JOURNAL OF COMPUTER AND SYSTEM SCIENCES 12, 198–221 (1976)

A Characterization of the Power of Vector Machines*

VAUGHAN R. PRATT

Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

AND

LARRY J. STOCKMEYER

Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

A new formal model of register machines is described. Registers contain bit vectors which are manipulated using bitwise Boolean operations and shifts. Our main results relate the language recognition power of such vector machines to that of Turing machines. A class of vector machines is exhibited for which time on a vector machine supplies, to within a polynomial, just as much power as space on a Turing machine. Moreover, this is true regardless of whether the machines are deterministic or nondeterministic.

1. INTRODUCTION

In the literature on register machines, including [3, 4, 12], the domain of numbers has held sway. In this paper we draw attention to a relatively neglected domain that nonetheless forms the basis for today's commercial computers, namely, that of bit vectors. Although the original motivation for using bit vectors was to represent numbers, the development of nonnumeric techniques in recent years has found many other uses for bit vectors, and we shall derive our results for machines that embody no notion of number. The arithmetic operations found in previous formalizations of register machines are replaced, in our case, by bitwise Boolean operations and shifts.[1]

* An earlier version of this paper appeared in [10]. This research was supported by the National Science Foundation under research grant GJ-34671.

[1] A paper of Hartmanis and Simon [5], based on our results, shows that introducing the notion of number has, to within a polynomial, no effect on the power of vector machines, and moreover, that the multiplication instruction may take over the role of the shift instruction. We discuss this further below.

We shall show that such bit vector machines (VM's) have an astonishing amount of power. Our main result is that, to within a polynomial, *time* on a vector machine supplies at least as much computational power as *space* on a Turing machine. With a minor restriction in the instruction set of the VM's, "at least" may be replaced by "just," regardless of any questions of determinacy or nondeterminacy. More precisely, any set accepted in space $S(n)$ by a nondeterministic Turing machine may be accepted in time $O((S(n))^2)$ by a deterministic vector machine. Conversely, any set accepted in time $T(n)$ by a nondeterministic vector machine may be accepted in space $O((T(n))^2)$ by a deterministic Turing machine. An immediate corollary is that $P = NP$ on vector machines; that is, sets accepted in nondeterministic polynomial time can be accepted in deterministic polynomial time.

Section 2 gives the basic definitions concerning vector machines. Section 4 discusses several restrictions to the basic model. In Section 6, we state the characterization outlined above and examine some corollaries. Sections 7 and 8 present the proofs of the two main results which comprise the characterization. In Section 5, we examine the effect of the arithmetic operations, addition and multiplication, on the power of vector machines.

2. Definitions

Just as models of numeric register machines generally admit arbitrarily large numbers, the vector machines we study here admit arbitrarily long vectors. Thus we define a *bit vector* to be an ultimately constant sequence of bits (elements of $\{0, 1\}$). Though the manufacturers of our machines may eschew numbers, we will want to be able to simulate operations on integers (say, in the guise of two's complement binary numerals), and so we adopt the traditional convention of writing the sequence of bits from *right* to *left*. The *length* of a vector ν, written $|\nu|$, is the number of significant bits in it, that is, the length of the shortest initial segment of the sequence whose removal would make the remaining sequence constant.

The vector machines one can buy today invariably include close approximations to the following types of instructions, which we take to form the basis for the definitions of a variety of closely related machines:

(i) $A \leftarrow$ constant, an instruction to load a constant bit-vector into register A;

(ii) $A \leftarrow {\sim}B$ and $A \leftarrow B \wedge C$, "bitwise parallel" Boolean operations;

(iii) $A \leftarrow B \uparrow C$ ($A \leftarrow B \downarrow C$), which shifts B left (right) a distance given by C; negative distances mean a right (left) shift; when shifting left the vacated positions are filled with 0's, and when shifting right the bits shifted out are discarded;

(iv) $A = 0$ and $A \neq 0$, predicates for testing whether A is 0 everywhere.

Concerning the complement instruction $A \leftarrow {\sim}B$, the entire vector is complemented. Hence, if B is ultimately 0, A will be ultimately 1, and vice versa. Concerning the shift instruction (iii), one must of course specify how the bit-vector contents of register C are to be interpreted as a shift distance. Several alternatives are discussed below. One might also include indirect addressing instructions in this repertoire. However, Hartmanis and Simon [5] observe that indirect addressing has, to within a polynomial, no effect on the power of vector machines.

We define a *vector machine* (*program*) to be a finite directed graph with one *start* vertex and a set of *accepting* vertices, and with edges each labeled with one assignment instruction or predicate. Certain registers are designated as *input registers*. If a program mentions m registers, say $A_1, \ldots, A_m$, then a *configuration* consists of (i) a node of the graph, and (ii) bit vectors $\nu_1, \ldots, \nu_m$ (which specify the "current" contents of $A_1, \ldots, A_m$). A *computation* of such a program is a path in the graph, together with an initial configuration of the machine in which each noninput register contains the zero vector, such that each predicate on the path is satisfied by the machine configuration at the moment "control reaches" the edge bearing that predicate. (This can readily be made more formal.) An *accepting computation* is a computation which is a path from the start vertex to an accepting vertex. The *time* of a computation is its length. The *space* of a computation is the maximum, over all configurations during that computation, of the sum of the lengths of the vectors in the configuration. A *deterministic* vector machine is one such that for any vertex and any machine configuration, only one edge may be followed. (Hence, in a deterministic program the only vertices with out-degree greater than 1 are those with at most one each of $A = 0$ and $A \neq 0$ leaving it, for some choice of register A.)

By taking the time of a computation to be its length, we have implicitly assigned unit cost to each instruction (including predicates). This measure, combined with the lengths to which we allow our vectors to grow, makes our machine an unrealistic model of conventional binary computers. In the context of current architecture, its main virtue is that we have nice results for it. However, bearing in mind the currently plummetting costs of processors, we do not feel it is unrealistic to suggest that future machines may benefit from some of our results. Although our main theorem is unlikely to yield algorithms of practical interest, our algorithms for matrix multiplication and transitive closure are well within practical limits, taking about $80 \cdot \log n$ instructions to multiply $n \times n$ Boolean matrices, using a small number of vectors each of length exactly n^3. Hence, on a machine with megabit vectors, 100×100 Boolean matrices may be multiplied using about 560 instructions, only a fortieth of the number of bits in the inputs!

In the sequel, we take liberties with this programming language. For example, we write programs in an Algol-like notation, often utilizing "while-loops"; other bitwise Boolean operations such as $\vee$ and $\oplus$ are used. However, it should be obvious how to translate such constructs into the austere language described above at a cost of a constant factor in time.

When the shift distance in instruction type (iii) is interpreted as a two's complement binary number, we call the class of machines with just these four types of instructions, and with no restrictions on their use, the class $\mathscr{V}$ of (unrestricted) vector machines. Unfortunately we do not know how to characterize this class, so we will shortly introduce related (possibly) weaker classes for which we have the characterization promised in the Introduction.

We consider mainly the acceptance problem rather than the transduction problem. Since a vector machine receives its input in a register, we only consider languages which are sets of binary words.

DEFINITION. Let $L \subseteq 1 \cdot \{0, 1\}^*$, and let $F(n)$ be a function from positive integers to real numbers. A vector machine V accepts L within time (space) $F(n)$ iff V has one input register, and for all $\omega \in 1 \cdot \{0, 1\}^*$:

(1) $\omega \in L$ iff there is an accepting computation C of V whose initial configuration has the bit vector $\cdots 000\omega$ in the input register; and

(2) if $\omega \in L$, then there is an accepting computation C as in (1) such that the time (space) of C does not exceed $F(|\omega|)$.

Remark. $|\omega|$ denotes the length of the (finite) word ω. In general, if ω is a finite binary word, we say that a register contains ω if the register contains $\cdots 000\omega$. We assume $L \subseteq 1 \cdot \{0, 1\}^*$ because a vector machine cannot distinguish among inputs ω, 0ω, 00ω, etc.

We let TM denote the class of *nondeterministic* Turing machines with two tapes, one a read-only input tape and the other a read/write work tape. A Turing machine is given input ω by writing $¢\omega¢$ on the input tape with the work tape entirely blank. The *time* of a Turing machine computation is its length; the *space* of a computation is the number of squares visited by the head on the work tape. Our definition of time (space) bounded language acceptance for Turing machines is analogous to that for vector machines. See [6] for further discussion of Turing machines and their computations.

DEFINITION. Let $\mathscr{C}$ denote a class of machines (e.g., $\mathscr{V}$ or TM). $\mathscr{C}$-TIME$(F(n))$ ($\mathscr{C}$-SPACE$(F(n))$) denotes the class of languages $L \subseteq 1 \cdot \{0, 1\}^*$ such that some member of $\mathscr{C}$ accepts L within time (space) $F(n)$.

$\mathscr{C}_D$ denotes the subclass of deterministic machines in $\mathscr{C}$.

In Sections 6–8, we show how vector machines and Turing machines can simulate one another, keeping careful track of how the resources used by the simulating machine depend on those used by the simulated machine. However, for the discussion of Sections 4 and 5, it is convenient to introduce notation which captures the idea of "simulation to within a polynomial." If $\mathscr{C}_1$ and $\mathscr{C}_2$ denote classes of machines and RES_1 and RES_2 each denote either TIME or SPACE, we say $\mathscr{C}_1\text{-RES}_1 \leqslant \mathscr{C}_2\text{-RES}_2$ iff there is a positive integer k such that $\mathscr{C}_1\text{-RES}_1(F(n)) \subseteq \mathscr{C}_2\text{-RES}_2((F(n))^k)$ for all $F(n) \geqslant \log n$. (For definiteness, all logarithms are taken to the base 2.)

3. VM's as Models of Parallelism

One way to view vector machines is as a model of parallel computation. By mentally "transposing" the machine and thinking of each bit position, over all vectors, as forming a small processor, we can consider the four types of instructions as providing facilities for

(i) initializing parts of the processors,
(ii) computing functions within the processors,
(iii) communicating between processors, and
(iv) testing the states of the processors.

Inasmuch as vector machines have an elegant definition, they form an elegant model of parallel computation. In our proofs establishing the power of VM's, we rely primarily on the VM's ability to set up and run exponentially many parallel processes in a polynomial amount of time.

4. Restrictions on VM's

We assumed in the definition of the class $\mathscr{V}$ that $B \uparrow C$ meant B shifted a distance given by C *interpreted as a binary number*. Without further qualifying this assumption we have not been able to characterize satisfactorily the power of vector machines. For example, if A initially contains 1, executing $A \leftarrow A \uparrow A$ n times will yield a binary number larger than

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$$

to height n. Yet for programs that merely accept or reject their input in t steps (as opposed to those that do transduction) we have no evidence contradicting the possibility that an arbitrary vector machine could be simulated with at most a polynomial increase in running time by a VM using vectors of length at most 2^t. Leaving this as an interesting open problem, we shall confine our attention in this paper to those vector machines that, one way or another, keep the length of their vectors to at most k^t after t steps, for some constant k.

One way to guarantee that vectors remain short is simply to forbid computations in which the vectors grow too large. We shall call the class of vector machines restricted in this way $\mathscr{V}_K$. The definition of this class must also take into account the initial length of the input vectors.

DEFINITION. A vector machine R belongs to the class $\mathscr{V}_K$ iff there is a constant k such that during any computation C of R, the length of the contents of any register does not exceed $k^t + n$, where t is the length of C, and $n = \max\{| \nu | \mid \nu$ is the contents of a register in the initial configuration of $C\}$.

A second way is to forbid the use of data subject to being shifted as data for shift distances. This can be implemented by distinguishing two types of data, one of which represents the bit vectors, the other shift distances. In [10] this was achieved by treating the shift distances as numbers rather than vectors. Only shifts and bit operations were permitted for the bit vectors, while only $+$, $-$, and integer-divide-by-two were allowed for the numbers. As Hartmanis and Simon [5] point out, this is a somewhat unusual machine architecture. A cleaner version of the same idea, adopted here, is to have only bit vectors, rather than a mixture of bit vectors and numbers, but to retain the distinction between data used for long shifts and data specifying the shift distance. The latter may only be shifted by plus or minus 1, the former only by data (interpreted as binary numbers) of the latter type. Both types of data admit bit-parallel operations, but the two types obviously cannot be allowed to communicate, except through shifts as just described. (One is tempted here to speculate about hierarchies of such types; each type may only be shifted a distance given by a lower type of data. We conjecture that this extension adds no power, to within a polynomial.) We shall refer to this class of machines as $\mathscr{V}_I$, the I indicating the presence of index registers, our term for those registers holding shift distance data.

DEFINITION. A vector machine is in the class $\mathscr{V}_I$ iff its registers can be partitioned into two disjoint sets, one set called *index registers* and the other called *vector registers*, such that (i) each Boolean operation in the program involves either only index registers or only vector registers; and (ii) each shift instruction is of the form $A \leftarrow B \uparrow\downarrow I$ or $I \leftarrow J \uparrow\downarrow 1$, where A and B denote vector registers, and I and J denote index registers. For language recognition, we require the input register to be a vector register. (Note that restrictions (i) and (ii) guarantee that the contents of a vector register are never assigned to an index register.)

LEMMA 4.1. *If $R \in \mathscr{V}_{\mathrm{I}}$ ($\mathscr{V}_{\mathrm{ID}}$), and each input register of R is a vector register, then $R \in \mathscr{V}_{\mathrm{K}}$ ($\mathscr{V}_{\mathrm{KD}}$). In particular,*

$$\mathscr{V}_{\mathrm{I(D)}}\text{-TIME}(F(n)) \subseteq \mathscr{V}_{\mathrm{K(D)}}\text{-TIME}(F(n)).$$

Proof. Given R, there is a constant c such that vector length in index (vector) registers is bounded above by $c + t$ (resp., $2^{c+t} + n$) after t steps, where n is the length of the longest input. This statement is easily verified by induction on t, and the result follows. ▮

The classes $\mathscr{V}_{\mathrm{I}}$ and $\mathscr{V}_{\mathrm{K}}$ are the principal objects of study in this paper. Our main results are summarized (using the notation $\leqslant$) as follows.

(*)	TM-SPACE $\leqslant \mathscr{V}_{\mathrm{ID}}$-TIME	(Theorem 6.1)
	$\leqslant \mathscr{V}_{\mathrm{K}}$-TIME	(Lemma 4.1)
	$\leqslant$ TM$_{\mathrm{D}}$-SPACE	(Theorem 6.2)
	$\leqslant$ TM-SPACE	(trivial).

Yet a third way to impose an upper bound of k^t on vector length is to interpret the shift distances as unary numbers, say by using the length of the shift distance data as the distance to be shifted, and its sign as the direction. An interesting variation on this idea forms the basis for the results of Hartmanis and Simon [5]. Their idea is to replace shifts by multiplication (of binary numbers). This allows them to shift vectors left, treating the shift distance data just as though they were unary numbers. They accomplish right shifts by shifting left everything but the item to be shifted right, which in a machine with indirect addressing will introduce no more than a factor proportional to the running time and otherwise will introduce only a constant factor. We shall call this class of machines $\mathscr{V}_{\mathrm{M}}$ (for multiplication). Combining our result that TM-SPACE $\leqslant \mathscr{V}_{\mathrm{ID}}$-TIME (Theorem 6.1) with the remarks above, it is straightforward to prove that TM-SPACE $\leqslant \mathscr{V}_{\mathrm{MD}}$-TIME. The main result of Hartmanis and Simon is that the converse also holds; namely, that $\mathscr{V}_{\mathrm{M}}$-TIME $\leqslant$ TM$_{\mathrm{D}}$-SPACE. Thus, $\mathscr{V}_{\mathrm{M}}$-TIME and $\mathscr{V}_{\mathrm{MD}}$-TIME also fit into the cycle (*) above. (By employing the multiplication algorithm of Proposition 5.3, this result that $\mathscr{V}_{\mathrm{M}}$-TIME $\leqslant$ TM$_{\mathrm{D}}$-SPACE is an immediate corollary of our Theorem 6.2.)

One drawback of the multiplication model is that it misleadingly focuses attention on the arithmetic operations provided, making it appear that the power of the machine is in some sense a comment on the power of multiplication. As we remarked earlier, the power really comes from cooperation between the logical and the shift operations; the arithmetic is just an obscure way of supplying one half of the source of parallelism. The convolution one normally thinks of as the source of real power in multiplication turns out not to play any role in the proof that TM-SPACE $\leqslant \mathscr{V}_{\mathrm{MD}}$-TIME. If numeric register machines with multiplication, but *without* bitwise Boolean operations, were as powerful as vector machines, then it *would* say something about the power of arithmetic.

In fact, there is evidence to support the conjecture that the Boolean operations play an essential role. Let $E = 1 \cdot \{0, 1\}^* \cdot 0$ be the set of binary representations of even positive integers. Let $\mathscr{V}_A$ be the class of (nondeterministic) register machine programs containing only instructions to load constants, test for zero, add, subtract, and multiply. Then we have the following; details will appear in a forthcoming paper.

THEOREM. *If $R \in \mathscr{V}_A$, and R accepts E within time $T(n)$, then there is a constant $c > 0$ such that $T(n) > cn$ for all n.*

However, it is clear that instructions to load constant 1, bitwise *and*, and test for zero are sufficient to accept E within constant time. Also, E can be accepted by a Turing machine within constant space. In particular, Theorem 6.1 (cf. (*)) is not true with $\mathscr{V}_A$ in place of $\mathscr{V}_{ID}$.

Hartmanis and Simon also suggest using concatenation of the significant digits of two vectors, e.g., $\cdots 00101$ *concat* $\cdots 00110$ is $\cdots 00101110$. It is not difficult to see how this could be used similarly to implement shifts. We find this machine more attractive than the multiplication machine, as it is less misleading.

5. Arithmetic

On commercial computers the above repertoire of instructions is by no means exhaustive. For our purposes, however, we do not need the other instructions (e.g., addition and multiplication of binary numbers) as we are concerned with characterizing the power of vector machines only to within polynomial time loss, and such instructions can be simulated with polynomial overhead using the above set. Let $\mathscr{C}_+$ denote the class $\mathscr{C}$ augmented by an instruction for binary addition. The results of this section, together with Lemma 4.1, establish the following.

THEOREM 5.1

$$\mathscr{V}_{M+}\text{-TIME} \leqslant \mathscr{V}_{I}\text{-TIME};$$
$$\mathscr{V}_{I+}\text{-TIME} \leqslant \mathscr{V}_{I}\text{-TIME};$$
$$\mathscr{V}_{K+}\text{-TIME} \leqslant \mathscr{V}_{K}\text{-TIME};$$

each inequality holding also in the deterministic case.

The results of this section are not necessary to the sequel.

PROPOSITION 5.1. *There is a machine in the class $\mathscr{V}_{\mathrm{ID}}$ which recognizes negative vectors within time $O(\log n)$, where n is the length of the vector.* (*A* negative *bit vector is ultimately* 1, nonnegative *ultimately* 0.)

Proof. The following procedure leaves X nonzero if and only if X was initially negative.

$$I \leftarrow 1;\ \text{while } X \neq 0 \text{ and } {\sim}X \neq 0 \text{ do } (X \leftarrow X \downarrow I;\ I \leftarrow I \uparrow 1). \quad \blacksquare$$

PROPOSITION 5.2. *The sum of two nonnegative binary numbers can be computed on a vector machine in $\mathscr{V}_{\mathrm{ID}}$ within a number of steps proportional to the logarithm of the maximum of the lengths of the inputs.*

Proof. The following algorithm implements addition using only the operations $\wedge$, $\vee$, $\oplus$, $\uparrow$, and test for zero. We leave to the reader the straightforward task of verifying its correctness.

```
$G \leftarrow A \wedge B$; $P \leftarrow A \vee B$;      ¢ Generate and propagate info ¢
$I \leftarrow 1$;                                        ¢ $I$ takes on values 1, 2, 4, 8,... ¢
while $P \neq 0$ do                                      ¢ Propagate carries through $G$ ¢
   ($G \leftarrow G \vee ((G \uparrow I) \wedge P)$;        ¢ Spread $G$ left ¢
   $P \leftarrow P \wedge P \uparrow I$;                    ¢ Clear used $P$ to avoid crosstalk ¢
   $I \leftarrow I \uparrow 1$);                            ¢ Double propagation distance ¢
$A \oplus B \oplus (G \uparrow 1)$. ∎
```

It is interesting to note that only monotonic operations were required to compute the vector G of carries.

COROLLARY 5.1. *Negation may be performed in $O(\log n)$ steps.*

Proof. Combine the fast sign-test algorithm with the identity $-x = ({\sim}x) + 1$ for two's complement negation. (We assume two's complement notation for no especial reason. One's complement notation admits a constant time negation algorithm, since $-x = {\sim}x$.)

COROLLARY 5.2. *Addition of either positive or negative numbers may be performed in time $O(\log n)$ by a machine in $\mathscr{V}_{\mathrm{ID}}$.*

PROPOSITION 5.3. *Multiplication of nonnegative binary numbers may be performed by a machine in $\mathscr{V}_{\mathrm{ID}}$ within time $O(\log n)$, where n is the length of the result.*

Proof. Without loss of generality, assume that $|A| = |B| = m =$ some power of 2, where A and B are the operands to be multiplied. Concatenate m copies of the $2m$ low-order digits of A, calling the result A^B. Concatenate $2m$ copies of the m low-order digits of B, calling the result B^A. Using the method of Section 7.2, this takes time $O(\log m)$. Thinking of B^A as a $2m$ by m matrix stored by rows, transpose B^A (See Section 7.1) and bitwise *and* the result with A^B, calling the result U. That is, $U \leftarrow \text{transpose}(B^A) \wedge A^B$. Imagine that U is structured into m fields, each of length $2m$. The product of A and B is just the sum of the m binary numbers in these m fields with appropriate displacements; i.e., the number in the ith field $(0 \leqslant i < m)$ is shifted left i before it is added into the sum.

It is easy to compute this sum within $O(\log m)$ steps on a machine in $\mathscr{V}_{\text{ID}+}$. First construct the mask $M = 1^{m^2}0^{m^2}$. Using M, we mask out the leftmost $m/2$ fields of U and shift these fields right $m^2 - m/2$, calling the result V. (The shift m^2 places them below the rightmost $m/2$ fields, and the shift $-m/2$ supplies the displacement mentioned above.) Execute

$$U \leftarrow (U + V) \wedge {\sim}M$$

so that U now contains $m/2$ fields. (Since the field width is $2m$, there is no interaction between fields when this addition is performed.) Now shift M right $m^2/2$, use M to mask out the leftmost $m/4$ fields of U, shift these fields right $m^2/2 - m/4$, perform the addition, and so on. After $\log_2 m$ iterations of this process, the rightmost field of U contains the desired product. This method takes time $O(\log m)$ on a machine in $\mathscr{V}_{\text{ID}+}$ and, therefore, time $O(\log^2 m)$ on a machine in $\mathscr{V}_{\text{ID}}$.

To reduce the time to $O(\log m)$ for $\mathscr{V}_{\text{ID}}$, we use the well-known technique of carry-save addition. Given three binary numbers X, Y, Z, the binary numbers $S = X \oplus Y \oplus Z$ and $C = ((X \wedge Y) \vee (Y \wedge Z) \vee (Z \wedge X)) \uparrow 1$, satisfy $S + C = X + Y + Z$. (On a bit-by-bit basis, think of S as the sum and C as the carry when adding three bits together. This instance of unary to binary conversion is called a full adder.) Given *four* binary numbers W, X, Y, Z, applying this method twice allows us to compute S and C such that $S + C = W + X + Y + Z$. In the multiplication procedure just described, replace U and V by U_S, U_C, V_S, and V_C. The addition $U + V$ at each iteration is replaced by the carry–save operations to compute a "new" U_S and U_C from the "old" U_S, U_C, V_S, and V_C. After $\log_2 m$ iterations, at fixed time per iteration, add U_S and U_C by the procedure of Proposition 5.2 to yield the desired product. ▌

6. The Characterization and Consequences

Our main results, which were summarized in (*) in Section 4, are stated as two theorems.

Theorem 6.1. $\text{TM-SPACE}(S(n)) \subseteq \bigcup_{c>0} \mathscr{V}_{\text{ID}}\text{-TIME}(c \cdot (S(n) + \log n)^2)$.

THEOREM 6.2. $\mathscr{V}_K$-TIME$(T(n)) \subseteq$ TM$_D$-SPACE$(T(n) \cdot (T(n) + \log n))$.

Theorems 6.1 and 6.2 are proved in Sections 7 and 8, respectively. First, we examine some immediate corollaries of these results.

The first corollary shows that, for vector machines in the classes $\mathscr{V}_I$ and $\mathscr{V}_K$, nondeterministic time is polynomially related to deterministic time. In particular, $P = NP$ (cf. [1, 7]) for $\mathscr{V}_I$ and $\mathscr{V}_K$.

COROLLARY 6.1. *Let* $T(n) \geqslant \log n$.

$$\mathscr{V}_I\text{-TIME}(T(n)) \subseteq \bigcup_{c>0} \mathscr{V}_{ID}\text{-TIME}(c(T(n))^4).$$

Proof. Immediate from Theorems 6.1 and 6.2 and Lemma 4.1. ∎

Remark. By Lemma 4.1, Corollary 6.1 is true with K in place of I. The exponent 4 can possibly be reduced by a direct simulation. If it can be reduced to 2, then, together with Savitch's [11] result that TM-SPACE$(S(n)) \subseteq$ TM$_D$-SPACE$((S(n))^2)$, we could say that a deterministic X can accept any set accepted by a nondeterministic Y, for $X, Y \in \{$space bounded TM$\} \cup \{$time bounded $\mathscr{V}_I\}$, with the bound being at most squared.

Another corollary follows immediately from Theorem 6.1, since all context-free languages are in TM$_D$-SPACE$(\log^2 n)$ [9], and all context-sensitive languages are in TM-SPACE(n) [8].

COROLLARY 6.2. *If* $L \subseteq 1 \cdot \{0, 1\}^*$ *is context-free, then* $L \in \mathscr{V}_{ID}$-TIME$(c \cdot \log^4 n)$ *for some c. If* $L \subseteq 1 \cdot \{0, 1\}^*$ *is context-sensitive, then* $L \in \mathscr{V}_{ID}$-TIME$(c \cdot n^2)$ *for some c.*

An interesting question is the relationship between the time required to perform a computation in a deterministic serial fashion and the time required by an unbounded parallel method. Can one always obtain a "polynomial in log" time improvement by going from serial to parallel computation? If we equate vector machines with parallel computation, then this question is equivalent to an open question concerning the "time-storage trade-off" relation [2] for Turing machines.

COROLLARY 6.3. *The following statements are equivalent.*

(1) *There is a k such that for all* $T(n) \geqslant n$,

$$\text{TM}_D\text{-TIME}(T(n)) \subseteq \bigcup_{c>0} \mathscr{V}_{ID}\text{-TIME}(c \cdot (\log T(n))^k).$$

(2) *There is a k such that for all* $T(n) \geqslant n$,

$$\text{TM}_D\text{-TIME}(T(n)) \subseteq \text{TM}_D\text{-SPACE}((\log T(n))^k).$$

Cook [2] has conjectured that (2) is false, even if $T(n)$ is restricted to be a polynomial.

7. Proof of Theorem 6.1

The proof of Theorem 6.1 proceeds by four main steps. First, we show that a vector machine in the class $\mathscr{V}_{\text{ID}}$ can transpose a Boolean matrix within time proportional to the logarithm of its size. Given this transposition procedure, it is easy to compute the product of $m \times m$ Boolean matrices within time $O(\log m)$. This matrix multiplication procedure is then used to compute the transitive closure of an $m \times m$ Boolean matrix within time $O(\log^2 m)$. Finally, a vector machine in $\mathscr{V}_{\text{ID}}$ simulates a space $S(n)$ bounded Turing machine M by first constructing the one-step transition matrix for instantaneous descriptions (i.d.'s) of M, and then computing the transitive closure of this matrix. A space $S(n)$ bounded computation of M on an input of length n can involve at most $m = n \cdot c^{S(n)}$ different i.d.'s for some constant c. Since the transition matrix is $m \times m$, the transitive closure computation takes time $O((S(n) + \log n)^2)$.

In the vector machine programs described in this section, we let $U, V, \ldots, Z$ (possibly subscripted) denote vector registers, and $I, J, K, \ldots, P, Q$ denote index registers.

Since index registers are used primarily to hold shift distances, comprehension of the programs is enhanced by thinking of index registers as containing integers (rather than binary representations). Recall that the operations $I \uparrow 1$ and $I \downarrow 1$ perform on integers the operations $2I$ and $\lfloor I/2 \rfloor$, respectively. Vector registers should be viewed as containing binary words. If ω is a word and m is a positive integer, ω^m denotes the word $\omega\omega\omega\cdots\omega$ (m times). We consistently denote integers (words) by lowercase Roman (Greek) letters.

7.1. *Matrix Transposition*

Throughout this paper the Boolean matrices we deal with each reside in a single vector. As such, they appear as one-dimensional matrices A (vectors) with elements a_i. However, we shall interpret them as n-dimensional $d_{n-1} \times d_{n-2} \times \cdots \times d_1 \times d_0$ matrices by decomposing i as $(\cdots(i_{n-1}d_{n-2} + i_{n-2})\, d_{n-3} + \cdots)\, d_0 + i_0$, where $0 \leqslant i_j < d_j$ for each j. This is exactly how n-dimensional arrays are mapped by compilers into linear storage. Furthermore, we shall insist that the dimensions d_j always be a power of 2; then if i is written in binary notation, its decomposition amounts to no more than the identification of n "fields" (contiguous substrings) of i whose concatenation yields i. We write

$$a_i = a_{i_{n-1}i_{n-2}\cdots i_1 i_0}\,.$$

The *transpose of A about coordinates p and q*, written $T_{pq}(A)$, is a $d_{n-1} \times d_{n-2} \times \cdots \times d_{q+1} \times d_p \times d_{q-1} \times \cdots \times d_{p+1} \times d_q \times d_{p-1} \times \cdots \times d_0$ matrix A' satisfying

$$a'_{i_{n-1}\cdots i_{q+1}i_p i_{q-1}\cdots i_{p+1}i_q i_{p-1}\cdots i_0} = a_{i_{n-1}\cdots i_{q+1}i_q i_{q-1}\cdots i_{p+1}i_p i_{p-1}\cdots i_0}\,.$$

That is, coordinates p and q ($p < q$) have been interchanged. If $n = 2$, this yields the usual definition of the transpose A^T, namely, $a^T_{ji} = a_{ij}$. Thus, given

$$A = \begin{pmatrix} a & b & c & d \\ e & f & g & h \end{pmatrix},$$

where a–h denote 0's and 1's, we have

$$A^{\mathrm{T}} = \begin{pmatrix} a & e \\ b & f \\ c & g \\ d & h \end{pmatrix}.$$

As bit vectors, these quantities are *abcdefgh* and *aebfcgdh*, respectively, which are moderately complicated permutations of each other. Our objective is to perform these permutations in time $O(\sum_j \log d_j)$.

We implement, in constant time, the primitive operation of transposition about coordinates each of size 2. That is, we show how to compute $a^T_{i'} = a_i$, where i' in binary is i with two bits interchanged. Once we can transpose individual bits of i, it is easy to see how whole blocks of bits can be transposed, in time proportional to the number of affected bits. We remind the reader of the algorithm for exchanging arbitrary-sized blocks β_p and β_q of data (in our case blocks of bits) using the identity

$$\cdots \beta_{p-1}\beta_q\beta_{p+1} \cdots \beta_{q-1}\beta_p\beta_{q+1} \cdots = \cdots \beta_{p-1}(\beta_p{}^{\mathrm{R}}(\beta_{p+1} \cdots \beta_{q-1})^{\mathrm{R}} \beta_q{}^{\mathrm{R}})^{\mathrm{R}} \beta_{q+1} \cdots,$$

and point out that reversing a block of n items takes $\lfloor n/2 \rfloor$ transpositions of pairs of items. Hence, transposing a matrix can be done in time proportional to the number of bits between and including the transposed bits in the binary representation of the index. In the applications to follow, the matrices are cubes (all dimensions of equal size), which simplifies transposition of blocks; it suffices to move two pointers through the two blocks in parallel, transposing a pair of bits at each step. The bits lying between the blocks need not be touched.

Having shown how to apply the index-bit-transposer, we turn now to its implementation. Consider the transposition about coordinates 0 and 2 of the $2 \times 2 \times 2$ cube Z whose linear representation is *abcdefgh*, with $a = z_{111}$, $b = z_{110}, \ldots, g = z_{001}$, $h = z_{000}$, where the indices of z are in binary. The transposition is *aecgbfdh*. If we look in Fig. 1 at what moved where, we observe three types of motion: (i) no motion; (ii) motion right three positions; (iii) motion left three positions.

In case (i), the stationary bits were, not surprisingly, those with bits 2 and 0 (the two bits at either end) the same, namely, 111, 101, 010, and 000, corresponding to a, c, f, and h, respectively. In case (ii), with $b = z_{110}$ and $d = z_{100}$, the bits were 1 and 0, respectively, while in case (iii) they were 0 and 1, respectively. Interchanging these bits gave rise to a left translation of $2^0 - 2^2$ bits for (ii) and $2^2 - 2^0$ bits for (iii). It should be clear that these three cases are all we need to consider in general. Hence the implementation separates the matrix into three components, shifts two components, then reassembles them. If bits p and q are to be transposed, the shift distances are $\pm(2^p - 2^q)$.

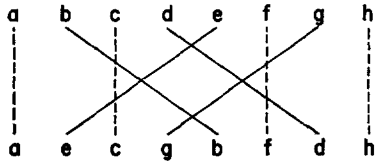

FIG. 1. The transpose T_{02} of the $2 \times 2 \times 2$ cube.

This leaves only the problem of separating the components. We introduce *masks* $\mu_{l,j}$ for $l \geqslant 1$ and $0 \leqslant j < l$; $\mu_{l,j}$ is that binary word of length 2^l whose ith bit is 1 just when the jth bit of i is 1. Hence,

$$\mu_{l,0} = \cdots 10101010,$$

$$\mu_{l,1} = \cdots 11001100,$$

$$\mu_{l,2} = \cdots 11110000, \text{ etc.}$$

(It is no coincidence that the columns are consecutive binary numbers.) If A is a vector of length 2^l, then $A \wedge (\mu_{l,p} \equiv \mu_{l,q})$ is the stationary component, $A \wedge (\mu_{l,p} \wedge {\sim}\mu_{l,q})$ moves left $2^q - 2^p$ positions, and $A \wedge ({\sim}\mu_{l,p} \wedge \mu_{l,q})$ moves left $2^p - 2^q$ positions.

Note that $\mu_{l,l-1}$ can be constructed in $l - 1$ iterations of

$$X \leftarrow ((X \uparrow I) \vee X) \uparrow I; \qquad I \leftarrow I \uparrow 1, \tag{1}$$

where X is initially 10 and I is initially 1. Any other $\mu_{l,j}$ $(0 \leqslant j < l)$ may then be produced using the identity

$$\mu_{l,j-1} = \mu_{l,j} \oplus (\mu_{l,j} \downarrow 2^{j-1}). \tag{2}$$

We can combine all of the above into the following procedure for transposing coordinates p and q, where $d_p = d_q = m =$ some power of 2. The parameters P and Q to the procedure are powers of 2 that identify the leftmost bits of the respective blocks; that is, if block p's leftmost bit is x, then P contains 2^x, and similarly for Q. The parameter M initially contains m, and L contains the length of the vector A being transposed ($=m^n$ if A is an n-dimensional cube).

Transpose (P, Q, L, M):

```
    $Y \leftarrow \mu_{\log L, \log P}$ ; $Z \leftarrow \mu_{\log L, \log Q}$ ;        ¢ Construct masks using (1) and (2) ¢
    do $\log_2 M$ times                                 ¢ Count by halving $M$ each time ¢
      ($A \leftarrow A \wedge (Y \equiv Z) \vee ((A \wedge Y \wedge \sim Z) \uparrow Q) \downarrow P \vee ((A \wedge \sim Y \wedge Z) \uparrow P) \downarrow Q$;
      $P \leftarrow P \downarrow 1; Q \leftarrow Q \downarrow 1$;                        ¢ Move pointers right ¢
      $Y \leftarrow Y \oplus (Y \downarrow P)$;                           ¢ Next mask, cf. (2) ¢
      $Z \leftarrow Z \oplus (Z \downarrow Q)$).
```

Note that we keep only two masks in storage at a time. The procedure *Transpose* (P, Q, L, M) runs within time $O(n \cdot \log m)$. (In subsequent applications of this procedure, we always have $n \leqslant 3$.)

7.2. *Multiplying Boolean Matrices*

Given $m \times m$ Boolean matrices A and B, the objective is to compute C satisfying

$$c_{ij} = \bigvee_k (a_{ik} \wedge b_{kj}).$$

We assume that A and B are stored initially in two vector registers, and that m (=a power of 2) is available in an index register. We shall accomplish the "$\wedge$," for all triples i, j, k, in one operation, and the "$\vee$" in $O(\log_2 m)$ operations. Clearly, at least m^3 bits must participate if the m^3 *and*'s are all to be done at once. This is done by expanding each of A and B to $m \times m \times m$ matrices (i.e., vectors of length m^3) A' and B' satisfying

$$a'_{kij} = a_{ik}, \qquad b'_{kij} = b_{kj}.$$

Then $C' = A' \wedge B'$ contains all the products necessary to form C.

We now show how to carry out this expansion. Let A'' denote an $m \times m \times m$ matrix satisfying

$$a''_{jik} = a_{ik}.$$

Since (from the vector viewpoint) A'' is just m copies of A concatenated together, A'' can be formed in $\log_2 m$ iterations of $(A \leftarrow A \vee (A \uparrow I); I \leftarrow I \uparrow 1)$, with I initially m^2. We may now form A' from A'' by transposing about 0 and 2, so that

$$a'_{kij} = a''_{jik} = a_{ik},$$

as desired. Similarly, form B'' satisfying

$$b''_{ikj} = b_{kj}$$

and transpose about 1 and 2, so that, as desired,

$$b'_{kij} = b''_{ikj} = b_{kj} .$$

Hence, we can form C' as $T_{02}(A'') \wedge T_{12}(B'')$, in $O(\log m)$ operations.

The desired product $C = AB$ can now be computed by *or*ing together the m blocks of length m^2 that comprise C', thereby implementing "$\vee_k$". The following suffices, and it, too, takes $O(\log m)$ steps.

$$I \leftarrow m^3,$$

$$\text{do } \log_2 m \text{ times } (I \leftarrow I \downarrow 1;\ C' \leftarrow C' \vee (C' \downarrow I));\ C \leftarrow C' \wedge 1^{m^2}.$$

Therefore, the product C can be computed in $O(\log m)$ steps. (The computation of m^2 and m^3 causes no problem here, because powers of 2 can trivially be multiplied within time proportional to the logarithms of their magnitudes; to wit, successively halve one multiplicand while doubling the other, until the former becomes 1. In the sequel, we implicitly use this ability to multiply powers of 2 in logarithmic time.)

7.3. *Transitive Closure*

If A is an $m \times m$ Boolean matrix, the transitive closure of A is defined by

$$A^* = E \vee A \vee A^2 \vee A^3 \vee \cdots,$$

where E denotes the $m \times m$ identity matrix. It is easy to see that also $A^* = (A \vee E)^m$. A vector machine in the class $\mathscr{V}_{\text{ID}}$ can compute A^* within time $O(\log^2 m)$ by successively squaring the matrix $(A \vee E) \log_2 m$ times, using the matrix multiplication procedure just described in Section 7.2. (Since the vector representation of E is $(0^m 1)^m$, it should be clear that E can be constructed in $O(\log m)$ steps.)

7.4. *Completion of the Proof*

Let M be a nondeterministic Turing machine which accepts a language L within space $S(n)$. Without loss of generality we can assume that M's work tape is one-way infinite to the left, M never moves the work head off the right end of the work tape nor moves the input head outside the area delimited by the endmarkers ¢ on the input tape, and M can accept an input only by entering a unique designated accepting state with the work tape entirely blank and both heads scanning the rightmost squares of their respective tapes. The necessary modifications to M are straightforward; see, for example, [6].

As mentioned earlier, the vector machine which simulates M first constructs the one-step transition matrix for i.d.'s of M. Our formalization of i.d.'s is the following. Say M has states Q and tape alphabet Γ. For positive integer n, let s' be the least power of 2 such that $s' \geqslant S(n) + 1$, and define an *n-i.d.* of M to be a word $\eta\tau$, where $\tau \in \Gamma^* \cdot \Gamma \cdot Q \cdot \Gamma^*$, $|\tau| = s'$, $\eta \in 0^* \cdot 1 \cdot 0^*$, and $|\eta| = n + 2$. Suppose M is given input $\omega \in 1 \cdot \{0, 1\}^*$, where $n = |\omega|$; write $¢\omega¢ = \omega_{n+1}\omega_n \cdots \omega_2\omega_1\omega_0$. Then the n-i.d. $\delta = 0^{n-l+1}10^l\tau_1 q\tau_2$, where $\tau_1\tau_2 \in \Gamma^*$, $q \in Q$, and $0 \leqslant l \leqslant n + 1$, describes the situation where M is in state q, $\tau_1\tau_2$ is written on the work tape, the work head is scanning the rightmost symbol of τ_1, and the input head is scanning ω_l.

In particular, $0^{n+1}1\#^{s'-1}q_0$ $(0^{n+1}1\#^{s'-1}q_a)$ is the unique *initial* (*accepting*) n-i.d., where q_0 (q_a) is the initial (accepting) state and $\#$ denotes the blank tape symbol in Γ.

Let Next_M denote the one-step transition relation defined on i.d.'s of M. If δ and δ' are n-i.d.'s and ω is an input of length n, then $\text{Next}_M(\omega, \delta, \delta')$ iff δ can reach δ' by one step in a computation of M on input ω.

We next define binary words γ_j, some of which serve to code n-i.d.'s of M. Let $\Sigma = Q \cup \Gamma$, let b be the least power of 2 such that $2^b \geqslant \text{card}(\Sigma)$, and choose a one-to-one map $h: \Sigma \rightarrow \{0, 1\}^b$. Extend the domain of h to Σ^* in the obvious way. Now fix a particular n, let s' be as above, and let $s = bs'$, $m' = 2^s$, n' = the least power of 2 such that $n' \geqslant n + 2$, and $m = m'n'$. For $0 \leqslant z \leqslant m' - 1$, let $\rho_z \in \{0, 1\}^s$ be a binary representation of z (possibly with leading zeros to make $|\rho_z| = s$). For $0 \leqslant l \leqslant n' - 1$, let $\mu_l = 0^{n'-l-1}10^l$. The binary words γ_j, $0 \leqslant j \leqslant m - 1$, of length m are now defined as follows. Write $j = n'z + l$, where $0 \leqslant l \leqslant n' - 1$ and $0 \leqslant z \leqslant m' - 1$, and define $\gamma_j = 0^{m-n'-s}\mu_l\rho_z$. (Thus, γ_j also depends on n; we rely on context to specify n.)

If $\delta = \eta\tau$ is an n-i.d. of M as above, then we say that γ_j *codes* δ iff $\eta \cdot h(\tau)$ is a suffix of γ_j. It is obvious that each n-i.d. is coded by some γ_j, and that each γ_j codes at most one n-i.d.

We now describe the operation of a deterministic vector machine $R \in \mathscr{V}_{\text{ID}}$ which accepts L within time $O((S(n) + \log n)^2)$. (In the remainder of the proof, the constant implicit in the "O-notation" depends on M, but not on the input ω.) Let W denote the input register, and assume R receives input ω. Let $n = |\omega|$. For the present, it is convenient to assume that R initially receives also the integer s' (defined above) in some index register. This assumption will later be removed. Given s', the other integers s, m', n', and m defined above can now be computed in index registers within time $O(S(n) + \log n)$, as the reader can easily verify. For example, n' is computed in register I by

$$I \leftarrow 2; \qquad \text{while} \quad (W \uparrow 2) \downarrow I \neq 0 \qquad \text{do} \quad I \leftarrow I \uparrow 1.$$

The first goal of R is to construct an $m \times m \times m$ Boolean matrix A such that for all i and j with $0 \leqslant i, j \leqslant m - 1$, if γ_i codes an n-i.d. δ, then $a_{ij0} = 1$ iff γ_j codes an n-i.d. δ' such that $\text{Next}_M(\omega, \delta, \delta')$. If γ_i does not code an n-i.d., or if $k \neq 0$, then the value of a_{ijk} is unimportant. Thus, R first constructs words

$$\nu_r = (\gamma_{m-1})^m(\gamma_{m-2})^m \cdots (\gamma_1)^m(\gamma_0)^m$$

and

$$\nu_c = (\gamma_{m-1}\gamma_{m-2} \cdots \gamma_1\gamma_0)^m.$$

These words are useful for computing all a_{ij0} in parallel. ν_r and ν_c can be divided into contiguous *segments* of length m such that, for all i and j, a copy of γ_i appears in ν_r and a copy of γ_j appears in ν_c, both in the segment whose rightmost bit occupies position $m^2i + mj$, which is the position a_{ij0} is eventually to occupy.

We first describe the macro *dup*, which is useful both in constructing these words and in subsequent procedures. If X contains the binary word β, and L and K contain integers l and k where k is a power of 2, then dup(X, L, K) halts within time $O(\log k)$ with $\bigvee_{i=0}^{k-1} (\beta \uparrow il)$ stored in X.

```
dup($X, L, K$): while $K > 1$ do
                    ($X \leftarrow X \vee (X \uparrow L)$;
                    $L \leftarrow L \uparrow 1$;
                    $K \leftarrow K \downarrow 1$).
```

The following procedure *layout* is used to construct both ν_r and ν_c. Layout(V, l) constructs

$$\alpha = \gamma_{m-1}0^{l-m}\gamma_{m-2}0^{l-m} \cdots \gamma_2 0^{l-m}\gamma_1 0^{l-m}\gamma_0$$

in register V within time $O(\log m + \log l)$, where the integer l, $l \geqslant m$, is available initially in some index register. *Layout* first constructs

$$\rho_{m'-1}0^{n'l-s}\rho_{m'-2}0^{n'l-s} \cdots \rho_1 0^{n'l-s}\rho_0$$

in register V by the following.

```
$V \leftarrow 0$; $I \leftarrow n'l$; $Z \leftarrow 1 \uparrow I$;
do $\log_2 m'$ times
    ($V \leftarrow (V \vee (V \uparrow I)) \vee Z$;
    $Z \leftarrow ((Z \vee (Z \uparrow I)) \uparrow I) \uparrow 1$;
    $I \leftarrow I \uparrow 1$).
```

Executing dup(V, l, n') now gives $(0^{l-s}\rho_{m'-1})^{n'} \cdots (0^{l-s}\rho_1)^{n'}(0^{l-s}\rho_0)^{n'}$ in V. *Layout* next constructs

$$(0^{l-n'}\mu_{n'-1}0^{l-n'} \cdots \mu_1 0^{l-n'}\mu_0)^{m'}$$

in register X by

```
$X \leftarrow 1$;
dup($X, l+1, n'$);
dup($X, n'l, m'$).
```

Finally, $V \leftarrow V \vee (X \uparrow s)$ gives the desired word α in V.

Now ν_r is constructed in V_1 by executing (layout(V_1, m^2); dup(V_1, m, m)). ν_c is constructed in V_2 by (layout(V_2, m); dup(V_2, m^2, m)). Thus, ν_r and ν_c can be constructed within time $O(\log m)$, that is, time $O(S(n) + \log n)$.

R must now compute the a_{ij0}. An exact description of this process is unnecessarily tedious. Our purpose is only to outline the general details sufficiently to allow the reader to construct the remainder easily. First execute:

```
$V_0 \leftarrow W \uparrow (s+1)$;
dup($V_0, m, m^2$).
```

This has the effect of constructing, for each γ_i in V_1, a copy of the input "opposite" the part of γ_i which codes the position of the input head. For example, Fig. 2 shows

$$\begin{array}{lcccccccccccccccccccc}
V_0: & 0 & \cdots & 0 & 0 & 0 & \omega_n & \multicolumn{3}{c}{\cdots\cdots\cdots\cdots} & \omega_l & \multicolumn{2}{c}{\cdots\cdots} & \omega_2 & \omega_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\
V_1: & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & c_s & \cdots & c_3 & c_2 & c_1 \\
V_2: & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & c_s' & \cdots & c_3' & c_2' & c_1' \\
V_3: & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0
\end{array}$$

FIG. 2. Segment$_0$.

the segment of V_0, V_1, and V_2 in which γ_{i_0} appears in V_1 and γ_{j_0} appears in V_2, for a particular i_0 and j_0, (cf. the definition of ν_r and ν_c above). In Fig. 2, $\omega = \omega_n \cdots \omega_2\omega_1$ is the input, and $\gamma_{i_0} = 0\cdots0010^l c_s \cdots c_2c_1$ and $\gamma_{j_0} = 0\cdots0010^{l'} c_s' \cdots c_2'c_1'$, where $c_k, c_k' \in \{0, 1\}$ for $1 \leqslant k \leqslant s$.

To simplify the discussion, we concentrate attention on this segment (henceforth called segment$_0$). The process we describe for computing $a_{i_0j_00}$ in segment$_0$ is actually being performed in parallel on all segments to compute all the a_{ij0}.

Figure 2 also shows segment$_0$ of register V_3 which contains $(0^{m-s-n-2}10^n10^s)^{m^2}$. The following program constructs this word within time $O(\log m)$.

```
$V_3 \leftarrow W \uparrow 1$;
dup($V_3, -1, n'$); ¢ $V_3$ contains $1^{n+1}$ ¢
$V_3 \leftarrow V_3 \oplus (V_3 \uparrow 1)$; ¢ $V_3$ contains $10^n1$ ¢
dup($V_3 \uparrow s, m, m^2$).
```

The next goal is to compute bits which determine the input symbol scanned by (the n-i.d. coded by) γ_{i_0}, and the direction the input head moved in going from γ_{i_0} to γ_{j_0}, and then to spread these bits over the rightmost s bits in segment$_0$. Let d be the least power of 2 such that $d \geqslant s + n'$. Execute:

$$V_0 \leftarrow V_0 \wedge V_1 ;$$
$$V_3 \leftarrow V_3 \wedge V_1 ;$$
$$\text{dup}(V_0, -1, d);$$
$$\text{dup}(V_3, -1, d).$$

The effect here is to spread 1's over the rightmost s bits of segment$_0$ in V_0 (resp., V_3) iff $\omega_l = 1$ (resp., $\omega_l = ¢$). (See Fig. 2.)

To compute bits for the input head motion, construct $(1^{m-s}0^s)^{m^2}$ in register Z and execute:

$$V_4 \leftarrow (V_1 \downarrow 1) \wedge V_2 \wedge Z;$$
$$V_5 \leftarrow V_1 \wedge V_2 \wedge Z;$$
$$V_6 \leftarrow V_1 \wedge (V_2 \downarrow 1) \wedge Z;$$

followed by dup$(V_k, -1, d)$ for $k = 4, 5, 6$. This spreads 1's across V_4 (resp., V_5, V_6) iff $l' = l - 1$ (resp., $l' = l$, $l' = l + 1$).

Now assume that γ_{i_0} codes an n-i.d. δ of M. The rightmost s bits of segment$_0$ contain all the information required to determine if γ_{j_0} codes an n-i.d. δ' such that Next$_M(\omega, \delta, \delta')$. It is easy to see that, given the input symbol and the input head shift information, each neighborhood of three symbols $\sigma_1\sigma_2\sigma_3 \in \Sigma^3$ in an n-i.d. δ determines a set $N_M(\sigma_1\sigma_2\sigma_3) \subseteq \Sigma^3$ such that some $\sigma_1'\sigma_2'\sigma_3' \in N_M(\sigma_1\sigma_2\sigma_3)$ must occupy the same neighborhood in any n-i.d. δ' which follows in one step from δ. (See, for example, [14].) R can thus examine each such neighborhood in $V_0 - V_6$, and determine whether the neighborhood is consistent with a legal move of M. This information is collected at the rightmost bit of segment$_0$ to constitute $a_{i_0 j_0 0}$. This process can clearly be performed within $O(s)$, that is, $O(S(n))$ steps (in fact, $O(\log s)$ steps are sufficient), and can be done in parallel for all segments to compute all the a_{ij0}. We let the reader supply any further details required to convince himself that the transition matrix A can be computed within time $O(S(n) + \log n)$.

By two applications of the transposition procedure, followed by an appropriate mask, we can construct within $O(\log m)$ steps an $m \times m$ matrix B satisfying $b_{ij} = a_{ij0}$. Now R calls the transitive closure procedure of Section 7.3 which computes B^* within $O(\log^2 m)$, that is, $O((S(n) + \log n)^2)$ steps. If integers e and f are such that γ_e (γ_f) codes the initial (accepting) n-i.d. of M, then M accepts ω iff $b^*_{ef} = 1$. It is not hard to see that e and f, and then $me + f$, can be computed within $O(\log m)$

steps. The bit b^*_{ef} can thus be extracted and tested. This completes the description of R under the assumption that s' is available initially.

If s' is not available, then R runs the entire procedure described above, with s' taking on successive powers of two, until it is discovered that M accepts the input. The first attempted value of s' should be $l = 2^{\lceil \log \log n \rceil}$ (which can be computed in $O(\log n)$ steps). If M does accept the input of length n, then R will discover this fact when $s' = l \cdot 2^u$, where $u = \lceil \log((S(n) + 1)/\log n) \rceil$. It follows that, when computing on accepted inputs of length n, R runs within time

$$\sum_{k=0}^{u} c \cdot (l \cdot 2^k + \log n)^2 \leqslant c' \cdot (S(n) + \log n)^2$$

for constants c and c' independent of n. (Recall that $l = O(\log n)$.) This completes the proof of Theorem 6.1.

Remark. The vector machine R above accepts L within space $O(m^3)$, that is, space $O(n^3 d^{S(n)})$ for some constant d depending on M. The time bound $O((S(n) + \log n)^2)$ still holds if R's shift distances are restricted to be powers of 2.

8. Proof of Theorem 6.2

Let R be a nondeterministic vector machine in the class $\mathscr{V}_K$ which accepts a language L within time $T(n)$. Assume the registers appearing in R's program are labeled $V_0, V_1, ..., V_m$ for some m; say V_0 is the input register. Let c be the constant such that vector length does not exceed $c^t + n$ after t steps, where n is the input length.

We describe the operation of a deterministic Turing machine M which accepts L within space $a \cdot T(n) \cdot (T(n) + \log n)$ for some constant a. By the classical constant factor "speedup" result [13], M can be modified to operate within space $T(n) \cdot (T(n) + \log n)$. For the moment, it is convenient to assume that $T(n)$ is tape constructable (cf. [6]); that is, given any input of length n, M can first delimit a block of $T(n)$ tape squares.

Within space $T(n)$, M can store a "choice sequence," which is a list of decisions made by R at each of its $T(n)$ steps. More precisely, a choice sequence is a word in $\{1, 2, 3, ..., d\}^*$ of length $T(n)$, where d is the outdegree of R's graph (program). If the edges directed out of each vertex are given unique labels from $\{1, 2, 3, ..., d\}$, then a choice sequence specifies, in the obvious way, a path through the graph starting at the designated start vertex. Thus M's outer loop cycles through all possible choice sequences in lexicographic order, which disposes of the issue of nondeterminism for R.

For each choice sequence, M attempts to make progress through R's graph (program) by following the specified path. While following a particular path, M keeps a counter t

equal to the number of edges followed thus far along the path. As M progresses along a path, M processes the instructions encountered on the edges as follows.

An instruction which changes the contents of a register (i.e., instruction types (i)–(iii) in Section 2) is not executed but is recorded on the work tape together with the current value of t.

If a predicate is encountered, it must be evaluated to determine whether M should continue following this choice sequence or cycle to the next one. Predicates are evaluated with the help of a procedure find(b, i, t) which returns the bth bit of the contents of register V_i at step t, for $0 \leqslant b \leqslant c^{T(n)} + n$, $0 \leqslant i \leqslant m$, and $0 \leqslant t \leqslant T(n)$. To test whether or not $V_i = 0$ at step t, it suffices to compute find(b, i, t) for $0 \leqslant b \leqslant c^t + n$. The recursive procedure *find* is as follows.

Find(b, i, t):

(1) If $t = 0$, then return (if $i = 0$ then the bth symbol of the input else 0);

(2) if the instruction recorded at step t does not change the contents of V_i, then return find$(b, i, t - 1)$;

(3) if "$V_i \leftarrow$ constant" is recorded at step t then return the bth bit of the constant;

(4) if "$V_i \leftarrow V_j \mathbin{@} V_k$" is recorded at step t, where @ denotes a Boolean operation, then return (find(b, j, t) @ find(b, k, t));

(5) if "$V_i \leftarrow V_j \uparrow V_k$" is recorded at step t then:

(5.1) let $B = c^t + n$, and calculate the length l of the contents of V_k at step t by:

```
$l \leftarrow l' \leftarrow 0$;
while $l' \leqslant B - 1$ do
(if find$(l', k, t) \neq$ find$(l' + 1, k, t)$ then $l \leftarrow l' + 1$; $l' \leftarrow l' + 1$);
```

(5.2) if $l > 1 + \log B$ then return find(B, j, t) (note that if $l > 1 + \log B$, then V_k contains an integer z with $|z| > B$. If $z < 0$, then find(B, j, t) is clearly the correct result. If $z > 0$, then V_j must be identically zero, and this is again correct);

(5.3) let $s =$ find(l, k, t), let z denote the integer with binary representation $\cdots sss$ find$(l - 1, k, t)$ find$(l - 2, k, t) \cdots$ find$(0, k, t)$, and

$$\text{return} \begin{cases} \text{find}(b - z, j, t) & \text{if } 0 \leqslant b - z < B, \\ \text{find}(B, j, t) & \text{if } b - z \geqslant B, \\ 0 & \text{if } b - z < 0. \end{cases}$$

This completes the description of M. It remains only to observe that M operates within space $O(T(n) \cdot (T(n) + \log n))$. First, space $O(T(n) \cdot \log T(n))$ is sufficient to record all integers t, $0 \leqslant t \leqslant T(n)$, in binary, together with the instructions executed at these steps. To bound the space used by *find*, first note that each variable, e.g., B, b, t, l, z, can be represented in binary within space $O(\log B)$. In the obvious stack implementation of *find*, each stack frame thus occupies space $O(T(n) + \log n)$. Stack depth is bounded above by $T(n)$, whence the bound $O(T(n) \cdot (T(n) + \log n))$. If $T(n)$ is not tape constructable, then, as in [11], M attempts the above procedure using k tape squares for $k = 1, 2, 3,\ldots$, until it is discovered that R accepts the input. This completes the proof of Theorem 6.2.

Remark. (Generalization to transduction.) Both Theorems 6.1 and 6.2 can be generalized (at least in spirit) to the transduction problem, i.e., the problem of computing a total function $f\colon 1 \cdot \{0, 1\}^* \to 1 \cdot \{0, 1\}^*$. A deterministic vector machine R computes f within time $T(n)$ if, for all ω, when started with ω in an input register, R halts within $T(|\omega|)$ steps with $f(\omega)$ in a designated output register. A Turing machine is given a separate output tape scanned by a one-way write-only head; the space of a computation is counted only on the work tape. We consider only deterministic machines for transduction problems.

The proof of Theorem 6.2 generalizes easily to the transduction problem. When M discovers that R has halted, M simply uses *find* to produce in order the bits of the output register.

The basic outline of the proof of Theorem 6.1 carries over to the transduction problem, although the details are more involved. The t-step transition matrix $A^{(t)}$ for a deterministic Turing machine M now has a binary word α in the (i, j)-position iff the ith i.d. of M can reach the jth i.d. in exactly t steps while producing α on the output tape. It is clear how to define multiplication of such matrices so that $A^{(2t)} = A^{(t)} \cdot A^{(t)}$; however, the implementation of this multiplication is slightly more involved than before because $A^{(t)}$ may contain words of varying lengths. It is possible, however, to carry out such multiplication on a VM within time $O(\log^2 m)$ which gives time $O(\log^3 m)$ for the analog of transitive closure. Using this method, the exponent 2 in Theorem 6.1 becomes 3 for transduction.

Acknowledgments

Michael J. Fischer and Michael O. Rabin were a source of helpful ideas. E. Gilbert suggested a simplification of our addition algorithm.

References

1. S. A. Cook, The complexity of theorem proving procedures, *in* "Proceedings of the Third Annual ACM Symposium on the Theory of Computing" (1971), pp. 151–158.
2. S. A. Cook, An observation on time-storage trade off, *J. Comput. System Sci.* **9** (1974), 308–316.

3. S. A. Cook and R. A. Reckhow, Time bounded random access machines, *J. Comput. System Sci.* 7 (1973), 354–375.
4. J. Hartmanis, Computational complexity of random access stored program machines, *Math. Systems Theory* 5 (1971), 232–245.
5. J. Hartmanis and J. Simon, On the power of multiplication in random access machines, *in* "Proceedings of the Fifteenth Annual Symposium on Switching and Automata Theory" (1974), pp. 13–23.
6. J. E. Hopcroft and J. D. Ullman, "Formal Languages and Their Relation to Automata," Addison–Wesley, Reading, Mass., 1969.
7. R. M. Karp, Reducibility among combinatorial problems, *in* "Complexity of Computer Computations" (R. E. Miller and J. W. Thatcher, Eds.), pp. 85–104, Plenum Press, New York, 1972.
8. S. Y. Kuroda, Classes of languages and linear-bounded automata, *Information and Control* 7 (1964), 207–223.
9. P. M. Lewis II, R. E. Stearns, and J. Hartmanis, Memory bounds for recognition of context-free and context-sensitive languages, *IEEE Conf. Rec. Switching Circuit Theory Logical Design* (1965), 191–202.
10. V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer, A characterization of the power of vector machines, *in* "Proceedings of the Sixth Annual ACM Symposium on the Theory of Computing" (1974), pp. 122–134.
11. W. J. Savitch, Relationships between nondeterministic and deterministic tape complexities, *J. Comput. System Sci.* 4 (1970), 177–192.
12. J. C. Shepherdson and H. E. Sturgis, Computability of recursive functions, *J. Assoc. Comput. Mach.* 10 (1963), 217–255.
13. R. E. Stearns, J. Hartmanis, and P. M. Lewis II, Hierarchies of memory limited computations, *IEEE Conf. Rec. Switching Circuit Theory Logical Design* (1965), 179–190.
14. L. J. Stockmeyer and A. R. Meyer, Word problems requiring exponential time, *in* "Proceedings of the Fifth Annual ACM Symposium on the Theory of Computing" (1973), pp. 1–9.