P-Uniform Circuit Complexity

ERIC W. ALLENDER

Rutgers University, New Brunswick, New Jersey

Abstract. Much complexity-theoretic work on parallelism has focused on the class NC, which is defined in terms of logspace-uniform circuits. Yet P-uniform circuit complexity is in some ways a more natural setting for studying feasible parallelism. In this paper, P-uniform NC (PUNC) is characterized in terms of space-bounded AuxPDAs and alternating Turing Machines with bounded access to the input. The notions of general-purpose and special-purpose computation are considered, and a general-purpose parallel computer for PUNC is presented. It is also shown that NC = PUNC iff all tally languages in P are in NC; this implies that the NC = PUNC question and the NC = P question are both instances of the ASPACE($S(n)$) = ASPACE,TIME($S(n)$, $S(n)^{O(1)}$) question. As a corollary, it follows that NC = PUNC implies PSPACE = DTIME($2^{n^{O(1)}}$).

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata* (*e.g., finite, push-down, resource-bounded*); *bounded-action devices* (*e.g., Turing machines, random access machines*); *relations among models*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*alternation and nondeterminism*; *parallelism*; *relations among modes*; F.1.3 [**Computation by Abstract Devices**]: Complexity Classes—*complexity hierarchies*; *reducibility and completeness*; *relations among complexity classes*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*classes defined by resource-bounded automata*

General Terms: Theory

Additional Key Words and Phrases: Alternating Turing machine, auxiliary pushdown automata, circuit complexity, exponential time, NC, P, precomposition, PSPACE, sparse sets, tally sets, uniformity

1. *Introduction*

With the advent of very large scale integrated (VLSI), it has become feasible to construct computers that 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. Much complexity–theoretic work in this area has focused on the class NC, the class of all languages for which there exists a logspace-uniform family of circuits $\{C_n\}$ of size polynomial in n and of depth $\log^{O(1)} n$. (Definitions of uniformity, circuits, etc. will be given in a later section.)

Ruzzo [40] has shown that the class NC remains the same if stronger notions of uniformity than logspace-uniformity are used. (See also [13] and [36] for more results concerning notions of uniformity.) That is, if we require not only that the

Portions of this work were supported by National Science Foundation grants MCS 81-03608 and CCR 88-10467.

Author's address: Department of Computer Science, Rutgers University, Hill Center, New Brunswick, NJ 08903.

functions $1^n \longrightarrow C_n$ be logspace-computable, but that they must be "easily" logspace-computable, there is no effect on the class NC. Indeed, if we define NC not in terms of circuits, but rather in terms of interconnected processors (RAMs or finite-state machines) we can do away with the uniformity condition entirely, as the characterizations of NC in terms of HMMs [19], SIMDAGs [25], WRAMs [15], etc. show. In this way, the "preprocessing" phase implicit in the logspace-uniformity condition for circuits is side-stepped. As has been observed before [19, 22, 25], these machines can be thought of as building their own interconnection network during the course of the computation. In the characterization of NC in terms of HMMs, 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.

Let us now take the view that it is okay to pack as much computational power into the preprocessing phase as is feasible. That is, we are interested in any problem for which a fast circuit family $\{C_n\}$ exists, with the only stipulation being that the function $n \longrightarrow C_n$ be feasible to compute. The natural formulation of this stipulation in complexity theory is P-uniformity [8]; the family of circuits $\{C_n\}$ is P-uniform if the function $n \longrightarrow C_n$ is computable in time polynomial in n. This gives rise to the class P-uniform NC (PUNC), the class of languages for which there exists a P-uniform family of circuits $\{C_n\}$ of size polynomial in n and depth $\log^{O(1)} n$.

PUNC has not been studied before (although P-uniform circuits of depth $\log n$ and $\log^2 n$ were considered in [8, 38, 43], and we list below some reasons that may partly explain why. At the same time, we present the contributions of this paper that, we believe, put PUNC on a more equal footing with NC.

First, NC has very nice characterizations in terms of general-purpose parallel computers such as SIMDAGs, WRAMs, etc. The fact that logspace-uniform circuit depth corresponds to parallel time on these machines has been taken as evidence that NC is the "right" setting in which to study parallelism. In fact, one might suppose that, because so much power has been placed in the preprocessing phase, problems in PUNC might be solvable only by "special-purpose" chips. However, in Section 5, we present a natural model of general-purpose computation on which PUNC is the class of problems solvable using $n^{O(1)}$ processors in $\log^{O(1)} n$ time.

Second, NC has many alternate characterizations in terms of other models of computation. For example, Ruzzo [40] has shown that NC can be characterized in terms of AuxPDAs and alternating Turing machines with simultaneous time and space bounds. It may have seemed unlikely that PUNC would have similar characterizations. Yet, in Section 4, PUNC is characterized in terms of AuxPDAs and alternating Turing machines with simultaneous bounds on space and access to the input. For instance, a language L is in PUNC iff it is accepted by a logspace-bounded AuxPDA that moves its input head $2^{\log^{O(1)} n}$ times.

Third, many researchers seem to have considered uniformity conditions to be inelegant and ungainly. For example, Cook [19] in discussing HMMs, cites as an advantage the fact that the HMM model has no uniformity condition, and Ruzzo [40] cites as undesirable the situation in which the circuit constructor is more powerful than the circuit. Uniformity conditions also cause some annoying difficulties when relating alternating time to circuit depth. Let NC^k denote the class of languages accepted by logspace-uniform circuits of depth $O(\log^k n)$. It was shown in [40] that $\mathrm{NC}^k = \mathrm{ASPACE,TIME}(\log n, \log^k n)$ 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 that constructs a circuit cannot be simulated in alternating log time. It was suggested in [20] and [40] that uniformity conditions stronger than logspace-uniformity be used so the equality $\mathrm{NC}^k = \mathrm{ASPACE,TIME}(\log n, \log^k n)$ would hold for all k. Note that these results are obtained by essentially "overpowering" the precomputation phase by making the uniformity condition very strong. In this paper we take the approach of "factoring out" the precomputation phase and dealing with it explicitly. In so doing, we show that close correspondences exist between P-uniform circuit depth and appropriately-chosen machine resources. For instance, we show that PUNC^k is the class of languages accepted by logspace-bounded alternating Turing machines that access their input only during the first $O(\log^k n)$ steps, for all $k \in \mathbf{N}$.

In Section 6, results are proved that clarify the relationship between PUNC and NC and other complexity classes. The main result of this section is

$$\mathrm{NC} = \mathrm{PUNC} \Leftrightarrow \text{all tally languages in } \mathbf{P} \text{ are in NC.}$$

In fact, we define a suitable notion of reducibility, $\leq_T^{\mathrm{NC}}$ reducibility, and show that there are tally sets complete for PUNC under $\leq_T^{\mathrm{NC}}$ reductions.

The characterizations of PUNC also clarify the complexity of one-way auxiliary pushdown automata, which have been studied before in [7], [11], [12], [17], [29], [47], and [48]. Some restrictions of AuxPDAs have been shown not to have a severe effect on the complexity of the languages they accept; in [23], a two-way deterministic (one-head) PDA is presented that accepts a language that is hard for P under logspace reductions. Some other restrictions have been shown to be more limiting; in [32] and [46], it was shown that logspace-bounded AuxPDAs whose pushdowns make at most a constant number of turns accept only languages in NLOG, and Ruzzo [40] showed that logspace-bounded AuxPDAs that run in time $2^{\log^{O(1)} n}$ accept only languages in NC. Here we show that logspace-bounded AuxPDAs that 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 that is hard for P. Also, since we show that each tally language in P can be accepted by a one-way logspace-bounded AuxPDA, it follows that one-way logspace-bounded AuxPDAs accept some sets that are complete for PUNC under $\leq_T^{\mathrm{NC}}$ reductions.

Many of the results presented here are also contained in the author's doctoral dissertation [2], and were announced first in an extended abstract that appeared as [4].

2. *Preliminaries*

We use the standard lexicographic ordering $\leq$ on strings, $|x|$ denotes the length of the string x, and $|S|$ denotes the cardinality of the set S. The empty string is denoted by ϵ. Logarithms are taken to the base 2.

A language is *sparse* if $|\{w \in L \mid |w| \leq n\}|$ is bounded by a polynomial in n; 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 for instance, we may write $|w| = \log w$. We 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 $\{00, 11, 01\}^*$.

The *characteristic sequence* for a language L is the infinite binary sequence $b_1, b_2, \ldots$ such that b_n is 1 iff n is in L.

A *circuit* for inputs of size n is a finite collection of AND, OR, and NOT 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 do 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 $\{C_n \mid n \in \mathbf{N}\}$ 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 $n \rightarrow C_n$ 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 value w. $\{C_n\}$ is a *family of circuits for* $L \subseteq \{0, 1\}^*$ if $\{C_n\}$ computes the characteristic function for L.

Background, and a more detailed discussion of circuit complexity, may be found in [40].

When referring to classes of functions, we make frequent use of "big Oh" 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 $f(n) \leq k2^{kn}$ for some k, etc.

The reader is expected to be familiar with standard complexity classes such as NP, P, etc. We use E to denote DTIME$(2^{O(n)})$, and NE to denote NTIME$(2^{O(n)})$. DLOG denotes DSPACE$(\log n)$ and NLOG denotes NSPACE$(\log n)$. The reader is also expected to be familiar with alternating Turing machines; the article [14] should be consulted for definitions of concepts related to alternation.

Auxiliary pushdown automata (AuxPDAs) are due to Cook [18]. An AuxPDA is a Turing machine with a pushdown store in addition to a worktape. When we bound the space used by an AuxPDA, we bound only the space used on the worktape; the space used on the pushdown store is "free." Useful results about AuxPDAs are summarized in [28]. We need the fact that the languages accepted in time $T(n)^{O(1)}$ are precisely the sets accepted by log $T(n)$ space-bounded deterministic and nondeterministic AuxPDAs [18].

The following easy proposition is not new, and is presented primarily to illustrate a technique that will be used in the proof of Theorem 4.1.

PROPOSITION 2.1. *Every tally set in P is accepted by a deterministic one-way logspace-bounded AuxPDA.*

PROOF. Let L be a tally set in P. Then the set $L' = \{n \mid 0^n \in L\}$ is in E. Thus, by [18] there is a linear-space-bounded AuxPDA M accepting L'. The one-way logspace-bounded AuxPDA accepting L will first scan across the input, keeping track of the number of symbols read. After it has read the entire input, it will reject if the input is not of the form 0^n for some n. Otherwise, it will use its worktape, with n written on it, to simulate both the input tape and the worktape of the linear-space-bounded AuxPDA M, accepting iff $n \in L'$. □

Some notation will be needed for complexity classes defined by simultaneously bounding more than one resource. **C**-uniform SIZE,DEPTH$(S(n)$, $(T(n))$ is the class of languages for which there exists a **C**-uniform family of circuits of size bounded by $S(n)$ and depth bounded by T(n), where **C** is some time- or space-complexity class. DSPACE,TIME$(S(n), T(n))$ is the class of languages accepted by deterministic Turing machines which operate simultaneously in space S(n) and time T(n). ASPACE,TIME$(S(n), T(n))$ and AuxPDASPACE,TIME$(S(n), T(n))$ are defined similarly for alternating Turing machines and AuxPDAs, respectively.

The two most-studied complexity classes defined in terms of simultaneous resource bounds are NC and SC:

$$\begin{aligned} \mathrm{NC} &= \text{DLOG-uniform SIZE,DEPTH}(n^{O(1)}, \log^{O(1)} n), \\ \mathrm{SC} &= \text{DSPACE,TIME}(\log^{O(1)} n, n^{O(1)}). \end{aligned}$$

We make frequent use of the following characterization of NC.

THEOREM 2.2 [40]

$$NC = ASPACE,TIME(\log n, \log^{O(1)} n) = AuxPDASPACE,TIME(\log n, 2^{\log^{O(1)} n}).$$

3. *PUNC*

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

Definition 3.1. *PUNC = P-uniform SIZE,DEPTH*$(n^{O(1)}, \log^{O(1)} n)$.

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 RAMs or finite-state machines, the same class of languages results. Similarly, we could have defined PUNC in terms of aggregates [22] or conglomerates [25] with P-uniform interconnection networks.

PROPOSITION 3.2. *The following are equivalent*:

(1) *L is in PUNC.*
(2) *L is accepted in time* $\log^{O(1)} n$ *by an aggregate with a P-uniform interconnection network.*
(3) *L is accepted in time* $\log^{O(1)} n$ *by a conglomerate with a P-uniform interconnection network.*

Proposition 3.2 can be proved by a slight adaptation of the proofs given in [19] and [21] showing how to simulate aggregates and conglomerates by circuit families.

PUNC is also closed under a broad class of reducibilities. The following definitions introduce classes of reductions that are useful for studying PUNC.

Definition 3.3. A language L is NC-reducible to S (written $L \leq_T^{\mathrm{NC}} S$) iff there is a logspace-uniform family of circuits $\{C_n\}$ of depth $\log^{O(1)} n$ for L, where the circuits $\{C_n\}$ are allowed to have *oracle* gates for S. An oracle gate for S is a node with some sequence $\langle g_1, g_2, \ldots, g_r \rangle$ of input gates; the gate takes on the output value 1 if the string $b_1 b_2 \cdots b_r \in S$, where b_i is the value output by g_i. For the purpose of defining the depth of C_n, this oracle node counts as depth $\log r$.

NC reductions are a straightforward generalization of NC^1 reductions, as defined in [20]; the only difference is that NC^1 reductions are computed by circuits of depth $\log n$, whereas NC reductions may be of depth $\log^k n$ for any k. Wilson has also studied a similar generalization of NC^1 reductions; in [49], Wilson defines, for any constant a, NC$_a$ reductions to be NC reductions computed by circuits of depth $\log^a n$.

NC reducibility is the natural NC restriction of $\leq_T^p$ reducibility. In a similar way, one can define the NC version of $\leq_m^p$ reducibility:

Definition 3.4. ($L \leq_m^{\mathrm{NC}} S$) iff there is a logspace-uniform family of circuits $\{C_n\}$ of depth $\log^{O(1)} n$, computing a function f, such that for all w, $w \in L \Leftrightarrow f(w) \in \mathrm{S}$.

In a similar way, we define $\leq_T^{\mathrm{PUNC}}$ and $\leq_m^{\mathrm{PUNC}}$ reductions, by changing the logspace-uniformity condition to P-uniformity in each of the preceding definitions. Note that every logspace reduction (as defined in [31]) is also a $\leq_m^{\mathrm{NC}}$ reduction, computed by circuits of depth $O(\log^2 n)$. Note also that the class of $\leq_T^{\mathrm{PUNC}}$ reductions clearly subsumes all of the other reduction classes considered here; thus the following proposition implies that PUNC is closed under all of these reducibilities.

PROPOSITION 3.5. *If $S \in PUNC$ and $L \leq_T^{PUNC} S$, then $L \in PUNC$.*

Thus PUNC has a robust definition in terms of circuit-based models of parallel computation. In the next two sections we present characterizations of PUNC in terms of some sequential models of computation, and also in terms of general-purpose parallel computers.

4. *An Alternate Characterization of PUNC*

In order to consider alternating Turing machines of sublinear time complexity (which is necessary in order to characterize NC in terms of alternating time) 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. First, however, we present another characterization by restricting how often AuxPDAs may move their input heads.

THEOREM 4.1. *The following are equivalent:*

(1) *$L \in PUNC$*
(2) *L is accepted by a logspace-bounded deterministic AuxPDA that moves its input head $O(2^{\log^{O(1)} n})$ times.*
(3) *L is accepted by a logspace-bounded nondeterministic AuxPDA that moves its input head $O(2^{\log^{O(1)} n})$ times.*

PROOF

(1) $\Rightarrow$ (2): 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 $\{O^n \# g \# i \# r \mid g$ is the name of a gate in C_n, $i \in \{1, 2\}$, and the rth bit of the name of the ith input of g is $1\}$ is also in P. Equivalently, the language $L' = \{n \# g \# i \# r \mid g$ is the name of a gate in C_n, $i \in \{1, 2\}$, and the rth bit of the name of the ith input of g is $1\}$ is in E. (Here we are assuming a reasonable naming convention, where the name of a gate has length $O(\log n)$; since there are only polynomially many gates this is not a restriction.) By the results of [18], E is the class of languages accepted by linear-space-bounded deterministic AuxPDAs; thus there is a deterministic AuxPDA that can determine if $n \# g \# i \# r \in L'$ using space linear in $|n| = O(\log n)$. This implies that a logspace-bounded deterministic AuxPDA with an input of length n, with n and g written on its worktape, can compute the names of the gates g_1 and g_2, which are the inputs to g in C_n. This computation is done by determining if $n \# g \# i \# r \in L'$, for $i \in \{1, 2\}$ and

$1 \leq r \leq O(\log n)$, which can be done by using the logspace-bounded worktape to simulate both the input tape and the worktape of the linear-space-bounded AuxPDA accepting L'. It is important to notice that this subcomputation can be done by the logspace-bounded AuxPDA *without moving its input head.*

Now let M be the deterministic logspace-bounded AuxPDA that on input w of length n, executes the following algorithm:

```
begin
  write $n$ in binary on the worktape.                  ($O(n)$ input head moves)
  $g := $ the output gate for $C_n$                       (0 input head moves)
  call EVALUATE($g$)
end

EVALUATE($g$)
begin
  if $g$ is an input gate for input $i$
    then return the value of input position $i$         ($O(n)$ input head moves)
    else let $g_1$ and $g_2$ be the inputs to $g$         (0 input head moves)
      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 $g$
end
```

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 $2^{\log^{O(1)} n}$.

(3) $\Rightarrow$ (1). Let L be accepted by M, a nondeterministic logspace-bounded AuxPDA that moves its input head at most $2^{\log^{O(1)}(n)}$ times on inputs of length n. As Mager showed in [34], 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 $\rightarrow$ between surface configurations: if $C_1 = (q, x, i, \Gamma, a)$ and $C_2 = (p, y, j, \Gamma, a)$ are surface configurations, then $C_1 \rightarrow C_2$ iff M, when started in state q with Γ on its pushdown, x on its worktape, its worktape head on the ith symbol of x, and its input head scanning an a, can make some sequence of moves *without moving its input head* and *without popping* Γ, 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 $\rightarrow$ have often been used before (e.g., [1], [18], and [33]); the primary difference between the relation $\rightarrow$ and, for example, the relation that was defined in [18] is that here we are only concerned with pairs of surface configurations that can be connected by a computation that 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 that, 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 $C_1 \rightarrow C_2$. (A simple variant of the algorithm used in [18] 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' that, 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 nondeterministically 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
end
```

The idea of the algorithm is that M' simulates M directly, except that at some times M' uses the information given by $h(|w|)$ to skip a number of moves of M that do not involve moving the input head.

To see that M' runs in time $2^{\log^{O(1)}(n)}$ on inputs of the form $w \# h(|w|)$, note first that every move in which M moves its input head is simulated directly by M', by choosing option (b). We show that M' makes at most $n^{O(1)}$ steps between any two steps that simulate moves in which M moves its input head. The running time of M' will thus be $n^{O(1)}2^{\log^{O(1)}(n)}$.

Note that there will always be an accepting computation of M' in which, between any two simulated moves of M that move the input head, all "pop" moves are performed before any "push" moves are performed. (All modifications to the stack are made when choosing option (b); 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 that can be performed without manipulating the stack is bounded by some polynomial $p(n)$, the number of moves performed by M' between any two simulated moves of M that move the input head is no more than $2p(n) * (n^l)$. By the comments in the previous paragraph, the running time of M' will thus be $n^{O(1)}2^{\log^{O(1)}(n)} = 2^{\log^{O(1)}(n)}$ on inputs of the form $w \# h(w)$.

Now by Theorems 1 and 2 of [39], there is an alternating logspace-bounded Turing machine that 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 form a new machine M'' which runs in time $\log^{O(1)}n$ on *all* inputs, and accepts inputs of the form $w \# h(w)$ iff they are accepted by M' (iff $w \in L$); note that M'' does not necessarily accept the same language as M'. Let L'' be the language accepted by M''.

$L'' \in$ NC, and thus there is a logspace-uniform family $\{D_n\}$ of circuits for L''. Let $r_n = |w \# h(n)|$ 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}. $\square$

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 far 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 that uses at least logspace can be simulated efficiently by a machine that 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 that *precedes* any access to the input, and to restrict the total number of nodes in the alternation tree that access the input.

THEOREM 4.2. *$L \in PUNC^k \Leftrightarrow L$ is accepted by a logspace-bounded alternating Turing machine that 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 [40].)

($\Rightarrow$) Let L be accepted by a P-uniform family $\{C_n\}$ of circuits of depth $O(\log^k n)$. L is accepted by the logspace-bounded alternating Turing machine M that operates as follows on inputs w of length n. (In the description of the following algorithm, the phrase "universally check the guess" is an abbreviation for "enter a universal state; along one branch verify that the guess is correct, and along the other branch continue with the rest of the simulation.")

```
begin
 Existentially guess $n$ and universally check the guess.
 $g :=$ the output gate of $C_n$.
 $p := \epsilon$.
          [$p$ represents a path in $\{left\text{-}input, right\text{-}input\}^*$ leading away from the gate $g$.]
 $accept.bit := 1$.
          [$accept.bit$ keeps track of the effect of NOT gates seen so far.]
 repeat
   Guess the type $t \in$ {AND, OR, NOT, INPUT} of the gate that is reached by following
   the path $p$ away from the gate $g$ in $C_n$, and universally check the guess.
   if $|p| = \log n$
     then
          [it is time to shorten the path, so no more than $\log n$ space is used]
       Existentially guess the name $h$ of the gate that is reached by following the path $p$
       away from the gate $g$ in $C_n$, and universally check the guess.
       $p := \epsilon$
       $g := h$
   if $t =$ AND, universally set $p := p.left\text{-}input$ and $p := p.right\text{-}input$.
   if $t =$ OR, existentially set $p := p.left\text{-}input$ or $p := p.right\text{-}input$.
   if $t =$ NOT, set $p := p.right\text{-}input$ and $accept.bit := 1 - accept.bit$.
 until $t =$ INPUT
 existentially guess the number $i$ such that the $i$th bit of the input should be read, and
 universally check the guess. Read the $i$th bit of the input and halt and accept iff the bit
 that is read $= accept.bit$.
end
```

It is clear that M accepts its input iff the input is in L.

The only steps in the algorithm that access the input occur outside the loop; each such step requires $O(\log n)$ time. The parts of the algorithm in which "M universally checks a guess" can be performed without accessing the input. Thus the number of steps that are executed before the input is accessed is $O(\log n)$ plus the number of steps that are executed while inside the **repeat** loop.

The only steps that occur in the loop that take more than unit time are in the "**if** $|p| > \log n$" statement. These steps take $O(\log n)$ time, and are executed only once every $\log n$ times through the **repeat** loop. Thus the total number of steps that are executed on a path before the input is read is $O(\log n)$ plus some constant times the number of times the **repeat** loop is executed. The number of times the **repeat** loop is executed is bounded by the depth of the circuit. Thus M accesses its input only during the first $O(\log^k n) + O(\log n) = O(\log^k n)$ steps.

($\Leftarrow$) Let L be accepted by a log space-bounded alternating Turing machine M that 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 ith input by input gates reading the ith 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, respectively. □

It is interesting to compare Theorem 4.2 to the characterization of SC given by Sudborough in [42]. Sudborough considered both one-way and two-way loglog-space-bounded alternating Turing machines, and he showed that SC is the class of all problems that are logspace-reducible to languages accepted by one-way loglog-space-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. (See also [16] and [30] for more results about one-way loglogspace-bounded alternating Turing machines.)

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 [39] relating AuxPDAs and alternating Turing machines.

THEOREM 4.3. *The following are equivalent*:

(1) *$L \in PUNC$*
(2) *L is accepted by a logspace-bounded alternating Turing machine that accesses its input only during the first $\log^{O(1)} n$ alternations.*
(3) *L is accepted by a logspace-bounded alternating Turing machine that, if it accepts an input, accepts via an alternation tree that contains $2^{\log^{O(1)}(n)}$ nodes that access the input.*

5. *A General-Purpose Parallel Computer for PUNC*

Existing parallel computation devices fall into one of two categories. Some are *special-purpose* devices, which are built to compute some fixed function very quickly. Other devices are programmable; there are several processors communicating over some interconnection system, and the system can be programmed to perform a variety of tasks. Let us call such devices *general-purpose* parallel computers.

Circuit-based models of computation are an abstraction, the study of which is motivated to some degree by the desire to understand the limitations and capabilities of special-purpose circuits. The circuit model ignores certain technology-dependent factors that affect the efficiency with which a given circuit design could actually be implemented. In the same way, models of parallel computation such as SIMDAGs, WRAMs, etc., are abstractions based on general-purpose parallel computers. (See [44] for a survey of these models.)

NC is the class of problems 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 general-purpose parallel computation that have been proposed in the past few years. This has been taken as evidence that NC truly captures the notion of efficient parallel computation. However, it was argued earlier in the paper 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. One might therefore suspect that NC models general-purpose parallel computation, and PUNC models special-purpose parallel computation. We show here that that is not necessarily the case; PUNC can also be characterized in terms of general-purpose parallel computers.

Models of general-purpose parallel computers such as SIMDAGs and WRAMs all share the characteristic that they have infinitely many processors. These processors are nearly identical, but each processor has a register which stores its "name", or address. The model we propose here involves augmenting each processor with a small amount of extra information, where the information given to each processor is feasible to compute. The justification for this is that, in any physical implementation of a general-purpose parallel computer, there will be only finitely many processors, and each processor will take some (feasible) amount of time to build. If faster performance can be obtained by building extra information into each processor, then that is sufficient reason to make such information available.

A *SIMDAG augmented with a sequence* s consists of a SIMDAG along with an infinite sequence of read-only registers $B_1, B_2, \ldots,$ where each B_i contains the ith bit of the sequence. 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. To make this transparent, we consider that register B_i resides 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 nth 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 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 a s_U. Furthermore, s_U has a natural and appealing definition.

Let U be any language complete for E under log-lin reductions. (See, e.g., [41].) 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 nth bit, see if $n \in U$ in time $2^{O(|n|)} = 2^{O(\log n)} = n^{O(1)}$.

THEOREM 5.1. *$L \in PUNC \Leftrightarrow L$ is accepted using $n^{O(1)}$ processors in time $\log^{O(1)} n$ time on a SIMDAG augmented with s_U.*

PROOF

($\Rightarrow$) Let L be accepted by a PUNC^k family of circuits $\{C_n\}$. There is a Turing machine M that on input $n \# r$ will accept iff the rth bit of C_n is 1, and M runs in time $n^{O(1)}$ and thus in time $2^{l|n\#r|}$ for some l. Let SVC^k be the Shallow Circuit Value problem (see [40]): SCV^k = $\{w\#C \mid C$ is the description of a circuit of depth $\leq \log^k |w|$ and C outputs a 1 when given w as input$\}$. SVC$^k \in$ NC, and $w\#C_{|w|} \in$ SVC^k iff $w \in L$.

Now consider a SIMDAG augmented with s_U that on input w, first constructs $w\#C_{|w|}$ in logarithmic time; the rth bit of $C_{|w|}$ may be obtained by consulting register $B_{M\#n\#r\#^{l|n\#r|}}$. After $w\#C_{|w|}$ is constructed, the SIMDAG will accept iff $w\#C_{|w|} \in$ SVC^k. This second part of the computation takes time $\log^{O(1)} n$, and thus so does the entire computation. The number of processors needed for the second part of the computation is polynomial, and the number of processors needed for the first part is bounded by the number of registers B_i that need to be consulted. Since $|M\#n\#r\#^{l|n\#r|}| = O(\log n)$, only polynomially-many registers B_i need to be consulted.

($\Leftarrow$) 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' that, on all inputs of the form $w \# s_U(1)\, s_U(2) \cdots s_U(|w|^k + k)$, runs in $\log^{O(1)}n$ time and uses $n^{O(1)}$ processors and accepts iff $w \in L$. M' can be modified to run in $\log^{O(1)}n$ time on *all* inputs and thus accept a (possibly different) 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. □

Theorem 5.1 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 [24] and [45], in which efficient general-purpose parallel computers are studied. The difference between this result and those results is that, in [24] and [45], a general-purpose general computer was considered to be a parallel computer with some fixed, rather limited interconnection network, that could take a program written perhaps for some other type of interconnection network *as input* and simulate it, much as a universal Turing machine does. Here, the efficiency of the simulation is not at issue since we are not trying to simulate one type of one computer by another. Instead, we have shown that there is one, fixed piece of "hardware" (namely, the SIMDAG augmented with s_U) on which any language in PUNC can be recognized in polylog time using a polynomial number of processors.

P-printable sequences can also be used to give a characterization of PUNC that is analogous to Pippenger's result that NC is the class of languages accepted by Turing machines that run in polynomial time and whose worktape heads make only $\log^{O(1)}n$ reversals [37]. Define a *Turing machine augmented with a sequence* s to be a Turing machine with the infinite sequence s written on one of its worktapes. The following result can be proved in almost the same way as Theorem 5.1.

THEOREM 5.2. *PUNC is the class of languages accepted by Turing machines augmented with s_U that run in polynomial time and whose worktape heads make only $\log^{O(1)}n$ reversals.*

PUNC is simply NC augmented by a feasible amount of precomputation. Some results dealing with precomputation in a more general setting were presented in [4].

6. *Tally Sets and PUNC*

This section explores the relationship between PUNC and NC and other complexity classes. In particular, the results in this section help to explain the nature of the computational power added by P-uniformity over logspace-uniformity.

Note that by Proposition 2.1 and Theorem 4.1, every tally set in P is in PUNC. (This is also easy to see directly.) Let $T_U = \{0^i \mid i \in U\}$, where U is the complete set for E used in the proof of Theorem 5.1. Note that T_U is a tally set in P, and thus is in PUNC.

THEOREM 6.1. *T_U is complete for PUNC under $\le_T^{NC}$ reductions.*

PROOF. As observed above, $T_U \in$ PUNC; thus it suffices to show that every set in PUNC is $\le_T^{\mathrm{NC}}$-reducible to T_U. Let $L \in$ PUNC. By Theorem 5.1, L is accepted by a SIMDAG M augmented by s_U which uses $n^{O(1)}$ processors and runs in $\log^{O(1)}n$ time. The SIMDAG M defines a $\le_T^{\mathrm{NC}}$ reduction, where the registers containing the bits of s_U correspond to oracle gates for T_U. □

COROLLARY 6.2. *The following are equivalent:*

(1) $L \in PUNC$.
(2) L *is* $\leq_T^{NC}$ *reducible to* T_U.
(3) L *is* $\leq_T^{NC}$ *reducible to some tally set in P.*

PROOF. (1) $\Rightarrow$ (2) follows from Theorem 6.1. (2) $\Rightarrow$ (3) is obvious. (3) $\Rightarrow$ (1) because every tally set in P is in PUNC and PUNC is closed under $\leq_T^{\mathrm{NC}}$ reductions. □

Stated another way, Corollary 6.2 characterizes PUNC as the closure under NC reductions of the class of tally sets in P.

COROLLARY 6.3. *NC = PUNC* $\Leftrightarrow$ *all tally languages in P are in NC* $\Leftrightarrow$ T_U *is in NC.*

PROOF. Immediate from Theorem 6.1 and from the fact that NC is closed under $\leq_T^{\mathrm{NC}}$ reductions. □

Ruzzo, in considering "U_B-uniformity" [40] (as opposed to P-uniformity), has recently proved some results with a flavor similar to Corollary 6.3 (W. L. Ruzzo, personal communication).

Much has been written about the consequences of sparse sets being complete for NP and other complexity classes containing P; Mahaney [35] has written a good survey of this area, including a large bibliography of papers dealing with this topic. However, almost nothing is known about NLOG or P in this regard. It was shown in [26] that no set complete for NLOG or for P under $1 - L$ reductions can be sparse; that result was subsequently improved in different ways by [5] and [29]. Nonetheless, nothing seems to be known about the consequences of there being a sparse set (or even a tally set) that is complete for NLOG or P under logspace reductions. Since logspace reductions are a special case of NC reductions, the following result remedies that situation.

COROLLARY 6.4. *P = PUNC iff there is a tally set complete for P under* $\leq_T^{NC}$ *reductions.*

PROOF. If P = PUNC, then T_U is complete for PUNC, and hence for P, under $\leq_T^{\mathrm{NC}}$ reductions, by Corollary 6.2. On the other hand, if there is a tally set $T \in$ P such that every set $L \in$ P is $\leq_T^{\mathrm{NC}}$ reducible to T, then by Corollary 6.2 every set in P is in PUNC. □

The same techniques show that P is contained in nonuniform NC $\Leftrightarrow$ there is a tally set that is hard for P under $\leq_T^{\mathrm{NC}}$ reductions $\Leftrightarrow$ there is a sparse set that is hard for P under $\leq_T^{\mathrm{NC}}$ reductions. It is not known if P = PUNC is equivalent to there being a sparse set complete for P under $\leq_T^{\mathrm{NC}}$ reductions; it is easy to show that equivalence holds if all sparse sets in P are in PUNC. Note in this regard that *all* sparse sets have *nonuniform* log depth circuits.

Tally languages have often been studied in conjunction with other types of sparse sets, and the results in this section can be generalized somewhat. One class of sparse sets that has been studied in some detail is the class of *P-printable* sets (see, e.g., [6] and [27]). S is P-printable if the function $1^n \rightarrow S \cap \{0, 1\}^n$ is computable in polynomial time. Clearly, all P-printable sets are in PUNC, and most obvious examples of sparse sets in PUNC are easily seen to be P-printable. Furthermore, there are some obvious connections between P-printable sets, P-printable sequences, and P-uniform circuits. It is shown in [6] that a set is P-printable iff it is sparse and accepted by a one-way logspace-bounded AuxPDA; thus, in light of Theorem 4.1 this suggests that perhaps all sparse sets in PUNC are P-printable. This suggestion is misleading, however, as it is also shown in [6] that if there is any sparse set in P that is not P-printable, then there is such a set in DLOG (and hence in PUNC). Allender [3] presents other results related to the question of whether or not all sparse sets in P are P-printable.

P-printable sets share most properties of tally sets. In the statement of Corollaries 6.2, 6.3, and 6.4, the word "tally" can be replaced by "P-printable" and the results are still true.

We have no results relating to the question of whether or not all sparse sets in P are in PUNC. However, the following result addresses a related issue.

THEOREM 6.5. *$E = NE \Leftrightarrow$ all sparse sets in NP are in PUNC.*

PROOF

($\Rightarrow$) In [27] it was shown that E = NE implies that all sparse sets in NP are P-printable, and hence in PUNC.

($\Leftarrow$) If all sparse sets in NP are in PUNC, then all tally sets in NP are in P, which implies that E = NE by [9]. □

There is yet one more interesting corollary to Theorem 6.1. Using the techniques presented in [9], it is possible to interpret results about classes of tally languages as results about "higher" complexity classes. Thus those techniques easily yield the following result.

COROLLARY 6.6. *$NC = PUNC$ iff $ASPACE,TIME(n, n^{O(1)}) = ASPACE(n)$.*

Since NC = ASPACE,TIME($\log n$, $\log^{O(1)} n$) and P = ASPACE($\log n$), Corollary 6.6 says that the NC = PUNC question and the NC = P question are both instances of the more general question of whether or not ASPACE,TIME($S(n)$, $S(n)^{O(1)}$) = ASPACE($S(n)$); in many regards they are essentially the same question. Since ASPACE(n) = E, Corollary 6.6 says that the NC = PUNC question is equivalent to the exponential-time analog of the NC = P question.

COROLLARY 6.7. *$NC = PUNC \Rightarrow PSPACE = DTIME(2^{n^{O(1)}})$.*

PROOF. It suffices to show that NC = PUNC implies DTIME($2^{n^{O(1)}}$) $\subseteq$ PSPACE. By Corollary 6.7, if NC = PUNC, then E = ASPACE,TIME(n, $n^{O(1)}$) $\subseteq$ ATIME($n^{O(1)}$) = PSPACE. Since everything in DTIME($2^{n^{O(1)}}$) is $\leq_m^p$ reducible to a set in E, and PSPACE is closed under $\leq_m^p$ reductions, the result follows. □

Indeed, the logspace-uniform circuit complexity of sets in PUNC seems closely tied to the complexity of sets in PSPACE. Corollary 6.7 shows that if NC = PUNC, then PSPACE contains very complex sets. On the other hand, if PSPACE = P, then PUNC contains sets that require large depth on logspace-uniform circuits, as the following result shows.

PROPOSITION 6.8. *If PSPACE = P, then PUNC is not contained in DLOG-uniform SIZE,DEPTH($n^{O(1)}$, n^k) for any k.*

PROOF. If PSPACE = P, then DSPACE($2^{O(n)}$) = E, and thus (by the space hierarchy theorem) for all k, there is some set L in E that is not in DSPACE(2^{kn}). Letting $T_L = \{0^n \mid n \in L\}$, we have (using the techniques of [9]) that T_L is in P but not in DSPACE(n^k). Clearly, T_L is in PUNC; however, by [10], T_L does not have DLOG-uniform circuits (or even DSPACE(n^k)-uniform circuits) of depth n^k.

7. *Open Problems*

Theorem 5.1 holds the promise of having practical applications, since it tells how to build a general-purpose parallel computer that can efficiently solve any problem for which a special-purpose chip can be built. Unfortunately, the practical utility of this result is limited since, as Rackoff has pointed out (C. Rackoff, personal communication), the number of registers B_i needed, although polynomial in the size of the input, is exponential in the size of the program M that builds the special-purpose chip. It would be nice to know if the proof can be improved so the constant factor is not so overwhelming. Alternatively, it might be possible to use Kolmogorov complexity arguments to show that no real improvement is possible. Such a result would show that there is, indeed, a real difference between special-purpose and general-purpose parallel computation.

Although we have shown the existence of sets complete for PUNC under $\leq_T^{\mathrm{NC}}$ reductions, it is not known if there are sets complete for PUNC under $\leq_m^{\mathrm{NC}}$ reductions. Note that for most complexity classes with complete sets, there are so-called "standard" complete sets, which are obviously complete under logspace reductions. Since every logspace reduction is computable by NC^2 circuits, it is easy to show that if PUNC has a set complete under logspace reductions, then $\mathrm{PUNC} = \mathrm{PUNC}^k$ for some k; thus PUNC probably has no "standard" complete set. We conjecture that there are no sets that are complete for PUNC under $\leq_m^{\mathrm{NC}}$ reductions.

Is there any evidence for the existence of sparse sets in P (or in SC) that are not in PUNC?

Are there any "natural" problems that seem to be in PUNC–NC?

ACKNOWLEDGMENTS. Discussions with a number of people influenced the direction in which this work developed: my thesis advisor: Kim King, Marc Graham, Rich DeMillo, Craig Tovey, Ray Miller, Steve Cook, Charles Rackoff, Jim Hoover, Pat Dymond, Larry Ruzzo, Steve Mahaney, Ron Book, and Osamu Watanabe.

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. Time and tape complexity of pushdown automaton languages. *Inf. Control 13* (1968), 186–206.
2. ALLENDER, E. W. Invertible functions. Doctoral dissertation. Georgia Institute of Technology, Atlanta, GA, 1985.
3. ALLENDER, E. W. The complexity of sparse sets in *P*. In *Proceedings of the 1st Structure in Complexity Theory Conference*. Lecture Notes in Computer Science, vol. 223. Springer-Verlag, New York, 1986, pp. 1–11.
4. ALLENDER, E. W. Characterizations of PUNC and precomputation. In *Proceedings of the 13th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 226. Springer-Verlag, New York, 1986, pp. 1–10.
5. ALLENDER, E. W. Isomorphisms and 1*L* reductions. *J. Comput. Syst. Sci. 36* (1988), 336–350.
6. ALLENDER, E. W., AND RUBINSTEIN, R. S. P-printable sets. *SIAM J. Comput. 17* (1988), 1193–1202.
7. BALCÁZAR, J. L., DÍAZ, J., AND GABARRÓ, J. On some "non-uniform" complexity measures. In *Proceedings of the 5th Conference on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 199, pp. 18–27.
8. BEAME, P. W., COOK, S. A., AND HOOVER, H. J. Log depth circuits for division and related problems. *SIAM J. Comput. 15* (1986), 994–1003.
9. BOOK, R. V. Tally languages and complexity classes. *Inf. Control 26*, (1974), 186–193.
10. BORODIN, A. On relating time and space to size and depth. *SIAM J. Comput. 6* (1977), 733–744.
11. BRANDENBURG, F.-J. On one-way auxiliary pushdown automata. In *Proceedings of the 3rd GI Conference*. Lecture Notes in Computer Science, vol. 48. 1977, pp. 133–144.

12. BRANDENBURG, F.-J. The contextsensitivity of contextsensitive grammars and languages. In *Proceedings of the 4th International Colloquium on Automata, Languages and Programming.* Lecture Notes in Computer Science, vol. 52. Springer-Verlag, New York, 1977, pp. 272–281.
13. BUSS, S. R. The Boolean formula value problem is in ALOGTIME. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (New York, N.Y., May 25–27). ACM, New York, 1987, pp. 123–131.
14. CHANDRA, A. K., KOZEN, D. C., AND STOCKMEYER, L. J. Alternation. *J. ACM 28,* 1 (Jan. 1981), 114–133.
15. CHANDRA, A. K., STOCKMEYER, L. J., AND VISHKIN, U. Constant depth reducibiilty. *SIAM J. Comput. 13* (1984), 423–439.
16. CHANG, J. H., IBARRA, O. H., RAVIKUMAR, B., AND BERMAN, L. Some observations concerning alternating Turing machines using small space. *Inf. Proc. Lett. 25* (1987), 1–9; Erratum: *Inf. Proc. Lett. 27* (1988), 53.
17. CHYTIL, M. P. Comparison of the active visiting and the crossing complexities. In *Proceedings of the 6th Conference on Mathematical Foundations of Computer Science.* Lecture Notes in Computer Science, vol. 53. Springer-Verlag, New York, 1977, pp. 272–281.
18. COOK, S. A. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM 19* (1971), 175–183.
19. COOK, S. A. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Math. 27* (1981), 99–124.
20. COOK, S. A. A taxonomy of problems with fast parallel algorithms. *Inf. Control 64* (1985), 2–22.
21. DYMOND, P. W. Simultaneous resource bounds and parallel computations. Doctoral dissertation, University of Toronto, Toronto, Ont., Canada, 1980.
22. DYMOND, P. W., AND COOK, S. A. Complexity theory of parallel time and hardware. *Inf. Computation 80* (1989), 205–226.
23. GALIL, Z. Some open problems in the theory of computation as questions about two-way deterministic pushdown automaton languages. *Math. Syst. Theory 10* (1977), 211–228.
24. GALIL, Z., AND PAUL, W. An efficient general-purpose parallel computer. *J. ACM 30,* 2 (Apr. 1983), 360–387.
25. GOLDSCHLAGER, L. M. A universal interconnection pattern for parallel computers. *J. ACM 29,* 4 (Oct. 1982), 1073–1086.
26. HARTMANIS, J., AND MAHANEY, S. Languages simultaneously complete for one-way and two-way log-tape automata. *SIAM J. Comput. 10* (1981), 383–390.
27. HARTMANIS, J., AND YESHA, Y. Computation times of NP sets of different densities. *Theoret. Comput. Sci. 34* (1984), 17–32.
28. HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.
29. HUYNH, D. T. Non-uniform complexity and the randomness of certain complete languages. Tech. Rep. TR 85-34. Computer Science Department, Iowa State Univ., Ames, IA, 1985.
30. ITO, A., INOUE, K., AND TAKANAMI, I. A note on alternating Turing machines using small space. *Trans. IEICE (Japan), Section E, 70* (1987), 990–996.
31. JONES, N. D. Space-bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci. 11* (1975), 68–85.
32. KING, K. N. Measures of parallelism in alternating computation trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (Milwaukee, Wis., May 11–13). ACM, New York, 1981, pp. 189–201.
33. KING, K. N. Alternating multihead finite automata. In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming.* Lecture Notes in Computer Science, vol. 115. Springer-Verlag, New York, 1981, pp. 506–520.
34. MAGER, G. Writing pushdown acceptors. *J. Comput. Syst. Sci. 3* (1986), 276–319.
35. MAHANEY, S. R. Sparse sets and reducibilities. In *Studies in Complexity Theory,* R. V. Book, Ed. Wiley, New York, 1986.
36. MIX BARRINGTON, D. A., IMMERMAN, N., AND STRAUBING, H. On uniformity within NC^1. In *Proceedings of the 3rd Structure in Complexity Theory Conference.* IEEE Computer Society Press, Washington, D.C., 1988, pp. 47–59.
37. PIPPENGER, N. On simultaneous resource bounds. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1979, pp. 307–311.
38. REIF, J. On threshold circuits and polynomial computation. In *Proceedings of the 2nd Structure in Complexity Theory Conference.* 1987, pp. 118–123.
39. RUZZO, W. L. Tree-size bounded alternation. *J. Comput. Syst. Sci. 21* (1980), 218–235.

40. RUZZO, W. L. On uniform circuit complexity. *J. Comput. Syst. Sci. 21* (1981), 365–383.
41. STOCKMEYER, L. J. The complexity of decision problems in automata theory and logic. Doctoral dissertation. Massachusetts Inst. of Technology, Cambridge, Mass., 1974.
42. SUDBOROUGH, I. H. Bandwidth constraints on problems complete for polynomial time. *Theoret. Comput. Sci. 26* (1983), 25–52.
43. VON ZUR GATHEN, J. Parallel powering. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing.* ACM, New York, 1984, pp. 31–36.
44. VISHKIN, U. *Synchronous Parallel Computation—A Survey.* Tech. Rep. TR 71. Department of Computer Science, Courant Inst., New York Univ., New York, 1983.
45. VISHKIN, U. A parallel-design distributed-implementation (PDDI general-purpose computer). *Theoret. Comput. Sci. 32* (1984), 157–172.
46. WECHSUNG, G. The oscillation complexity and a hierarchy of context-free languages. In *Proceedings of the 2nd International Conference on Fundamentals of Computation Theory.* Akademie-Verlag, Berlin, GDR, 1979, pp. 508–515.
47. WECHSUNG, G. A note on return complexity. *Elektronische Informationsverarbeitung und Kybernetik 16* (1980), 139–146.
48. WECHSUNG, G., AND BRANDSTADT, A. A relation between space, return and dual return complexities. *Theoret. Comput. Sci. 9* (1979), 127–140.
49. WILSON, C. B. On the decomposability of NC and AC. In *Proceedings of the 4th Structure in Complexity Theory Conference.* IEEE Computer Society Press, Washington, D.C., 1989, pp. 124–131.

RECEIVED SEPTEMBER 1986; REVISED JULY 1988 AND DECEMBER 1988; ACCEPTED DECEMBER 1988