INVERTIBLE FUNCTIONS

A THESIS

Presented to

The Faculty of the Division of Graduate Studies

By

Eric Warren Allender

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in the School of Information and Computer Science

Georgia Institute of Technology

September, 1985

ACKNOWLEDGEMENTS

I wish to acknowledge the contributions of two groups of people. One group consists of those persons who helped determine the directions in which my research developed. The other group consists of persons who helped determine the directions in which I developed.

To the first group belong first and foremost my advisor: Kim King, and the other members of my committee: Ray Miller, Rich DeMillo, and Craig Tovey. Thanks is due to them for helping me to learn what is interesting and what is not.

Theorem 3.6 grew out of a discussion with Marc Graham. The origins of Theorems 6.4 and 6.5 can be traced to a discussion with Jin-yi Cai. Chapter 7 was inspired by some comments from Charles Rackoff. Other observations which have found their way into the dissertation first came up in separate conversations with Steve Cook, Jim Hoover, Pat Dymond, and Larry Ruzzo.

I would like to thank Don Alton and Ted Baker for first introducing me to the pleasures of theoretical computer science. Thanks are due to Jim Allchin, Lucio Chiaraviglio, Mike Merritt, John Muller, Barbara Smith-Thomas, Jerry Spinrad, and Gopalakrishna Vijayan, in addition to those mentioned above, for creating a favorable environment in which to pursue the study of theoretical issues. Thanks are also due to the National Science Foundation for supporting me under grant number MCS 81-03608, and to Ed Rumiano for keeping the paperwork flowing and keeping the checks coming; that also helped create a favorable environment.

The past five years would have unthinkable without the reality breaks provided by the Chattahoochee Country Dance Society, the Seed and Feed Marching Abominable Band, and my racketball partner, John Yntema.

To my mother and father, I owe all that I am.

To Claire Todd, my bride, I owe all that I shall be. This is dedicated to her.

TABLE OF CONTENTS

Abstract

This thesis explores the question of how hard a function must be to compute, for its inverse to be difficult. Results in the thesis delineate a sharp boundary between classes of machines computing only functions with easy inverses and classes which contain machines which compute functions whose inverses are intractable.

We show that if a function is easy enough to compute, then its inverse is easy to compute, in the sense that it can be computed in polynomial time. In fact, we prove the stronger result that the inverses of such functions can be computed extremely quickly on a parallel computer which has a feasible number of processors. In order to make that notion precise, we define a new complexity class, PUNC, which models the notion of "feasible parallelism" more naturally than classes which have been studied previously. We present several equivalent characterizations of PUNC and explore relationships between PUNC and other complexity classes.

The results and techniques which we develop lead to a number of peripheral results. The characterizations of PUNC lead to a complexity-theoretic treatment of precomputation. Our results about classes of invertible functions yield corollaries about "ranking" functions, which have applications to the theory of data compression. Those results also lead to theorems about the complexity of sparse sets and about the structure of complexity classes.

CHAPTER I

Introduction

One-Way Functions

There has been a great deal of interest lately in "one-way" functions: functions which are easy to compute, but whose inverses are hard to compute. One-way functions have been useful in designing public-key cryptosystems [RSA-78, Ya-82] and secure pseudo-random number generators [BM-84, Ya-82, Le-85]. It has been shown that if one-way functions exist, then probabilistic algorithms can be simulated quickly by deterministic machines [Ya-82]. One-way functions have also been studied in relation to questions about the structure of complexity classes [JY-85, Yo-83, Wa-85]. Because of the importance of one-way functions, we consider the following question to be fundamental:

How hard must a function be to compute for its inverse to be intractable?

In other words, we are interested in proving lower bounds on the complexity of one-way functions.

In order for this question to be meaningful, we must have some notion of the complexity of computing a function, and we must agree on what the proper notion of "inverse" is, and on what it means for a function to be intractable. Unfortunately, there is little agreement on these matters; certain definitions are appropriate in some settings but not in others. In this section we will review past work which relates to one-way functions, noting how the notion of "one-wayness" varies according to the circumstances. Then, having completed the review, we will present the definitions which we will be using as we consider the question presented above.

An *inverse* of a function f is a function g such that $f(g(y)) = y$ for all y in the range of f. That is, g takes an element y and finds some x which is mapped to y by f, if such an x exists. Note that, according to this very general definition, a function f will have more than one inverse if it is not one-one or if it is not onto. This notion of inverse is general enough to include most other notions of inverse which have been considered in the literature.

For reasons which have been discussed at length elsewhere (see, e.g., [GJ-79]), a function f is considered to be easy to compute if there is a Turing machine computing f which has a running time bounded by a polynomial in the length of its input. Although that is the standard notion, it is not uncommon for a function to be considered easy if it can be computed in *random* polynomial time: that is, if there is a Turing machine which is allowed to flip coins and to make decisions based on the outcomes of the flips, and which, on input x of length n, halts in polynomial time and outputs $f(x)$ with probability $1-(2^{-n})$. For instance, primality testing is in random polynomial time [Ra-80, SS-77], although it is not known if primality testing can be done in deterministic polynomial time (although see [Mi-76]). (The definition of random polynomial time is robust in the sense that minor changes to the definition do not change the class of functions defined. More precise definitions can be found in [Gi-77, Za-82].)

If both a function f and an inverse for f are easy, then clearly f is not a one-way function. Thus the simplest, and least restrictive, definition of one-wayness states that f is one-way iff f is computable in polynomial time and no inverse of f is computable in polynomial time.

Unfortunately, many functions are one-way according to this definition even though they have inverses which are conceptually very simple. For instance, the function f defined by $f(x) = 1^{\lceil \log |x| \rceil}$† is one-way simply because it shrinks its input by more than a polynomially-bounded amount. Such functions are called "dishonest"; a function f is *honest* iff there is a polynomial p such that for all y in the range of f, there is an x such that $p(|y|) \geq |x|$ and $f(x) = y$. Clearly, only honest functions can be inverted in polynomial time.

The existence of honest one-way functions is an open problem.

Proposition 1.1:

P $\neq$ NP $\Leftrightarrow$ there exists an honest function f which can be computed in polynomial time, but which has no inverse which can be computed in polynomial time.

Proof: ($\Rightarrow$) Let $f(A,y) = y$ if y is an instance of SAT and A is a satisfying truth assignment to the variables of y; $f(A,y) = a \vee \neg a$ otherwise. If f has an inverse g computable in polynomial time, then $y \in$ SAT iff $f(g(y)) = y$, and thus P = NP.

($\Leftarrow$) Let f be an honest function computable in polynomial time, and let q be a polynomial such that for all $y \in$ range(f), there exists an x such that $f(x) = y$ and $q(|y|) \geq |x|$. Then the set L $= \{x\#y \mid$ there is a string w such that $|xw| \leq q(|y|)$ and $f(xw) = y\}$ is in NP. If L is in P, then it is easy to compute an inverse for f by building x bit-by-bit. □

It may happen that a function f is honest and one-way, but has some inverse g such that, while g is not computable in polynomial time, there is a program computing g which requires only polynomial time for a nonnegligible fraction of the inputs (or, equivalently, there is a polynomial-time Turing machine which computes a function g' which agrees with g on a large portion of the inputs). Such a function f may not be hard enough to invert to be useful in some situations.

For example, consider the following use of one-way functions. User passwords must be stored in some way by an operating system. If the passwords are all stored together in a

†Notation is defined in the "Preliminaries" section: Chapter 2.

file somewhere, then anyone who finds a way to read the contents of that file can log onto any other user's account. One way to add more security is to use some one-way function f: for each user with password w, the operating system stores $f(w)$ in its password file. When the user logs in and types the password w, the system computes $f(w)$ and verifies that the password is correct. However, anyone who finds a way to read the password file has no way to recover the password w from $f(w)$, since that involves inverting f, which is hard, since f is one-way. (This was, in fact, the original use of one-way functions; the Titan operating system project investigated protecting user passwords in this way [Wi-68]. This scheme is also used in the UNIX† system [MT-79].) If, however, there is a function g' which runs in polynomial time and computes an inverse for f on one third of the inputs, then the level of security provided by this scheme is not acceptably high.

The question of what is a "nonnegligible fraction of the inputs" is a ticklish one (see, e.g., [Jo-84, Le-85]). For instance, it may be that f is hard to invert for all but $1/n^2$ of the inputs of length n (or for all but $1/n^{\sqrt{n}}$ of the inputs of length n), but it is precisely those inputs on which f is easy to invert which are "likely" to appear. One way to tackle this problem which has yielded a number of interesting and useful results is to postulate that inputs to a given problem are distributed according to the output of some feasible process, where the inputs to the feasible process are distributed uniformly. (A very informal justification of this view is the following: consider the inputs to a problem and the probability distribution on those inputs. Where did those inputs come from? What produced the probability distribution? It is reasonable to hypothesize the existence of a deterministic or probabilistic process which is producing inputs to the problem.) The class of probability distributions which can be produced in this way is very large.

†UNIX is a Trademark of Bell Laboratories.

In measuring the "one-wayness" of a function f with respect to some distribution h, we are thus interested in finding a probability function $q(n)$ such that for any honest polynomial-time function g, the probability that $f(g(h(x))) \neq h(x)$ is at least $q(|x|)$, where the inputs x are uniformly distributed. (Equivalently, we are measuring the probability that $f(g(y)) \neq y$, where the inputs y are distributed according to the probability that they appear in the range of h, where the inputs to h are uniformly distributed.) Let us call a function which can be computed in polynomial time and whose inverse is intractable in this sense for all large n *$q(n)$-securely one-way with respect to h*. If h is the identity function, we shall simply call such a function *$q(n)$-securely one-way*.

The aforementioned system for encoding passwords in an operating system requires a high level of security; ideally the one-way function used in the system should be nearly $1-1/2^n$-securely one-way. One-way functions used in public-key cryptosystems also need to be quite secure. It is perhaps surprising that functions which are as little as $1/k$- or $1/p(n)$-secure for some constant k or polynomial p also have applications. Such functions can be used to construct secure pseudo-random number generators.

Pseudo-Random Number Generators

A pseudo-random number generator is a feasible (deterministic or random) algorithm which takes a short random seed x of length n and produces a long sequence of bits $b_1, b_2, \ldots, b_{p(n)}$, where p is some polynomial. A pseudo-random number generator is *secure* if for all i, it is hard to predict b_{i+1} from $b_1, b_2, \ldots, b_i$. The sense in which the next bit must be "hard" to compute varies from author to author. For instance, Yao [Ya-82] and Levin [Le-85] require that no probabilistic polynomial-time Turing machine can predict the next bit correctly with probability $1/2 + 1/p(n)$ for any polynomial p for seeds of size n. Using this notion of security, there is no feasible algorithm which can infer the next bit of the sequence for all seed lengths n. Blum and Micali [BM-84] considered a stronger notion of security: they require that for all seed lengths n there is no feasible algorithm which can infer the next bit of the sequence. Thus they stipulate that no circuit of size polynomial in n predicts b_{i+1} from $b_1, b_2, \ldots, b_i$ on seed length n with probability $1/2 + 1/p(n)$.

The connection between one-way functions and pseudo-random number generators was first pointed out by Shamir and Rivest [Sh-81]. Intuitively, if f is one way, then x is hard to infer from $f(x)$, $f(x)$ is hard to infer from $f(f(x))$, and so forth, and thus $f^{p(n)}(x)$, $f^{p(n)-1}(x), \ldots,$ $f(x)$, x seems to have some of the characteristics of a pseudo-random sequence. Shamir [Sh-81] gives a generator which produces a sequence of *numbers* (not *bits*), where the problem of inferring the next *number* in the sequence is equivalent to inverting the RSA encryption function [RSA-78]. Note that inferring the next *number* in the sequence may be much harder than predicting the next *bit*. Blum and Micali [BM-84] were the first to present a pseudo-random number generator which is secure in the sense given above, assuming that the discrete logarithm problem is $1-\omega(1/p(n))$-secure for all polynomials p. Yao showed in [Ya-82] that any honest one-way one-one function which is $1/p(n)$-secure with respect to certain kinds of distributions gives rise to a secure pseudo-random number generator. Levin [Le-85] showed that the existence of pseudo-random number generators is *equivalent to* the existence of (not necessarily one-one) one-way functions f which are $1/k$-secure with respect to f^i for $1 \leq i \leq n^3$ on inputs of length n (for any constant k).

The link between one-way functions and pseudo-random number generation leads to some surprising results about complexity classes. The class BPP was defined by Gill in [Gi-77] to be the class of languages accepted by probabilistic polynomial-time Turing machines, as presented above. It was observed by Yao [Ya-82] that if pseudo-random number generators exist, then a probabilistic Turing machine M which flips a coin $p(n)$ times can be simulated by a probabilistic Turing machine M′ which flips a coin only n times (to get the random seed) and then generates a pseudo-random sequence of length $p(n)$ which it uses to simulate $p(n)$ coin flips. If M′ does not compute the same function as M, then it can be shown that the bits of the pseudo-random sequence can be inferred by a polynomial-time probabilistic Turing machine, which would be a contradiction. In this way, we can reduce the amount of coin flipping by any polynomially-bounded amount.

Now consider a deterministic simulation of a probabilistic machine. The best containment result which is known is the trivial result BPP $\subseteq$ DTIME($2^{n^{O(1)}}$), which is obtained by simulating a given probabilistic machine M by deterministically writing out all possible sequences of coin flips which M may have made, and simulating M on each sequence of flips. However, if (secure) one-way functions exist, then so do pseudo-random number generators, and a given probabilistic machine M can be simulated indirectly, by instead simulating the machine M′ which flips coins much less often than M, as considered above. In this way one can show BPP $\subseteq \bigcap_{e>0}$ DTIME($2^{n^{e}}$) [Ya-82].

Encryption

One-way functions have many applications in cryptography, both in private-key systems (because of the link between one-way functions and pseudo-random number generators) and in public-key cryptosystems.

In a *private-key cryptosystem*, two users A and B share a secret key k. When A wants to send a message M to B, A applies an encryption routine $E(k,M)$, and, to receive the message, B applies a decryption routine $D(k,E(k,M)) = M$. A *one-time pad* is a private-key cryptosystem in which $|k| = |M|$ and the encryption and decryption routines involve simply applying the exclusive OR operation to the operands. One-time pads were shown by Shannon to be secure from cryptanalysis [Sh-49]. A drawback to one-time pads, however, is that the length of the key is equal to the length of the message. This drawback can be avoided by using pseudo-random number generators. Let A and B share a short secret key k_1. Then A

and B can compute a longer key k by generating a pseudo-random sequence with k_1 as the seed. Such a system is secure from all feasible cryptanalytic attacks [Ya-82].

A major problem with private-key cryptosystems is that every pair of users must share a secret key, and some secure method must be provided for distributing the secret keys. In order to sidestep this problem, public-key cryptosystems were introduced in [DH-76]. A great deal of research has centered on public-key cryptography in the intervening decade.

A *public-key cryptosystem* consists of encryption and decryption algorithms E and D, as well as a key generator G which produces pairs (k, k_p) where k is the secret (or private) key, and k_p is the public key. It is required that

(1) for all messages M, $D(k,E(k_p,M)) = M$,

(2) E and D are easy to compute,

(3) given k_p and $E(k_p,M)$, it is hard to compute M.

If a given user Alice wants to receive encrypted messages, she uses G to generate a pair (k, k_p). She keeps k secret but makes k_p public (hence the name public-key cryptosystem). Now if Bob wants to send Alice a message M, Bob looks up Alice's public key and sends Alice $E(k_p,M)$. To read the message, Alice computes $D(k,E(k_p,M))$ and reads M.

Good treatments of public-key cryptography can be found in [DDD-83, DH-76, Ko-81, Le-79].

Let us now consider a particular public-key cryptosystem, and let S be the set of all valid public keys. Let f be the function which takes (k_p,M) to $(k_p,E(k_p,M))$. By point (3) above, f is one-way. Note that f is also one-one on $S\times\Sigma^*$, since if $E(k_p,M) = E(k_p,M')$, then $M = D(k,E(k_p,M)) = D(k,E(k_p,M')) = M'$. Thus any public-key cryptosystem gives rise to a one-way function which is one-one on the set of all "valid" inputs. Because it is one-one functions which are of the greatest interest in this setting, many authors do not consider a function to be one-way unless it is one-one (e.g., [GS-84, Ya-82, Yo-83]).

If the cryptosystem is secure, the function f will be very securely one-way on $S \times \Sigma^*$. Unfortunately, since the existence of one-way functions implies $P \neq NP$, it is currently not known how to prove that a given cryptosystem is secure, even in the much weaker sense of being hard to crack for infinitely many (key, message) pairs. Some public-key cryptosystems have been shown to be secure if certain apparently-intractable problems such as factoring are indeed intractible [Ra-79], but the goal of finding a cryptosystem which is NP-hard to crack has proved elusive. Indeed, there is a considerable body of evidence that no such cryptosystem exists.

The question of the existence of NP-hard public-key cryptosystems reduces to the question of whether the problem of inverting f on $S \times \Sigma^*$ is NP-hard: that is, can SAT be solved in polynomial time relative to an oracle for g, where g is any function which agrees with the inverse of f on $S \times \Sigma^*$? A number of researchers have addressed this question recently [SY-82, Gr-84, GS-84, Re-85], and a number of surprising consequences would follow if such a function f could be shown to exist. It would be even more surprising if the range of such a function f were in coNP; the next proposition shows that the existence of such a function would imply NP = coNP.

Proposition 1.2: [BH-77, Br-79, Br-83, Mi-76]

If there exists an honest function f, computable in polynomial time, such that for some set $A \in NP$, f is one-one on A, $f(A) \in coNP$, and f^{-1} is hard to compute on $f(A)$, then $P \subsetneq NP \cap coNP$. (Thus in particular if $P = NP \cap coNP$ then there are no honest one-way bijections.)

Proof: Let $T = \{(x,y) \mid x \in f(A) \text{ and } f^{-1}(x) \geq y\}$. To accept T nondeterministically, guess a z such that $z \in A$ and $f(z) = x$ (here we require f to be honest) and check that $z \geq y$. To accept the complement of T, check that $x \notin f(A)$ and if so, accept. Otherwise guess a z such that $z \in A$

and $f(z) = x$ and check that $z < y$. Thus $T \in$ NP $\cap$ coNP. If $T \in$ P, then f^{-1} can be computed in polynomial time via a binary search strategy. □

This result shows why it is difficult to prove that an honest one-one function is one-way. Since it is beyond the reach of current techniques in complexity theory to prove non-linear lower bounds on the time complexity of natural problems in NP, the only technique available for showing that a problem is hard is to reduce some known hard problem to it. If it were possible to reduce an NP-complete problem to the problem of computing f^{-1} for some honest one-one function f whose range is in coNP, it would follow from the above proposition that NP=coNP, which is unlikely. Thus the best one could hope for is to try to reduce some problem complete for NP $\cap$ coNP to f^{-1}. Unfortunately, it seems unlikely that such problems exist, since by [Gu-83, Si-82] there exist oracles relative to which there are no complete problems for NP $\cap$ coNP.

Brassard, in a study motivated by complexity considerations in cryptography, was able to show the existence of an honest bijection h which is one-way relative to some oracle (in fact, relative to an oracle for h) [Br-83].

The Structure of Complete Sets

Interest in one-way functions has also been generated by developments in the theory of NP-completeness. During the initial flurry of activity following the discovery of NP-complete sets, it became clear that some NP-complete sets were very closely related in the sense that each seemed to be a simple reencoding of the others. Furthermore, certain similar NP-complete problems shared the property that good approximate solutions for them could be computed efficiently. Other researchers noticed that often there were reductions between NP-complete problems that preserved the number of ways a given input could be accepted. Thus people began looking for a way to examine the structure of the class of NP-complete sets. (See e.g., [ADP-80, BH-77, LL-78, GJ-79].)

One way this work progressed was to consider p-isomorphisms: bijections f such that both f and f^{-1} are computable in polynomial time. If one grants that functions computable in polynomial time are easy to compute, it follows that if there is a p-isomorphism reducing A to B, then A and B are just equivalent ways of reencoding the same problem. The surprising result of the investigation into the structure of the class of NP-complete sets under p-isomorphism is the following [BH-77]: all "natural" NP-complete sets can easily be shown to be p-isomorphic. This led to the

Berman-Hartmanis Conjecture: [BH-77] All NP-complete sets are p-isomorphic.

Several remarks about this conjecture are in order. First, we note that the Berman-Hartmanis conjecture is motivated in part by an analogy between the theory of NP-completeness and the theory of recursive functions. (See [Ro-66] for definitions and concepts relating to recursive function theory.) In this analogy, NP corresponds to the class of recursively enumerable (r. e.) sets, NP-complete sets correspond to complete r. e. sets, and p-isomorphism corresponds to recursive isomorphism. All complete r. e. sets are recursively isomorphic. The proof of this fact uses the fact that if C is a complete r. e. set, then C is recursively isomorphic to $C \times \Sigma^*$. Any set C having the property that C is isomorphic to $C \times \Sigma^*$ is called a *cylinder*. It is a short step now to define the polynomial analogue of cylinders: a set C will be called a *p-cylinder* if C is p-isomorphic to $C \times \Sigma^*$. It turns out that all "natural" NP-complete sets can easily be shown to be p-cylinders and hence p-isomorphic to SAT. The Berman-Hartmanis conjecture is true if and only if all NP-complete sets are p-cylinders.

One final note about p-cylinders: C is a p-cylinder if and only if there is a one-one, invertible, polynomial-time computable *padding function* p such that $p(x,y) \in C \Leftrightarrow x \in C$. (This and other facts about p-cylinders are in [Do-82, Yo-83, MY-85], although the original ideas, in another formulation, date back to [BH-77].)

The Berman-Hartmanis conjecture has stimulated much work on the structure of complexity classes under p-isomorphism. A basic result is the following:

Proposition 1.3: [BH-77]

Let f reduce A to B, and let g reduce B to A, where f and g are one-one, invertible, and length-increasing polynomial-time computable functions. (A function f is *length-increasing* if $|f(x)| > |x|$ for all x.) Then A and B are p-isomorphic.

Thus, when investigating the structure of the p-isomorphism classes of the NP-complete sets, the natural reducibility relation to consider is one-one, length-increasing, invertible Karp reductions. (In [Re-83], these are called LIFP reductions.) Let us write $A \leq_{\mathrm{LIFP}} B$ if A is reducible to B via a LIFP reduction. If furthermore, B is *not* LIFP-reducible to A, we write $A <_{\mathrm{LIFP}} B$. By Proposition 1.3, if $A \leq_{\mathrm{LIFP}} B$ and $B \leq_{\mathrm{LIFP}} A$, then A and B are p-isomorphic.

By the results of [MY-85, Re-83, Yo-83], we now know that if the Berman-Hartmanis conjecture is false, then the structure of the NP-complete sets under $\leq_{\mathrm{LIFP}}$ is chaotic; if there exist non-p-isomorphic NP-complete sets, then any countable partial order can be embedded in the partial order of the p-isomorphism classes of the NP-complete sets under $\leq_{\mathrm{LIFP}}$.

Analogous questions have been formulated for other complexity classes (e.g., PSPACE, DTIME(2^{lin})) and for other reducibilities (e.g., logspace reducibility). An important fact is the following.

Proposition 1.4:

If L is complete for DTIME($2^{O(n)}$) under polynomial-time reductions, then L is complete under one-one, length-increasing polynomial-time reductions.

This was first proved in [Be-77]; increasingly simple proofs have since appeared in [Do-82] and [Wa-85]. Analogous results hold for PSPACE under logspace reductions, and for higher deterministic complexity classes.

Corollary:

All languages complete for DTIME($2^{O(n)}$) under polynomial-time reductions are p-isomorphic if the inverse of every length-increasing one-one polynomial-time computable function is computable in polynomial time.

The converse to this corollary is still open, although significant progress was made by Young in [Yo-83]. Young was actually considering the class NP, rather than the higher classes. For each honest, one-one, polynomial-time computable function f, and for every integer k, Young defines a set $K_{f,k}$ which is NP-complete. Furthermore, Young offers evidence that if f is not invertible, then $K_{f,k}$ is not p-isomorphic to SAT. Since Young believes that such non-invertible functions f exist, he makes the

Conjecture: [Yo-83] The Berman-Hartmanis conjecture is false.

Other results relating to the Berman-Hartmanis conjecture have appeared in [JY-85, Ha-83, Ku-83, Wa-85]. Logspace isomorphisms were investigated in [Ha-78].

Finite Functions

One-way functions have also been studied in a very different setting in investigations into the complexity of finite functions. A finite function is a function f: $\{0,1\}^n \rightarrow \{0,1\}^n$ for some n. The complexity of a finite function f, $C(f)$, is usually defined as the size of the smallest circuit computing f.

Boyack [Bo-85] investigated one-one finite functions and found an infinite sequence of finite functions f_n: $\{0,1\}^n \rightarrow \{0,1\}^n$ such that $C(f_n{}^{-1}) > C(f_n)$. In [Bo-85] a sequence (f_n) of one-one finite functions is considered to be one-way if the set $\{C(f_n{}^{-1}) / C(f_n)\}$ is unbounded. It is not known if a sequence of one-one finite functions f which is one-way in this sense exists.

In [BL-85] a sequence (f_n) is considered to be one-way if the set $\{\log C(f_n{}^{-1}) / \log C(f_n)\}$ is unbounded. (The functions f_n were not required to be one-one.) It was shown in [BL-85] that such a sequence exists iff every set in NP has polynomial-sized circuits. It was also shown in [BL-85] that a sequence (f_n) of one-one finite functions exists which is one-way with respect to constant-depth circuits: more specifically, each function in the sequence can be computed by a circuit of polynomial size and depth four, but there is no depth d and polynomial bound $p(n)$ such that $f_n{}^{-1}$ can be computed on a circuit of depth d and size $p(n)$ for all n. Other functions which are one-way with respect to small classes are considered in [RT-84].

Lower Bounds

The preceding sections presented motivation for the study of one-way functions, and illustrated how the concept denoted by "one-way function" varies according to circumstances. The question of proving lower bounds on the complexity of one-way functions has not been addressed previously. In addressing this question, we will use the simplest interesting notion of one-wayness: an honest function f which can be computed in polynomial time will be considered to be one-way if no inverse of f can be computed in polynomial time. The next difficulty to overcome is the question of how to measure the difficulty of computing a function.

The approach of automata-based complexity theory has been to equate the difficulty of computing a function f with the complexity of the simplest machine computing f, where the complexity of a machine is determined by its time and memory requirements, its storage structures, and the way it accesses its input data.

A bewildering variety of storage structures has been considered in the literature. Among these, we mention counters [FMR-68], reversal-bounded counters [Ib-78], reversal-bounded pushdown stores [BB-74], reset tapes [BGW-79], Post tapes [Br-80], stacks [GGH-67], nonerasing stacks [HU-67], and checking stacks [Gr-69]. Common types of allowable access to the input data include having multiple input heads and restricting the number of times an input head may cross the boundary between two cells (such heads are said to be *crossing bounded*). Multiple input tapes are also commonly considered.

The result we report here is that there are three natural classes of automata C_1, C_2, and C_3 such that any function computed by a machine in C_1, C_2, or C_3 is invertible.

The classes C_1, C_2, and C_3 lie along a spectrum with C_1 having powerful storage facilities and weak access to the input data, and class C_3 having unrestricted access to the input and only weak storage structures.

class	**access to input**	**storage structure**	**bells and whistles**
C_1	one-way	logspace worktape, pushdown store	extra one-way input tapes
C_2	k-crossing bounded	logspace worktape	extra one-way input tapes, can write on input tape
C_3	unrestricted	m counters, each l-reversal bounded	extra one-way input tapes.

Theorem 4.3: Let f be an honest function computed by a machine in C_1, C_2, or C_3. Then f has an inverse which is computable in polynomial time.

We also present evidence that Theorem 4.3 cannot be extended to other natural classes of machines. We consider a large number of classes of machines. For each class C considered (and, we believe, for essentially every class which can be defined using those storage structures and types of access to input data which have been considered in the literature), we are either able to present a machine in C computing an honest function whose inverse is NP-hard, or we are able to present a subexponential-time algorithm for inverting any honest function computed by a machine in C, and we indicate why it is unlikely that a faster general method will be found.

We actually prove a stronger version of Theorem 4.3: we show that any honest function computed by a machine in any of the three classes has an inverse which can be computed very quickly in parallel. In order to present those results, let us first review the theory of the complexity of parallel computation.

Parallel Computation

With the advent of VLSI, it has become feasible to construct computers which exhibit massive parallelism; chips with thousands of processors are no longer unimaginable. Motivated by the possibility of so much parallelism, complexity theory has picked up the question of determining what class of problems can be solved much more quickly in parallel than on sequential computers.

A great many different models of parallel computation have been proposed and studied (see, e.g., [Co-81, Co-83, Vi-83] for a survey), but when minor differences in the models are ignored, there are basically two different categories into which all those models fall: those based on general-purpose parallel computers and those based on circuits.

General-Purpose Parallel Computers

Sequential computation on "real" computers is often modelled by RAM's (see, e.g. [AHU-74]). What could be more natural than to use several RAM's which can communicate in some way as a model for parallel computation? Models such as SIMDAG's [Go-82], WRAM's [CSV-84], HMM's [Co-81], etc. [Vi-83] are essentially just that; they differ only in the way that the RAM's are connected.

For example, a SIMDAG consists of an infinite sequence of parallel processors P_1, P_2, ..., each of which can access an infinite "global" (shared) memory, as well as having an infinite "local" (private) memory. A CPU contains the program and broadcasts instructions to the first k processors, where k is stored in a register in the global memory. Each P_i has the index i stored in a special register in its local memory, and thus the P_i may be accessing different memory locations, even though they are all executing the same instruction. More complete definitions may be found in [Go-82].

The SIMDAG models "single-instruction stream, multiple-data stream" (SIMD) parallelism, to use the terminology of [Fl-66]; the WRAM models "multiple-instruction stream, multiple-data stream" (MIMD) parallelism. Instead of having a single CPU broadcasting instructions, a WRAM has an infinite sequence of processors, where each processor executes its own copy of the program; processors may be executing different parts of the program at any given time. As in the SIMDAG, communication is through a shared memory.

There are a number of different varieties of WRAM-like models which differ primarily in how read and write conflicts are handled. These models are discussed in [Vi-83].

Hardware Modification Machines (HMM's) differ from SIMDAG's and WRAM's in that processors communicate via an interconnection network which is constructed during the course of the computation, instead of through a shared memory. (In addition, the processors in HMM's are finite state machines, instead of RAM's.)

Note that in all of these models, individual machines differ only in the program they execute; that is, the only difference between two SIMDAG's is the program stored in the CPU. Thus, any parallel algorithm can be executed on a SIMDAG simply by changing programs. In that sense, all of these models are general-purpose parallel computers.

Fortunately, all of these models are approximately equivalent, in the following sense: if a given problem can be solved on one of these models in time $T(n)$ using $P(n)$ processors, then it can be solved on any of the other models in time $T(n)^{O(1)}\log^{O(1)}P(n)$ using $P(n)^{O(1)}T(n)^{O(1)}$ processors. (Some models correspond much more closely than this suggests.) These models of parallel computation are thus approximately equivalent in the same sense in which models of sequential computation such as Turing machines and RAM's are approximately equivalent: fundamental classes of problems, such as the class of problems solvable in polynomial time, remain the same regardless of which model is used.

In modelling parallel computation, we wish to model the situation in which the number of processors is greater than the length of the input; however it is clear that any physically-realizable parallel computer will have to have a "feasible" number of processors. For this reason, much complexity-theoretic research on parallelism has focused on the case in which the number of processors is bounded by a polynomial in n, that is, $\mathrm{P}(n) = n^{\mathrm{O}(1)}$. If we also restrict $\mathrm{T}(n)$ to be bounded by a polynomial in n, then we have that if a given problem can be solved on one model of a general-purpose parallel computer in time $\mathrm{T}(n)$ using $n^{\mathrm{O}(1)}$ processors, then it can be solved on any of the other models in time $\mathrm{T}(n)^{\mathrm{O}(1)}\log^{\mathrm{O}(1)}n$ using $n^{\mathrm{O}(1)}$ processors.

Circuits

Since VLSI technology provides motivation for the study of large-scale parallelism, it is natural to model that sort of parallelism using circuits. A circuit is a collection of AND, OR, and NOT gates and input and output nodes, along with an acyclic interconnection network linking the gates to each other and to the input and output nodes. A function f is computed by a family $\{C_n : n \geq 1\}$ of circuits iff each circuit C_n has n input nodes and for every input x of length n, C_n outputs $f(x)$ when given input x. Some authors consider circuits of unbounded fan-in and fan-out, while others restrict their attention to circuits of fan-in 2. Sometimes primitive operations other than AND, OR, and NOT are considered. Perturbing the model in this way does not affect complexity results very much; for instance a circuit of size $\mathrm{S}(n)$ with unbounded fan-in and fan-out can be replaced by an equivalent circuit with fan-in and fan-out 2, where the size and depth of the new circuit are larger than the original circuit only by a factor of $\log \mathrm{S}(n)$.

A significant difference between circuit-based complexity and machine-based complexity is that some arbitrarily complex and even undecidable sets have trivial circuit complexity. For example, let S be any undecidable subset of $\{0\}^*$. S is accepted by the circuit family $\{C_n\}$ where C_n accepts 0^n if $0^n \in S$, and accepts nothing otherwise.

One way around this problem is to restrict attention to families of circuits which can be effectively or efficiently constructed. Since this amounts to requiring that there be a uniform method of constructing the members of a family of circuits, this is called a *uniformity condition*: $\{C_n\}$ is a *DSPACE (S(n))-uniform (DTIME(T(n))-uniform) family of circuits* if the function $n \rightarrow C_n$ is computable in space $S(n)$ (time $T(n)$).

Work on uniform circuit complexity has usually concentrated on the computation done *by the circuit*, and effort has been expended to make sure that the computation of the function $n \rightarrow C_n$ does not "overpower" the computation done by the circuit C_n. Thus Borodin and others [Bo-77, Pi-79, Co-79] considered DSPACE($\log S(n)$)-uniform circuits of size $S(n)$; notice that *any* function whose output has size $S(n)$ requires at least $\log S(n)$ space to compute. Since most attention has been on circuits of "feasible" size, i.e., C_n has size $n^{O(1)}$, these studies of uniform circuit complexity have concentrated on logspace-uniform families of circuits.

One pleasant aspect of the logspace-uniformity condition is that time and number of processors on a general-purpose parallel computer correspond to depth and size of logspace-uniform circuits.

Proposition 1.5: (See, e.g., [Ho-80].)

Let $T(n)$ be bounded by a polynomial in n. Then a function can be computed on a SIMDAG using $n^{O(1)}$ processors in time $T(n)^{O(1)}\log^{O(1)}n$ iff it can be computed by a logspace-uniform family of circuits $\{C_n\}$ of size $n^{O(1)}$ and depth $T(n)^{O(1)}\log^{O(1)}n$.

Thus the two major competing models of parallel computation, general-purpose parallel computers and circuits, are very closely linked. This can be taken as evidence that both models are essentially "correct," and that, e.g., problems with very fast feasibly-parallel solutions on one model will also have fast parallel solutions on any other reasonable model of parallel computation. As further evidence that the classification of the complexity of problems using these models is robust, we offer the following.

Proposition 1.6: Let T(n) be bounded by a polynomial in n. Then

SIMDAG Processors ($n^{O(1)}$) Time ($T(n)^{O(1)}\log^{O(1)}n$)

= logspace-uniform circuit Size ($n^{O(1)}$) Depth ($T(n)^{O(1)}\log^{O(1)}n$) [Ho-80]

= Turing machine Time ($n^{O(1)}$) Reversal ($T(n)^{O(1)}\log^{O(1)}n$) [Pi-79]

= Alternating Space ($n^{O(1)}$) Time ($T(n)^{O(1)}\log^{O(1)}n$) [Ru-81]

= Alternating Space ($n^{O(1)}$) Alternation ($T(n)^{O(1)}\log^{O(1)}n$) [Ru-81]

= Alternating Space ($n^{O(1)}$) Treesize ($T(n)^{O(1)}\log^{O(1)}n$) [Ru-80]

= NAuxPDA Space ($n^{O(1)}$) Time ($T(n)^{O(1)}\log^{O(1)}n$) [Ru-80]

= DAuxPDA Space ($n^{O(1)}$) Time ($T(n)^{O(1)}\log^{O(1)}n$) [Ru-80]

(Concepts such as alternating Turing machines and auxiliary pushdown automata will be discussed later.)

Two other models of parallel computation, conglomerates [Go-82] and aggregates [DC-80] have also been studied. They are similar to circuits, except the interconnection networks need not be acyclic, and nodes in the networks are RAM's instead of gates. An interesting feature of the aggregate model is that networks of size less than n can be considered. Aggregates and conglomerates with logspace-uniform networks of size $n^{O(1)}$ compute the same functions in time $T(n)^{O(1)}\log^{O(1)}n$ as logspace-uniform circuits of size $n^{O(1)}$ and depth $T(n)^{O(1)}\log^{O(1)}n$.

<u>NC</u>

The class NC has has received a great deal of attention recently. NC is intended to model the class of problems for which essentially optimal parallel algorithms exist.

Note that if a function can be computed in parallel using $P(n)$ processors in time $T(n)$, then the problem can be solved on a sequential computer in time aproximately $T(n)P(n)$. Thus, if the fastest sequential algorithm for a problem requires time $T(n)$, the fastest parallel solution using $P(n)$ processors one can hope for is $T(n)/P(n)$.

If a problem can be solved in time $n^{O(1)}$ on a Turing machine, and one wishes to find a fast parallel algorithm for the problem using $n^{O(1)}$ processors, in principle, one could hope to find a parallel solution which runs in $O(1)$ time. While that makes sense on some models of parallel computation (e.g., SIMDAG's and unbounded fan-in circuits), on some other models, such as circuits of fan-in 2, very little can be computed in time less than $\log n$. Defining a class of problems to be the class which can be solved in time $\log n$ using $n^{O(1)}$ processors runs the risk of being overly dependent on the particular model of parallel computation which is being used. However, as Proposition 1.6 shows, a "fudge factor" of $\log^{O(1)} n$ is sufficient to cover over any idiosyncrasies of the individual models of parallel computation. It is for these reasons that NC is defined as the class of functions which can be computed by logspace-uniform families of circuits of size $n^{O(1)}$ and depth $\log^{O(1)} n$.

The name "NC" stands for "Nick's Class." The name was proposed by Cook [Co-79] in recognition of the contributions of [Pi-79].

PUNC

The discussion thus far has been intended to motivate the definition of NC and to review some relevant results. At this point, however, we wish to present evidence that NC inadequately models the notion of "feasible parallelism"; this will in turn motivate the definition of a new complexity class, PUNC. Results presented in this thesis show that PUNC has many of the same pleasing theoretical properties which have been used to motivate NC.

NC has very pleasant characterizations in terms of the various models of general-purpose parallel computers which have been proposed. However, those models all share one curious feature. As has been observed before [Go-82, Co-81, DC-80], the various models of general-purpose parallel computers can be thought of as building their own interconnection networks during the course of the computation. In the characterization of NC in terms of HMM's, this is particularly evident. NC can thus be viewed as the class of problems for which fast "self-organizing" feasibly-parallel solutions exist. We argue that the "self-organizing" condition is an unnatural restriction.

Consider the characterization of NC in terms of circuits. We argue that the logspace-uniformity condition of NC is unnatural. In modelling the class of problems for which extremely fast circuits can be built, it makes sense to consider families of circuits $\{C_n\}$ such that the function $n \rightarrow C_n$ is feasible; i.e., we wish to model computation by circuits such that there is some method of building circuits for inputs of size n in a reasonable amount of time. The natural way to model this in complexity-theoretic terms is P-uniformity: the family of circuits $\{C_n\}$ is P-uniform iff the function $n \rightarrow C_n$ is computable in time polynomial in n. (Comments similar to these were made in [BCH-84, vzG-84], where P-uniform circuits of depth $\log n$ and $\log^2 n$ were presented for a variety of problems.) This gives rise to the class P-Uniform NC (PUNC), the class of functions for which there exists a P-uniform family of circuits $\{C_n\}$ of size polynomial in n and of depth $\log^{O(1)} n$.

PUNC has not been studied before (although P-uniform circuits were considered in [BCH-84, vzG-84]), and the following reasons may partly explain why. First, NC has many alternate characterizations, and the study of NC has given rise to a pleasing theoretical structure [Co-83]. This can be taken as evidence that NC is the "right" setting in which to study parallelism. Second, many consider uniformity conditions to be ungainly; for instance, Cook [Co-81], in discussing HMM's, cites as an advantage the fact that the HMM model has no uniformity condition, and Ruzzo [Ru-81] cites as undesirable the situation in which the circuit constructor is more powerful than the circuit. It may have been assumed that P-

Uniform NC was an awkward concept, a class unlikely to have pleasing alternate characterizations. In Chapter 3, we give alternate charactizations of PUNC in terms of auxiliary pushdown automata and alternating Turing machines.

Recall that NC has a very appealing characterization in terms of general-purpose parallel computers. One might suppose that, because so much power has been placed in the pre-processing phase, problems in PUNC might be solvable only by "special-purpose" chips. However, in Chapter 3 we present a natural model of general-purpose parallel computation on which PUNC is the class of problems solvable using $n^{O(1)}$ processors in $\log^{O(1)}n$ time.

We also prove results about the relationship between PUNC and NC; for instance:

Theorem 3.7: NC = PUNC $\Leftrightarrow$ every tally language in P is in NC.

(A *tally language* is a subset of $\{0\}^*$.) This leads to results about exponential-time complexity classes.

These results also clarify the complexity of one-way auxiliary pushdown automata, which have been studied before in [Br-77a, Br-77b, Ch-77, WB-79, We-80]. Some restrictions of AuxPDA's have been shown not to affect severely the complexity of the languages they accept; in [Ga-77] a two-way deterministic (one-head) PDA is presented which accepts a language which is hard for P under logspace reductions. Some other restrictions have been shown to be more limiting; in [Ki-81a, We-79] it was shown that logspace-bounded AuxPDA's whose pushdowns make at most a constant number of turns accept only languages in NLOG, and Ruzzo [Ru-81] showed that AuxPDA's which run in time $2^{\log^{O(1)}n}$ accept only languages in NC. Here we show that AuxPDA's which move their input heads at most $2^{\log^{O(1)}n}$ times accept exactly the languages in PUNC, and thus it seems unlikely that any such machine accepts a language which is hard for P.

CHAPTER II

Preliminaries

The reader is assumed to be familiar with deterministic and nondeterministic Turing machines and with the basic notions of time and space complexity. A good introduction to this material can be found in [HU-79].

Alternating Turing machines are a generalization of nondeterministic Turing machines. Configurations of nondeterministic Turing machines are analogous to existential quantification, since a configuration is accepting if there *exists* an accepting computation starting from that configuration. Alternating Turing machines result when configurations analogous to universal quantification are also considered; a configuration is either universal, in which case it accepts iff *all* computations rooted at that configuration accept, or it is existential. The term "alternation" refers to the alternation between universal and existential configurations which occurs along the computation paths.

Alternation has proved to be a useful concept in complexity theory, particularly in explicating the difference between time and space complexity; for instance, P is equal to the class of languages which can be accepted by logspace-bounded alternating Turing machines [CKS-81]. Further results and more precise definitions relating to alternation may be found in [CKS-81].

Auxiliary pushdown automata (AuxPDA's) are very closely related to alternating Turing machines, and were studied before the concept of alternation had been formulated. An AuxPDA is a space-bounded Turing machine with a pushdown store; the space bound does not apply to the pushdown store. Space and time on an AuxPDA correspond closely to space and time on an alternating Turing machine; a language is accepted by an alternating Turing machine in space $S(n)$ and time $T(n)^{O(1)}$ iff it is accepted by a (deterministic or nondeterministic) AuxPDA in space $S(n)$ and time $2^{T(n)^{O(1)}}$ [Ru-80]. We may use the term NAuxPDA (DAuxPDA) to refer to nondeterministic (deterministic) auxiliary pushdown automata.

A *circuit* for inputs of size n is a finite collection of AND, OR, and NOT gates, output gates, and n input nodes, along with an acyclic interconnection network linking the gates to each other and to the input and output nodes. We will not distinguish between a circuit and its description in some suitable description language. The *size* of a circuit is the number of gates it contains. The *depth* of a circuit is the length of the longest path in the network from an input node to an output node.

A *family of circuits* is a set $\{C_n \mid n \in \mathbb{N}\}$ where C_n is a circuit for inputs of size n. $\{C_n\}$ is a *DSPACE(S(n))-uniform (DTIME(T(n))-uniform)* family of circuits if the function $1^n \rightarrow C_n$ is computable on a Turing machine in space $S(n)$ (time $T(n)$).

$\{C_n\}$ *computes the function* f: $\{0,1\}^* \rightarrow \{0,1\}^*$ if, for every word w of length n, the output nodes of C_n take on the values $f(w)$ when the input nodes take on the values of w. $\{C_n\}$ is a *family of circuits for* $L \subseteq \{0,1\}^*$ if $\{C_n\}$ computes the characteristic function for L.

Background, and more detailed discussions of circuit complexity, may be found in [Bo-77, Ru-81].

If w is a string, then w^i denotes the string which consists of i copies of w; thus $0^4 = 0000$. The length of a string w is denoted by $|w|$. The cardinality of a set S is denoted by $|S|$. For any real number x, $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x, and $\lceil x \rceil$ denotes the least integer greater than or equal to x. All logarithms are taken base two.

When referring to classes of functions, we make frequent use of "big O" abbreviations. For example, we write $f(n) = \log^{O(1)}n$ iff $f(n) \leq k\log^k n + k$ for some k, $f(n) = 2^{O(n)}$ iff $f(n) \leq$ $f(n) \leq k2^{kn} + k$ for some k, etc. In addition, $f(n) = o(g(n))$ iff for all integers k there exists an n_k such that $n > n_k \Rightarrow kf(n) < g(n)$, and $f(n) = \omega(g(n))$ iff for all integers k there exists an n_k such that $n > n_k \Rightarrow f(n) > kg(n)$.

A language L is *sparse* if $|\{w \in L \mid n \geq |w|\}|$ is $n^{O(1)}$; L is a *tally language* if $L \subseteq \{0\}^*$.

In order to use strings in {0,1} to represent numbers and vice-versa, we use the standard method of letting the string w denote the number whose binary representation is $1w$. Thus the empty string denotes 1, "0" denotes 2, etc. We shall often refer to languages $L \subseteq \{0, 1, \#\}^*$. This is merely a notational convenience; such an L should be thought of as a subset of $\{00, 11, 01\}^*$.

A *configuration* of a Turing machine M is a tuple (q,w,i,j,l,a,b), where q is a state of M, w is a string of worktape symbols (the *worktape contents*), i is an integer (the *input head position*), j is an integer (the *output head position*), l is an integer, $1 \leq l \leq |w|$ (the *worktape head position*), a is the i-th input symbol, and b is the j-th output symbol. As usual, the relation $C_1 \vdash C_2$ holds if M, in configuration C_1, can enter C_2 in one move. $\vdash^*$ is the reflexive, transitive closure of $\vdash$.

We will use the following abbreviations.

P = DTIME($n^{O(1)}$)

NP = NTIME($n^{O(1)}$)

EXPTIME = DTIME($2^{O(n)}$)

NEXPTIME = NTIME($2^{O(n)}$)

DLOG = DSPACE($\log n$)

NLOG = NSPACE($\log n$)

PSPACE = DSPACE($n^{O(1)}$)

AuxPDA(S(n)) = the class of languages acepted by auxiliary pushdown automata with worktape bounded by S(n)

DTIME,SPACE(T(n),S(n)) = the class of languages accepted by deterministic Turing machines which operate simultaneously in space S(n) and time T(n)

ASPACE,TIME(S(n),T(n)) = the class of languages accepted by alternating Turing machines which operate simultaneously in space S(n) and time T(n)

ASPACE,ALTERNATION(S(n),T(n)) = the class of languages accepted by alternating Turing machines which operate in space S(n) where the number of alternations between universal and existential states along any computation path is bounded by T(n).

ASPACE,TREESIZE(S(n),T(n)) = the class of languages accepted by alternating Turing machines which operate in space S(n), such that, if an input of length n is accepted, it is accepted by an accepting computation tree of size T(n).

AuxPDASPACE,TIME(S(n),T(n))= the class of languages acepted by auxiliary pushdown automata which operate in time T(n) with worktape bounded by S(n)

C-uniform SIZE,DEPTH(S(n),D(n)) = the class of languages for which there exists a **C**-uniform family of circuits of size bounded by S(n) and depth bounded by D(n), where **C** is some time- or space-complexity class.

NC = DLOG-uniform SIZE,DEPTH($n^{O(1)}$,$\log^{O(1)} n$)

SC = DTIME,SPACE($n^{O(1)}$,$\log^{O(1)} n$)

(SC stands for "Steve's Class"; SC was named by Pippenger (see [Ru-81]) in recognition of the contributions of [Co-79].)

Note that NC is defined as a class of languages. It is sometimes convenient to consider NC to be a class of functions; a function f is in NC iff there is a logspace-uniform family of circuits computing f.

Proposition 2.1: [Co-83] A function f is in NC iff $\{i\#x \mid \text{the } i\text{-th bit of } f(x) \text{ is } 1\}$ is in NC.

Some relevant facts about these complexity classes are listed below:

Proposition 2.2:

P	= ASPACE($\log n$)	[CKS-81]
	= AuxPDA($\log n$)	[Co-71]
	= DLOG-uniform SIZE,DEPTH($n^{O(1)}, n^{O(1)}$)	[Pi-79]
	= P-uniform SIZE,DEPTH($n^{O(1)}, n^{O(1)}$)	[Sa-72]
EXPTIME	= ASPACE(n)	[CKS-81]
	= AuxPDA(n)	[Co-71]
	= DSPACE(n)-uniform SIZE,DEPTH($2^{O(n)}, 2^{O(n)}$)	[Bo-77, PF-79, Ru-81]
	= EXPTIME-uniform SIZE,DEPTH($2^{O(n)}, O(n)$)	
NC	= ASPACE,TIME($\log n, \log^{O(1)} n$)	[Ru-81]
	= ASPACE,ALTERNATION($\log n, \log^{O(1)} n$)	[Ru-81]
	= ASPACE,TREESIZE($\log n, 2^{\log^{O(1)} n}$)	[Ru-81]
	= AuxPDASPACE,TIME($\log n, 2^{\log^{O(1)} n}$)	[Ru-81]

DSPACE($\log^{O(1)} n$) = ATIME($\log^{O(1)} n$)   PSPACE = ATIME($n^{O(1)}$)   [CKS-81]

CHAPTER III

Characterizations of Parallel Complexity Classes

The class of problems (languages) for which extremely fast parallel algorithms can be efficiently constructed is a class of some interest. As outlined in the introduction, PUNC is an attempt to capture this class in complexity-theoretic terms.

Although PUNC is defined as a class of languages, we may also say that a function f is in PUNC. It is clear what is meant by this.

PUNC is a robust class in the sense that it is not overly dependent upon idiosyncracies of the circuit model. In particular, if we allow circuits with unbounded fan-in, or if we consider P-uniform networks of RAM's or finite-state machines, the same class of languages results. Similarly, we could have defined PUNC in terms of aggregates [DC-80] or conglomerates [Go-82] with P-uniform interconnection networks.

Theorem 3.1:

L is in PUNC iff

L is accepted by an aggregate with a P-uniform interconnection network iff

L is accepted by a conglomerate with a P-uniform interconnection network.

PUNC is also closed under a broad class of reducibilities.

Theorem 3.2:

PUNC is closed under logspace reductions (defined in [Jo-75]) and NC_1 reductions (defined in [Co-83]).

If f is in PUNC, L is in PUNC, and for all w, $w \in L' \Leftrightarrow f(w) \in L$, then L′ is in PUNC.

Thus PUNC has a robust definition in terms of circuit-based models of parallel computation. In the rest of this chapter we will present characterizations of PUNC in terms of some sequential models of computation, and also in terms of general-purpose parallel computers. Results about the relationships which hold between PUNC and other complexity classes will also be presented.

Auxiliary Pushdown Automata and Alternating Turing Machines

In order to consider alternating Turing machines of sublinear time complexity (which is necessary in order to characterize NC in terms of alternation) a special "random-access" feature has to be contrived which allows alternating Turing machines to access specified bits of the input in unit time. This is a powerful feature, and it makes sense to restrict its use. In this section, we show that such a restriction in fact gives one way to characterize PUNC. Another characterization is given by restricting how often AuxPDA's may move their input heads.

Theorem 3.3: The following are equivalent:

(i) L ∈ PUNC

(ii) L is accepted by a logspace-bounded deterministic AuxPDA which moves its input head $O(2^{\log^{O(1)} n})$ times.

(iii) L is accepted by a logspace-bounded nondeterministic AuxPDA which moves its input head $O(2^{\log^{O(1)} n})$ times.

Proof:

(i) ⇒ (ii): Let L be accepted by a P-uniform family $\{C_n\}$ of circuits of depth $\log^{O(1)}n$. Since the function $n \rightarrow C_n$ is computable in polynomial time, the language $L' = \{0^n 1 r b \mid$ the r-th bit of C_n is $b\}$ is in P, and is thus accepted by some logspace-bounded deterministic AuxPDA. Thus, assuming a suitable encoding for circuits, one can easily see that a logspace-bounded AuxPDA can carry out the following computation *without moving its input head*: start with n and g written on the worktape, where g is the name of a gate in C_n, and find the inputs to g by obtaining the bits of C_n one-by-one. Now let M be the logspace-bounded AuxPDA which, on input w of length n, executes the following algorithm:

```
(1) Write $n$ in binary on the worktape.          ($O(n)$ input head moves)
(2) $g :=$ the output gate for $C_n$.             (0 input head moves)
(3) call EVALUATE ($g$)

EVALUATE($g$)
    if $g$ is an input gate for input $i$
        then read input position $i$              ($O(n)$ input head moves)
        else let $g_1$ and $g_2$ be the inputs to $g$
            store $g_2$ on the stack
            $t_1 :=$ EVALUATE ($g_1$)
            put $g_2$ on the worktape and store $t_1$ on the stack
            $t_2 :=$ EVALUATE ($g_2$)
            use $t_1$ and $t_2$ to get the value of $g$
```

It is easy to verify that for circuits of depth r, the number of head moves performed by the algorithm is $O(n + n2^r)$. Since the depth of C_n is $\log^{O(1)}n$, the number of input head moves performed by M is $2^{\log^{O(1)}n}$.

(iii) ⇒ (i): Let L be accepted by M, a nondeterministic logspace-bounded AuxPDA which moves its input head at most $2^{\log^{O(1)}n}$ times on inputs of length n. As Mager showed in [Ma-69], we may assume without loss of generality that the height of the pushdown of M is always $\leq n^l$ for some l.

Let us define a *surface configuration* of M to be a 5-tuple (q,x,i,Γ,a), where q is a state of M, x is a string of worktape symbols (the *worktape contents*), i is an integer, $1 \leq i \leq |x|$ (the *worktape head position*), Γ is a pushdown symbol (the *stack top*), and a is an input symbol. We now define a binary relation $\leadsto$ between surface configurations: if $C_1 = (q,x,i,\Gamma,a)$ and $C_2 = (p,y,j,\Gamma,a)$ are surface configurations, then $C_1 \leadsto C_2$ iff M, when started in state q with Γ on its pushdown, x on its worktape, its worktape head on the i-th symbol of x, and its input head scanning an a, can make some sequence of moves *without moving its input head* ending with that same Γ on top of the stack with M in state p with worktape contents y, and worktape head position j. Surface configurations and relations similar to $\leadsto$ have often been used before (e.g., [AHU-68, Co-71, Ki-81b]); the primary difference between the relation $\leadsto$ and e.g. the relation which was defined in [Co-71] is that here we are only concerned with pairs of surface configurations which can be connected by a computation which leaves the input head fixed.

Let $W(n)$ be the set of all surface configurations whose worktape contents have length $\leq \log n$. There is a polynomial-time algorithm which, on input 1^n, writes down all elements of $W(n)$ and then outputs all pairs of surface configurations (C_1,C_2) such that $C_1 \leadsto C_2$. (A simple variant of the algorithm used in [Co-71] will suffice.) Let us denote by $h(n)$ the output of this program on input 1^n.

It is now a simple matter to construct a nondeterministic AuxPDA M′ which, on inputs of the form $w\#h(|w|)$, runs in time $2^{\log^{O(1)} n}$ and accepts iff M accepts w. M′ uses the following algorithm:

```
begin
    C := the initial surface configuration of M on input $w$
    while C is not a halting configuration
        Choose to either
            (a) set C := D, where (C,D) appears in the string to the right of the #, or
            (b) set C := D where D is the surface configuration of M after executing some
                move which is legal from C
    endwhile
    accept iff C is an accepting state
```

To see that M′ runs in time $2^{\log^{O(1)} n}$ on inputs of the form $w\#h(|w|)$, note that there will always be an accepting computation of M′ in which, between any two simulated moves of M in which M moves its input head, all "pop" moves are performed before any "push" moves are performed. (A "push" followed by a "pop" with no intervening moves of the input head can be simulated by choosing option (a).) Since the height of the stack is bounded by n^l and since the number of moves which can be performed without manipulating the stack is bounded by some polynomial $p(n)$, the number of moves performed by M′ between simulated moves of M in which M moves its input head is no more than $2p(n)n^l$. Since M moves its input head $2^{\log^{O(1)}n}$ times, the running time of M′ on inputs of the form $w\#h(|w|)$ is no more than $2p(n)n^l 2^{\log^{O(1)}n} = 2^{\log^{O(1)}n}$.

Now by Theorems 1 and 2 of [Ru-80], there is an alternating logspace-bounded Turing machine which simulates M′, and on inputs of the form $w\#h(|w|)$ runs in time $\log^{O(1)}n$. Clearly, this machine can be modified to run in time $\log^{O(1)}n$ on *all* inputs. Let L′ be the language accepted by this machine. L′ ∈ NC, and thus there is a logspace-uniform family $\{D_n\}$ of circuits for L′. Let $r_n = |w\#h(|w|)|$ for any string w of length n. Now $\{C_n\}$ is a P-uniform family of circuits for L, where C_n is constructed by computing $h(n)$ and D_{r_n}, and "hard-wiring" $h(n)$ into D_{r_n}. □

It is known that the relationship between alternating time and logspace-uniform circuit depth is very close; letting NC^k denote the class of languages accepted by logspace-uniform circuits of depth $O(\log^k n)$ it was shown in [Ru-81] that NC^k $\doteq$ ASPACE,TIME($\log n$, $\log^k n$). for all $k \geq 2$. Unfortunately, equality does not seem to hold for $k = 1$. This is because deterministic logspace seems to be more powerful than alternating log time, and thus a logspace-computable function which constructs a circuit cannot be simulated in alternating log time.

Ruzzo considered a number of uniformity conditions in [Ru-81], and showed that the class NC remained the same even when defined in terms of much stronger uniformity conditions than logspace uniformity (i.e., conditions which require not only that the functions

$n \rightarrow C_n$ be computable in logspace, but that they be "easily" logspace computable). The corresponding classes NC^k seem sensitive to the uniformity condition, however. In particular, he considered a uniformity condition which we will call Alogtime-uniformity, which essentially requires that an alternating Turing machine be able to recognize pieces of the interconnection network of C_n in $\log n$ time, and he showed that Alogtime-uniform NC^k = ASPACE,TIME($\log n$, $\log^k n$). for all $k \geq 1$.

What is going on in these results is that the part of the computation which constructs the circuits is being "overpowered" by the rest of the computation. In order to get sharp results, a very strong and somewhat artificial uniformity condition must be used. As we shall see, sharp results relating alternation and P-uniform circuit depth can be obtained by "factoring out" the part of the computation which constructs the circuits, instead of "overpowering" it. Before that correspondence can be shown, however, there is one difficulty which must be discussed.

It is clear that restricting motion of the input head is the natural way to restrict access to the input for an AuxPDA. It is less clear what the correct way is to restrict an alternating Turing machine. Simply restricting the number of accesses to the input along any computation path is not sufficient, since *any* alternating Turing machine which uses at least logspace can be simulated efficiently by a machine which never accesses its input more than *once* on any computation path. The reasonable ways of restricting access to the input seem to be to restrict the computation which *precedes* any access of the input tape, and to restrict the total number of nodes in the alternation tree which access the input.

Definition: $PUNC^k$ = P-uniform SIZE,DEPTH ($n^{O(1)}$, $O(\log^k n)$), for all $k \geq 1$.

Theorem 3.4:

$L \in PUNC^k \Leftrightarrow L$ is accepted by a logspace-bounded alternating Turing machine which accesses its input only during the first $O(\log^k n)$ steps, for all $k \geq 1$.

Proof: (The proof of this theorem is an adaptation of the proofs of theorems 3 and 4 in [Ru-81].)

($\Rightarrow$) Let L be accepted by a P-uniform family $\{C_n\}$ of depth $O(\log^k n)$. L is accepted by the logspace-bounded alternating Turing machine M which operates as follows on inputs w of length n:

```
(1) Existentially guess $n$ and universally check the guess.
(2) $p :=$ the empty string   ($p$ represents a path in $\{left\text{-}input, right\text{-}input\}^*$ leading away
                               from the output gate of $C_n$.)
(3) $accept.bit := 1$
(4) repeat
        guess the type $t \in$ {AND, OR, NOT, INPUT} of the gate which is reached by
        following the path $p$ away from the output gate of $C_n$, and universally check the
        guess.
        If $t$ = AND, universally set $p := p.left\text{-}input$ and $p := p.right\text{-}input$.
        If $t$ = OR, existentially set $p := p.left\text{-}input$ and $p := p.right\text{-}input$.
        If $t$ = NOT, set $p := p.right\text{-}input$; $accept.bit := accept.bit + 1 \pmod 2$.
        If $t$ = INPUT, guess which input should be read and universally check the guess.
    until $t$ = INPUT
    Read the input and accept and halt iff the specified bit = $accept.bit$.
```

Since those parts of the loop in which M "universally checks a guess" can be performed without accessing the input, it is clear that M accesses the input only during the first $O(\log^k n) + O(\log n)$ steps.

($\Leftarrow$) Let L be accepted by a logspace-bounded alternating Turing machine M which accesses its input only during the first $T(n) = O(\log^k n)$ steps. We construct a circuit with gates labeled (t,α), where $0 \leq t \leq T(n)$, and α is a configuration of M of length $\leq \log n + 1$. The output gate is $(0,\alpha_0)$, where α_0 is the initial configuration of M. For each gate (t,α), connect as inputs all gates $(t+1,\beta)$ such that $\alpha \vdash \beta$. Replace all gates (t,α) where α is a configuration reading the i-th input by input gates reading the i-th input. Replace all gates (t,α) where $t > T(n)$ by a constant 1 or 0, depending on whether or not M accepts when started in configuration α (note that M, when started in configuration α, does not consult its input). Replace by a constant 0 all gates (t,α) in which α is a configuration of length $\log n + 1$. Each remaining gate (t,α) is either an AND gate or an OR gate, depending on whether α is a universal or an existential configuration. □

It is interesting to compare Theorem 3.4 to the characterization of SC given by Sudborough in [Su-83]. Sudborough considered both one-way and two-way loglogspace-bounded alternating Turing machines, and he showed that SC is the class of all problems which are logspace-reducible to languages accepted by one-way loglogspace-bounded alternating Turing machines. Thus both SC and PUNC are characterized in terms of space-bounded alternating Turing machines with restricted access to the input.

We close this section with two further characterizations of PUNC, the proofs of which follow easily using the techniques used here and the results proved in [Ru-80] relating AuxPDA's and alternating Turing machines.

Theorem 3.5: The following are equivalent:

(i) $L \in$ PUNC

(ii) L is accepted by a logspace-bounded alternating Turing machine which accesses its input only during the first $O(\log^{O(1)}n)$ alternations.

(iii) L is accepted by a logspace-bounded alternating Turing machine which, if it accepts an input, accepts via an alternation tree which contains $O(2^{\log^{O(1)}n})$ nodes which access the input.

A General-Purpose Parallel Computer for PUNC

NC is the class of problems solvable using $n^{O(1)}$ processors in time $\log^{O(1)}n$ on a SIMDAG, on a WRAM, and on almost all of the many approximately-equivalent models of parallel computation which have been proposed in the past few years. (For surveys, see [Co-81, Vi-83].) This may be taken as evidence that NC truly captures the notion of efficient parallel computation. However, we argued in Chapter 1 that PUNC, and not NC, better captures that notion, at least when one is trying to model the class of problems solvable efficiently by special-purpose chips. Since SIMDAG's, WRAM's, and the like are *general-purpose* parallel computers, one might suspect that NC models general-purpose parallel computation, and PUNC models special-purpose parallel computation. We show here that that is not the case.

Models of general-purpose parallel computers such as SIMDAG's and WRAM's all share the characteristic that they have infinitely many processors. Let us now take the position that it makes just as much sense to provide infinitely many bits, *where the bits are no harder to produce than are the processors*; i.e., the n-th bit can be produced in time polynomial in n. This section explores the consequences of taking that position.

A *SIMDAG augmented with a sequence s* consists of a SIMDAG along with an infinite sequence of read-only global registers $B_1, B_2, \ldots$, where each B_i contains the i-th bit of the sequence s. When counting the number of processors used by a SIMDAG during a computation, we include the number of registers B_i accessed during the computation. (Equivalently, we could let B_i reside in processor P_i.)

A sequence s is *P-printable* if the language $\{0^n \mid \text{the } n\text{-th bit of } s \text{ is } 1\}$ is in P, that is, if the n-th bit of s can be obtained in time polynomial in n.

The reader should already suspect that L is in PUNC iff L is accepted using $n^{O(1)}$ processors in time $\log^{O(1)}n$ on a SIMDAG augmented with a P-printable sequence. However, we will prove more. There is a universal P-printable sequence s_U such that L is in PUNC iff L is accepted using $n^{O(1)}$ processors in time $\log^{O(1)}n$ on a SIMDAG augmented with s_U. Furthermore, s_U has a natural and appealing definition.

For any language L, the *characteristic sequence for L*, s_L, is an infinite sequence of 0's and 1's such that the r-th character of s_L is 1 iff $r \in L$.

Now let U be any language complete for EXPTIME under log-lin reductions. (See, e.g., [St-74].) For a concrete example, let $U = \{M\#w\#^{l|w|} \mid M \text{ accepts } w \text{ in time } 2^{(l+1)|w|}\}$. The characteristic sequence s_U is P-printable; to obtain the n-th bit, see if $n \in U$ in time $2^{O(|n|)} = 2^{O(\log n)} = n^{O(1)}$.

Theorem 3.6: $L \in$ PUNC $\Leftrightarrow$ L is accepted using $n^{O(1)}$ processors in time $\log^{O(1)} n$ on a SIMDAG augmented with s_U.

Proof: ($\Leftarrow$) If L is accepted using n^k+k processors in time $\log^{O(1)} n$ time on a SIMDAG augmented with s_U, then clearly there is a SIMDAG M′ which, on all inputs of the form $w\#s_U(1)s_U(2)\ldots s_U(|w|^k+k)$, runs in $\log^{O(1)} n$ time and uses $n^{O(1)}$ processors and accepts iff $w \in$ L. M′ can be made to run in $\log^{O(1)} n$ time on all inputs and thus accepts a language $L' \in$ NC. Using a program that prints s_U and a program constructing NC circuits for L′, it is easy to construct PUNC circuits for L.

($\Rightarrow$) Let L be accepted by a PUNC family of circuits $\{C_n\}$. Clearly, there is a language $L' \in$ NC such that $w\#C_{|w|} \in L'$ iff $w \in L$. (This can be seen by consulting the proof of Theorem 3.3, or by a simple reduction to the "Shallow Circuit Value" problem defined in [Ru-81].) There is a Turing machine M which on input $n\#r$ will accept iff the r-th bit of C_n is 1, and M runs in time $n^{O(1)} = 2^{O(|n\#r|)}$. Now a SIMDAG augmented with s_U on input w of length n may simulate the SIMDAG which accepts L′ on input $w\#C_n$. Whenever the r-th bit from C_n is required, consult register $B_{M\#n\#r\#^{l|n\#r|}}$ for the appropriate constant l. Note that the number of registers B_i needed for inputs w of size n is bounded by a polynomial in n, since $|M\#n\#r\#^{l|n\#r|}|$ is $O(\log n)$. □

Theorem 3.6 says that there is a single general-purpose parallel computer on which all problems in PUNC can be solved quickly; thus this has somewhat the same flavor as the results presented in [GP-83, Vi-84], in which efficient general-purpose parallel computers are studied. Unfortunately, the practical utility of this result is limited since, as Rackoff has pointed out [Ra-85], the number of registers B_i needed is exponential in the size of the program M.

Theorem 3.6 makes obvious the connection between P-uniform circuits and characteristic sequences of languages in EXPTIME. Other closely-related concepts are P-recognizable real numbers (using the "standard left cut" definition in [Ko-83]) and P-printable sets (sets S such that the function $n \rightarrow S \cap \{w \mid n \geq |w|\}$ is computable in time polynomial in n [HY-84]). Also, P-printable sequences may be defined as sequences s such that, for some k and M_v, every prefix of s is in $K_v(\log n, n^k)$, where $K_v(\log n, n^k)$ is the "generalized Kolmogorov complexity" measure of [Ha-83]. For a related discussion, see [Ko-84].

PUNC is simply NC augmented by a feasible amount of precomputation. Rackoff [Ra-85] has observed that P-printable sequences can be used to formulate a complexity theory for sequential computation with precomputation. We discuss this further and present some new results in Chapter 7.

Tally Languages and Complexity Classes

The complexity of languages in PUNC is intimately connected with the complexity of tally languages in P. In this section we explore that relationship.

Theorem 3.7: NC = PUNC $\Leftrightarrow$ all tally languages in P are in NC $\Leftrightarrow$ $\{0^i \mid i \in U\}$ is in NC.

Proof: ($\Rightarrow$) Clearly, all tally languages in P are in PUNC. Thus if NC = PUNC, then all tally languages in P are in NC, and trivially $\{0^i \mid i \in U\}$ is in NC.

($\Leftarrow$) Assume $\{0^i \mid i \in U\}$ is in NC, and let L have a family of PUNC circuits $\{C_n\}$. Let M be a machine running in time $2^{(l+1)n}$ for some l which accepts input $n\#r$ iff the r-th bit of C_n is 1. That is, $M\#n\#r\#^{l|n\#r|} \in U$ iff the r-th bit of C_n is 1. Note that since $\{0^i \mid i \in U\}$ is in NC, an AuxPDA with $M\#n\#r\#^{l|n\#r|}$ written on its worktape can decide in $2^{\log^{O(1)} n}$ time if $M\#n\#r\#^{l|n\#r|} \in U$ and hence obtain the r-th bit of C_n.

Clearly, there is a language $L' \in$ NC such that $w\#C_{|w|} \in L'$ iff $w \in L$; let L′ be accepted by a logspace-bounded AuxPDA M′ which runs in time $2^{\log^{O(1)} n}$. L is accepted by a logspace-bounded AuxPDA M_1 which, on input w, simulates M′ on input $w\#C_{|w|}$. Whenever the r-th bit from $C_{|w|}$ is required, M_1 writes $M\#n\#r\#^{l|n\#r|}$ on its worktape, and in $2^{\log^{O(1)}n}$ time obtains the bit using the method outlined above. The total running time is $2^{\log^{O(1)}n}$. □

Ruzzo has recently proved some results with a flavor somewhat similar to this [Ru-85]. He considers U_B uniformity, where a circuit family $\{C_n\}$ is U_B-uniform if the function $n \rightarrow C_n$ is computable in space (depth of C_n) [Bo-77, Ru-81]. Among other results, he shows that if U_B-NC$^k \subseteq$ P for some $k>1$, fast simulations of space-bounded computations are possible.

Tally languages have often been studied in conjunction with sparse sets, and the techniques used here lead to some new results and observations about the class of sparse sets in P. This is discussed in Chapter 6.

Relationships between PUNC and other complexity classes may be clarified by making an analogy between "polynomial-level" complexity classes and "exponential-level" classes. DLOG, NLOG, and P are analogous in this sense to DSPACE(n), NSPACE(n), and EXPTIME, respectively. Recalling that NC = ASPACE,TIME($\log n, \log^{O(1)}n$) and SC = DTIME,SPACE($n^{O(1)}, \log^{O(1)}n$), we can define exponential-time analogs ENC = ASPACE,TIME($n, n^{O(1)}$) and ESC = DTIME,SPACE($2^{O(n)}, n^{O(1)}$). The following proposition is easily verified.

Theorem 3.8: ENC = AuxPDASPACE,TIME($n, 2^{n^{O(1)}}$)

= ASPACE,ALTERNATION($n, n^{O(1)}$)

= DSPACE(n)-uniform SIZE,DEPTH($2^{O(n)}$, $n^{O(1)}$)

The natural exponential-time analog of PUNC turns out to be EXPTIME itself. Let EUNC denote the class of all languages for which there exist EXPTIME-uniform circuits of size $2^{O(n)}$ and depth $n^{O(1)}$.

Theorem 3.9: EUNC = EXPTIME

Proof: Clearly EUNC $\subseteq$ EXPTIME. Now let L $\in$ EXPTIME. In time $2^{O(n)}$ there is time to write down all 2^n strings of size n, test each one for membership in L, and build a circuit of depth $\log n + \log 2^n = O(n)$ which accepts L. Thus EXPTIME $\subseteq$ EUNC. $\square$

The analogous complexity classes at the polynomial and exponential levels are diagrammed in Figure 3-1.

$$\begin{tikzcd}
\mathrm{DSPACE}(n) \arrow[r, dash] \arrow[drr, dash] & \mathrm{NSPACE}(n) \arrow[r, dash] & \mathrm{ENC} \arrow[r, dash] \arrow[dr, dash] & \mathrm{EUNC} = \mathrm{EXPTIME} \\
 & & \mathrm{ESC} \arrow[ur, dash] \arrow[r, dash] & \mathrm{PSPACE}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathrm{DLOG} \arrow[r, dash] \arrow[drr, dash] & \mathrm{NLOG} \arrow[r, dash] & \mathrm{NC} \arrow[r, dash] \arrow[drr, dash] & \mathrm{PUNC} \arrow[r, dash] & \mathrm{P} \\
 & & \mathrm{SC} \arrow[urr, dash] \arrow[rr, dash] & & \mathrm{DSPACE}(\log^{O(1)} n)
\end{tikzcd}$$

Figure 3-1: A line between two complexity classes indicates that the class on the left is contained in the class on the right. The following inclusions are also known:

$$\mathrm{P} \subseteq \mathrm{ENC} \cap \mathrm{ESC}$$
$$\mathrm{P} \subsetneq \mathrm{EXPTIME}$$
$$\mathrm{NLOG} \subsetneq \mathrm{DSPACE}(\log^{O(1)} n) \subsetneq \mathrm{DSPACE}(n)$$
$$\mathrm{NSPACE}(n) \subsetneq \mathrm{PSPACE}$$

Collapse of two classes at the "polynomial" level implies collapse of the corresponding exponential-time classes, because of the following result, of the type proved in [Bo-74]:

Theorem 3.10:

Let E_1 and E_2 be the exponential-time analogs of P_1 and P_2, respectively, where $E_1 \subseteq E_2$. Then $E_1 = E_2 \Leftrightarrow$ every tally language in P_2 is in P_1.

Collapse of complexity classes at the exponential level seldom implies collapse at the polynomial level. However, Theorems 3.7 and 3.10 yield the following corollary.

Corollary 3.11: NC = PUNC $\Leftrightarrow$ ENC = EXPTIME

That is, NC = PUNC is equivalent to the exponential-time analog of the NC = P question.

It is interesting to note in this regard that the exponential-time analog of the SC $\subseteq$ PUNC question has an affirmative answer. On the other hand, there is evidence that SC $\not\subseteq$ PUNC. In [Pi-81], Pippenger presents a pebbling argument as evidence that SC $\not\subseteq$ NC. Loosely translated, Pippenger's result shows that any logspace-bounded AuxPDA which tries to solve the "Narrow Circuit Value Problem" by determining the value of each gate in the circuit by determing the values of its predecessors must run for $\Omega(2^{n/\log^3 n})$ steps. In fact, Pippenger proves the (marginally) stronger result that any such AuxPDA must make $\Omega(2^{n/\log^3 n})$ moves which "consider the predecessors" of gates. If we assume that most such moves must involve access to the input, then Pippenger's argument can be taken as evidence that the Narrow Circuit Value Problem is not in PUNC.