LIMITED NONDETERMINISM IN PARALLEL MODELS OF COMPUTATION

by

Marty J. Wolf

Computer Sciences Technical Report #946

July 1990

LIMITED NONDETERMINISM IN PARALLEL MODELS OF COMPUTATION

by

Marty J. Wolf

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

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN-MADISON
1990

TABLE OF CONTENTS

Abstract iv
Acknowledgements vi
Chapter 1. INTRODUCTION 1
1.1 Motivation and Overview 1
1.2 Definitions 4
Chapter 2. THE HISTORY OF BOUNDED NONDETERMINISM 7
2.1 Bounded Nondeterminism in Real-Time Computations 8
2.2 Relativizations with Bounded Nondeterminism 12
2.3 Bounded Nondeterminism in Other Models 16
2.3.1 Bounded Nondeterminism in Context-Free Languages 17
2.3.2 Bounded Nondeterminism in Finite Automata 20
2.4 Nondeterminism in Parallel Models 22
2.4.1 Nondeterministic PRAM's 23
2.4.2 Nondeterministic HMM's 25
2.4.3 Nondeterministic SC and Nondeterministic NC 29
Chapter 3. NONDETERMINISTIC NC 31
3.1 Motivation and Definitions 31
3.2 Relationships with Well Known Classes 33
Chapter 4. PROBLEMS IN NNC CLASSES 47
4.1 Generalizations of Problems in NC 48
4.2 Restrictions of NP-complete Problems 48
4.3 Quasigroup Isomorphism 51
Chapter 5. NONDETERMINISTIC AC 67
5.1 Definitions 67
5.2 Separation Results 69
Chapter 6. THE NNC HIERARCHY 75
Chapter 7. FUTURE WORK AND CONCLUSIONS 79
7.1 Summary 79
7.2 Future Work 80
7.2.1 Circuit Definitions for NSC 80
7.2.2 Refining the Complexity of NP-complete Problems 81

7.2.3 Other Potential Problems in NNC(polylog) 83
7.2.4 The Relationship Between RNC and NNC(polylog) 83
Chapter 8. BIBLIOGRAPHY 85

ABSTRACT

This thesis addresses issues surrounding nondeterminism in parallel computation. By quantifying and limiting the amount of nondeterminism available to various parallel computing devices, new complexity classes are developed that could potentially separate NC from NP. Let NNC(polylog) denote the class of languages accepted by an NC circuit family with a polylogarithmic number of nondeterministic gates. This class contains a version of the quasigroup (Latin square) isomorphism problem. We also show that NNC(polylog) $\subseteq$ DSPACE(polylog). This leads to a previously unknown space bound for the quasigroup isomorphism problem, namely that the quasigroup isomorphism problem, the Latin square isotopism problem and the Latin square graph isomorphism problem are all in DSPACE($\log^2 n$). The only other known bound for these problems is Miller's time bound of $n^{\log_2 n + O(1)}$. Our results generalize Lipton, Snyder and Zalcstein's DSPACE($\log^2 n$) algorithm for the group isomorphism problem.

Let NAC$^0(f(n))$ denote the class of languages accepted by an AC^0 circuit family with $f(n)$ nondeterministic gates on n bit inputs. We show that Parity is not accepted by NAC$^0(f(n))$ circuits when $f(n)$ is at most polynomial in $\log n$. Our results partially locate NAC$^0(f(n))$ within NP, namely, AC$^0 \subsetneq$ NAC$^0(n^\varepsilon) \subseteq$ NP for all $\varepsilon > 0$ and AC^0 $\subseteq$ NAC^0(polylog) $\subsetneq$ NAC$^0(n \log n) \subseteq$ NP.

A nondeterministic NC hierarchy is also developed through the use of bounded quantification. It shares many properties of the well-known polynomial time hierarchy. One difference, however, is the apparent lack of an equivalent definition in terms of oracles.

ACKNOWLEDGMENTS

Many people were instrumental in helping me see this thesis to completion. I thank Eric Bach for his patience and guidance. I also thank Anne Condon and Debby Joseph for their helpful comments throughout the development of the work presented in this thesis. I am grateful to Paul Terwilliger and Miron Livny for donating the time and energy required to serve on my final defense committee. I also owe a huge debt to Turhan Rahman. Without his encouragement, I might never have gone on to graduate school in computer science.

Many people have offered me support in other ways. I am deeply indebted to Linda, for all of her love, encouragement and support throughout the years. I am also grateful to Elise and Emily for helping me remember there is so much more to life than graduate school. Graduate school would have been a less fulfilling experience if it had not been for Bubber and the deep friendship we share. I thank all of the Hackers for the good times on and off the court. I also thank Mom and Dad, especially for having faith in me when they did not understand what I was doing or even what my goals were.

This work was supported in part by NSF grant DCR-8552596.

CHAPTER 1

INTRODUCTION

1.1. Motivation and Overview

Nondeterminism was first introduced in a computational setting in 1959. Rabin and Scott, in their seminal paper on finite state machines, noted that requiring all finite state machines to be described with deterministic transition functions leads to cumbersome descriptions for even some elementary operations [RS59]. Their motivation, therefore, for introducing nondeterminism was quite benign. They wanted to describe finite state machines as simply as possible, and since they showed that deterministic finite automata accept the same languages as nondeterministic ones, there was no harm in introducing the concept.

As time passed, however, the concept of nondeterminism began to take on special importance in computation theory. Many have undertaken research to try to answer the question, "Does nondeterminism help?" for various computing devices when considering different resources. There are three results that answer this question in slightly different ways. Savitch has shown that when considering space bounds on Turing machines, nondeterminism only helps a little. In other words, he proved that a deterministic Turing machine can simulate a nondeterministic Turing machine with at most

squaring the amount of space used [Sa]. (Definitions of unfamiliar terms can be found in Section 1.2.)

Paul, Pippenger, Szemerédi and Trotter showed that in the case of linear time on a Turing machine nondeterminism does help [PPST]. In other words, there is a language accepted by a nondeterministic multi-tape Turing machine in linear time that is not accepted by a deterministic multi-tape Turing machine in linear time, regardless of the number of tapes.

The other significant answer to the question of "Does nondeterminism help?" was for polynomially time-bounded Turing machines (the P =? NP question). Even though there is strong evidence that P is different than NP, Baker, Gill and Solovay gave strong technical evidence that answering the P =? NP question is going to be quite difficult and will require developing new proof techniques [BGS].

Many have chipped away at the P =? NP question, and much has been learned about the potentially rich structure of the problems and complexity classes lying between P and NP. Kintala developed a different approach to the P=?NP question by considering what happens when the nondeterministic Turing machine is allowed to make a bounded number of nondeterministic moves. His approach, however, failed to yield much insight into the P =? NP question. The notion of limited nondeterminism did provide some information about other types of computing devices such as real-time Turing machines. We will look at many of his results in Chapter 2, as well as the results of others' studies of limited nondeterminism.

Encouraged by some of the positive results of Kintala, we begin a study of restricted nondeterminism in circuit models of parallel computation. In Chapter 3 we formally define restricted nondeterministic parallel models of computation. We show that certain parallel models with the ability to make just a few nondeterministic choices accept some languages not known to be in P. In Chapter 4 we look at some of these languages. As a result of the study of the nondeterministic parallel complexity of the quasigroup or Latin square isomorphism problem, we develop previously unknown space bounds for the problem. (Throughout this thesis we use the terms "problem" and "language" interchangeably.) The space bound we develop matches that developed by Lipton, Snyder, and Zalcstein for the less general group isomorphism problem in [LSZ].

In Chapter 5 we look at constant depth, polynomial size, unbounded fan-in circuits, also known as AC^0 circuits. (AC^0 is formally defined in Section 1.2.) Unbounded fan in circuits of constant depth are one of the simplest parallel computing models. They are very weak; for example, they are unable to compute the Parity function. Because Parity cannot be computed with an AC^0 circuit, $AC^0 \subsetneq NP$. Furthermore, AC^0 is, in some sense, the most powerful complexity class known to be strictly contained in NP. We improve this result by showing that adding a restricted amount of nondeterminism to an AC^0 circuit does not allow it to compute the Parity function. We also give an upper bound on the least powerful nondeterministic AC^0 class that AC^0 is contained in.

In Chapter 6 we develop a restricted nondeterministic version of the polynomial time hierarchy. We show that our new hierarchy shares some properties of the polynomial time hierarchy, although not all.

Finally, in Chapter 7 we present some interesting open questions and suggest possible extensions of this work.

Before continuing we give definitions of many of the basic computing devices and complexity classes we will be referring to throughout this paper.

1.2. Definitions

Our Turing machine models are the usual ones, and we assume the reader is familiar with Turing machines as described in Chapters 7 and 12 of Hopcroft and Ullman [HU]. Briefly, a Turing machine is a finite state machine with some fixed number of infinite work tapes, one of which contains the input. For a deterministic machine, M, we say that a language L is accepted in time T(n) by M if and only if on all inputs w of length n, M makes no more than T(n) moves and halts in an accepting state if and only if $w \in L$. L is said to be in DTIME(T(n)). If M is nondeterministic, then L is said to be in NTIME(T(n)). When considering deterministic space bounds, we are often concerned with bounds that are less than linear. In this case we adopt the "random access input" Turing machine. This type of Turing machine has no head for the input tape. Instead, it has a special index tape on which it writes some number i in binary, which takes log i time, and whenever the machine enters a read state, the contents of the index tape is erased and replaced with the i^{th} bit of the input. If the Turing machine is to be

used to compute a function, it also has an output tape which upon which the result is written. This tape is not counted when determining space bounds. For a deterministic machine M, we say that a language L is accepted in space $S(n)$ by M if and only if on all inputs $w \in L$ of length n there is a valid computation leading to an accepting state that uses no more than $S(n)$ work tape cells. Then L is said to be in DSPACE($S(n)$).

We are also interested in other important complexity classes associated with the Turing machine. We let $P = \bigcup_{i \geq 1} DTIME(n^i)$ and $NP = \bigcup_{i \geq 1} NTIME(n^i)$. There are two space complexity classes of interest. They are $PSPACE = \bigcup_{i \geq 1} DSPACE(n^i)$ and DSPACE (polylog) $= \bigcup_{i \geq 1} DSPACE(\log^i n)$. (In general, we will use the term *polylog* to denote $\bigcup_{i \geq 1} \log^i n$.) We also mention a class with simultaneous resource bounds. A set is in the class SC if there is a Turing machine running simultaneously in polynomial time and polylog space that accepts it. To denote the complement of a class, we attach the prefix "co-" to the name of the class. For example, the complement of NP is denoted co-NP.

We also study deterministic parallel complexity classes developed from boolean circuits. A boolean circuit is a finite directed acyclic graph with labeled nodes (called gates). Each gates is labeled as either an input gate, a NOT gate, an AND gate, or an OR gate. The AND, OR, and NOT gates are allowed to get inputs from any other gate or their inputs may be forced to be either the constant 1 or the constant 0. One gate is distinguished as the output gate. The depth of the circuit is the length of the longest path between any input node and the output node. The size of a circuit is the number of

nodes in the circuit. Consider a family of circuits $C_0, C_1, \cdots$ where each C_n denotes a circuit with n inputs. Furthermore, suppose that a description of C_n can be generated by a deterministic Turing machine in DSPACE(log n). A description of C_n is a list of gates, that for each gate includes its type and which gates it gets its inputs from. This family of circuits is said to be LOGSPACE uniform. There are other, more restrictive types of uniformity discussed by Cook in [Co] and Ruzzo in [Ru81], but for the types of problems we consider, LOGSPACE uniformity suffices. A circuit family accepts a language if for each n there is a circuit C_n that accepts strings in the language of length n.

We distinguish whether or not the number of inputs to each gate is bounded by an independent constant. NC^k is defined as the set of languages accepted by LOGSPACE-uniform circuit families where each gate has a constant number of inputs, the depth bounded by $\log^k n$ and the size is bounded by n^j for some j on inputs of size n. We define NC to be $\bigcup_{k\geq 0} NC^k$. In the case that the number of inputs to each gate is unbounded we define AC^k as the set of languages accepted by a LOGSPACE-uniform circuit families with depth bounded by $\log^k n$ and size bounded by n^j for some j on inputs of length n. We define AC to be $\bigcup_{k\geq 0} AC^k$.

CHAPTER 2

THE HISTORY OF BOUNDED NONDETERMINISM

Traditionally, nondeterministic machines have been allowed to make a nondeterministic move at each step of a computation. Kintala and Fischer were the first to consider the effects of limiting the number of nondeterministic moves a machine makes. They observe in [KF77] that algorithms computing most NP-complete problems use only a linear number of nondeterministic moves even though the algorithm may make a polynomial number of steps. They contrast this with the fact that at that time the best known nondeterministic algorithm for recognizing primes required $O(n^2)$ nondeterministic moves even though the recognition of primes is not known to be NP-complete. (This was later improved to $O(n)$ nondeterministic moves by Pomerance [Po].) Motivated by this peculiar situation, Kintala in conjunction with others studied the effect of limiting the amount of nondeterminism available to various models of computation.

In the first three sections of this chapter we explore some of the results obtained by Kintala and his colleagues. First we present Kintala and Fischer's result that for real-time Turing machines there is an infinite proper hierarchy based on restricted amounts of nondeterminism. In Section 2 we present Kintala and Fischer's work on oracle Turing machines with a restricted number of nondeterministic moves, as well as

look at more recent work done by Alvarez, Diaz, and Toran that shows the existence of complete sets for the complexity classes introduced by Kintala and Fischer. In the third section we present some of the work done by Kintala and later by Nasyrov on restricted nondeterminism in context-free languages, as well as look at the economy of description offered by the presence of nondeterminism in finite state automata. In the fourth section we discuss work done by others in the area of applying nondeterminism, both limited and unlimited, to various parallel models of computation.

2.1. Bounded Nondeterminism in Real-Time Computations

Perhaps one of the most appealing results involving bounded nondeterminism was first presented by Kintala and Fischer in [KF77] and later in [FK]. They show the existence of an infinite proper hierarchy based on amount of nondeterminism available to a special class of Turing machines called real-time Turing machines.

A real-time Turing machine behaves like a standard Turing machine except that it is required to read a new input symbol at every step. Thus, a real-time machine never takes more than n steps to accept or reject an input of length n. This also implies the obvious upper bound on the number of nondeterministic moves the machine can make. Languages accepted by deterministic real-time Turing machines are called real-time languages and languages accepted by nondeterministic real-time Turing machines are called quasi-real-time languages.

In order to develop the infinite hierarchy, it is necessary to establish a notion of real-time constructible functions. Real-time constructible functions are defined in

terms of real-time countable functions, a notion introduced by Yamada [Ya62]. If $h(n) \geq n$ is a strictly monotone function, we associate an infinite binary string $\alpha^h = \alpha_0^h \alpha_1^h \alpha_2^h \cdots$ with the function h. To define α^h, let $\alpha_m^h = 1$ if $m = h(n)$ for some $n \geq 1$ and 0 otherwise. If a deterministic Turing machine can produce one bit of α^h at each step for a strictly monotone function $h(n) \geq n$ then $h(n)$ is *real-time countable*. $h^{-1}(n)$, the inverse of such an $h(n)$, is defined to be the largest m such that $h(m) \leq n$. A function $g(n) \leq n$ is *real-time constructible* if there exists a real-time countable function $h(n)$ such that $g(n) = h^{-1}(n)$. $Q_{g(n)}$ denotes the class of languages accepted by real-time multitape Turing machines making at most $g(n)$ nondeterministic moves.

Intuitively, let $h(n) \geq n$ be a real-time countable function and let $g(n) = h^{-1}(n)$. Then there is some Turing machine, M_h that produces one bit of α^h at every step. We modify M_h to yield a new machine, M_g, such that on inputs of length n, M_g generates a list of all of the positions of α^h that contain a 1. Thus, M_g computes the range of h up to the given input.

Let $g(n)$ be $o(n)$ and let y^R denote the reversal of y. We define the language $L_{g(n)}$. A string w is in $L_{g(n)}$ if the following hold:

(1) $|w| = n \geq 1$;
(2) $w = x_1 2 x_2 2 \cdots 2 x_r 3 y^R$ where $r = g(n) + 1$ and for all j, $x_j \in \{0,1\}^*$;
(3) For all i and j, $\big|\, |x_i| - |x_j| \,\big| \leq 1$;
(4) There is an s such that $y = x_s$.

Lemma 2.1.1: [FK] For any given real-time constructible $g(n)$, such that $g(n)$ is $o(n)$, $L_{g(n)} \in Q_{g(n)}$.

This result is not difficult to show since $g(n)$ is real-time constructible. The real-time Turing machine accepting $L_{g(n)}$ performs two actions concurrently. First, it verifies that the input is of the correct form, and secondly at every 2 in the input, it guesses whether to write down the following x_i. Thus, the machine makes only $g(n)$ guesses.

To demonstrate a proper, infinite nondeterminism hierarchy, Kintala and Fischer go on to show that if $2^{h(n)}$ is $o(g(n))$, then $Q_{h(n)} \subsetneq Q_{g(n)}$.

Lemma 2.1.2: [FK] If $g(n)$ and $h(n)$ are any two real-time constructible functions such that $g(n)\log(g(n))$ is $o(n)$ and $h(n)$ is $o(\log(g(n)))$, then $L_{g(n)} \notin Q_{h(n)}$, thus $Q_{h(n)} \subsetneq Q_{g(n)}$.

The proof of this lemma is quite technical and involves information counting arguments. Essentially, it relies on the fact that a real-time machine making only a few nondeterministic moves cannot distinguish between all of the possible configurations of a real-time machine making more nondeterministic moves.

Kintala and Fischer make an important observation about nondeterministic guesses. The next result demonstrates that careful analysis of the choices made by any nondeterministic Turing machine leads to an economy of guesses in the right situations.

Lemma 2.1.3: [FK] If $g(n)$ is any real-time constructible function with $g(n) = o(n)$, then $L_{g(n)} \in Q_{\log g(n)}$.

The important step in the proof of this lemma requires noting that in any correct computation all but one of the nondeterministic choices made by the machine in Lemma 2.1.1 are the same. In all but one instance the machine decides not to write down the subsequent x_i. We generalize this observation for nondeterministic Turing machines in the following lemma.

Lemma 2.1.4: Let $f(n)$ be some function that takes natural numbers to natural numbers. Let L be a language accepted by M, a nondeterministic machine that at every step has at most two possible choices. Also assume that for every input there exists an n such that in any correct computation M makes n nondeterministic choices with all but $f(n)$ of the choices the same. If $f(n)$ is $o(n/\log n)$, then there exists a Turing machine M' accepting L making fewer nondeterministic moves than M. Furthermore, M' makes only $f(n)\log n$ nondeterministic moves.

Proof: Instead of making the guesses that M does, M' guesses at which steps M makes nonstandard choices. Thus, M' begins its simulation of M by guessing at which steps M makes the nonstandard choices. Then it simulates M, and whenever M is supposed to make a nondeterministic move, M' looks up whether it should make the standard or nonstandard choice. Since there are $f(n)$ nonstandard choices, and to write each of the step numbers takes $O(\log n)$ nondeterministic moves, the total number of nondeterministic choices made by M' is $o(n)$. □

The proof of Lemma 2.1.3 is based on the fact that only one of the $g(n)$ guesses made by the machine in Lemma 2.1.1 is nonstandard. Thus, an application of Lemma 2.1.4 coupled with the fact that $g(n)$ is real-time constructible gives the general idea of the proof.

Let $\log^{(k)} n$ denote k applications of the log function to n. Then $Q_0 \subsetneq \cdots \subsetneq Q_{\log^{(k+1)} n} \subsetneq Q_{\log^{(k)} n} \subsetneq \cdots Q_{\log\log n} \subsetneq Q_{\log n}$ for sufficiently large n. This follows from Lemmas 2.1.2 and 2.1.3 and is stated more generally in the following theorem.

Theorem 2.1.5: [FK] If a real-time constructible function $g(n)$ is $o(\log n)$ and another real-time constructible function $h(n)$ is $o(g(n))$, then there exists a language L such that $L \in Q_{g(n)} - Q_{h(n)}$.

To prove this theorem, Kintala and Fischer show that $L = L_{2^{g(n)}}$ is sufficient.

2.2. Relativizations with Bounded Nondeterminism

Baker, Gill, and Solovay give convincing evidence that settling the P =? NP question will be difficult [BGS]. They show that there exist oracles that cause the classes P and NP to coincide and there exist oracles that cause P and NP to be distinct. In fact, they believe their results "suggest that the study of natural, specific decision problems offers a greater chance of success in showing P ≠ NP than constructions of a more general nature [BGS]." Even with this admonition, researchers have continued looking at general approaches to solving the P =? NP problem.

Kintala and Fischer developed a potential hierarchy of classes lying between P and NP [KF80]. This potential hierarchy is based on the number of nondeterministic moves available during computation. They proved results with respect to this potential hierarchy similar to those of Baker, Gill and Solovay. Later, Alvarez, Diaz, and Toran in [ADT] showed the existence of complete sets for some of the classes introduced in [KF80]. We will present some results from both Kintala and Fischer's paper and Alvarez, Diaz and Toran's paper and it will be apparent from the discussion that all of the classes lying between P and NP introduced by Kintala and Fischer have complete sets.

The classes studied by Kintala and Fischer are defined in terms of nondeterministic Turing machines. Let

$$P_{g(n)} = \{ L \mid L \subseteq \{0,1\}^* \text{ and } L \text{ is accepted by a polynomial-time bounded Turing machine making } O(g(n)) \text{ nondeterministic moves.}\}$$

The basic NP-complete problems given in Garey and Johnson are solvable with a linear number of nondeterministic moves. And since it seems that most other NP-complete problems are also solvable with a linear number of nondeterministic moves the limited nondeterminism classes of most interest are those with fewer than a linear number of nondeterministic moves. The specific classes studied by Kintala and Fischer are of the form $P_{\log^k n}$. Since enumerating all $2^{c \cdot \log n}$ possibilities of a machine making $O(\log n)$ nondeterministic moves results in only a polynomial increase in running time, we have $P = P_{\log n}$. Thus, determining that $P_{\log^k n} \subsetneq P_{\log^{k+1} n}$ for any $k \geq 1$ would imply that $P \neq$

NP. The relativized results obtained by Kintala and Fischer, like the results of Baker, Gill and Solovay, suggest that showing this proper containment (or equality) is difficult.

To show relativized results we present a notion of an oracle Turing machine. A Turing machine with oracle X is a Turing machine with an additional work tape called a query tape. The oracle Turing machine also has three special states, the query state, the no state and the yes state. Whenever the query state is entered the action of the machine at the next step is dictated by the oracle. The contents of the query tape are treated as a single string, and if that string is in the set X, the machine is placed in the ''yes'' state, otherwise it is placed in the ''no'' state. This model applies to both deterministic and nondeterministic Turing machines.

Thus, for any given function $g(n)$ and any oracle X,

$$P^X_{g(n)} = \{L \mid L \subseteq \{0,1\}^* \text{ and } L \text{ is accepted by a polynomial-time bounded oracle Turing machine with oracle } X \text{ making at most } O(g(n)) \text{ nondeterministic moves.}\}$$

A nondecreasing function $g(n)$ is *time-constructible* if there exists some constant n_0 such that for all $n > n_0$ either $g(n) \geq n$ and $g(n)$ is real-time countable or $g(n) < n$ and $g(n)$ is real-time constructible. Since Baker, Gill, and Solovay have demonstrated that any PSPACE-complete set A ensures $P^A = NP^A$, it follows that if $g(n)$ is a time-constructible function then any such A will cause $P^A_{g(n)} = P^A_{g(n)+n^\varepsilon}$ for all $\varepsilon \geq 0$. Using techniques similar to Baker, Gill and Solovay's, Kintala and Fischer showed that there is an oracle that separates two such classes as well.

Theorem 2.2.1: [Ki77] For any time-constructible function $g(n)$ and any $\varepsilon \geq 0$ there is a recursive oracle Bg depending on ε such that $P^{Bg}_{g(n)} \subsetneq P^{Bg}_{g(n)+n^{\varepsilon}}$.

Of course this theorem can be applied to the $P_{\log^k n}$ classes to obtain separation of the various levels. Perhaps one of the more interesting results in [KF80] indicates that there are questions about the $P_{\log^k n}$ hierarchy that are not equivalent to corresponding questions about P and NP.

Theorem 2.2.2: [KF80] For every $k \geq 2$ there is an oracle D_k such that

$$P^{D_k}_{\log^1 n} \subsetneq P^{D_k}_{\log^2 n} \subsetneq \cdots \subsetneq P^{D_k}_{\log^k n} = P^{D_k}_{\log^{k+1} n} = \cdots NP^{D_k}.$$

Alvarez, Diaz, and Toran present problems complete for the $P_{\log^k n}$ classes [ADT]. These problems are generalizations of P-complete problems. Consider, for example, this generalization of the circuit value problem. We define

$CVP^k = \{\langle x,y\rangle \mid x \in \{0,1\}^*$ and y encodes a boolean circuit with $|x| + \lceil \log^k |x| \rceil$ input gates and there is a string $x \in \{0,1\}^*$ of length $\lceil \log^k |x| \rceil$ such that the circuit encoded by y with input xz outputs a 1$\}$

CVP^k is obviously in $P_{\log^k n}$ and the proof of completeness is almost identical to the proof that the circuit value problem is hard for P. They also claim that similar modifications of other P-complete problems give more problems complete for $P_{\log^k n}$. Alvarez, Diaz, and Toran note that restrictions of NP-complete problems do not appear to be complete for $P_{\log^k n}$. In fact, there is some strong evidence that this is the case. Restrictions of NP-complete problems such as dominating set in a tournament and

CNF-SAT($\log^k n$) (for complete descriptions of these problems, see Section 4.2) all have the property that after some initial guessing phase, the verification phase can be done in NC^1. In other words, these problems are in $NNC^1(\log^k n)$ (defined in Section 3.1), a nondeterministic version of NC^1 with $\log^k n$ guessing inputs. Now $NNC^1(\log^k n) \subseteq P_{\log^k n}$ and we believe this containment is proper for the same reason NC^1 is believed to be in P. Thus, the observation of Alvarez, Diaz, and Toran follows.

2.3. Bounded Nondeterminism in Other Models

In addition to Turing machines, other models of computation such as pushdown automata and finite state machines have been used to study bounded nondeterminism. Kintala studied restricted nondeterminism in pushdown automata and expressed his results in terms of context-free languages, showing the existence of a proper infinite hierarchy of context-free languages based on the number of nondeterministic moves needed to accept them [Ki78]. In [Na], Nasyrov generalizes some of the work of Kintala and arrives at some of the same conclusions. However, Nasyrov's framework is general enough to provide a classification for all context-free languages and establish an additional infinite hierarchy within the context free languages.

When Rabin and Scott first introduced nondeterministic finite automata in [RS59], they showed that nondeterministic finite automata are no more powerful with respect to the languages they accept than are deterministic finite automata. The power set construction they use to prove this equivalence suggests, however, that there is an economy in the size of nondeterministic machines over deterministic machines that accept

the same languages. In fact, Meyer and Fischer have shown the existence of languages $L_k \subseteq \{0,1\}^*$ such that there is a k-state nondeterministic finite automaton that accepts L_k and the minimal deterministic finite automaton accepting L_k has at least 2^k states [MF]. Kintala and Wotschke explore this economy and show that in certain cases a nondeterministic finite automaton making fewer nondeterministic moves is somehow less succinct [KW]. We will review some of their results and observations in the second subsection of this section. First, however, we will look at the issue of restricting nondeterminism in context-free languages.

2.3.1. Bounded Nondeterminism in Context-Free Languages

Kintala observed that the obvious nondeterministic pushdown automaton accepting the context-free language $L_1 = \{ww^R \mid w \in \{0,1\}^*\}$, where w^R is the reversal of w, makes $n/2$ nondeterministic moves, while there is a nondeterministic pushdown automaton accepting the context-free language $L_2 = \{0^n1^m \mid n,m \geq 0; m = n \text{ or } m = 2n\}$ that makes only one nondeterministic move. It is known that both L_1 and L_2 are not deterministic context-free languages. Armed with this observation, Kintala establishes a proper infinite hierarchy of languages between the deterministic context-free languages and the context-free languages. First, Ogden's Pumping Lemma for context-free languages (see [HU]) is used to show the following theorem.

Theorem 2.3.1.1: [Ki78] Let $L_k = \{0^n1^{i \cdot n} \mid 1 \leq i \leq k\}$. L_k can be expressed as the union of k deterministic context-free languages, but cannot be expressed as the union of fewer than k deterministic context-free languages.

To establish the hierarchy Kintala notes that a nondeterministic pushdown automaton making a constant number of nondeterministic moves, where each move consists of a fixed number of choices, can be simulated by another nondeterministic pushdown automaton that makes one initial nondeterministic move to choose the appropriate state and then proceeds deterministically. If k-CFL denotes the class of languages accepted by such a nondeterministic pushdown automaton making an initial k-ary nondeterministic move, then $L_k \in k$-CFL. The following corollary to Theorem 2.3.1.1 is immediate.

Corollary 2.3.1.2: [Ki78] 1-CFL $\subsetneq$ 2-CFL $\subsetneq \cdots \subsetneq k$-CFL $\subsetneq$ CFL.

Thus, as with real-time machines, Kintala has established a hierarchy of context-free languages that is based on the amount of nondeterminism available to the corresponding machine.

Nasyrov generalized the notion of nondeterminism available to pushdown automata. This was done by considering deterministic versions of context-free languages. The deterministic version of a context-free language is defined by coding into the language the nondeterministic moves of a pushdown automaton that accepts it. Let $L \subseteq (\Sigma \cup \Delta)^*$. Let $Er_\Delta(L)$ denote the homomorphic image of L, where for all $\sigma \in \Sigma$, $Er_\Delta(\sigma) = \sigma$ and for all $\delta \in \Delta$, $Er_\Delta(\delta) = \lambda$, where λ is the empty word. The language we are interested in is $Er_\Delta(L)$, any context-free language with an alphabet Σ. L is a deterministic version of that language with characters from Δ representing the nondeterministic moves of the pushdown automaton that accepts $Er_\Delta(L)$ ($\Sigma \cap \Delta = \varnothing$). Any

context-free language is $M(n)$ *determinisable* if there exists a deterministic context-free language $L_d \subseteq (\Sigma \cup \Delta)^*$ such that 1) $Er_\Delta(L_d) = L$, and 2) for all $w \in L_d$, $|w| - |Er_\Delta(w)| \leq M(|Er_\Delta(w)|)$. Essentially, 1) says that if you remove the nondeterministic moves from L_d you are left with L, and 2) says there is a bound on the number of nondeterministic moves the machine can make.

Using this notion of determinism, Nasyrov partitions all of the context-free languages into three types.

Type 1: Context-free languages L such that in the deterministic version L_d there is only one nondeterministic marker at the beginning of each word.

Type 2: Context-free languages L such that in the deterministic version L_d there there are a constant number of nondeterministic markers in each word.

Type 3: Context-free languages L such that in the deterministic version L_d there are a linear number of nondeterministic markers in each word.

The Type 1 context-free languages are the same as those studied by Kintala. In fact, Nasyrov goes on to show the same result as Kintala did for that class of languages. Nasyrov also shows that a similar hierarchy exists for the Type 2 context-free languages.

Theorem 2.3.1.3: [Na] If for all integers $k > 0$, F_k denotes the class of all $M(n)=k$ determinisable context-free languages, then $F_k \subsetneq F_{k+1}$.

Nasyrov also shows the existence of context-free languages of Type 3 that are not of Type 2. In addition, he shows that no nondeterminism hierarchy exists for context-free languages of Type 3.

Theorem 2.3.1.4: [Na] For each context-free language L and each $\varepsilon > 0$ there exists a pushdown automaton, A, such that $L(A) = L$ and A is $M(n) = c \cdot n$ determinisable for all $c \leq \varepsilon$.

This result is not too surprising however. Having a linear number of nondeterministic markers in a word is equivalent to a pushdown automaton having an exponential number of possible computations. Thus, even though there are rich nondeterminism hierarchies for context-free languages that use only a little nondeterminism, context-free languages that require a linear number of nondeterministic moves for acceptance are all equivalent in some sense.

2.3.2. Bounded Nondeterminism in Finite Automata

In [KW], Kintala and Wotschke demonstrate that there is a "nondeterminism" hierarchy associated with finite automata as well. Their techniques and results parallel some of the techniques and results Kintala and Fischer showed in relation to real-time Turing machines. Consider any two functions $r(k)$ and $s(k)$ such that $r(k)$ is $o(\log k)$ and $s(k)$ is $o(r(k))$. Kintala and Wotschke show that for any two such functions there exists sequences of languages, $L_r = R_1, \cdots, R_n, \cdots$ and $L_s = S_1, \cdots, S_n, \cdots$ such that the nondeterministic finite automaton (NFA) accepting the languages of L_r make at most $r(k)$ nondeterministic moves and the NFA accepting the languages of L_s make at

most $s(k)$ nondeterministic moves, where k is the number of states in the automaton. Thus, the NFA for L_r are considerably more succinct relative to the minimal deterministic finite automaton (DFA) that accept languages in L_r than are the NFA for L_s relative to the minimal DFA that accept the languages of L_s. Here succinctness is measured as the ratio of the number of states of the nondeterministic automaton to the number of states of the minimal deterministic automaton for a given language.

Let $L_{h,k} = \{x\,1y \mid x,y \in \{0,1\}^*; |x| \leq k-1; |y| = k;\ x \text{ has at most } h \text{ 1's in it}\}$. It is this class of languages that gives the desired succinctness hierarchy. The following lemma gives lower bounds on the number of states of the DFA accepting $L_{h,k}$, as well as upper bounds on the number of nondeterministic moves made by certain NFA.

Theorem 2.3.2.1: [KW] Any DFA accepting $L_{h,k}$ must have at least $\sum_{i=0}^{h} \binom{k}{i}$ distinct states. Also, there is an $O(k^2)$-state NFA accepting $L_{h,k}$ making at most $\log h$ nondeterministic moves on any input.

Thus, by choosing $r(k)$ to be $o(\log k)$ and by picking a sufficiently large k, there is an $h = 2^{r(k)} < k$. Using this k and h, we pick $L_{h,k}$. By choosing larger values for k we obtain the sequence of languages L_r. Now any function $s(k)$, such that $s(k)$ is $o(r(k))$, yields the sequence L_s, and we have a hierarchy based on the economy of description offered by the presence of nondeterministic states.

2.4. Nondeterminism in Parallel Models

The idea of nondeterministic parallel computation is not new. Fortune and Wyllie were among the first to study a parallel model of computation with the ability to make nondeterministic choices [FW]. They studied parallel random access machines (PRAMs), defined in Section 2.4.1, and showed that $O(\log n)$ time on a nondeterministic PRAM is equivalent to NP. They also showed that any problem in NP can be sped up exponentially using nondeterministic parallelism. They point out that if such exponential speedups are not possible by applying deterministic parallelism to problems in P, one also shows that P $\neq$ DSPACE(polylog).

Dymond studied the complexity classes obtained by giving two different versions of nondeterministic hardware modification machines (NHMMs), defined in Section 2.4.2, the ability to make nondeterministic choices [Dy]. Nondeterministic hardware modification machines are a weaker model than nondeterministic PRAMs. However, depending on how an NHMM is allowed to make nondeterministic choices, Dymond showed that nondeterministic HMMs obtain the same sort of exponential speed-up over sequential time that Fortune and Wyllie showed for nondeterministic PRAMs. Dymond also suggests definitions for nondeterministic NC (NNC) and nondeterministic SC (NSC). He goes on to show, however, that his NNC is exactly the same as NP.

Moriya, Iwata, and Kasai show an intimate relationship between NSC and NNC defined in terms of a more traditional computational device--the Turing machine. In terms of Turing machines, NC is defined to be the class of sets accepted by Turing

machines using polynomial time and a polylog number of read/write head reversals. Moriya, Iwata, and Kasai show that a one tape NSC machine is equivalent in power to a one tape NNC machine [MIK].

2.4.1. Nondeterministic PRAM's

Fortune and Wyllie define a PRAM to be an unbounded set of synchronous numbered processors combined with an unbounded global memory, a set of input registers and a finite program [FW]. Each of the processors has an accumulator, unbounded local memory, and a program counter. A flag is associated with each processor that indicates if it is running. The program consists of a number of (labeled) instructions, where an instruction is one of LOAD, STORE, ADD, SUBTRACT, JUMP, JUMPONZERO, READ, FORK, or HALT. The JUMP and JUMPONZERO operate on a label and change the program counter to the associated label. A PRAM behaves nondeterministically if two instructions have the same label. The FORK instruction allows a processor to start another processor. The new processor's memory is cleared and the contents of the executing processor's accumulator is copied into the new processor's accumulator. The new processor is started at the indicated label. The HALT instruction causes the processor to halt. The PRAM halts whenever two processors try to simultaneously write to the same memory location or when the first processor executes the HALT instruction. In the first case the PRAM rejects and in the second case the PRAM accepts only if there is a 1 in the first processor's accumulator.

The three main results of [FW] are given here without proof. The first one is essentially the basis for the Parallel Computation Thesis which states that time on a parallel machine is equivalent to space on a deterministic machine. The second one demonstrates the exponential speedup obtained by nondeterministic parallelism. Crucial to the proofs of both of these theorems is the fact that in t steps a PRAM can start up to 2^t processors.

Theorem 2.4.1.1 [FW] A language L is accepted in polylog time on a deterministic PRAM if and only if L is in DSPACE(polylog) and a language L' is accepted in polynomial time on a deterministic PRAM if and only if L' is in PSPACE.

Theorem 2.4.1.2 [FW] A language L is accepted in $O(\log n)$ time on a nondeterministic PRAM if and only if L is in NP.

In the proof of these theorems, the PRAMs seemingly communicate an exponential amount of information through the global memory. However, Fortune and Wyllie go on to prove that only a polynomial number of memory locations are needed to show Theorem 2.4.1.2. Thus, restricting the amount of global storage does not yield a new and interesting complexity class. However, in the deterministic case, restricting the amount of global storage seems to limit what the PRAM can accept in polynomial time, as they gave the following partial classification.

Theorem 2.4.1.3 [FW] Co-NP is contained in the class of sets accepted by deterministic polynomial time-bounded, polynomial global storage-bounded PRAMs.

Conversely, the best known upper bound for the class of sets accepted by polynomial time-bounded, polynomial global storage-bounded deterministic PRAMs is PSPACE.

The proof of Theorem 2.4.1.3 takes advantage of the fact that if two processors try to write to the same global storage location the PRAM halts and rejects. If a PRAM tries to accept the complement of an NP-complete problem such as Satisfiability, in linear time it starts an exponential number of processors, each of which tries a unique assignment of values to the variables of the given boolean formula. Since the PRAM will halt and reject if two or more processors try to write to the same location, the PRAM accepts sets in co-NP. If the PRAM does not halt due to conflicting simultaneous writes, the first processor inspects the global storage location and if something is written there, it rejects, otherwise it accepts.

2.4.2. Nondeterministic HMM's

In the study of nondeterministic parallel computing, Dymond has noted that for most parallel models either the deterministic version of the machine can simulate the nondeterministic with only a polynomial loss in time or the nondeterministic parallel machine can achieve an exponential speed-up over a nondeterministic sequential computation [Dy]. Dymond shows how two different versions of the Hardware Modification Machine (HMM) can be used to obtain both types of results mentioned above, thus offering a model that provides insight into why two such differing results generally hold.

A HMM is an infinite collection of synchronous finite state transducers, a finite number of which are active at any given time. Each accepts k inputs, and based on those inputs and its state, it computes an output symbol, enters a new state, and possibly changes the origin of its inputs. In the deterministic case all units have the same transition function. As mentioned, each unit can change the source of its inputs. The lines through which a unit gets its inputs are called "taps," and in each step a unit can redirect any of its taps to a unit that it either already is tapping or a unit that one of its neighbors is tapping. More precisely, unit U_i can direct a tap to a unit U_j if U_i already has a tap to U_j or there exists another unit, U_k, such that U_i has a tap to U_k and U_k has a tap to U_j.

Initially, only one unit, U_0, is active. In one step, any active unit can activate another unit by directing one of its taps to the inactive unit and specifying the initial state and initial locations of the taps of the new unit. The taps of the new unit must be placed on a unit within one of the activating unit.

Finally, a HMM has "random" access to the input through a special fixed binary tree structure of inactive units. This allows a unit to obtain any bit of the input in $O(\log n)$ steps. Initially, U_0 has one of its taps tapping the root of the input tree. The HMM accepts an input if U_0 ever enters a halting state.

Dymond defines two different nondeterministic HMM's. The first is called mono-NHMM since U_0 is the only unit allowed to make nondeterministic moves. These nondeterministic choices are then broadcast to other units through taps. The

other nondeterministic HMM is called multi-NHMM since all of the units are allowed to make independent nondeterministic moves.

An HMM, mono-NHMM, or multi-NHMM accepts a given input in time t if U_0 enters a halting state before it has made t transitions. Similarly, any of the machines accepts a given input in hardware h if U_0 enters a halting state without there ever having been more than h units active. An HMM, H, accepts a language $L \subseteq \{0,1\}^*$ in time $T(n)$ (in hardware $H(n)$) if for every w, $|w| = n$, H enters a halting state in time $T(n)$ (respectively, in hardware $H(n)$) and accepts if and only if $w \in L$. This definition can be applied to mono-nondeterministic hardware modification machines and multi-nondeterministic hardware modification machines by the appropriate choice of the machine H. Thus, when describing complexity classes associated with the various types of hardware modification machines, we list the type of the machine, the resources we are considering and then the bound on those resources. For example, mono-NHMM-HARDWARE-TIME(n^2, log n) is the class of sets accepted by a mono-nondeterministic hardware modification machine in hardware n^2 and in log n time.

Dymond shows that for most time bounded mono-NHMM's the addition of nondeterminism does not allow the machine to accept more than in the deterministic case. He also shows that a multi-NHMM is exponentially faster than a nondeterministic Turing machine. These results are expressed formally in the next two theorems.

Theorem 2.4.2.1 [Dy] For all $T(n) \in \Omega(\log n)$, HMM-TIME(T) = mono-NHMM-TIME(T).

Theorem 2.4.2.2 [Dy] For all $T(n) \in \Omega(\log n)$, $\bigcup_{k>0}$ NTIME(k^T) = multi-NHMM-TIME(T).

In the proof of Theorem 2.4.2.1, the amount of hardware grows exponentially. Essentially the deterministic HMM makes an exponential number of copies of the nondeterministic HMM and each copy simulates behavior on one of the possible guesses. The proof of Theorem 2.4.2.2 is similar to our proof in the next chapter that NNC(poly) = NP.

Dymond also considers classes with simultaneous resource bounds. He gives equivalent definitions of the classes NC and SC in terms of HMM's, and then defines nondeterministic versions of these classes with multi-NHMM's.

Definitions:

$$\text{NNC} = \bigcup_{k>0} \text{multi-NHMM-HARDWARE,TIME}(n^k, \log^k n).$$

$$\text{NSC} = \bigcup_{k>0} \text{multi-NHMM-HARDWARE,TIME}(\log^k n, n^k).$$

It is not difficult to show that NNC = NP. The key fact is that the NHMM can make a polynomial number of nondeterministic choices, giving it the ability to guess the entire computation of an NP machine. It may seem reasonable to consider the classes obtained by replacing the multi-nondeterministic hardware modification machines in the above definitions by mono-nondeterministic hardware modification machines. Dymond observes that the class NSC is the same regardless of the choice of

machines. This suggests the inherent upper bound on the number of useful nondeterministic moves made by a device is directly related to its ability to write them down or keep track of them in some manner. Similarly, with PRAMs, a nondeterministic choice is of little value, unless it is communicated to all of the computing elements. The class ''mono-NNC'' seems to be weaker than ''multi-NNC''. In fact, in Chapter 3 we will show that mono-NNC is the same as NNC(polylog), a class we define in terms of nondeterministic NC circuits.

2.4.3. Nondeterministic SC and Nondeterministic NC

Moriya, Iwata, and Kasai consider the relationship between the nondeterministic versions of SC and NC. Both SC and NC can be defined in terms of deterministic multi-tape Turing machines with different simultaneous resource bounds. SC is obtained by limiting the Turing machine to polylog space and polynomial time. NC is obtained by limiting the Turing machine to polylog read/write head reversals and polynomial time. Nondeterministic versions of these classes are defined using nondeterministic Turing machines. It is well-known and easy to establish that nondeterministic SC (NSC) is contained in nondeterministic NC (NNC), although the containment is not known to be proper. In [MIK], Moriya, Iwata, and Kasai show that if both classes are restricted to one work tape, they become equivalent. We let the subscript denote the number of work tapes available to the Turing machine.

Theorem 2.4.3.1 [MIK] $NNC_1 = NSC_1$.

This result is further improved by an observation made by Parberry in [Pa]. He notes that by a well known result, which states that any k-tape Turing machine can be simulated by a 1-tape Turing machine (see Theorem 7.2 of [HU]), all of NSC is contained in NSC_1. Thus, $NSC = NNC_1$.

CHAPTER 3

NONDETERMINISTIC NC

Fortune and Wyllie [FW] and Dymond [Dy] both show that for different models nondeterministic NC and NP are the same. Both models, however, allow a supply of nondeterministic choices that is limited only by the running time of the given device. Our idea is to limit the number of nondeterministic choices a priori. We use the circuit model as a natural means to apply Kintala's idea of limiting nondeterminism and develop new and interesting complexity classes.

3.1. Motivation and Definitions

By giving NC circuits a limited amount of nondeterminism, we develop potentially interesting complexity classes between NC and NP as well as demonstrate their relationship to well known DSPACE classes. Nondeterministic NC (NNC) circuits are developed by uniformly adding "*guessing gates*" to families of LOGSPACE uniform NC circuits. We define NNC($f(n)$) to be the class of sets accepted by LOGSPACE uniform families of NC circuits with at most $O(f(n))$ nondeterministic gates or guess gates, where n is the length of the input. We will refer to a circuit from such a family as a uniform NNC circuit, and say that such a circuit accepts an input if it outputs a 1.

The guessing gates give the circuit a set of guessing inputs, y, in addition to the ordinary inputs, x. An NNC circuit is said to accept x if and only if there is some string of guessing bits y that causes the circuit to output a 1. Thus, $f(n)$ can be thought of as the maximum number of guess bits used in computations on inputs of length n. The classes we study here are of the form NNC($\log^k n$) and NNC(n^k). We will often abuse notation and write NNC(*class*) where *class* is a class of functions. For example, we define NNC(polylog) = $\bigcup_{k\geq 1}$ NNC($\log^k n$) and NNC(poly) = $\bigcup_{k\geq 1}$ NNC(n^k). We can refine NNC classes to include an index that indicates the depth of the circuit. For example, we let NNC^k($\log^j n$) denote the class of languages accepted by an NC^k circuit with $O(\log^j n)$ guessing gates. This definition is consistent with the notation in Cook [Co] in that the exponent of NC indicates the depth of the circuit.

Note that NNC complexity classes share an intimate relationship with RNC, a random version of NC. RNC circuits and NNC circuits can be thought of as computing in exactly the same manner, but with different acceptance criteria. An NNC circuit accepts if and only if there is at least one string of guess bits that causes the circuit to output a 1, whereas an RNC circuit accepts if and only if at least, say, ¾ of the possible strings of random (or guess) bits cause the circuit to output a 1. Unfortunately, RNC circuits may use a polynomial number of random bits, thus, we can only say that RNC $\subseteq$ NNC(poly). This is not surprising since in the next section we will show that NNC(poly) = NP. On the other hand, we know that any problem that has an RNC algorithm using $O(f(n))$ random bits is in NNC(f(n)).

One of our first goals is to determine how much nondeterminism is needed by an NC circuit in order for the associated complexity class to be new and "interesting." In the next section we will see that NNC(polylog) lies between NC and NP with neither containment known to be proper. We also characterize NNC(polylog) in terms of Dymond's nondeterministic hardware modification machines. In addition, we show that certain NNC classes are contained in certain DSPACE classes.

3.2. Relationships with Well Known Classes

We would like to find a class of functions C such that $NC \subsetneq NNC(C) \subsetneq NP$. We begin by considering the class NNC(poly). We will show that this class is too powerful since NNC(poly) = NP. Fortune and Wyllie [FW] obtained a similar result using a nondeterministic PRAM model for nondeterministic NC, as did Dymond [Dy] using a multi-NHMM model for nondeterministic NC. The advantage of the NC circuit model has over these previous models is the straightforward manner in which the amount of nondeterminism used in computations can be quantified. Thus, next we consider the function $\log n$ and find that NNC($\log n$) is too weak, since NNC($\log n$) = NC. The most interesting case is when C is the class of functions that are bounded by polynomials in $\log n$. In Chapter 4 we show a form of quasigroup isomorphism is in NNC(polylog). Since this problem is not known to be in P and this problem is not known to be NP-complete, the class NNC(polylog) joins P, RP, ZPP and RNC as a candidate for a class to separate NC and NP.

We begin by considering the class NNC(poly). It is not hard to see that NNC(poly) $\subseteq$ NP.

Lemma 3.2.1: NNC(poly) $\subseteq$ NP.

Proof. Assume the set A is in NNC(poly). Then given x, an input of length n, it is easy to construct an NP-machine to simulate the NNC circuit. First the NP machine computes a description of the NNC circuit that accepts words in A of length n, then guesses the outputs of the guess gates, and then since there are only a polynomial number of gates in the circuit, the NP machine can compute the output of each gate including the output gate in polynomial time. □

The next goal is to show NP $\subseteq$ NNC(poly), which implies NNC(poly) = NP. To show this we consider a nondeterministic one-tape Turing machine, M, that accepts a set in NP and construct an NNC(poly) circuit that accepts inputs of a given length. For a given input, the circuit "guesses" an instantaneous descriptor (ID) for each move of an accepting computation of M on that input. An ID is a string of bits representing the contents of the tape, the position of the tape head (written in binary), and the current state of the machine M. Then for each pair of adjacent ID's a small circuit verifies that the second follows from the first via a legal move of M, based on the head position and the state of M. Furthermore, if each pair of adjacent ID's represent legal moves and the state of the last ID is an accepting state the circuit accepts, otherwise it rejects.

Theorem 3.2.2: NP = NNC(poly).

Proof. Lemma 3.2.1 shows one direction of the theorem. We now give more details of the argument outlined above.

Let M be a nondeterministic polynomial time bounded Turing machine accepting a language L in NP. Without loss of generality, we assume M has only one tape and that every computation on inputs of length n takes $q(n)$ steps, where q is some polynomial. Now consider a computation of M on input x of length n to be a table of instantaneous descriptors, where each ID represents the string currently on the work tape, the position on the tape of the read/write head written in binary, and the state of the machine (also in binary). Also, for the table to represent an accepting computation, ID_1 represents the initial state of the Turing machine, ID_i must follow from ID_{i-1} via a transition rule of M, and the last ID, $ID_{q(n)}$, must represent a final state of M if and only if $x \in L$.

To simulate the computation of M on x, C_n first guesses every ID of an accepting computation of M. There are $O(q(n)^2)$ bits to guess since the length of each ID is $O(q(n))$ and there are $q(n)$ ID's. Then for each pair of adjacent ID's there is a small circuit for testing whether ID_i follows from ID_{i-1} via a legal move of M. If ID_1 is correct, each ID follows from the previous one, and the state of $ID_{q(n)}$ is a final state, then the circuit accepts, otherwise it rejects.

The circuit used to verify that adjacent ID's represent legal moves of M is easy to construct as well. In parallel, it verifies that all the bits have remained the same from ID_{i-1} to ID_i except those in the area of the read/write head. In the area of the

read/write head, a constant size circuit verifies that the move from ID_{i-1} to ID_i is a legal move of M by looking it up in a table.

Verifying ID_1 represents the initial state of M and verifying the state of $ID_{q(n)}$ is a final state are straightforward.

Since these circuits can be uniformly constructed, we find that for each set in NP there is a uniform family of NNC(poly) circuits that accepts it. □

Since NNC(poly) = NP, we need to consider guesses shorter than polynomial ones to find a new and interesting nondeterministic version of NC. As the next theorem indicates, allowing log n guess bits does not allow the circuit to compute anything not in NC.

Theorem 3.2.3: $\mathrm{NNC}^k(\log n) = \mathrm{NC}^k$.

Proof. It is obvious that $\mathrm{NC}^k \subseteq \mathrm{NNC}^k(\log n)$. We now prove the other direction. Since there are only $2^{O(\log n)}$ (which is at most a polynomial in n) possible different guesses that can be made by the guessing gates, the circuit enumerates all of the possible guesses, and with duplicate copies of the NC circuit computes in parallel on each possible guess, accepting if at least one of the copies accepts. The circuit is still of polynomial size (although considerably larger), and the depth of the circuit is increased only by an additional $O(\log n)$. □

Since $O(\log n)$ guess bits are too weak and polynomially many guess bits are too powerful, we settle on the class NNC(polylog) as potentially interesting, as it is easy to see that $\mathrm{NC} \subseteq \mathrm{NNC}(\mathrm{polylog}) \subseteq \mathrm{NP}$. We fall short of our goal, though, since none of

the containments are known to be proper.

Next we completely characterize NNC(polylog) in terms of Dymond's nondeterministic hardware modification machines [Dy] discussed in Section 2.4.3. Dymond introduces the class mono-NNC = $\cup_{k>0}$ mono-NHMM-HARDWARE,TIME(n^k, $\log^k n$) as a class that "appears to fall properly between NP and NC" [Dy]. By characterizing mono-NNC in terms of nondeterministic circuits we establish that the mono-NHMM's ability to modify its interconnection pattern does not allow it to take undue advantage of the nondeterministic output of the one nondeterministic finite state transducer. This lends some evidence to Dymond's statement that mono-NNC may lie properly between NC and NP. An NC machine makes no nondeterministic moves, a mono-NNC machine makes a polylog number, and an NP machine makes a polynomial number. First we show how a mono-NHMM efficiently simulates a nondeterministic circuit.

Lemma 3.2.4:

$NNC^k(\log^j n) \subseteq$ mono-NHMM-HARDWARE,TIME(poly, $\log^{\max\{k,j\}} n$).

Proof. First we make the following simplifying assumption about the NNC circuit. We will assume that the circuit consists of levels numbered 0, 1, 2 $\cdots$, t such that level 0 contains all of the input and nondeterministic gates, the output gate is the only gate at level t, and gates at any other level will receive inputs only from gates on the previous level.

Let $L \in \mathrm{NNC}^k(\log^j n)$. We will describe a mono-NHMM, M, that accepts L. The idea of the proof is that for inputs of length n, M deterministically builds a copy of the circuit C_n that accepts inputs of length n and then simulates the circuit. There will be a unit in M corresponding to each gate in the circuit. In addition, the nondeterministic unit of M will supply a nondeterministic input to each unit representing one of the nondeterministic gates of the circuit. As a matter of convenience we will refer to a unit that represents a specific gate simply by mentioning the gate.

Each level of the circuit in M will have a coordinator. The coordinator for level i initiates construction of a binary tree that allows units from level $i+1$ to access the output of gates at level i. This binary tree is similar in construction to the one that allows access to the inputs of M, with the coordinator situated at the root of the tree. This tree contains taps in both directions: from parent to child and from child to parent, thus, allowing gates on level i to move their input taps through the coordinator out to level $i-1$, and allowing gates in level $i+1$ to move their input taps to the gates at level i.

M begins its simulation by having its one active unit start another unit that will represent the output node of the circuit which is at level t. Since there is only one node at this level, this node acts as its own coordinator. In the next step, the coordinator for level $t-1$ of the circuit is started by the output node. The output node attaches a tap to the coordinator of level $t-1$, so that later it can attach its input taps to the correct gates at level $t-1$.

Once a coordinator for any level i is started, it then takes on two tasks. First it starts the coordinator at level $i-1$, then it begins building a binary tree such that the leaf nodes of the tree will represent gates in the circuit. The tree is built in the manner described by Dymond in [Dy]. Once the tree is built, the gates at level $i+1$ move their input taps through the coordinator at level $i+1$ to the coordinator at level i and then down through the tree to the appropriate gates. This process continues until construction of the entire circuit is completed, and then the HMM begins its simulation of the circuit.

Dymond shows how to build a binary tree of depth d in $O(d)$ time in [Dy]. Thus, the tree at each level can be built in $O(\log n)$ time. Since the trees are all built simultaneously, the dominating cost is starting the $O(\log^k n)$ coordinators. After the circuit is built and before the simulation can begin the nondeterministic unit must generate a nondeterministic input for each of the $O(\log^j n)$ nondeterministic gates. Since the time for the simulation of the circuit is $O(\log^k n)$, the total time taken is going to be the maximum of this value and the cost of generating the nondeterministic bits. □

The other direction involves a simulation that is not quite as efficient. There is a $\log n$ factor slow down.

Lemma 3.2.5: Mono-NHMM-HARDWARE,TIME(poly, $\log^k n$) $\subseteq$ NNC^{k+1}($\log^k n$).

Proof: Let $L \in$ mono-NHMM-HARDWARE,TIME(poly,$\log^k n$), and let M be a mono-NHMM that accepts L. We will show how to construct C_n, an NNC^{k+1}($\log^k n$) circuit that determines membership in L for words of length n. In this simulation, for

each finite state transducer of M we have one subcircuit that mimics its behavior. Essentially the subcircuit will take as input a description of the transducer, which includes the internal state and the identity of the units its taps are receiving input from, and then broadcast its output and move to the new state.

The circuit that simulates M has a subcircuit, S_i, for each unit U_i started by M. Each S_i is divided into levels, one level corresponding to each step made by U_i. As the computation proceeds, each unit of M is either active or idle. Our circuit ignores output generated by subcircuits representing idle units. If U_i is active however, at each level S_i receives as input the new state of the unit, the location of its taps and the output of all of the units from the previous step. After determining which inputs to discard and which to keep, S_i looks up in the transition table what state it will move to and where each tap will be attached for the next step. If U_i activates another unit U_j, S_i initializes the state and tap information of S_j. Once S_j is activated S_i broadcasts its output to the subcircuits on the next level, and it passes along state and tap information to the next level of subcircuit that will compute the next step for S_i.

There will be one subcircuit S_0 computing for the nondeterministic unit. S_0 may be faced with more than one possible next move. Since the number of possible moves from any state is bounded by a constant, using a constant number of nondeterministic bits, S_0 guesses which move to make, otherwise the behavior of this subcircuit is the same.

Since S_0 may have to make a nondeterministic choice at every level, the circuit needs $O(\log^k n)$ guessing gates. Unneeded guesses are ignored. It takes constant time to look up a move. Each level of a subcircuit must sift through a polynomial number of outputs from the previous step, thus, deciding which values are needed for its taps takes $O(\log n)$ time. Since this is done once on each of $O(\log^k n)$ steps taken by M the total depth of C_n is $O(\log^{k+1} n)$. □

Theorem 3.2.6: NNC(polylog) = mono-NHMM-HARDWARE,TIME(poly,polylog).

Proof: This follows directly from the definition of NNC(polylog) and Lemmas 3.2.4 and 3.2.5. □

It is interesting to note that Theorem 3.2.6 does not seem to generalize to other classes. For example, $\text{NNC}^k(n^\varepsilon) \subseteq$ mono-NHMM-HARDWARE,TIME(poly, n^ε) for $\varepsilon<1$. However, it is not clear that mono-NHMM-HARDWARE,TIME(poly, n^ε) is contained in $\text{NNC}^k(n^\delta)$ for any k and for some $\varepsilon, \delta<1$. The first containment follows since n^ε is much larger than $\log^k n$ and thus, initializing the nondeterministic gates becomes the dominating cost. The strongest statement we are able to make about a reverse containment is the following.

Lemma 3.2.7: mono-NHMM-HARDWARE,TIME(n^δ, n^ε) $\subseteq \text{NNC}^1(n^\gamma)$ where $\delta,\varepsilon<1$ and $\gamma=\delta+\varepsilon$.

Proof: The proof of this lemma differs somewhat from the proof of Lemma 3.2.5. It is similar, though, to the proof of Theorem 3.2.2.

Let $L \in$ mono-NHMM-HARDWARE,TIME(n^{δ},n^{ε}) be accepted by the mono-NHMM machine M. We will build a circuit C_n that accepts words in L of length n. Since M has a sublinear number of units and each makes a sublinear number of moves, C_n can simply guess the state and the values output by each unit at every step of the computation. Then, in a manner similar to Theorem 3.2.2, C_n can verify in $O(\log n)$ depth that all of the moves are consistent.

The amount of information guessed for one step of one unit's computation is constant. For each unit $O(n^{\varepsilon})$ guesses are made. Thus, since there are $O(n^{\delta})$ units, $\gamma=\delta+\varepsilon$ suffices. □

Next we establish a relationship between NNC classes and DSPACE classes that is instrumental in our result of the next chapter that the quasigroup isomorphism question is decidable in DSPACE($\log^2 n$). Lemma 3.2.8 shows that NNC(polylog) circuits that are deep and require little guessing and NNC(polylog) circuits that are shallow and require much guessing can both be simulated by a deterministic polylog-space bounded Turing machine. By virtue of Theorem 3.2.6, this result also shows that mono-NHMM-HARDWARE,TIME(poly,polylog) is in DSPACE(polylog), an observation not made in [Dy]. Using the more refined definitions of NNC complexity classes, this result is stated more formally as the following lemma.

Lemma 3.2.8: For all $k,j \geq 1$, $\text{NNC}^k(\log^j n) \subseteq \text{DSPACE}(\log^m n)$, where $m = \max\{k,j\}$.

Proof. The deterministic Turing machine that simulates the NNC circuit begins by counting the number, N, of nondeterministic gates in the circuit. The Turing

machine then writes the lexicographically first bit string, B, of length N on a work tape. Using a technique presented by Borodin [Bo] the Turing machine simulates the circuit by recursively evaluating first the left input and then the right input of a given gate, starting with the output gate. (I.e, the Turing machine does a depth first search of the circuit starting at the output gate.) At any time the Turing machine need only keep track of the status of each gate (which input is being computed and the value of the other input if it is known) on a path from the output gate to an input gate or a nondeterministic gate. If the name of a gate is ever needed, it can be recomputed using the status of each gate on the path starting with the output gate. Note that we store only a constant amount of information for each gate on the path and that we have at most one path active at any time. We continue this recursive procedure until a nondeterministic gate is encountered, at which point the computation is temporarily stopped. The name of this gate is written on a work tape, then M counts the number of gates, k, that precede it in the circuit description. Next M retrieves bit $k+1$ of B and uses it as the output of the current nondeterministic gate. If the circuit accepts, the Turing machine accepts, otherwise the Turing machine increments B and repeats the process until all the bit strings of length N have been tested.

For $k \geq j$, the depth of the circuit is larger than the number of nondeterministic bits, therefore the Turing machine is given enough space to simulate the circuit. For $j \geq k$, the bit string requires more space than does the simulation of the circuit, so the Turing machine is given enough space to write down the bit string. In both cases, the

Turing machine has enough space to complete both tasks. □

Corollary 3.2.9 is an obvious consequence of Lemma 3.2.8 since NNC(polylog) = $\bigcup_{k\geq 1}\bigcup_{j\geq 1}$ NNC^k(log^j n).

Corollary 3.2.9: NNC(polylog) $\subseteq$ DSPACE(polylog).

It would be desirable to show that either NNC(polylog) = DSPACE(polylog) or that NNC(polylog) $\subsetneq$ DSPACE(polylog). With this goal in mind, we present the following lemma which begins to explore similar relationships between NNC(poly) and DSPACE(poly).

Lemma 3.2.10: For all k, NNC(n^k) $\subseteq$ DSPACE(n^k).

Proof. Since n^k grows much faster than any polylog function, use the construction of Lemma 3.2.8. □

As in the polylog case it would be desirable to show that for some k, DSPACE(n^k) $\subseteq$ NNC(n^k). However, this containment is unlikely as the next lemma and theorem indicate.

Lemma 3.2.11: If there exist integers k and j such that DSPACE(n^k) $\subseteq$ NNC(n^j) then for all r there is an integer s such that DSPACE(n^r) $\subseteq$ NNC(n^s).

Proof. The proof of this claim is via the familiar translation technique (see [HU]).

Let L_1 be any language in DSPACE(n^r), and let M_1 be an n^r space-bounded Turing machine accepting L_1. Let # be a symbol not in the alphabet of L_1, and let L_2 = $\{x\#^i \mid M_1$ accepts x using $(|x| + i)^k$ space$\}$. Thus, we construct a Turing machine M_2

such that on input $x\#^i$, M_2 marks off $(|x| + i)^k$ tape cells and then simulates M_1. M_2 accepts $x\#^i$ if and only if M_1 accepts x, and it does so using $(|x| + i)^k$ tape cells. Thus, L_2 is in DSPACE(n^k), and by hypothesis L_2 is in NNC(n^j). Let C_2 denote the NNC(n^j) circuit that accepts the input $x\#^i \in L_2$.

Next we describe an NNC(n^s) circuit, C_1, that accepts words in L_1 of length n. On input x of length n, C_1 guesses an i. Next C_1 simulates C_2 on $x\#^i$, and C_1 accepts if and only if C_2 accepts. Since $|i|$ is $O(n^r)$ and C_2 guesses $O((n+i)^j) = O(n^{r \cdot j})$ bits, we let $s = r \cdot j$. Also note that the depth of C_1 is no more than that of C_2 and that the number of gates in C_2 (and thus, C_1) is polynomial in $|x|^r$ and thus, polynomial in $|x|$. Therefore we have an s such that DSPACE(n^r) $\subseteq$ NNC(n^s). □

Theorem 3.2.12 presents strong evidence that the containment in Lemma 3.2.10 is proper.

Theorem 3.2.12: If there exist k and j such that DSPACE(n^k) $\subseteq$ NNC(n^j) then PSPACE = NP.

Proof. From Theorem 3.2.2 we know that NP = NNC(poly) = $\bigcup_{k \geq 1}$ NNC(n^k), and by definition PSPACE = $\bigcup_{k \geq 1}$ DSPACE(n^k). Since Lemma 3.2.11 under the same hypothesis as this theorem states that for all r there is an s such that DSPACE(n^r) $\subseteq$ NNC(n^s), we have PSPACE $\subseteq$ NNC(poly) = NP, giving the desired result. □

Theorem 3.2.12 provides some information about whether NNC(n^k) is properly contained in any DSPACE(n^j) class. It would be desirable to find similar information

concerning the containment of NNC($\log^k n$) classes in DSPACE($\log^j n$) classes. Doing so, however, seems unlikely since DSPACE($\log n$) $\subseteq$ NC = NNC($\log n$). Thus, to show a result such as "If there exist k and j such that DSPACE($\log^k n$) $\subseteq$ NNC($\log^j n$) then an unlikely collapse" would require developing a proof technique that forces the unlikely collapse for $k \geq 2$ but does not force the collapse for $k = 1$. Unfortunately, even the need for such a peculiar proof technique gives us little insight into whether or not NNC(polylog) is properly contained in DSPACE(polylog).

CHAPTER 4

PROBLEMS IN NNC CLASSES

In this chapter we show that NNC(polylog) contains some nontrivial and interesting problems. These problems are all related in that the running time of their best known sequential algorithms is $n^{\log_2 n + O(1)}$. These algorithms have this time bound because they enumerate the $O(n^{\log n})$ possible solutions and then determine if any are correct. We take advantage of two traits shared by all of these problems. First, they all have some short (polylog) guess that will lead to a correct solution, and second, for each problem it can be determined quickly in parallel if a guess leads to the correct solution.

We present three types of problems. The first group contains problems artificially contrived from problems in NC. The second group consists of problems that are restricted versions of NP-complete problems. And the final group contains the group and quasigroup isomorphism problems and some related problems. The restricted NP-complete problems seem to be easier to decide because the verification phase can be done in NC^1. This seems to be the case for all restrictions of NP-complete problems where the restriction shrinks the solution search space since, as seen earlier, $NP \subseteq NNC^1(poly)$. The problems related to the group isomorphism problem, however, all seem to need an NC^2 verifier, and the verifier for a problem contrived from a

problem in NC depends on where in the NC hierarchy the problem lies.

4.1. Generalizations of Problems in NC

Adapting the approach of Alveraz, Diaz and Toran used to develop complete problems for $P_{\log^k n}$ [ADT], we add a ''guess'' part to a language in NC, thus, keeping it in NNC(polylog) while making it seem likely that it is not in NC. This is a straightforward exercise, but it is useful since it demonstrates that the NNC classes are not sparsely populated and furthermore suggests possible candidates for complete problems. We will look at one such extension here. Let

$$\mathrm{CFL}^k = \{\langle G,w\rangle \mid G \text{ is context free grammar and } w \in \Sigma^*, \text{ where } \Sigma \text{ is the terminal alphabet for } G, \text{ and there a string } x \text{ such that } |x| \leq \lceil \log^k |w| \rceil \text{ and the string } wx \text{ is generated by } G\}$$

Theorem 4.1.1: CFL^k is in $\mathrm{NNC}^2(\log^k n)$.

Proof: This follows since context free language recognition is in NC^2 [Ru80], and the string x can be guessed by the nondeterministic gates. □

This technique can be applied to languages in NC^k to obtain other languages that are not likely to be in NC^k, but still are in NNC^k(polylog).

4.2. Restrictions of NP-complete Problems

In this section we show that NNC^1(polylog) contains natural nontrivial problems. The first problem is a restriction of the maximum clique problem. It involves finding a dominating set in a tournament.

Definition: A *tournament* is a directed graph $T=(V,E)$ where every two distinct vertices $u,v \in V$ are connected by exactly one edge, that is, $|E \cap \{ (u,v),(v,u) \}| = 1$. If the edge is (u,v) then we say that u dominates v.

A dominating set, S, in a tournament is a set such that every node in V is either in S or is dominated by a node in S. The following fact is attributed to Erdös by Megiddo and Vishkin in [MV].

Fact: If T is a tournament with $n \geq 2$ vertices, then the size of the minimum dominating set in T is $\leq \lceil \log_2 n \rceil$.

Proof. Let $d(v)$ denote the outdegree of v. Clearly $\sum_{v \in V} d(v) = \frac{n(n-1)}{2}$. By the pigeon hole principle there must be at least one node that dominates $\lceil \frac{n-1}{2} \rceil$ nodes. Place this node in the dominating set. Now remove that node and all of the nodes it dominates from T. The resulting tournament has no more than $\frac{n}{2}$ nodes. This procedure can be applied at most $\lceil \log_2 n \rceil$ times, generating a dominating set of the appropriate size. □

Thus, finding a dominating set in a tournament is in $\mathrm{NNC}^1(\log^2 n)$.

Theorem 4.2.1: Given a tournament T and an integer k, the set $\{<T,k> \mid T$ has a dominating set of size $\leq k\}$ is in $\mathrm{NNC}^1(\log^2 n)$.

Proof. We will construct an $\mathrm{NNC}^1(\log^2 n)$ circuit that accepts inputs of length n in the set. It begins by guessing a dominating set. Once the dominating set is in hand,

there is a small subcircuit for each vertex in the tournament that verifies that either the vertex is in the dominating set or one of the nodes that dominates it is in the dominating set. This can be done easily with an NC^1 circuit since equality can be tested in NC^1. Now since the size of the smallest dominating set is no more than $\lceil \log_2 n \rceil$ and the length of the name of each vertex in the tournament is $O(\log n)$, the circuit only guesses $O(\log^2 n)$ bits of information. □

Megiddo and Vishkin show that by enumerating all of the possible dominating sets it is easy to decide in $n^{O(\log n)}$ sequential time whether or not a tournament has a dominating set of size less than or equal to k [MV]. No polynomial time algorithm is known to exist for this problem.

Megiddo and Vishkin also study a restricted form of the satisfiability problem. This version, defined next, also has a best known sequential running time of $n^{O(\log n)}$.

Definition: Let C be a fixed constant. A satisfiable formula ϕ is said to be in CNF-SAT($\log^k n$) if ϕ is in conjunctive normal form and the number of variables in the formula is bounded by $C \cdot \log^k n$, where n is the number of clauses.

Theorem 4.2.2: For every $k \geq 1$, CNF-SAT($\log^k n$) is in $NNC^1(\log^k n)$.

Proof: Again we construct a circuit to accept inputs of length n. Since there are $O(\log^k n)$ variables, the circuit guesses a value for each. Then for each clause a subcircuit verifies that at least one variable is true, and a final AND gate verifies that each clause has at least one true variable. Since the subcircuits essentially test equality they can easily be implemented in $O(\log n)$ depth. □

4.3. Quasigroup Isomorphism

Since Miller [Mi] has shown the quasigroup (Latin square) isomorphism problem has an $n^{\log_2 n + O(1)}$ sequential algorithm, and since $n^{\log_2 n} = 2^{\log_2^2 n}$ this problem is a natural candidate for being in NNC($\log^2 n$). Lipton, Snyder and Zalcstein [LSZ] have a DSPACE($\log^2 n$) algorithm for group isomorphism, but, as Miller [Mi] points out, their technique relies on the associativity of groups and does not readily generalize to quasigroups. Miller goes on to present a more general technique requiring $n^{\log_2 n + O(1)}$ time for the quasigroup isomorphism problem. In this section we show that quasigroup isomorphism, Latin square isotopism and Latin square graph isomorphism can all be decided in $\text{NNC}^2(\log^2 n)$. As a result of the relationship between NNC(polylog) and DSPACE(polylog) demonstrated in the previous chapter, we develop an DSPACE($\log^2 n$) algorithm for the quasigroup (Latin square) isomorphism problem. This is a previously unknown space bound for this problem (see [Mi]). We also show that problems closely related to the quasigroup isomorphism problem are also in $\text{NNC}^2(\log^2 n)$, and, thus, in DSPACE($\log^2 n$). For review we give the definitions of Latin squares, groups and quasigroups.

Definition: A *Latin square* is an $n \times n$ grid with each of the integers $1, 2, \cdots, n$ appearing exactly once in each row and column.

If each of the integers $1, 2, \cdots, n$ appears as a label for exactly one row and exactly one column then the Latin square can be viewed as a multiplication table of a quasigroup. We formalize the definitions of groups and quasigroups by considering the

following four properties of a set Q with an associated binary operation *.

For all $a, b, c \in Q$:

1. There is a unique x such that $a*b = x$.
2. There is a unique x such that $a*x = b$.
3. There is a unique x such that $x*a = b$.
4. $(a*b)*c = a*(b*c)$.

Definition: Q is a *quasigroup* if * satisfies properties 1, 2 and 3.

Definition: Q is a *group* if * satisfies properties 1, 2, 3 and 4.

Thus, a quasigroup is more general than a group. In this section we view a quasigroup of order n as a binary function on $\{1, 2, \cdots, n\}$, so that the corresponding multiplication table is a Latin square.

Viewing Latin squares L and L' as trinary relations $<,,>$ and $<,,>'$, L is *isomorphic* to L' if there exists a permutation σ such that if $<x,y,z> \in L$ then $<\sigma(x),\sigma(y),\sigma(z)>' \in L'$. Two quasigroups are isomorphic if their corresponding Latin squares are isomorphic. A more general notion of an isomorphism is an isotopism. L is *isotopic* to L' if there exist permutations α, β, γ such that if $<x,y,z> \in L$ then $<\alpha(x),\beta(y),\gamma(z)>' \in L'$. Thus, an isomorphism simultaneously interchanges rows, columns and values in L to get L', and an isotopism independently interchanges rows, columns and values in L to get L'. Miller [Mi] showed that quasigroups of order n are generated by at most $\lfloor \log_2 n \rfloor$ elements, and we take advantage of this fact to develop the

circuit for quasigroup isomorphism testing, Latin square isotopism testing, and Latin square graph isomorphism testing. Since quasigroups are more general than groups, our construction also shows group isomorphism is also in $NNC^2(\log^2 n)$. Note that we assume that quasigroups are presented as their corresponding Cayley (multiplication) tables.

Before getting to the main results of this section we develop a useful representation for elements of quasigroups in terms of generators. A generating set is a subset of the quasigroup such that every element in the quasigroup can be expressed as a product of elements from that set. Since quasigroups are not necessarily associative, when an element $q \in Q$ is written as the product of generators from a particular generating set, it must be fully parenthesized, for example, $q = (((g_1 * g_2) * (g_3 * g_2)) * g_3)$. We associate with $q \in Q$ the parse trees of all such expressions with internal nodes representing multiplications and leaves representing generators. If q is in the generating set, then q is represented by a tree consisting of a single node labeled q. If q is not in the generating set and $q = p * r$, then q is represented by a tree where the root represents the multiplication between the element represented by the left subtree (p) and the element represented by the right subtree (r). This definition gives infinitely many representations for each element in Q, necessarily including some very large representations. We will show that every element in the quasigroup has a tree of small depth[1] that

[1] The depth of a node is the number of edges between it and the root. The depth of a tree is the maximum depth of a node over all nodes in the tree.

represents it, then the large representations will not hinder us.

Before showing that each element has a shallow representation we develop some notation, establish some facts about quasigroups and their associated multiplication tables, and state some assumptions. Let $depth(q)$ be the depth of the shallowest tree representing $q \in Q$. Note that the depth an element in the generating set is 0. Since $Q = \{1, 2, \cdots, n\}$ we assume that the elements are listed so that their depths are nondecreasing. Namely, for all $i, j \in Q$, if $i \leq j$ then $depth(i) \leq depth(j)$. Let M denote the multiplication table for Q, and let M_k denote the upper left hand $k \times k$ portion of Q.

There are two important facts to keep in mind about quasigroups with these properties.

Fact 1: If r is the maximum element of depth d, then no element s such that $depth(s) \geq d+2$ appears in M_r.

Fact 2: If $depth(n) = d$ then for every depth between 1 and d there is at least one element of that depth.

Fact 1 is true because if such an s did appear in M_r the $depth(s) \leq d+1$. Fact 2 holds since an element of depth $i > 0$ is the product of two elements at least one of which has depth $i - 1$.

We introduce the following notation to help show that for certain elements there is another element about twice as large that is only slightly deeper. Let $b_1, b_2, \cdots, b_k$ be the elements of depth d and $c_1, c_2, \cdots, c_m$ be the elements of depth $d+1$ arranged in increasing order. As a notational convenience, we let M_b refer to M_{b_k} and M_c refer

to M_{c_m}. We also refer to elements up to b_k as "small" elements and elements strictly larger than b_k as "big" elements.

For a fixed b_1 we use Diagram 1 to help clarify our argument.

We will show by some counting arguments that if there is an element l with $depth(l) \leq depth(b_1)+2$ then l is about twice the size of b_1. In order to show the

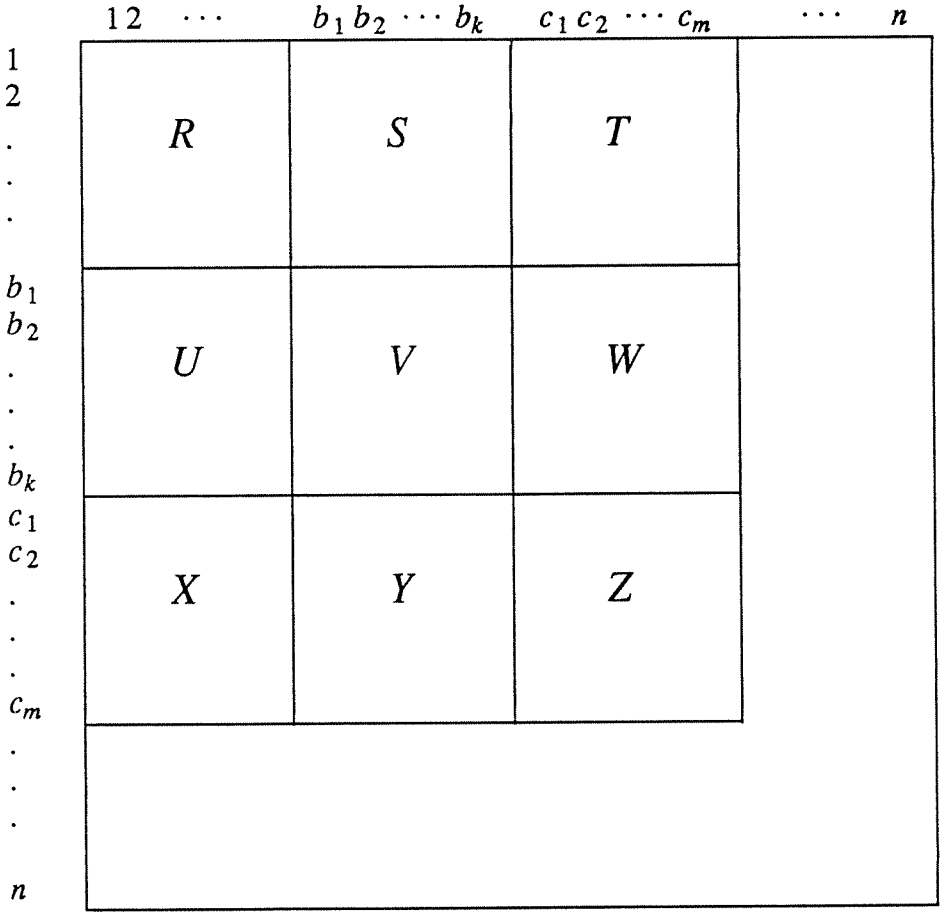

Diagram 1. Subtables of M.

existence of such an l, we show there exists a column in $T \cup W \cup Z$ with many big elements. This happens only when there are a few small elements in $T \cup W \cup Z$. It is easy to give upper bounds on the number of small elements in W and Z. In order to show that only a few small elements occur in T we prove that only a few big elements appear in S. By Fact 1, we know that only the elements $1, 2, \cdots, c_m$ can appear in $R \cup S$. Furthermore, $c_1, c_2, \cdots, c_m$ cannot appear in R. To establish upper bounds on the number of small elements in T, we show that there is at most one small element in T for each big element in S. We then prove an upper bound on the number of big elements in S to obtain an upper bound on the number of small elements in T.

The following lemma establishes an upper bound on the number of small elements that occur in $T \cup W \cup Z$.

Lemma 4.3.1: If the number of big elements in S is at most $m \cdot k$. Then there are at most $2m \cdot k + m^2$ small elements in $T \cup W \cup Z$.

Proof. First note that W is a $k \times m$ subtable. Thus W has at most $m \cdot k$ small elements. Similarly, Z is an $m \times m$ subtable and has at most m^2 small elements.

Next we consider the rows of $R \cup S \cup T$. By Fact 1 all of the values in M_b are no larger than c_m. If $R \cup S$ were filled entirely with small values then none of the elements of T would be small. For each small element that does not appear in $R \cup S$ there must be some c_i in S. (Recall that by Fact 1 no c_i can appear in R.) Thus, each big element in S causes at most one element in T to be small. Since there are at most $m \cdot k$ big elements in S, there are at most $m \cdot k$ small elements in T. Thus, the total number of

small elements in $T \cup W \cup Z$ is no more than $2m \cdot k + m^2$. □

The next lemma shows the existence of an l that is not too much deeper than b_1, yet it is about twice as large as b_1.

Lemma 4.3.2: If there exists a column of $T \cup W \cup Z$ that contains at most $2k+m$ small elements then there exists an l such that $l \geq 2b_1 - 2$ and $depth(l) \leq d+2$.

Proof. Each column of $T \cup W \cup Z$ has c_m elements. Assume that column c_j of $T \cup W \cup Z$ has at most $2k+m$ small elements. Since $T \cup W \cup Z$ is part of the multiplication table for a quasigroup, each of the elements in column c_j must be unique. The remaining $c_m - 2k - m$ slots must be filled with elements of the quasigroup whose values are at least c_1. By simply counting we find that we are forced to place an $l \geq c_1 + c_m - 2k - m - 1$ into column c_j in $T \cup W \cup Z$. Note that $c_i = b_1 + k + i - 1$. Thus $l \geq 2b_1 - 2$. Furthermore, since l appears in M_c, $depth(l) \leq depth(c_j) + 1 = d + 2$. □

Now we prove a lemma which implies that the depth of each element does not grow too rapidly relative to its value.

Lemma 4.3.3: For all $q \in Q$ such that $q \geq 3$ and $depth(q) > depth(q-1)$ either $2q - 2 > n$ or there exists an $l \in Q$ such that $depth(l) \leq depth(q)+2$ and $l \geq 2q - 2$.

Proof: Pick any $q \geq 3$ such that $depth(q) > depth(q-1)$ and $2q - 2 \leq n$. Let $b_1 = q$, and let $depth(b_1) = d$. If $k + m \geq b_1 - 3$ then $depth(k+m+1) \leq d+2$. So assume that $k + m < b_1 - 3$. We will use the argument outlined above. We want to

show there is a column in $T \cup W \cup Z$ with only a few small elements. First we show that there are at most $m \cdot k$ big elements in S. By Fact 1, M_b contains only elements up to c_m, thus the big elements in S are between c_1 and c_m. Since each c_j can appear at most once in each column, a total of at most $m \cdot k$ big elements appear in S.

Now by an application of Lemma 4.3.1, we know that there are at most $2m \cdot k + m^2$ small elements in $T \cup W \cup Z$. Since there are m columns in $T \cup W \cup Z$, there is at least one column that has no more than $2k + m$ small elements. An application of Lemma 4.3.2 gives the desired result. □

Unfortunately Lemma 4.3.3 does not apply to all elements in Q. Next we build a series of elements a_i with the property that $depth(a_i) = depth(a_{i-1}) + 1$ and $depth(a_0) = 1$. Each a_i corresponds to a different b_1. If G is the generating set for Q that is being used to build the trees and $|G| = g$, then Fact 2 tells us that $a_0 \geq g + 1$, $a_1 \geq g + 2$, and $a_2 \geq g + 3$, and by Lemma 4.3.3 $a_k \geq 2a_{k-3} - 1$. Next we establish a lower bound for a_k.

Lemma 4.3.4: Let $a_0 \geq g + 1$, $a_1 \geq g + 2$, $a_2 \geq g + 3$, and $a_k \geq 2a_{k-3} - 1$ for $k \geq 3$. Then for $k \geq 3$ and $k \equiv 0 \pmod 3$, $a_k \geq g \cdot 2^{k/3} + 1$, $a_{k+1} \geq (g+1) \cdot 2^{k/3} + 1$, and $a_{k+2} \geq (g+2) \cdot 2^{k/3} + 1$.

Proof: The proof is by induction on k. For the base case let $k = 3$.

$$a_3 \geq 2a_0 - 1 \geq 2g + 1 = g \cdot 2^1 + 1.$$

$$a_4 \geq 2a_1 - 1 \geq 2g + 3 = (g+1) \cdot 2^1 + 1.$$

$$a_5 \geq 2a_2 - 1 \geq 2g + 5 = (g+2)\cdot 2^1 + 1.$$

For the inductive step assume the lemma holds for all $k < l$ and that $l \equiv 0 \pmod 3$. By assumption $a_{l-3} \geq g\cdot 2^{(l-3)/3} + 1$. Since $a_l \geq 2a_{l-3} - 1 \geq 2(g\cdot 2^{(l-3)/3} + 1)$ - 1 = $g\cdot 2^{l/3} + 1$ we have the desired result for a_l. A similar construction works for a_{l+1} and a_{l+2}. □

Next we show explicit upper bounds on the depths of the a_k.

Lemma 4.3.5: If the a_k are as in Lemma 4.3.4 then $depth\,(a_k) \leq 3\lceil \log_2 a_k \rceil + 1$.

Proof: We prove the result for $k \equiv 0 \pmod 3$. The proofs for the other two cases are similar.

Since $depth\,(a_k) = depth\,(a_{k-1}) + 1$ and $depth\,(a_0) = 1$, $depth\,(a_k) = k + 1$. Thus

$$\begin{aligned} \lceil \log_2 a_k \rceil + 1 &= 3\lceil \log_2 (g\cdot 2^{k/3}) + 1 \rceil + 1 \\ &\geq 3\lceil \log_2 (2^{k/3}) \rceil + 1 \\ &\geq k + 1 \\ &= depth\,(a_k). \end{aligned}$$

□

We now come to the main result about representing elements of quasigroups in terms of the generators.

Theorem 4.3.6: Let Q be a quasigroup of order $n \geq 4$ with $G = \{g_1, g_2, \cdots, g_k\}$ generating Q. Assume that the elements of Q are ordered so that their depths are nondecreasing. Then for the generating set G $depth\,(n) \leq 3\lceil \log_2 n \rceil + 1$.

Proof: We consider the series of a_i described above. Let $a_{\max}$ be the largest element of this series. Obviously, $a_{\max} \leq n$. Also, since $a_{\max}$ is the largest element that has depth exactly one more than its predecessor, $depth(n) = depth(a_{\max})$. By Lemma 4.3.5, $depth(n) = depth(a_{\max}) \leq 3\lceil \log_2 a_{\max} \rceil + 1 \leq 3\lceil \log_2 n \rceil + 1$. □

With this shallow tree representation of elements of a quasigroup, we now can show that quasigroup isomorphism can be accepted with an $NNC^2(\log^2 n)$ circuit.

Theorem 4.3.6: Given two multiplication tables M_1 and M_2, representing the quasigroups Q_1 and Q_2, respectively, the set $\{(Q_1, Q_2) \mid Q_1$ is isomorphic to $Q_2\}$ is in $NNC^2(\log^2 n)$, where n is the order of the two quasigroups.

Proof. We first give a general overview of the circuit that tests for the isomorphism, and later we give a more detailed construction. The circuit to test the isomorphism begins by guessing two sets of generators G_1 for Q_1 and G_2 for Q_2 in parallel. We will assume that G_1 and G_2 are ordered in some manner and that order determines the isomorphism, i.e., the i^{th} element of G_1 is mapped to the i^{th} element of G_2. Next in parallel the circuit verifies that G_1 generates Q_1, G_2 generates Q_2, and that the mapping guessed is an isomorphism. Verifying G_i generates Q_i involves two general steps. First, elements that are known to be generated by the generators are marked as being in the quasigroup, with the guessed generators marked initially. Second, in parallel each marked element of the quasigroup is multiplied by every marked element of the quasigroup, generating more elements known to be in the quasigroup. To verify the guessed mapping is an isomorphism the circuit performs two tasks in parallel.

First, in parallel for each pair of elements g, $h \in G_1$, the circuit tests whether g and h are the same, and if so, it verifies that their images in G_2 are identical. Second, in parallel for each pair of elements g, $h \in G_2$, the circuit tests whether g and h are the same, and if so, it verifies that their preimages in G_1 are identical.

We now provide more details of the isomorphism testing circuit. Miller [Mi] has shown that a quasigroup of size n has at most $\log_2 n$ generators, and we note that each generator is $\log_2 n$ bits long, thus, the circuit has $2 \cdot \log_2{}^2 n$ guessing gates, half of which are used to guess G_1, the set of generators of Q_1, and the rest are used to guess G_2, the set of generators of Q_2.

To verify that G_1 does generate Q_1 we use the following subcircuit with $3\lceil \log_2 n \rceil + 1$ levels. An identical circuit will verify G_2 generates Q_2. Each level of the subcircuit corresponds to a copy of the multiplication table of Q_1, and the i^{th} level, $level_i$, computes all the elements of Q_1 that can be expressed as the product of at most 2^i elements of G_1. In order to do this, $level_i$ receives from $level_{i-1}$ all of the elements of Q_1 that can be expressed as the product of at most 2^{i-1} generators, and then in parallel multiplies each of those elements by every element it received from the previous level by looking up the product in the input multiplication table. Instead of receiving inputs from the previous level, $level_1$ receives inputs from the guessing gates, as the generators are the only elements of Q_1 that can be expressed as the product of at most one element of G_1. After the final level, a check is made to insure that all of the elements of Q_1 have been generated.

From Theorem 4.3.6 we know that each element in a quasigroup of order n has a tree with depth no more than $3\cdot\lceil\log_2 n\rceil + 1$ representing it. Thus, after $3\cdot\lceil\log_2 n\rceil + 1$ levels either all of the elements are generated or G_i is not a generating set for Q_i, and therefore there are only $O(\log n)$ levels in the circuits that verify G_i generates Q_i. At each level, there are at most n^2 multiplications taking place, with each multiplication requiring an $O(\log n)$ depth circuit with polynomial size. Since there are $O(\log n)$ levels, we have an overall depth of $O(\log^2 n)$, and the overall circuit size is polynomial.

To verify that the guessed mapping is an isomorphism the circuit checks that the mapping between the from the elements of the first generating set to the elements of the second generating set is well defined, namely both one-to-one and onto. If this mapping is well defined then the mapping implies an isomorphism between the two quasigroups since each element in a quasigroup can be written in terms of the corresponding generators. Let $G_1 = \{g_1, g_2, \cdots, g_k\}$, and let $G_2 = \{h_1, h_2, \cdots, h_k\}$. The circuit verifies for all $1 \le i,j \le k$ if $g_i = g_j$ then $h_i = h_j$, and if $h_i = h_j$ then $g_i = g_j$. Since there are $\log_2 n$ elements in each G_i, there are $\log_2{}^2 n$ pairs that must be tested. Each test can be computed by an $O(\text{loglog } n)$ depth and $O(\log n)$ size circuit. Thus, quasigroup isomorphism is in $\text{NNC}^2(\log^2 n)$. □

If we let L be the Latin square associated with Q_1 and L' be the Latin square associated with Q_2 in the previous proof, then we can view the guessing of the generating sets as the guessing of a permutation σ that for all $x,y,z \in G_1$ takes triples $<x,y,z> \in L$ to triples $<\sigma(x),\sigma(y),\sigma(z)>' \in L'$. Now σ can also be applied to elements of Q_1 not in

G_1 if they first are written as the products of generators. To show Latin square isotopism is in $NNC^2(\log^2 n)$ we guess three permutations α, β, γ from three generating sets of L to three generating sets of L' and show that if $<x,y,z> \in L$, then $<\alpha(x),\beta(y),\gamma(z)>' \in L'$.

Theorem 4.3.7: Given two Latin squares L and L' as lists of triples of the form $<a,b,c>$ and $<x,y,z>'$, respectively, then the set $\{(L, L') \mid L$ is isotopic to $L'\}$ is in $NNC^2(\log^2 n)$, where n is the size of the two Latin squares.

Proof. To show two Latin squares are isotopic we need three surjective functions between L and L'. The circuit begins by guessing subsets $A, B \subset L$ and subsets $A', B' \subset L'$. Let $A = \{a_1, a_2, \cdots, a_k\}$, $B = \{b_1, b_2, \cdots, b_k\}$, $A' = \{a'_1, a'_2, \cdots, a'_k\}$ and $B' = \{b'_1, b'_2, \cdots, b'_k\}$, where $k = \lceil \log_2 n \rceil$. Let $C = \{a_1 * b_1, a_2 * b_2, \cdots, a_k * b_k\}$, and $C' = \{a'_1 *' b'_1, a'_2 *' b'_2, \cdots, a'_k *' b'_k\}$, where $*$ and $*'$ are the appropriate binary operators for the respective quasigroups. If we show that A, B and C each generate L and that A', B' and C' each generate L', then we can find the surjective maps needed to take L to L' as follows:

α: For $x \in L$, if $x = a_i$ then $\alpha(x) = a'_i$.
Else if $x = y*z$ then $\alpha(x) = \alpha(y) *' \alpha(z)$.

β: For $x \in L$, if $x = b_i$ then $\beta(x) = b'_i$.
Else if $x = y*z$ then $\beta(x) = \beta(y) *' \beta(z)$.

γ: For $x \in L$, if $x = a_i * b_i$ then $\gamma(x) = a'_i *' b'_i$.
Else if $x = y*z$ then $\gamma(x) = \gamma(y) *' \gamma(z)$.

Thus, we need only test that each of α, β and γ is well defined on A and A', B and B', and C and C', respectively. Using a circuit similar to the one used in deciding quasigroup isomorphism we find that Latin square isotopism can be decided by a circuit with $O(\log^2 n)$ guessing gates and $O(\log^2 n)$ depth. Thus, Latin square isotopism is in $\mathrm{NNC}^2(\log^2 n)$. □

Latin squares give rise to a special class of graphs called Latin square graphs. A Latin square graph consists of n^2 nodes, one corresponding to each of the triples of the Latin square. Two nodes $<x,y,z>$ and $<u,v,w>$ are adjacent if $x=u$, $y=v$ or $z=w$. Namely, two nodes are adjacent if they are in the same row or column of the Latin square or if they share the same value. Thus, Latin square graphs of size n consist of $3 \cdot n$ n-cliques.

To show Latin square graph isomorphism is in $\mathrm{NNC}^2(\log^2 n)$ we need another notion of isotopism. Two Latin squares L and L' are *conjugate* if $<x_1, x_2, x_3> \in L$ implies $<x_{\alpha(1)}, x_{\alpha(2)}, x_{\alpha(3)}>' \in L'$, where α is a permutation in S_3. L and L' are *main class isotopic* if we can get from L to L' by a conjugation and an isotopic map. Since there are only six permutations in S_3, main class isotopism can be decided in $\mathrm{NNC}^2(\log^2 n)$ by giving the circuit that decides isotopism the ability to guess which one of the six permutations in S_3 to use. The next result from Miller [Mi] gives the relationship between Latin squares and Latin square graphs.

Lemma 4.3.8: [Mi] Let L and L' be two Latin squares and $G(L)$ and $G(L')$ be the associated Latin square graphs. L is main class isotopic to L' if and only if $G(L)$ is iso-

morphic to G(L').

In Lemma 4.3.9 we give a means to retrieve the Latin square from a Latin square graph with a circuit with depth no more than $O(\log^2 n)$ depth. The algorithm given is the obvious parallelization of Miller's sequential algorithm that does the same thing in $O(n^3)$ sequential time.

Lemma 4.3.9: We can retrieve the Latin square from the Latin square graph with a polynomial sized circuit with depth less than $O(\log^2 n)$.

Proof. Let $L(l_{ij})$ be an $n \times n$ matrix used to store the Latin square.

(1) Pick two adjacent nodes x_1 and x_2.

(2) In parallel find the n nodes adjacent to both x_1 and x_2. All but two of the nodes form an n-clique with x_1 and x_2. In parallel, label each node in the clique $x_3, \cdots, x_n$. One node not adjacent to any of $x_3, \cdots, x_n$ is labeled y_2.

(3) Associate m_{1j} with x_j, and in parallel set m_{1j} to j.

(4) In parallel find the clique associated with x_1 and y_2, $\{x_1, y_2, y_3, \cdots, y_n\}$.

(5) Each x_i shares an edge with some y_j, $2 \le i, j \le n$. Order the y_j's so that x_j shares an edge with y_j.

(6) Associate m_{j1} with y_j, and in parallel set m_{j1} to j for $2 \le j \le n$.

(7) In parallel for each of the $(n-1)^2$ remaining nodes z of the graph:
   a) If z is adjacent to x_1 then z is adjacent to a unique y_i and a unique x_j, $2 \le i, j \le n$. Set m_{ij} to 1.
   b) Else z is not adjacent to x_1, and there are unique integers i, j and k such that z is adjacent to x_i, y_j, x_k, and y_k. Set m_{ij} to k.

Each of the steps can be computed by an $O(\log n)$ depth circuit with a polynomial number of gates, thus, the Latin square can be easily retrieved from the Latin square graph. □

Theorem 4.3.10: Given two Latin square graphs G_1 and G_2, the set $\{(G_1,G_2) \mid G_1$ is isomorphic to $G_2\}$ is in $\text{NNC}^2(\log^2 n)$, where n is the size of the associated Latin squares.

Proof. This follows directly from Theorem 4.3.7, Lemma 4.3.8, Lemma 4.3.9 and the definition of main class isotopism. □

We now reach the main result of this section.

Theorem 4.3.11: Quasigroup isomorphism, Latin square isotopism and Latin square graph isomorphism are in $\text{DSPACE}(\log^2 n)$.

Proof. This follows directly from Theorem 4.3.6, Theorem 4.3.7, Theorem 4.3.10 and Lemma 3.2.8. □

CHAPTER 5

NONDETERMINISTIC AC

Perhaps the central question in complexity theory is, ''Does nondeterminism help?'' Kintala and Fischer showed that in the case of real-time Turing machines it does help [KF]. In this chapter we answer this question positively for constant depth circuits. We do this by applying some powerful lower bound results developed by Ajtai [Aj], Furst, Saxe, and Sipser [FSS], Håstad [Ha], and Fagin, Klawe, Pippenger, and Stockmeyer [FKPS].

5.1. Definitions

We begin by defining a nondeterministic version of AC^k. Recall that AC^k is the class of languages accepted by LOGSPACE uniform families of polynomial size, $\log^k n$ depth circuits with unbounded fan-in ''AND'', and ''OR'' gates. We define NAC^k to be the class of sets accepted by LOGSPACE uniform families of AC^k circuits with $O(f(n))$ nondeterministic gates, where n is the length of the input. A circuit from such a family is called a uniform NAC^k circuit and such a circuit accepts an input if and only if it outputs a 1 on that input. As is the case with nondeterministic NC circuits, the guessing gates give an NAC circuit a guessing input, y, in addition to the ordinary input, x. An NAC circuit accepts x if and only if there is at least one string of

guessing bits y that causes the circuit to output a 1. All circuits are considered LOGSPACE uniform.

It is well known that $AC^{k-1} \subseteq NC^k \subseteq AC^k$ for all $k \geq 1$, and thus, AC = NC. Essentially, the proof of the first containment relies on the fact that each gate with unbounded fan-in of k can be replaced by a tree of bounded fan-in gates of depth at most $O(\log k)$. Since k can be no bigger than the number of gates in the circuit, the depth of the circuit is increased by no more than a factor of $O(\log n)$.

It is easy to show that the nondeterministic version of these containments holds as well. Namely, for all $k \geq 1$ and for all $f(n)$ we have $NAC^{k-1}(f(n)) \subseteq NC^k(f(n)) \subseteq AC^k(f(n))$.

Let

$$\text{Parity} = \{w \mid w \in \{0,1\}^*, \text{ the number of 1's in is } w \text{ odd}\}.$$

Ajtai [Aj] and Furst, Saxe and Sipser [FSS] independently established that Parity is not in AC^0 and, thus, $AC^0 \subsetneq NC^1$. In the next section we will show that Parity $\notin$ NAC^0(polylog). Therefore, $NAC^0(\text{polylog}) \neq NC^1$. Looking at the proofs that the dominating set in a tournament problem and the CNF-SAT($\log^k n$) problem (problems discussed in Section 4.2) are in NNC^1(polylog), we find that after the guessing phase, the circuit essentially only tests equality. Since equality can be tested in constant depth, we have a peculiar situation where some seemingly hard to compute problems are in NAC^0(polylog), yet the simple Parity prob[illegible] is not. This suggests that some

problems are made difficult only because exhaustive search is required to find a solution, while for other problems, verification that a given solution is correct is the difficult part. Thus, when studying the parallel complexity of problems it may be useful to separate out the complexity of the two phases. The parallel nondeterministic complexity classes NAC and NNC are useful tools for the study of this separation.

Finally we show that AC^0 is strictly different from a number of new classes. It was previously known that $AC^0 \subsetneq NP$. We show that $AC^0 \subsetneq NAC^0(n^\varepsilon) \subseteq NP$ for all $0 < \varepsilon < 1$.

5.2. Separation Results

To show that Parity could not be computed by polynomial size, constant depth unbounded fan-in circuits, Ajtai [Aj] and Furst, Saxe, and Sipser [FSS] independently showed that any constant depth unbounded fan-in circuit computing Parity required at least superpolynomial size. Later Håstad improved this result and showed that such a circuit would require at least exponential size [Ha], and by a result of Yao this bound is close to optimal [Ya85]. In this section we apply Håstad's construction to the nondeterministic case and prove that $NAC^0(\text{polylog}) \subsetneq NNC^1(\text{polylog})$.

Theorem 5.2.1: [Ha] There exists an absolute constant n_0 such that for all k, there are no depth k parity circuits of size $\leq 2^m$ where $m = (1/10)^{k/(k-1)} n^{1/(k-1)}$ and $n > n_0^k$.

We make the following general observation about $NAC^0(f(n))$ circuits.

Lemma 5.2.2: Every $\mathrm{NAC}^0(f(n))$ circuit of size $p(n)$ can be simulated by a constant depth, unbounded fan-in deterministic circuit of size at most $O(p(n)2^{f(n)})$.

Proof: Fix k and let C denote some $\mathrm{NAC}^0(f(n))$ circuit with size $p(n)$. We convert C to a constant depth, unbounded fan-in deterministic circuit by making $2^{f(n)}$ copies of C, removing the nondeterministic gates, and hardwiring in each of the possible guesses in their place. We then add an OR gate that takes an input from each of copies of C, for the new circuit accepts if and only if one of the copies of C accepts. Thus, the size of the circuit is $O(p(n)2^{f(n)})$, and the depth of the circuit is at most one more than that of C. □

Using Theorem 5.2.1 and Lemma 5.2.2 with $f(n) = \log^k n$ we have the following result.

Theorem 5.2.3: For all $k \geq 0$, Parity is not in $\mathrm{NAC}^0(\log^k n)$.

Since Parity can be readily computed with an NC^1 circuit, the next two corollaries are immediate.

Corollary 5.2.4: For all $k, j \geq 1$, $\mathrm{NAC}^0(\log^k n) \subsetneq \mathrm{NNC}^1(\log^j n)$.

Corollary 5.2.5: For all $k \geq 1$, $\mathrm{NAC}^0(\log^k n) \neq \mathrm{NC}^1$.

Of course, this also implies that $\mathrm{NAC}^0(\log^k n)$ is different from P and NP. Thus, Parity, a problem easy to solve both sequentially and in parallel, is not in a class that contains some problems that are probably not solvable in polynomial time (such as dominating set in a tournament). This suggests that we, as researchers, have blindly

assumed that there is some linear ordering on the difficulty of various problems—problems in NP are more difficult than those in P, problems in P are more difficult than those in NC, and so on. By treating nondeterminism as a limited resource in parallel computation we have been able to develop another ordering on the difficulty of the solutions to various problems. Perhaps by considering other resource bounds we can gain insight into what features make certain problems difficult to solve and gain a better understanding of the relative difficulties of solutions to various problems.

Now we give improved upper bounds on how much nondeterminism is needed to allow an NAC^0 circuit to accept some language not accepted by an AC^0 circuit. The obvious upper bound is that $AC^0 \subsetneq NAC^0(n \log n)$. Parity is obviously in $NAC^0(n \log n)$ since a circuit could guess the output of each of the bounded input gates used by an NC^1 parity circuit and then verify that each bit is correct. We will show that $AC^0 \subsetneq NAC^0(n^{\varepsilon})$ for any $\varepsilon > 0$. Before getting to that result, however, we present a result of Fagin, Klawe, Pippenger and Stockmeyer that is an important part of our proof.

We will define a complexity measure that approximates how close a symmetric function f approximates the constant function. A symmetric function is any function whose value does not depend on the order of the input bits. Thus, f is *symmetric* if and only if $f(x_1, x_2, \cdots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)})$ for all permutations π of $\{1, 2, \cdots, n\}$. The *spectrum* of such an n-place function f is a word $w \in \{0, 1\}^{n+1}$, where for $0 \leq i \leq n$, w_i, the i^{th} character of w, equals the value of f when exactly i

variables are set to 1 and the remaining variables are set to 0. Since f is symmetric, w completely characterizes f.

The complexity measure we are interested in is a function of the longest run of all 0's or all 1's in $w = w_0 w_1 \cdots w_n \in \{0, 1\}^{n+1}$. Let $\Gamma(w)$ denote the longest such run. For w, the spectrum of f, let the *measure* of f be defined as $M(w) = n + 1 - \Gamma(w)$. In other words, $M(w)$ is the length of w minus the longest run of all 0's or all 1's. $M(w)$ is the minimum number of variables that must be set to constant values in order for the resulting function to compute the constant function. Consider for example, the Parity function. For the Parity function, $w = 01010101...$, so $M(w) = n$. In other words, all of the variables need to be set before the Parity function can be forced to mimic the constant function.

Let $F = \{f_1, f_2, \cdots\}$ be a family of symmetric functions, where f_n has n variables. Then μ_F, the *measure of the family* F, is defined by $\mu_F(n) = M(w)$ where w is the spectrum of f_n.

The following result, due to Fagin, Klawe, Pippenger and Stockmeyer, says functions that do not closely approximate the constant function do not have small sized constant depth circuits.

Lemma 5.2.6: [FKPS] Let $\varepsilon > 0$ be such that $\mu_F(n) \geq n^{\varepsilon}$ for infinitely many n. Then F is not computable by a family of AC^0 circuits.

Theorem 5.2.7: $AC^0 \subsetneq NAC^0(n^{\varepsilon})$ for any $\varepsilon > 0$.

Proof. Pick any $\varepsilon > 0$. For any $\delta < \varepsilon$ consider the language

$L = \{ x \mid \text{exactly } \lceil |x|^{\delta} \rceil \text{ bits of } x \text{ are set to 1} \}$.

L is clearly in $NAC^0(n^{\varepsilon})$. An NAC^0 circuit accepting $L \cap \{0,1\}^n$ guesses the location of the bits that are set to one, and then a small subcircuit for each bit of input verifies that it is a 1 if it is one of the guessed locations and a 0 otherwise. This takes $\Omega(n^{\delta} \log n) = \Omega(n^{\varepsilon})$ guess bits, and the circuit is of polynomial size and constant depth.

Now consider the family, F, of functions f_n each of which computes membership in L on strings of length n. Certainly each function in F is a symmetric function. Thus, we can compute μ_F. For sufficiently large n, it is not hard to see that if we set $\lceil n^{\varepsilon} \rceil$ variables to 1, the function becomes identically 0. Thus, μ_F is $\Omega(n^{\varepsilon})$. By Lemma 5.2.6, L is not in AC^0, and we have the desired result. □

Theorems 5.2.3 and 5.2.7 shed more light on the relationship between AC^0 and NP. It was previously known that $AC^0 \subsetneq NP$, and that AC^0 is the most powerful class known to be distinct from NP. Theorem 5.2.3 helps strengthen this result to NAC^0(polylog). We now have $AC^0 \subseteq NAC^0(\text{polylog}) \subsetneq NAC^0(\text{polylog}) \subseteq NP$. We also have $AC^0 \subsetneq NAC^0(n^{\varepsilon}) \subseteq NAC^0 \subseteq NP$.

There are two desirable extensions of Theorem 5.2.7. The first would be to show that $AC^0 \subsetneq NAC^0$(polylog). One way to do this would be to improve the result of Fagin, Klawe, Pippenger and Stockmeyer. Unfortunately, they have already shown that it may not be possible to improve their result much.

Lemma 5.2.8: [FKPS] If for sufficiently large n, $\mu_F(n) \leq \log^k n$ for some k, then F has *non-uniform* AC^0 circuits.

This lemma could be strengthened (from our point of view) in two ways. The first involves showing a similar result does not hold for *uniform* circuits. Possibly, Lemma 5.2.8 does not hold in the case of uniform circuits. In this case we would have $AC^0 \subsetneq NAC^0(\text{polylog})$.

A second way to improve this result would be to remove the requirement that F be a set of symmetric functions and consider some other sort of measure. This would essentially require developing a new technique for showing that languages in $NAC^0(\text{polylog})$, such as tournament isomorphism, are not in AC^0. The only technique known to show languages are not in AC^0 is to reduce problems to Parity. Unfortunately, those techniques are also powerful enough to show that Parity and problems related to Parity are not in $NAC^0(\text{polylog})$ as well.

A second interesting result would be to show that $NAC^0(\text{polylog}) \subsetneq NAC^0(n^\varepsilon)$ for all $\varepsilon > 0$. Intuitively, this result seems to be the most likely of the three mentioned since n^ε seems to grow much faster than $\log^k n$. But we have no means of attacking this problem yet.

CHAPTER 6

THE NNC HIERARCHY

In some sense, PSPACE, NP, and P capture the essence of what can be computed sequentially without costing too much in terms of space, nondeterministic time, and deterministic time, respectively. Similarly, DSPACE(polylog), NNC(polylog), and NC capture the essence of what can be computed in parallel without costing too much in terms of space, nondeterministic parallel time, and deterministic parallel time. This suggests that contrasting the two groups of classes will offer insight into why P-complete problems are inherently sequential, as well as aid in deciding which parts of NP-complete problems are inherently sequential. The similarities between PSPACE and DSPACE(polylog), NP and NNC(polylog), and P and NC also suggest the possibility of developing a hierarchy at the NNC level similar to the polynomial time hierarchy presented in Stockmeyer [St].

Let NNC denote NNC(polylog). We can alternately define NNC in terms of languages definable by polylog bounded quantification over the variables of relations in NC. We similarly define co-NNC.

Definition: A set A is in NNC if and only if there is an NC computable predicate P and a number k such that $x \in A$ *iff* $\exists y$ such that $|y| \leq \log^k |x|$ and $P(x, y)$ holds.

Definition: A set B is in co-NNC if and only if there is an NC computable predicate P and a number k such that $x \in B$ *iff* $\forall y$ such that $|y| \leq \log^k |x|$ and $\neg P(x, y)$ holds.

We let Σ_1^{NNC} = NNC and Π_1^{NNC} = co-NNC. We recursively develop a hierarchy of classes with the following definition.

Definition: $S \in \Sigma_{k+1}^{NNC}$ if for some $T \in \Pi_k^{NNC}$, $x \in S$ *iff* $\exists y$ such that $|y| \leq \log^k |x|$ for some k and $(x, y) \in T$. Π_k^{NNC} is defined to be co-Σ_k^{NNC}. We let the NNC hierarchy (NNCH) be $\bigcup_{k \geq 1} \Sigma_k^{NNC}$.

As Stockmeyer [St] showed for the polynomial time hierarchy, if two levels of the NNCH are equal, then the NNC hierarchy is finite and collapses to that level.

Theorem 6.1: If $\Sigma_k^{NNC} = \Sigma_{k+1}^{NNC}$ then $\Sigma_k^{NNC} = \Sigma_j^{NNC}$ for all $j \geq k$.

Proof. The proof is analogous to the proof of the corresponding theorem for the polynomial time hierarchy. Let $A \in \Sigma_{k+2}^{NNC}$. Then $x \in A$ if and only if there exists a y, where $|y| \leq \log^j |x|$ for some j, such that for all z, where $|z| \leq \log^i |x|$ for some i, and $P_k(x, y, z)$ holds where P_k is a Σ_k^{NNC} predicate. Now the sentence

"for all z, where $|z| \leq \log^i |x|$ for some i, and $P_k(x, y, z)$ holds"

is in Π_{k+1}^{NNC}, and by hypothesis it is also in Π_k^{NNC}. Thus, by definition, the statement

"there exists a y, where $|y| \leq \log^j |x|$ for some j, such that for all z, where $|z| \leq \log^i |x|$ for some i, and $P_k(x, y, z)$ holds"

is in Σ_{k+1}^{NNC}, and thus, $\Sigma_{k+1}^{NNC} = \Sigma_{k+2}^{NNC}$. Proceeding inductively, we get the desired result.

□

Stockmeyer has also shown that the polynomial time hierarchy is contained in PSPACE. We show an analogous result for the NNC hierarchy.

Theorem 6.2: NNCH $\subseteq$ DSPACE(polylog).

Proof. Let P(x, y) be a DSPACE(polylog) computable predicate. Let Q(x) be defined by $\exists y(|y| \leq \log^k |x|$ and P(x, y)). Now Q is computable in DSPACE(polylog) since we need only check all potential values of y. Since NC $\subseteq$ DSPACE(polylog) and NNC $\subseteq$ DSPACE(polylog) we have NNCH $\subseteq$ DSPACE(polylog). □

One property of the polynomial time hierarchy the NNC hierarchy does not seem to possess is an equivalent definition in terms of relativized complexity classes. The difficulty is not in relativizing circuits, but in finding a relativization technique that yields an equivalent definition. Wilson has explored the notion of oracle circuits and issues involved in relativizing NC [Wi]. An oracle circuit behaves as any circuit except that it may have any number of "oracle gates." An oracle gate takes k inputs and treating those inputs as a string outputs a 1 if and only if that string is contained in the oracle set. The depth incurred by such a query is log k.

Defining the NNC hierarchy in terms of the oracle circuits introduced by Wilson seems to lead to a hierarchy more powerful than the one developed using quantification. We use S^{NNC} and P^{NNC} to denote the relativized analogs of Σ^{NNC} and Π^{NNC}, respectively.

Theorem 6.3: Let $S_0^{NNC} = P_0^{NNC} =$ NC. Define S_{k+1}^{NNC} = NNC(S_k^{NNC}) and P_{k+1}^{NNC} = co-NNC(P_k^{NNC}), where the class listed in parenthesis is the type of oracle available. Then

for all $k \geq 1$, $\Sigma_k^{NNC} \subseteq S_k^{NNC}$ and $\Pi_k^{NNC} \subseteq P_k^{NNC}$.

Proof. We will show that $\Sigma_k^{NNC} \subseteq S_k^{NNC}$. The proof that $\Pi_k^{NNC} \subseteq P_k^{NNC}$ proceeds analogously.

Let $L \in \Sigma_k^{NNC}$. Thus, $x \in L$ *iff* $\exists y$ such that $|y| \leq \log^j n$ for some j and $(x,y) \in$ R for some Π_{k-1}^{NNC} computable predicate R, where $n = |x|$. We will proceed inductively on k.

When $k = 1$, R is NC computable. An NNC circuit existentially guesses the appropriate y since $|y| \leq \log^j n$, and then verifies that $(x,y) \in$ R.

For $k \geq 1$, assume that $\Sigma_{k-1}^{NNC} \subseteq S_{k-1}^{NNC}$. Note that since Π_{k-1}^{NNC} is the complement of Σ_{k-1}^{NNC}, it follows that $\Pi_{k-1}^{NNC} \subseteq S_{k-1}^{NNC}$. Now an NNC circuit existentially guesses y since $|y| \leq \log^j n$ and with one oracle query determines if $(x,y) \in$ R. Thus, $\Sigma_k^{NNC} \subseteq S_k^{NNC}$.

□

Now it is not clear that $\Sigma_k^{NNC} = S_k^{NNC}$ and $\Pi_k^{NNC} = P_k^{NNC}$ for $k > 1$. This is because the S_k^{NNC} circuits are able to query the oracle about strings of polynomial length. It is also not clear that Σ_k^{NNC} and P_k^{NNC} are contained in DSPACE(polylog). Since any oracle gate can have a polynomial number of inputs, it may not be possible to simulate such a circuit in DSPACE(polylog). The NNCH defined in terms of quantification is more like the polynomial time hierarchy in that it is contained in the corresponding space class. However, the NNC hierarchy defined in terms of oracle circuits may be an interesting restriction of the polynomial time hierarchy worthy study.

CHAPTER 7

FUTURE WORK AND CONCLUSIONS

This work is an extended treatment of limiting nondeterminism in parallel models of computation. After reviewing the main results of this thesis, we will suggest some avenues for further research.

7.1. Summary

Circuits offer a convenient means for quantifying the amount of nondeterminism used in parallel computation. By quantifying the amount of nondeterminism we separated the search part of computation from the verification part.

We have seen that NNC(polylog) is contained in DSPACE(polylog). This containment, in conjunction with the representation for quasigroup elements in terms of generators developed in Chapter 4 has yielded a new space bound for the Latin square and quasigroup isomorphism problems. By artificially adding a guessing part to languages in NC, as we did with context free language problem in Section 4.1, we have shown that each of the NNC^k(polylog) classes is at least as rich in problems as the NC^k classes. We have also shown that restrictions of NP-complete problems lead to problems with solutions that have a polylog amount of guessing coupled with verification phases in NC.

In the case of constant depth, unbounded fan-in circuits we have shown that nondeterminism helps. There are languages an AC^0 circuit cannot accept, that NAC^0 circuits allowed to guess less than a linear amount of information can accept. Furthermore, NAC^0 circuits guessing less than a linear amount of information still cannot compute the Parity function, thus, making them provably weaker than an NC^1 circuit which can make no guesses.

7.2. Future Work

We present some ideas that have sprung from working on this thesis. These ideas are quite varied in both difficulty and detail.

7.2.1. Circuit Definitions for NSC

Since NSC computations are restricted to polylog space, they are in a real sense a form of restricted nondeterministic computation. Running in polynomial time, they can make a nondeterministic move at every step. But the inability to write down the outcome of every nondeterministic move, limits the value of such a move. In Section 2.4.3 we saw that NSC is completely characterized by Turing machines running in polynomial time, with polylog head reversals on one work tape. Is a characterization of NSC possible in terms of NNC circuits? It may be useful to know what in a circuit corresponds to the number of tapes on a Turing machine.

7.2.2. Refining the Complexity of NP-complete Problems

Using NNC circuits it may also be possible to refine the classification of NP-complete problems. Stearns and Hunt [SH] develop the concept of a problem's ''power index'' which is a measure of an NP-complete problem's complexity relative to the Satisfiability problem. They argued that the CLIQUE problem is easier that certain other NP-complete problems since they showed it to have a power index of 1/2, while most other NP-complete problems they studied have power indices of 1. We let $G = (V, E)$ denote a graph with vertex set V and edge set E. Assume that G is represented by an adjacency list. We define

$$\text{CLIQUE} = \{\langle G, j\rangle \mid j \text{ is a positive integer, } j \le |V|, \text{ and } G \text{ contains a clique of size } j \text{ or more}\}$$

By considering NNC classes we hope to make distinctions among the complexities of NP-complete problems. For ease of presentation, we let NNC(n^k) denote the class NNC($n^k \log n$), where the n^k indicates the number of things guessed and the log n denotes the size of each thing guessed. The following theorem may provide an initial step in that direction, as it provides evidence consistent with Stearns and Hunt that CLIQUE is easier than other NP-complete problems.

Theorem 7.2.1: CLIQUE is in NNC($n^{1/2}$), where n is the number of edges in the graph.

Proof. First of all the circuit determines if there are enough edges in the graph to support a clique of size j. If there are enough edges, the circuit guesses the j nodes in

the clique and determines the $(j^2 - j)/2$ edges of the clique that must be in the graph. Then by scanning the input the circuit verifies in parallel that each of the edges in the clique is in the input. The circuit described is of polynomial size and polylog depth, so we need only to verify that the amount of guessing is within the given bounds. Guessing j nodes requires $O(j \cdot \log|G|)$ bits, but since there are $O(j^2)$ edges in the graph, the number of bits guessed is $O(|G|^{1/2}\log|G|)$. Thus, CLIQUE can be accepted by an NNC($n^{1/2}$) circuit. □

It may seem reasonable to assume that using reductions most NP-complete problems can now be shown to be in NNC($n^{1/2}$) since most NP-complete problems have LOGSPACE reductions to CLIQUE. Most reductions, however, cause at least a squaring of the size of the problem, and thus, this technique cannot be used to show other NP-complete problems are in NNC($n^{1/2}$). This is true even for the close relatives of the CLIQUE problem, VERTEX COVER and INDEPENDENT SET, since using the CLIQUE circuit to accept either of these would require computing the complement of the input graph. Thus, even though the CLIQUE problem falls into NNC($n^{1/2}$), it seems most other NP-complete problems fall into NNC(n) and NNC(n^2), suggesting that using the exponent as a guide to relative complexities of NP-complete problems leads to too coarse of a refinement. Changing the definition of NNC(n^k) somewhat may prove useful. If we let NNC($c \cdot n^k$) be the class of languages accepted by NC circuits with $c \cdot n^k \log\ n + d$ guessing gates, we establish more accurately how much nondeterminism is needed to solve a given problem, and the leading coefficient of the poly-

nomial, c, may be a useful measure of relative complexities of NP-complete problems. For example, can the Hamiltonian circuit of a graph be computed quickly if half of the edges in the circuit are guessed? Of course, studying such a measure of complexity will require stringent definitions to prevent the masking guessing by a large alphabet.

7.2.3. Other Potential Problems in NNC(polylog)

One property of problems in NNC(polylog) is that they all have sequential running times of $O(n^{\log n})$. Babai and Luks have shown that tournament isomorphism also has a sequential running time of $O(n^{\log n})$ [BL]. However, the algorithm they give is recursive with an $O(n^{\log n})$ depth of recursion. This algorithm is not the "search and verify" type algorithm. Kannan has shown that a number of problems are polynomially equivalent to the tournament isomorphism problem [Ka]. None of these problems seem solvable by an NNC(polylog) algorithm. There seems to be a sequential nature about them. This suggests that there are two types of problems with $O(n^{\log n})$ running times--those that are parallelizable (with some nondeterminism) and those that are inherently sequential. It would be useful to develop a classification scheme that shows that problems such as tournament isomorphism are all related (much like P-complete problems are related) and probably harder than problems in NNC(polylog).

7.2.4. The Relationship Between RNC and NNC(polylog)

We suspect that RNC and NNC(polylog) are incomparable, although the evidence is not clear. In some sense, RNC and NNC(polylog) are very different complexity

classes. For a set to be in RNC there must be many (a polynomial number) polynomial length witnesses for every string in the set. On the other hand, a string in an NNC(polylog) set needs only one witness, and that witness can be short—it only needs to be of polylog length. We do have some weak evidence that RNC ≠ NNC(polylog). We note that using the obvious approach, quasigroup isomorphism cannot be shown to be in RNC. If the two input quasigroups are cyclic groups of order n it is not difficult to show that fixing the mapping between one pair of generators fixes the mappings between all of the remaining pairs of generators. This forces the probability of finding a string that encodes an isomorphism between the two groups to be less than $1/O(n^{\log n})$ which is certainly less than $1/p(n)$, where p is some polynomial.

Another open problem worth considering is determining the intersection of RNC and NNC(polylog). In light of the recent result of Berger and Rompel [BR] that $(\log^k n)$-wise independence can be simulated in NC, it seems reasonable to suspect that RNC $\cap$ NNC(polylog) = NC.

CHAPTER 8

BIBLIOGRAPHY

[Aj] M. Ajtai, Σ_1^1-formulae on finite structures, *Annals of Pure and Applied Logic*, **24** (1983).

[ADT] C. Alvarez, J. Diaz, and J. Toran, Complexity classes with complete problems between P and NP-C, *1989 Fundamentals of Computation Theory* in *Lecture Notes in Computer Science* V. 380, Springer-Verlag, New York, (1989) 13-24.

[BL] L. Babai and E.M. Luks, Canonical labeling of graphs, *Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing* **15** (1983).

[BGS] T. Baker, J. Gill, and R. Solovay, Relativization of the P =? NP question, *SIAM Journal of Computing* **4** (1975) 431-442.

[BR] B. Berger and J. Rompel, Simulating ($\log^c n$)-wise independence in NC, *Proceedings of the Thirtieth Annual IEEE Symposium on the Foundations of Computer Science* **30** (1989).

[Bo] A.B. Borodin, On relating time and space to size and depth, *SIAM Journal of Computing* **6** (1975) 733-744.

[CKS] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer, Alternation, *Journal of the Association of Computing Machinery* **28** (1981), 114-133.

[Co] S.A. Cook, The classification of problems which have fast parallel algorithms, *1983 Foundations of Computation Theory* in *Lecture Notes in Computer Science* V. 158, Springer-Verlag, New York, (1983) 78-93.

[Dy] P.W. Dymond, On nondeterminism in parallel computation, *Theoretical Computer Science* **47** (1986) 111-120.

[FKPS] R. Fagin, M.M. Klawe, N.J. Pippenger, and L. Stockmeyer, Bounded-depth, polynomial-size circuits for symmetric functions, *Theoretical Computer Science* **36** (1985) 239-250.

[FK] P.C. Fischer and C.M.R. Kintala, Real-time computations with restricted nondeterminism, *Mathematical Systems Theory* **12** (1979) 219-231.

[FW] S. Fortune and J. Wyllie, Parallelism in random access machines, *Proceedings of the Tenth ACM Symposium on the Theory of Computing* **10** (1978) 114-118.

[FSS] M. Furst, J.B. Saxe, and M. Sipser, Parity, circuits, and the polynomial-time hierarchy, *Mathematical Systems Theory* **17** (1984) 13-27.

[GJ] M.R. Garey and D.S. Johnson. *Computers and Intractability*. New York: W.H. Freeman and Company, 1979.

[Ha] J. Håstad, Almost optimal lower bounds for small depth circuits, *Proceedings of the Eighteenth ACM Symposium on the Theory of Computing* **18** (1986) 6-20.

[Ho] C. Hoffman Subcomplete generalizations of graph isomorphism, *Journal of Computer and System Sciences* **25** (1982) 332-348.

[HU] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, Mass., 1979).

[Ka] S. Kannan, Tournament and graph isomorphism complete problems, unpublished manuscript (1990).

[Ki77] C.M.R. Kintala, Computations with a restricted number of nondeterministic steps, Ph.D. Dissertation, Pennsylvania State University, University Park, PA, (1977).

[Ki78] C.M.R. Kintala, Refining nondeterminism in context-free languages, *Mathematical Systems Theory* **12** (1978) 1-8.

[KF77] C.M.R. Kintala and P.C. Fischer, Computation with a restricted number of nondeterministic steps, *Proceedings of the Ninth ACM Symposium on the Theory of Computing* **9** (1977) 178-185.

[KF80] C.M.R. Kintala and P.C. Fischer, Refining nondeterminism in relativized polynomial-time bounded computations, *SIAM Journal of Computing* **9** (1980) 46-53.

[KW] C.M.R. Kintala and D. Wotschke, Amounts of nondeterminism in finite automata, *Acta Informatica* **13** (1980) 199-204.

[LSZ] R.J. Lipton, L. Snyder and Y. Zalcstein, Complexity of the word and

isomorphism problems for finite groups, *Proceedings of the Conference on Information Sciences and Systems* **10** (1976) 33-35.

[MF] A.R. Meyer and M.J. Fischer, Economies of description by automata, grammars and formal systems, *Proceedings of the Twelfth Annual IEEE Symposium on the Foundations of Computer Science* **12** (1971) 188-191.

[Mi] G.L. Miller, On the $n^{\log n}$ isomorphism technique, *Proceedings of the Tenth ACM Symposium on the Theory of Computing* **10**(1978) 51-58.

[MV] N. Megiddo and U. Vishkin, On finding a minimum dominating set in a tournament, *Theoretical Computer Science* **61** (1988) 307-316.

[MIK] E. Moriya, S. Iwata, and T. Kasai, A note on some simultaneous relations among time, space, and reversal for single work tape nondeterministic Turing machines, *Information and Control* **70** (1986) 179-185.

[Na] I.R. Nasyrov, The degrees of nondeterminism in pushdown automata, *1987 Fundamentals of Computation Theory* in *Lecture Notes in Computer Science* V. 278, Springer-Verlag, New York, (1987) 339-342.

[Pa] I. Parberry, A note on nondeterminism in small, fast parallel computers, *IEEE Transactions on Computers* **38** (1989) 766-767.

[PPST] W.J. Paul, N. Pippenger, E. Szemerédi, and W.T. Trotter, On determinism versus non-determinism and related problems, *Proceedings of the Twenty-fourth Annual IEEE Symposium on the Foundations of Computer Science* **24** (1983) 429-438.

[Po] C. Pomerance, Very short primality proofs, *Mathematics of Computation* **48** (1987) 315-322.

[RS59] M.O. Rabin and D. Scott, Finite automata and their decision problems, *IBM Journal of Research and Development* **3** (1959) 114-125.

[RS82] R.L. Rivest and A. Shamir, How to reuse a ''write-once'' memory, *Proceedings of the Fourteenth ACM Symposium on the Theory of Computing* **14** (1982) 105-113.

[Ru80] W.L. Ruzzo, Tree-size bounded alternation, *Journal of Computer and System Sciences* **21** (1980) 218-235.

[Ru81] W.L. Ruzzo, On uniform circuit complexity, *Journal of Computer and System*

Sciences **22** (1981) 365-383.

[Sa] W. Savitch, Relationships between nondeterministic and deterministic tape complexities, *Journal of Computer and System Sciences* **4** (1970) 177-192.

[SH] R.E. Stearns and H.B. Hunt, III, On the complexity of the satisfiability problem and the structure of NP, *Technical Report 86-21,* State University of New York at Albany, Computer Science Department (1986).

[St] L.J. Stockmeyer The polynomial time hierarchy, *Theoretical Computer Science* **3** (1977) 1-22.

[Wi] C.B. Wilson, Parallel computation and the NC hierarchy relativized, *Proceedings of the First Structure in Complexity Theory Conference* **1** (1986) 362-382.

[Wo] M.J. Wolf, Nondeterministic circuits, space complexity and quasigroups, *University of Wisconsin Computer Sciences Technical Report #870* (1989).

[Ya62] H. Yamada, Real-time computation and recursive functions not real-time computable, *IEEE Transactions on Electronic Computers* **11** (1962) 753-760.

[Ya85] A.C. Yao, Separating the polynomial-time hierarchy by oracles, *Proceedings of the Twenty-sixth Annual IEEE Symposium on the Foundations of Computer Science* **26** (1985) 1-10.