© Copyright 1995
Rakesh Kumar Sinha

Some Topics in Parallel Computation and Branching Programs

by

Rakesh Kumar Sinha

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1995

Approved by_____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Some Topics in Parallel Computation and Branching Programs

by Rakesh Kumar Sinha

Chairperson of the Supervisory Committee: Professor Paul Beame
Department of Computer Science
and Engineering

There are two parts of this thesis: the first part gives two constructions of branching programs; the second part contains three results on models of parallel machines.

The branching program model has turned out to be very useful for understanding the computational behavior of problems. In addition, several restrictions of branching programs, for example ordered binary decision diagrams, have proven to be successful data structures in several VLSI design and verification applications. We construct a branching program of $o(n \log^3 n)$ nodes for computing any threshold function on n variables and a branching program of $o(n \log^4 n)$ nodes for determining the sum of n variables modulo a fixed divisor. These are improvements over constructions of size $\Theta(n^{3/2})$ due to Lupanov [Lup65].

The second part of this thesis deals with parallel computation. A wide variety of parallel machines with fundamentally different architectures has resulted in a variety of theoretical models, each trying to capture the behavior of parallel machines belonging to a particular class.

We first prove a separation result between bounded *communication width* CREW (concurrent read, exclusive write), and EREW (exclusive read, exclusive write) PRAMs, where the communication width of a PRAM is defined to be the size of the global memory available for writing. We prove that a Boolean decision tree of height h can be easily evaluated in time $O(\sqrt{h})$ on a CREW PRAM with communication width 1 using $2^{O(h)}$ processors but requires $\Omega\left(\frac{h}{m+\log^* h}\right)$ time on any EREW PRAM with communication width m, even without any restriction on

the number of processors.

We then consider augmented PRAMs that have multiprefix operation for certain operators available as primitives [Ble90, RBJ88]. We prove that such PRAMs can be simulated by unbounded fan-in circuits with gates for AND, OR, NOT, and operations corresponding to the multiprefix primitives of the PRAMs. This gives a way of translating lower bound results proven for the case of circuits to the case of these augmented PRAMs.

Our last result is an optimal $\Theta\left(\frac{\log p}{\log\log p}\right)$ running time algorithm for computing the sum of integers on a $\sqrt{p} \times \sqrt{p}$ sub-bus mesh when each processor starts with one $O(\log p)$ bit integer.

Table of Contents

List of Figures **iv**

Chapter 1: Introduction **1**

1.1 Branching programs 3

1.2 Parallel Computation 5

Chapter 2: Introduction to Branching Programs **11**

2.1 Motivation and Definitions 11

2.1.1 Relationship to Turing Machines, Circuits, and Formulas . . 13

2.1.2 Ordered Binary Decision Diagrams 16

2.1.3 Restrictions and Extensions of the Branching Program Model 16

2.2 Some Basic Results 18

2.2.1 Time-Space Trade-off Results 18

2.2.2 Size Complexity Results 18

Chapter 3: Symmetric Functions on Branching Programs **21**

3.1 Lower Bounds 22

3.1.1 Constant Width Branching Programs 22

3.1.2 Communication Complexity Technique 23

3.2 Upper Bounds 26

3.2.1 Constant Width Branching Programs 26

3.2.2 Computing Sum of the Bits 27

3.2.3 Modular Arithmetic Branching Programs (MA-programs) . . 31

3.2.4 MA-programs for arbitrary symmetric function 35

3.3 Branching Programs for Threshold and Mod functions 36

3.3.1 Main Technical Lemma 38

3.3.2 MA-programs for Threshold Functions 39

3.3.3 MA-programs for Approximate Division 42

3.3.4 MA-programs for Mod Functions 45
3.3.5 MA-programs for Strong Threshold Problems 49
3.4 Conclusion 52

Chapter 4: CREW PRAMs versus EREW PRAMs 57
4.1 Introduction 57
4.2 Communication Width 59
4.3 Separating Different Variants of PRAMs 60
4.4 Previous Attempts at Separating CREW and EREW PRAMs . . . 61
4.5 A separation result between CREW(m) and EREW(m) PRAMs . . 63
4.5.1 A Bound on the Number of Processors Doing Useful Work . 64
4.5.2 A CREW(1) Upper Bound for Evaluating Decision Trees . . 67
4.5.3 Lower Bounds for Processor-limited PRAMs 68
4.5.4 A Near Optimal EREW(m) Lower Bound for Small m . . . 72
4.6 Conclusion 77

Chapter 5: Multiprefix PRAMs 79
5.1 Introduction 79
5.2 Definitions 83
5.3 Simulation 84
5.3.1 Simulation of Memoryless PRAMs by Circuits 86
5.3.2 Simulation of Multiprefix PRAMs by Memoryless Multiprefix PRAMs 89
5.4 Conclusion 92

Chapter 6: Computing SUM on the Sub-Bus Mesh 93
6.1 Introduction 93
6.1.1 The Sub-Bus Mesh Model 93
6.1.2 Related Results 96
6.2 Algorithms 99
6.2.1 PARITY Algorithm 99
6.2.2 SUM Algorithm 102
6.3 Conclusion 108

Chapter 7: Final Thoughts **111**

Bibliography **113**

List of Figures

3.1 Branching program for computing $(x_1 + x_2 + x_3 + x_4) \bmod 3$ 29
3.2 CHAIN$< p_1, p_2, \ldots, p_k >$ 33
3.3 Intervals S, R, and R' 49

6.1 Subdivision in Lemma 6.6 101
6.2 Subdivision in Lemma 6.8 102
6.3 Subdivision in Lemma 6.10 103
6.4 Subdivision in Lemma 6.11 104
6.5 Subdivision in Lemma 6.13 106
6.6 the SUM algorithm 110

ACKNOWLEDGMENTS

I would like to express my deep gratitude towards my professors and fellow graduate students for their help and encouragement during my stay at University of Washington.

My advisor, Paul Beame, gave me complete freedom regarding choice of research problems, provided expert guidance, and helped me in every possible way. All the results in Chapters 4 and 5 have been obtained with his direct collaboration, and the results in other chapters have benefitted from his suggestions. Paul also supported me with an assistantship throughout my stay here.

Most of my knowledge of probabilistic techniques and algebraic methods is from working with Martin Tompa. Martin contributed generously to many of my research results while refusing to take any credit for them. I am also grateful to him for his encouragement and enthusiasm which were vital for my survival in the graduate school. Both Paul and Martin patiently read through multiple drafts of my research papers. Their suggestions greatly improved the presentation and, in several instances, also simplified the results.

I have received far more support and guidance from the other theory faculty, Richard Anderson, Richard Ladner, and Larry Ruzzo, than what is due to a non-advisee. Most of the results in my thesis have been obtained in collaboration with various colleagues. I would like to thank my co-authors, Anne Condon, Faith Fich, Richard Ladner, Jordan Lampe, and Jayram Thathachar, for allowing me to include these results in my thesis. Graduate school would not have been so much fun without my friends, Ka Chai, Donald Chinn, Melanie Fulgham, Simon Kahan, Tracy Kimbrel, Joan Lawry, Joao Setubal, and many others, who all went through the trauma of listening to my half-baked research ideas.

Finally I would like to thank my family; I owe almost everything in my life to their vision, encouragement, and support.

To live is to battle with trolls
in the vaults of heart and brain.

To write: that is to sit
in judgement over one's self. ——- Ibsen

To Mama, Ma, Babuji, Ritu, and the rest of my family.

Chapter 1

INTRODUCTION

The goal of theoretical computer science is to make computers more efficient by gaining a better understanding of the computational difficulties of various problems. For any given problem, ideally we would like to learn the most efficient way to solve it. There are two complementary activities that go towards achieving this goal. We can show an *upper bound* on the computational resources needed by designing an algorithm. Alternatively, we can prove a *lower bound* on the computational resource requirements by proving that there does not exist an efficient algorithm for solving that particular problem.

The study of upper bounds is self-justified. Because computers have fixed resources, there is a clear motivation to develop solutions making efficient utilization of those resources.

Study of lower bounds is equally important. Besides the purely intellectual issue of trying to get a better understanding of the inherent difficulty of problems, there are several practical advantages. The most important of these is the concentration of efforts on the right problems. In other words, we would like to direct our efforts so as to maximize the possible gains. Algorithm designers are always in search of more efficient solutions to problems. The search stops only when we have a lower bound that matches the performance of the currently best known algorithm, because we know that all attempts at further optimization of the algorithm are going to be wasted. For many problems, even though we have not been able to prove actual lower bounds, we have been able to provide strong empirical evidence of their inherent difficulty. For example, while no strong lower bounds are known for any NP complete problem, we know that researchers all over the world have not been able to find an efficient solution for them even after decades of effort. So

the NP completeness proof of a particular problem is often taken as an evidence that we have hit a wall. And even though it is certainly possible to find an efficient solution for it, common sense dictates that we should diversify our efforts in trying to get around NP completeness. Very often such pessimistic results (either actual lower bounds or evidence of difficulty) force us to reformulate the problems and in many instances we have been able to develop solutions that are quite acceptable in practice. For instance, a closer examination of the problem at hand may reveal that we may be trying to solve a problem more general than what is needed and the restricted practical problem does admit an efficient solution. In other cases, it may be acceptable to settle for algorithms that either give only approximate solutions or are efficient most but not all of the time (two examples are probabilistic algorithms and deterministic algorithms with good average case behavior). In all such cases, a pessimistic outlook on the original problem channels our efforts in a direction where we have a better chance of making progress. The theory of lower bounds has been indirectly responsible for generating a rich theory of approximate, probabilistic, and good average case algorithms. Even the purely intellectual issue of gaining a better understanding of the computational difficulty of problems has often paid rich dividends in the long term. For example, the concept of nondeterminism has no basis in real machines but as an intellectual tool its study has greatly enriched our understanding of the difficulty of many real life problems on real machines (for example, identifying problems as NP complete and therefore realizing that most likely they will not admit efficient solutions).

Out of the two, the study of upper bounds has enjoyed more success. We have been able to develop a rich body of ideas and efficient algorithms for a variety of problems occurring in real life. For many problems, improved algorithms have enabled us to solve instances several orders of magnitude larger than what was achievable even ten or twenty years ago.

Unfortunately, the field of lower bounds has not been very successful. While we have been quite successful in identifying candidate problems that appear to be inherently difficult to solve (for example, the complete problems for various complexity classes), we have had very little success in proving actual lower bounds.

The choice of problems for proving lower or upper bounds is critical. Clearly, one would like to consider problems that practitioners encounter so that the efforts

invested in understanding the difficulty of problems has immediate practical bearing. However, most problems arising in practice have too many details. It is more feasible to study a simpler problem stripped of some of the details if there is an easy translation of results from the simpler problem to the original problem. There are at least two advantages of studying the simpler problem: first, stripping the unnecessary details helps us focus on the truly important aspects of the problem; second, considering a more general problem may in fact speak to a whole class of related problems arising in practice rather than just one.

It is particularly important to study the behavior of elementary problems. Since their solutions are used as subroutines in many algorithms, their performance have a significant impact on the performance of a variety of algorithms for solving other problems. Another reason to study elementary problems is that their understanding can be seen as a first step towards a better understanding of more complicated problems.

There are two parts of this thesis: The first part of this thesis deals with *branching programs* and the second part deals with three different models of parallel machines.

1.1 Branching programs

Turing machines and random access machines (RAMs) have proven to be very useful models for the purpose of designing algorithms. However, they do not seem to be quite adequate for studying lower bounds. A good model should be able to correctly predict the behavior on real world machines. In the case of Turing machines, because of their sequential access feature, we have several examples of problems (details are given in Chapter 2) which have very efficient solutions on real machines (with random access) but are provably difficult to solve on Turing machines. In other words, there is a danger that any lower bound proved on the model may not be saying anything meaningful about the behavior of the given problem on real machines.

RAMs are a more realistic model of practical machines. They would have been good models to prove lower bounds on, except that there is a growing consensus that the model is not simple enough to facilitate lower bound arguments. We will

elaborate more on this in Chapter 2. *Branching programs* redress many of these problems.

Informally, a branching program is a directed acyclic graph where each node reads an input variable and control flows to other nodes depending on the value of that input variable. It is an abstraction of many other computing models: The nodes of the branching program represent configurations of a machine, and edges define transitions on inputs. It has been a very attractive model to study because of its simplicity. At the same time the model is powerful enough to model the performance of real machines.

Recently, variants of branching programs (for example, ordered binary decision diagrams) have turned out to be very useful data structures for design, verification, and testing of digital systems (see e.g. Bryant [Bry92], Burch et al. [BCL$^+$94]). The complexity of these design, verification, and testing algorithms is often directly related to the size of the underlying data structures. This gives added motivation for efficient computation of functions by branching programs.

Chapter 2 is an introduction to the branching program model. We outline some motivation to study the branching program model, state the definitions and summarize some previously known results.

In Chapter 3, we give two constructions of branching programs, each one computing a very natural class of elementary functions. Given n input bits, the k-th *threshold* function (for $0 \leq k \leq n$) is defined to have value 1 if and only if at least k of them are one. For fixed a and d, the *mod* function is defined to have value 1 if and only if the number of 1's in the input is congruent to $a \bmod d$. We construct simple branching programs of $o(n \log^3 n)$ nodes for computing any threshold function and $o(n \log^4 n)$ nodes for computing any mod function. These are the first improvements in nearly thirty years of a construction of size $O(n^{3/2})$ due to Lupanov [Lup65]. Because the behavior of these two classes of functions depends only on the number of 1's in the input, we introduce a new model *Modular Arithmetic Branching Programs* (MA-programs) that operates on integers. There is a direct mapping from MA-programs to branching programs, and presenting our construction in terms of MA-programs allows us to highlight the important details without getting unduly distracted. Our main technical contribution is an MA-program for

computing "approximate division."

1.2 Parallel Computation

The second part of this thesis deals with parallel computation, whose study has become a very important part of Computer Science.

Because of the decreased cost of hardware, it has become feasible to have several processors in a single computer, which suggests a very simple scheme for reducing the time needed to solve any problem. If we are really lucky, we may be able to break a given problem into some large number p of independent subproblems such that solving any of the subproblems takes about $\frac{1}{p}$ of the time to solve the original problem. In this case, having a parallel computer with p processors will enable us to solve the problem in $\frac{1}{p}$ of the time needed on a sequential computer.

Of course, very few problems in practice can be subdivided so easily into independent parts. Typically, the best we can do is to break the problem into subproblems such that each subproblem can be solved much more efficiently than the original problem and very little interaction is needed among processors solving the subproblems. Achieving such a speed-up requires contributions from both algorithm designers and computer architects: The architects build parallel computers where the interaction between processors can be carried out efficiently, whereas the goal of algorithm designers is to minimize the need for interaction between processors.

Algorithm designers need an abstract model of the parallel machine. The choice of a good model is critical for the success of the parallel computation field. Algorithm designers and computer architects have a symbiotic relationship: The former have to choose a model that reflects the realities of real machines. On the other hand, if many good algorithms are designed on an abstract model then there is incentive for architects to provide an efficient implementation of that model in hardware. However, the two parties put conflicting requirements on the model: algorithms designers want the model to be abstract enough to hide implementation details so that it is simple and easy to analyze, whereas the concern of computer architects is to make sure that the model admits an efficient implementation.

A model that is very close to the real machines is not necessarily a good model.

To give an example, every computer (or in fact any finite system) can be precisely modeled by a finite state automaton, but FSAs have not turned out to be a popular model for designing algorithms. On the other hand, Turing machines or RAMs, despite their physically unrealizable features of infinite memory and unlimited word-size, have proved to be very useful platforms for designing algorithms which run efficiently on real machines. The goal of a good model should be to provide a programmer's view of the machine.

In the case of sequential computers, even with the seemingly endless variations in the architecture of real machines, RAMs or Turing machines seem to provide a general model, which up to a first approximation, provides a good indication of performance on any real computer (at least for problems that are not I/O-bound).

Unfortunately, we don't yet have such a general model of parallel computation. There is a plethora of parallel machines with completely different architectures. Just as in the case of sequential computers, we would like to have a model such that algorithms designed on it run efficiently on machines with seemingly different architectures. At least for now, there seems to be a consensus among computer scientists that the differences in architectures here are so fundamental that we need separate design techniques for different architectures.

Because we do not yet have a consensus on appropriate logical realization of parallel computers, a variety of theoretical models of parallel computation have been studied.

In the second half of the thesis, we will consider three of these models. We briefly outline the results in each chapter. The detailed definitions and motivation for each problem are included at the beginning of each chapter.

In Chapter 4, we will consider *parallel random access machines* (PRAMs) which have been the most popular model of shared memory machines for describing parallel algorithms and analyzing parallel complexity of problems.

The PRAM model consists of a global shared memory and a set of processors. Processors operate in lock step, and any processor is allowed to access any location in the memory. The model has several different variants, which all differ in whether or not they allow more than one processors to *concurrently* access any given memory location. The three most popular models are CRCW (concurrent read, concurrent

write), CREW (concurrent read, exclusive write), and EREW (exclusive read, exclusive write) PRAMs. In the case of CRCW PRAMs, we have additional variations depending on how write conflicts are arbitrated. It is clear from the definition that the CRCW variant is at least as powerful as the CREW variant, which, in turn, is at least as powerful as the EREW variant.

It is not immediately clear which of these three models is the most appropriate platform for designing algorithms; there is a trade-off between ease of design and efficiency of implementation: the powerful variants are the most attractive for describing algorithms, but they are also the most expensive to implement.

This makes it important to understand the relative power of these variants. If there is an efficient simulation of the CRCW variant by the EREW variant then algorithms can be designed on CRCW PRAMs and the simulation algorithm can then act as a compiler to translate them to weaker EREW PRAMs. On the other hand, if there is a large gap in the powers of CRCW and EREW PRAMs then extra effort should be invested in designing algorithms on the EREW model or finding an efficient way to build CRCW PRAMs.

The two big questions are determining the relative power of (1) CRCW and CREW PRAMs and (2) CREW and EREW PRAMs.

The results in Cook et al. [CDR86], Kutyłowski [Kut91] and Dietzfelbinger et al. [DKR94] prove that for computing several functions (including the OR of n bits), the CRCW model is at least $\Omega(\log n)$ times faster than the CREW PRAM. The problem of determining the relative power of CREW and EREW PRAMs remains open.

We make partial progress in Chapter 4 by showing that if the size of the memory through which processors communicate is severely restricted, then a decision tree can be evaluated more efficiently on a CREW PRAM than on any EREW PRAM.

The PRAM model provides a very high level programmer's view of parallel machines. This is a very appealing feature for algorithm designers. Of course, we pay a price for this convenience: the read and write primitives, assumed to take unit cost on the model, are quite expensive to implement in hardware even for the case of the weakest EREW variant.

We argue that this fact in itself is not necessarily a big flaw of the model. One can raise a similar objection to the unit cost RAM model, where memory references and all other primitive instructions are assumed to take unit cost on the model but may take non-constant time on a real computer. Different machines have different implementations of primitives of the theoretical model, so that any general model gives a running time which is not the same as that on real machines. The usefulness of any model derives mainly from its ability to predict relative performance of algorithms and problems: so if a model predicts that algorithm A is more efficient than algorithm B, it should be the case that an implementation of A on any computer is indeed more efficient than an implementation of B. Thus we can pick an algorithm based on its efficiency (relatively speaking) on the model and be sure that it is the best available choice to run on any real machine.

The real danger of having primitives that are expensive to implement on real machines is in wrongly predicting the relative performance of problems. As an extreme example, consider functions which have the same complexity as read and write primitives on real machines. We would like the model to predict that all these functions have very efficient solutions. Unfortunately, many of these functions turn out to have no constant time implementation in terms of read and write primitives of the PRAM model. (The PARITY function is an example of such a function [BH89].)

There are two possible approaches one can take at this point: One choice is to strip down the model, leaving only primitives that are easy to implement. This is a reasonable choice but it goes against the basic philosophy that the most important function of a model is to provide a high level view of the machine.

The other approach is to augment the PRAM model with a set of primitives that have the same hardware complexity as reads and writes.

A variety of theoretical and empirical work [Ble89, Ble90, CBZ90, KRS86, RBJ88, PS88, KRS88] has suggested that parallel prefix computations for certain associative operations can be done in time comparable to implementing reads or writes. Providing these extra primitives makes many algorithms simpler and/or efficient. We call these models *multi-prefix* PRAMs and parameterize them by the set of associative operations for which parallel prefix computations are allowed at unit cost. The model becomes unreasonable if we allow arbitrary associative

operations. A natural restriction is to limit the *bandwidth* of the operations which, loosely speaking, is a measure of the size of packets that have to be propagated in the underlying network implementing the multiprefix PRAM. In Chapter 5, we show that under a natural bandwidth restriction on any such multi-prefix PRAM, it can be efficiently simulated by an unbounded fan-in circuit with special gates for the associative operations allowed in the multi-prefix PRAM. This gives one way of translating the known lower bound results proved on circuits to the case of multiprefix PRAMs.

Finally in Chapter 6, we consider a sub-bus mesh model. In a mesh connected computer, there is a processor placed on every grid point of a 2-dimensional mesh. The processors have the capability to broadcast data to their left, right, up, or down. In every cycle of computation, processors can dynamically partition themselves into groups such that the data broadcast by any processor reaches all other processors in its partition. There are several variants of the model which all differ in the kinds of partitioning they allow. The architecture is very attractive because of its regular design and low cost of processor interconnection. It has been popular both among practitioners building parallel machines and among theoreticians needing a model to design parallel algorithms on [Bat80, HS86, Lei92, LS91, MS89, MPKRS93, RPK88, Sto86]. We will focus on the sub-bus mesh computer, which has been implemented on the commercially available MasPar MP-1 [Bla90].

The sub-bus mesh computer is a single-instruction multiple-data (SIMD) machine. All broadcasts in any particular step are in the same direction. In the case of left or right broadcasts, every row gets partitioned into sets of consecutive columns so that within any partition, at most one processor broadcasts; similarly, in the case of up or down broadcasts, every column gets partitioned into sets of consecutive rows so that within any partition, at most one processor broadcasts. We assume that broadcasts take unit time.

We consider the problem of computing the sum of bits on a $\sqrt{p} \times \sqrt{p}$ sub-bus mesh when each processor starts with one input bit, and give an asymptotically optimal algorithm, running in time $O\left(\frac{\log n}{\log \log n}\right)$.

Most results in this thesis have already been published as journal or conference papers. The branching program construction for threshold functions in Chapter 3

first appeared in [ST94]; the separation result of Chapter 4 appeared in [BFS]; and the sub-bus algorithm of Chapter 6 appeared in [CLLS].

Chapter 2

INTRODUCTION TO BRANCHING PROGRAMS

2.1 Motivation and Definitions

Theoretical computer scientists need to model real machines in order to analyze the computational behavior of problems.

By far, Turing machines and RAMs have been the two most successful models for theoretical study of computational problems. They have helped us generate a large body of algorithms. We have also been able to identify candidate problems which have strong theoretical and empirical evidence of being inherently difficult (for example, complete problems for various complexity classes). Unfortunately, though, the goal of proving actual lower bounds has been largely elusive. There has been a growing realization of inadequacies of the Turing machine and RAM models for proving lower bounds. We first outline the difficulties with the Turing machine model.

A series of relativization results have shown that most of the known lower bound techniques on Turing machines are not powerful enough to prove strong lower bounds (see e.g. Baker, Gill, and Solovay [BGS75]). An even more serious concern is that Turing machines access their memory in sequential order. Because all real machines have random access, a lower bound proved on Turing machines that relies crucially on the sequential access feature may not say anything meaningful about resource requirements on real machines. We will give two examples. As our first example, consider the problem of checking whether two halves of a given input string of length $2n$ are equal. Cobham's classical result [Cob66] states that for any Turing machine solving this problem, the product of its time and space requirement is at least $\Omega(n^2)$. However, the proof crucially relies on the fact that in order to read input symbols located far apart on the tape, the Turing machine has to spend a lot of time. In current digital technology, where random access is fairly

efficient, this problem has an easy solution using space $O(\log n)$ and time $O(n)$. In other words, the model is making false predictions about the computational difficulties of certain problems. Even on the Turing machine, if we allow two heads, the problem becomes simpler (solvable in space $O(1)$ and time $O(n)$). This again is not very desirable because for small polynomial lower bounds (the only ones that we have been able to prove so far) the model is not very robust. As another example, the Turing machine model can not differentiate between the complexity of sorting n integers and merging two sorted lists of n integers each [Tom78]; even though merging seems an easier problem to compute on real machines.

RAMs are an attractive model because of their closeness to real machines. The main objection against the RAM model (which also applies to Turing machines) is that the model has too many features: A lower bound proof consists of showing that no matter what an algorithm does, it can not solve the given problem efficiently. The complexity of such a proof is often proportional to the number of different types of activities allowed on the model.

This suggests that while Turing machines and RAMs have been very successful for many tasks, in order to make progress we need to analyze models that are easier and at the same time at least as powerful as real machines.

The branching program model seems to fulfill both these requirements. Its many variants have long been popular for studying complexity of functions (see, for example, the survey paper of Razborov [Raz91]). It is a simple model which is amenable to combinatorial analysis and at the same time is powerful enough to simulate many other computational models. In particular, it redresses the input addressing mechanism of the Turing machine model by providing random access to inputs. We will show its relation to other models, for example, RAMs, circuits, formulas, and Turing machines. Branching programs have been particularly useful for proving that, for certain problems, both time and space can not be limited concurrently. These lower bounds on space and time also apply to the logarithmic cost RAM model (Proposition 2.3). Recent use of some restrictions on the branching programs model (for example, *ordered binary decision diagrams*) as a data structure in circuit design and verification has given added motivation to construct efficient representation of functions in terms of branching programs.

We define branching programs for computing integer functions of n input bits $x_1, x_2, \ldots, x_n$.

Definition 2.1 *A* branching program *is a directed acyclic graph with a designated* source *node and some number of* sink *nodes. Each sink node is labeled with an integer. Each non-sink node has out-degree two and the two outgoing edges are labeled $x_i = 0$ and $x_i = 1$ for some input variable x_i. In this case we say that the node* reads *the variable x_i. The branching program computes a function in the following way: it is easy to see that any setting of input bits $x_1, x_2, \ldots, x_n$ defines a unique path from the source node to one of the sink nodes in which all edge labelings are consistent with the assignment of $x_1, x_2, \ldots, x_n$; the label of this sink node is the value of the function.*

There are two interesting measures associated with branching programs: length and size. If each node takes one clock cycle to decide which of its two outgoing edges is consistent with the given input then the time needed to compute the value of the function is exactly the maximum distance from the source node to any of the sink nodes. This maximum distance is called the *length* of the branching program. The hardware requirement is measured by *size*, which is the number of nodes in the branching program.

The branching program model is an abstraction of many other computing models. The nodes of the branching program represent configurations of a machine, and edges define transitions on input bits. We have already defined *time* on branching program as its length; we define *space* as the logarithm of its size. The justification for defining space in this way comes from the following two theorems relating branching programs to Turing machines and RAMs.

2.1.1 Relationship to Turing Machines, Circuits, and Formulas

Since we allow different branching programs for each value of input length, without requiring any connection between these branching programs, they can even compute non-recursive functions. To relate their power to those of Turing machines, we also need to define nonuniform versions of Turing machines. An $s(n)$ space bounded *nonuniform* Turing machine is allowed to hardwire $2^{O(s(n))}$ bits of advice. (See page 279 of Wegner [Weg87] for a definition of nonuniform Turing machines.)

Proposition 2.2 *(Cobham [Cob66], Pudlák and Zák [PZ83]) For any $s = \Omega(\log n)$, branching programs using space $O(\log s)$ compute exactly the same class of functions as nonuniform Turing machines using space $O(\log s)$.*

Proof sketch: We will state the basic intuition of the proof; a complete proof appears on page 415 of Wegner [Weg87]. Given a branching program, a Turing machine can compute the same function by simulating the path from source to the sink node that is consistent with the given input. In order to simulate the branching program, we need space equal to the logarithm of the size of the branching program to be able to index its nodes. Conversely, given a nonuniform Turing machine, we construct an equivalent branching program whose nodes are the configurations of the Turing machine and edge transitions correspond to changes of configuration on input bits. □

In particular this theorem implies that polynomial size branching programs compute exactly the same class of functions as log space bounded nonuniform Turing machines.

The time and space requirements of branching programs give a lower bound on the (respective) time and space requirements of logarithmic cost RAMs. (See page 23 of van Emde Boas [vEB90] for a definition of the logarithmic cost RAM model.) There are several possible ways of defining space on logarithmic cost RAMs (see the discussion on page 27 of [vEB90]). The following proposition holds for any definition of space which ensures that the space requirement of any particular computation is greater than or equal to the number of bits needed to describe configurations of the machine during this computation. All reasonable definitions of space on the logarithmic cost RAM model satisfy this property.

Proposition 2.3 *(Borodin and Cook [BC82]) The time and space complexity of any Boolean function on the branching program model is at most (up to a constant multiplicative factor) its time and space complexity on the logarithmic cost RAM model.*

Proof sketch: Given any RAM M, we can construct an equivalent branching program P on the input variables. P contains a node corresponding to each possible

configuration of M. The node corresponding to the initial configuration of M becomes the start node of P; and nodes corresponding to final configurations of M become sink nodes of P. We simulate every transition of M as follows. If M reads the input variable x_i in configuration c and achieves configuration c_0 or c_1 depending on the value of x_i then we connect the node corresponding to c to nodes corresponding to c_0 and c_1 by directed edges labeled $x_i = 0$ and $x_i = 1$. It is straightforward to verify that P computes the same Boolean function as M . Also, if M has space complexity s, its configurations can be encoded with at most s bits so that P has at most 2^s nodes and its space complexity is at most s. It is also easy to see that the depth of P is equal to the maximum number of input reads on any computation path of M , so that time on P is bounded above by the cumulative time spent in reading input variables by M. Notice that this proposition holds even if we make all computations except reading the inputs free for RAMs. □

The size complexity of branching programs is known to be closely related to other well studied models of computation. The following theorem relates the size complexity of branching programs to size complexities of circuits and formulas (see Wegner [Weg87] for definitions of circuits and formulas).

Proposition 2.4 *The size complexity of any Boolean function in the branching program model is at least one third of its circuit size and at most one more than its formula size.*

Proof sketch: The proof is not very hard (and can be found, for example, on page 416 of [Weg87]). Given a branching program, we will define a transformation to obtain an equivalent circuit. First, we reverse all the edges and make the source node the output node of the circuit. Then we replace each node in the branching program reading x_i by a multiplexor which, depending on the value of x_i, outputs one of the two values feeding this gate. Because each multiplexor gate can be built with 3 gates from the DeMorgan basis, we get a circuit of size three times the size of the original branching program. Showing the relationship between branching program and formula size is also easy. Given two branching programs computing Boolean functions f_1 and f_2, it is straightforward to construct branching programs for computing $f_1 \vee f_2$ or $f_1 \wedge f_2$ in size at most the sum of the sizes of the original branching programs. On the other hand, since formula size is measured as the

number of gates, the sum of the sizes of the optimal formulas for f_1 and f_2 is one less than the size of the optimal formula for computing $f_1 \vee f_2$ or $f_1 \wedge f_2$. An easy induction on the structure of the formula proves that the branching program size is at most one more than the formula size. □

2.1.2 Ordered Binary Decision Diagrams

The recent attention on *ordered binary decision diagrams* (OBDDs) and their variants by VLSI chip designers has given fresh motivation for efficient computation of functions by branching programs. OBDDs are branching programs with the extra restriction that there is a fixed ordering of 1 to n such that if a node reading x_i is an ancestor of another node reading x_j then i precedes j in the predetermined order. In particular, it implies that any input variable appears at most once on any source to sink path. VLSI researchers need some form of representation for functions. Many conventional representations, like Karnaugh map or truth table, have the undesirable property that their size is exponential in the number of variables. On the other hand, many concise representations preclude any easy manipulation or formal analysis. OBDDs seem to have a good balance: they provide compact representation for many functions arising in practice and at the same time, they can be manipulated very efficiently. They have been extensively used in the design, verification, and testing of digital systems (see e.g. Bryant [Bry92], Burch et al. [BCL$^+$94]). In order to make these systems efficient, the size of the representation has to be minimized. For many functions, no efficient representation with OBDDs is possible. Building on the success of OBDDs, researchers have tried either to define other models which are similar to OBDDs (see e.g. Bryant and Chen [BC94]) or to relax some of the restrictions of the OBDDs to allow more efficient representations while still retaining the ease of manipulation (see the discussion at the end of [Bry92]).

2.1.3 Restrictions and Extensions of the Branching Program Model

We now discuss some variants of the branching program model. In a *leveled* branching program, the nodes are partitioned into levels $L_0, \ldots, L_l$ such that every edge from a node in L_i goes to a node in L_{i+1}. We will say that a node is in level i when it belongs to L_i. The *width* of a leveled branching program is defined to be the maximum number of nodes in any level of the branching program. An *oblivious*

branching program is a leveled branching program in which all out-edges from any particular level access the same input variable.

Given a leveled branching program of width w and length ℓ, it can be easily converted into an equivalent oblivious branching program of width $w+1$ and length at most ℓw by creating w levels for each level in the original branching program.

It is also possible to define branching programs in a more general manner. We say that a path is a *semantic* path if it starts from the source and does not contain two edges labeled $x_i = 0$ and $x_i = 1$ for any variable x_i. Intuitively, on any given input, control follows one of the semantic paths in the branching program. In a more general definition of branching programs, we allow cycles in the underlying graph as long as no semantic path contains a cycle. For such a general branching program, we define its length to be the length of the longest semantic path. Pippenger has shown an easy way of converting any such general branching program of size s and length ℓ into a leveled branching program of size $(\ell + 1)s$ and the same length ℓ (details are given in Borodin et al. [BFK$^+$81]). Pippenger's transformation makes $\ell + 1$ copies of each node in the original branching program and connects them in a way such that the ith node on any semantic path is the ith copy of some node in the original branching program.

There have been two important generalizations of the branching program model. (1) In the model we have defined, any setting of input variables corresponds to a unique source to sink path. There are several nondeterministic variants of the branching program model where any setting of input corresponds to zero or more paths in the underlying graph. The nondeterministic variants differ in their acceptance criteria as well as on the types of underlying graphs they allow (see Razborov's survey paper [Raz91]). (2) It is also possible to define branching programs to compute function over non-Boolean domains. If each input variable comes from a domain of size R, then in the ensuing model each non-sink node has R outgoing edges. The model was introduced by Borodin and Cook [BC82] and has been subject to extensive studies (for example, [Abr91, Abr90, Bea91]).

2.2 Some Basic Results

2.2.1 Time-Space Trade-off Results

Because of their ability to model both space and time, branching programs and their many variants have been particularly useful in proving many time-space trade-off results. That is, for solving certain problems, it has been shown that both time and space can not be limited at the same time [Abr91, Abr90, Bea91, BC82, BFK$^+$81, Yao88, BFMadH$^+$87, MNT90, PSP93] (also see Borodin's excellent survey article [Bor93]). Unlike Turing machines, branching programs allow random access of inputs, so that lower bounds proved on the branching program model provide a strong guarantee of the difficulty of problems on real machines.

The general proof technique is to define some notion of "progress" towards computing a given function. Then break the computation into many sub-stages and argue that during any sub-stage most inputs make very little "progress". For many multi-output functions, the number of outputs has turned out to be a useful notion of progress. For single output functions, finding the "right" measures of progress has been much more difficult and for that reason it has been more challenging to prove bounds on single output functions.

2.2.2 Size Complexity Results

As is traditional in complexity theory, we will concentrate on decision functions, that is, functions with the range {accept, reject}. Despite the apparent simplicity of the branching program model, researchers have had very little success in proving interesting bounds. By a counting argument we know that almost all functions have size complexity $\Theta(\frac{2^n}{n})$ on branching programs, but the best lower bound for a function in NP is only $\Omega\left(\frac{n^2}{\log^2 n}\right)$. This bound was first proved by Nečiporuk [Neč66] for a somewhat contrived function. Beame and Cook (unpublished) noticed that Nečiporuk's technique can be applied to prove the same lower bound for the element distinctness problem. Nečiporuk's technique is essentially a counting argument and applies to the stronger switching network model (see [Raz91] for definition).

Theorem 2.5 *[Neč66] Let $b_1, b_2, \ldots, b_m$ be any partition of input variables into m groups. Then setting all variables outside b_i gives a sub-function of f on the*

variables in b_i. For any function f that depends on all its input variables, let $s_i(f)$ be the total number of distinct such sub-functions on variables in b_i, obtained by fixing all variables outside b_i in all possible ways. Then the size of any branching program computing f is

$$\Omega\left(\sum_{i=1}^{m}\frac{\log s_i(f)}{\log\log s_i(f)}\right)$$

Proof sketch: We will state the basic intuition behind the proof; a complete proof is given on page 422 of [Weg87]. Start with a branching program for computing f. Partition the nodes of the branching program based on the input variable they are reading. Let t_i be the number of nodes reading a variable in b_i. We will prove that $t_i = \Omega\left(\frac{\log s_i(f)}{\log\log s_i(f)}\right)$. Notice that this is enough to prove the theorem because the size of the branching program is equal to the sum of the t_i's.

To prove the bound on t_i, notice that any setting of all variables outside b_i leaves a reduced branching program on the nodes reading variables in b_i. This branching program has at most t_i nodes, and must be computing one of the $s_i(f)$ sub-functions. It is easily seen by a counting argument that the number of different branching programs of size at most t_i is less than ${t_i}^{3t_i}$. Thus ${t_i}^{3t_i} > s_i(f)$, which gives $t_i = \Omega\left(\frac{\log s_i(f)}{\log\log s_i(f)}\right)$. □

Beame and Cook considered the *element distinctness problem.*

Definition 2.6 *Given n bits of input $x_1, x_2, \ldots, x_n$, divide them into $m = \lfloor\frac{n}{2\log n}\rfloor$ blocks $b_1, b_2, \ldots, b_m$ of consecutive bits such that every block b_i contains at least $2\log n$ input bits. Then $ED(x_1, x_2, \ldots, x_n) = 1$ if and only if there exists $i \neq j$ such that b_i and b_j, interpreted as integers, have the same value.*

Corollary 2.7 *(Beame and Cook) Every branching program computing $ED(x_1, x_2, \ldots, x_n)$ has $\Omega\left(\frac{n^2}{\log^2 n}\right)$ nodes.*

Proof sketch: Notice that $b_1, b_2, \ldots, b_m$ form a partition of the input. Beame and Cook proved that for any i, $\log s_i(ED) = \Omega(n)$, which combined with the previous theorem states that the size of any branching program computing the element distinctness function is $\Omega\left(\frac{n^2}{\log^2 n}\right)$. □

Notice that Nečiporuk's technique counts nodes associated with different input variables. Since there is no natural association of nodes and input variables in circuits, this technique has not yet been extended to the case of circuits. We remind the reader that the best unconditional size lower bound for circuits computing a function in NP is less than $3n$ [Blu84]; a circuit size lower bound of $\Omega\left(\frac{n^2}{\log^2 n}\right)$ would amount to a complexity breakthrough.

Chapter 3

SYMMETRIC FUNCTIONS ON BRANCHING PROGRAMS

Symmetric functions are a class of Boolean functions that depend only on the number of 1's in the input. They, being so natural, have been studied by many researchers. Nečiporuk's technique does not prove any non-trivial bounds for symmetric functions. It is intriguing that even for these most fundamental functions, we can not characterize their exact complexity on branching programs.

There is an obvious branching program (Corollary 3.11, to be proved later) of size $O(n^2)$ for computing the number of 1's in the input and therefore any symmetric function; and any branching program computing a non-trivial function must have at least n nodes. We would like to bridge the gap between these upper and lower bounds.

We will concentrate on two very natural classes of symmetric functions.

Definition 3.1 *For any $k, 0 \leq k \leq n$, the k-th* threshold *function, Th_k, is defined to be one if and only if at least k of the n inputs are one.* Majority *is the most interesting of these functions and is defined to be one if and only if at least half of the input bits are one.*

Definition 3.2 *For fixed a and d, the* mod *function, $mod_{a,d}$, is defined to be one if and only if the number of* 1*'s in the input is congruent to a* mod *d.*

We will start by summarizing some known lower and upper bound results, and will close the chapter by giving two constructions: an oblivious branching program of size $o(n \log^3 n)$ for computing any threshold function; and an oblivious branching program of size $o(n \log^4 n)$ for computing any mod function. All previously known constructions for these two classes of functions had size $\Omega(n^{3/2})$ [Lup65]. Our constructions are better described in terms of a new model *Modular Arithmetic*

Branching Programs (MA-programs) that we introduce in Subsection 3.2.3. There is an easy translation from MA-programs to branching programs, but presenting our construction in terms of MA-programs lets us highlight the key ideas of our construction.

3.1 Lower Bounds

Given the lack of success in proving strong lower bounds, many researchers have turned to the study of restricted branching program models. The hope is that the insight gained might eventually be of use in attacking the general model.

3.1.1 Constant Width Branching Programs

For the case of width two branching programs, Borodin et al. [BDFP86] proved an $\Omega\left(\frac{n^2}{\log n}\right)$ length lower bound for computing the majority function. Yao [Yao83] improved this to a super-polynomial lower bound. Shearer (unpublished) proved an exponential lower bound for the problem of checking whether the number of 1's in the input is a multiple of three. For width three or more, the best lower bound for a symmetric function is much smaller. Chandra et al. [CFL83] were the first to prove a super-linear $\Omega(nW(n))$ length lower bound on arbitrary constant-width branching programs. ($W(n)$ is the inverse of the Van der Waerden function and is less than 10 for all practical values of n.) They used Ramsey theoretic arguments and as we noted, their bound is barely super-linear; however, their result applies to the stronger rectifier-switching model which is one of the natural nondeterministic extensions of the branching program model. For the class of non-constant threshold functions, Barrington and Straubing [BS91], using algebraic techniques, improved this bound to $\Omega(n \log\log n)$. Alon and Maass [AM88] and Babai et al. [BPRS90] independently proved that any oblivious branching program of width $w \leq \sqrt{n}$ for majority has length $\Omega(\frac{n \log n}{\log w})$. Their bounds apply to all but a vanishingly small fraction of symmetric functions and are currently the best known length lower bounds for oblivious branching programs computing symmetric functions. Since any constant-width branching program can be transformed into an equivalent oblivious branching program with a constant blow-up in length and width, we get that for computing all but a vanishingly small fraction of symmetric functions, any constant-width branching program requires length $\Omega(n \log n)$. We will give some

intuition on this result in the next subsection.

3.1.2 Communication Complexity Technique

Some of the best known lower bound results on branching programs (including [AM88] and [BPRS90]) are based on *communication complexity.*

Informally, communication complexity measures the amount of communication needed to compute the given function when two parties are each given half of the input.

We assume that two processors (conveniently named *Alice* and *Bob*) with unlimited computational power are each given one half of the input and their goal is to co-operatively compute the given function with as little exchange of information as possible. Communication complexity is defined as the minimum number of bits of information that Alice and Bob have to exchange on any input. In general, the communication complexity depends on how we partition the input. We will consider two variations:

(1) *worst-partition* communication complexity measures the maximum amount of communication needed, over all possible partitions.

(2) *optimal-partition* communication complexity measures the minimum amount of communication needed, over all possible partitions.

The communication complexity measure was first introduced by Yao [Yao79] and has turned out to be very useful in a number of contexts (e.g., circuit depth lower bound, VLSI time area trade-off). There are several papers containing detailed and precise treatment (see, for example, Karchmer's thesis [Kar89]).

As a warm up, we will outline a very simple lower bound argument on the size of OBDDs.

Proposition 3.3 *Any OBDD computing a function with optimal-partition communication complexity C has at least 2^C nodes.*

Proof: Consider any OBDD P computing a function f with optimal-partition communication complexity C. We will define a communication game for computing f. Let the order of variables for P be $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$. Alice gets the first $\frac{n}{2}$ variables

in this order and Bob gets the remaining variables. Both Bob and Alice also have their individual copies of P. Alice starts by simulating the computation of P until she encounters a variable that doesn't belong to her private set. She then communicates the identity of the node that is reached by giving its index. Then Bob takes over and completes the simulation of P.

This protocol exchanges only $\log m$ bits where P has m nodes. Because $C > \log m$, P has at least 2^C nodes. □

To give some intuition on the length lower bound on the oblivious branching program ([AM88] and [BPRS90]), consider the simpler case of proving lower bound on an oblivious branching program P with the following restriction: If the length of P is $n\gamma$ then the levels can be divided into γ groups of consecutive levels such that all the odd numbered groups of levels (first, third, fifth etc.) read the first $\frac{n}{2}$ input variables and all the even numbered groups of levels (second, fourth etc.) read the last $\frac{n}{2}$ input variables.

We claim that the communication complexity (under the partition in which the first player receives the first half of the inputs) of the function being computed by this branching program is at most $\gamma \log w$, where w is the width of the branching program. This is very easily proved by the following communication game:

Alice receives the first $\frac{n}{2}$ input variables and Bob receives the remaining input variables. Both players also have their individual copies of P. The two players alternately simulate the computation of P. Whenever the player doing the simulation encounters a variable that does not belong to its private set, it communicates the identity of the node that is reached and the other player takes over.

There are γ switch-overs between players and each switch-over involves $\log w$ bits of communication, giving a total of $\gamma \log w$ bits of communication. If P is computing a function with communication complexity C (under the partition in which the first player receives the first half of the inputs), we obtain a lower bound of $\frac{C}{\log w}$ on γ and a lower bound of $\frac{nC}{\log w}$ on the length of the branching program.

The heart of [AM88, BPRS90] is a Ramsey like theorem which states that given any oblivious branching program, there is a restriction that leaves a significant portion of the variables unset and results in a branching program with a structure similar to what we have described above.

Theorem 3.4 *[AM88, BPRS90] For any function f, and any $m \leq n$, let $C_f(m)$ denote the maximum worst-partition communication complexity of f under a restriction that leaves m variables unset (the maximum is taken over the set of restrictions). Then there is a constant $b > 0$ such that every oblivious branching program of width $w \leq \sqrt{n}$ for computing function f has length at least $\frac{n\gamma}{\log w}$, where*

$$\gamma = \Omega\left(C_f\left(\frac{n}{b^\gamma}\right)\right)$$

The previous theorem applies only to oblivious branching programs. For the case of arbitrary branching programs, Pudlák [Pud84] used a Ramsey theoretic argument to prove an unconditional size lower bound of $\Omega\left(\frac{n \log\log n}{\log\log\log n}\right)$ for computing most threshold functions (including majority). Babai et al. [BPRS90] improved this to an unconditional size lower bound of $\Omega\left(\frac{n \log n}{\log\log n}\right)$ for computing majority. This bound is a generalization of Theorem 3.4 and also applies to almost all symmetric functions.

Theorem 3.5 *[BPRS90] For any function f, and any $m \leq n$, let $C_f(m)$ denote the maximum worst-partition communication complexity of f under a restriction that sets exactly $n - m$ variables to zero (the maximum is taken over the set of restrictions). Then there is a constant $b > 0$ such that every branching program for computing function f has size at least $n\gamma$, where*

$$\gamma \log \gamma = \Omega\left(C_f\left(\frac{n}{b^\gamma}\right)\right)$$

The following is a corollary of Theorems 3.4 and 3.5.

Corollary 3.6 *For the problem of computing majority or $mod_{a,\lfloor\sqrt{n}\rfloor}$ for any a, every branching program has size $\Omega\left(\frac{n \log n}{\log\log n}\right)$ and every oblivious branching program of width $w \leq \sqrt{n}$ has length $\Omega\left(\frac{n \log n}{\log w}\right)$.*

Razborov [Raz90], and Karchmer and Wigderson [KW93] proved unconditional size lower bounds of $\Omega(n \log\log\log^\star n)$ for computing majority on nondeterministic extensions of the branching program model. Razborov proved it for rectifier-switching networks; Karchmer and Wigderson proved it for span programs. The

results are based on a characterization of nondeterministic branching program size in terms of the covering number of a set of functionals. These are the only unconditional super-linear size lower bounds known on nondeterministic branching programs.

3.2 Upper Bounds

In the direction of upper bounds, progress has been even more elusive.

3.2.1 Constant Width Branching Programs

The majority function has been the subject of many studies. For a long time it was widely believed that any constant width branching program for computing majority must have super-polynomial length. The lower bound results of Borodin et al. [BDFP86] and Yao [Yao83], described earlier, were seen as steps leading to a proof of super-polynomial lower bound on the length of constant width branching programs computing majority.

Barrington [Bar89], in a very surprising result, proved that there are polynomial size branching programs of width five for computing majority. In fact, his result is much more general and applies to all functions computable by fan-in two Boolean circuits of logarithmic depth (which includes all symmetric functions).

Theorem 3.7 *[Bar89] For any function $s(n)$ which is at least polynomial, constant width branching programs of size $s(n)^{O(1)}$ compute exactly the same class of functions as fan-in 2 Boolean circuits of depth $O(\log s(n))$.*

An efficient simulation of branching programs by circuits was already known. Barrington [Bar89] proved that a Boolean circuit of depth d can be simulated by a branching program of width five and size $d4^d$. He considered a restricted class of branching programs which can be thought of as computing permutations. For a given Boolean circuit, he gives a simple recursive way of constructing an equivalent branching program belonging to this restricted class. The width five comes from the fact that the smallest unsolvable permutation group has order five. The size blow-up in the simulation of circuits by branching programs was later improved by Cai and Lipton [CL89], and Cleve [Cle91]. Cleve showed that any formula of size t

can be simulated by a branching program of width w and length $t^{1+1/O(w)}$. Unfortunately, in all these constructions, the resulting branching programs for computing majority are not efficient in size; they are, in fact, larger than the obvious $O(n^2)$ construction.

3.2.2 Computing Sum of the Bits

For any input $\vec{x}$, let $\|\vec{x}\|$ denotes the number of 1's in $\vec{x}$. The most obvious way to compute any symmetric function is to first compute $\|\vec{x}\|$. Unfortunately we pay heavily for this direct approach. The following proposition says that at least for the restricted case of oblivious branching programs, computing $\|\vec{x}\|$ always requires size $\Omega(n^2)$.

Proposition 3.8 *Any oblivious branching program computing $\|\vec{x}\|$ on n-bit input $\vec{x}$ has size at least*

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}.$$

Proof: For any input variable x_i, define ℓ_i to be the highest numbered level where x_i is accessed. Assume without loss of generality that $\ell_1 < \ell_2 < \cdots < \ell_n$. It is easy to verify that level $(\ell_i + 1)$ has at least $(i+1)$ nodes, corresponding to $(i+1)$ different values of the partial sum $x_1 + x_2 + \cdots + x_i$. Therefore the size of the branching program is at least

$$(\text{Number of nodes in level } 0) + \sum_{i=1}^{n} \text{Number of nodes in level } (\ell_i + 1) = \sum_{i=1}^{n+1} i$$

□

One possible approach is to receive partial information about the input by computing many functions on $\|\vec{x}\|$, but modulo a set of small, pairwise relatively prime numbers. Our main technical tool is the following classical theorem which lets us construct an integer from its values computed modulo many small primes.

The Chinese Remainder Theorem: Given pairwise relatively prime numbers $p_1, p_2, \ldots, p_k$, the set of equations

$$x \equiv a_j \pmod{p_j} \qquad\qquad 1 \leq j \leq k$$

has a unique solution for x between 0 and $\prod p_j - 1$ given by

$$x = \left[\sum_{j=1}^{k} (a_j m_j)((m_j)^{-1} \bmod p_j) \right] \mod \prod p_j$$

where $m_j = \frac{\prod p_i}{p_j}$.

A proof can be found in any standard textbook on Number Theory (for example, Hardy and Wright [HW79]).

We will illustrate this approach by an example.

Definition 3.9 *For any $k \leq n$, the* exact-k *function, E_k, accepts an input $\vec{x}$ if and only if $\|\vec{x}\| = k$.*

Using the Chinese remainder theorem, it is enough to choose a set of primes with product greater than n and verify that the number of 1's is congruent to k modulo each one of those primes.

This requires branching programs for computing $\|\vec{x}\|$ modulo fixed positive integers. The next lemma will give the construction.

To facilitate the presentation of our constructions described later in this chapter, we will adopt a slightly different view of branching programs. In any branching program, a given input defines a path from the source node to one of the sink nodes, and we can view the branching program as routing inputs from the source node to sink nodes. According to our definition of branching programs, every leveled branching program contains exactly one node (the source node) in its level zero. Sometimes, it will be convenient to relax this to consider leveled branching programs with more than one node in level zero, where we have not specified which of these nodes is the source node. Instead, we will think of these leveled branching programs as routers from nodes in level zero to nodes in the last level, where fixing the input and the particular node in level zero from which the input starts fixes the sink node which this input will be reaching.

Proposition 3.10 *For any positive integer $q > 0$, there is an oblivious branching program with $n+1$ levels, each consisting of q nodes, numbered from 0 to $q-1$,*

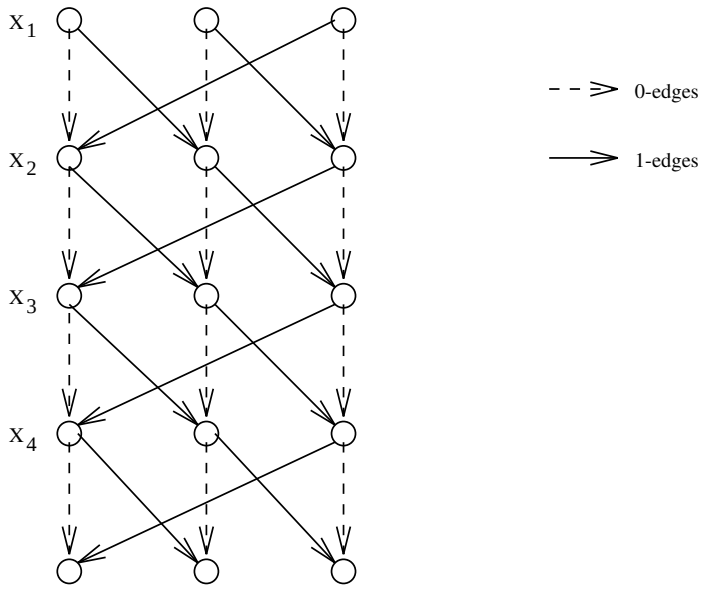


Figure 3.1: Branching program for computing $(x_1 + x_2 + x_3 + x_4) \bmod 3$

such that starting from node s, $0 \leq s < q$, in level zero, any input $\vec{x}$ reaches node $(s + \|\vec{x}\|) \bmod q$ in the last level.

Proof: We describe this oblivious branching program. All the out-edges from level i access the input variable x_i. Transitions between adjacent levels of nodes are defined as follows: node j makes a transition to node j of the next level if its associated variable is zero; otherwise, it makes a transition to node $(j + 1) \bmod q$ of the next level. (See Figure 3.1). □

Corollary 3.11 *There is an oblivious branching program of size $O(n^2)$ for computing $\|\vec{x}\|$ and therefore any symmetric function.*

We can build more complex branching programs by interconnecting many such routers.

We will be repeatedly using the following corollary of the prime number theorem. (A proof can be derived from [RS62, Corollary 1 and 2].)

Theorem 3.12 *For any constant $c > 0$, there exist constants N, $c' > 0$ such that for all $n \geq N$, there are at least $\left(\frac{c \log n}{\log \log n}\right)$ primes between $\log n$ and $c' \log n$.*

We can use this corollary to construct efficient branching programs for computing exact functions.

Proposition 3.13 *There is an oblivious branching program of size $O\left(\frac{n \log^2 n}{\log \log n}\right)$ for computing any exact function on n inputs.*

Proof: From the Chinese remainder theorem, in order to compute E_k, it is enough to choose a set of primes whose product is greater than n and verify that the number of 1's is congruent to k modulo each one of these primes. From Proposition 3.10, the resulting branching program will have size equal to $(n+1)$ times the summation of all the primes. If we use the previous theorem to choose $\lceil \frac{\log n}{\log \log n} \rceil$ primes of size $\Theta(\log n)$ each such that their product is greater than n, then the summation of all the primes is $O\left(\frac{\log^2 n}{\log \log n}\right)$ and therefore the resulting branching program is of size $O\left(\frac{n \log^2 n}{\log \log n}\right)$. □

Lupanov [Lup65] used the idea of receiving partial information by computing modulo small primes, combined with a trick of identifying common subcomputations, to beat the trivial $O(n^2)$ bound for computing any symmetric function. He constructed oblivious branching programs of size $O(\frac{n^2}{\log n})$ for computing arbitrary symmetric functions. We will prove this later as Theorem 3.21. We will first describe a model which simplifies the presentation of Lupanov's result as well as our improved constructions of branching programs for computing threshold and mod functions.

The building blocks of all these constructions are branching programs described in Proposition 3.10. Since the behavior of these branching programs, on any input $\vec{x}$, depends only on $\|\vec{x}\|$, it is easier to describe them in terms of a new model that operates on integers. The basic element of this model are routers that will capture the behavior of branching programs described in Proposition 3.10.

3.2.3 Modular Arithmetic Branching Programs (MA-programs)

Definition 3.14 *An MA-program(Modular Arithmetic Branching Program)* Γ *is defined by a triplet* (B, M, v)*, where* B *is a set of* boxes*, and* M *defines the* connection *between the boxes. We will shortly explain* v*.*

Each box has an associated integer constant $q > 0$*, which is called the* modulus *of the box. A box with modulus* q *(also called a* q*-box) consists of* q input nodes *and* q output nodes*, each of which is numbered from* 0 *to* $q - 1$*.*

The connection M *maps a subset of the set of output nodes of each box to the set of input nodes of other boxes; the output nodes which are not mapped by* M *are called* sinks*. We further assume that this mapping does not result in any cycles.* v *is one of the input nodes of* Γ *that is designated as the* source *node;*

We will be mainly interested in defining the sink node that is reached by any particular integer x starting from the source node v as follows: if x reaches an input node s of a q-box, it is routed to the output node $(x + s) \bmod q$; if x reaches a non-sink output node t of some box, it is routed to the input node that the connection M maps it to.

Definition 3.15 *An MA-program* Γ *separates two disjoint sets of integers* S_1 *and* S_2*, if for all* $x \in S_1, y \in S_2$*,* x *and* y *reach different sink nodes in* Γ*.*

The lemma below shows that for computing symmetric functions there is an easy translation from MA-programs to branching programs.

The *size* of any MA-program is defined as the summation of the moduli of all its boxes.

Lemma 3.16 *Let* f *be an* n*-variable symmetric Boolean function and* Γ *be an MA-program of size* S *that separates the sets*

$$\{\|\vec{x}\| \ : \ f(\vec{x}) = 0\}, \text{ and } \{\|\vec{x}\| \ : \ f(\vec{x}) = 1\}.$$

Then there is a branching program of size $(n + 1)S$ *that computes* f*.*

Proof: Let $\Gamma = (B, M)$. We will define a simple transformation to obtain a branching program P that computes f. We replace every q-box in B with a branching program described in Proposition 3.10, where the input nodes of the q-box correspond to the nodes in level 0 of the branching program, and the output nodes of the q-box correspond to the nodes in the last level of the branching program. If M maps an output node to an input node in Γ, we identify the corresponding nodes in P. The source and the sinks of Γ are designated the source and the sinks, respectively, of P.

Because each q-box in Γ is replaced with a branching program of size $(n+1)q$, the size of P is $(n+1)S$.

It is straightforward to verify that any input $\vec{x}$ reaches the same sink node in P as integer $\|\vec{x}\|$ in Γ. Because inputs reaching any sink node in P are either all from $f^{-1}(0)$ or all from $f^{-1}(1)$, we label all sink nodes receiving inputs from $f^{-1}(1)$ "accept" and all sink nodes receiving inputs from $f^{-1}(0)$ "reject." □

The above lemma implies that for n-variable symmetric Boolean functions, it is enough to study the behavior of MA-programs on the set $Z_{n+1} = \{i \mid 0 \leq i \leq n\}$.

MA-programs provide a general paradigm for computing many symmetric Boolean functions. But our constructions for computing threshold and mod functions have a particularly simple structure which can be described as a subclass of MA-programs called *chain MA-programs.*

Definition 3.17 *A chain MA-program is an MA-program with the added restriction that its boxes can be ordered such that the non-sinks of the ith box are mapped to the input nodes of the $(i+1)$-st box.* CHAIN$< p_1, p_2, \ldots, p_k >$*, for some $k \geq 1$, is the set of chain MA-programs with k boxes whose i-th box has modulus p_i (see Figure 3.2).*

The corollary below follows easily from Lemma 3.16.

Corollary 3.18 *Let f be an n-variable symmetric Boolean function and Γ be a chain MA-program of size S that separates the sets*

$$\{\|\vec{x}\| \ : \ f(\vec{x}) = 0\}, \text{ and } \{\|\vec{x}\| \ : \ f(\vec{x}) = 1\}.$$

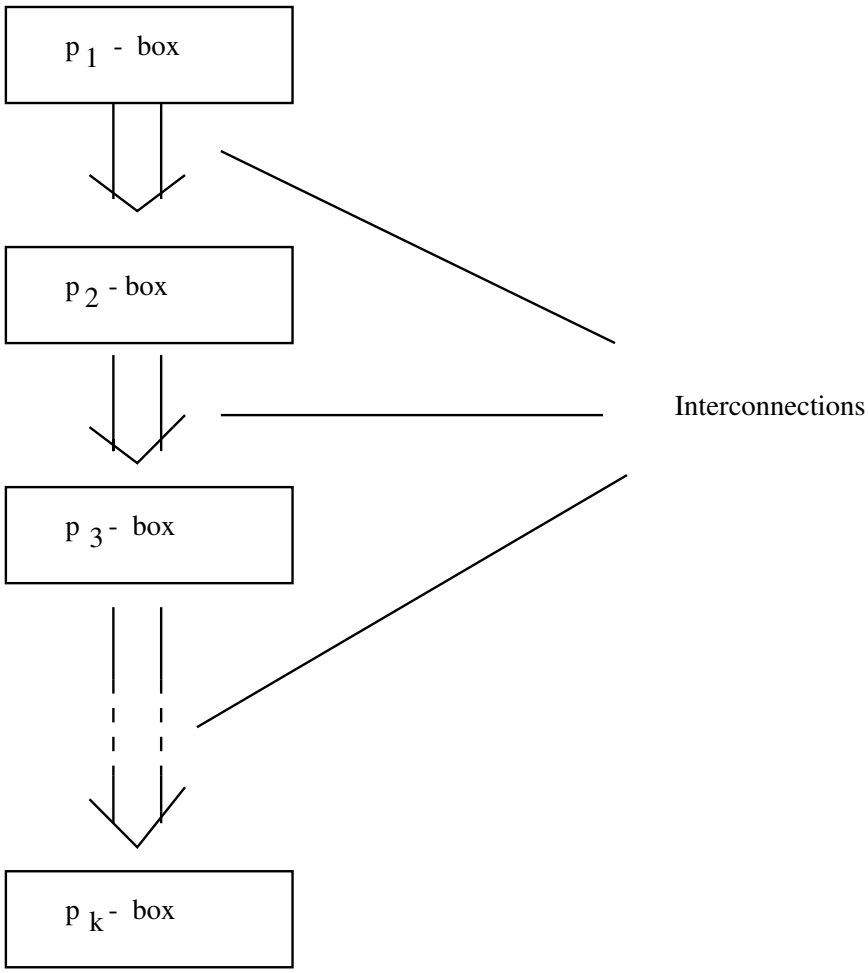


Figure 3.2: CHAIN$< p_1, p_2, \ldots, p_k >$

Then there is an oblivious *branching program of size* $(n+1)S$ *that computes* f.

For any MA-program Γ, we define its *modulus* to be the least common multiple of the moduli of the boxes in Γ. The importance of the modulus is explained in the lemma below.

Proposition 3.19 *Let* Γ *be an MA-program of modulus* m. *Then any two integers* x *and* y *satisfying* $x \equiv y \pmod{m}$ *reach the same sequence of input/output nodes in* Γ.

To keep our notation simple, we will extend the definition of input and output nodes so that for any integer s, the output(input) node s of a q-box will refer to the output(input) node $s \bmod q$. Also, at many places in our construction, it will be convenient to index a set of p elements as $iq \mod p$, $0 \leq i < p$, for some integer q that is relatively prime to p. The validity of this indexing scheme follows from

Fact 1 *For relatively prime positive integers p and q,*

$$\{iq \mod p \mid 0 \leq i < p\} = \{i \mid 0 \leq i < p\}$$

Our constructions will rely on certain transformations on MA-programs to obtain new MA-programs; this is explained in the proposition below.

Lemma 3.20 (Translation lemma) *Let Γ be an MA-program with modulus m. Then for any two integers a and l such that a is relatively prime to m, there is an MA-program Γ', with the same set of boxes as Γ, with the following property:*

There is a one-to-one mapping σ that for each box b, maps the set of input(output) nodes of b in Γ to the set of input(output) nodes of b in Γ' such that for any two integers y and x with $y \equiv ax + l \pmod m$, if x in Γ reaches the sequence of input/output nodes $v_1, v_2, \ldots, v_k$, where v_1 is the source and v_k is a sink, then y in Γ' reaches the sequence of input/output nodes $\sigma(v_1), \sigma(v_2), \ldots, \sigma(v_k)$.

Proof: We first define σ.

σ(input node s of box b in Γ) = input node $as - l$ of box b in Γ'.

σ(output node t of box b in Γ) = output node at of box b in Γ'.

The source and sinks of Γ' are respectively the image of the source and sinks of Γ under the mapping σ.

The connection in Γ' is defined as follows: if output node t is mapped to input node s in Γ then we map output node $\sigma(t)$ to the input node $\sigma(s)$ in Γ'.

Let x and y be as in the statement of the lemma. Then an easy induction on i, $1 \leq i \leq k$, will prove that the ith input/output node reached by $ax + l$ in Γ' is $\sigma(v_i)$. Then, from Proposition 3.19, we can infer that y reaches the same sequence of input/output nodes as $ax + l$, which proves the lemma. □

3.2.4 MA-programs for arbitrary symmetric function

Lupanov constructed a branching program of size $O\left(\frac{n^2}{\log n}\right)$ for computing any symmetric function. This result follows as a corollary of the next theorem.

Theorem 3.21 *[Lup65] For every partition of Z_{n+1} into two disjoint sets, there is an MA-program of size $O\left(\frac{n}{\log n}\right)$ separating these two sets.*

Proof: We will prove that for relatively prime numbers p and q and for every partitioning of Z_{pq} into sets S and T, there is an MA-program of size $p + q2^q$ separating S and T. Then choosing $q = \log n - 2\log\log n$, and p to be the smallest prime between $\lceil\frac{n+1}{q}\rceil$ and $2\lceil\frac{n+1}{q}\rceil$ will give us the desired size bound. (It is a standard result in Number Theory that for any integer x, there is at least one prime number between x and $2x$. See, for example, [RS62, Equation 3.9].) We will describe the construction informally. Start with a p-box and designate its input node 0 as the source node. For $0 \leq i < p$, define S_i to be the subset of integers from S that reach the output node i of the p-box. Similarly, define T_i to be the subset of integers from T that reach the output node i of the p-box. Let us define the q-signatures of the these sets as follows:

$$S_i^q = \{x \bmod q \mid x \in S_i\}$$

$$T_i^q = \{x \bmod q \mid x \in T_i\}$$

It is easy to verify that for every i, S_i^q and T_i^q form a partition of Z_q.

In order to separate S and T, it is enough to separate each of S_i and T_i and because of the above observation, we can separate S_i and T_i by using a q-box.

Our first attempt will be to use p q-boxes so that the output node i of the p-box is mapped to the input node 0 of the ith q-box. That, of course, gives an MA-program of size $p + pq > n$, which is less efficient than the trivial construction of size $n + 1$.

To reduce the size, notice that if $S_i^q = S_j^q$ (and therefore $T_i^q = T_j^q$), then both output nodes i and j of the p-box can be connected to the input node 0 of the same q-box.

There are 2^q possible values for the sets S_i^q, and all the output nodes with the same value of S_i^q can be connected to the same q-box. So we need only 2^q q-boxes, giving an MA-program of size $p + q2^q$.

The size bound can be further reduced to $p + 2^q$ by noticing that if S_i^q and S_j^q are cyclic shifts of each other then they can still be connected to the same q-box (but to different input nodes). This reduces the number of q-boxes by a factor of q but does not improve the asymptotic size bound. □

Applying Lemma 3.16, we get Lupanov's result for computing arbitrary symmetric functions.

Corollary 3.22 *[Lup65] There is an oblivious branching program of size* $O\left(\frac{n^2}{\log n}\right)$ *for computing any symmetric function on* n *inputs.*

3.3 Branching Programs for Threshold and Mod functions

We would like to improve the $O(n^2/\log n)$ upper bound (Corollary 3.22) on the size of branching programs computing symmetric functions.

Because nothing better is known for computing general symmetric functions, attention has turned to computing individual functions. In Proposition 3.13, we already saw an easy way to compute exact functions more efficiently. Unfortunately, there is no easy way to replicate this technique to other important classes of symmetric functions, for example, threshold and mod functions.

Lupanov [Lup65] constructed a branching program of size $O(n^{3/2})$ for computing any threshold function. No progress had been made on Lupanov's construction in nearly thirty years. Similarly, for the case of mod functions (that is verifying whether the number of 1's in the input is congruent to a modulo d, for fixed a and d), when d is a prime power close to $\sqrt{n}$, all previously known constructions for computing mod functions had size $\Omega(n^{3/2})$. (In Section 3.3.4, we outline an obvious construction of size $O(n^{3/2}\log n)$.)

This led Razborov [Raz91] to pose the following open problem:

Open Problem (Razborov [Raz91]) Does every rectifier-switching network computing the majority of n bits have size $n^{1+\Omega(1)}$?

We settle this problem in a strong negative way. Our main results on branching programs are:

Theorem 3.23 *There is an oblivious branching program of size*

$$O\left(\frac{n\log^3 n}{\log\log n\log\log\log n}\right)$$

for computing any threshold function on n inputs.

Theorem 3.24 *There is an oblivious branching program of size*

$$O\left(\frac{n\log^4 n}{(\log\log n)^2}\right)$$

for computing any mod function on n inputs.

For the case of threshold functions, the size bound is within $O\left(\frac{\log^2 n}{\log\log\log n}\right)$ of the size lower bound of Babai et al. [BPRS90] (see Corollary 3.6). Our method also yields a spectrum of branching programs, one for each width greater than $\log n$. For width $\log n \leq w \leq n$, the length of the resulting branching program is $O\left(\frac{n\log^2 n}{\log w\log\log\log n}\right)$, which is within $O\left(\frac{\log n}{\log\log\log n}\right)$ of the length lower bound of [AM88, BPRS90] (see Corollary 3.6). Our constructions are a generalization of Lupanov's construction for computing majority [Lup65].

Both of our constructions have other nice properties. For example, for any ℓ, between levels ℓn and $(\ell+1)n-1$, the variables are accessed in the order $x_1, x_2, \ldots, x_n$. This property is strongly reminiscent of the fixed ordering of the input variables in OBDDs. Hence we believe that our result may be of practical significance.

We first state our main technical theorem and then use it to construct MA-programs for computing threshold function. We then prove the main technical theorem by constructing MA-programs computing "approximate division." This should be of independent interest. We close this chapter by giving a construction of MA-programs for computing mod functions. We give a general scheme for converting an MA-program computing a threshold function to an MA-program computing the mod function. This transformation scheme requires that the MA-program computing threshold functions satisfy some additional properties. The

heart of the construction for MA-programs computing mod functions is an alternate construction of MA-programs for computing threshold functions that satisfies these additional properties. Although the alternate construction is slightly inefficient in size, compared to our first construction of threshold functions, it has many nice properties that may make it extensible for computing other classes of functions.

3.3.1 Main Technical Lemma

In order to study the usefulness of MA-programs in constructing branching programs for threshold and mod functions, we need the following definitions:

Definition 3.25 *An* interval *of length L is the set of integers $\{b+i \mid 0 \leq i < L\}$, for some integer b, which we will denote by $[b, b+L)$. For an interval I and for any integer t, which is at most the length of I, define I^t_{low} to be the set consisting of the t smallest integers in I and define I^t_{high} as $I \backslash I^t_{low}$. An* integer threshold problem *is a pair (I, t), where I is an interval and t an integer, at most the length of I. An MA-program solves this integer threshold problem if it separates I^t_{low} and I^t_{high}.*

If Γ is an MA-program that solves the integer threshold problem (Z_{n+1}, t), then by applying Lemma 3.16, we obtain a branching program that computes the threshold-t function.

We state our main technical theorem. There are two parts of this theorem: we will use part (A) to construct branching programs for computing threshold functions and part (B) to construct branching programs for computing mod functions.

Theorem 3.26 *Let $k \geq 2$ and let $p_1, p_2, \ldots, p_k$ be any set of pairwise relatively prime numbers with $p_k < \min\{p_2, p_3, \ldots, p_{k-1}\}$. Define $M = \prod_{1 \leq i \leq k} p_i$ and $M' = M/p_k$. For $t \leq L \leq M$, let I be the interval $[0, L)$. Then there is a $\mathcal{D} \in$ CHAIN$< p_1, \ldots, p_k >$ such that, restricted to integers in I, any sink node in $\mathcal{D}$ receives integers that are either all from I^t_{low} or all from I^t_{high} or all from $I_{mid} \subseteq I$, where,*

(A) If $L \leq (p_k - k + 2)M'$ then I_{mid} is an interval of length $(2k-2)M'$.

(B) If $t < M'$ then I_{mid} is a union of two intervals $[M - (k-2)M', M)$ and $[0, (k-1)M')$.

We will prove this theorem in Subsection 3.3.3. The proof uses the construction of an MA-program for computing *approximate division* (which we will define later). For intuition on why this theorem is useful for computing the integer threshold function, let us concentrate on part (A) of the theorem. Any sink which receives integers that are either all from I^t_{high} or all from I^t_{low} is properly separating these sets and can be ignored. We only have to take care of the sinks receiving integers from I_{mid}. If we choose k to be much smaller than p_k then I_{mid} is of a considerably smaller length than I. We take all sinks receiving integers from I_{mid} and connect them to another chain MA-program which we construct recursively for integers in I_{mid}.

Theorem 3.29, in the next section, makes this intuition precise. For our recursive construction, we will need to deal with intervals which do not necessarily start from zero. We restate Theorem 3.26 (A) for a general interval (not necessarily starting at zero) as a corollary, which can be deduced by applying Lemma 3.20 with $a = 1$.

Corollary 3.27 *Let $k \geq 2$ and let $p_1, p_2, \ldots, p_k$ be any set of pairwise relatively prime numbers with $p_k < \min\{p_2, p_3, \ldots, p_{k-1}\}$. Define $M = \prod_{1 \leq i \leq k} p_i$ and $M' = M/p_k$. For $t \leq L$, let I be an interval of length $L \leq (p_k - k + 2)M'$. Then there is a $\mathcal{D} \in \text{CHAIN}< p_1, \ldots, p_k >$ such that, restricted to integers in I, any sink node in $\mathcal{D}$ receives integers that are either all from I^t_{low} or all from I^t_{high} or all from $I_{mid} \subseteq I$, where, I_{mid} is an interval of length $(2k-2)M'$.*

3.3.2 MA-programs for Threshold Functions

We will construct a chain MA-program $\mathcal{T}$ of size $O\left(\frac{\log^3 n}{\log\log n \log\log\log n}\right)$ that solves the integer threshold problem (Z_{n+1}, t). Theorem 3.23 will then follow by invoking Corollary 3.18 to give the mapping between $\mathcal{T}$ and an oblivious branching program.

We will construct $\mathcal{T}$ in stages. Each stage consists of a chain MA-program from Corollary 3.27 which receives integers from a particular interval. It solves the given integer threshold problem correctly on some of the integers and passes the remaining integers — which come from an interval considerably smaller than the interval for the current stage — to the chain MA-program in the next stage.

Our construction for threshold functions will be using certain facts about prime numbers.

Proposition 3.28 *There is a constant C such that for all $L \geq C$, there exists $k > 0$ and k pairwise relatively prime integers $p_1, p_2, \ldots, p_k$ satisfying the following properties. Define $M = \prod_{1 \leq i \leq k} p_i$ and $M' = M/p_k$.*

(A) $p_1 + p_2 + \cdots + p_k = O\left(\frac{\log^2 L}{\log\log L}\right)$

(B) $L < M'(p_k - k + 2)$

(C) $(2k-2)M' < \frac{8L}{\log\log L} < L$.

Proof: From [RS62, Theorem 4] we know that there is a constant C_0 such that for $L \geq C_0$, the product of all primes between $\log L$ and $4\log L$ is at least $\frac{L}{4\log L}$. We choose enough primes $p_2 > p_3 > \cdots > p_k$ in this range so that their product is in $[\frac{L}{4\log L}, L]$. Moreover, because each of the primes is at least $\log L$, $k - 1 \leq \frac{\log L}{\log\log L}$. Choose p_1 to be a power of two such that $\prod_{1 \leq i \leq k} p_i \in [2L, 4L]$. Then $p_1 \leq 16\log L$, which proves (A).

Now we will prove (B).

$$\begin{aligned} M'(p_k - k + 2) &= \frac{M}{p_k}(p_k - k + 2) \\ &\geq \frac{2L}{p_k}(p_k - k + 2) \\ &= 2L - (k-2)\frac{2L}{p_k} \\ &> L \end{aligned}$$

for all $l \geq C_1$, where C_1 is a constant.

Finally $(2k-2)M' \leq \frac{2\log L}{\log\log L} \cdot \frac{4L}{\log L} < \frac{8L}{\log\log L} < L$ for all $L \geq C_2$, where C_2 is a large constant.

We complete the proof by choosing $C = \max(C_0, C_1, C_2)$. □

The main result on the size of a chain MA-program solving the integer threshold problem (Z_{n+1}, t) is the special case $L = n + 1$ of the following theorem.

Theorem 3.29 *Let $S(L)$ be the smallest number such that there exists a chain MA-program of size $S(L)$ that solves any fixed integer threshold problem on an interval of length L. Then*

$$S(L) = O\left(\frac{\log^3 L}{\log\log L \log\log\log L}\right).$$

Proof: We will prove the theorem by induction on L. Let C be the constant in the previous proposition.

Base case ($L < C$): If we use a C-box, every integer in the interval will reach a different sink node. The size of this MA-program is $C = O(1)$.

Induction step : Use the previous proposition to choose pairwise relatively prime integers $p_1, p_2, \ldots, p_k$ satisfying parts (A), (B), and (C). Let (I, t) be our integer threshold problem, where the length of I is L. From part (B), $L \leq (p_k - k + 2)M'$ so we can apply Corollary 3.27 to obtain $\mathcal{D}$. Consider the sinks which do not satisfy the property that they receive integers which are all from I^t_{high} or all from I^t_{low}. By Corollary 3.27, these sinks receive integers that are all from an interval I_{mid} of length at most $(2k-2)M'$. Our original problem has now been reduced to solving an integer threshold problem on the interval I_{mid}. By the induction hypothesis and part (C), this can be solved by a chain MA-program to which we connect all the sinks of $\mathcal{D}$ corresponding to I_{mid}.

From part (A), the size of $\mathcal{D}$ is $O\left(\frac{\log^2 L}{\log\log L}\right)$, and from part (C), the length of I_{mid} is less than $\frac{8L}{\log\log L}$.

So, $S(L) \leq O\left(\frac{\log^2 L}{\log\log L}\right) + S(\frac{8L}{\log\log L})$.

The recursion goes for $O\left(\frac{\log L}{\log\log\log L}\right)$ levels and each level contributes size $O\left(\frac{\log^2 L}{\log\log L}\right)$, which proves our claim. □

Corollary 3.30 *For $\log n \leq w \leq n$, there is an oblivious branching program of width at most w and length $O\left(\frac{n\log^2 n}{\log w \log\log\log n}\right)$ for computing any threshold function.*

Proof sketch: The construction mirrors the one given in the previous proof. The only significant difference is that in the induction step, we choose primes $4w >$

$p_2 > p_3 > \cdots > p_k > w$. Then $k = O\left(\frac{\log n}{\log w}\right)$ and each level of the recursion contributes a branching program of length $kn = O\left(\frac{n \log n}{\log w}\right)$. □

In a *syntactic read-k* branching program, every input variable is read at most k times on any root to leaf path (see Borodin et al. [BRS93] for definitions, motivations, and a survey of results).

Corollary 3.31 *For $\log n \leq w \leq n$, there is an oblivious branching program of width at most w for computing any threshold function such that any variable is read $O\left(\frac{\log^2 n}{\log w \log\log\log n}\right)$ times on any root to leaf path.*

3.3.3 MA-programs for Approximate Division

This section contains the proof of Theorem 3.26. We will be using the following theorem which is our main technical contribution.

Note: In all our constructions of chain MA-programs in this section, the source node is the input node 0 of the first box and the sinks are the output nodes of the last box; we will refer to the output node t of the last box as the sink node t.

Theorem 3.32 *Let $k \geq 2$ and let $p_1, p_2, \ldots, p_k$ be any set of pairwise relatively prime numbers with $p_k < \min\{p_2, p_3, \ldots, p_{k-1}\}$. Define $M = \prod_{1 \leq i \leq k} p_i$, $M' = M/p_k$. Then there is a $\mathcal{D} \in \text{CHAIN}< p_1, p_2, \ldots, p_k >$ such that for all $0 \leq \ell < p_k$, integers from $[0, M)$ reaching the sink node $\ell M'$ belong to the set*

$$\{(\ell M' + i) \bmod M \mid 0 \leq i < (k-1)M'\}.$$

Note: If we could strengthen this theorem so that the integers from $[0, M)$ reaching the sink node $\ell M'$ belong to the interval $[\ell M', (\ell+1)M')$ then we would have a division of the interval $[0, M)$ into p_k *disjoint* subintervals. In the present form, we are performing an approximate division of $[0, M)$ into p_k *overlapping* subintervals. We refer to this as "division" because the MA-program is computing (or approximating) $\lfloor x/M' \rfloor$ on input x. We have the exact division for the case $k = 2$ which forms the basis of Lupanov's construction of branching programs of size $O(n^{3/2})$ for computing majority [Lup65]. In Theorem 3.43, we give some evidence of the difficulty of computing exact division for the case $k \geq 3$.

Proof: We will first describe the construction of $\mathcal{D}$. Then we will characterize the set of integers reaching any particular sink node and prove that it satisfies the conditions of the theorem.

Definition 3.33 *Let* $p_{k+1} = 1$, *and for all* i, $1 \leq i \leq k$,

$$N_i = \frac{M}{p_i p_{i+1}}.$$

We state two simple properties of the N_i's which we will need in the proof of correctness of our construction. These can be easily deduced from the definition of the N_i's.

Fact 2 *For all* $1 \leq i, j \leq k$,

(a) $(N_i, p_i) = (N_i, p_{i+1}) = 1$.

(b) *If* $j \notin \{i, i+1\}$ *then* $N_i \equiv 0 \pmod{p_j}$.

We now describe $\mathcal{D}$ by giving the connection between the ith box (p_i-box) and the $(i+1)$th box (p_{i+1}-box), for $1 \leq i < k$:

> For $0 \leq u < p_i$, the output node uN_i of the ith box is connected to the input node $-uN_i$ of the $(i+1)$th box.

Note that the connections are well-defined because of Fact 2(a). We will use the following lemma to determine the set of integers from $[0, M)$ that reach any particular sink node of $\mathcal{D}$.

Lemma 3.34 *Let* $X = \sum_{1 \leq j \leq k} a_j N_j$, *where* $0 \leq a_j < p_j$. *Then* X *reaches the sink node* $a_k M'$ *of* $\mathcal{D}$.

Proof: Applying Fact 2(b), we get

$$X \equiv a_1 N_1 \pmod{p_1}. \tag{3.1}$$

$$\text{For } 1 < j \leq k,\ X \equiv a_{j-1}N_{j-1} + a_jN_j \pmod{p_j}. \tag{3.2}$$

We will prove by induction on $j, 1 \leq j \leq k$, that X reaches the output node a_jN_j of the jth box, which will be enough to prove our claim because $N_k = M'$.

(Base case, $j = 1$): X starts at the source, which is the input node 0 of the first box, and the result follows from Equation 3.1.

(Induction step, $1 < j \leq k$): By the induction hypothesis, X reaches the output node $a_{j-1}N_{j-1}$ of the $(j-1)th$ box, which is mapped to the input node $-a_{j-1}N_{j-1}$ of the jth box. Applying Equation 3.2 proves the induction step. □

Continuing the proof of the theorem, let us now evaluate the set G_ℓ of integers from $[0, M)$ reaching the sink node $\ell M'$. Observe that for any integer $X \in [0, M)$, by solving Equations 3.1 and 3.2 repeatedly for $j = 1, 2, \ldots k$, we can determine a sequence of $0 \leq a_j < p_j,\ 1 \leq j \leq k$ such that $X = (\sum_{1 \leq j \leq k} a_jN_j) \bmod M$. Therefore, by Lemma 3.34, we can infer that G_ℓ is *exactly* the set

$$\{(\ell M' + \sum_{1 \leq j < k} a_jN_j) \bmod M \mid 0 \leq a_j < p_j, 1 \leq j < k\}.$$

Because $p_k < \min\{p_2, p_3, \ldots, p_{k-1}\}$, for $0 \leq a_j < p_j, 1 \leq j < k$,

$$a_jN_j < p_j\frac{M}{p_jp_{j+1}} = \frac{M}{p_{j+1}} \leq \frac{M}{p_k} = M';\ \text{hence,}$$

$$\sum_{1 \leq j < k} a_jN_j < (k-1)M',$$

and therefore,

$$G_\ell \subseteq \{(\ell M' + i) \bmod M \mid 0 \leq i < (k-1)M'\}$$

□

We are now ready to prove Theorem 3.26.

Proof: (of Theorem 3.26) For both part (A) and part (B), the MA-program $\mathcal{D}$ is obtained by applying Theorem 3.32. Then, restricted to integers in $[0, M)$, the sink node $\ell M'$ of the kth box receives integers in the set

$$G_\ell \subseteq \{(\ell M' + i) \bmod M \mid 0 \leq i < (k-1)M'\}.$$

If $\ell \leq p_k - k + 1$ then $\ell M' + (k-1)M' \leq M$.

$$\text{So,} \quad G_\ell \subseteq [\ell M', \ell M' + (k-1)M'). \tag{3.3}$$

On the other hand, if $p_k - k + 2 \leq \ell < p_k$ then

$$\begin{aligned} G_\ell &\subseteq [\ell M', M) \bigcup [0, \ell M' + (k-1)M' - M) \\ &\subseteq [M - (k-2)M', M) \bigcup [0, (k-1)M') \qquad (3.4) \\ &\text{(because } \ell M' \geq (p_k - k + 2)M' = M - (k-2)M' \text{ and } \ell M' \leq M) \end{aligned}$$

We will analyze these sets for each of part (A) and part (B) when the inputs are actually restricted to come from $I = [0, L) \subseteq [0, M)$.

(**Part (A)**) We claim that for all $0 \leq \ell < p_k$, $G_\ell \cap I$ is contained in an interval of length $(k-1)M'$. This is easily seen by considering two cases:

Case 1 ($\ell \leq p_k - k + 1$): The claim follows from Equation 3.3.

Case 2 ($\ell \geq p_k - k + 2$): Since $L \leq (p_k - k + 2)M'$, Equation 3.4 gives $G_\ell \cap I \subseteq [0, (k-1)M')$, which is also an interval of length $(k-1)M'$.

Now consider any sink that receives an integer from I^t_{low} as well as an integer from I^t_{high}. Because of the claim above, any integer reaching this output node must be at least $t - (k-1)M'$ and at most $t + (k-1)M' - 1$. Then, setting $I_{mid} = [t - (k-1)M', t + (k-1)M')$ proves the theorem.

(**Part (B)**) Since $t < M'$, from Equation 3.3, we know that for $1 \leq \ell \leq p_k - k + 1$, $G_\ell \subseteq [M', M) \subseteq [t, M)$, so that $G_\ell \cap I \subseteq [t, L) \subseteq I^t_{high}$. This implies that any sink node $\ell M'$, which receives integers from I^t_{low} as well as I^t_{high}, satisfies $\ell \geq p_k - k + 2$ or $\ell = 0$. Hence,

$$G_0 \bigcup \left(\bigcup_{p_k - k + 2 \leq \ell < p_k} G_\ell \right) \subseteq [0, (k-1)M') \bigcup [M - (k-2)M', M) = I_{mid},$$

where the containment follows from Equation 3.3 and Equation 3.4. □

3.3.4 MA-programs for Mod Functions

In this section, we will construct a branching program on n variables of size $o(n \log^4 n)$ that accepts an input $\vec{x}$ if and only if $\|\vec{x}\| \equiv a \pmod{d}$, for fixed a

and d. We may assume without loss of generality that $a = 0$ and $1 < d \leq n$, because if $d > n$ then the problem is equivalent to computing the *exact function* E_a (which is defined to be 1 if the number of 1's in the input equals a), for which there is a simple branching program of size $o(n\log^2 n)$ (Proposition 3.13), and if $a \neq 0$ we can pad the input with $(d-a)$ 1's and maintain the same asymptotic size.

Two of the obvious approaches for computing mod functions are: (1) Use the construction in Proposition 3.10 to compute the exact sum mod d with a branching program of size $\Theta(nd)$. (2) Compute the OR of $\left\lfloor \frac{n}{d} \right\rfloor + 1$ exact functions, $E_0, E_d, E_{2d}, \ldots$. Each exact function can be computed with a branching program of size $O\left(n \log^2 n\right)$, giving a total size of $O\left(\frac{n^2 \log^2 n}{d}\right)$.

The first construction is efficient for small values of d; whereas the second one is better for large values of d. Furthermore, if d is a product of very large or very small prime powers then we can combine the Chinese remainder theorem with these approaches to construct efficient branching programs for checking divisibility modulo d. The hardest case for all these approaches is when d is a prime power close to $\sqrt{n}$. In this case, all previously known constructions for computing mod functions have size $\Omega(n^{3/2})$.

Our construction is based on a simple yet general scheme (Lemma 3.37 presented later) for transforming MA-programs computing threshold functions to MA-programs computing mod functions. Notice that the set of inputs that need to be accepted are exactly

$$\{\vec{x} \ : \ \|\vec{x}\| = kd, \text{ for } 0 \leq k < t\},$$

where we set $t = \left\lfloor \frac{n}{d} \right\rfloor + 1$. We will use much of the machinery developed in the previous sections. By Lemma 3.16, we know that it suffices to construct an MA-program $\mathcal{M}$ that separates the sets of integers $S_{d,n}$ and $Z_{n+1} \backslash S_{d,n}$, where

$$S_{d,n} = \{xd \mid 0 \leq x < t\}.$$

To achieve this, we will first suggest a simple scheme to modify the MA-programs constructed in Theorem 3.29. The modification is based on the corollary below, which can be easily deduced from the translation lemma, Lemma 3.20. Then, we will show that this scheme almost works but fails to produce the desired MA-program. The rest of the proof will be the construction of an alternate MA-program

that allows the modification to work.

Corollary 3.35 *Let Γ be an MA-program with modulus m that separates the (disjoint) sets of integers S_1 and S_2. Then for all integers a, where $(a, m) = 1$, there is an MA-program Γ' of the same size and modulus as Γ that separates the (disjoint) sets R_1 and R_2, where for $i = 1, 2$,*

$$R_i = \{y \mid y \equiv ax \pmod{m} \text{ for some } x \in S_i\}$$

Let $\mathcal{T}$ be an MA-program of modulus $m > n$ which solves the integer threshold problem (Z_{n+1}, t). Assume for the moment that $(d, m) = 1$. One way to construct $\mathcal{M}$ is to apply Corollary 3.35 to $\mathcal{T}$ with $a = d$. Then $\mathcal{M}$ will separate the sets

$$\begin{aligned} \{y \mid y \equiv dx \pmod{m} \text{ for } 0 \leq x < t\} &= S_{d,n}, \text{ and} \\ \{y \mid y \equiv dx \pmod{m} \text{ for } t \leq x \leq n\}. & \end{aligned}$$

Unfortunately, the latter set does not include all of $Z_{n+1} \backslash S_{d,n}$. There are two possible ways to modify this approach. The first possibility is to modify $\mathcal{T}$ in a different manner so that it separates the sets

$$\begin{aligned} \{y \mid y \equiv dx \pmod{n+1} \text{ for } 0 \leq x < t\} &= S_{d,n}, \text{ and} \\ \{y \mid y \equiv dx \pmod{n+1} \text{ for } t \leq x \leq n\} &= Z_{n+1} \backslash S_{d,n}. \end{aligned}$$

Unfortunately we do not know how to do such a modification. The other possible approach which we actually take is the following: Notice that the MA-program $\mathcal{T}$ has a modulus considerably larger than the length of the interval for which it is solving the integer threshold problem. (This is because of property (B) in Proposition 3.28 and the fact that the modulus of $\mathcal{T}$ is the least common multiple of the moduli of MA-programs constructed in different stages.) The latter property was crucial for the proof of Theorem 3.26 (A) and all the constructions in Section 3.3.2. However, if we start with a $\mathcal{T}$ that solves the integer threshold problem over an interval of length equal to its modulus, we can transform it to obtain a branching program for computing the mod function.

Definition 3.36 *An MA-program solves the* strong *integer threshold problem* (I, t) *if it separates* I^t_{low} *and* I^t_{high}*, and its modulus is equal to the length of the interval* I*.*

Lemma 3.37 *For a fixed* $1 < d \leq n$*, if we have a chain MA-program* Γ *of size* S *that solves the strong integer threshold problem* (Z_m, t)*, where* $m > n$*,* $t = \left\lfloor \frac{n}{d} \right\rfloor + 1$*, and* $(d, m) = 1$*, then we have a chain MA-program of size* S *that separates* $S_{d,n}$ *and* $Z_{n+1} \backslash S_{d,n}$*.*

Proof: Apply Corollary 3.35 to Γ with $a = d$ to obtain another chain MA-program of size S that separates the sets

$$\begin{aligned} \{y \mid y \equiv dx \pmod{m} \text{ for } 0 \leq x < t\} &= S_{d,n}, \text{ and} \\ \{y \mid y \equiv dx \pmod{m} \text{ for } t \leq x < m\} &= Z_m \backslash S_{d,n} \supseteq Z_{n+1} \backslash S_{d,n}. \end{aligned}$$

□

The following theorem states that the chain MA-program promised in the statement of the previous lemma can be constructed.

Theorem 3.38 *For* $k \geq 2$ *and pairwise relatively prime numbers* $p_1 > p_2 > \cdots > p_k > 2k$ *let* $M = \prod_{1 \leq i \leq k} p_i$*,* $M' = M/p_k$*, and* $r = \max\{p_1, p_2, \ldots, p_k\}$*. Then, for any* $t < M'$*, there is a chain MA-program* $\mathcal{T}$ *of size* $O(k^2 r^2)$ *that solves the strong integer threshold problem* $([0, M), t)$*.* [1]

The next subsection is devoted to the proof of this theorem. For now, we use this theorem to prove our main theorem on the size of branching programs computing mod functions.

Proof: (of Theorem 3.24) Because d is at most n, d has at most $\frac{\log n}{\log \log n}$ prime factors greater than $\log n$. Theorem 3.12 says that we can choose $k = \frac{\log n}{\log \log n} + 1$ primes $p_1 > p_2 > p_3 > \cdots > p_k > \max(2k, \log n)$, of size $\Theta(\log n)$ each, such that

[1] If $t \geq M'$, we can still construct an MA-program of this size. Although it is not a chain MA-program, this program can still be mapped to an oblivious branching program. In any case, we only need $t < M'$ to prove Theorem 3.24.

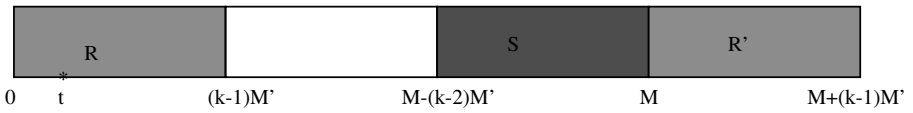


Figure 3.3: Intervals S, R, and R'

none of the p_i's divide d. It is clear that M' (in fact the product of any $k-1$ of these primes) is greater than n. Let $r = \max\{p_1, p_2, \ldots, p_k\}$.

Because $\left\lfloor \frac{n}{d} \right\rfloor + 1 \leq n < M'$, we can apply Theorem 3.38 to obtain a chain MA-program and then apply Lemma 3.37 followed by Corollary 3.18 to obtain the required oblivious branching program.

The size of the resulting oblivious branching program is

$$O\left((n+1)k^2r^2\right) = O\left(\frac{n\log^4 n}{(\log\log n)^2}\right).$$

□

3.3.5 MA-programs for Strong Threshold Problems

This subsection contains the proof of Theorem 3.38.

Proof: Since $t < M'$, we can apply Theorem 3.26(B) to construct $\mathcal{D} \in$ CHAIN$< p_1, p_2, \ldots, p_k >$ for integers in $I = [0, M)$. Then the sinks, that do not receive integers that are either all from I^t_{low} or all from I^t_{high}, receive integers in the set $I_{mid} = S \cup R$, where,

$$S = [M-(k-2)M', M), \text{ and } R = [0, (k-1)M').$$

Let $R' = \{M + x \mid x \in R\} = [M, M + (k-1)M')$ so that

$$S \cup R' = [M-(k-2)M', M + (k-1)M')$$

(see Figure 3.3). Suppose Ψ_1 and Ψ_2 are two chain MA-programs, each of modulus M, that solve the integer threshold problems $(S \cup R', M)$ and (R, t) respectively. Then, since $x \in R$ if and only if $x + M \in R'$, by Proposition 3.19, Ψ_1 also solves

the problem of separating S and R. Therefore, the chain MA-program obtained by first connecting those sinks of $\mathcal{D}$ reached by integers in I_{mid} to Ψ_1, and then connecting those sinks of Ψ_1 reached by integers in R to Ψ_2 solves the integer threshold problem $([0, M), t)$ and has modulus M— this is the chain MA-program $\mathcal{T}$ that we seek.

In order to construct Ψ_1 and Ψ_2, note that both $S \cup R'$ and R are intervals of length at most $2kM'$. We will show that for any integer threshold problem (I, t) where the length of I is at most $2kM'$, we can construct a chain MA-program of modulus M and size at most k^2r^2 that solves this problem. The construction is a special case of the lemma below with $j = k - 1$, and $b = 2k$. Before we state and prove the lemma, we can estimate the size of $\mathcal{T}$ which is

$$\text{Size of } \mathcal{D} + \text{Size of } \Psi_1 + \text{Size of } \Psi_2 \leq kr + 2(k^2r^2) = O(k^2r^2).$$

□

Notation: For $1 \leq j \leq k$, define $M_j = p_1 p_2 \ldots p_j$.

Lemma 3.39 *For $k \geq 2$ and pairwise relatively prime numbers $p_1 > p_2 > \cdots > p_k > 2k$, define $M_j = p_1 p_2 \ldots p_j$ for $1 \leq j \leq k$. Then for all $1 \leq j < k < b < p_k$, and any integer threshold problem (I, t) where I is an interval of length at most bM_j, there is a chain MA-program of modulus $p_k M_j$ and size at most j^2r^2, solving the problem.*

Proof: We will prove this by induction on j. For the base case ($j = 1$), the length of I is at most $bp_1 < p_k p_1 < r^2$. Thus we can use a $p_1 p_k$-box so that every element of I reaches a different sink node.

For the induction step, we will assume that the claim holds for $j - 1$ and show that it is true for j.

We would like to apply Corollary 3.27 but we have two restrictions that need to be satisfied. (1) The modulus of every box in the resulting chain MA-program should divide $p_k M_j$; this suggests using a partial product of p_i's as the modulus of each box. (2) Corollary 3.27 forces the modulus of the last box to be smaller than the modulus of all other boxes except the first.

We apply Corollary 3.27 to construct a chain MA-program $\mathcal{A}$ of modulus $p_k M_j$ using $\left\lceil \frac{j+1}{2} \right\rceil$ boxes with the following parameters:

Case 1(j is even): $\mathcal{A} \in \text{CHAIN}< p_1, p_2p_3, p_4p_5, \ldots, p_{j-2}p_{j-1}, p_jp_k >$

Case 2(j is odd): $\mathcal{A} \in \text{CHAIN}< p_1p_2, p_3p_4, \ldots, p_{j-2}p_{j-1}, p_jp_k >$

Notice that Corollary 3.27 can be applied because in each case, the modulus of the last box is smaller than the moduli of all other boxes (except the first box), and

$$\begin{aligned}
&\left(p_jp_k - \left\lceil \frac{j+1}{2} \right\rceil + 2\right) M_{j-1} \\
&\geq \left((p_k - 1)p_j + p_j - \frac{j+2}{2} + 2\right) M_{j-1} \\
&> \left(bp_j + 2k - \frac{j}{2} + 1\right) M_{j-1} \\
&> bp_jM_{j-1} = bM_j
\end{aligned}$$

From Corollary 3.27, we know that those sinks that do not receive integers that are all from I^t_{low} or all from I^t_{high} receive integers in an interval $I_{mid} = \hat{I}$ of length at most

$$2\left(\left\lceil \frac{j+1}{2} \right\rceil - 1\right) M_{j-1} \leq 2\left(\frac{j+2}{2} - 1\right) M_{j-1} = jM_{j-1} < bM_{j-1}.$$

There must be some t' such that $\hat{I}^{t'}_{low} \subseteq I^t_{low}$ and $\hat{I}^{t'}_{high} \subseteq I^t_{high}$. We can apply the induction hypothesis to obtain a chain MA-program $\mathcal{B}$ of modulus p_kM_{j-1} and size at most $(j-1)^2r^2$, solving the integer threshold problem $(\hat{I}, t')$.

Connecting the sinks of $\mathcal{A}$ (corresponding to $\hat{I}$) to $\mathcal{B}$ gives us the desired chain MA-program of modulus p_kM_j and size at most

$$\left\lceil \frac{j+1}{2} \right\rceil r^2 + (j-1)^2r^2 \leq \left(\frac{j+2}{2}\right) r^2 + (j-1)^2r^2 < j^2r^2.$$

□

3.4 Conclusion

We have shown a way of computing approximate division very efficiently. This allowed us to construct a nearly optimal size branching program for computing any threshold or mod function. We hope that our techniques can be applied to other classes of symmetric functions or to Boolean formulas for symmetric functions.

We presented our constructions in terms of MA-programs, which helped us highlight the key ideas of our constructions. Many of the previously known constructions are also more easily understood in terms of MA-programs (for example, [Lup65]). But even for computing symmetric functions, MA-programs do not capture the full generality of branching programs. They correspond to the subset of branching programs for which computation can be broken into blocks such that within any block, a function of the sum of the input bits is computed. It is quite possible that the most efficient constructions for symmetric functions do not follow this pattern. We already know of constructions (for example, Barrington [Bar89]; also see the discussion following Corollary 3.41) that can not be translated to MA-programs.

For computing majority, $E_{n/2}$, or $\mathrm{mod}_{0,\lfloor\sqrt{n}\rfloor}$. the best size lower bound is $\Omega\left(\frac{n\log n}{\log\log n}\right)$ [BPRS90], which translates to a size lower bound of $\Omega\left(\frac{\log n}{\log\log n}\right)$ on MA-programs. The proof is non-trivial and uses a very nice Ramsey theoretic lemma (Theorem 3.5). There is a well established tradition in theoretical computer science to look at more structured models if we are unable to prove lower bounds on general models. Given the limited success in proving lower bounds on branching programs, we suggest looking at MA-programs as a first step. As an evidence that they indeed are easier to prove lower bounds on, we give a simple argument for a stronger lower bound of $\Omega\left(\frac{\log^2 n}{\log\log n}\right)$ on the size of MA-programs computing majority, $E_{n/2}$, or $\mathrm{mod}_{0,\lfloor\sqrt{n}\rfloor}$.

Theorem 3.40 *For any functions f on integers, we define its* discriminator *$disc(f)$ to be the largest integer such that for all $\ell \leq disc(f)$, there are two inputs that are ℓ apart and are assigned different values by the function. Then any MA-program computing a function f with $disc(f) \geq n^{\epsilon}$ for $\epsilon > 0$ has size $\Omega\left(\frac{\log^2 n}{\log\log n}\right)$.*

Proof: Suppose Γ is the MA-program computing such a function, and let the set of moduli of the boxes in Γ be $\{p_1, p_2, \ldots, p_k\}$. Then by Proposition 3.19, the

least common multiple of $p_1, p_2, \ldots, p_k$ is at least n^ϵ. We will prove that this forces $\sum p_i$, the size of Γ, to be at least $\Omega\left(\frac{\log^2 n}{\log\log n}\right)$.

First divide the set of p_i's into two sets SMALL and LARGE, where SMALL consists of those p_i's that are less than $\frac{\epsilon \log n}{4}$ and LARGE consists of the remaining p_i's. It is known that the least common multiple of all integers between 1 and x is at most $(2.83)^x$ [RS62, Theorem 12]. Thus the least common multiple of all numbers in SMALL is at most $(2.83)^{\frac{\epsilon \log n}{4}} < (4)^{\frac{\epsilon \log n}{4}} = n^{\epsilon/2}$. Because the least common multiple of integers in the union of SMALL and LARGE is at least n^ϵ and the integers in SMALL have least common multiple at most $n^{\epsilon/2}$, the least common multiple of the integers in LARGE must be at least $n^{\epsilon/2}$. We will prove that the summation of the integers in LARGE, which is a lower bound on the size of Γ, is $\Omega\left(\frac{\log^2 n}{\log\log n}\right)$.

If any of the integers in LARGE is at least $\log^2 n$ we are done; otherwise, they are all less than $\log^2 n$. Because their least common multiple is at least $n^{\epsilon/2}$, there must be at least $\frac{\log n^{\epsilon/2}}{\log\log^2 n} = \frac{\epsilon \log n}{4 \log\log n}$ integers in LARGE, each one at least $\frac{\epsilon \log n}{4}$, proving that the sum of integers in LARGE is $\Omega\left(\frac{\log^2 n}{\log\log n}\right)$. □

Corollary 3.41 *Any MA-program computing Th_d or E_d for $1 \leq d \leq n$, or $mod_{0,d}$ for $d = \Omega(n^\epsilon)$ and $\epsilon > 0$ has size $\Omega\left(\frac{\log^2 n}{\log\log n}\right)$.*

This corollary highlights some limitations of the MA-program model. There are simple branching programs of linear size for computing AND and OR functions, but any construction based on MA-programs has size $\Omega\left(\frac{n \log^2 n}{\log\log n}\right)$. From the proof of Proposition 3.13, we get that the lower bound is tight for the case of exact functions, but there is a polylog gap between the currently known upper and lower bounds for the case of computing majority or $\mathrm{mod}_{0,\lfloor\sqrt{n}\rfloor}$.

In Theorem 3.32, we construct a $\mathcal{D} \in \text{CHAIN}_< p_1, p_2, \ldots, p_k >$ such that for all $0 \leq \ell < p_k$, integers from $[0, M)$ reaching the sink node $\ell M'$ belong to the set

$$I_\ell = \{(\ell M' + i) \bmod M \mid 0 \leq i < (k-1)M'\}.$$

When $k = 2$, integers in each I_ℓ belong to an interval of length M', such that the set of I_ℓ's form a partition of $[0, M)$. However, for $k \geq 3$, the I_ℓ's are overlapping intervals.

Definition 3.42 *An MA-program in* CHAIN$< p_1, p_2, \ldots, p_k >$ *computes* exact division *if*

(1) The sinks of the MA-program are the output nodes of the p_k-box, and

(2) for all $0 \leq \ell < p_k$, integers from $[0, M)$ reaching the sink node ℓ belong to an interval of length M', and these intervals, taken together, form a partition of $[0, M)$.

A chain MA-program computing exact division will obviate the need for recursion in Theorem 3.29 (because, by padding, we can make the threshold t a multiple of M'), thereby giving an optimal size chain MA-program for computing majority. More importantly, it will greatly simplify both our constructions. Unfortunately, we can prove that for $k \geq 3$, a chain MA-program computing exact division does not exist. Therefore any construction for computing exact division has to have a more complicated structure.

Theorem 3.43 *For $k \geq 3$, exact division is not possible using a chain MA-program composed of p_j-boxes, $1 \leq j \leq k$, where $p_1, p_2, \ldots, p_k$ are relatively prime and each p_j is at least 2.*

Proof: For $1 \leq j \leq k$, define $M_j = p_1 p_2 p_3 \cdots p_j$.

The following claim easily follows from the relative primality of M_{j-1} and p_j, and Proposition 3.19 applied to the first $j-1$ boxes.

Claim 1 **:** For $1 \leq j \leq k$ and restricted to an interval of length M_j, every output node of the p_j-box receives M_{j-1} integers.

This claim says that, restricted to integers in $[0, M)$, there are exactly $M_{k-1} = M'$ integers reaching any output node of the p_k-box. We would like to prove that these M' integers do not come from an interval of length M'. We will prove this by contradiction. Suppose that these M' integers come from the interval $[b, b + M')$. Then we can make several claims restricted to the integers in this interval.

Claim 2 **:** Every input node of the p_k-box receives a multiple of p_1 integers.

From Claim 1 (for $j = k-1$), every output node of the p_{k-1}-box receives M_{k-2} integers, which is a multiple of p_1. Claim 2 holds because every input of the p_k-box receives integers from zero or more output nodes of the p_{k-1}-box.

Claim 3 : At least one input node of the p_k-box receives two integers that are different modulo p_k.

We will prove Claim 3 by contradiction. Partition the interval $[b, b + M')$ into G_i's for $0 \leq i < p_k$, where

$$G_i = \{x \mid x \equiv i \pmod{p_k}\} \bigcap [b, b + M').$$

If the present claim is not true then integers reaching every input node of the p_k-box all come from the same G_i, and from Claim 2, every input node of the p_k-box receives some multiple of p_1 integers. That implies that p_1 divides $|G_i|$ for all $0 \leq i < p_k$. Because $[b, b + M')$ is an interval,

$$\max\{|G_i|\} - \min\{|G_i|\} \leq 1.$$

Because $\sum |G_i| = M'$ is not divisible by p_k (= the number G_i's), the $|G_i|$'s can not all be equal. Therefore

$$\max\{|G_i|\} - \min\{|G_i|\} = 1.$$

But p_1 divides every $|G_i|$ so that p_1 divides $\max\{|G_i|\} - \min\{|G_i|\})$; that is, p_1 divides 1. This is a contradiction, so Claim 3 must be true.

To complete the proof of the theorem, notice that the integers reaching the input node specified in Claim 3 are going to reach different output nodes of the p_k-box. □

Constant width branching programs present an interesting challenge. The length lower bound on oblivious branching programs in [AM88, BPRS90] (Corollary 3.6) also applies to the function $E_{n/2}$ and is tight for width $w \in \Omega(\log n)$ (by choosing primes of size $\Theta(w)$ in the proof of Proposition 3.13), but there is a relatively large gap between lower and upper bounds for width $w \in o(\log n)$. Because of Proposition 3.19 and the fact that the product of all primes less than x is $2^{O(x)}$ ([RS62, Equation 3.15]), any MA-program computing a function with non-constant discriminator (we define discriminator in Theorem 3.40) has at least one box with non-constant modulus. In other words, for computing functions with non-constant discriminators, MA-programs can not be used to construct constant

width branching programs. In particular, discriminator of every function in Corollary 3.41 is $\Omega(n^\epsilon)$, so MA-programs can not be used to construct sub-logarithmic width branching programs for computing any of those functions. MA-programs capture one very natural way of computing symmetric functions. Concentrating on constant width branching programs will force us to look beyond MA-programs, hopefully resulting in yet another general technique for computing symmetric functions.

We already know of polynomial size constant width branching programs for computing majority. But all such constructions are based on Barrington's construction [Bar89], which is indirect in nature as it starts from a Boolean formula and constructs an equivalent branching program. To the best of my knowledge, the resulting branching programs for specific functions (like majority) have no easy description. In order to better understand the computational power of constant width branching programs, it is important to have alternate constructions of constant width branching program that have simple, short descriptions.

Open problem: Construct a simple branching program of polynomial size and width $o(\log n)$ for computing majority.

Open problem: Can the length lower bound in [AM88, BPRS90] be improved for width $w \in o(\log n)$?

Chapter 4

CREW PRAMS VERSUS EREW PRAMS

4.1 Introduction

Parallel random access machines (PRAMs) have been the model of choice for describing parallel algorithms on shared memory machines.

A PRAM consists of a number of processors and a global shared memory. Each processor has its local storage and control. The global shared memory consists of *cells*, which can be individually addressed and are of equal size, called the *word-size* of the machine. Any processor is allowed to access any cell.

Global memory cells are of two types: for computing a function on n inputs, we have n *read-only* cells which store the input; the remaining cells can be read as well as written and are called *common cells*. The processors of the PRAM act synchronously. Computation proceeds in rounds consisting of "read", "compute", and "write" cycles. In the "read" cycle, every processor possibly reads a cell; it then performs some computation during the "compute" cycle and may write into one of the (common) cells in the "write" cycle.

One of the common cells is designated the output cell and its contents at the end of the computation is designated to be the output. We measure *time* as the number of rounds in the computation.

Depending on whether or not we allow more than one processor to concurrently read from or write to a memory cell, we obtain different models of PRAMs and complexity classes associated with them. In the case of *exclusive* access, we allow at most one processor to access any particular cell, whereas in *concurrent* access we allow any number of processors to access any given cell at the same time. The three most popular models are the CRCW (concurrent read, concurrent write), CREW (concurrent read, exclusive write), and EREW (exclusive read, exclusive write) PRAMs (see [JáJ92, Fic93].) In the case of concurrent write, we need some

way to arbitrate write conflicts. There are several variations; we will describe two of them. In a *priority* CRCW PRAM, processors are assigned distinct priorities and in the case of more than one processor trying to write into the same cell, the cell receives the value that the highest priority processor is trying to write. In the *common* CRCW PRAM model, any valid algorithm must ensure that all processors trying to write into any given cell at the same time are all trying to write the same value, which becomes the contents of the cell at the end of the write step.

The PRAM model has been very successful for developing algorithms. A large body of algorithms has been discovered for a variety of problems (see, for example, [JáJ92]). Unfortunately, we have not had the same kind of success in proving lower bounds. We will survey some of the known lower bound results.

Cook, Dwork, and Reischuk [CDR86], by an elegant argument, showed that any CREW PRAM takes $\Omega(\log n)$ time to compute the OR of n bits. The result was improved by Kutyłowski [Kut91] and Dietzfelbinger et al. [DKR94] who determined the exact complexity of OR.

The results in [CDR86, Kut91, DKR94] (as well as many other lower bound results on PRAMs) all have a similar flavor: they prove that a CREW PRAM running for a small number of steps can distinguish very few inputs. At any step of a given PRAM algorithm, we can partition the input into equivalence classes so that the algorithm can not distinguish between two inputs belonging to the same equivalence class. Clearly at the end of the computation, inputs belonging to any particular equivalence class are all going to be either accepted or rejected. At the start of the computation, the algorithm can not distinguish any two inputs, that is, all the inputs belong to the same equivalence class. As the computation progresses, the partitioning of equivalence classes gets finer and finer. There are several measures (for example, sensitivity, certificate complexity, polynomial degree, granularity; see Nisan [Nis91] for definitions) to estimate how fine the partitioning of inputs is and therefore to measure the "progress" of the computation.

A lower bound proof consists of choosing one of these measures and then showing that any algorithm running for a short time makes less progress than what is needed to compute the given function.

Nisan [Nis91] gave several characterizations of complexity of any function on

CREW PRAMs in terms of these measures.

Beame and Håstad [BH89], in a very important breakthrough, proved optimal $\Omega\left(\frac{\log n}{\log\log n}\right)$ bounds for computing the parity of n bits on a priority CRCW PRAM for the case when either the number of processors or the number of shared memory cells is bounded by a polynomial. Their proof combines the restriction technique (used to prove size lower bounds on constant depth circuits [Hås87, FSS81]) with a degree argument and also applies to many other functions such as computing the sum of the bits.

The results we have described so far are very appealing because they place absolutely no restrictions on the instruction set of individual processors or the word-size of the machine. So lower bounds are more of a communication lower bound. That is, they show the intrinsic limitation of global memory as a medium of communication. The results of [CDR86, Kut91, DKR94] do not put any restrictions on the number of processors either.

4.2 Communication Width

The PRAM model provides a very high level abstraction of real parallel machines. This is an attractive feature for the programmer, but this ease of programming comes with a heavy price: implementing PRAMs in hardware is non-trivial. In particular, no efficient implementation of global shared memory is known. There has also been a growing realization among practitioners that communication is the single most precious resource in parallel computing. Because shared memory is the only medium of communication among processors, and restricting the size of shared memory will restrict the number of messages that can be concurrently transmitted, it is important to treat the size of shared memory as a very important complexity measure.

We define *communication width* of a PRAM to be the number of common cells, that is cells that are available for reading as well as writing. We denote by EREW(m), CREW(m), and CRCW(m) the respective PRAM models with communication width m. Vishkin and Wigderson [VW85] initiated the study of PRAMs with bounded communication width. Their paper gives several motivations for studying such models. For example, the "Ethernet" can be thought of

as a PRAM with one common cell. They proved lower bounds on the CREW(m) model in terms of two measures – *everywhere sensitivity* and *somewhere sensitivity*. Their results imply an $\Omega(\sqrt{n/m})$ lower bound on the problem of computing the PARITY of n bits and an $\Omega\left(\left(\frac{n}{m}\right)^{1/3}\right)$ lower bound on the complexity of the OR function. Beame [Bea86], by considering a different measure, granularity, proved a stronger $\Omega(\sqrt{\frac{n}{m}})$ lower bound on the OR function. Azar [Aza92] analyzed the threshold function Th_k on priority CRCW PRAM(m) with a polynomial number of processors. He proved a lower bound of $\Omega\left(\frac{k}{m}\right)$ for the case $k \leq n^{1/2-\epsilon}$ and a lower bound of $\Omega\left(\frac{n^{1/2-\epsilon}}{m}\right)$ for $k \geq n^{1/2-\epsilon}$, where ϵ is any number greater than zero.

Mansour et al. [MNV94] consider a CRCW(m) model with $n >> p >> m$ (where p is the number of processors) and determine the exact complexity of the list reversal problem. Their paper also contains several justifications for restricting the communication width of the model.

4.3 Separating Different Variants of PRAMs

A basic issue in parallel complexity theory is to understand the relative power of different variants of PRAMs. When computer architects propose new designs, a lot of effort goes into selecting the right set of primitives. Providing a very powerful set of primitives usually means increased hardware and software cost, and it is not always clear that having powerful primitives makes it easier to solve problems. Determining relative power can take one of two paths: (1) we can either show that the models have comparable power by means of a simulation or (2) we can prove a separation between their powers, where a separation result entails (a) the choice of a problem whose performance is interesting to study, (b) an efficient algorithm for that problem on the stronger model, and (c) a lower bound proof that no algorithm on the weaker model can match the performance on the stronger model.

Both kinds of results have their rewards: In the case of an efficient simulation of the powerful variant by a seemingly weaker variant, algorithms can be designed on the powerful variant and then translated to the weaker variant for implementation purposes. On the other hand, a separation result gives at least a partial justification for providing the set of primitives available on the stronger model because certain problems are provably more efficient in the presence of those primitives.

Boppana [Bop89] proved that given n integers, determining whether they are all distinct requires at least $A(n,p)$ time on a common CRCW PRAM with $p \geq n$ processors, where

$$A(n,p) = \frac{n \log n}{p \log\left(\frac{n \log n}{p} + 1\right)}.$$

Because this problem has an $O(1)$ time algorithm on a priority PRAM with n processors, this result shows a separation between the priority and common variants of the n processor CRCW PRAM model. Boppana, generalizing the results of Kučera [Kuč82] and Chlebus et al. [CDHR88], also showed that one step of any n-processor priority CRCW PRAM can be simulated in $O(A(n,p))$ steps by a common CRCW PRAM with p processors.

Boppana's result uses Ramsey's theorem and his lower bound argument does not work if the input integers are restricted to come from a small domain. Edmonds [Edm91] improved Boppana's result to prove the same lower bound even if the input integers are restricted to come from a domain of size $2^{\theta(n)}$.

Because the OR of n bits can be computed easily in constant time on a CRCW PRAM, the $\Omega(\log n)$ lower bound results of [CDR86, Kut91, DKR94] prove a separation between the powers of CRCW and CREW PRAMs.

Since CRCW, CREW and EREW are the three most popular variants of the PRAM model and separations between variants of the CRCW, and between the CRCW and CREW models have already been shown, the big open problem is to determine the relative power of CREW and EREW PRAMs.

4.4 Previous Attempts at Separating CREW and EREW PRAMs

Snir [Sni85] proved that the problem of searching a sorted list is more difficult on the EREW PRAM than on the CREW PRAM. This result is unsatisfactory on two counts: (1) Separation is proved for a partial function, which may not say anything about their relative power for problems defined over complete domains: to give an example, later in the chapter, we will define the CROW PRAM (concurrent read, owner write) model. It is known that the problem of computing the OR of n bits, when at most one of the bits is 1, is more efficiently solvable on a CREW PRAM than on any CROW PRAM, but Nisan [Nis91] has proved that CROW and CREW

PRAMs take the same time (up to a constant) for problems defined over their full domains.

(2) The result uses Ramsey theory and relies crucially on the fact that the input integers come from an extremely large domain relative to the number of inputs (at least doubly exponential in the number of inputs). Essentially, the author shows that there is a large subset of the domain for which the state of computation at each point depends only on the relative ordering of the input values. The use of such enormous inputs is unsatisfactory. The unlimited word-size assumption of the PRAM model (just as in the case of unit cost RAMs) is a convenient abstraction to keep the model simple. We certainly do not want our results to exploit this feature. Such results may say more about the difficulty of handling very large numbers than about the inherent difficulty of solving the problem on reasonable size domains. The problem size n should be a fair measure of the input size. That is, we should be able to divide the input into n reasonable size pieces, so that each input integer is small. As an example, consider the problem of finding the largest element from a set of n integers on an n-processor CRCW PRAM. With no restrictions on the size of the inputs, Fich et al. [FMadHRW85] have shown a lower bound of $\Omega(\log\log n)$, but if the input integers are restricted to be at most $O(n^k)$ in value, an algorithm running in time $O(k)$ is known [FRW88].

Gafni, Naor, and Ragde [GNR89] extended Snir's result to a full domain. This takes care of our first objection but their result still uses Ramsey theory and requires a very large domain for input integers. The other extreme, when each of the n inputs is a single bit, is open.

Open problem: Is there a function $f : \{0,1\}^n \longrightarrow \{0,1\}$ that can be computed faster by a CREW PRAM than by an EREW PRAM.

Fich and Wigderson [FW90] have made some progress by resolving this question in a special case when there is a restriction imposed on where processors can write in shared memory. The EROW PRAM is an EREW PRAM in which each processor is said to "own" one shared memory cell and that is the only cell to which it is allowed to write. Processors are still allowed to read from any cell. The CROW PRAM [DR86] is the CREW PRAM restricted in the same manner. Fich and Wigderson proved that the EROW PRAM requires $\Omega(\sqrt{\log n})$ time to compute a

Boolean function that requires only $O(\log\log n)$ time on the CROW PRAM. The CROW PRAM never requires more than a constant factor more time than the CREW PRAM to compute any function defined on a complete domain (although the simulation may require a substantial increase in the number of processors) [Nis91]. However, the restriction to the owner write model with a single memory cell per processor seems more drastic for exclusive read machines. A fast simulation of EREW PRAMs by EROW PRAMs seems unlikely. In the case of CROW PRAMs, allowing processors to own more than one shared memory cell does not change the power of the model. But in the case of the EROW model, we do not even see any obvious way to extend the lower bound proof of Fich and Wigderson to allow each processor to own an arbitrary number of memory cells. There is another issue here: for an exclusive write machine, information can be communicated by the fact that no processor writes into a cell at a given time step. This is not allowed in the case of owner write machines. A close examination of the proofs in [CDR86, Kut91, DKR94] reveals that bounding the amount of information communicated in this way is usually the hardest part of the lower bound argument. For example, in [CDR86], the proof for the CROW PRAM model is presented as a warm up step and the bulk of the proof is devoted to proving a slightly weaker lower bound for the case of CREW PRAMs.

This leaves open the following question: Is there a separation between CREW and EREW PRAMs for any function $f : \{0,1\}^n \to \{0,1\}$ without the owner-write restriction? We cannot answer this question in general but we can when the amount of shared memory through which processors can communicate is small. Our main result in this chapter is:

Theorem 4.1 *For all $m \in o(\sqrt{\log n})$, there is a function on n Boolean variables that can be solved asymptotically faster on a CREW(1) PRAM than on any EREW(m) PRAM.*

The rest of this chapter is devoted to proving this theorem.

4.5 A separation result between CREW(m) and EREW(m) PRAMs

We show that a special case of the problem considered by Fich and Wigderson can be solved in time $O(\sqrt{\log n})$ by a CREW(1) PRAM, but requires $\Omega(\frac{\log n}{\log^* n})$ time on

any EREW(1) PRAM, where the $\log^*$ function is defined as follows: Let $\log^{(j)} n$ be the result of taking j successive logarithms of n, and define $\log^* n$ to be the smallest integer j such that $\log^{(j)} n \leq 2$.

It is easy to see that the sequential time complexity of this problem is $\log_2 n$, which is almost matched by our lower bound for the EREW(1) PRAM. Because an EREW(1) PRAM can simulate one step of an EREW(m) PRAM in m steps, an $\Omega(\frac{\log n}{m \log^* n})$ lower bound for EREW(m) PRAM is immediate. We can prove a somewhat stronger lower bound of $\Omega(\frac{\log n}{m+\log^* n})$ on any EREW(m) PRAM.

We would like to extend our separation between CREW and EREW PRAMs to larger communication width. Our hope is that some of the techniques developed for small communication widths will turn out to be useful even for the general case. For example, the technique in the lower bound result for the OR function on CREW(1) PRAMs [VW85, Bea86] is very similar to the technique that Kutyłowski [Kut91] eventually used in his optimal bound for the OR on general CREW PRAMs.

Our lower bound proof consists of three parts. First we show that any EREW(1) PRAM running for a short time can only have a small number of processors doing useful work. We then determine the time complexity of our problem on CREW(1) and CRCW(1) PRAMs with limited numbers of processors. This also implies an $\Omega(\log^{2/3} n)$ time bound for the EREW(1) PRAM. Finally, we show that there are subproblems (obtained via restrictions) on which the number of processors doing useful work is drastically reduced. Applying this result recursively, we obtain a nearly optimal EREW(m) PRAM lower bound for small m.

4.5.1 A Bound on the Number of Processors Doing Useful Work

For a function defined on n variables, we assume that a PRAM starts with its input stored in n read-only cells. In addition, there are m common cells. We do not place any restrictions on the number of processors, word size, or the computational power of individual processors.

For any EREW or CREW PRAM computing a function f, we say that a processor p *writes by time* t if, on some input to f, p writes into some memory cell during the first t steps. Since shared memory is the only means of communication, we can assume that for any PRAM running for t steps, only the processors that

write by time t are involved in the computation.

The bounds in [Bea86] (see also [VW85]) show that for any CREW(1) PRAM at time t and for any given input vector e, at least a $2^{-O(t^2)}$ fraction of all inputs vectors are indistinguishable from e, from the point of view of any individual processor. If a processor writes on e it also writes on all the other inputs that it can not distinguish from e, in this case at least $2^{-O(t^2)}$ fraction of all inputs. Because the machine only has exclusive writes, that is, in any time step at most one processor can write on any given input, it is easily seen that there are at most $2^{O(t^2)}$ processors that write by time t. This bound is tight – see, for example, the algorithm given in the proof of Theorem 4.5. For EREW(m) PRAMs, we now show a considerably smaller bound.

Lemma 4.2 *Consider any EREW(m) PRAM computing a function with domain* $\{0,1\}^n$. *Then for all* $t \geq 0$, *at most* $m(2^{t+1} + 2^t - 3) \leq m2^{t+2}$ *processors write by time* t.

Proof: For any processor P, time t, and input x, let $P^t(x)$ be the set of input variables that P reads during the first t steps on input x. Since a processor reads at most one cell per step, $|P^t(x)| \leq t$.

For $1 \leq j \leq t$, let $R(j)$ be the set of processors that do not read any common cell on any input for the first $j-1$ steps, but do read one of them on some input at step j.

Consider any processor $P \in R(j)$ and suppose that, on input $x \in \{0,1\}^n$, P reads some common cell at step j. At step j, processor P must decide whether to read that common cell based on the values of the variables in $P^{j-1}(x)$. The fraction of all inputs that agree with x on these variables is at least $2^{-(j-1)}$, since $|P^{j-1}(x)| \leq j-1$. On all these inputs, P reads the same common cell in step j. At most one processor can read that cell in step j on any particular input, so that there are at most 2^{j-1} processors in $R(j)$ which do so over all inputs. Since there are m such cells, $|R(j)| \leq m2^{j-1}$.

Similarly, if $W(j)$ is the set of processors that do not read any common cell on any input during the first j steps, but do write into one of them on some input at step j, then $|W(j)| \leq m2^j$. ($|W(j)| = |R(j+1)|$ because a processor reads at step j before writing in step j.) Notice that any processor that

writes by time t has either read the common cell during the first t steps or not. In the first case, it belongs to $\bigcup_{j=1}^{t} R(j)$, and in the second case, it belongs to $\bigcup_{j=1}^{t} W(j)$. Thus the number of processors that write by time t is bounded above by

$$\sum_{j=1}^{t} |R(j)| + \sum_{j=1}^{t} |W(j)| \leq \sum_{j=1}^{t} m2^{j-1} + \sum_{j=1}^{t} m2^{j} \leq m(2^t - 1 + 2^{t+1} - 2). \qquad \square$$

We will now construct an EREW(m) PRAM computing a function with domain $\{0,1\}^{m2^t}$ such that there are at least $m2^{t-2}$ processors that write by time t. Thus the bound in Lemma 4.2 is optimal to within a small constant factor. We begin by considering the case $m = 1$.

Lemma 4.3 *There is an EREW(1) PRAM computing a function with domain $\{0,1\}^{2^t}$ for which, over all inputs, there are at least 2^{t-2} processors that write during step t.*

Proof: Before beginning the construction, we examine the write operation of an EREW(1) PRAM. We view the selection of a processor to write during step t as a competition among processors, where the selection is arbitrated by the input vector. A processor is a 'potential winner' if there is some setting of the input bits that would cause it to write during step t. Let $b_1 = 1$, $b_j = \sum_{i=1}^{j-1}[b_i + 1] < 2^j$, $k_1 = 1$, and $k_j = \sum_{i=1}^{j-1} k_i = 2^{j-2}$, for $j \geq 2$. For any j, we construct an EREW(1) PRAM algorithm that has k_j potential winners. This algorithm runs for j steps, does not access the common cell, and is arbitrated by only b_j bits of input. Notice that this is enough to prove the lemma as we can modify the algorithm to make the winning processor write at step j.

The claim is proved by induction on j. The case $j = 1$ is trivial. For larger values of j, consider an input of length b_j which is partitioned into disjoint groups $X_1, X_2, \ldots, X_{j-1}$ of length $b_1, b_2, \ldots, b_{j-1}$, respectively, as well as one extra group of $j-1$ bits: $y_1, y_2, \ldots, y_{j-1}$. Let $G_1, \ldots, G_{j-1}$ be disjoint sets containing $k_1, \ldots, k_{j-1}$ processors, respectively, for a total of k_j. By our induction hypothesis, for each $i < j$, we can select a winner from among G_i during step i based on the input in X_i. The winner from G_i reads $y_1, y_2, \ldots, y_{j-i}$ in steps $i+1, i+2, \ldots, j$ respectively.

This processor is a winner in step j if and only if $y_1 = y_2 = \cdots = y_{j-i-1} = 0$ and $y_{j-i} = 1$. It is easy to verify that read conflicts never occur. □

Corollary 4.4 *There is an EREW(m) PRAM computing a function with domain $\{0,1\}^{m2^t}$ for which, over all inputs, there are at least $m2^{t-2}$ processors that write during step t.*

Proof: Run m separate copies of the algorithm in Lemma 4.3, one per common cell, on separate portions of the input. □

4.5.2 A CREW(1) Upper Bound for Evaluating Decision Trees

We now define a Boolean function and show that it can be computed in $O(\sqrt{\log n})$ time on a CREW(1) PRAM. In the next two sections, we will prove that this function requires significantly more time to be computed on an EREW(1) PRAM. Specifically, we interpret the $n = 2^h - 1$ input variables as the labels of the internal nodes in a complete Boolean decision tree D_h of height h, taken in some fixed order. (For example, we could use breadth first order, that is, the root is labeled x_1 and the left and right children of the node labeled x_i are labeled x_{2i} and x_{2i+1} respectively.) The leaves of D_h that are left children are labeled 0; those that are right children are labeled 1. Given an input, proceed down from the root, going left when a node labeled by a variable with value 0 is encountered and going right when a node labeled by a variable with value 1 is encountered. The value of the function $F_h : \{0,1\}^{2^h-1} \rightarrow \{0,1\}$ is the label of the leaf node that is reached.

There is a trivial sequential algorithm that computes F_h in h steps. It is unknown whether one can do better than this on an EREW(1) PRAM (see Theorem 4.14). However, the following lemma shows that F_h can be computed substantially faster on a CREW(1) PRAM.

Theorem 4.5 *If $\binom{t}{2} \geq h$, then there is a CREW(1) PRAM that computes F_h in t steps.*

Proof: For each of the 2^h root-leaf paths in the decision tree D_h, we assign a group of $t-1$ processors. Exactly one of these root-leaf paths is the correct path.

In the following algorithm, the common cell will contain values of the variables labeling the first $\binom{j+1}{2}$ nodes on the correct root-leaf path at the end of step $j+1$.

The jth processor in each group is active for the first $j+1$ steps. During the first j steps, it reads the j variables labeling nodes $\binom{j}{2}+1$ through $\binom{j+1}{2}$ on its root-leaf path. At step $j+1$, it reads the common cell, which contains the values of the variables labeling the first $\binom{j}{2}$ nodes on the correct root-leaf path. (When $j=1$, the common cell contains no information.) At this point, the jth processor in each group knows whether or not its root-leaf path agrees with the correct root-leaf path at the first $\binom{j+1}{2}$ nodes. Among the processors whose paths agree on these first nodes, a prespecified one (for example, the jth processor in the leftmost of these groups) appends the bits that it has read to the previous contents of the common cell. Thus, at the end of step $j+1$, the common cell contains values of the labels of the first $\binom{j+1}{2}$ nodes along the correct root-leaf path.

To compute F_h, we modify the algorithm slightly so that at the last step, instead of appending bits to the common cell, a processor writes the value of the leaf determined by the h internal nodes on its path, i.e. the leaf node in D_h that is reached. □

4.5.3 Lower Bounds for Processor-limited PRAMs

In this section, we show that to compute F_h quickly we need to have many processors doing useful work even on a CRCW(1) PRAM. Using our bounds from Lemma 4.2 on the number of processors doing useful work, this will give an $\Omega(h^{2/3})$ lower bound for the EREW(1) PRAM. In the next section, we will improve this bound to a nearly optimal $\Omega(\frac{h}{\log^* h})$ by combining the techniques of this section with a new *restriction technique.*

A *restriction* is a partial function that sets the values of some input variables. For any restriction r that sets some input variables to 0 or 1, let $r(F_h)$ be the function F_h with restriction r applied to it. Define the *depth* of r, $d(r)$, to be the minimum depth of any node v in D_h, the underlying decision tree of F_h, such that the path from root to v is consistent with r and the subtree rooted at v does not contain any variables set by r. Note that in order to compute $r(F_h)$, we must compute $F_{h-d(r)}$.

Define the *history* of common cells on any input to be the sequence of vectors of values which they take on that input with one vector per step of the PRAM. Our lower bound proofs proceed by fixing the history of the common cells. We use the following result of Vishkin and Wigderson [VW85].

Lemma 4.6 *[VW85] For any CRCW(m) PRAM computing any function F and running for t steps, there is a restriction r of F which sets at most* $m\binom{t+1}{2}$ *variables such that the history of the common cells for the first t steps is the same for all inputs consistent with r.*

Proof sketch: We will sketch the essential ideas of the proof for the case $m = 1$. We will prove by induction on t that there is a restriction r_t such that the history of the (unique) common cell for the first t steps is the same for all inputs consistent with r_t.

r_0 is the empty restriction, that is, r_0 does not set any variables. For $t \geq 1$, we will show a way to obtain r_t from r_{t-1} by setting at most $t-1$ *additional* variables, which will be enough to prove the claim. From now on, consider only the inputs consistent with r_{t-1}. We consider two different cases.

Case 1: If in step t, no processor writes (in the common cell) on any input then $r_t = r_{t-1}$. Clearly in this case, if the history of the common cell is the same up to step $t-1$ then it is also the same up to step t.

Case 2: If in step t, there is at least one processor P and at least one input e such that P writes on e then we will extend r_{t-1} to obtain r_t such that P can not distinguish between e and any other input consistent with r_t. Then P is going to write the same value on all inputs consistent with r_t, and therefore if the history of the common cell is the same up to step $t-1$ then it will also be the same up to step t.

There are two possible avenues for P to gain information about input e: by reading the common cell and by reading the input variables in read-only cells. Because the history of the common cell has been the same for all inputs, P can not distinguish between e and any other input, based on what it has learned by reading the common cell. P reads at most $t-1$ input variables on e in the first $t-1$ steps, so we obtain r_t by setting those $t-1$ variables to their values in e.

□

Lemma 4.7 *For any CRCW(m) PRAM computing F_h for $h \geq 0$ and running for t steps, there is a restriction of depth at most $m\binom{t+1}{2}$ such that the history of the common cells for the first t steps is the same for all inputs to F_h consistent with this restriction.*

Proof: If $h \leq m\binom{t+1}{2}$, we can apply a restriction of height h so that there is exactly one input consistent with this restriction, and the lemma is trivially true. So assume that $h \geq m\binom{t+1}{2}$. By Lemma 4.6, there is a restriction r which sets at most $m\binom{t+1}{2}$ variables such that the history of the common cells for the first t steps is the same for all inputs consistent with r.

We define a restriction r' that is consistent with r by tracing a path of length at most $m\binom{t+1}{2}$ from the root of F_h one node at a time, as follows. If r sets the variable at the current node then let r' sets this variable to be consistent with r and takes the branch corresponding to this value. Otherwise, consider the number of variables that are set by r in each subtree and take the branch which leads to the subtree with the smaller number of such variables. Each time we extend the path by one node there is at least one less variable set by r in the subtree reached by that path. So, by the time the path reaches length $m\binom{t+1}{2}$, we will have reached the root of a subtree with none of its variables set. We set all variables outside this subtree in some manner consistent with r. Clearly, the resulting restriction r' has depth at most $m\binom{t+1}{2}$ and for all inputs consistent with r', the common cells have the same history for the first t steps. □

In particular, this implies that the CREW(1) PRAM algorithm to compute F_h given in the proof of Theorem 4.5 is within one step of optimal:

Theorem 4.8 *If a CRCW(1) PRAM computes F_h in t steps, then $\binom{t+1}{2} \geq h$.*

Proof: Consider any CRCW(1) PRAM that computes F_h in t steps. By Lemma 4.7, there is a restriction r of depth at most $\binom{t+1}{2}$ such that the history of the common cell for the entire computation is the same for all inputs to F_h consistent with r. In particular, the answer produced by the computation is the same for all

these inputs. Thus $r(F_h)$ must be a constant function, which in turn implies that $\binom{t+1}{2} \geq h$. □

The following theorem shows that, with a limited number of processors, a larger lower bound may be obtained.

Theorem 4.9 *Any CRCW(1) PRAM with p processors that computes F_h requires at least $\frac{2h}{3\lceil 1+\sqrt{\log p}\rceil}$ steps.*

Proof: If $p = 1$, an easy adversary argument shows that computing F_h has complexity at least $h \geq \frac{2h}{3\lceil 1+\sqrt{\log 1}\rceil}$. Therefore, assume $p \geq 2$. The proof proceeds by induction on h. The base case when $h = 0$ is trivial. We will prove it for higher values of h. Let $t = \lceil \sqrt{\log p} \rceil \geq 1$.

Suppose there is a CRCW(1) PRAM with p processors that computes F_h in T steps. Then, from Theorem 4.8, $\binom{T+1}{2} \geq h$. If $\binom{t+1}{2} \geq h/3$, then $T(t+1)/2 \geq h/3$. (This is most easily seen by a case analysis: if $t \geq T$, then $\binom{T+1}{2} \geq h$ implies $T(t+1)/2 \geq h$; on the other hand, if $t < T$, then $\binom{t+1}{2} \geq h/3$ implies $T(t+1)/2 \geq h/3$.) This implies that $T \geq 2h/3(t+1)$ as required. Therefore, assume that $\binom{t+1}{2} < h/3$.

By Lemma 4.7, there is a restriction r of depth at most $\binom{t+1}{2}$ that fixes the history of the common cell for the first t steps. Consider the computations of the CRCW(1) PRAM on all inputs consistent with r. Since each input variable has at most two different values and the value of the common cell is fixed at each time step, it follows that each processor is in one of at most 2^i states at the end of step $i < t$. Now the state of a processor determines which memory cell it will read next. Thus at most $p\sum_{i=0}^{t-1} 2^i < 2^{t^2+t}$ different input variables are read during the first t steps by all processors on all these inputs.

Let v be any node of depth $d(r)$ such that the path from the root to v is consistent with r and the subtree rooted at v does not contain any variables set by r. Consider the 2^{t^2+t} nodes at distance $t(t+1)$ from v. There is at least one node w such that the subtree rooted at w contains input variables that no processor can possibly read in the first t steps. Let r' be a restriction that extends r by setting the variables labeling all ancestors of w so as to cause the path from the root of

D_h to w to be followed. Only the variables labeling nodes in the subtree rooted at w are left unset. All remaining variables are set arbitrarily. The restriction r' has depth at most $\binom{t+1}{2}+t(t+1) = 3\binom{t+1}{2} < h$. By construction, the functions $F_{h-d(r')}$ and $r'(F_h)$ are identical, up to the renaming of variables. Since no processors have read any input variables of this subfunction at time t, it follows from the induction hypothesis that at least $\left[2h - 6\binom{t+1}{2}\right]/3(t+1)$ additional steps are required to compute this subfunction. Therefore $T \geq t + \left[2h - 6\binom{t+1}{2}\right]/3(t+1) = \frac{2h}{3(t+1)}$. □

We note that this lower bound is asymptotically optimal even for a CREW(1) PRAM:

Corollary 4.10 *For all integers $1 \leq p \leq 2^h$, the complexity of F_h on a CRCW(1) or CREW(1) PRAM with p processors is $\Theta(h/\sqrt{\log p})$.*

Proof: The lower bound follows from Theorem 4.9. For the upper bound, notice that the algorithm in Theorem 4.5 shows that, with p processors, a CREW(1) PRAM can evaluate a decision tree of height $\Theta(\log p)$ in time $O(\sqrt{\log p})$. To compute F_h with p processors on a CREW(1) PRAM, we simply apply this algorithm sequentially $O(h/\log p)$ times to obtain a running time of $O(h/\sqrt{\log p})$. □

Using the bounds of Lemma 4.2 we have:

Corollary 4.11 *Any EREW(1) PRAM computing F_h must run for at least $\frac{1}{3}h^{\frac{2}{3}}$ steps.*

Proof: Suppose the EREW(1) PRAM runs for T steps. Then, by Lemma 4.2, we can assume that it has at most $p = 2^{T+2}$ processors. Now, applying Theorem 4.9, it follows that $T \geq \frac{2h}{3\lceil 1+\sqrt{T+2}\rceil}$. Since $\lceil 1 + \sqrt{T+2}\rceil \leq 3\sqrt{T}$ for all integers $T > 0$, we obtain $T \geq \frac{1}{3}h^{2/3}$, as required. □

4.5.4 A Near Optimal EREW(m) Lower Bound for Small m

We strengthen the arguments of the previous section to prove a nearly optimal $\Omega(\frac{h}{\log^* h})$ lower bound on the time for an EREW(1) PRAM to compute F_h and, more generally, to obtain an $\Omega(h/(m + \log^* h))$ lower bound for an EREW(m)

PRAM computing F_h. The key to this improvement is a new lemma that replaces Lemma 4.2 in the argument which shows that we can select a large subset of inputs on which very few processors ever do useful work. We use this to obtain a stronger version of Lemma 4.7 for the EREW(m) PRAM. This involves recursively applying the argument of the previous section to obtain a better lower bound.

Lemma 4.12 *For integers T and $h \geq 2T + 2 + \lceil \log m \rceil$, and any EREW($m$) PRAM computing F_h, there is a restriction r of depth $2T + 2 + \lceil \log m \rceil$ such that at most $2mT$ processors write by time T on inputs consistent with r.*

Proof: From Lemma 4.2, we can assume that the EREW(m) PRAM has at most $m2^{T+2}$ processors. For each processor P, let $s(P)$ denote the set of input variables that P reads before it reads any common cell during the first T steps of computation on any input to F_h. Define $S = \bigcup_P s(P)$. As in the proof of Theorem 4.9, since each input variable has at most two different values, $|s(P)| < 2^T$ and thus $|S| < m2^{2T+2}$.

Consider $m2^{2T+2}$ of the nodes at depth $2T + 2 + \lceil \log m \rceil$ from the root of D_h. For at least one such node v, none of the nodes in the subtree rooted at v is labeled by variables in S. Set the variables labeling ancestors of v so that the path from the root to v is followed in D_h. All variables labeling nodes in the subtree rooted at v are left unset. Set all other variables outside this subtree arbitrarily. Let r be the resulting restriction.

From now on, consider only the inputs consistent with r. The subtree rooted at v does not contain any input variables from S, so no processor reads any unset variable until after it has read one of the common cells. Since the PRAM is exclusive read, for each step $j \leq T$ and for each common cell, there is at most one processor that does not read any common cell on any input for the first $j - 1$ steps but does read that cell on some input at step j. Hence, at most mT processors read some common cell in the first T steps. Similarly, at most mT processors from among those that have not read any common cell may write in the first T steps. So, altogether, there are at most $2mT$ processors that can write into some common cell on inputs consistent with r. □

In order to describe the behavior of our recursive construction it is convenient to introduce a simple notation for the bounds that we obtain. Let $A(m,2) = 3m$ for $m \geq 1$ and

$$A(m,T) = (2T+2+\lceil \log m \rceil) + \left\lceil \frac{T}{\lceil \log T \rceil} \right\rceil (A(m,\lceil \log T \rceil) + 2\lceil \log T \rceil + 1 + \lceil \log m \rceil),$$

for $m \geq 1$ and $T \geq 3$.

We will sketch a proof that $A(m,T) \in O(Tm + T\log^* T)$.

$$\begin{aligned} & A(m,T) \\ < \;& (2T+3+\log m) + \left(1 + \frac{T}{\lceil \log T \rceil}\right)[A(m,\lceil \log T \rceil) + 2\lceil \log T \rceil + \log m + 2] \\ = \;& \left(1 + \frac{T}{\lceil \log T \rceil}\right) A(m,\lceil \log T \rceil) + \left(4T + \frac{2T}{\lceil \log T \rceil} + 2\lceil \log T \rceil + 5\right) + \log m \left(\frac{T}{\lceil \log T \rceil} + 2\right) \\ < \;& \left(1 + \frac{T}{\lceil \log T \rceil}\right) A(m,\lceil \log T \rceil) + 8T\left(1 + \frac{\log m}{\lceil \log T \rceil}\right). \end{aligned}$$

(Because $T \geq 3$)

Define $\lceil \log \rceil^{(j)} T$ as follows:

$$\begin{aligned} \lceil \log \rceil^{(j)} T &= T, \quad \text{if } j = 0 \\ &= \left\lceil \log \left(\lceil \log \rceil^{(j-1)} T \right) \right\rceil, \quad \text{if } j > 0. \end{aligned}$$

Define $F_{-1} = 1$, and

$$F_i = \prod_{0 \leq j \leq i} \left(1 + \frac{\lceil \log \rceil^{(j)} T}{\lceil \log \rceil^{(j+1)} T}\right), \quad G_i = \lceil \log \rceil^{(i)} T \left(1 + \frac{\log m}{\lceil \log \rceil^{(i+1)} T}\right), \quad \text{for } i \geq 0.$$

Expanding the recursive expression for $A(m,T)$, we get

$$A(m,T) < F_i \cdot A\left(m, \lceil \log \rceil^{(i+1)} T\right) + 8 \sum_{0 \leq j \leq i} F_{j-1} G_j, \qquad \text{for } i \geq 0.$$

Because $F_i \in O\left(\frac{T}{\lceil\log\rceil^{(i+1)}T}\right)$, $F_{j-1}G_j \in O\left(T\left(1+\frac{\log m}{\lceil\log\rceil^{(j+1)}T}\right)\right)$, and $\sum_{0\le j\le i} F_{j-1}G_j \in O\left(iT+\frac{T\log m}{\lceil\log\rceil^{(i+1)}T}\right)$,

$$A(m,T) = O\left[\frac{T}{\lceil\log\rceil^{(i+1)}T}\cdot A\left(m,\lceil\log\rceil^{(i+1)}T\right)+iT+\frac{T\log m}{\lceil\log\rceil^{(i+1)}T}\right], \quad \text{for } i\ge 0.$$

Choose $i = \log^* T - 1$. Then from the definition of the $\log^*$ function,

T is greater than a tower of 2's of height $i+1$, and T is less than or equal to a tower of 2's of height $i+2$.

It is easily seen by induction that for any $0 \le j \le i+1$, $\lceil\log\rceil^{(j)}T$ is greater than a tower of 2's of height $i+1-j$ and is less than or equal to a tower of 2's of height $i+2-j$. So, $1 < \lceil\log\rceil^{(i+1)}T \le 2$, and $A(m,T) \in O(Tm + T\log^* T + T\log m) \subseteq O(Tm + T\log^* T)$.

Lemma 4.13 *For any integer $T \ge 2$, and $h \ge A(m,T)$, and any EREW(m) PRAM computing F_h, there is a restriction r of depth at most $A(m,T)$ such that the history of the common cells for the first T steps is the same for all inputs to F_h consistent with r.*

Proof: The proof is by induction on the value of T.

For $T = 2$, $A(m,T) = 3m = m\binom{T+1}{2}$ and the claim follows from Lemma 4.7.

For larger values of T, first use Lemma 4.12 to obtain a restriction r_0 of depth $2T + 2 + \lceil\log m\rceil$ such that at most $2mT$ processors write by time T on inputs consistent with r_0. Let $\ell = \lceil\log T\rceil$. Break the T steps of the computation into $\lceil T/\ell\rceil$ subintervals each of length at most ℓ. It is sufficient to prove the following claim:

Claim: For all $i \le \lceil\frac{T}{\ell}\rceil$, there is an extension r_i of r_0 of depth at most $2T+2+\lceil\log m\rceil + i(A(m,\ell)+2\ell+1+\lceil\log m\rceil)$ so that restricted to inputs consistent with r_i and during the first i subintervals,

1. the history of common cells is the same, and

2. none of these $2mT$ processors read any variables left unset by r_i.

This is proved by induction on i. The case $i = 0$ is trivial, so suppose $i \geq 1$. Assume that a restriction r_{i-1} with the desired properties exists. We will first extend r_{i-1} to a restriction r'_i satisfying (1), and then extend r'_i to r_i satisfying (2) as well.

Without loss of generality, we may assume that r_{i-1} sets all variables labeling nodes outside of some tree of height $h - \text{DEPTH}(r_{i-1})$. Then $r_{i-1}(F_h)$ is the same as $F_{h-\text{DEPTH}(r_{i-1})}$ up to the renaming of variables. Since $\ell < T$, it follows from the induction hypothesis that there is a restriction of depth $A(m, \ell)$ such that the history of the common cells for the first ℓ steps is the same for all inputs to $F_{h-\text{DEPTH}(r_{i-1})}$ consistent with this restriction. None of the $2mT$ processors have read any variables left unset by r_{i-1} during the first $i-1$ subintervals on inputs to F_h consistent with r_{i-1}. Therefore there is also a restriction r'_i that extends r_{i-1} with depth $d(r_{i-1}) + A(m, \ell)$, such that the history of common cells for the first i subintervals is the same for all inputs to F_h consistent with r'_i.

As in the proof of Theorem 4.9, each of the $2mT$ processors can read at most $2^\ell - 1$ input variables during the ith subinterval, over all inputs. Thus there is an extension r_i of r'_i with depth $d(r'_i) + \lceil \log(2mT) \rceil + \ell \leq 2T + 2 + \lceil \log(m) \rceil + i(A(m, \ell) + 2\ell + 1 + \lceil \log m \rceil)$ so that none of the processors have read any inputs of F_h left unset by r_i during the first i subintervals. This is what was required. □

We can use this lemma to derive the near optimal lower bound for the EREW(m) PRAM for small m.

Theorem 4.14 *Any EREW(m) PRAM computing F_h must run for $\Omega(\frac{h}{m+\log^* h})$ steps.*

Proof: Suppose there is an EREW(m) PRAM computing F_h that runs for T steps, where $A(m, T) \leq h - 1$. By Lemma 4.13, there is a restriction r of depth $A(m, T)$ such that the answer in the common memory cells is the same for all inputs to $r(F_h)$. However, there are two inputs in $r(F_h)$ that reach the same node of depth $h - 1$, but should have different answers. Thus $A(m, T) \geq h$. Since $A(m, T) \in O(Tm + T \log^* T)$, it follows that $T \in \Omega(\frac{h}{m+\log^* h})$. □

Theorem 4.14 and Theorem 4.5 immediately give our main theorem.

Theorem 4.15 *There is a function on n Boolean variables that can be solved in $O(\sqrt{\log n})$ time on a CREW(1) PRAM but requires $\Omega(\frac{\log n}{m+\log^* n})$ time on every EREW(m) PRAM.*

Our main separation theorem is a straightforward corollary of the above theorem.

Theorem 4.1 For all $m \in o(\sqrt{\log n})$, there is a function on n Boolean variables that can be solved asymptotically faster on a CREW(1) PRAM than on any EREW(m) PRAM.

4.6 Conclusion

We would like to extend this separation result between CREW and EREW models to the case of larger communication width. We define *general* PRAMs as PRAMs with unlimited communication width. The separation result of [CDR86, Kut91, DKR94] applies to general CRCW and CREW PRAMs and it would be nice to prove a similar result about CREW and EREW PRAMs. We hope that our lower bound argument gives some insight even for the general case. There is at least some precedence for this: Kutyłowski's [Kut91] optimal lower bound for general CREW PRAMs, computing the OR function, uses techniques similar to the lower bound result, for the OR function, on CREW(1) PRAMs [VW85, Bea86].

Evaluating a Boolean decision tree, the function for which we showed a separation between CREW(1) and EREW(1), has the same complexity (up to a constant additive term) on general CREW and EREW PRAMs, so it can not be used to separate general CREW and EREW PRAMs:

Proposition 4.16 *The complexity of F_h on CREW or EREW PRAMs is* $\log h + \Theta(1)$.

Proof sketch: The lower bound on CREW PRAMs follows from a degree argument [DKR94]; the upper bound follows from the following algorithm: for each node of D_h, assign a processor and a common cell. In the first step, every processor reads the value of the corresponding input variable. If this input variable is zero, the processor writes the index of the left child in the corresponding common cell;

otherwise, the processor writes the index of the right child in the corresponding common cell. Now interpreting the contents of common cells as values of pointers in a list, the path that is taken in the decision tree is precisely the path obtained by following these pointers starting from the root node. We can use a standard EREW PRAM pointer doubling algorithm to evaluate F_h in $\log h + O(1)$ steps. □

However a generalization of this function, considered in Fich and Wigderson [FW90], seems like a good candidate function.

Definition 4.17 *[FW90] Given any $m(2^h - 1) + 2^m$ bits of input, they can be interpreted as a Boolean decision tree of height $h-1$, where each node is labeled with one of $\{x_1, x_2, \ldots, x_m\}$. The first $m(2^h-1)$ bits of the input are interpreted as giving labels of each of the 2^h-1 nodes in the decision tree; and the last 2^m bits of the input are interpreted as giving values of $x_1, x_2, \ldots, x_m$. $F_{m,h} : \{0,1\}^{m(2^h-1)+2^m} \rightarrow \{0,1\}$ is defined as the value of this decision tree.*

Proposition 4.18 *[FW90] The complexity of $F_{m,h}$ on CREW PRAMs is $\Theta(\log m + \log h)$, and it can be solved on an EROW PRAM in $O(2^m + \log h)$ steps.*

Open problem: Determine the exact complexity of $F_{m,h}$ on general EREW PRAMs.

Chapter 5

MULTIPREFIX PRAMS

5.1 Introduction

In the study of parallel computation, the PRAM has been a very useful model. The model is attractive because it abstracts away most of the messy details of implementing algorithms on parallel machines. The flip side of this is that the model has no efficient implementation. The read and write primitives are typically implemented by routing packets on a fixed connection network of processors. Global memory is distributed among these processors and any processor, wanting to access a memory location, sends a packet to the node containing the desired memory location. If the access is "read" then a packet with the contents of the memory location is returned to the originating processor; whereas if the request is "write", the contents of the designated cell are updated. If many processors try to access the block of memory located at the same node, memory contention can really slow the system down. The usual solution in this case is to combine messages destined for the same memory location in a single read or write cycle. Such a simulation of a PRIORITY CRCW PRAM by an FFT Network was shown by Ranade [Ran87] and this was extended to a variety of other networks by Leighton, Maggs, and Rao [LMR88] among others. Currently there is no practical solution for implementing an n-processor read or write that does better than $O(\log^2 n)$ for deterministic schemes or $O(\log n)$ for probabilistic schemes. This is of some concern because the PRAM model assumes reads and writes to be primitive operations which take unit cost.

But we realize that the assumption of unit cost for primitives is never strictly true on any model. The usefulness of algorithmic models is in supplying a suitable abstraction of real machines that helps in algorithms design. The real concern is the following: any algorithm can be thought of as giving a two step simulation of the given problem in hardware: first, the problem is solved in terms of primitives of the machine; second, each of those primitives is simulated in hardware. The

danger of having very powerful primitives is that for certain problems, this two step simulation process may be lot more expensive than a direct simulation in hardware. For example, Beame and Håstad [BH89] have shown that the parity function, which can be solved on realistic machines in essentially the same time as a multiprocessor read or write, requires $\Omega(\log n/\log\log n)$ time on a PRIORITY CRCW PRAM. In other words, computing parity on PRIORITY CRCW PRAM will take time $\Omega(\log n/\log\log n)$ times the time to implement reads or writes, which is a slow-down of $\Omega(\log n/\log\log n)$.

As we argued in Chapter 1, the solution is to augment the model with a set of primitives that can be implemented in time comparable to implementing reads and writes, in the hope that a richer instruction set will help algorithm designers.

Some practical and theoretical works for parallel machines [Ble89, Ble90, CBZ90, KRS86, RBJ88, PS88, KRS88] have suggested that *multiprefix operations* for certain multiary operators be allowed at unit cost. We will call all such models *multiprefix PRAMs*. Later we will give a precise definition of multiprefix operations and multiprefix PRAMs. Informally, the effect of computing a multiprefix operation $\odot$ on arguments $x_1, x_2, \ldots, x_k$ is to compute the k prefixes $\odot(x_1, x_2, \ldots, x_i)$ for $1 \leq i \leq k$. In every round of the computation of a multiprefix PRAM, processors partition themselves into groups. Within certain groups, all processors belonging to the group perform a multiprefix operation on their private data. As a result of this computation, each processor receives the prefix corresponding to its position in the order, sorted according to the processor indices. There are several models which differ in the kinds of partitioning (of processors) they allow.

Multiprefix operations have a basis in many existing parallel machines and proposed architectures. In the implementation of the CRCW PRAM on the NYU Ultracomputer, a number of additional operations such as *Fetch-and-add* were included [GGKR83]. These operations used the combining network to perform computation during a concurrent memory access and were used as synchronization primitives. Blelloch [Ble89] considered *scan* operations, where all processors are restricted to be in the same group. He also considered *segmented scans* where every group consists of processors with consecutive processor indices, and all processors perform multiprefix operation based on the same multiary operator. Blelloch [Ble89] showed that scan primitives for integer-add, integer-max, integer-min,

OR, and AND can be implemented as efficiently as reads and writes on a connection machine [Hil85]. He gives many examples where these primitives reduce the running time, sometimes by as much as a factor of $\Theta(\log n)$. In many cases, this also simplifies the program. Chatterjee et al. [CBZ90] discuss implementation of scan operations on a CRAY Y-MP [Cra88]. Again the scan primitives turn out to be very powerful, even for the normally hard case of manipulating irregular and dynamically changing structures. The appendix in Blelloch's Ph.D. thesis [Ble90] gives a brief history of the scan operation.

Ranade et al. [RBJ88] consider a generalization of the scan operation, where arbitrary partitioning of processors is allowed. The authors give strong justification for an architecture providing these primitives. Because we will be giving a simulation of multiprefix PRAMs, we adopt this (strongest) definition of multiprefix operations.

The fetch-and-op of Gottlieb et al. [GLR83] is an indeterminate version of the mutiprefix operation. The effect of fetch-and-op is also to compute a set of prefixes, but the order of inputs is undetermined. In this regard, the multiprefix operation can be considered a particular implementation of fetch-and-op.

We begin by defining a *multiprefix* (MP) operation. We number the processors of the (multiprefix) PRAM as $P_1, P_2, \ldots$

Definition 5.1 *A multiprefix operation MP$(L, v, \odot)$ takes three arguments: L is the address of a memory location; v is the private data of the processor performing this operation; and $\odot$ is a multiary operator. Let $S = \{P_{i_j} : 1 \leq j \leq k\}$ be any set of k processors such that $i_j \leq i_{j+1}$ for $1 \leq j < k$. Suppose that the memory location L contains the value v_0, each processor P_{i_j} in S performs the operation MP$(L, v_j, \odot)$, and no processor outside S performs a multiprefix operation referring to memory location L. Then, as a result of these multiprefix operations, each processor P_{i_j} will receive $\odot(v_0, v_1, \ldots, v_{j-1})$, and L will contain $\odot(v_0, v_1, \ldots, v_k)$. A valid algorithm makes sure that in any round of the computation, all processors performing a multiprefix operation referring to a particular memory location have the same multiary operator.*

To give an example, if the memory location L contains value 0, and for $1 \leq i \leq n$,

the i^{th} processor P_i performs MP$(L, i, +)$, then as a result of this operation P_i will receive $1 + 2 + \cdots + i - 1$ and L will contain $1 + 2 + \cdots + n$.

The cost of implementing multiprefix computations as part of the PRAM simulations by a network of processors is that each processor or switch in that network becomes a little more complicated for each new multiprefix operator added. We parameterize the multiprefix PRAM by a set $\mathcal{O}$ of allowable multiprefix operators.

Definition 5.2 *Let $\mathcal{O}$ be any set of operations. Then an $\mathcal{O}$-multiprefix PRAM is a priority CRCW PRAM with extra multiprefix primitives from the set $\mathcal{O}$. For computing a function of n inputs, we start with input values stored in n designated memory locations. The computation proceeds in steps. Each step consists of a "write" phase, followed by a "multiprefix" phase, followed by a "read" phase. In the write (multiprefix, read) phase, each processor can perform at most one write (respectively, multiprefix, read) operation. The output is the contents of a specially marked output cell at the end of the computation.*

The two most interesting complexity measures to us are *time* and *number of processors*.

It is easy to show that arbitrary multiprefix operations can lead to unreasonable models. For example, if concatenation of bit strings is allowed as a multiprefix primitive then, as shown in the algorithm below, the entire input can be collected into a single location in a single step.

Algorithm:
Comment $x_1, x_2, \ldots, x_n$ is the input
Step 1 For $2 \leq j \leq n$, processor P_j reads x_j.
Step 2 Comment: Let L be the memory location holding x_1.
For $2 \leq j \leq n$, processor P_j executes MP$(L, x_j,$"concatenate"$)$.
Comment: At this point L contains concatenation of $x_1, x_2, \ldots, x_n$.

There are two objections to this: first, if, as is traditional in lower bound study of PRAMs, we do not place any restrictions on the computational power of individual processors, a processor can read all input bits in one more step and compute

the given function. The more serious objection is that such operations require the transmission of n-bit values in the network. A natural limitation then is on *bandwidth* of the operation, defined as the number of bits of information of each input and output value. It is reasonable to set this to be equal to the word-length of the machine. Our main result shows that if bandwidth of the multiprefix operation is so limited (and the domain of the operation has an identity element) then the PRAM itself may be very efficiently simulated by an unbounded fan-in circuit with special gates for the operations in $\mathcal{O}$. Notice that other than a word-length restriction we are not making any assumption which otherwise limits the power of processors in the PRAM.

In order to route messages in a network of p processors, addresses must be transmitted so that a word-length of at least $\log p$ is most desirable. Furthermore, most specific operations that have been proposed can be implemented using $\log p$-bit values. If the word-length is $\Theta(\log p)$, our results imply that anything that can be computed by a multiprefix PRAM in time T using p processors can be computed by an unbounded fan-in circuit of depth $O(T)$ and size polynomial in p^T, having gates for the operations in $\mathcal{O}$.

Bellantoni [Bel91] showed a simulation of restricted word-sized PRIORITY CRCW PRAMs by unbounded fan-in circuits. We extend his result and the techniques are very similar although there are some significant differences in the details required to handle multiprefix operations. We should note that our simulation is not limited to multiprefix operations based on associative binary operations and thus we can handle a wider variety of functions, for example, threshold functions.

5.2 Definitions

To keep our simulation result strong, we do not place any restrictions on the computational power of individual processors. However, as we argued earlier, we only allow multiprefix operations with bounded "bandwidth."

Definition 5.3 *For any integer $\mu > 0$, VALID(μ) is the set of operations $\odot$ such that*

1. *For all $k \geq 0$, if each of $x_1, x_2, \ldots, x_k$ is less than 2^μ then $\odot(x_1, x_2, \ldots, x_k)$ is also less than 2^μ. Notice that "addition" does not satisfy this, but many simple variations on addition, for example, addition modulo 2^μ satisfy this property.*

2. *$\odot$ has a* unit. *That is, there exists $1_\odot$ such that adding arguments with value $1_\odot$ does not change the operation's value. Formally, for all $x_1, \ldots, x_k$, if there exists $1 \leq i_1 < i_2 \cdots < i_m \leq k$ such that $i \notin \{i_1, \ldots, i_m\}$ implies $x_i = 1_\odot$ then $\odot(x_1, \ldots, x_k) = \odot(x_{i_1}, \ldots, x_{i_m})$.*

Notice that VALID(μ) includes a wide variety of functions including all *threshold functions*. We call μ the *word-length* of the machine and make sure that the contents of any memory location can be represented by at most μ bits.

Definition 5.4 *For any $p, \mu > 0$, and $\mathcal{O} \subseteq$ VALID(μ), an MP-PRAM($\mathcal{O}, p, \mu$) is an $\mathcal{O}$-multiprefix PRAM with p processors such that any value that any processor attempts to write is less than 2^μ.*

Next, we define the family of circuits that will be used to simulate these PRAMs.

Definition 5.5 *For any $\mu > 0$, and $\mathcal{O} \subseteq$ VALID(μ), MP-Circuit($\mathcal{O}, \mu$) is the set of unbounded fan-in circuits, with gates computing AND, OR, NOT, and functions in $\mathcal{O}$. We assume that each gate computing a function in $\mathcal{O}$ receives its set of inputs encoded in binary, and has its output also encoded in binary. For small values of μ, this assumption is hardly restrictive since we can always compute any function of these inputs or outputs by additional circuitry of size $O(\mu 2^\mu)$.*

5.3 Simulation

There are several motivations for designing efficient simulations between different models. If seemingly different models have efficient simulations on each other then it is an evidence of the robustness of those models. That is, results proved on such models are indeed saying something about the complexity of the problem, rather than quirks of the model. Because we have had very limited success in proving lower bounds, simulation results are also attractive because they give an easy way

of translating known lower bound results on the model that is simulating to the model being simulated.

Between circuits and PRAMS, it is well known that an unbounded fan-in circuit with s gates, e edges, and depth d can be simulated by a common CRCW PRAM with e processors and s shared memory cells in time $d + 1$. The simulation is straightforward: the PRAM has a cell for each gate of the circuit and a processor for each edge and values are propagated from the input of the circuit towards its output node.

Stockmeyer and Vishkin [SV84] showed a simulation of CRCW PRAMs by unbounded fan-in circuits. Their result is somewhat unsatisfactory because, for their simulation, they have to severely restrict the instruction set of processors in CRCW PRAMs, disallowing even natural operations like multiplication.

Theorem 5.6 *(Stockmeyer and Vishkin [SV84]) A priority PRAM with p processors running in time T, where each processor has a limited instruction set and the input is given in n blocks of n bits each, can be simulated by an unbounded fan-in circuit of depth $O(T)$ and size bounded by a polynomial in n, p, and T.*

Using the known lower bounds on parity [Hås87, FSS81, Yao85, Ajt83], they managed to prove a non-constant time lower bound for the problem of computing parity on PRAMs with polynomial number of processors.

By completely different techniques, Beame and Håstad [BH89] proved an optimal lower bound without placing restriction on the instruction set of individual processors of CRCW PRAMs.

For the case of limited word-size, Bellantoni gave a simulation of PRAMs by unbounded fan-in circuits, without any restriction on the computational power of individual processors of PRAMs.

Theorem 5.7 *(Bellantoni [Bel91]) A priority CRCW PRAM with word-size μ and p processors running in time T can be simulated by an unbounded fan-in circuits of depth $O(T)$ and size $2^{O(T\mu)}p^{O(1)}$.*

We prove a similar theorem regarding simulation of multiprefix PRAMs by MP-circuits.

Theorem 5.8 *For any $p, \mu, T > 0$, and $\mathcal{O} \subseteq$ VALID(μ), any MP-PRAM($\mathcal{O}, p, \mu$) running in time T can be simulated by an MP-Circuit($\mathcal{O}$, μ) of depth $O(T)$ and size $O(2^{4\mu T}(pT)^{O(1)}\mu)$.*

We also state a corollary of the above theorem. The rest of the chapter is devoted to proving this theorem and the corollary.

Corollary 5.9 *If m is a prime, and r is not a power of m then for any $p = 2^{\log^{O(1)} n}$, any MP-PRAM($\{MOD_m\}, p, \log p$) solving MOD_r on n bits runs for $\Omega(\frac{\log n}{\log \log n})$ time, where $MOD_m(x_1, \ldots, x_n)$ is defined to be 0 if $\sum X_i \equiv 0 \pmod m$, and 1 otherwise.*

Following the model of Bellantoni [Bel91] in our simulation, we initially assume that we do not have to worry about simulating memory. This motivates the following definition.

Definition 5.10 *For any $p, \mu > 0$, and $\mathcal{O} \subseteq$ VALID(μ), let a memoryless MP-PRAM($\mathcal{O}$,p,μ) be an MP-PRAM($\mathcal{O}, p, \mu$) such that after each step its entire global shared memory is reset to zero.*

The proof consists of two parts: first, we show that a memoryless machine can be efficiently simulated by circuits; next, we show how to simulate a general multiprefix PRAM by a memoryless multiprefix PRAM.

5.3.1 Simulation of Memoryless PRAMs by Circuits

Lemma 5.11 *For any p, μ, T, and $\mathcal{O} \subseteq$ VALID(μ), any memoryless MP-PRAM($\mathcal{O}$,p,μ) running in time T can be simulated by an MP-Circuit($\mathcal{O}, \mu$) of depth $O(T)$ and size $2^{2\mu T} p^{O(1)} \mu T$.*

Proof: In any step, a processor reads μ bits of information and receives another μ bits of information by performing a multiprefix operation. Thus, after t steps, a processor can be in one of at most $2^{2\mu t}$ possible states. Since, in any step, a processor can be in one of three ("write", "multiprefix", or "read") phases, there are at most $N(t) = 3p \sum_{j=1}^{t} 2^{2\mu j} \leq 3p2^{2\mu(t+1)}$ memory locations that ever get accessed during the first t steps by any processor.

If the memoryless multiprefix PRAM runs for T steps then there are at most $N(T)$ memory locations that possibly ever get accessed. We can index these memory locations by using $\log N(T)$ bits.

For any t, we construct a constant depth circuit to simulate step t of the computation of the memoryless multiprefix PRAM. We refer to this as the stage(t) circuit.

As we remarked earlier, any processor by the end of step $t-1$ has received $2\mu(t-1)$ bits of information. Together, the p processors receive $2p\mu(t-1)$ bits of information. The stage(t) circuit takes as input these $2p\mu(t-1)$ bits and outputs $2p\mu t$ bits which will be input to the stage($t+1$) circuit. Next, we describe the stage(t) circuit.

First, we use $2\mu(t-1)$ bits, for each processor P, to extract the following information for step t.

1. W_P = Index of the memory location into which P writes.
2. M_P = Index of the memory location for multiprefix operation of P.
3. R_P = Index of the memory location that P reads.
4. Multiprefix operation of P.
5. Value that P is going to write.
6. Value for the multiprefix operation of P.

Altogether this is $O(\log N(t)+\mu) = O(\log N(t))$ bits of information. For each of the p processors, we can build a decoder which, given $2\mu(t-1)$ bits, outputs these $O(\log N(t))$ bits for that processor. Each of these decoders can be constructed as a constant depth circuit of size $O(2^{2\mu t}\log N(t))$. So, this part of the simulation can be performed by a constant depth circuit of size $O(p2^{2\mu t}\log N(t))$.

Our task is to compute the 2μ bits of information that any processor P receives as a result of the READ and multiprefix operation in step t. Since a processor does not receive any information by writing into a cell, we do not need to simulate the

WRITE operation explicitly. As we will see later, we still need to know the value in certain memory cells at the end of the write phase in order to simulate the READ and multiprefix operations.

Assume that the gates have been arranged in such a fashion that the gates corresponding to lower indexed processors are to the left of higher numbered processors. Let us first consider the multiprefix operation. Compare M_P with W_Q for every processor Q. Select the highest priority matched processor L. From the gates encoding L's state extract the value v that L is going to write. Then memory location M_P will contain the value v at the end of the write phase in step t. Consider a gate g computing the multiprefix operation of P. The leftmost input to g is v. To compute the other inputs of g, compare M_P with M_Q for every processor Q strictly to the left of P. If there is a match then the value for the multiprefix operation of Q is sent to g; otherwise the unit for the multiprefix operation of P is sent to gate g. The output of this gate is what P receives as a result of the multiprefix operation.

The simulation of the READ operation is done similarly. As in the previous simulation, we find the value v that memory location R_P will contain at the end of the write phase in step t. Next, we find out if a multiprefix operation has been performed on memory location R_P. We compare R_P with M_Q for every processor Q. All the matched processors should have the same multiprefix operation. If there is a match, the value in the location R_P has been updated by a multiprefix operation. Consider a gate g computing the multiprefix operation of these matched processors. The leftmost input to g is v. For each processor Q, if there is a match we send the value, which Q inputs to its multiprefix operation defined above, to gate g, otherwise we send the unit for the function computed by g. The output of this gate is what P receives as a result of the multiprefix operation.

In both these constructions, we are comparing $O(\log N(t))$ bits of information among processors. So this can be performed by circuits of constant depth and $p^{O(1)} \log N(t)$ size. For each processor P, placing the 2μ bits of information next to the $2\mu(t-1)$ bits that P has received in the first $(t-1)$ steps (which is input to the stage(t) circuit) constitutes the input to the stage$(t+1)$ circuit.

The stage(t) circuit has constant depth and $O(2^{2\mu t} p \log N(t)) + p^{O(1)} \log N(t)$

size. Since $N(t) \leq 3p2^{2\mu(t+1)}$, $\log N(t)$ is $O(\log p + \mu t)$. This says that the stage(t) circuit has constant depth and $O(p^{O(1)}2^{2\mu t}\mu t)$ size which in turn implies that the final circuit has depth $O(T)$ and size

$$O\left(p^{O(1)} \sum_{1 \leq t \leq T} 2^{2\mu t}\mu t\right) = O\left(p^{O(1)}\mu T \sum_{1 \leq t \leq T} 2^{2\mu t}\right) = O\left(p^{O(1)}\mu T 2^{2\mu T}\right).$$

□

Next, we show how to simulate a general multiprefix PRAM by a memoryless multiprefix PRAM.

5.3.2 Simulation of Multiprefix PRAMs by Memoryless Multiprefix PRAMs

Lemma 5.12 *For any $p, \mu, T > 0$, and $\mathcal{O} \subseteq$ VALID(μ), any MP-PRAM($\mathcal{O}, p, \mu$) M running in time T can be simulated by a memoryless MP-PRAM($\mathcal{O}$,$2pT + n$,μ) M' running in time $2T - 1$, where n is the size of the input.*

Proof: The essential idea of this proof follows that of Bellantoni [Bel91] but there are some additional complications due to the multiprefix phase. The $2pT + n$ processors of M' are numbered $Q_1, \ldots, Q_n$, and P'_{tj}, P''_{tj}, $1 \leq t \leq T, 1 \leq j \leq p$. The p processors of M are numbered $P_1, \ldots, P_p$. We will sketch a simulation that almost works but has a few problems. Later, we will suggest modifications to fix these problems.

Each step t of M is simulated by the set of p processors $P'_{t1}, \ldots, P'_{tp}$ of M'. Somehow, we need to remember the contents of all the relevant memory locations of M. In the very first step of M', $Q_1, \ldots, Q_n$ read the input from memory locations $L_1, \ldots, L_n$. In all subsequent steps they write back the input in the corresponding locations. The general idea is that any processor updating the contents of a memory location should write that value in all subsequent steps. For any t, the set of p processors $P'_{t1}, \ldots, P'_{tp}$ acquire the necessary information about the input in the first $t - 1$ steps so that they can simulate step t of M. In all steps subsequent to t, they perform the same write and multiprefix operation as in step t. We need to arrange the priorities of processors so that the processor writing the current value has the highest priority. It is useful to think of any processor being in either read or write mode. In the read mode, a processor receives all the information without

making any writes. It then simulates the behavior of a processor in M, and makes a transition to the write mode where it writes the same value in all subsequent steps.

This simplistic scheme has its problems: any processor in its read mode should be able to get the information that the corresponding processor of M receives by a multiprefix operation. However, unlike READ, we can not allow multiple copies of the same processor to perform a multiprefix operation. We fix this problem by doubling the number of steps. The real simulation is performed in odd steps, whereas even steps are used to pass the information received as a result of the multiprefix operations.

Another problem is that both WRITE and multiprefix operations affect memory locations, and they may interact in a tricky way. For example, if a series of writes and multiprefix operations are performed on the same memory location L, then the net result is the same as obtained by performing that series of operations starting with the last WRITE. Furthermore, we restrict all processors performing multiprefix operation on the same memory location in M' (as well as in M) to have the same operation. So, if for a particular memory location L, the operation for round t_1 is $\odot_1$ and for round t_2 it is $\odot_2$, we cannot allow these two multiprefix operations to be done in the same step (which our simplistic scheme would have required in all steps subsequent to t_1 and t_2).

We avoid all these complications by letting another set of p processors $P''_{t1}, \ldots, P''_{tp}$ remember the contents of the memory locations on which a multiprefix operation has been performed in step t. Now, it suffices for processors $P'_{t1}, \ldots, P'_{tp}, P''_{t1}, \ldots, P''_{tp}$ to perform only a WRITE (and not multiprefix) operation in all steps subsequent to t. Assume that the processors of M have been assigned priorities according to the order $P_p < P_{p-1} < \cdots < P_2 < P_1$. The $2pt+n$ processors of M' are assigned priorities according to the order $Q_n \cdots < Q_1 < P'_{1p} \cdots < P'_{11} < P''_{1p} \cdots P''_{11} < P'_{2p} \cdots < P'_{21} < \cdots < P'_{pp} \cdots < P''_{p1}$. The details are as follows.

In step $2t-1$, $\quad 1 \leq t \leq T$,

- If $t > 1$ then $Q_1, \ldots, Q_n$ WRITE $X_1, \ldots, X_n$ back into input locations $L_1, \ldots, L_n$ respectively.

- $P'_{t1}, \ldots, P'_{tp}$ perform the same WRITE as $P_1, \ldots, P_p$ in step t of M.

- $P'_{k1}, \ldots, P'_{kp}$, for $k < t$, perform the same WRITE operation as they did in step $2k-1$.

- $P''_{k1}, \ldots, P''_{kp}$, for $k < t$, WRITE back the values they READ in step $2k-1$.

- $P'_{t1}, \ldots, P'_{tp}$ perform the same multiprefix operation as $P_1, \ldots, P_p$ in step t of M.

- If $t = 1$ then $Q_1, \ldots, Q_n$ READ the input $X_1, \ldots, X_n$ from input locations $L_1, \ldots, L_n$ respectively.

- $P'_{k1}, \ldots, P'_{kp}, P''_{k1}, \ldots, P''_{kp}$, for $k > t$, perform the same READ as $P_1, \ldots, P_p$ in step t of M.

- $P''_{t1}, \ldots, P''_{tp}$ READ the memory locations where $P'_{t1}, \ldots, P'_{tp}$ have performed the multiprefix operation.

In step $2t$, $1 \leq t \leq T-1$,

- $P'_{t1}, \ldots, P'_{tp}$ WRITE the value that they received as a result of their multiprefix operation in step $2t-1$ in some designated p memory locations.

- $P'_{k1}, \ldots, P'_{kp}, P''_{k1}, \ldots, P''_{kp}$, for $k > t$ READ those designated memory locations.

□

We are now ready for the proofs of Theorem 5.8 and Corollary 5.9. We restate them for convenience.

Theorem 5.8 For any $p, \mu, T > 0$, and $\mathcal{O} \subseteq \text{VALID}(\mu)$, any MP-PRAM$(\mathcal{O}, p, \mu)$ running in time T can be simulated by an MP-Circuit$(\mathcal{O}, \mu)$ of depth $O(T)$ and size $O(2^{4\mu T}(pT)^{O(1)}\mu)$.

Proof: Immediate from Lemma 5.11, Lemma 5.12 and the observation that if the output of the MP-PRAM$(\mathcal{O}, p, \mu)$ depends on n inputs then $pT \geq n$. □

For the special case $\mu = \log p$, we have the following corollary.

Corollary 5.13 *For any $T, p > 0$, and $\mathcal{O} \subseteq$ VALID($\log p$), any MP-PRAM($\mathcal{O}$, p, $\log p$) running in time T can be simulated by an MP-Circuit($\mathcal{O}$, $\log p$) of depth $O(T)$ and size $p^{O(T)}$.*

Corollary 5.9 If m is a prime, and r is not a power of m then for any $p = 2^{\log^{O(1)} n}$, any MP-PRAM($\{MOD_m\}, p, \log p$) solving MOD_r on n bits runs for $\Omega(\frac{\log n}{\log \log n})$ time, where $MOD_m(x_1, \ldots, x_n)$ is defined to be 0 if $\sum X_i \equiv 0 \pmod m$, and 1 otherwise.

Proof: Suppose that the MP-PRAM($\{MOD_m\}, p, \log p$) runs for T steps. Then by Corollary 5.13, there is an unbounded fan-in circuit with AND, OR, NOT, MOD_m gates of depth $O(T)$ and size $p^{O(T)}$ which computes MOD_r. However, Smolensky[Smo87] proved that any such circuit has size $\Omega(2^{n^{1/2T}})$. This says that

$$p^{O(T)} = \Omega(2^{n^{1/2T}}).$$

Substituting $p = 2^{\log^{O(1)} n}$, we get $T = \Omega(\frac{\log n}{\log \log n})$. □

5.4 Conclusion

We showed that a multiprefix PRAM with restricted bandwidth can be very efficiently simulated by an unbounded fan-in circuit with special gates for computing the multiprefix operations. In the case of PRIORITY CRCW PRAMs, lower bounds for unbounded fan-in circuits [Hås87] provided the pattern for PRAM lower bounds [BH89]. What we demonstrate is that for natural restrictions of multiprefix PRAM word-length, bounds for circuits with prefix operations directly translate into bounds for the multiprefix PRAM. In the case of lower bounds, it still remains to extend the results for unbounded fan-in circuits with more powerful gates as primitives such as those described in [Raz87], [Smo87] and [HMP$^+$93].

Parberry and Schnitger [PS88] defined and analyzed *TRAMs* (Threshold RAMs) that are CRCW PRAMs whose write resolution rule corresponds to computing a threshold function of the values that processors are attempting to write. Parberry and Schnitger, by using techniques of Stockmeyer and Vishkin [SV84], were able to show an efficient simulation of TRAMs by *threshold circuits* which are unbounded fan-in circuits with gates computing AND, OR, NOT, and threshold functions. Our techniques can be employed to give an alternate proof of their result.

Chapter 6

COMPUTING SUM ON THE SUB-BUS MESH

6.1 Introduction

6.1.1 The Sub-Bus Mesh Model

Mesh connected computers have found a lot of appeal with computer architects because of their low cost of processor interconnections. Their simple and regular design has also made them an attractive model with theoreticians interested in designing parallel algorithms. The model has several variants which have all been the subject of extensive study [HS86, Lei92, LS91, MS89, MPKRS93, RPK88, Sto86]. We will focus on the *sub-bus mesh* model, which has been implemented on the commercially available MasPar MP-1 [Bla90].

A sub-bus mesh is a single-instruction multiple-data (SIMD) two-dimensional array of processors. An $m_1 \times m_2$ sub-bus mesh has a processor placed at every grid point of a mesh with m_1 rows and m_2 columns. The processors are connected by m_1 row-busses and m_2 column-busses. Each row-bus connects all the processors belonging to a given row. Similarly, each column-bus connects all the processors belonging to a given column. There is a switch on every segment of the bus connecting two adjacent grid points. In every computational cycle, the processors, by using these switches, can dynamically segment the busses so that each segment becomes a dedicated bus for the use of the set of consecutive processors connected to this segment.

Because of the SIMD nature of the machine, we assume that all broadcasts in any particular step are in the same direction (that is, left, right, up, or down); furthermore, any valid algorithm makes sure that in any broadcast step, exactly one processor on any segment of the bus does the broadcast which reaches all other processors connected to this segment.

Now we define the model formally. As is standard practice, to keep the model

simple, we describe a rather simple version of the sub-bus mesh computer architecture. Actual machines have a richer organization.

Each processor knows its row and column index. The sub-busses go in four directions, `up`, `down`, `right`, and `left`. All the row and column busses are assumed to be circular. Processor (x, y) is immediately below processor $(x, (y+1) \bmod m_1)$ and immediately to the left of processor $((x+1) \bmod m_2, y)$. Each processor is a RAM with local memory. The instruction set of processors includes direct and indirect Boolean operations, arithmetic operations, shifts, and comparisons. In addition, for any fixed $q > 0$, we allow the processors to compute *q-parity*(x) for $1 \leq x < \max(m_1, m_2)$ in one step, where for any integer x, q-parity(x) is defined as the summation modulo q of the digits in the base q representation of x. Processor $(0, 0)$ is the special *front-end* processor which runs the parallel program. The front-end can also perform normal branching operations and issue *parallel instructions*.

We find it somewhat unsatisfactory to treat q-parity as a primitive instruction, but our main result in this chapter — an optimal algorithm for computing SUM — is crucially dependent on the availability of this instruction. For any fixed $q, \ell \geq 0$, and any integer x between q^ℓ and $q^{\ell+1} - 1$,

$$q\text{-parity}(x) = [1 + q\text{-parity}(x - q^\ell)] \bmod q.$$

Thus, on any one processor, we can compute q-parity(x) for a fixed q and all integers x between 0 and $\max(m_1, m_2)$ in $O(\max(m_1, m_2))$ steps of preprocessing. Also notice that if $q > x$ then q-parity$(x) = x$. So we can compute q-parity(x) for all required values of q and x, that is, for $0 < q < x < \max(m_1, m_2)$, in $O((\max(m_1, m_2))^2)$ steps of preprocessing. If preprocessing is not allowed, q-parity can be computed in uniform NC^1 [BIS90] (details are given in [CLLS]).

A parallel instruction issued by the front-end has the form "**if** <*condition*> **then** <*statement*>." Each processor evaluates the condition, which can be any sequence of non-branching operations which evaluates to a Boolean value. If the condition is true then the processor is said to be *active*, otherwise it is said to be *inactive*. Only the active processors execute the statement part of the instruction.

There are two kinds of statements, *local operations* and *segmented broadcasts*. A local operation is just a typical non-branching RAM operation executed at each

processor. A segmented broadcast has the form

`broadcast_`*direction*`[`*distance*`] .` *variable* $\leftarrow$ *variable*`;`

The *direction* can be either `left, right, up` or `down`. The variable *distance* must be the same for all processors. When an active processor i executes the instruction `broadcast_right[`d`] .y` $\leftarrow$ `x` then the location `y` at the processors $(i+1) \bmod m_2, (i+2) \bmod m_2, ..., (i+j) \bmod m_2$ receive the value stored in location `x` of processor i, where processors $(i+1) \bmod m_2, (i+2) \bmod m_2, ..., (i+j-1) \bmod m_2$ are inactive and either $j = d$, or $j < d$ and processor $(i+j) \bmod m_2$ is active. The segmented broadcasts to the left, up, or down are similar in nature. In all case, the particular row or column bus is partitioned into non-overlapping segments. Each segment behaves like a sub-bus of the bus which includes all the processors. The MasPar MP-1 implements the segmented broadcast as `xnetc`. Table 1 describes the result of a segmented broadcast to the right on processors in a given row.

	`broadcast_right[2].y = x`							
`PID`	0	1	2	3	4	5	6	7
active	no	no	yes	yes	no	no	no	yes
`x`	a	b	c	d	e	f	g	h
`y`	h	h	*	c	d	d	*	*

Table 1. Demonstration of segmented broadcast. The * indicates that the value of `y` did not change because of the broadcast.

We say that an $m_1 \times m_2$ mesh computes a function on $p = m_1 m_2$ inputs, if we start with the p inputs distributed one per processor, and at the end of the computation, the front-end knows the answer.

The most important complexity measure for us is *time*. For the purpose of analyzing our algorithms we consider time to be evaluated using the unit cost RAM criterion where the values operated upon must have length $O(\log p)$. Each sequential operation by the front-end, each parallel operation used in evaluating the condition in a parallel instruction, and each statement of a parallel instruction costs 1 in our model. We do not charge for the broadcast of parallel instructions

by the front-end to the mesh processors. We assume that cost is dominated by the cost of executing the parallel instruction.

We consider the problem of summing the input bits on a $\sqrt{p} \times \sqrt{p}$ mesh when each processor starts with an input integer of length $O(\log p)$. It is a very basic operation and is used as a subroutine in many important algorithms.

Theorem 6.1 *[CLLS] Any algorithm for computing SUM on the $\sqrt{p} \times \sqrt{p}$ sub-bus must run for at least $\Omega\left(\frac{\log p}{\log \log p}\right)$ steps.*

Proof: It was shown in [CLLS] that a priority CRCW PRAM can simulate a sub-bus mesh computer to within a constant factor of the time and within a polynomial number of processors.

Since Beame and Håstad [BH89] have proved a lower bound of time $\Omega\left(\frac{\log p}{\log \log p}\right)$ on the time to compute SUM on a priority CRCW PRAM with a polynomial number of processors, a similar lower bound follows for the case of sub-bus mesh. □

There is a trivial algorithm (Proposition 6.4) to compute any associative function in time $\Theta(\log p)$. Our main result in this section is an optimal algorithm for computing SUM.

Theorem 6.2 *On a $\sqrt{p} \times \sqrt{p}$ mesh with each processor having an input integer of length $O(\log p)$, SUM can be computed in time*

$$O\left(\frac{\log p}{\log \log p}\right).$$

The algorithm uses mixed radix arithmetic, the Chinese remainder theorem and recursion to achieve the result.

6.1.2 Related Results

A great deal of research has been done on the sub-bus mesh and related models.

The sub-bus mesh architecture was first investigated by Reisis and Prasanna Kumar [RPK88]. We will restate their observations about computing several basic

functions. These will be used as subroutines in our optimal algorithm for computing SUM.

Proposition 6.3 *The OR or AND of input bits can be computed in constant time on any sub-bus mesh.*

Proof: We will give the algorithm for computing OR.

First, each processor, whose input bit is 1, tries to broadcast 1 to all other processors in its row. Clearly a processor receives 1 if and only if at least one input bit in its row is 1. This has the effect of computing OR within each row. Then every processor in the leftmost column, for which the OR of input bits within its row is 1, tries to broadcast 1 to all other processors in its column. The front-end processor receives a 1 if and only if the OR of all the input bits is 1.

Using DeMorgan's law, AND can similarly be computed in constant time. □

Proposition 6.4 *For any binary associative operation, $\oplus$, the function REDUCE-$\oplus(x_0, x_1, \ldots, x_{p-1}) = x_0 \oplus x_1 \oplus \cdots \oplus x_{p-1}$ can be computed in $O(\log p)$ time on any $m_1 \times m_2$ sub-bus mesh such that $m_1 m_2 = p$.*

Proof: On an $m_1 \times m_2$ sub-bus mesh, using a "binary-tree algorithm" (that is, by repeatedly combining adjacent pairs of inputs and thereby halving the number of inputs in each step) we can reduce inputs within any row in time $\lceil \log m_2 \rceil$. This leaves m_1 partial results, one per row. Another "binary tree algorithm" on these m_1 partial results lets us reduce all the inputs in an additional $\lceil \log m_1 \rceil$ steps. The running time is $\lceil \log m_2 \rceil + \lceil \log m_1 \rceil = O(\log p)$. □

For the case of the $1 \times p$ sub-bus mesh, Condon et al. [CLLS] proved a lower bound of $\log p$ on the PARITY function and a lower bound of $\log_3 (2 \min(k, p-k))$ on the kth threshold function. That paper also contains Theorem 6.2 and an efficient simulation of sub-bus meshes by CRCW PRAMs. The lower bound on PARITY on the $1 \times p$ sub-bus mesh was (earlier) independently obtained by MacKenzie [Mac93]. This lower bound is tight because of Proposition 6.4.

Two variants of the mesh computer are closely related to the sub-bus mesh. First, there is the full-bus mesh, which does not allow segmentation of busses.

That is, on a left or right broadcast, at most one processor on any row is active and its data reaches all other processors in its row, and similarly for up and down broadcasts. The MPP of Goodyear and NASA is an example of a full-bus two-dimensional mesh computer [Bat80]. Full-bus meshes are generally less powerful than sub-bus meshes: Both PARITY and finding the minimum of input vales requires $\Omega(p^\alpha)$ time for some $\alpha > 0$ on full-bus meshes [BNP91, PKR87].

Second, there is the reconfigurable mesh, in which every processor connects to the busses by 4 ports (N, S, E, W) – 3 for processors on the sides; 2 for processors in the corners. By internally connecting some subsets of these ports to each other, the executing program is allowed to change the topology of the mesh[LS91]. Several prototype of reconfigurable mesh computers, though non-commercial, have been built. The model comes in two flavors: In the *cross-over* model, processors are allowed to independently connect their N-S ports together and their E-W ports together; such connections are not allowed in the *non-cross-over* model.

It is well known that PARITY can be computed in constant time on the cross-over $m_1 \times m_2$ mesh if $m_1, m_2 \geq 3$. The basic idea of the algorithm is the same as in the branching program for computing parity in Proposition 3.10 (see [LS91] for details). Thus the $\Theta(\log p)$ bound for PARITY on the sub-bus mesh demonstrate that the sub-bus mesh computer architecture is strictly more powerful than the full-bus mesh computer architecture, but strictly less powerful than the reconfigurable cross-over mesh computer. In another work on the PARITY function, MacKenzie [Mac93] proved a lower bound of $\Omega((\log m_1)/m_2)$ for computing PARITY on an $m_1 \times m_2$ non-cross-over reconfigurable mesh model, thereby showing a separation between the powers of the two flavors of the reconfigurable mesh model.

The SUM function has also been previously studied on the reconfigurable mesh. Nakano [Nak93], improving a result of Nakano, Masuzawa and Tokura [NMT91], developed algorithms for summing n binary values on an $n \times m$ reconfigurable mesh in time $O(\log n/\sqrt{m/\log n})$. His result also uses the Chinese remainder theorem, but does not apply directly to the sub-bus mesh architecture.

6.2 Algorithms

This section gives an optimal algorithm for computing SUM on a $\sqrt{p} \times \sqrt{p}$ sub-bus mesh model (Theorem 6.2). As a warm-up step, we first solve a simpler problem: We show that on a $\sqrt{p} \times \sqrt{p}$ sub-bus mesh, where each processor starts with an input bit, the PARITY function can be computed in time $O\left(\frac{\log p}{\log \log p}\right)$.

6.2.1 PARITY Algorithm

We will introduce a series of problems, in increasing order of difficulty. The algorithm for each problem will lead to the next one with some fresh tricks. This will help us concentrate on one idea at a time.

We will describe the algorithms informally. Each of the algorithms below can be executed on a *sub-array* of the $\sqrt{p} \times \sqrt{p}$ mesh. By an *array* or *sub-array*, we mean a sub-bus mesh of the full mesh which may be non-square and non-contiguous. In the case it is non-contiguous, it is assumed that the processors between any two processors in the sub-array are inactive so as not to interfere with communication between the processors in the sub-array. Furthermore, any of the algorithms below can be executed in parallel on disjoint sub-arrays of the full $\sqrt{p} \times \sqrt{p}$ array such that their computations do not interfere with each other. If the algorithm is executed on an $m \times n$ sub-array, then we say processor (i, j) is the processor in the (i, j)-th position (the i-th column and j-th row) of the sub-array, where $0 \leq i < n$ and $0 \leq j < m$. Although it is not generally the case that processor (i, j) has its column-index $= i$ and row-index $= j$, it will always be the case that i, j, and the dimensions of the sub-array can be computed from the row and column index of the processor and other local data in constant time.

Lemma 6.5 *On a $2^n \times n$ array with each processor in the bottom row having an input bit, the parity of the input bits can be computed in constant time.*

Proof: If we think of the input vector as the bit representation of an integer, each input corresponds to an integer between 0 and $2^n - 1$. The basic idea is for processors in row j (for $0 \leq j < 2^n$) to check if the input corresponds to integer j. This will be enough because then any processor with knowledge of j can compute the PARITY function as 2-parity(j).

First the processors in the bottom row do a broadcast of the inputs up the columns. Then processor (i, j) checks whether the input agrees, on the ith bit, with the bit representation of integer j. A constant time AND (Proposition 6.3) within every row will tell the processors in the leftmost column if the input corresponds to an integer equal to their row index. For j, such that the input– considered as an integer– equals j, the processor $(0, j)$ communicates the value of j to the front-end processor, which in turn computes 2-parity(j). □

We saw that with exponentially many rows we can compute the parity in constant time. In general, if we have more than a constant number of rows, we can beat the straightforward $O(\log n)$ time algorithm.

Lemma 6.6 *On an $m \times n$ array with each processor in the bottom row having an input bit, the parity of the input bits can be computed in time $O\left(\frac{\log n}{\log \log m}\right)$.*

Proof: Let $\ell = \lfloor \log m \rfloor$. Subdivide the original $m \times n$ array into $\lceil \frac{n}{\ell} \rceil$ sub-arrays placed side by side such that each sub-array has m rows and at most ℓ columns (see Figure 6.1). As in the previous proof, we can compute the parity of ℓ input bits in any sub-array in constant time. This leaves $\lceil \frac{n}{\ell} \rceil$ partial results in the bottom row of a sub-array of dimension $m \times \lceil \frac{n}{\ell} \rceil$. Repeating the process $O\left(\frac{\log n}{\log \ell}\right) = O\left(\frac{\log n}{\log \log m}\right)$ times we have the parity of all the n bits. □

So far we have been assuming that only the processors in the bottom row have inputs. Let us now consider the case where each processor has an input.

Lemma 6.7 *On an $m \times n$ array with each processor having an input bit, the parity of the input bits can be computed in time $O\left(\log m + \frac{\log n}{\log \log m}\right)$.*

Proof: First, in parallel, the processors within each column run the one-dimensional PARITY algorithm described in Proposition 6.4. This part takes $O(\log m)$ time. At this point, we have partial results stored in the bottom row. From the previous lemma, the parity of these partial results can be computed in an additional $O\left(\frac{\log n}{\log \log m}\right)$ steps. □

We are ready to give our PARITY algorithm.

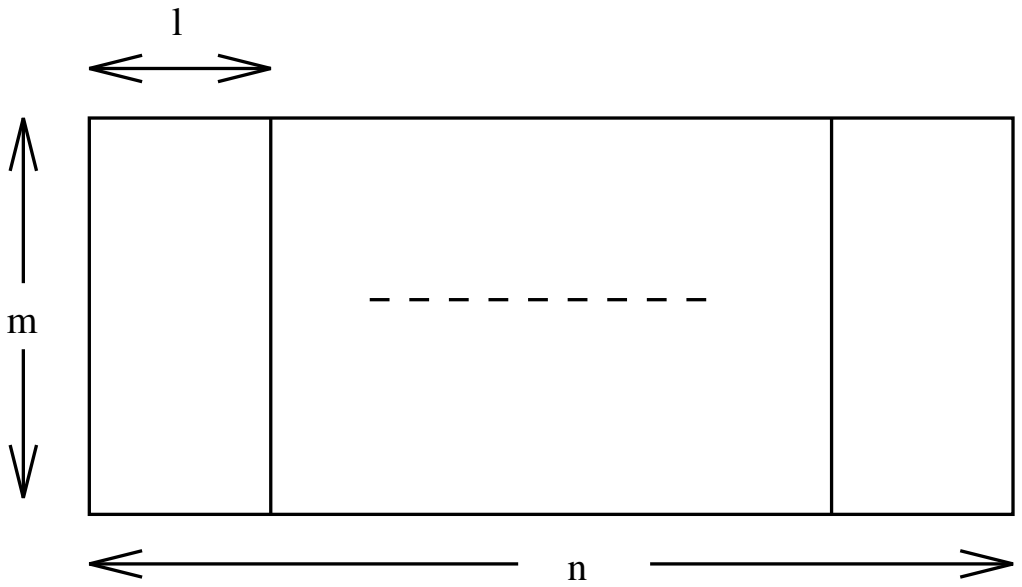


Figure 6.1: Subdivision in Lemma 6.6

Theorem 6.8 *On a $\sqrt{p} \times \sqrt{p}$ mesh with each processor having an input bit, PARITY can be computed in time $O\left(\frac{\log p}{\log \log p}\right)$.*

Proof: Subdivide the original $\sqrt{p} \times \sqrt{p}$ mesh into $\lfloor \frac{\sqrt{p}}{m} \rfloor$ sub-arrays placed on top of each other such that each sub-array has $\sqrt{p}$ columns and its number of rows is between m and $2m$ (see Figure 6.2). Each of these sub-arrays computes the parity of its input bits in parallel. By the previous lemma, this takes $O\left(\log m + \frac{\log \sqrt{p}}{\log \log m}\right)$ time and leaves $\lfloor \frac{\sqrt{p}}{m} \rfloor$ partial results in the leftmost column. Consider the $\lfloor \frac{\sqrt{p}}{m} \rfloor \times \sqrt{p}$ sub-array such that each processor in the leftmost column knows one of the $\lfloor \frac{\sqrt{p}}{m} \rfloor$ partial results. Apply Lemma 6.6, with 90 degrees rotation, to compute the parity of these partial results in an additional $O\left(\frac{\log (\sqrt{p}/m)}{\log \log \sqrt{p}}\right)$ time. The total running time is

$$O\left(\log m + \frac{\log p}{\log \log m} + \frac{\log p}{\log \log p}\right).$$

Choosing $m = \left\lceil 2^{\frac{\log p}{\log \log p}} \right\rceil$, we get a bound of $O\left(\frac{\log p}{\log \log p}\right)$. □

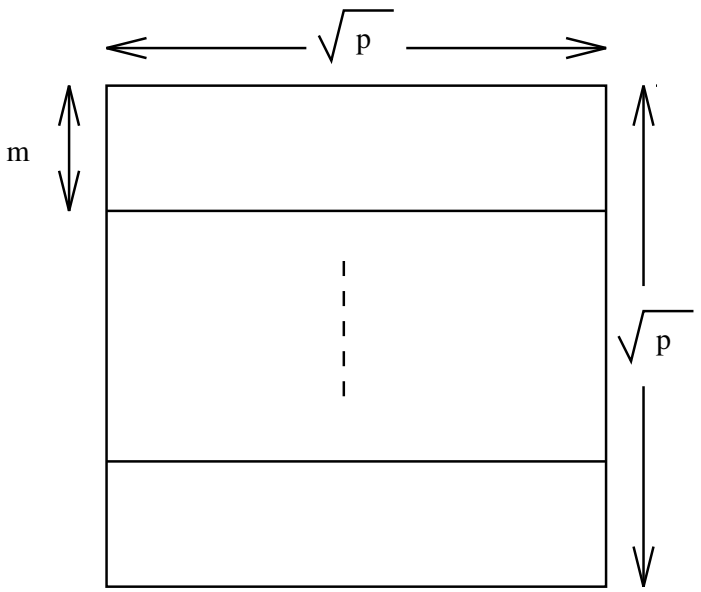


Figure 6.2: Subdivision in Lemma 6.8

6.2.2 SUM Algorithm

As in the PARITY algorithm, we will solve a series of problems, in increasing order of difficulty. The last of these algorithms will be our algorithm for computing the SUM function on a $\sqrt{p} \times \sqrt{p}$ sub-bus mesh. Figure 6.6 summarizes the major steps in the computation of the SUM function.

Computing PARITY is the same as computing the sum of the inputs modulo 2. As in the construction of branching programs for threshold and mod functions (Chapter 3), we will be using the Chinese remainder theorem (Section 3.2.2) to construct the value of SUM from its value computed modulo many small primes.

Lemmas 6.5 and 6.6 can be generalized to compute the sum, modulo a small integer, of inputs on the bottom row. For all the problems below we assume that the inputs are non-negative integers of length $O(\log p)$.

Lemma 6.9 *If $q > 0$, then on a $q^n \times n$ array with each processor having q, and with each processor in the bottom row having an input integer, the sum of the inputs modulo q can be computed in constant time.*

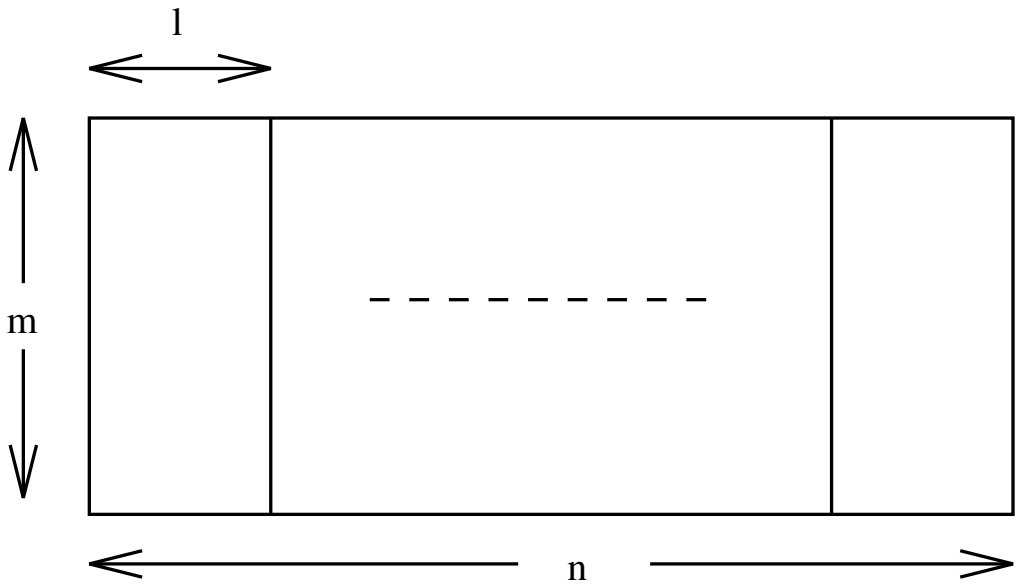


Figure 6.3: Subdivision in Lemma 6.10

Proof: The proof mimics that of Lemma 6.5. If we reduce each of the n inputs modulo q, there are q^n possible values of input. For $0 \leq j < q^n$, think of j as an integer written in base q. As in the computation of parity, processors in row j are responsible for checking whether the input reduced modulo q is the same as the integer j written in base q. In particular, processor (i, j) checks whether the i-th input, reduced modulo q, is equal to the i-th q-ary digit of j. In a constant number of steps, the front-end knows the value of the integer j such that the inputs reduced modulo q are the same as the integer j written in base q. It can then compute the summation modulo q in one extra step as q-parity(j). □

Lemma 6.10 *If $q > 0$, then on an $m \times n$ array with each processor having q, and with each processor in the bottom row having an input integer, the sum of the inputs modulo q can be computed in time $O\left(\frac{\log n}{\log(\log_q m)}\right)$.*

Proof: Let $\ell = \lfloor \log_q m \rfloor$. The original $m \times n$ array can be divided into $\lceil \frac{n}{\ell} \rceil$ sub-arrays placed side by side such that each sub-array has m rows and at most ℓ columns (see Figure 6.3). As in the previous proof, we can compute the sum

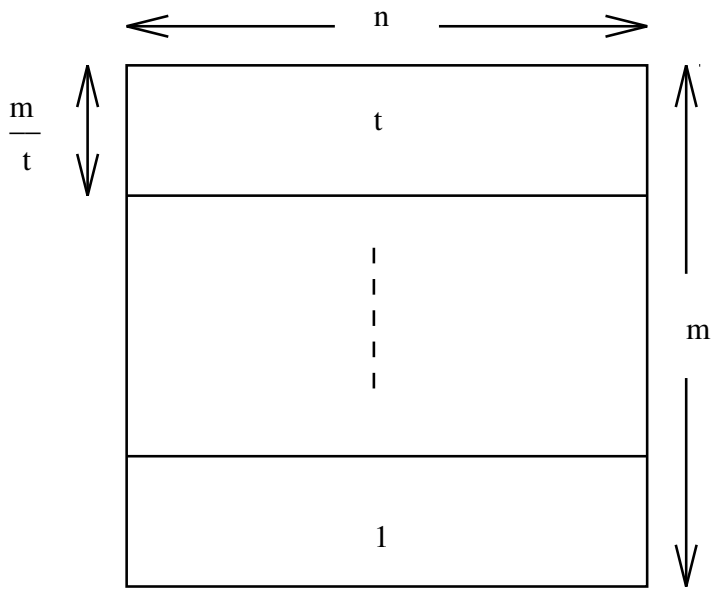


Figure 6.4: Subdivision in Lemma 6.11

modulo q of ℓ input bits in any sub-array in constant time. This leaves $\lceil \frac{n}{\ell} \rceil$ partial results in the bottom row of a sub-array of dimension $m \times \lceil \frac{n}{\ell} \rceil$. Repeating the process $O\left(\frac{\log n}{\log \ell}\right) = O\left(\frac{\log n}{\log(\log_q m)}\right)$ times gives the sum of all the n integers modulo q. □

By the Chinese remainder theorem we know that if we can compute the sum modulo sufficiently many small integers, we can compute the exact sum.

For any integer $t > 0$, let $P[t]$ denote the product of all primes between 1 and t.

Lemma 6.11 *If $0 < t < \min(m, n)$ and $t = O(\log p)$, then on an $m \times n$ array with each processor in the bottom row having an input integer, the sum of the inputs modulo $P[t]$ can be computed in time $O\left(\log t + \frac{\log n}{\log(\log_t m)}\right)$.*

Proof: Subdivide the original $m \times n$ array into t sub-arrays placed on top of each other such that each sub-array has n columns and its number of rows is either $\lfloor \frac{m}{t} \rfloor$ or $\lfloor \frac{m}{t} \rfloor + 1$ (see Figure 6.4). We already know how to compute the sum modulo small primes. Our plan is to let the jth (say, from bottom) sub-array compute the

sum modulo j and then apply the Chinese remainder algorithm to compute the sum modulo $P[t]$.

To begin with, processors in the bottom row broadcast the input values up the columns. Because the Chinese Remainder Theorem only needs the value of the summation modulo prime numbers, we compute the summation modulo j only if j is a prime. The jth sub-array decides whether j is a prime in two stages: A number j is prime if and only if it is not divisible by any number between 1 and $\sqrt{j}$. In the first stage, assign $\sqrt{j}$ processors in the first row to check for each possible divisor. In the second stage, these processors compute an AND of their results. Only processors in the j-th sub-array for prime j participate in all subsequent steps. The j-th sub-array computes a_j, the sum of the inputs modulo j. By Lemma 6.10, this can be done in

$$O\left(\frac{\log n}{\log(\log_j(m/t))}\right) = O\left(\frac{\log n}{\log(\log_t(m/t))}\right) = O\left(\frac{\log n}{\log(\log_t m)}\right)$$

time.

Next, in $O(\log t)$ steps, each processor in the j-th sub-array computes $P[t]$ (using Proposition 6.4), and $m_j = P[t]/j$. The processors in the j-th sub-array compute $(a_j m_j)((m_j)^{-1} \bmod j)$. This can be done in constant time. The nontrivial part is computing $((m_j)^{-1} \bmod j)$. There are at most j possible values for the inverse. We assign j processors in the top row of the j-th sub-array for each possible value of the inverse. In one step, each of these assigned processors can check whether it has the right value of the inverse. The processor corresponding to the right value of the inverse broadcasts this to all other processors. By the Chinese remainder theorem, the summation modulo $P[t]$ is $[\sum(a_j m_j)((m_j)^{-1} \bmod j)] \bmod P[t]$, which can be computed in $O(\log t)$ steps (using Proposition 6.4).

The running time of the algorithm is $O\left(\log t + \frac{\log n}{\log(\log_t m)}\right)$. □

We now consider the case where *each* processor has an input.

Lemma 6.12 *If $0 < t < \min(m, n)$ and $t = O(\log p)$, then on an $m \times n$ array with* each *of the mn processors having an input integer, the sum of the inputs modulo $P[t]$ can be computed in time $O\left(\log m + \frac{\log n}{\log(\log_t m)}\right)$.*

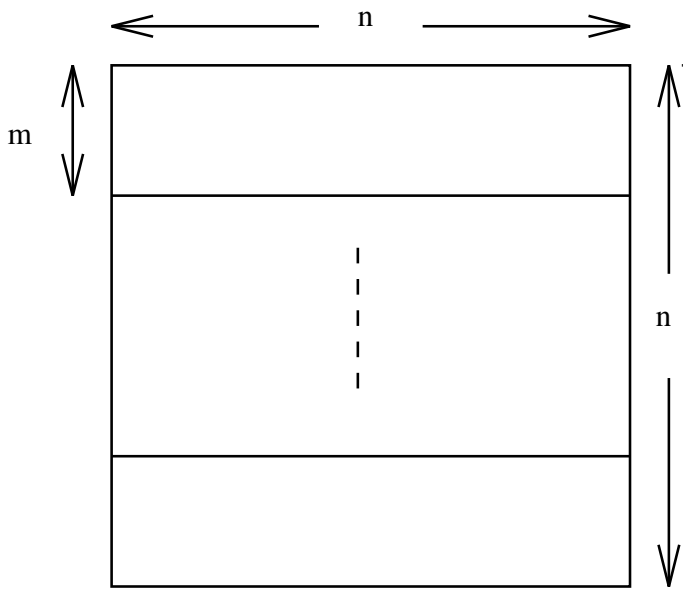


Figure 6.5: Subdivision in Lemma 6.13

Proof: First compute the sum within every column in time $O(\log m)$ and then apply the algorithm in the previous lemma. The running time is

$$O\left(\log m + \log t + \frac{\log n}{\log(\log_t m)}\right) = O\left(\log m + \frac{\log n}{\log(\log_t m)}\right).$$

□

Lemma 6.13 *If $0 < t < m < n$ and $t = O(\log p)$, then on an $n \times n$ array with* each *of the n^2 processors having an input integer, the sum of the inputs modulo $P[t]$ can be computed in time $O\left(\log m + \frac{\log n}{\log(\log_t m)}\right)$.*

Proof: Subdivide the original $n \times n$ mesh into $\lfloor \frac{n}{m} \rfloor$ sub-arrays placed on top of each other such that each sub-array has n columns and its number of rows is between m and $2m$ (see Figure 6.5). Each of these sub-arrays computes, in parallel, the sum of their inputs modulo $P[t]$. By the previous lemma, this takes time

$$O\left(\log m + \frac{\log n}{\log(\log_t m)}\right)$$

and leaves $\lfloor \frac{n}{m} \rfloor$ partial results in the leftmost column. Assume that the remaining $n - \lfloor \frac{n}{m} \rfloor$ procesors in the leftmost column are holding the value 0. Now apply Lemma 6.11, with 90 degrees rotation, on the $n \times n$ array to obtain the requisite summation modulo $P[t]$ in an additional

$$O\left(\log t + \frac{\log n}{\log(\log_t n)}\right) = O\left(\log m + \frac{\log n}{\log(\log_t m)}\right)$$

time. □

We are ready to prove our main theorem, which we restate below.

Theorem 6.2 *On a $\sqrt{p} \times \sqrt{p}$ mesh with each processor having an input integer of length $O(\log p)$, SUM can be computed in time*

$$O\left(\frac{\log p}{\log \log p}\right).$$

Proof: Let t be the smallest integer such that $P[t]$ is greater than the largest possible value of the SUM. Because each of the p input integers is an $O(\log p)$ bit integer, from [RS62, Equation 3.16, page 70], we get $t = O(\log p)$.

Now apply the algorithm in the previous lemma on a $\sqrt{p} \times \sqrt{p}$ mesh, for this choice of t, and $m = \left\lceil 2^{\frac{\log p}{\log \log p}} \right\rceil$. Then we compute the summation of input integers modulo $P[t]$.

1. Because $P[t]$ is greater than the largest possible value of SUM, computing summation of inputs modulo $P[t]$ is actually giving us the SUM.

2. Running time of the algorithm is

$$O\left(\log m + \frac{\log p}{\log(\log_t m)}\right).$$

But for large values of p, $\log_t m = \Omega\left(\frac{\log p}{(\log \log p)^2}\right)$. Thus the running time of the algorithm is

$$O\left(\frac{\log p}{\log \log p}\right).$$

□

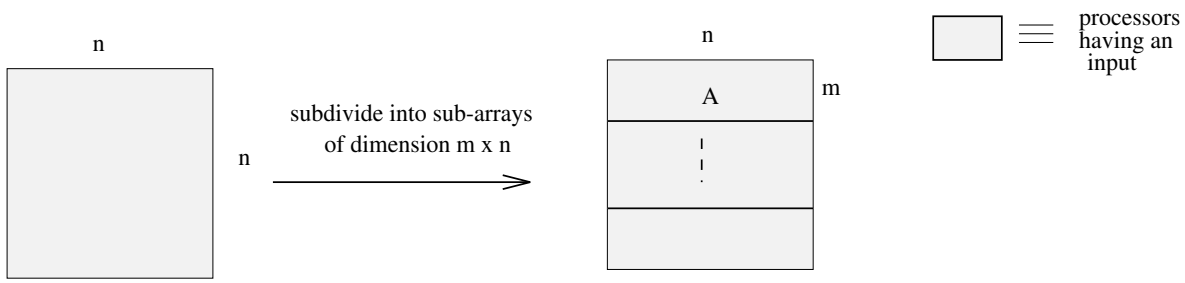
Figure 6.6 summarizes the major steps in the computation of the SUM function.

It is interesting to note that if we assume that the individual processors can operate on integers of arbitrary length in constant time, then using the technique of Theorem 6.2, the sum of p integers of length $2^{O(\sqrt{\log p/\log\log p})}$ can be computed in time $O\left(\frac{\log p}{\log\log p}\right)$.

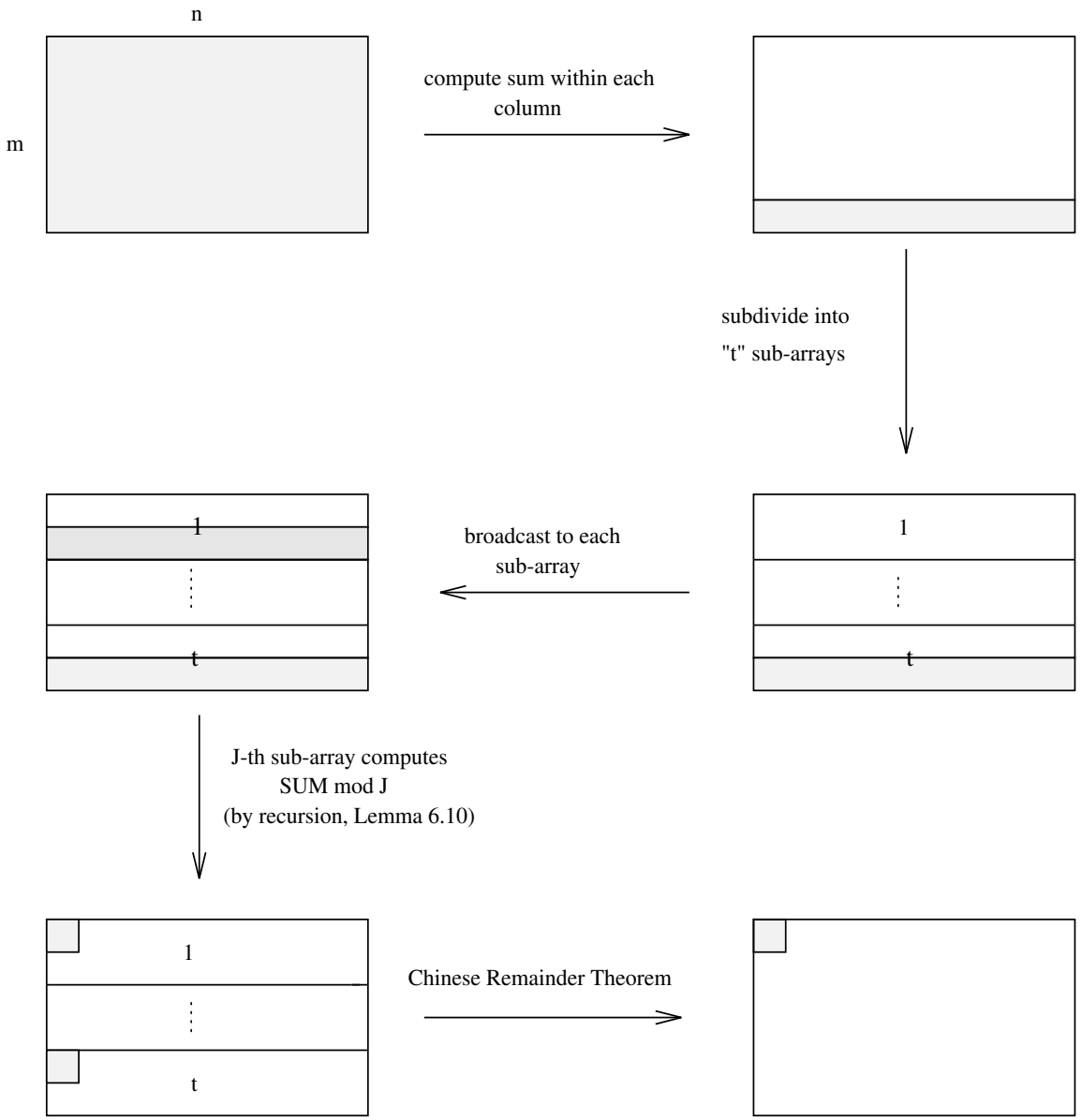
6.3 Conclusion

Because machines based on the sub-bus mesh model are commercially available [Bla90], this model deserves further study. We gave an optimal algorithm for computing the SUM function, but it is unsatisfactory on two counts: First, the algorithm is complicated and the speed-up by a factor of $\Theta(\log\log p)$ has too large a constant factor to be significant. Second, it depends on the availability of q-parity functions as primitives. Even though the algorithm is impractical, we believe that it introduces some nice ideas. Hopefully, some of these ideas can be used to design a practical algorithm for computing SUM and other functions. To borrow a term from David Johnson, in the current form, ours is a "negative-negative" result, that is, it rules out an $\omega\left(\frac{\log p}{\log\log p}\right)$ lower bound on the SUM function.

Open problem: Design a simple practical algorithm for the SUM function that runs in $o(\log p)$ steps.



Focus on sub-array A:



Back to original n x n mesh:

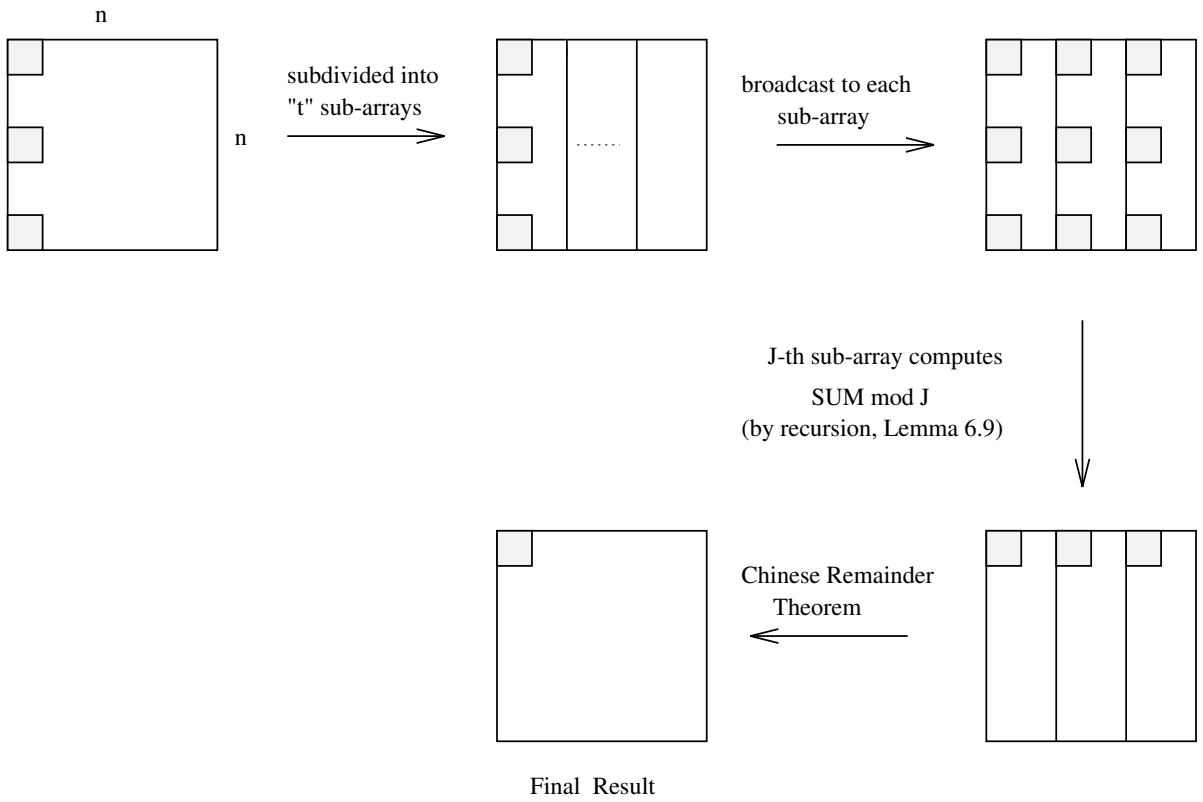


Figure 6.6: The SUM Algorithm

Chapter 7

FINAL THOUGHTS

We have proved several results on branching programs and models of parallel computation. Many of these results suggest obvious open problems related to their possible extensions. We have included all such open problems at the end of each chapter.

OBDDs have opened up a whole new area of research. For any form of representation, there is a trade-off between the size of the representation and the ease of manipulation. Among variants of branching programs, OBDDs form one extreme of the spectrum: they are very easy to manipulate but the resulting sizes are not always small. For applications where the size of the representation becomes a bottle-neck, we have to look beyond OBDDs – to more powerful forms of representations. The study of any such model has to be a two prong attack: designing efficient representations for interesting functions as well as developing better manipulation routines. Unfortunately, with the majority of these representation forms, simple counting arguments prove that only a negligible fraction of functions can possibly have efficient representation. Luckily for us, this minuscule fraction includes many functions that are needed in real applications. But it also makes the task of choosing the right representations more challenging because this activity has to be somewhat empirical in nature. That is, we have to choose a data structure that works on the subset of applications that are of interest to practitioners.

For the case of parallel computation, one of the biggest challenges is to develop better theoretical models. Some of the initial efforts on modeling were a little misdirected because we concentrated on the computational cost but ignored the communication cost. This had the unfortunate effect of generating a whole class of algorithms with a very fine grain of parallelism. These algorithms were predicted to run very efficiently on models ignoring the communication cost, but had terrible performance on real parallel machines. This suggests that communication cost

should be an important complexity measure of any algorithm. The wide variety of architectures of parallel machines and the resulting lack of a universal theoretical model has also hindered growth of the field of parallel algorithms. "Porting" – tuning an application running on one parallel machine to run efficiently on another parallel machine – is a nontrivial activity often requiring detailed knowledge of architectures of both machines. It would be nice to avoid this replication of effort. Ideally one would like to have a theoretical model such that algorithms can be described on this model and then fine-tuned for individual machines. Unfortunately, we do not see that happening in the near future. In the meanwhile, parallel algorithm designers have to continue working on a variety of models with an eye towards developing a more general universal model.

Bibliography

[Abr90] Karl R. Abrahamson. A time-space tradeoff for Boolean matrix multiplication. In *Proceedings 31st Annual Symposium on Foundations of Computer Science*, pages 412–419, St. Louis, MO, October 1990. IEEE.

[Abr91] Karl R. Abrahamson. Time–space tradeoffs for algebraic problems on general sequential models. *Journal of Computer and System Sciences*, 43(2):269–289, October 1991.

[Ajt83] Miklós Ajtai. Σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[AM88] Noga Alon and Wolfgang Maass. Meanders and their applications in lower bounds arguments. *Journal of Computer and System Sciences*, 37:118–129, 1988.

[Aza92] Yossi Azar. Lower bounds for threshold and symmetric functions in parallel computation. *SIAM Journal on Computing*, 21(2):329–338, April 1992.

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[Bat80] K.E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, C-29:836–840, 1980.

[BC82] Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM Journal on Computing*, 11(2):287–297, May 1982.

[BC94] R. E. Bryant and Y. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CS-94-160, Carnegie Mellon University, School of Computer Science, 1994.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, 1994.

[BDFP86] A. Borodin, D. Dolev, F. Fich, and W. Paul. Bounds for width two branching programs. *SIAM Journal on Computing*, 15:549–560, 1986.

[Bea86] Paul W. Beame. *Lower Bounds in Parallel Machine Computation.* PhD thesis, University of Toronto, 1986. Department of Computer Science Technical Report 198/87.

[Bea91] Paul W. Beame. A general time-space tradeoff for finding unique elements. *SIAM Journal on Computing*, 20(2):270–277, 1991.

[Bel91] S. Bellantoni. Parallel random access machines with bounded memory wordsize. *Information and Computation*, 91:259–273, 1991.

[BFK+81] Allan Borodin, Michael J. Fischer, David G. Kirkpatrick, Nancy A. Lynch, and Martin Tompa. A time-space tradeoff for sorting on non-oblivious machines. *Journal of Computer and System Sciences*, 22(3):351–364, June 1981.

[BFMadH+87] Allan Borodin, Faith E. Fich, Friedhelm Meyer auf der Heide, Eli Upfal, and Avi Wigderson. A time-space tradeoff for element distinctness. *SIAM Journal on Computing*, 16(1):97–99, February 1987.

[BFS] P. Beame, F. Fich, and R. Sinha. Separating the power of EREW and CREW PRAMs with small communication width. *Information and Computation.* To appear.

[BGS75] T. P. Baker, J. Gill, and R. Solovay. Relativizations of the P=?NP question. *SIAM Journal on Computing*, 4:431–442, 1975.

[BH89] Paul W. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, July 1989.

[BIS90] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274–306, December 1990.

[Bla90] T. Blank. The MasPar MP-1 architecture. In *COMPCON Spring 90 - The Thirty-Fifth IEEE Computer Society International Conference*, pages 20–24, February 1990.

[Ble89] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[Blu84] N. Blum. A Boolean function requiring 3n network size. *Theoretical Computer Science*, 28:337–345, 1984.

[BNP91] A. Bar-Noy and D. Peleg. Square meshes are not always optimal. *IEEE Transactions on Computers*, 40:138–147, 1991.

[Bop89] Ravi B. Boppana. Optimal separations between concurrent-write parallel machines. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 320–326, Seattle, WA, May 1989.

[Bor93] Allan Borodin. Time space tradeoffs (getting closer to the barrier?). In *4th International Symposium on Algorithms and Computation*, pages 209–229, Hong Kong, December 1993.

[BPRS90] László Babai, Pavel Pudlák, V. Rödl, and Endre Szemerédi. Lower bounds to the complexity of symmetric Boolean functions. *Theoretical Computer Science*, 74:313–324, 1990.

[BRS93] Allan Borodin, A. A. Razborov, and Roman Smolensky. On lower bounds for read-k times branching programs. *Computational Complexity*, 3:1–18, October 1993.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagram. *ACM Computing Surveys*, 24(3):283–316, 1992.

[BS91] David A. Mix Barrington and Howard Straubing. Superlinear lower bounds for bounded-width branching programs. In *Proceedings, Structure in Complexity Theory, Sixth Annual Conference*, pages 305–313, Chicago, IL, June 1991. IEEE.

[CBZ90] S. Chatterjee, Guy E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing '90*, pages 666–675. IEEE, 1990.

[CDHR88] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write pram models. In *13th Symposium on Mathematical Foundations of Computer Science*, volume 324 of *Lecture Notes in Computer Science*, pages 231–239. Springer-Verlag, 1988.

[CDR86] Steven A. Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, February 1986.

[CFL83] Ashok K. Chandra, M. L. Furst, and Richard J. Lipton. Multi-party protocols. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 94–99, Boston, MA, April 1983.

[CL89] Jin-Yi Cai and Richard J. Lipton. Subquadratic simulations of circuits by branching programs. In *30th Annual Symposium on Foundations of Computer Science*, pages 568–573, Research Triangle Park, NC, October 1989. IEEE.

[Cle91] R. Cleve. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1:91–105, 1991.

[CLLS] A. Condon, R. Ladner, J. Lampe, and R. Sinha. Complexity of sub-bus computation. *SIAM Journal on Computing*. To appear.

[Cob66] Alan Cobham. The recognition problem for the set of perfect squares. Research Paper RC-1704, IBM Watson Research Center, 1966.

[Cra88] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual*. 1988.

[DKR94] Martin Dietzfelbinger, Mirek Kutyłowski, and Rüdiger Reischuk. Exact time bounds for computing Boolean functions on PRAMs without simultaneous writes. *Journal of Computer and System Sciences*, 48(2):231–254, April 1994.

[DR86] Patrick W. Dymond and Walter L. Ruzzo. Parallel random access machines with owned global memory and deterministic context-free language recognition. In Laurent Kott, editor, *Automata, Languages, and Programming: 13th International Colloquium*, volume 226 of *Lecture Notes in Computer Science*, pages 95–104, Rennes, France, July 1986. Springer-Verlag.

[Edm91] Jeff Edmonds. Lower bounds with small domain size on concurrent write parallel machines. In *Proceedings, Structure in Complexity Theory, Sixth Annual Conference*, pages 322–332, Chicago, IL, June 1991. IEEE.

[Fic93] Faith E. Fich. The complexity of computation on the parallel random access machine. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 20, pages 843–899. Morgan Kaufmann, 1993.

[FMadHRW85] Faith Fich, Friedhelm Meyer auf der Heide, Prabhakar Ragde, and Avi Wigderson. One, two, three ... infinity: Lower bounds for parallel computation. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, Providence, RI, May 1985.

[FRW88] Faith E. Fich, Prabhakar Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation. *SIAM Journal on Computing*, 17:606–627, 1988.

[FSS81] M. Furst, J. B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. In *22nd Annual Symposium on Foundations of Computer Science*, pages 260–270, Nashville, TN, October 1981. IEEE.

[FW90] Faith E. Fich and Avi Wigderson. Towards understanding exclusive read. *SIAM Journal on Computing*, 19(4):717–727, 1990.

[GGKR83] A. Gottlieb, R. Grishman, Clyde P. Kruskal, and L. Rudolph. The NYU Ultracomputer — designing an MIMD parallel machine. *IEEE Transactions on Computers*, TC-32:175–189, 1983.

[GLR83] A. Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient co-ordination of very large number of cooperating sequential processes. *ACM Transactions on Programming Languages and Systems*, April 1983.

[GNR89] E. Gafni, Joseph Naor, and Prabhakar Ragde. On separating the EREW and CREW PRAM models. *Theoretical Computer Science*, 68(3):343–346, 1989.

[Hås87] Johan Håstad. *Computational Limitations of Small-Depth Circuits.* MIT Press, 1987. ACM Doctoral Dissertation Award Series (1986).

[Hil85] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[HMP+93] András Hajnal, Wolfgang Maass, Pavel Pudlák, Márió Szegedy, and György Turán. Threshold circuits of bounded depth. *Journal of Computer and System Sciences*, 46(2):129–154, April 1993.

[HS86] W.D. Hillis and G.L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29:1170–1183, 1986.

[HW79] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, fifth edition, 1979.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[Kar89] Mauricio Karchmer. *Communication Complexity: A New Approach to Circuit Depth.* MIT Press, 1989.

[KRS86] Clyde P. Kruskal, L. Rudolph, and Marc Snir. Efficient synchronization in multiprocessors with shared memory. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 218–228, Calgary, Alberta, Canada, August 1986.

[KRS88] Clyde P. Kruskal, L. Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC 13572, IBM, March 1988.

[Kuč82] L. Kučera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93–96, April 1982.

[Kut91] Mirek Kutyłowski. The complexity of Boolean functions on CREW PRAMs. *SIAM Journal on Computing*, 20(5):824–833, 1991.

[KW93] Mauricio Karchmer and Avi Wigderson. On span programs. In *Proceedings, Structure in Complexity Theory, Eighth Annual Conference*, pages 102–111, San Diego, CA, May 1993. IEEE.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[LMR88] F. Thomson Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 256–271, White Plains, NY, October 1988. IEEE.

[LS91] H. Li and Q.F. Stout. *Reconfigurable Massively Parallel Computers.* Prentice-Hall, Inc., 1991.

[Lup65] O. B. Lupanov. On the problem of realization of symmetric Boolean functions by contact schemes. *Probl. Kibernet*, 15:85–99, 1965. In Russian.

[Mac93] P. D. MacKenzie. A separation between reconfigurable mesh models. In *7th International Parallel Processing Symposium*, 1993.

[MNT90] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, Baltimore, MD, May 1990.

[MNV94] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. In *Proceedings of the Twenty-Sixthh Annual ACM Symposium on Theory of Computing*, pages 372–381, Montreal, Quebec, Canada, May 1994.

[MPKRS93] R. Miller, V. K. Prasanna Kumar, D. I. Reisis, and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42:678–692, 1993.

[MS89] R. Miller and Q. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, 38:321–340, 1989.

[Nak93] K. Nakano. An efficient algorithm for summing up binary values on a reconfigurable mesh. Research Report 93-003, Advanced Research Laboratory, Hatoyama, Saitama 350-03, Japan, January 1993.

[Neč66] E. Nečiporuk. On a Boolean function. *Soviet Math. Doklady*, 7:999–1000, 1966.

[Nis91] Noam Nisan. CREW PRAMs and decision trees. *SIAM Journal on Computing*, 20(6):999–1007, December 1991.

[NMT91] K. Nakano, T. Masuzawa, and N. Tokura. A sub-logarithmic time sorting algorithm on a reconfigurable array. *IEICE Transactions*, E74(11):3894–3901, 1991.

[PKR87] V.K. Prasanna Kumar and C.S. Raghavendra. Array processor with multiple broadcasting. *Journal of Parallel and Distributed Computing*, 4:173–190, 1987.

[PS88] Ian Parberry and George Schnitger. Parallel computation with threshold functions. *Journal of Computer and System Sciences*, 36(3):278–302, 1988.

[PSP93] B. Patt-Shamir and D. Peleg. Time-space tradeoffs for set operations. *Theoretical Computer Science*, 110:99–129, 1993.

[Pud84] Pavel Pudlák. A lower bound on the complexity of branching programs. In Michal P. Chytil and V. Koubek, editors, *Mathematical Foundations of Computer Science 1984: Proceedings, 11th Symposium*, volume 176 of *Lecture Notes in Computer Science*, pages 480–485, Praha, Czechoslovakia, September 1984. Springer-Verlag.

[PZ83] Pavel Pudlák and S. Zák. Space complexity of computations. Technical report, University of Prague, 1983.

[Ran87] A. G. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194, Los Angeles, CA, October 1987. IEEE.

[Raz87] A. A. Razborov. Lower bounds for the size of circuits with bounded depth with basis $\{\wedge, \oplus\}$. *Mat. Zametki*, 1987.

[Raz90] A. A. Razborov. Lower bounds on the size of switching-and-rectifier networks for symmetric boolean functions. *Mathematical Notes of the Academy of Sciences of the USSR*, 48(6):79–91, 1990.

[Raz91] A. A. Razborov. Lower bounds for deterministic and nondeterministic branching programs. In Lothar Budach, editor, *Eighth International Conference on Fundamentals of Computation Theory*, volume 529 of *Lecture Notes in Computer Science*, pages 47–60, Gosen, Germany, September 1991. Springer-Verlag.

[RBJ88] A. G. Ranade, Sandeep Bhatt, and S. L. Johnsson. The Fluent abstract machine. In *Proc. 5th M.I.T. Conference on Advanced Research in VLSI*, pages 71–94, 1988.

[RPK88] D. Reisis and V.K. Prasanna Kumar. VLSI arrays with reconfigurable buses. In *Supercomputing 1987*, volume 297 of *Lecture Notes in Computer Science*, pages 732–743. Springer-Verlag, 1988.

[RS62] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[Smo87] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 77–82, New York, NY, May 1987.

[Sni85] Marc Snir. On parallel searching. *SIAM Journal on Computing*, 14:688–708, 1985.

[ST94] R. Sinha and J. Thathachar. Efficient oblivious branching programs for threshold functions. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 309–317, Santa Fe, NM, November 1994. IEEE.

[Sto86] Q. F. Stout. Meshes with multiple buses. In *27th Annual Symposium on Foundations of Computer Science*, pages 264–273, Toronto, Ontario, October 1986. IEEE.

[SV84] Larry J. Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, May 1984.

[Tom78] Martin Tompa. *Time-Space Tradeoffs for Straight-Line and Branching Programs*. PhD thesis, University of Toronto, July 1978. Department of Computer Science Technical Report 122/78.

[vEB90] Peter van Emde Boas. Machine models and simulations. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 1–66. M.I.T. Press/Elsevier, 1990.

[VW85] Uzi Vishkin and Avi Wigderson. Trade-offs between depth and width in parallel computation. *SIAM Journal on Computing*, 14(2):303–314, May 1985.

[Weg87] Ingo Wegner. *The Complexity of Boolean Functions*. John Wiley & Sons, 1987.

[Yao79] A. C. Yao. Some complexity questions related to distributive computing. In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 209–213, Atlanta, GA, April-May 1979.

[Yao83] A. C. Yao. Lower bounds by probabilistic arguments. In *24th Annual Symposium on Foundations of Computer Science*, pages 420–428, Tucson, AZ, November 1983. IEEE.

[Yao85] A. C. Yao. Separating the polynomial hierarchy by oracles: Part I. In *26th Annual Symposium on Foundations of Computer Science*, pages 1–10, Portland, OR, October 1985. IEEE.

[Yao88] A. C. Yao. Near-optimal time-space tradeoff for element distinctness. In *29th Annual Symposium on Foundations of Computer Science*, pages 91–97, White Plains, NY, October 1988. IEEE. To appear in SIAM Journal on Computing.

Vita

Rakesh Kumar Sinha was born in Godda (India) in November, 1965. He attended the local high school, where his father taught, and later got his Bachelors degree from I. I. T. Kanpur, and his M.S. and Ph.D. degrees from University of Washington, Seattle. He joined the faculty at Florida International University, Miami, in Fall of 1995.