are over, the number of basic actions performed in the p phases is at least $p/p + p/(p-1) + \ldots + p/1 = pH_p > p \ln p$. Therefore, work must be at least

$n - p + p \ln p$. Finally, we observe that when $p \geq 3$, $-1 + \ln p > 0.05 \ln p$, and so the result follows by setting c to 0.05. □

The above lower bound is stronger than the one shown by Buss et al. [24] in Theorem 3.5 in the case when $p = n$. Indeed Buss et al. shows that there is a constant $c > 0$, such that work of any deterministic algorithm that solves Write-All for $p \geq 3$ processors and $n \geq 1$ cells using Test-And-Set is bounded from below by expression $n + cp \ln(n/2p)$. We will see that this expression is $O(n)$. There are two cases. The first case is when $2p > n$. This means that $\ln(n/2p)$ is negative, and so $n + cp \ln(n/2p) < n$. The second case is when $2p \leq n$. Then $n/2p \geq 1$, and so $\ln(n/2p) \leq n/2p$. Thus $p \ln(n/2p) \leq n/2$. Combining the two cases ensures that $n + cp \ln(n/2p) \leq 3/2n$, for any $p \geq 3$ and $n \geq 1$.

7.5 Conjecture on how to efficiently construct low-contention permutations

This section presents a conjecture and supporting analysis indicating that it may be possible to efficiently construct a list of q permutations on $[q]$ such that the list has contention $O(q \log^2 q)$.

The algorithm AWT of Anderson and Woll uses lists of q permutations on $[q]$. We have seen that contention of a list used to instantiate the AWT algorithm affects work of the instance. Specifically, it is desirable to produce lists with as low contention as possible, because for any fixed number of processors n and permutations q, work of an instance that uses a list decreases when contention of the list decreases. It is know that there exist lists with contention $O(q \log q)$ but no known polynomial time in q construction of such lists exists. Kanellakis and Shvartsman proposed [85] a way of creating a list of q permutations on $[q]$ in time polynomial in q. The authors conjectured that the lists have contention $O(q \log q)$ but they provided no analysis. This section shows a partial analysis of contention of the lists of permutations proposed by Kanellakis and Shvartsman. Our result is hybrid as it includes analytical and empirical parts. The analytical result shows that contention of the list with respect to a subset of permutations on $[q]$ is $\Theta(q \log^2 q)$. The empirical results provide evidence that our analysis covers the worst case scenario. Therefore, we conjecture that the list of q permutations on $[q]$ of Kanellakis and Shvartsman has contention $\Theta(q \log^2 q)$.

7.5.1 Permutations of Kanellakis and Shvartsman

Definition 7.5.1. *For any $q > 0$ such that $q + 1$ is prime, we define $D_q = \langle \delta_1, \ldots, \delta_q \rangle$ to be a list of q permutations $\delta^j(i) = ji \text{ mod } (q + 1)$, $i = 1, \ldots, q$.*

7.5.2 Analytical bound

In this section, we show a bound on contention of D_q with respect to permutations from a subset of all possible permutations on $[q]$. We start by showing an upper bound on the number of left-to-right maxima for D_q.

Let $L(q,i)$ denote the number of left-to-right maxima in the permutation δ_i of D_q. Let $L(q)$ be the number of left to right maxima in all the permutations of D_q, i.e., $L(q) = \sum_{i=1}^{q} L(q,i)$. The next result shows the growth of $L(q)$.

Theorem 7.5.1 ([27]). *For any $q > 0$ such that $q+1$ is a prime number,*

$$L(q) = \frac{3}{\pi^2}(q+1)(\ln(q+1))^2 + O(q \ln q (\ln\ln q)^2).$$

Theorem 7.5.1 gives a bound on contention of D_q with respect to the identity permutation. The next theorem shows that the same bound applies to a subset of permutations on $[q]$.

Theorem 7.5.2. *For any prime $q+1$, contention of D_q with respect to any $\delta \in D_q$, is $Cont(D_q, \delta) = O(q \log^2 q)$.*

Proof. Lemma follows from the fact that D_q is a subgroup of the symmetric group of all permutations on $[q]$, and from Theorem 7.5.1. □

7.5.3 The conjecture and experimental support

We conduct an experimental study that leads us to a conjecture that $L(q)$ bounds contention of D_q from above.

We present the results of an exhaustive study where we evaluate contention of D_q with respect to all permutations on $[q]$. The exhaustive search is performed for each prime: $2, 3, 5, 7, 11, 13$. For a given prime $q+1$, we calculate contention of D_q with respect to all permutations on $[q]$. Contention of D_q with respect to any permutation is *never* greater than $L(q)$ – the contention with respect to the identity permutation, see Table 1. For some permutations contention is equal to $L(q)$. The number of such permutations, $|\{\delta \text{ on } [q] : Cont(D_q, \delta) = L(q)\}|$, is given in the fourth column of the table; the last column shows the ratio of the number to $q!$. Observe that the number of such permutations has to be at least q (cf. Theorem 7.5.2).

$q+1$	$q!$	$L(q)$	equal to $L(q)$	ratio
2	1	1	1	1
3	2	3	2	1
5	24	9	8	0.33
7	720	17	12	0.016
11	3628800	37	60	0.000016
13	479001600	49	48	0.0000001
17	20922789888000	75	Still running	Unknown

Table 1: Summary of exhaustive search.

In the exhaustive search, contention of D_q with respect to any permutation is at most $L(q)$, and the ratio of the number of permutations that yield contention $L(q)$ to $q!$ diminishes as q grows. These observations lead us to the following conjecture.

Conjecture 7.5.1. *WISH:* $Cont(D_q) = L(q) = O(q \log^2 q)$.

```
shared variables: $f$, array $w[0,\dots,n-1]$, array $tab[0,\dots,n-1]$ of $\{L,R\}\times 2^{2^{\mathbb{N}}}\times 2^{\mathbb{N}}$;
initially $f=0$, $w[x]=0$, $tab[x]=\langle 0,\emptyset,\emptyset\rangle$, for $x=0,\dots,n-1$

COLLIDE
01 local variables: $dir, U, D, dir', U', D', c, s, e, x, failed$
02 $U := \{[0,n-1]\}$, tips are unmarked unless explicitly marked
03 $s := 0$; $e := n-1$; $dir := R$
04 while true
05     $D := \emptyset$
06     for $x := s$ to/downto $e$
07         $w[x] := 1$ ; TAS($x$)
08         $\langle dir', U', D'\rangle := tab[x]$
09         if $failed$ then goto 11
10         $D := D \cup \{x\}$
11     if not $failed$ then $U := U \setminus D$
12     else
13         if $maxlen\,(U) > maxlen\,(U')$ then $U := U' \cap U$
14         else
15             $U := U \cap U'$
16         if $dir \neq dir'$ then $U := U \setminus (D \cup D')$
17         else
18             if $D \neq \emptyset$ then $U := U \setminus D$
19             else $U := U \setminus (D' \setminus \{x\})$
20             mark tip $x$ of the interval in $U$ that contains $x$
21     remove from $U$ any interval of length 1 with a marked tip
22     if $U$ is empty then set $f$ to 1 and Halt
23     if tips of all intervals are marked then
24         let $[a,b]$ be an interval among the longest ones in $U$
25         $c := a + \frac{b-a+1}{2} - 1$
26         $U := (U \setminus \{[a,b]\}) \cup \{[a,c],[c+1,b]\}$
27     let $s$ be an unmarked tip of an interval from $U$ and $e$ the other tip
28     if $s \leq e$ then $dir := R$ else $dir := L$

TAS($x$)
01     begin atomic
02         if $tab[x].D$ is $\emptyset$ then
03             $failed := false$
04             $tab[x] := \langle dir, U, D \cup \{x\}\rangle$
05         else $failed := true$
06     end atomic
```

Figure 4: Deterministic algorithm for the Certified Write-All problem for an asynchronous shared memory machine as executed by any processor.

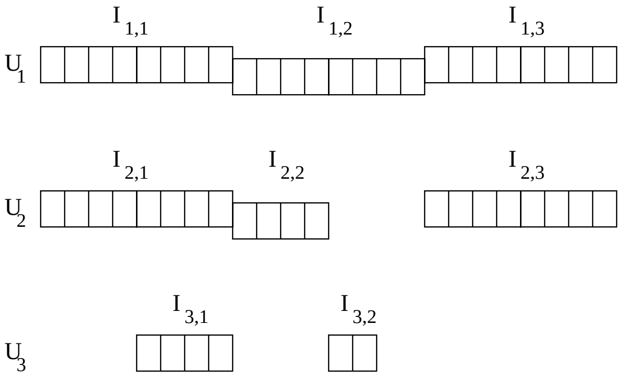

Figure 5: This figure illustrates the definition of regular and monotonic collections of intervals. In the example above, we have three collections U_1, U_2, and U_3 of intervals of cells. These collections are regular for the following reasons. Any collection has intervals which lengths are powers of two, and which differ by factor of 2. Any two intervals from any two distinct collections are either disjoint, or one is contained in the other. If one is contained in the other, then the subset must be properly aligned i.e., the subset must be equal to one interval among a power of two equi-length intervals that partition the superset. Note that collections U_1, U_2 are monotonic, because for each interval of U_2 there is a superset interval in U_1. On the other hand, collections U_2, U_3 are not monotonic, because there is no interval in U_2 that contains interval $I_{3,2}$.

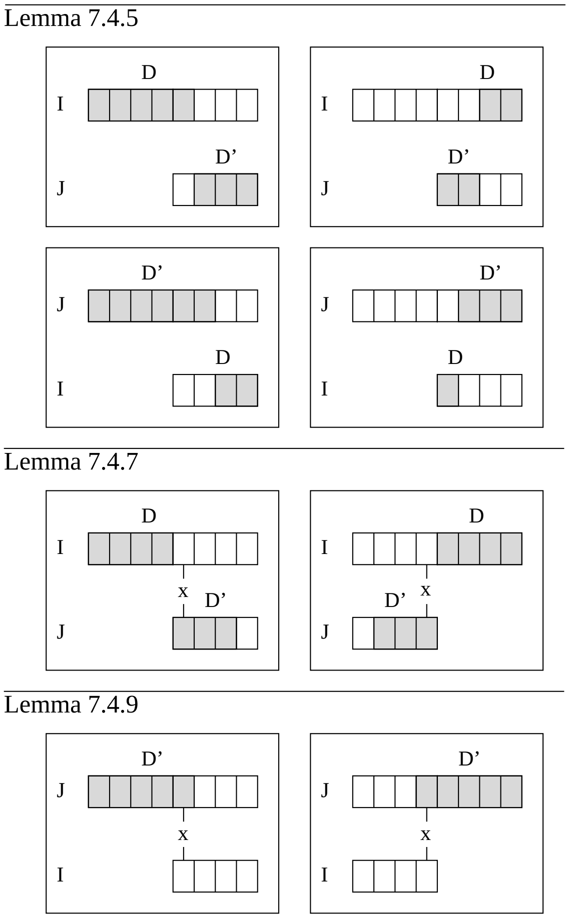

Figure 6: This figure illustrates the assumptions of Lemmas 7.4.5, 7.4.7, and 7.4.9.

Chapter 8

Conclusions and future work

The thesis presented how distributed devices that are disconnected for long and unknown periods can efficiently perform a set of tasks. Specifically, the thesis considered n distributed devices that must perform t independent tasks, known to each device. The goal was to schedule work of the devices locally, in the absence of communication, so that when communication is established between some devices at some later point of time, the devices that connect have performed few tasks redundantly beyond necessity. The thesis showed a lower bound of $a^2/(t-b+a) \cdot (1-(t-b)/(an-a))$ on the number of tasks wastefully performed by at least two devices when they establish communication after having performed $a \leq b$ tasks. The bound shows that wasted work must increase as the devices progress in their work. The thesis showed deterministic and randomized constructions that will allow all devices to avoid wasting work nearly optimally. Specifically, for any group of h devices, when each of them has performed a tasks at the moment when they establish communication, worst-case wasted work of the devices in the group is at most $\binom{h}{2} \cdot a^2/t \cdot (1 + O(t^{-\epsilon} + h^{-1}))$. One of the insights that this thesis offers is that when any disconnected device randomly selects its next task, from among the tasks remaining to be done, then this device will avoid duplicating the work of other devices quite well. The thesis uses, and is related to, several mathematical theories. Techniques from design theory, linear algebra, and graph theory have been applied to derive deterministic constructions. The distributed scheduling problem studied in the thesis is related to the sphere packing problem. The solutions give rise to a construction of latin squares with specific uniformity properties. The lower bound generalizes the Second Johnson Bound of coding theory.

In addition, the thesis studied scheduling algorithms for shared memory systems. The thesis investigated algorithms for the Certified Write-All problem. The thesis demonstrated how to create near-optimal instances of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll, and that the choice of the number q of permutations used by instances of the AWT algorithm is critical for obtaining an instance of the AWT algorithm with near-optimal work. The thesis also showed a work-optimal deterministic algorithm for the asynchronous Certified Write-All problem for a nontrivial number of processors $p \leq (n/\log n)^{1/4}$.

There are several follow-up directions on the work presented in the thesis. Our research presented in Chapters 4,5 and 6 makes some assumptions that may not be valid in practice, and it would be interesting to research how to remove these assumptions. Specifically, the techniques presented in the thesis assume that each device knows all tasks at the beginning of computation. This may be unrealistic, for example, when the description of all tasks takes substantial amount of space and consequently it may be too costly to distribute all tasks to each processor at the beginning of computation. Hence, it would be interesting to look into techniques of efficient distribution of tasks to processors. In addition, the thesis assumed that processors may establish communication after having performed an arbitrary number of tasks (total asynchrony). In practice it may be the case that the number of tasks performed by reconnecting processors could be approximated well by a "well-behaved" random variable. This information seems to be important for mechanisms for distribution of tasks.

It seems that it is possible to reduce the space complexity of the algorithm presented in Section 7.4.1 by modifying data structures. Specifically, instead of creating a new U during each iteration, one could try to reuse the parts of U that have not changed since the prior iteration. This should decrease space complexity, but may increase work complexity as the new representation of U may be more "dispersed". It should be possible to tighten the analysis of work complexity by further exploring the information flow between processors. It seems possible that the actual work complexity of the algorithm is $O(t + p^3 \log t)$.

There are several research directions following the results presented in Section 7.3.2. Any AWT algorithm has a progress tree with internal nodes of fanout

q. One could consider generalized AWT algorithms where fanout does not need to be uniform. Suppose that a processor that visits a node of height i, uses a collection $R^i_{q(i)}$ of $q(i)$ permutations on $[q(i)]$. Now one could choose different values of $q(i)$ for different heights i. Does this technique enable any improvement of work as compared to the case when $q = q(1) = \ldots = q(h)$? What are the best values for $q(1), \ldots, q(h)$ as a function of n? Suppose that we are given a relative cost κ of performing a write to the cell of the array w, compared to the cost of executing any other basic action. What is the shape of the progress tree that minimizes work? These questions give rise to more complex optimization problems, which would be interesting to solve.

Bibliography

[1] Abramson, D., Sosic, R., Giddy, J., Hall, B.: Nimrod: a tool for performing parametrised simulations using distributed workstations. Fourth IEEE International Symposium on High Performance Distributed Computing (HPDC'95) (1995) 112–121

[2] Alon, N., Spencer, J. H.: The probabilistic method. Second edition. John Wiley & Sons Inc., New York (2000). With an appendix by Paul Erdős, A Wiley-Interscience Publication

[3] Anderson, R.J., Woll, H.: Wait-free Parallel Algorithms for the Union-Find Problem. Extended version of the STOC'91 paper of the authors, November 1 (1994)

[4] Anderson, R.J., Woll, H.: Algorithms for the Certified Write-All Problem. SIAM Journal on Computing, Vol.26 (5) (1997) 1277–1283

[5] Attiya, H., Welch, J.: Distributed Computing, Fundamentals, Simulations, and Advanced Topics. McGraw-Hill Publishing Company (1998)

[6] Aumann, Y., Bender, M.A., Zhang, L.: Efficient execution of nondeterministic parallel programs on asynchronous systems. Information and Computation, Vol. 139(1) (1997) 1–16 (Preliminary version: SPAA'96)

[7] Aumann, Y., Kedem, Z.M., Palem, K.V., Rabin, M.O.: Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs. 34th IEEE Symposium on Foundations of Computer Science (1993) 271–280

[8] Aumann, Y., Rabin, M.O.: Clock construction in fully asynchronous parallel systems and PRAM simulation. Theoretical Computer Science, Vol. 128 (1994) 3–30 (Preliminary version: FOCS'92)

[9] Awerbuch, B., Azar, Y.: Local Optimization of Global Objectives: Competitive Distributed Deadlock Resolution and Resource Allocation. 35th IEEE Symposium on Foundations of Computer Science (1994) 240–249

[10] Azuma, K.: Weighted sums of certain dependent random variables. Tôhoku Math. J. Vol. 2(19) (1967) 357–367

[11] Babaoglu, O., Davoli, R., Giachini, L., Baker, M.: Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Proc. of Hawaii International Conference on Computer and System Science, Vol. 2 (1995) 612–621

[12] Bartal, Y., Byers, J., Raz, D.: Fast, Distributed Approximation Algorithms for Positive Linear Programming with Applications to Flow Control. IEEE Symposium on Foundations of Computer Science (1997) 303–312

[13] Baratloo, A., Dasgupta, P., Kedem, Z.M.: CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. 4th International Symposium on High Performance Distributed Computing HPDC'95, (1995) 122–129

[14] Bermond, J.C., Bond, J., Sotteau, D.: On regular packings and coverings, in Combinatorial Design Theory. (Colbourn, C.J., Mathon, R.A. eds.) Annals of Discrete Mathematics, Vol. 34 (1987) 81–100

[15] Bernstein, A.J.: Program Analysis for Parallel Processing. IEEE Transactions on Electronic Computers, EC-15(5) (1966) 757–762

[16] Beutelspacher, A., Rosenbaum, U.: Projective Geometry: From Foundations to Applications. Cambridge University Press (1998)

[17] Birman, K.P.: The process group approach to reliable distributed computing. Communications of the ACM, Vol. 36(12) (1993) 37–53

[18] Birman, K.P., van Renesse, R.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, Los Alamitos, CA (1994)

[19] Blumofe, R.D., Leiserson, C.E.: Space-Efficient Scheduling of Multithreaded Computations. SIAM Journal on Computing, Vol. 27(1) (1998) 202–229

[20] Brightwell, G., Ott, T.J., Winkler, P.: Target shooting with programmed random variables. Annals of Applied Probability, Vol. 3(5) (1995) 834–853

[21] Brouwer, A.E.: Block Designs, in Handbook of Combinatorics. (Graham, R., Grotschel, M., Lovász, L. eds.) Elsevier (1995)

[22] Brunett, S., Czajkowski, K., Fitzgerald, S., Foster, I., Johnson, A., Kesselman, C., Leigh, J., Tuecke, S.: Application Experiences with the Globus Toolkit. 7th IEEE Symposium on High Performance Distributed Computing (1998) 81–91

[23] Buhler, J.P., Lenstra, H.W., Pomerance, C.: The development of the number field sieve. Vol. 1554 of Lecture Notes in Computer Science, Springer-Verlag (1994)

[24] Buss, J., Kanellakis, P.C., Ragde, P.L., Shvartsman, A.A.: Parallel Algorithms with Processor Failures and Delays. Journal of Algorithms, Vol. 20 (1996) 45–86 (Preliminary versions: PODC'91 and Manuscript'90)

[25] Chang, H., Tait, C., Cohen, N., Shapiro, M., Mastrianni, S., Floyd, R., Housel, B., Lindquist, D.: Web browsing in a wireless environment: disconnected and asynchronous operation in ARTour Web Express. Proceedings of the third annual ACM/IEEE international conference on Mobile computing and networking (1997) 260–269

[26] Chlebus, B.S., De Prisco, R., Shvartsman, A.A.: Performing Tasks on Restartable Message-Passing Processors. Distributed Computing, Vol. 14(1) (2001) 49–64

[27] Chlebus, B., Dobrev, S., Kowalski, D., Malewicz, G., Shvartsman, A., Vrto, I.: Towards Practical Deterministic Write-All Algorithms. 13th Symposium on Parallel Algorithms and Architectures SPAA'01, (2001) 271–280

[28] Chlebus, B.S., Gambin, A., Indyk, P.: PRAM computations resilient to memory faults. 2nd European Symposium on Algorithms ESA'94, (1994) 401–412

[29] Chlebus, B., Kowalski, D.: Randomization Helps to Perform Tasks on Processors Prone to Failure. 13th Int. Symposium on Distributed Computing, LNCS 1693 (1999) 284–296

[30] Chlebus, B., Kowalski, D., Lingas, A.: The Do-All Problem in Broadcast Networks. 20th ACM Symposium on Principles of Distributed

Computing (2001) 117–126

[31] Chockler, G.V., Keidar, I., Vitenberg, R.: Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys, Vol. 33(4) (2001) 1–43

[32] Colbourn, C.J., Dinitz, J.H., Stinson, D.R.: Applications of combinatorial designs to communications, cryptography, and networking. Surveys in Combinatorics (J.D. Lamb and D.A. Preece, eds.), Cambridge University Press (1999) 37–100

[33] Colbourn, C. J., van Oorschot, P. C.: Applications of Combinatorial Designs in Computer Science. ACM Computing Surveys, Vol. 21(2) (1989) 223–250

[34] Cole, R., Zajicek, O.: The APRAM: Incorporating Asynchrony into the PRAM Model. 2nd ACM Symposium on Parallel Algorithms and Architectures SPAA'89, (1989) 169–178

[35] Cole, R., Zajicek, O.: The Expected Advantage of Asynchrony. 3rd ACM Symposium on Parallel Algorithms and Architectures SPAA'90, (1990) 85–94

[36] Cole, R., Hopcroft, J.: On edge coloring bipartite graphs. SIAM Journal on Computing, Vol. 11(3) (1982) 540–546

[37] Communications of the ACM, Special Issue on Group Communication Services. Vol. 39(4) (1996)

[38] Conway, J.H., Sloane, N.J.A.: Sphere Packings, Lattices and Groups. Third Edition. Springer-Verlag, Series: Grundlehren der mathematischen Wissenschaften, Vol. 290 (1998)

[39] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., van Eicken, T.: LogP: Towards a Realistic Model of Parallel Computation. 4th ACM Principles and Practices of Parallel Programming (1993) 1–12

[40] Dasgupta, P., Kedem, Z.M., Rabin, M.O.: Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. 15th International Conference on Distributed Computing Systems ICDCS'95, (1995) 467–474

[41] DCGrid. Entropia Inc. http://www.entropia.com

[42] Denes, J., Keedwell, A.D.: Latin Squares and their Applications. Academic Press (1974)

[43] Deng, X., Papadimitriou, C.H.: Competitive Distributed Decision-Making. Algorithmica, Vol. 16(2) (1996) 133–150

[44] De Prisco, R., Mayer, A., Yung, M.: Time-Optimal Message-Efficient Work Performance in the Presence of Faults. 13th ACM Symposium on Principles of Distributed Computing. (1994) 161–172

[45] Dijkstra, E.W.: Solution of a problem in concurrent programming and control. Communications of the ACM, Vol. 8(9) (1965) 569

[46] Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM, Vol. 17(11) (1974) 643–644

[47] Dodis, Y., Halevi, S.: Incremental Codes. 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'01) (2001) 75–89

[48] Dolev, D., Malki, D.: The Transis Approach to High Availability Cluster Communications. Communications of the ACM, Vol. 39(4) (1996) 64–70

[49] Dolev, S., Segala, R., Shvartsman, A.A.: Dynamic Load Balancing with Group Communication. Intl Colloquium on Structural Information and Communication Complexity. (1999) 111–125

[50] Dudley, R.M.: Real analysis and probability. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove (1989)

[51] Dwork, C., Halpern, J., Waarts, O.: Performing Work Efficiently in the Presence of Faults. SIAM J. on Computing, Vol. 27(5) (1998) 1457–1491

[52] Eager, D.L., Lazowska, E.D., Zahorjan, J.: Adaptive load sharing in homogeneous distributed systems. IEEE Transactions on Software Engineering, Vol. 12(5) (1986) 662–675

[53] Ezhilchelvan, P., Macedo, R., Shrivastava, S.: Newtop: A Fault-Tolerant Group Communication Protocol. IEEE International Conference on Distributed Computing Systems, (1995) 296–306

[54] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, Vol. 32(2) (1985) 374–382

[55] Fischer, M.J., Moran, S., Rudich, S., Taubenfeld, G.: The wakeup problem. SIAM Journal on Computing, Vol. 25 (1996) 1332–1357

[56] Fortune, S., Wyllie, J.: Parallelism in Random Access Machines. 10th ACM Symposium on Theory of Computing, (1978) 114–118

[57] Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. International J. Supercomputer Applications, Vol. 15(3) (2001) 200–222

[58] Frankl, P., Rödl, V., Wilson, R.M.: The Number of Submatrices of a Given Type in a Hadamard Matrix and Related Results. Journal of Combinatorial Theory, Series B, Vol. 44 (1988) 317–328

[59] Frontier - The Premier Internet Computing Platform. Parabon Computation, Inc. (2000) (http://www.parabon.com/clients/internetComputingWhitePaper.pdf)

[60] Galil, Z., Mayer, A., Yung, M.: Resolving Message Complexity of Byzantine Agreement and Beyond. 36th IEEE Symposium on Foundations of Computer Science (1995) 724–733

[61] Garofalakis, J.D., Spirakis, P.G., Tampakas, B., Rajsbaum, S.: Tentative and Definite Distributed Computations: An Optimistic Approach to Network Synchronization. Theoretical Computer Science, Vol. 128(1&2) (1994) 63–74 (Preliminary version: WDAG'92)

[62] Gasieniec, L., Pelc, A., Peleg, D.: The Wakeup Problem in Synchronous Broadcast Systems. SIAM Journal on Discrete Mathematics, Vol. 14(2) (2001) 207–222

[63] Genome@home. Pande Group, Chemistry Department, Stanford University. (http://gah.stanford.edu)

[64] Georgiades, S., Mavronicolas, M., Spirakis, P.: Optimal, Distributed Decision-Making: The Case of No Communication. International Symposium on Fundamentals of Computation Theory. (1999) 293–303

[65] Georgiou, Ch., Russell, A., Shvartsman, A.: Work-Competitive Scheduling for Cooperative Computing with Dynamic Groups. 35th

ACM Symposium on Theory of Computing (2003) (Preliminary version: Brief announcement PODC'02.)

[66] Georgiou, Ch., Shvartsman A: Cooperative Computing with Fragmentable and Mergeable Groups. International Colloquium on Structure of Information and Communication Complexity. (2000) 141–156

[67] Gibbons, P.B.: A more practical PRAM model. 2nd ACM Symposium on Parallel Algorithms and Architectures SPAA'89, (1989) 158–168

[68] The Grid: Blueprint for a New Computing Infrastructure. Ian Foster and Carl Kesselman (eds.) Morgan-Kaufmann (1998)

[69] Grid Initiatives and Projects. (http://www.gridforum.org/L_Involved_Mktg/inint.htm)

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

[71] Guy, R.K.: Unsolved Problems in Number Theory, second edition. Springer-Verlag. New York (1994)

[72] Harary, F.: Graph Theory. Reading, MA: Addison-Wesley (1994)

[73] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems, Vol. 13(1) (1991) 124–149 (Preliminary version: PODC'88)

[74] Hayden, M.: Doctoral Thesis, The Ensemble System. TR98-1662, Cornell University (1998)

[75] Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association, Vol. 58 (1963) 13–30

[76] Hopcroft, J.E., Karp., R.M.: A $O(n^{5/2})$ algorithm for maximum matching in bipartite graphs. SIAM Journal on Computing, Vol. 2(4) (1973) 225–231

[77] Hughes, D.R., Piper, F.C.: Design Theory. Cambridge University Press (1985)

[78] Imielinski, T., Korth, H. (eds): Mobile Computing. Kluwer Academic Publishers (1996)

[79] The Intel(r) Philanthropic Peer-to-Peer Program. (http://www.intel.com/cure)

[80] Irani, S., Rabani, Y.: On the Value of Coordination in Distributed Decision Making. SIAM Journal on Computing, Vol. 25(3) (1996) 498-519

[81] Johnson, S.M.: A New Upper Bound for Error-Correcting Codes. IEEE Transactions on Information Theory, Vol. 8 (1962) 203–207

[82] Joung, Y.-J.: Asynchronous Group Mutual Exclusion. 17th Annual ACM Symposium on Principles of Distributed Computing (1998) 51–60

[83] Joseph, A.D., deLespinasse, A.F., Tauber, J.A., Gifford, D.K., Kaashoek, M.F.: Rover: A toolkit for mobile information access. 15th ACM Symposium on Operating Systems Principles (1995) 156–171

[84] Kanellakis, P.C., Shvartsman, A.A.: Efficient Parallel Algorithms Can Be Made Robust. Distributed Computing, Vol. 5(4) 1992 201–217 (Preliminary version: PODC'89)

[85] Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997)

[86] Kaplansky, I.: Linear Algebra and Geometry (a second course). Allyn and Bacon, Inc. Boston (1969)

[87] Karp, R.M., Luby, M., Meyer auf der Heide, F.: Efficient PRAM Simulation on a Distributed Memory Machine. Algorithmica, Vol. 16 (1996) 517–542 (Preliminary version: STOC'92)

[88] Karp, R.M., Wigderson, A.: A fast parallel algorithm for the maximal independent set problem. Journal of the ACM, Vol. 32(4) (1985) 762–773

[89] Kedem, Z.M., Palem, K.V., Rabin, M.O., Raghunathan, A.: Efficient Program Transformations for Resilient Parallel Computation via Randomization. 24th ACM Symposium on Theory of Computing (1992) 306–318

[90] Kedem, Z.M., Palem, K.V., Raghunathan, A., Spirakis, P.: Combining Tentative and Definite Executions for Very Fast Dependable Parallel Computing. 23rd ACM Symposium on Theory of Computing (1991) 381–390

[91] Kedem, Z.M., Palem, K.V., Spirakis, P.: Efficient Robust Parallel Computations. 22nd ACM Symposium on Theory of Computing (1990) 138–148

[92] Kistler, J.J., Satyanarayanan, M.: Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems, Vol. 10(1) (1992) 3–25

[93] Knuth, D.E.: The Art of Computer Programming Vol. 3 (third edition). Addison-Wesley Pub Co. (1998)

[94] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., Lebofsky, M.: Seti@home - massively distributed computing for seti. Computing in Science & Enginering, Vol. 3(1) (2001) 78-83 (http://setiathome.ssl.berkeley.edu)

[95] Kruskal, C.P., Rudolph, L., Snir, M.: A Complexity Theory of Efficient Parallel Algorithms. Theoretical Computer Science, Vol. 71(1) (1990) 95–132 (Preliminary version ICALP'88)

[96] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21(7) (1978) 558–565

[97] Larson, S.M., England, J.L., Desjarlais, J.R., Pande, V.S.: Thoroughly sampling sequence space: large-scale protein design of structural ensembles. Protein Science, Vol. 11(12) (2002) 2804–2813 (http://gah.stanford.edu)

[98] Litzkow, M.J., Livny, M., Mutka, M.W.: Condor - a hunter of idle workstations. 8th International Conference of Distributed Computing Systems (ICDCS'88) (1988) 104–111

[99] Lovász, L.: Combinatorial Problems and exercises. North-Holland Pub. Co, (1979)

[100] Malewicz, G., Russell, A., Shvartsman, A.A.: Distributed Cooperation During the Absence of Communication. 14th International Conference on Distributed Computing, LNCS Vol. 1914 (2000) 119–133 (Preliminary version: Brief announcement PODC'00)

[101] Malewicz, G., Russell, A., Shvartsman, A.A.: Optimal Scheduling for Disconnected Cooperation. 8th International Colloquium on Structural

Information and Communication Complexity (2001) 259–274 (Brief announcement. PODC'01)

[102] Malewicz, G.: A Work-Optimal Deterministic Algorithm for the Asynchronous Certified Write-All Problem. 22nd ACM Symposium on Principles of Distributed Computing (PODC'03) (2003) 255–264

[103] Malewicz, G.: A Method for Creating Near-Optimal Instances of a Certified Write-All Algorithm. 11th Annual European Symposium on Algorithms (ESA'03) (2003) to appear

[104] Malewicz, G., Russell, A., Shvartsman, A.: Local Scheduling for Distributed Cooperation. Invited paper, IEEE International Symposium on Network Computing and Applications (2001) 244–255

[105] Martel, C., Park, A., Subramonian, R.: Work-optimal asynchronous algorithms for shared memory parallel computers. SIAM Journal on Computing, Vol. 21(6) (1992) 1070–1099 (Preliminary version: FOCS'90)

[106] Martel, C., Subramonian, R.: How to emulate synchrony. Technical Report CSE-90-26, UC Davis (1990)

[107] Martel, C., Subramonian, R.: Asynchronous PRAM Algorithms for List Ranking and Transitive Closure. International Conference on Parallel Processing ICPP'90, Vol. 3 (1990) 60–63

[108] Martel, C., Subramonian, R.: On the Complexity of Certified Write-All Algorithms. Journal of Algorithms, Vol. 16(3) (1994) 361–387 (Preliminary version: Technical Report CSE-91-24, UC Davis)

[109] Mills, W.H., Mullin, R.C.: Coverings and packings, in Contemporary Design Theory: A collection of Surveys. (Dinitz, J.H., Stinson, D.R. eds.) John Wiley & Sons (1992) 371–399

[110] Mirchandaney, R., Towsley, D., Stankovic, J.A.: Analysis of the Effects of Delays on Load Sharing. IEEE Transactions on Computers, Vol. 38(11) (1989) 1513–1525

[111] Mitzenmacher, M.: How Useful Is Old Information? IEEE Transactions on Parallel and Distributed Systems, Vol. 11(1) (2000) 6–20

[112] Morgan, J.P., Preece, D.A., Rees, D.H.: Nested Balanced Incomplete Block Designs. Discrete Mathematics, Vol. 231 (2001) 351–389

[113] Moser, L.E., Melliar-Smith, P.M., Agarawal, D.A., Budhia, R.K., Lingley-Papadopolous, C.A.: Totem: A Fault-Tolerant Multicast Group Communication System. Communications of the ACM, Vol. 39(4) (1996) 54–63

[114] Mullender, S.J., Vitányi, P.: Distributed Match-Making. Algorithmica, Vol. 3 (1988) 367–391

[115] Mutka, M.W., Livny, M.: Profiling Workstations' Available Capacity for Remote Execution. 12th IFIP WG7.3 International Symposium on Computer Performance (Performance '87) (1987) 529–544

[116] Naor, J., Roth, R.M.: Constructions of Permutation Arrays for Certain Scheduling Cost Measures. Random Structures and Algorithms, Vol. 6(1) (1995) 39–50

[117] Netzer, R.H.B., Miller, B.P.: On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. International Conference on Parallel Processing ICPP'90, Vol. 2 (1990) 93–97

[118] Nishimura, N.: Asynchronous Shared Memory Parallel Computation. 3rd ACM Symposium on Parallel Algorithms and Architectures SPAA'90, (1990) 76–84

[119] The Olson Laboratory FightAIDS@Home project. (http://www.fightaidsathome.org/discovery.asp)

[120] Papadimitriou, C.H., Yannakakis, M.: On the value of information in distributed decision-making. 10th ACM Symposium on Principles of Distributed Computing, (1991) 61–64

[121] Papadimitriou, C.H., Yannakakis, M.: Linear Programming without the Matrix. 25th ACM Symposium on Theory of Computing (1993) 121–129

[122] Parabon Computation, Inc. (http://www.parabon.com)

[123] Preece, D.A.: Nested Balanced Incomplete Block Designs. Biometrika, Vol. 54(3) (1967) 479–486.

[124] Preece, D. A.: Fifty Years of Youden Squares: A Review. Bulletin of the Institute of Mathematics and its Applications, Vol. 26 (1990) 65–75.

[125] van Renesse, R., Birman, K.P., Maffeis, S.: Horus: A Flexible Group Communication System. Communications of the ACM, Vol. 39(4) (1996) 76–83

[126] Richards, W.G.: Virtual screening using grid computing: the screensaver project. Nature Reviews Drug Discovery, Vol. 1 (2002) 551–555 (http://www.chem.ox.ac.uk/curecancer.html)

[127] Rosenberg, A.L.: Accountable Web-computing. IEEE Transactions on Parallel and Distributed Systems, Vol. 14(3) (2003) 97–106

[128] The RSA Factoring By Web project (http://www.npac.syr.edu/factoring)

[129] Russell, A., Shvartsman, A.: Distributed Computation Meets Design Theory: Local Scheduling for Disconnected Cooperation. Bulletin of the EATCS 77 (2002) 120–131

[130] Satyanarayanan, M.: Fundamental Challenges in Mobile Computing. 15th ACM Symposium on Principles of Distributed Computing (1996) 1–7

[131] Shamir, E., Schuster, A.: Communication Aspects of Networks Based on Geometric Incidence Relations. Theoretical Computer Science, Vol. 64(1) (1989) 83–96

[132] Shvartsman, A.A.: Achieving Optimal CRCW PRAM Fault-Tolerance. Information Processing Letters, Vol. 39(2) (1991) 59–66

[133] Stinson, D.R.: Packings, in CRC Handbook of Combinatorial Designs. (Colbourn, C.J., Dinitz, J.H. eds.) CRC Press (1996) 409–413

[134] Subramonian, R.: Designing synchronous algorithms for asynchronous processors. 4th ACM Symposium on Parallel Algorithms and Architectures SPAA'92, (1992) 189–198

[135] Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice Hall (2001)

[136] Theimer, M.M., Demers, A.J., Petersen, K., Spreitzer, M.J., Terry, D.B., Welch, B.B.: Dealing with Tentative Data Values in Disconnected Work Groups. Workshop on Mobile Computing Systems and Applications, (1994) 192–195

[137] Valiant, L.: A Bridging Model for Parallel Computation. Communications of the ACM, Vol. 33(8) (1990) 103–111

[138] Voogd, J.M., Sloot, P.M.A.,van Dantzig, R.: Crystallization on a sphere. Future Generation Computer Systems, Vol. 10 (1994) 359–361

[139] Weth, C., Kraus, U., Freuer, J., Ruder, M., Dannecker, R., Schneider, R., Konold, M. Ruder, H.: XPulsar@home - Schools help Scientists. Proceedings der CCGrid, Brisbane (2001) 588–594