A Characterization of the Power of Vector Machines

Vaughan R. Pratt
Michael O. Rabin †
Larry J. Stockmeyer
Massachusetts Institute of Technology
Cambridge, Massachusetts

1. Introduction

Random access machines (RAMs) are usually defined to have registers that hold integers. While this captures in part the structure of a commercial computer, it overlooks an implementation-dependent feature of most binary oriented machines, namely their ability to operate bit by bit on the bit vectors used to represent integers. Typical operations are bit-wise Boolean operations (and,or,not, etc.) and shifts by an amount specified in some register. These operations are ideal for certain problems, such as dealing with sets represented as bit vectors, some parsing algorithms [4], propositional calculus theorem proving, and analysis of sorting networks. A RAM so implemented we shall call a vector machine.

Just as one allows RAM registers to contain any integer, one would like a vector machine register to contain any bit pattern, which we may consider to be semi-infinite to the left, with the least significant bit at the right-hand end. We shall show that the bit operations in such a machine provide a remarkable amount of computing power. In fact, we can provide relatively tight upper and lower bounds on the power of vector machines in terms of the power of space-bounded Turing machines. Any set accepted in space $S(n)$ by a non-deterministic Turing machine may be accepted in time $c_1S^2(n)$ (some constant c_1) by a deterministic vector machine. Conversely, any set accepted in time $T(n)$ by a non-deterministic vector machine may be accepted in space $c_2T^2(n)$ (some constant c_2) by a deterministic Turing machine. An immediate corollary is that, on a vector machine, the sets accepted in non-deterministic polynomial time can all be accepted in deterministic polynomial time. Thus the analogue for vector machines of the Cook-Karp $P=NP$ question for Turing machines is settled in the affirmative.

Vector machines are worth studying for at least two reasons. The fact that they are derived from RAMs in such a plausible way should make us more cautious of algorithms whose impressive performance is only achieved on a "not-quite-but-almost" standard RAM. Now that we know this variant is so powerful, we might justifiably steer clear of it. Alternatively, we might try to build one. The simple and regular structure of a vector (and,not, shifts by powers of two and and a test for zero are a sufficient set of operations for them) makes them attractive from a manufacturing point of view, unlike Illiac IV, whose regular structure appears at such a macroscopic level that semi-conductor technology cannot take advantage of it. For the traveling salesman problem and the like, construction of sufficiently long vectors may be too expensive at present. For transitive closure of Boolean matrices and context-free parsing, for which our techniques yield times of $O(\log^2 n)$ and $O(\log^4 n)$ respectively requiring vectors of length $O(n^2)$ and $O(n^4)$ respectively, the requisite lengths may be economically feasible.

This research was supported by the National Science Foundation under research grant GJ-34671.

† Present address: Hebrew University, Jerusalem, Israel.

2. Definitions

In this section we describe the essential features of a vector machine. Although the motivation in the previous section suggests that a vector machine should simply be a RAM with additional bit-wise Boolean and shift operations, we have not succeeded in finding any reasonable upper bound on the power of such a machine. The difficulty is that very rapidly growing functions may be computed simply by repeatedly shifting a vector a distance equal to its value. We have as yet found no use for the particular functions so computable; neither have we found a way to compute them in a small amount of space on a Turing machine. Hence we distinguish vector registers and index registers, the latter being standard RAM equipment containing natural numbers but not bit vectors. While this weakens our original motivation, in practical terms it makes vector machines more attractive from a constructor's point-of-view since it simplifies the hardware associated with a vector register.

Let us now define the registers, operands, operations, and predicates of a vector machine. There are two sets of registers, $I_0, I_1, I_2, \ldots$ (index registers) and $V_0, V_1, V_2, \ldots$, (vector registers). For each i, I_i contains a non-negative integer while V_i contains a bit vector semi-infinite to the "left", all but a finite number of whose bits are the same. The length of a vector is the number of significant digits in it; thus the lengths of 0,1,10,11,100,... are 0,1,2,2,3,...respectively. For indirect addressing, II_i and VI_i refer respectively to I_k and V_k, where k is the contents of I_i.

The following table summarizes the relevant objects and what they may consist of.

I-constant:	any non-negative integer
V-constant:	any bit vector semi-infinite to the left with all but a finite number of bits the same.
I-register:	I_i , II_i , $i \geq 0$
V-register:	V_i , VI_i , $i \geq 0$
I-operand:	I-constant, I-register
V-operand:	V-constant, V-register
Operation:	I-register ← I-operand $\pm$ I-operand
	I-register ← $\lfloor$I-operand/2$\rfloor$
	V-register ← V-operand f V-operand (any Boolean function f of two variables)
	V-register ← V-operand $\downarrow\uparrow$ I-operand (shift left ($\uparrow$) or right ($\downarrow$) by value of I-operand).
Predicate:	I-operand $=, \neq, <, \geq, \leq, >$ I-operand
	V-operand $=, \neq$ V-operand

A vector machine is a set of index and vector registers, and a directed graph each of whose edges is labelled with a predicate or an operation as defined above. One vertex of the graph is distinguished as the start vertex, and a subset of the vertices is distinguished as accepting vertices. Informally, a computation of a vector machine is a path through its graph starting from the start vertex and proceeding at each step via an edge whose predicate is satisfied by the state (register contents) of the machine (as defined by the initial state and the computation thus far) or whose operation is used to update the state of the machine. Initially, V_0 contains the input and all other registers are zero. We leave to the reader the task of formalizing the definition of a computation. An accepting computation is a computation whose final vertex is an accepting vertex. The time of a computation is its length. The space of a computation is the maximum, over all states of that computation, of the sum of the lengths of the vectors in each state.

A deterministic vector machine is one such that for each machine state and each vertex, at most one edge leaving that vertex may be followed when the machine is in that state. We shall sometimes refer to vector machines without this restriction as non-deterministic vector machines.

3. Programming Examples

The complexity results to follow have an abstract flavor that may discourage practically minded people from taking vector machines seriously. After all, the theoretician who scoffs at a time T^2 simulation loss can scarcely appreciate the economics of the world outside.

To demonstrate that vector machines can operate in valuable time bounds we describe linear-time algorithms for establishing satisfiability of propositional calculus formulas in conjunctive normal form, and for testing whether a given sorting network works. Floyd (conversation) has independently discovered the latter algorithm, and the former arises in an obvious way from the method of truth tables, so no particular novelty on our part attaches to the method of either; our point is that a linear time bound is possible for both problems. We also give a time $O(\log n)$ multiplication algorithm for $n \times n$ matrices.

We begin with the satisfiability tester. The method of truth tables is to try every combination of assignments of truth values to the variables of the formula, and for each assignment to evaluate the truth of the formula. Given n variables, the 2^n evaluations may all be carried out in parallel using the bitwise parallelism of the vector machine. With each assignment we associate one position in a vector. A uniform way to do this is to represent assignments as n-bit integers in the obvious way which in turn directly give the corresponding bit position, counting the least significant bit as zero. Then the i-th variable's value for assignment j is given by the j-th digit of the vector of 2^n bits formed by repeatedly concatenating copies of the pattern $0^{2^i}1^{2^i}$. The value of the formula is computed in exactly the same way as when each variable is only assigned one truth value, except that all operands are now bit vectors of 2^n bits instead of individual truth values.

The only trick required is the generation of all n patterns for the n variables in $O(n)$ steps. This is done in two stages. The pattern for the $(n-1)$-st variable is simply $1^{2^{n-1}}$. The pattern for the j-th variable can be computed in $O(1)$ steps from that for the $(j+1)$-st variable, by a left shift 2^j places and an exclusive or (which we write as $\oplus$). Hence the two stages, each taking $O(n)$ operations are: (i) determine the $(n-1)$-st variable's pattern; (ii) determine all the others'.

We now give the details of the program. We assume that the formula to be tested is in conjunctive normal form and is in V_0. If the contents of V_0 are expressed in octal, the octal digits denote: 2:0, 3:1, 4:$\wedge$, 5:$\vee$, 6:$\neg$. No other octal digits may appear within the formula. Beyond the formula, all bits of V_0 must be zero. Variables are represented as a string of 0's and 1's (octal 2's and 3's in V_0) whose reverse represents a binary integer i identifying the variable x_i. The formula $0\vee\neg1\vee01\wedge1\vee\neg01$ should be parsed to read $(x_0 \vee \bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$ (or equivalently, its reverse).

The following program enters an accepting state when started in the start state precisely when V_0 contains a satisfiable formula. The high-level notation should be self-explanatory.

A useful "macro" for most of the algorithms of this paper is one to copy an integer from V_0 to R_0.

```
copy :      R0←0;
            while  V0 ∧ 6 = 2 do                  test for binary digit symbol
               (R0 ← 2R0;                         "shift"  R0 left
               if  V0 ∧ 1 = 1  then R0←R0+1;      Copy the bit in  V0.
               V0 ← V0 ↓ 3).                      Next symbol
```

Whenever the expression "copy"occurs in a program, the above definition of copy is to be substituted for it.

Another useful macro puts a block of 2^n 1's in V_0.

```
block n:   V0 ← 1;  R2 ← 1;
           for R0 from n by -1 to 1 do
               V0 ← V0 ∨ (V0 ↑ R2) ;
               R2 ← R2 + R2).
```

Wherever, say, "block R1" appears in a program, it is displaced by the definition, with all occurrences of "n" replaced by "R1". Clearly "block n" takes O(n) steps. At termination, the value of R_2 is 2^n.

The set of patterns that are repetition of $0^{2^j}1^{2^j}$ for $0 \le j \le n$ are stored in the array R_{j+8} $(0 \le j \le n)$ as follows. We assume n is in whatever register is named when we invoke patternset, e.g. we might write "patternset R_3" in a program.

```
patternset n: block n;  R1 ← R2;
              for R0 from n+8 by -1 to 8 do
                  (VR0 ← V0;  R1 ← R1/2;  V0 ← V0 ⊕ (V0↑ R1)).
```

(Recall that "⊕" is exclusive-or.

It is easy to see that "patternset n" takes O(n) steps. Note that R_2 still contains 2^n on termination, because of "block n".

We now give the main program for the satisfiability checker.
check: First initialize everything

```
V1←V0;                                  Save a copy of input
R1←0;                                   Initialize the "largest variable" reg
while V0≠0 do                           Scan the formula
   (copy;                                   Copy variable name to R0.
    if R0>R1 then R1←R0;                    Update "largest variable".
    while V0∧4≠0 do V0←V0↓3)                Skip delimiters
patternset R1;                          Build all patterns
Then evaluate the formula
V3 ← ¬ 0;                               Initialize "∧"  accumulator to true
V2 ← 0  ;                                      and "∨"  accumulator to false.
while V0 ≠ 0                            Scan the formula again
    (copy: R0←R0+8; V1←VR0                  Get value of variable
    if V0 ∧ 7 = 6 then (V1← ¬ V1; V0←V0↓3);  If "¬" then complement.
    V2 ← V2 ∨ V1;                           "OR" into ∨ accummulator
    if V0∧3= 0 then (V3←V3∧V2; V2←0)        "AND" into ∧ accummulator
    V0←V0↓3);                               Next symbol
if (V0↓R2)↑ R2≠ V0 then accept.         Test for satisfiability
```

Note that "accept" is a vertex, not an edge. It should be clear that the number of steps required to check the formula in V_0 is $O(\ell+n)$, where ℓ is the length of the formula and n is the variable whose name is the largest. If all variables with names less than n appear in the formula, the bound simplifies to $O(\ell)$.

The next algorithm tests networks to ensure that they sort. For our purposes a network of order n is a sequence of pairs (i,j) satisfying $0 \le i < j < n$. Such pairs are called comparators. To apply a network to an array A[0:n-1] means carrying out, for each comparator (i,j) in the sequence in turn, the operation "if A[i] > A[j] then swap (A[i], A[j])". A network sorts when it may be applied to any array A[0:n-1] to yield an ordered array.

It is a well-known theorem that a network sorts any array if and only if it sorts any array containing only 0's and 1's. Thus we want to check the 2^n possible such arrays.

The approach is to simulate the effect of the network, one comparator at a time, on all possible inputs. We represent the outcome of each application of a comparator as a vector of 2^n bits, whose i-th bit (counting the least significant bit as 0, as before) is 1 if and only if, for some initial input A, the binary representation of i can appear after this application as the contents of A[0:n-1] (counting A[0] as the least significant bit of i). The network sorts if and only if, at the end of the sequence, the only positions k containing 1's are of the form $k=2^n-2^j$ for some integer $j\geq 0$, since only such numbers, when written in binary, are of the form $1^{n-j}0^j$, corresponding to a sorted array.

The idea behind the algorithm is that the effect of a comparator (i,j) on a vector representing all possible configurations of A just before application of the comparator is very simple. All bits in positions whose binary representation has its i-th bit 1 and j-th bit 0 are shifted left 2^j-2^i positions, corresponding to interchanging the 1 and the 0. Other bits stay put.

The bits to be moved may be identified with the same patterns we used in the satisfiability checker; let M_k be the vector of 2^n bits formed by repeatedly concatenating $0^{2^k}1^{2^k}$, for $0 \leq i < n$. Then the entire operation described in the preceding paragraph may be carried out on vector V thus:

$$M \leftarrow M_i \wedge \bar{M}_j \qquad \text{(says i is 1 and j is 0)}$$
$$V \leftarrow (V \wedge \bar{M}) \vee ((V \wedge M) \uparrow (2^j-2^i)).$$

At the end of the sequence, the n+1 bit positions of the form 2^n-2^j are set to zero. If V is then zero, the network sorts; otherwise it does not, and we have a record of what non-sorted outputs are possible.

The reader should experience no diffiulty in generating the details of the algorithm, whose basic structure is much the same as for the satisfiability checker.

The constraint i<j on a comparator may be relaxed if arbitrary networks are to be tested. In this case, whenever j<i, the shift $\downarrow(2^i-2^j)$ should be carried out in place of $\uparrow(2^j-2^i)$. One criticism of our model might be that registers should hold integers, not natural numbers.

We turn now to the problem of O(log m) time multiplication of mxm Boolean matrices, a result we use in the main Theorem in Section 4. Define the "and-or" product C of two mxm Boolean matricies A and B by $c_{ij} = \bigvee_{k\geq 0}^{m-1} (a_{ik} \wedge b_{kj})$, $0 \leq i,j \leq m-1$. We show that a deterministic vector machine can compute this product in O(log m)† steps provided that A and B are initially received "stored by columns" (or "stored by rows") in two vector registers. For simplicity assume m is a power of 2; $m = 2^p$.

For the moment, let us assume that A (B) is "stored by rows" in vector register V_0 (V_1) at positions which are multiples of m, other bits of V_0 and V_1 being 0. That is, initially "position†† of a_{ij} (b_{ij}) in V_0 (V_i)" = $m^2i + mj$, $0 \leq i,j \leq m-1$. In general when a format for storing a matrix in a vector register is given, we assume all bits of the register not being used to store matrix elements are 0. Assume integer m is available in some index register.

In the following programs, the job of writing code which initializes index registers will be generally left to the reader. Other useful integers such as m^2 can be computed O(log m) steps by successive doubling.

---

†All logarithms are taken to base 2.
†† Positions are numbered right to left starting with 0.

In order to compute all the $\wedge$-operations of the product in one bit-wise step, we would like to construct strings †

$$w_r = (\text{row } m-1)^m(\text{row } m-2)^m \ldots (\text{row } 1)^m(\text{row } 0)^m$$

and

$$w_c = ((\text{col } m-1)(\text{col } m-2)\ldots(\text{col } 1)(\text{col } 0))^m,$$

where $\text{"row } i\text{"} = a_{i,m-1}a_{i,m-2}\cdots a_{i,1}a_{i,0}$, $0 \le i < m$,

and $\text{"col } j\text{"} = b_{m-1,j}b_{m-2,j}\cdots b_{1,j}b_{0,j}$, $0 \le j < m$.

In order to see how a vector machine can produce w_r within $O(\log m)$ steps, we first view the subproblem of compressing one row of A . Assume initially that "position of a_j in V_0" $= mj$, $0 \le j < m$. We desire a program at whose conclusion we have "position of a_j in V_0" $= j$, $0 \le j < m$, and all other bits of V_0 are 0 . Therefore, a_j should be shifted right $j(m-1)$, $0 \le j < m^{p-1}$ Let $\beta_i(j)$ denote the i^{th} bit of the binary representation of j. Writing $j = \sum_{i=0}^{p-1} \beta_i(j) \cdot 2^i$ shows that it is sufficient for each j to execute (for $i = p-1,p-2,\ldots,2,1,0,$ do (shift a_j right $2^i(m-1)$ iff $\beta_i(j) = 1$)). Of course the program must do this for all j in parallel. Consider the following program for compressing one row.

<u>one row</u>: First construct M to be $1^{(m/2)m}0^{(m/2)m}$. Then

$k_1 \leftarrow (m/2)(m-1)$;

$k_2 \leftarrow (m/4)m$;

While $k_1 \ge m-1$ do <u>rslide</u> (V_0,M).

The macro "<u>rslide</u>(V,M)" is

<u>rslide</u>(V,M): $V \leftarrow (V \wedge \bar{M}) \vee ((V \wedge M) \downarrow k_1)$;

$k_1 \leftarrow k_1/2$;

$M \leftarrow M \downarrow k_2$;

$k_2 \leftarrow k_2/2$.

The dual macro is "<u>lslide</u>", where "$\downarrow k_1$" becomes "$\uparrow k_1$". The effect of these macros is to slide (i.e. shift) by an amount k_1 just those bits of V with matching 1's in the mask M, and to update M, k_1 and k_2.

We now sketch a proof of correctness for <u>one row</u>. Define <u>trunc</u>$(j,k) = \sum_{i=0}^{k} \beta_i(j) \cdot 2^i$. Consider the following induction hypothesis.

For $k = 1,2,3,\cdots,p$, the following conditions hold just before the k^{th} execution of "rslide":

(1). $k_1 = 2^{p-k} \cdot (m-1)$;

(2). $M = 1^{(m/2)m}\ 0^{m \cdot 2^{p-k}}$;

(3). "position of a_j in V_0" $= (m-1) \cdot \text{trunc}(j,p-k) + j, 0 \le j < m$; and hence

(4). all a_j occur at positions congruent to 0 (mod 2^{p-k+1}).

The base $k=1$ is obvious. Assume the hypothesis holds for some fixed k and examine the effect of the k^{th} execution of "rslide". Note first for all j, $0 \le j < m$: (i) $\beta_{p-k}(j) = 1$ implies "position of a_j" $\ge m \cdot \text{trunc}(j,p-k) \ge m \cdot 2^{p-k}$; and (ii) $\beta_{p-k}(j) = 0$ implies "position of a_j" $\le m.\text{trunc}(j,p-k) + j \le m \cdot (2^{p-k}-1) + (m-1) < m \cdot 2^{p-k}$.

Since $M = 1^{(m/2)m}\ 0^{m \cdot 2^{p-k}}$ at this time, "rslide" will mask out precisely those a_j such that $\beta_{p-k}(j) = 1$ and shift these a_j right by the proper amount $k_1 = 2^{p-k} \cdot (m-1)$.

† t^m denotes ttt...t (m times) where t is a string and m is an integer.

Moreover, these a_j are "ored" back into V_0 at positions congruent to $2^{p-k} \pmod{2^{p-k+1}}$. Therefore, by (4) of the induction hypothesis, no shifted a_j is "ored" into a position already occupied by some other a_j which was not shifted by this "rslide". This completes our description of the details required to prove the above inductive statement and thus prove correctness.

w_r is constructed by first compressing all rows in parallel to obtain w_r' = (row m-1) 0^{m^2-m} (row m-2) $0^{m^2-m} \cdots$ (row 1) 0^{m^2-m} (row 0) and then copying this string appropriately to obtain w_r stored in V_0.

construct $w_r(V_0)$: First construct M_1 to be $(1^{(m/2)m}0^{(m/2)m})^m$:

```
M1 ← 1 ;
dup(M1,1,(m/2)m) ;
M1 ← M1 ↑ (m/2)m ;
dup(M1,m^2,m) ;
k1 ← (m/2)(m-1) ;
k2 ← (m/4)m ;
while k1 ≥ m-1 do rslide(V0,M1);
Now V0 contains w'r.
dup (V0,m,m).
```

Given a vector V containing just the word w, with $|w| = \ell$, dup(V,ℓ,n) forms w^n in O(log n) steps, provided n is a power of 2.

```
dup(V,ℓ,n): while n>1 do
                (V ← Vv(V↑ℓ));
                ℓ ← 2ℓ;
                n ← n/2.
```

To construct w_c we first view the subproblem of compressing the entire contents of V_1 to obtain w_c' = (col m-1)(col m-2)...(col 1)(col 0). Recall that "position of b_{ij} in V_1" = m^2i+mj, and note "position of b_{ij} in w_c'" = $mj + i$. Therefore b_{ij} should be shifted right $i(m^2-1)$, $0 \le i,j \le m-1$. A little thought shows that a "scaled up by m" version of the previous compression program constructs w_c' from the contents of V_1.

construct $w_c(V_1)$: First construct M_2 to be $1^{(m/2)m^2}0^{(m/2)m^2}$.

```
M2 ← 1 ;
dup(M2,1,(m/2)m^2) ;
M2 ← M2 ↑ (m/2)m^2 ;
k1 ← (m/2)(m^2-1) ;
k2 ← (m/4)m^2 ;
while k1 ≥ m^2-1 do rslide (V1,M2);
Now V1 contains w'c
dup(V1,m^2,m).
```

Correctness of construct w_c is proven in a way completely analogous to that of one row.

Now C = AB is computed by performing $V_0 \wedge V_1$ (that is $w_r \wedge w_c$) and then fanning-in the contributions to c_{ij} for all i,j. The following procedure accepts matrix inputs and produces the matrix product in the "expanded stored by rows" (position (entry(i,j)) = $m^2i + j$) format.

matrixprod(V_0,V_1): construct $w_r(V_0)$;
construct $w_c(V_1)$;
$V_0 \leftarrow V_0 \wedge V_1$;
Construct M_3 to be $(1^{m/2}0^{m/2})^{m^2}$:
$M_3 \leftarrow 1$;
dup(M_3,1,m/2) ;
$M_3 \leftarrow M_3 \uparrow m/2$;
dup(M_3,m,m^2) ;
$k_1 \leftarrow m/2$; $k_2 \leftarrow k_1$;
while $k_1 \geq 1$ do rslide(V_0,M_3).

V_0 now contains C stored in the proper input format; that is, "position of c_{ij} in V_0" = $m^2 i + j$, and all other bits of V_0 are 0. This is convenient for performing a chain of products (as in the transitive closure algorithm to follow).

Since matrices may be initially available only in some more compressed format, we now give procedures which efficiently (within time O(log m)) translate between a "compressed stored by columns" (position(entry(i,j))= mj + i) format and the proper I/O format for matrixprod. The procedures are given without further comment.

expand(V_0): First construct M_4 to be $(1^{m/2}0^{m/2})^{m^2}$:
$k_1 \leftarrow (m/2)(m^2-1)$;
$k_2 \leftarrow m/4$;
while $k_1 \geq m^2-1$ do lslide(V_0,M_4) .

compress(V_1): This code is the portion of construct w_c preceding the comment.

It is clear that construct w_r , construct w_c , matrixprod , expand and compress all run within time O(log m) . These procedures require space $O(m^3)$.

Remark. These procedures can be easily modified to involve only shifts by powers of 2. For example, "$\downarrow 2^k(m-1)$" can be replaced by "$\uparrow 2^k$" followed by "$\downarrow 2^k m$" . In matrixprod , only a fixed number (six) of vector registers are used.

We have considered the "and-or" product since it is most compatible with the Boolean nature of the model. However, the basic idea carries over to other kinds of operations, such as +/x and min/+ matrix multiplication. If the exterior operation costs c_1 and the interior c_2 , then the cost of a matrix multiplication is

$$c_2 + O(c_1 \log m) .$$

The application for matrix multiplication in the next section involves its use as a component of a procedure which computes the closure of an mxm Boolean matrix A within $O(\log^2 m)$ steps. The transitive closure of A is defined by

$$A^* = I \vee A \vee A^2 \vee A^3 \vee \ldots$$

It is easy to see that also $A^* = (I \vee A)^m$ if A is mxm. Our approach to computing A^* is therefore to square $(I \vee A)$ log m times. The $O(\log^2 m)$ time bound is immediate. The following algorithm gives the details.

closure(V_0): First set all $a_{ii} = 1$. Then:
Construct M_5 to be $(0^m 1)^m$:
$M_5 \leftarrow 1$;
dup(M_5, m^2+1, m) ;
$V_0 \leftarrow V_0 \vee M_5$;
for k_3 from 1 to log m by 1 do
($V_1 \leftarrow V_0$;
matrixprod(V_0, V_1)) .

Remark. Here, closure can be optimized somewhat by constructing masks M_1, M_2, M_3

once and shifting them back to their original positions after each call on matrixprod , although this saves at most a constant factor in the time bound. It follows from a previous remark that closure can be programmed using only shifts by powers of 2 . Six vector registers are sufficient, or four if masks are not saved. We observe that closure consumes time $O(\log^2 m)$ and space $6m^3$, distributed over the six vectors.

4. The Characterization

In the introduction we outlined a fairly tight characterization of time bounded vector machines in terms of space bounded Turing machines when the machines are to be used for set recognition, as distinct from transduction. The purpose is to state and prove (in outline) the characterization and to examine some of its consequences.

Since vector machines initially receive inputs in vector registers, we assume that the sets to be accepted have already been coded into binary; we resolve any possible disputes about how long an input really is by requiring an appropriately chosen end-marker, whether or not the machine reads it, and apply the definition of length of a vector (number of significant digits) to the issue of length of an input.

The definitions of time and space bounded acceptance by vector machines are completely analogous to corresponding definitions for Turing machines and are not repeated here.

Let $\left\{\begin{matrix}\text{VM-NTIME}(T(n)) \\ \text{VM-DTIME}(T(n))\end{matrix}\right\}$ $\left(\left\{\begin{matrix}\text{VM-NSPACE}(S(n)) \\ \text{VM-DSPACE}(S(n))\end{matrix}\right\}\right)$ denote the classes of sets accepted by $\left\{\begin{matrix}\text{non-deterministic} \\ \text{deterministic}\end{matrix}\right\}$ vector machines within time $T(n)$ (within space $S(n)$) where the complexity bounds $T(n)$ and $S(n)$ are given as functions of the input length n . Let TM-NSPACE(S(n)) , etc. denote corresponding classes for multitape Turing machines with separate read-only input. A Turing machine is given input x by writing ¢x¢ initially on the input tape with the input head scanning the leftmost ¢ . Assume all such Turing machines considered have been programmed never to move the input head outside the domain delimited by the endmarkers. The variant with a separate tape is adopted so that it makes sense to consider sets being accepted in less than linear space, for example $\log^2 n$ space for recognition of context-free languages.

We are mainly interested in Turing machines as space bounded acceptors. As such, their power equals that of more general models of serial computation, for example RAM's. That is, there is a natural definition of "space" for RAM's for which TM's and RAM's can simulate one another with at most constant factor space loss.

In order to state Theorem 1 precisely, a notion of "real-time countability" for vector machines is needed. The following suffices.

Definition. A function $F : N \to N$ is VM-countable iff there is a constant $c > 0$ and a deterministic vector machine V_F such that for all n , if V_F is started on any input of length n , it runs for time $\leq c.F(n)$ and halts with $F(n)$ in some designated index register.

We do not attempt a characterization of VM-countable functions, but simply note that $(\lceil \log n \rceil)^k$ for any integer $k \geq 0$ is VM-countable, as are all real-time countable functions.

The characterization is stated as two theorems.

Theorem 1. Let S(n) be VM-countable. Then $\text{TM-NSPACE}(S(n)) \subseteq \bigcup_{c>0} \text{VM-DTIME}(c.(S(n)+\log n)^2)$.

Theorem 2. Let T(n) be tape constructable. Then $\text{VM-NTIME}(T(n)) \subseteq \text{TM-DSPACE}(T(n).(T(n)+\log n))$.

Before proving these theorems, let us examine some consequences of these results. One corollary is that the four complexity concepts {deterministic, non-deterministic} x {time, space} are all polynomially related for vector machines. This is in contrast to the situation for Turing machines where "DTIME vs SPACE" and "NTIME vs DTIME" are open questions.

Corollary 3. Assume $F(n)$ is VM-countable and tape-constructable and $F(n) \geq \log n$.

(1) $\text{VM-NTIME}(F(n)) \subseteq \bigcup_{c>0} \text{VM-DTIME}(c.F^4(n))$.

(2) $\text{VM-NTIME}(F(n)) \subseteq \bigcup_{c>0} \text{VM-DSPACE}(c.F^2(n))$.

(3) $\text{VM-NSPACE}(F(n)) \subseteq \bigcup_{c>0} \text{VM-DSPACE}(cF^2(n))$.

(4) $\text{VM-NSPACE}(F(n)) \subseteq \bigcup_{c>0} \text{VM-DTIME}(cF^2(n))$.

Proof of Corollary 3 is implicit in the diagram of Figure 1 . Arrows "→" should be read as inclusion "⊆" . Circled arrows denote trivial inclusions and involve at most a constant factor increase in the complexity bound. The exponent 4 in (1) can probably be reduced by a direct simulation. If it can be reduced to 2 , we could then say that a deterministic X can simulate a non-deterministic Y for any X,Y ∈ {space bounded TM's} ∪ {time bounded VM's}, with the bound being at most squared.

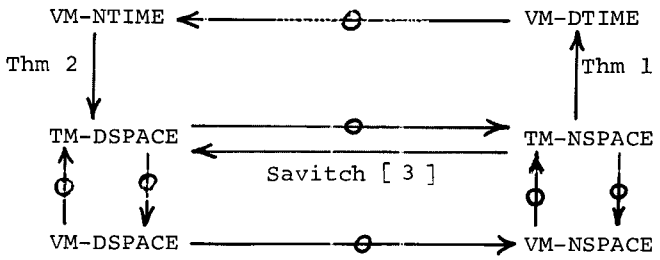

Figure 1

Another corollary follows immediately from Theorem 1, the fact that all context-free languages are in $\text{TM-DSPACE}(\log^2 n)$, and the fact that all context-sensitive languages are in $\text{TM-NSPACE}(n)$. [1]

Corollary 4. If L is a context-free language then $L \in \text{VM-DTIME}(\log^4 n)$. If L is a context-sensitive language then $L \in \text{VM-DTIME}(n^2)$.

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 "DTIME vs SPACE" relation for Turing machines. Of course, there is no reason to suppose that vector machines are the most powerful possible forms of parallel computers, even to within a polynomial.

Corollary 5. Let $\mathcal{F} = \{T(n) \mid T(n) \geq n \text{ and } (\forall k \in N)[\log^k T(n) \text{ is VM-countable and tape-constructable}]\}$. The following statements are equivalent.

(1) There is a k such that, for all $T(n) \in \mathcal{F}$, $\text{TM-DTIME}(T(n)) \subseteq \bigcup_{c>0} \text{VM-DTIME}(c.\log^k T(n))$.

(2) There is a k such that, for all $T(n) \in \mathcal{F}$, $\text{TM-DTIME}(T(n)) \subseteq \bigcup_{c>0} \text{TM-DSPACE}(\log^k T(n))$.

Cook [2] has conjectured that (2) is false, at least in the case where $\mathcal{F}$ is taken to be the class of polynomials. The implication (2) ⊃ (1) is significant if we view vector machines as a reasonable model of unbounded parallel computation. The implication (1) ⊃ (2) is somewhat weak in that vector machines may not be general enough, as we remarked above.

Proof of Theorem 1. A careful proof is unnecessarily tedious. Our purpose is only to outline the general details sufficiently to allow the reader to construct the remainder easily. The basic idea is to simulate a space S(n) bounded Turing machine M by first constructing the one-step transition matrix for all possible instantaneous descriptions (i.d.'s) of M, and then computing the transitive closure of this matrix. Since M can enter at most $nc^{S(n)}$ different i.d.'s for some constant c, the transitive closure computation requires only time $O((S(n) + \log n)^2)$.

Let $L \in \text{TM-NSPACE}(S(n))$. Let M be a non-deterministic TM with one work-tape (in addition to the input tape) which accepts L within space S(n) . (There is an obvious "real-space" simulation of a multi-tape machine by a one-tape machine.) Assume M uses no more than space S(n) on all inputs of length n , for all n .

Say M has states Q , input alphabet $I = \{0,1,¢\}$, and work-tape alphabet Γ .

Our formalization of i.d.'s is as follows. An n-i.d. of M is a string of the form τw where $\tau \in (00)^*.(11).(00)^*$, $|\tau| = 2n+4$, $w \in \Gamma^*.Q.\Gamma^*$, and $|w| = 2S(n)+1$. Suppose M is given input ¢x¢ where $x \in 1.\{0,1\}^*$ and $|x| = n$. Then the n-i.d. $\delta = (00)^{i-1}(11)(00)^{n-i+2}w_1 q w_2$ where $w_1, w_2 \in \Gamma^*$ and $q \in Q$, describes the situation in

which M is in state q, $w_1 w_2$ is written on the worktape, the worktape head is scanning the first symbol of w_2, and the input head is scanning the i-th symbol of ¢x¢ . The initial n-i.d. is $(11)(00)^{n+1}\#^{S(n)}q_0\#^{S(n)}$ where # denotes the blank tape symbol in Γ. Assume M may accept an input of length n only by entering the unique accepting n-i.d. $(11)(00)^{n+1}\#^{S(n)}q_a\#^{S(n)}$ where q_a is an accepting state.

Let Next_M denote the one-step transition predicate on the i.d.'s of M . For any n, if δ,δ' are n-i.d.'s and $x \in 1.\{0,1\}^*$, then $\underline{\text{Next}}_M(x,\delta,\delta')$ iff δ can reach δ' by one step in a computation of M on input ¢x¢ .

We now describe the operation of a deterministic vector machine V_M which simulates M . Since M involves possibly large alphabets I,Q,Γ, we think of vector registers as storing character strings rather than just bit strings. This can be implemented by using a binary block code. Let $\Sigma = I \cup Q \cup \Gamma \cup R$ be all the symbols we shall need, and choose integer b so that $2^b \geq |\Sigma|$. Choose a 1-1 code $h: \Sigma \to \{0,1\}^b$; in particular define $h(\#) = 0^b$. Extend $h: \Sigma^* \to \{0,1\}^*$ in the obvious way.

Suppose V_M receives input x of length n in register V_0 . As a technical convenience, assume V_0 actually contains a slightly coded string P(¢c¢) . Define P by P(0) = 01 , P(1) = 11 , P(¢) = 10, and extend $P : I \to \{0,1\}^*$ in the obvious way. This precoding can be eliminated although the following proof becomes awkward. Because of the precoding we must reconstruct n, the length of the original input x. The reader can easily devise a time O(log n) procedure which constructs a string 10^{n-1} by using the fact that P(¢x¢) is of length 2n+4 .

Since S(n) is VM-countable, O(S(n)) steps can be taken to obtain S(n) in some index register. Let $m' = 2^{b(2S(n)+1)}$, let n' be the least power of 2 such that $n' \geq n+2$, and let $m = n'm'$. These integers n', m', m can now be computed (by successive doubling) in other index registers within O(S(n)+log n) steps.

Next we define bit strings c_j , $0 \leq j < m$, of length m . Some of these strings code n-i.d.'s of M . If $0 \leq i < n'$, $0 \leq z < m'$, and $j = n'z + i$, then $c_j = 0^k(00)^{n'-i-1}(11)(00)^i w$ where $w \in \{0,1\}^*$, $|w| = b(2S(n)+1)$, w is a binary representation (possibly with leading zeroes) of integer z , and integer k is chosen so that $|c_j| = m$.

If $\delta = \tau w$ ($\tau \in (00)^*(11)(00)^*$, $w \in \Gamma^* Q \Gamma^*$) is an n-i.d. of M, then $\tau h(w)$ is a suffix of c_j for some j , $0 \leq j < m$. In this case we say that c_j <u>codes</u> δ .

V_M's eventual goal is to construct an mxm Boolean matrix A such that if c_i codes an n-i.d. δ of M , then $a_{ij}=1$ iff (c_j codes an n-i.d. δ' of M and $\text{Next}_M(x,\delta,\delta')$) . If c_i does not code an n-i.d., the value of a_{ij} is unimportant. Since the next goal is to compute A^*, V_M constructs A in the proper input format for the procedure <u>closure</u> described earlier; that is, "position of a_{ij}" = $m^2 i + mj$.

Thus V_M's first goal is to construct strings $\nu_r = (c_{m-1})^m (c_{m-2})^m \cdots (c_1)^m (c_0)^m$ and $\nu_c = (c_{m-1}c_{m-2}\cdots c_1 c_0)^m$. In outline, the process is:

(1) First the "w" parts of the $\{c_j\}$ are constructed in vector register V in increasing numerical order, the low order bit of each w occurring at a position equal to a multiple of $n'm^2$.

(2) V is then copied appropriately so that w's begin at each multiple of m^u, u=1 or 2.

(3) The "(00)*(11)(00)*" parts are constructed in V_{temp} .

(4) Finally the two parts are combined into V to yield $c_{m-1}0^{m-m^u}c_{m-2}0^{m-m^u}\cdots c_1 0^{m-m^u}c_0$ in register V within O(log m + log m' + log n) steps , assuming m is a power of 2.

To construct ν_r we carry out the above with u=2 , giving us m^2 spacing between the items, and then do <u>copy</u>(V,m,m). For ν_c we do the above with u=1 , giving us no spacing between items, and then do <u>copy</u>(V,m^2,m) . Thus these two vectors can be constructed in

time $O(\log m + \log n) = O(S(n) + \log n)$.

Now V_M must compute the entries of A . First execute:

$$V_0 \leftarrow V_0 \uparrow b(2S(n)+1) \ ;$$
$$\underline{\text{copy}}(V_0, m, m^2) \ .$$

This has the effect of constructing a copy of the input $P(\text{¢}x\text{¢})$ "opposite" the "(00)*(11)(00)*" parts of all c_j in v_r . Say $P(\text{¢}x\text{¢}) = x_0x_1x_2\ldots x_nx_{n+1}$ where $x_i \in \{0,1\}$.

For simplicity we view the problem of computing $a_{i_0j_0}$ for a particular i_0, j_0. It should be clear that the process for one can be done for all in parallel. The necessary tricks are:

(1) Using v_r as a mask to select the appropriate character from the input;

(2) Using copy to "smear" the character over the segment of v_r relevant to a_{ij} ;

(3) Constructing a character that encodes which way the head moved (+1,0,-1,other);

(4) Similarly smearing the head-motion character;

(5) Inspecting locally and in parallel all neighborhoods of the segment to determine whether a legal step has taken place;

(6) Collecting that information into one place to constitute $a_{i_0j_0}$.

Note that (5) can be done by a series of Boolean operations and short shifts that mimic a logic circuit made of 2-input gates whose inputs correspond to such a neighborhood. There is no need to check whether legal steps have occurred in two places since the start configuration for M has only one place where a legal step can be taken, namely where the head is on the work-tape. Of course, the rest of the segment needs to be checked to make sure that the tape does not change unpredictably. All of the operations (1) to (5) can be carried out in $O(\log n + \log S(n))$ steps.

Given that A has been constructed, all that remains is to form the transitive closure of A , and we are done.

Note that V_M uses space $O(n2^{dS(n)})$ for some constant d . V_M accesses a fixed (depending only on M) number of vector and index registers; thus our results hold whether we consider VM's with or without indirect addressing. V_M can be modified to involve only shifts by powers of 2 and still run within time $O((S(n)+\log n)^2)$, which may be of value to potential VM manufacturers, who might appreciate the simplicity resulting from only having to implement shifts by powers of 2 . Even in the algorithms of Section 3 practically all shifts could be carried out as shifts by a power of 2 not introducing more than a constant factor into the time bounds.

Proof of Theorem 2. In space $T^2(n)$ (where $T(n)$ is the time bound for the simulated vector machine V) the simulating Turing machine M_V can store the contents of all index registers accessed by the vector machine V . It can also store a "choice-sequence", which is a list of the decisions made by V (which is non-deterministic), in only space $T(n)$. Thus M_V's outer loop will cycle through all possible choice-sequences, which disposes of the issue of non-determinism for V .

For each choice sequence, M_V attempts to make progress through V's graph (program), consulting the choice sequence where appropriate. It updates and consults index registers in the obvious way. It simply ignores vector register operations until it needs to evaluate a predicate involving vector registers in order to tell whether it may follow the next edge. The strategy is to set up a goal of the form "find(i,j,k)" , where i is the i-th bit of V_j at step k of the computation. Note that a goal can be represented in space $T(n)$. To test, say, $V_3=0$, it suffices to enumerate "find(i,3,t)" for all $i \leq 2^{T(n)}$, since vectors cannot get longer than this, assuming that the test is required at the t-th step. The test succeeds if and only if all values of "find" return 0 (note that this deals automatically with the case when all but finitely many of the bits of V are ones). To compute "find" requires, in essence, backtracking through the computation and setting up further subgoals at times t-1 , t-2,... . Note that at each time a bounded number of subgoals will have to be put on a stack, which introduces a further factor of $T(n)$ into our space requirements, bringing the space bound up to $T^2(n)$. We leave to the reader the task of filling in the details of the recursive subgoal generator.

5. Generalization to Transduction

Theorems 1 and 2 relate the power of Turing machines and Vector machines only for set recognition problems. More generally, both types of machines can carry out transductions. Both theorems generalize (at least in spirit) to transduction problems. Time constraints prohibit our giving proofs in this paper. Proofs will appear elsewhere.

Let $f: 1.\{0,1\}^* \to 1.\{0,1\}^*$ be a transduction we wish to compute. For simplicity, assume f is total. A deterministic vector machine A computes f if A contains a special halting vertex v_h (that is, there are no edges directed out of v_h) and for x in $1\{0,1\}^*$, if A is started with x in register V_0, there is a computation which terminates at vertex v_h and leaves $f(x)$ in register V_1. A Turing machine is given a special output tape on which to write the result of its transduction. This tape is scanned by a left-moving write-only head which may or may not print an output symbol at each step. The notion of transduction by Turing machines is straightforward to formalize, as are the notions of transductions being computed within time $T(n)$ or space $S(n)$ by vector machines or Turing machines (where n is again the length of the input). Attaching the prefix "TR-" to a deterministic complexity class (e.g. TR-VM-DTIME(T(n))) defines a corresponding complexity class of functions.

If the space required to write the output is counted as work space, then Theorem 1 generalizes fairly easily.

Theorem 3. Let $f \in$ TR-TM-DSPACE(S(n)) . Let S'(n) be VM-countable and satisfy $(\forall x)[|f(x)| \leq S'|x|]$ and $(\forall n)[\ S(n) \leq S'(n)\ ]$. Then $f \in$ TR-VM-DTIME$(c.(S'(n)+\log n)^2)$ for some constant c .

However note that it makes sense to consider a function f being computed within space $S(n)$ where $|f(x)|$ grows much faster than $S|x|$. For example, consider the integer multiplication function f_m which produces the binary representation of the product of the first and last halves of its argument, each interpreted as a binary number. It is not hard to see that $f_m \in$ TR-TM-DSPACE(log n). However $|f(x)| = |x|-1$ for many inputs x .

With more work one can show the following.

Theorem 4. Let S(n) be VM-countable.
Then TR-TM-DSPACE(S(n)) $\subseteq \bigcup_{c>0}$ TR-VM-DTIME$(c(S(n)+\log n)^3)$.

The exponent 3 , which is the best we presently know, may be improvable.

The proof of Theorem 2 extends easily to transduction.

Theorem 5. Let T(n) be tape constructable.
Then TR-VM-DTIME(T(n)) $\subseteq$ TR-TM-DSPACE(T(n)(T(n)+log n)) .

Bibliography

[1] Aho, A. , J. Hopcroft and J. Ullman. 1968. Time and tape complexity of pushdown automaton languages. Information and Control 13, 186-206.

[2] Cook, S.A. 1973. An observation on Time-Storage trade-off. Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (SIGACT), 29-37.

[3] Savitch, W.J. 1970. Relationships between nondeterministic and deterministic tape complexities, Journal of Computer and System Sciences 4, 177-192.

[4] Younger, D. 1967. Recognition and parsing of context-free languages in time n^3. Information and Control 10, 2, 189-208.