A New Parallel Vector Model, With Exact Characterizations of NC^k

Kenneth W. Regan*
State University of New York at Buffalo

Abstract

This paper develops a new and natural parallel vector model, and shows that for all $k \geq 1$, the languages recognizable in $O(\log^k n)$ time and polynomial work in the model are exactly those in NC^k. Some improvements to other simulations in parallel models and reversal complexity are given.

1 Introduction

This paper studies a model of computation called the *Block Move* (BM) model, which makes two important changes to the Pratt-Stockmeyer vector machine (VM). It augments the VM by providing bit-wise *shuffle* in one step, but restricts the *shifts* allowed to the VM. Computation by a BM is a sequence of "block moves," which are finite transductions on parts of the memory. Each individual finite transduction belongs to uniform NC^1 and is computable in at most logarithmic time on the common parallel machine models. Hence counting the number of block moves is a reasonable measure of parallel time. It is also natural to study restrictions on the kinds of finite transducers S a BM can use. We write BM(gh) for the model in which every S must be a *generalized homomorphism* (gh), and BM(ap) for the restriction to S such that the monoid of transformations of S is *aperiodic*. Every gh is an NC^0 function, and Chandra, Fortune, and Lipton [6] showed that every aperiodic transduction belongs to AC^0. The object is to augment the the rich theory of classes within NC^1 and algebraic properties of automata which has been developed by Barrington and others [2, 5, 4, 15, 3].

Karp and Ramachandran [14] write EREW^k, CREW^k, CRCW^k, and VM^k for $O(\log^k n)$ time and polynomial work on the three common forms of PRAM and the VM, respectively, and cite the following results:

- For all $k \geq 1$, $\mathrm{NC}^k \subseteq \mathrm{EREW}^k \subseteq \mathrm{CREW}^k \subseteq \mathrm{CRCW}^k = \mathrm{AC}^k$ [23].
- For all $k \geq 2$, $\mathrm{NC}^k \subseteq \mathrm{VM}^k \subseteq \mathrm{AC}^k$ [24].

Now write BM^k, $\mathrm{BM}^k(\mathrm{gh})$, and $\mathrm{BM}^k(\mathrm{ap})$ for $O(\log^k n)$ time and polynomial work on the BM forms above. The main result of this paper is:

- For all $k \geq 1$, $\mathrm{NC}^k = \mathrm{BM}^k(\mathrm{gh})$.

We also observe that $\mathrm{BM}^k \subseteq \mathrm{NC}^{k+1}$ and $\mathrm{BM}^k(\mathrm{ap}) \subseteq \mathrm{AC}^k$. The main theorem is noteworthy for being an exact characterization by a simple machine without using alternation. From its proof we derive several technical improvements to results on

*The author was supported in part by NSF Research Initiation Award CCR-9011248
Author's current address: Computer Science Department, 226 Bell Hall, UB North Campus, Buffalo, NY 14260-2000. Email: *regan@cs.buffalo.edu*, tel.: (716) 645–3189, fax: (716) 645–3464.

Turing machine reversal complexity, with reference to [7] and [16]. Sections 2-4 define the model and give fairly full sketch proofs of the main results, and Section 5 gives other results and open problems.

2 The BM Vector Model

The model of finite transducer we use is the *deterministic generalized sequential machine* (DGSM), as formalized in [12] (see also [9]). A DGSM S is like a Mealy machine, but with the ability to output not just one but zero, two, three, or more symbols in any one transition. A special case is when a finite function $h : \Sigma^d \rightarrow \Gamma^e$ $(d, e > 0)$ is extended to a function $h_* : \Sigma^* \rightarrow \Gamma^*$ as follows: (1) for all $x \in \Sigma^*$ with $|x| < d$, $h_*(x) = \lambda$ (the empty string), and (2) for all $x \in \Sigma^*$ and $w \in \Sigma^d$, $h_*(wx) = h(w)h_*(x)$. Then h_* is called a *generalized homomorphism* (gh) with *ratio* $d : e$. Three important examples with $\Sigma = \Gamma = \{0, 1\}$ are $A(x)$, which takes the AND of each successive pair of bits and thus is 2:1, $O(x)$ similarly for OR, and bit-wise negation $N(x)$, which is 1:1. Another is *dilation* $D(x)$, which doubles each bit of x and is 1:2; e.g., $D(101) = 110011$.

The BM model has a single tape, and an alphabet Γ in which the blank B and endmarker \$ play special roles. A BM M has four "pointers" labeled a_1, b_1, a_2, b_2, and some number $m \geq 4$ of "pointer markers." The finite control of M consists of "DGSM states" $S_1, \ldots S_r$ and finitely many "move states." Initially, the input x is left-justified in cells $0 \ldots n-1$ of the tape with \$ in cell n and all other cells blank, one pointer marker with b_1 and b_2 assigned to it is in cell n, and cell 0 holds all other pointer markers with a_1 and a_2 assigned there. The *initial pass* is by S_1. The computation is a sequence of passes, and if and when M halts, the content of the tape up to the first \$ gives the output $M(x)$. In a move state, each pointer marker on some cell a of the tape may be moved to cell $\lfloor a/2 \rfloor$, $2a$, or $2a + 1$ or left where it is. Then the four pointers are redistributed among the m markers, and control branches according to the symbol in the cell now occupied by pointer a_1. Each move state adds 1 to both the *work* $w(x)$ and the *pass count* $R(x)$ of the computation. A GSM state S executes the *block move*

$$S\,[a_1 \ldots b_1]\ \textit{into}\,[a_2 \ldots b_2]$$

defined as follows: Let z be the string held in locations $[a_1 \ldots b_1]$—if $b_1 < a_1$, then z is read right-to-left on the tape. Then $S(z)$ is written into locations $[a_2 \ldots b_2]$, overwriting any previous content, *except* that any blank B appearing in $S(z)$ leaves the symbol in its target cell unchanged. Control passes to a unique move state. We may also suppose wlog. that each move state sends control to some DGSM state. If $a_1 \leq b_1$ and $a_2 \leq b_2$, the pass is called *left-to-right*. It falls out of our main theorem that left-to-right passes alone suffice for all NC^1 computations, extending the known observation that strings can be reversed in $O(\log n)$ vector operations (see [11]).

The work in the move is defined to be $|z|$, i.e. $|b_1 - a_1| + 1$. The *validity condition* is that the intervals $[a_1 \ldots b_1]$ and $[a_2 \ldots b_2]$ must be disjoint, and the *strict boundary condition* is that the output must exactly fill the target interval; i.e., that $|S(z)| = |b_2 - a_2| + 1$. The former can always be met at constant-factor slowdown by buffering outputs to an unused portion at the right end of the tape, and so we may ignore it.

We leave the reader to check that the strict boundary condition is observed in our simulations—in most moves, a_1, b_1, a_2, b_2 are multiples of powers of two.

The original vector model of Pratt and Stockmeyer [19] is a RAM M with arbitrarily many arithmetical registers R_i and some number k of "vector registers" (or tapes) V_j, each of which holds a binary string. M uses standard RAM operations on the arithmetical registers, bitwise AND, OR, and negation on (pairs of) vector tapes, and *shift* instructions of the form $V_k := R_i \uparrow V_j$, which shift the contents of tape V_j by an amount equal to the integer n_i in R_i and store the result in V_k. If n_i is positive, n_i-many 0's are prepended, while if n_i is negative (a left shift), the first n_i-many bits of V_j are discarded.[1] These shift instructions essentially provide "random access" to the vector tapes. The two *main points* of the BM as opposed to the VM are the realistic provision of constant-time *shuffle*, and the constraint on random-access. Having the former improves the bounds given by Hong [11] for many basic operations, and these operations seem not to be helped by full random-access anyway. The main theorem of this paper is that these adjustments lead to exact characterizations of the classes NC^k.

3 Basic List Operations

Much of the following is standard, but there are several innovations from Lemma 3.5 onward. These innovations come in situations where previous cited work used quantities which are powers of 2, and where our work seems to require double powers of 2; i.e., that lengths be normalized to numbers of the form 2^{2^b}. All logs below are to base 2. To save space we introduce a useful shorthand for hierarchical lists.

We call a sequence $(x_1, x_2, \ldots, x_l)$ of nonempty binary strings a *list*, and represent it by the string $X = x_1^{\#} x_2^{\#} \cdots x_l^{\#}$. Here $x^{\#}$ means that the last symbol of x is a "compound symbol," either $(0, \#)$ or $(1, \#)$, and similarly for the other *list separator symbols* $\$, \%, @$. Informally, the separators have "precedence" $\% < \# < @ < \$$. The *length* of the list is l, and the *bit-length* is $\sum_{j=1}^{l} |x_i|$. If each string x_i has the same length s, then the list X is *normal*, a term used by Chen and Yap [7] when also l is a power of 2 (see below). A normal list represents an $l \times s$ Boolean matrix in row-major order. Many of the technical lemmas concern larger objects we call *vectors of lists*, with the general notation

$$\begin{aligned} \mathcal{X} &= X_1^{\$}\,; X_2^{\$}\,; \ldots\,; X_m^{\$} \\ &= x_{11}^{\#} x_{12}^{\#} \cdots x_{1l_1}^{\$} x_{21}^{\#} x_{22}^{\#} \cdots x_{2l_2}^{\$} \cdots x_{m-1,l_{m-1}}^{\$} x_{m1}^{\#} \cdots x_{ml_m}^{\$}. \end{aligned}$$

If each list X_i is normal with string length s_i, then $\mathcal{X}$ is a vector of Boolean matrices. If also $s_1 = s_2 = \ldots = s_m = s$ and $l_1 = l_2 = \ldots = l_m = l$, we call $\mathcal{X}$ *fully normal*. Finally, if s and l are also powers of 2 (notation: $s = 2^b$, $l = 2^c$), then $\mathcal{X}$ is *supernormal*. We often treat vectors of lists as simple lists subdivided by \$, and even as single strings. Let the bits of x be $a_1 a_2 \ldots a_s$:

Basic string operations

[1] It has been observed that the distinction between arithmetic and vector registers is inessential, shifts being the same as multiplication and division by powers of 2, and that left shifts are unnecessary—see [21, 1, 25, 14, 24] for these observations and alternate forms of the VM. The original form makes the clearest comparison to the BM.

- R stands for "replicate string": $R(x) = xx$.
- D stands for "dilate" as above: $D(x) = a_1 a_1 a_2 a_2 \ldots a_s a_s$.
- H splits x into two halves $H_1(x) = a_1 \ldots a_{s/2}$, $H_2(x) = a_{s/2+1} \ldots a_s$ (s even).
- J joins two strings together: $J(H_1(x), H_2(x)) = x$.
- S shuffles the first and second halves of a string x of even length; i.e., $S(x) = a_1 a_{s/2+1} a_2 a_{s/2+2} \ldots a_{s/2} a_s$. Note that SJ shuffles two strings of equal length.
- U is "unshuffle," the inverse of S: $U(x) = a_1 a_3 a_5 \ldots a_{s-1} a_2 a_4 \ldots a_s$.

List operations

- $D_\#$ duplicates list elements: if $X = x_1^\# x_2^\# \ldots x_l^\#$, then $D_\#(X) = x_1^\% x_1^\# x_2^\% x_2^\# \ldots x_l^\% x_l^\#$. Note $D_\#^2(x) = x_1^\% x_1^\% x_1^\% x_1^\# x_2^\% x_2^\% x_2^\% x_2^\# \ldots x_l^\% x_l^\% x_l^\% x_l^\#$.
- $S_\#$ shuffles lists element-wise: $S(X) = x_1^\# x_{l/2+1}^\# x_2^\# x_{l/2+2}^\# \ldots x_{l/2}^\# x_l^\#$, defined when l is even. $S_\# J$ shuffles two lists of the same length l element-wise.
- $U_\#$ is the inverse of $S_\#$; $HU_\#$ breaks off odd and even sublists.
- $T_\#(a_{11} \cdots a_{1s}^\# a_{21} \cdots a_{l1} \cdots a_{ls}^\#) = a_{11} a_{21} \cdots a_{l1} a_{12} \cdots a_{l2} \cdots a_{1s}^\# a_{2s}^\# \cdots a_{ls}^\#$.
- $B_{\%\#}$ places a % halfway between every two #s in a list, when all such distances are even. $B_{\#\#}$ inserts more #s instead.

When applied to a vector list $\mathcal{X}$, these operations treat $\mathcal{X}$ as a single list of strings subdivided by #, but preserve the higher-level \$ separators. The operations $D_\$$, $S_\$$, $U_\$$, $T_\$$, and $B_{\$\$}$, however, treat $\mathcal{X}$ as a list of strings subdivided by \$. Note that we have allowed for marked symbols being displaced in matrix transpose $T_\#$. All the string and list operations *op* above may be *vectorized* via the general notation

$$\vec{op}(\mathcal{X}) = op(X_1)^\$ \, op(X_2)^\$ \ldots op(X_m)^\$.$$

A convenient general property is that $(op_1 \vec{\circ}\, op_2)(\mathcal{X}) = \vec{op}_1(\vec{op}_2(\mathcal{X}))$. Sometimes we omit the $\circ$ and compose operators from right to left. We note that for any $\mathcal{X}$, $\vec{D}_\#(\mathcal{X}) = D_\#(\mathcal{X})$, but for instance with $l = m = 8$:

$$\vec{U}_\#(\mathcal{X}) = x_{11}^\# x_{13}^\# x_{15}^\# x_{17}^\# x_{12}^\# \ldots x_{18}^\$ x_{21}^\# x_{23}^\# \ldots x_{26}^\# x_{28}^\$ \ldots x_{81}^\# x_{83}^\# \ldots x_{86}^\# x_{88}^\$,$$

while

$$U_\#(\mathcal{X}) = x_{11}^\# x_{13}^\# x_{15}^\# x_{17}^\# x_{21}^\# x_{23}^\# x_{25}^\# x_{27}^\# \ldots x_{85}^\# x_{87}^\# x_{12}^\# x_{14}^\# x_{16}^\# x_{18}^\$ x_{22}^\# \ldots x_{86}^\# x_{88}^\$.$$

Note the displacement of \$ signs in the latter. For Lemma 3.2 below we want instead $U'_\#(\mathcal{X}) = x_{11}^\# x_{13}^\# x_{15}^\# x_{17}^\$ x_{21}^\# x_{23}^\# x_{25}^\# x_{27}^\$ \ldots x_{85}^\# x_{87}^\$ x_{12}^\# x_{14}^\# x_{16}^\# x_{18}^\$ x_{22}^\# \ldots x_{86}^\# x_{88}^\$$. However, our BMs need not rely on the markers, but can use counters for b and c (prepended to the data) to compute the long expressions given below. We omit the details of these counters here, but note some places where $B_{\#\#}$ and $B_{\%\#}$ are explicitly used.

Lemma 3.1 *Let X be a normal list with $s = 2^b$ and l even. Then*

(a) $B_{\%\#}(X) = S^{b-1} \circ (\textit{mark every 2nd bit of the first } 2l \textit{ with } \%) \circ U^{b-1}(X)$.

(b) $T_\#(X) = U^b(X)$.

(c) For any $a \geq 1$, $D^a_{\#}(X) = T^{-1}_{\#} D^a(T_{\#}(X)) = S^b D^a U^b(X)$.

(d) $S_{\#}(X) = S^{b+1}(J(U^b H_1(X), U^b H_2(X)))$.

(e) $U_{\#}(X) = J(S^b(H_1 U^{b+1}(X)), S^b(H_2 U^{b+1}(X)))$.

Item (c) technically needs a final application of $B_{\%\#}$. Alternatively, the % markers can be introduced during the D^a part. The key point of the next lemma and its sequel is that the number of passes to compute $\vec{S}_{\#}(\mathcal{X})$ and $\vec{U}_{\#}(\mathcal{X})$ is independent of m. Analogous operations given by Hong on pp111–115 of [11] for the Pratt-Stockmeyer model have an $O(\log m)$ term in their time.

Lemma 3.2 *Let $\mathcal{X}$ be super-normal with $s = 2^b$, $l = 2^c$, and m even. Then*

(a) $\vec{U}_{\#}(\mathcal{X}) = S_{\$}(U'_{\#}(\mathcal{X})) = S^{b+c+1}(J(U^c H_1(U^{b+1}(\mathcal{X})), U^c H_2(U^{b+1}(\mathcal{X}))))$.

(b) $\vec{S}_{\#}(\mathcal{X}) = S_{\#}(U_{\$}(\mathcal{X})) = S^{b+1}(J(S^c(H_1 U^{b+c+1}(\mathcal{X})), S^c(H_2(U^{b+c+1}(\mathcal{X}))))$.

Lemma 3.3 *Assuming good initial pointer and marker placements,*

(a) Each of the string operations is computable in $O(1)$ passes by a BM(gh)*.*

(b) With reference to Lemma 3.1, $D^a_{\#}(X)$ is computable in $O(a+b)$ passes, and the other four list operations in $O(b)$ passes.

(c) The two vector operations in Lemma 3.2 are computable in $O(b+c)$ passes.

The product of two Boolean matrices X and Y of size $2^b \times 2^b$ is computed by

$$M(X,Y) = O^b ASJ(D^b_{\#}(X), R^b T_{\#}(Y)).$$

Define the *outer-product* of two strings $x = a_1 \ldots a_2$ and $y = b_1 \ldots b_s$ by $V(x,y) = a_1 b_1 a_1 b_2 \cdots a_1 b_s a_2 b_1 \ldots a_2 b_s \ldots a_s b_1 \ldots a_s b_s$. Then $V(x,y) = SJ(D^b(x), R^b(y))$, and $AV(x,y)$ gives the $s \times s$ Boolean outer-product matrix. The corresponding list operation is applied to adjacent pairs of strings in a list, viz.: $V_{\#}(X) = V(x_1,x_2)^{\#} V(x_3,x_4)^{\#} \ldots V(x_{l-1},x_l)^{\#}$. We also vectorize these operations:

Lemma 3.4 *For super-normal vector lists $\mathcal{X}$ with $l = 2^c$, $s = 2^b$:*

(a) $\vec{V}_{\#}(\mathcal{X}) = SJ(D^b H_1(U_{\#}\mathcal{X}), D^b_{\#} H_2(U_{\#}\mathcal{X}))$, and is computable in $O(b)$ passes.

(b) $\vec{M}(\mathcal{X},\mathcal{X}) = O^b ASJ(D^b_{\#} H_1(D_{\#} U_{\#}(\mathcal{X})), D_{\$} T_{\#} H_2((D_{\#} U_{\#}(\mathcal{X})))$ provided that $b = c$, and is then computable in $O(b)$ passes.

The following way to do vectorized conversion from binary to unary notation works for binary numbers whose lengths are powers of 2. Let $X = n_1^{\#} n_2^{\#} \ldots n_m^{\#}$, where each n_i is a string of length $s = 2^b$ in binary notation (thus $0 \leq n_i \leq 2^s - 1$), with the most significant bit first. The object is to convert each n_i to $0^{2^s - n_i - 1} 1 0^{n_i}$. Call the resulting vector $W_b(X)$. Note that $W_0(X)$ is just the homomorphism which maps $0 \to 01$ and $1 \to 10$ (or technically, $0^{\#} \to 01^{\#}$ and $1^{\#} \to 10^{\#}$).

Lemma 3.5 *For all $b > 0$, $W_b(X) = AV_{\#} W_{b-1} B_{\#\#}(X)$, and is computable in $O(s)$ passes.*

Proof. Consider any item $n_i = a$ on the list X. $B_{\#\#}$ converts that item to $a_1^{\#} a_2^{\#}$, such that $a = a_1 2^{s/2} + a^2$. Arguing inductively, suppose that W_{b-1} then converts $a_1^{\#} a_2^{\#}$ to $0^{2^{s/2}-a_1-1}10^{a_1}{}^{\#}0^{2^{s/2}-a_2-1}10^{a_2}{}^{\#}$. Then the outer-product of adjacent pairs of elements produces $O^{2^s-a}10^a$ as needed, and by Lemma 3.4(a), is computed in $O(s)$ passes. The time t_b to compute W_b has the recursion $t_b = O(b) + t_{b-1} + O(2^b)$, with solution $t_b = O(2^b) = O(s)$. $\square$

Lemma 3.6 *The list I_b of all strings of length $s = 2^b$ in order can be generated in $O(s)$ passes by a BM(gh).*

Proof. $I_0 = 0^{\#}1^{\#}$, and for $b \geq 1$, $I_b = S_{\#}J(D^s(I_{b-1}), R^s(I_{b-1}))$. The timing recursion is similar to before. $\square$

Next, given a single-tape Turing machine M with state set Q and work alphabet Γ, define the *ID alphabet* of M to be $\Gamma_I := (Q \times \Gamma) \cup \Gamma$. A valid ID of M has the form $I = x(q,c)y$ and means that M is in state q scanning character c. The *next move* function steps M through to the next ID.

Lemma 3.7 *The next move function of a single-tape TM T can be extended to a total function $\delta_M : \Gamma_I^* \to (\Gamma_I \cup \{+,-,!\})^*$, and there is a 2:1 generalized homomorphism $h : (\Gamma_I \cup \{+,-,!\})^* \to (\Gamma_I \cup \{+,-,!\})^*$ such that*

(a) δ_T is the composition of three 3:3 generalized homomorphisms.

(b) Consider strings $I \in \Gamma_I^*$ of length 2^b. If I is a valid ID and T goes from I to an accepting (resp. rejecting) ID within t steps, staying within the 2^b allotted cells of the tape, then $h^b(\delta_T^t(I)) = +$ (resp. $-$). Else $h^b(\delta_T^t(I)) = !$.

(c) The above properties can be obtained with Γ_I re-coded over $\{0,1\}$.

Proof Sketch. One of the three gh's updates cells $0, 3, 6, \ldots$ of the tape of T, one does $1, 4, 7, \ldots$, and the last does $2, 5, 8 \ldots$ If T halts and accepts, the homomorphisms detect this and replace the symbol (q,c) by $+$, and similarly with $-$ for rejection. If T is detected to move its head off the tape, the homomorphisms introduce a ! symbol. The same happens if T "crashes" in I or if I is a string over Γ_I with more than one "head" and two of these "heads" come within two cells of each other. Finally h is written so that it preserves a $+$ or $-$ under iteration iff every other character it reads belongs to Γ. $\square$

We note a trick needed to extend the above idea for multitape TMs. For alternating TMs, it appears that the k-to-1 tape reduction theorem of Paul, Prauss, and Reischuk [17] holds even for log-time bounds, but we still allow for multiple tapes.

Lemma 3.8 *Let T be a TM with $k \geq 2$ tapes, which is constrained to operate within space $s = 2^b$ on each tape. Then b moves by T can be simulated in $O(b)$ passes by a* BM(gh). *Furthermore, this operation can be vectorized; i.e., applied to a list of IDs of T.*

Proof Sketch. The contents and head position for each tape are represented as described before Lemma 3.7, and then IDs of T are encoded by shuffling these representations. The difficulty in simulating one move by T is that heads of T on different tapes may be up to s cells apart, and it is impossible to record and propagate the information about the character each head is scanning in one pass by a gh. Let $g = |\Gamma|$. The solution is to do g^k passes, one for each k-tuple of symbols that the heads could possibly be scanning. The simulating BM M has a special character for each k-tuple over Γ, and all symbols in the output of each pass are marked by the special character for the tuple assumed in that pass. Each such pass appends its output to the right end of the tape. After the g^k passes, we have g^k candidates for the next move by T. The process is repeated on the list of candidates until it produces a tree of depth b with g^{bk} leaves, where each branch is a possible b moves by T. Then with b passes by a 2:1 gh on a spare copy of the tree, M can verify for each node whether the tuple assumed for it was correct. In $O(bg^k) = O(b)$ more passes, M can propagate the marking of bad nodes from parents to all descendents, leaving just one correct branch. In $O(b)$ more passes, this information is transferred back to the original copy of the tree. $\square$

Remarks for later reference: For simulating the next b steps by T, it is not necessary to erase all the bad leaves—the '!' markings of bad IDs can be copied forward in later passes. Now suppose $g = |\Gamma|$ is a power of 2. When the next-move operation is vectorized and applied once to a level of nodes $N_1, \ldots, N_j$ of the tree, the next level comes out in the order

$$[\text{child 1 of } N_1]^{\#} \ldots [\text{child 1 of } N_j]^{\$} [\text{child 2 of } N_1]^{\#} \ldots [\text{child } g^k \text{ of } N_j]^{\$}.$$

Now applying $S_{\#}$ $(k \log g)$-many times would bring all the children of each node together. However, since the list elements are $s = 2^b$ symbols long, this would take $O(b)$ passes, and since later we will have $n = 2^s$, this would make the overall pass count $O(\log n \operatorname{loglog} n)$. Instead we wait until the bottom-level IDs have been converted to single-symbol values $+$, $-$, or ! (or 0, 1, !) by b-many applications of a 2:1 gh. Then children can be brought together at each level via the string operation $S^{k \log g}$, and since k and g are independent of n, the number of passes per level is constant. This will be used when T is an alternating TM with binary branching.

4 Main Theorem

Cook [8] made it standard to refer to the U_{E^*} uniformity condition of Ruzzo [20] in defining the classes NC^k and AC^k for $k \geq 1$. For AC^1 and higher this is equivalent to older conditions of log-space uniformity, and we use this for NC^k with $k \geq 2$ below. Using U_{E^*} uniformity gives the identity NC^k = ATISP$(O(\log^k n), O(\log n))$ for all $k \geq 1$ [20], where ATISP refers to simultaneously time- and space-bounded alternating Turing machines. Thus NC^1 is also called ALOGTIME. A circuit is *leveled* if its nodes can be partitioned into $V_0, \ldots, V_d$ such any wire from a node in some V_i goes to a node in some V_{i+1}. The *width* of a leveled circuit equals $\max_i |V_i|$. The circuit is *layered* if in addition the input level V_0 consists of inputs x_i and their negations, and the remaining levels alternate AND and OR gates. For all $k \geq 1$, an

NC^k circuit can be converted to an equivalent layered NC^k circuit, within the same condition of uniformity.

Theorem 4.1 *For all $k \geq 1$, $\mathrm{BM}^k(gh) = \mathrm{NC}^k$.*

Proof. (Sketches) (1) $\mathrm{BM}^k(\mathrm{gh}) \subseteq \mathrm{NC}^k$. Let M have m markers, DGSM states $S_1, \dots S_r$, and a tape alphabet Γ of size g. Let $p(n)$ be a polynomial which bounds the work, and also the space, used by M on inputs of length n. The simulating circuit C_n is organized vertically into *blocks*, each of which simulates $b(n) = \epsilon \log n$ moves by M. Each block is organized horizontally into *segments*, each of which has $p(n)+1$ inputs and outputs. The extra input/output is a "flag," where '1' stands for "good" and '0' for "bad." For simplicity we describe C_n as though gates compute finite functions over the tape alphabet of M; conversion to a Boolean circuit expands size and depth by only a constant.

A *setting* consists of a positioning of the m markers, an assignment of the four pointers to the markers, and the current GSM. This makes $p(n)^m m^4 r$ possible settings. Given a setting before a pass, the total number of possible settings after the pass and subsequent move state is at most g, one for each tape character pointer a_1 might scan. Thus the total number of possible sequences of settings in the course of $b(n)$ moves is $B(n) := g^{b(n)} = n^{\epsilon \log g}$, which is polynomial. This is the point of constraining the machine's "random access."

Each block has $p(n)^m m^4 B(n)$ segments operating in parallel. Each individual segment corresponds to a different initial setting and one of the $B(n)$ sequences which can follow. Each segment is leveled with width $p(n)+1$, and has $b(n)$ "pass slices," each representing $p(n)$ tape cells plus the flag. Each pass slice corresponds to a setting and has the "hard wiring" for the block move $S_i[a_1 \dots b_1]$ *into* $[a_2 \dots b_2]$ carried out in the DGSM state S_i of that setting. Since S_i is one of finitely many gh's, each slice has constant depth. The flag gate in a pass slice is hard-wired to check that, based on the character scanned by a_1, the subsequent move state sets the pointers according to the setting in the next slice. If not, the flag is set to 0, and this is propagated downward.

The circuit carries the invariant that at the beginning of the ith block, there are $p(n)^m m^4 B(n)$ copies of the correct configuration of the tape of M after $b(n) \cdot i$ moves. Of these, those whose initial setting is incorrect already have the flag of their top slice set to 0; only the $B(n)$-many segments with the correct setting have flag 1. After $b(n)$ pass slices, *exactly one* of the segments has flag 1, namely the one whose sequence of settings M actually followed on the particular input. The other $p(n)^m m^4 B(n) - 1$ segments have flag 0. The rest of the block must locate the segment with flag 1 and route $B(n)$ copies of its $p(n)$ outputs among those segments in the next block whose initial setting matches the final setting of the source. This is standard: in $O(\log n)$ depth the segments with flag 0 are zeroed out, then bitwise comparison in an $O(\log n)$ depth butterfly pattern automatically replicates the correct one. The interconnect pattern is explicitly defined and regular enough to make the circuit U_{E^*} uniform.

(2) $\mathrm{NC}^k \subseteq \mathrm{BM}^k$ for $k \geq 2$: Let $\epsilon > 0$ and a log-space uniform family $[C_n]_{n=1}^{\infty}$ of layered circuits of depths $d(n)$ be given. We claim there is an equivalent family

$[C'_n]_{n=1}^{\infty}$ of circuits which have the following properties:

(a) C'_n is broken into *blocks*, each of depth $b(n) = \lceil \epsilon \log_2 n \rceil$. Let $w(n)$ be the next power of 2 higher than the width of C_n. Each block describes a circuit with $w(n)$ outputs, and is broken in parallel into $w(n)$ *formulas*, one for each output. Each formula has alternating levels of AND and OR with negations at the inputs. The formula has input variables $u_1, \ldots, u_{w(n)}$, some of which are dummy variables, together with their negations. (The negations are needed only in the first block.)

(b) C'_n has a special encoding E_n as a vector of lists, where each list represents a block and has $w(n)$ elements denoting the formulas in that block. The formulas are written in infix notation, and since they are full binary trees in which levels of AND and OR alternate, only the variables need be written. Each variable u_i is represented by the string $0^{w(n)-i}10^{i-1}\,\%$; its negation $\overline{u}_i$ by $0^{w(n)-i}{-}10^{i-1}\,\%$

(c) There is a log-space Turing machine which on input 0^n outputs the string E_n.

The proof of this claim is similar to proofs that log-space uniformity is preserved under other circuit normalizations, and omitted here. To finish part (2) we need to build a BM(gh) M such that on any input x of length n, M constructs E_n in $O(\log^2 n)$ passes and polynomial work, and evaluates E_n in $O(d_n)$ passes.

Since the formulas have size $2^{b(n)}$ which is polynomial, there is a polynomial $r(n)$ which bounds the length of E_n. It is well known that having E_n computable from 0^n in logspace is the same as having a log-space machine T which given 0^n and $i \leq r(n)$ decides the ith bit of E_n. We may then eliminate the input tape and make T start with i and n in binary notation on a single worktape with explicit endmarkers constraining T to logarithmic space $s_0(n)$. Let $s = s(n)$ be the next highest power of 2 after $s_0(n) \cdot C$, where C is the constant implied by the transformation from the ID alphabet of T to $\{0, 1\}$ in Lemma 3.7. Then T always halts within 2^s steps. Put $b = \log_2 s$.

The BM M first calculates n from $|x|$ and generates the list I of all binary strings of length $s(n)$ in lex order via Lemma 3.6. Each element in I is a potential ID of T. M then applies the next move function of T *once* to every element. The resulting list of strings of length s is no longer in lex order. Now this list can be converted to a Boolean matrix which represents the next-move function by running binary-to-unary on the list. This takes $O(s)$ passes. Then the matrix is raised to the power of 2^s by iterated squaring, in $O(s^2)$ passes. After conversion from unary back to binary, this produces a list of the final IDs reached from each given ID by T. Finally, $r(n)$ copies are made of this list by $O(\log n)$ replications. Then the numbers i from 1 to $r(n)$ are generated and aligned with successive copies. The answer given by T for each i can be computed by iterating a 2:1 gh h like that in Lemma 3.7 which preserves $+$ or $-$ in a final ID iff the corresponding starting ID held i, n. Iterating h $2s$-many times, and then changing $+$ to 1 and $-$ to 0, yields exactly the string E_n.

The circuit is evaluated by replicating the output of the previous block, and shuffling this with the list of unary variable identifiers $\pm u_j = 0^{w(n)-j} \pm 10^{j-1}$. The single '1' (or -1) then pulls off the input value of u_j from the previous block. The remainder of the block is evaluated by alternating $A(\cdot)$ and $O(\cdot)$.

(3) $\mathrm{NC}^1 \subseteq \mathrm{BM}^1(\mathrm{gh})$. Here we use the identity $\mathrm{NC}^1 = \mathrm{ALOGTIME}$. Let T be an ALOGTIME machine which accepts some language A. It is well known (see [22, 4]) that T can be converted to an ALOGTIME machine T' with the following properties: T' alternates existential and universal steps, and each non-terminal configuration has exactly two successors (called "right" and "left"). Each branch ignores the input tape until it reaches a terminal configuration, at which point the contents of a designated "address tape" specify in binary an integer i, $1 \leq i \leq n$, and the configuration accepts or rejects depending only on the value of the ith input. (Our simulation appears not to require the additional feature that every branch records its own sequence of left-right choices.) Now let n' be the next number above n of the form 2^{2^b}; then $n < n' \leq n^2$. We may further modify T' so that each of its k-many worktapes is constrained to $s = 2^b$ cells, each branch runs for exactly t steps, where $t = O(s)$ and t is a multiple of b, and each branch writes $n' - i$ right-justified on the address tape instead of i. Let Γ be the alphabet of T' and let $g := |\Gamma|$; we may suppose that g is a power of 2 (or even that $g = 2$).

The simulation begins with the single blank initial worktape ID I_λ. The computation by T' is simulated in block of b moves by the process of Lemma 3.8, but modified to produce both successors of every ID. Thus with reference to the proof, every ID has $2g^k$ children, two of which are legitimate—it is important that one good ID is in the first g^k and the other in the second g^k when the tree is flattened into a list. After t steps, there are $(2g^k)^t$ IDs, of which 2^t are legitimate. Since the tapes of T' are shuffled, one pass by a k:1 gh leaves just the address tape contents in these IDs, written in binary with extra trailing 0s out to length exactly $s = 2^b$. Then the unary-to-binary conversion of Lemma 3.5 is applied, with a slight change to preserve '!' markings in bad IDs. Now there is a polynomial-sized list of elements of the form $!^{n'}$ or $0^{i-1}10^{n'-i}$; the latter comes out that way because T' wrote $n' - i$ to address the ith bit. Then trailing dummy symbols are appended to pad the input x out to length n', and this is replicated and shuffled bit-wise with the list. One pass then picks up the input bit x_i addressed by each good terminal ID, and writes the value, 0 or 1, given by T' to that ID. Then s more passes by a 2:1 gh leave just the values 0, 1, or '!' of each terminal ID, bad ones included. The resulting string z has length $(2g^k)^t$ and contains exactly 2^t 0 or 1 symbols. Per remarks following Lemma 3.8, applying $S^{1+k\log g}$ to z brings the children of each bottom node together, and by the "important" note above, among each $2g^k$ children of a good node, exactly one symbol in the left-hand g^k is Boolean, and similarly for the right-hand g^k.

Now let $Er_!$ be the partially defined 2:1 gh which maps

$$!! \mapsto ! \qquad !0 \mapsto 0 \qquad 0! \mapsto 0 \qquad !1 \mapsto 1 \qquad 1! \mapsto 1,$$

Let $A_!$ be the total extension of $Er_!$ which behaves like AND on 00,01,10, and 11, and let $O_!$ be similar for OR. For argument's sake, suppose the last alternation by T' in each branch is AND. After $k \log g$ applications of $Er_!$ to $S^{1+k\log g}(z)$, the two Boolean values under each of these nodes are brought together, and then one application of $A_!$ evaluates this level. Under bad nodes, every character remains '!'. The next level is similarly evaluated by applying $O_! Er_!^{k\log g} S^{1+k\log g}$, in $O(1)$ passes. Doing this for $O(t) = O(s) = O(\log n)$ total passes evaluates the entire tree. $\square$

Corollary 4.2 (to (1) and [6])*: For all $k \geq 1$,* $\mathrm{BM}^k(ap) \subseteq \mathrm{AC}^k$.

Let "BM_0" stand for BMs which are provided with any finite set of NC^0 operations to use in block moves, and which may have more than one tape. In general a machine is *oblivious* if the movements of its tape heads depend only on the length of the input.

Corollary 4.3 *Every* BM_0 *which runs in polynomial work and $R(n) = \Omega(\log n)$ passes, can be simulated in polynomial work and $O(R(n))$ passes by a* $\mathrm{BM}(gh)$ *with a single tape which is oblivious and only makes left-to-right passes.* $\square$

5 Other Results and Conclusion

There are several definitions of *reversal complexity* for multitape Turing machines, where in a given transition, each head may stay stationary (S) as well as move left (L) or right (R). The older "strict" criterion of Kameda and Vollmer [13] is the same as counting any S move as a reversal. The newer one [18, 11, 16, 7] counts a reversal only when a head moves L which has previously moved R, or vice-versa. (Some other sources do not count reversals on the input tape.)

Lemma 5.1 *Every BM M which makes $R(n)$ passes can be simulated by a 2-tape TM T which makes $O(R(n))$ reversals.*

Proof Sketch. The first tape of T equals the tape of M, while the second tape is used to buffer the output in block moves. The second tape also helps T move the m markers to new positions in at most $2m$ reversals. $\square$

For a BM(gh), or more generally when every GSM in M translates input symbols to output symbols in some finite ratio $d\!:\!e$, $R(n)$ corresponds in this simulation to a notion of reversal complexity which is intuitively midway between the "strict" and the standard one: in every interval between reversals, each tape head must operate at some "fixed stride."

Parberry [16] showed that TMs which run in space $s(n)$ and $r(n)$ reversals can be simulated by uniform circuits of depth $O(r(n) \log^2 s(n))$ and width polynomial in $s(n)$. Chen and Yap [7] showed that any $r(n)$ reversal bounded multitape TM can be simulated by a 2-tape TM in $O(r(n)^2)$ reversals. In the case where the TM runs in polynomial space we obtain an inprovement:

Corollary 5.2 *A multitape TM which runs in polynomial space and $r(n)$ reversals can be simulated by a 2-tape TM in $O(r(n) \log^2 r(n))$ reversals.*

Finally, we term a BM M to be a *cascading finite automaton* (CFA) if M consists of a single DGSM S which is iterated left-to-right on its own output. (The validity condition is ignored.) For instance, the language D_1 of balanced parentheses is acceptable by a CFA S which skips the leading '('—writing '!' to reject if the first symbol is ')'—and thereafter translates

$$((\mapsto (\qquad)) \mapsto) \qquad () \mapsto \lambda \qquad)(\mapsto \lambda.$$

Then for all $x \neq \lambda$, either $S(x) = !$ or $x \in D_1 \iff S(x) \in D_1$, and always $|S(x)| \leq |x|/2$. Hence iterating S recognizes D_1 in $O(\log n)$ passes and linear work.

Open Problem 1. Does every language accepted by a CFA in $O(\log n)$ passes belong to NC^1? (Such languages do belong to one-way logspace [10].)

Open Problem 2. For $k \geq 1$, is $\mathrm{AC}^k = \mathrm{BM}^k(\mathrm{ap})$? In general, how do conditions on the structure of GSMs allowed to a BM correspond to circuit classes?

To conclude, the BM is a natural model which offers finer complexity analyses, and we look toward its further use on basic open problems in the NC hierarchy.

References

[1] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory.* Springer Verlag, 1988.

[2] D. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. Comp. Sys. Sci.*, 38:150–164, 1989.

[3] D. Mix Barrington, K. Compton, H. Straubing, and D. Thérien. Regular languages in NC^1. *J. Comp. Sys. Sci.*, 44:478–499, 1992.

[4] D. Mix Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comp. Sys. Sci.*, 41:274–306, 1990.

[5] D. Mix Barrington and D. Thérien. Finite monoids and the fine structure of NC^1. *J. ACM*, 35:941–952, 1988.

[6] A. Chandra, S. Fortune, and R. Lipton. Unbounded fan-in circuits and associative functions. *J. Comp. Sys. Sci.*, 30:222–234, 1985.

[7] J. Chen and C. Yap. Reversal complexity. *SIAM J. Comp.*, 20:622–638, 1991.

[8] S. Cook. A taxonomy of problems with fast parallel algorithms. *Info. Control*, 64:2–22, 1985.

[9] T. Harju, H.C.M. Klein, and M. Latteux. Deterministic sequential functions. *Acta Informatics*, 29:545–554, 1992.

[10] J. Hartmanis, N. Immerman, and S. Mahaney. One-way log tape reductions. In *Proc. 19th FOCS*, pages 65–72, 1978.

[11] J.-W. Hong. *Computation: Similarity and Duality.* Research Notes in Theoretical Computer Science. Wiley, 1986.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley, Reading, MA, 1979.

[13] T. Kameda and R. Vollmar. Note on tape reversal complexity of languages. *Info. Control*, 17:203–215, 1970.

[14] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 871–941. Elsevier and MIT Press, 1990.

[15] P. McKenzie, P. Péladeau, and D. Thérien. NC^1: The automata-theoretic viewpoint. *Computational Complexity*, 1:330–359, 1991.

[16] I. Parberry. An improved simulation of space and reversal bounded deterministic Turing machines by width and depth bounded uniform circuits. *Inf. Proc. Lett.*, 24:363–367, 1987.

[17] W. Paul, E. Prauss, and R. Reischuk. On alternation. *Acta Informatica*, 14:243–255, 1980.

[18] N. Pippenger. On simultaneous resource bounds. In *Proc. 20th FOCS*, pages 307–311, 1979.

[19] V. Pratt and L. Stockmeyer. A characterization of the power of vector machines. *J. Comp. Sys. Sci.*, 12:198–221, 1976.

[20] W. Ruzzo. On uniform circuit complexity. *J. Comp. Sys. Sci.*, 22:365–373, 1981.

[21] J. Simon. *On some central problems in computational complexity.* PhD thesis, Cornell University, 1975.

[22] M. Sipser. Borel sets and circuit complexity. In *Proc. 15th STOC*, pages 61–69, 1983.

[23] L. Stockmeyer and U. Vishkin. Simulations of parallel random access machines by circuits. *SIAM J. Comp.*, 13:409–422, 1984.

[24] J. Trahan, M. Loui, and V. Ramachandran. Multiplication, division, and shift instructions in parallel random access machines. *Theor. Comp. Sci.*, 100:1–44, 1992.

[25] P. van Emde Boas. Machine models and simulations. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1–66. Elsevier and MIT Press, 1990.