Reprinted from JOURNAL OF COMPUTER AND SYSTEM SCIENCES
All Rights Reserved by Academic Press, New York and London

Vol. 36, No. 3, June 1988
Printed in Belgium

Parallel Computation with Threshold Functions

IAN PARBERRY AND GEORG SCHNITGER*

Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania 16802

Received August 1, 1986; revised February 20, 1987

We study two classes of unbounded fan-in parallel computation, the standard one, based on unbounded fan-in ANDs and ORs, and a new class based on unbounded fan-in threshold functions. The latter is motivated by a connectionist model of the brain used in artificial intelligence. We are interested in the resources of time and address complexity. Intuitively, the address complexity of a parallel machine is the number of bits needed to describe an individual piece of hardware. We demonstrate that (for WRAMs and uniform unbounded fan-in circuits) parallel time and address complexity is simultaneously equivalent to alternations and time on an alternating Turing machine (the former to within a constant multiple, and the latter a polynomial). In particular, for constant parallel time, the latter equivalence holds to within a constant multiple. Thus, for example, polynomial-processor, constant-time WRAMs recognize exactly the languages in the logarithmic time hierarchy, and polynomial-word-size, constant-time WRAMs recognize exactly the languages in the polynomial time hierarchy. As a corollary, we provide improved simulations of deterministic Turing machines by constant-time shared-memory machines. Furthermore, in the threshold model, the same results hold if we replace the alternating Turing machine with the analogous threshold Turing machine, and replace the resource of alternations with the corresponding resource of thresholds. Threshold parallel computers are much more powerful than the standard models (for example, with only polynomially many processors, they can multiply two integers, compute the parity function and sort in constant time), and appear less amenable to known lower-bound proof techniques.

1. INTRODUCTION

There has recently been a growing interest in the time complexity of *massively parallel* computers, that is, parallel machines with an excessively large number of processors and unbounded fan-in. Two machine models which have become popular are the *MIMD shared-memory machine*, a non-uniform collection of random-access machines which communicate via a shared memory (a MIMD parallel computer may have a different program for each processor, see Flynn [10]), and the *unbounded fan-in circuit*, a nonuniform combinational circuit built from unbounded fan-in AND and OR gates. The important resources for the former are (unit-cost) time and number of processors, and for the later, size (number of wires) and depth (maximum length of any path from an input to an

* Research supported by NSF Grant DCR-84-07256.

0022-0000/88 $3.00

output). There is a well-known result [37] which states that simultaneous size and depth of nonuniform unbounded fan-in circuits is equivalent to simultaneous processors and time on a MIMD shared-memory machine, the former related by a constant multiple, and the latter by a polynomial. Recent research has centered on the number of processors required to obtain constant time on a massively parallel machine. Independently, Furst, Saxe, and Sipser [12] and Ajtai [3] proved that a superpolynomial number of processors is required to compute the parity of n bits in constant time on a nonuniform model. Recently, Andrew Yao [43] has shown that $2^{n^{1-\lambda}}$ processors are necessary, for some real number $\lambda > 0$. We provide a matching upper bound in this paper.

We wish to characterize *uniform* unbounded fan-in parallel computers, such as the WRAM, a SIMD shared-memory machine with simultaneous writes. Limited characterizations have been attempted, but they appear to fail when the number of processors get too large or the parallel running time gets too small. The parallel computation thesis of Goldschlager [13, 14] is an attempt to characterize the power of fast parallel computers. One interpretation states that fast parallel computers recognize exactly the languages in POLYLOGSPACE. For convenience, we will call this the *first parallel computation thesis*. The extended parallel computation thesis of Dymond [8, 9] is an attempt to characterize the power of small, fast parallel computers. One interpretation states that small, fast parallel computers recognize exactly the languages in NC. (NC is the class of languages recognizable in polynomial time and polylog reversals by a k-tape deterministic Turing machine [30]). For convenience, we will call this the *second parallel computation thesis*.

These parallel computation theses appear to have lost popularity due to the increasing interest in massively parallel computers. Blum [4] demonstrated that neither the first nor the second parallel computation thesis holds for shared-memory machines with a large number of processors. It was shown in [25] that the first parallel computation thesis holds for shared-memory machines with word-size bounded by a polynomial in parallel running time and that it appears to fail otherwise; in particular, constant parallel time is possible for any recursive function if word-size (and number of processors) is large enough. One of our aims in this paper is to provide a characterization of unbounded fan-in parallelism which is valid for sub-logarithmic, or even constant parallel time.

We will also find that this characterization extends to a wider class of parallel computation. The standard parallel models described above derive their power from the ability to compute unbounded fan-in Boolean ANDs and ORs. We consider parallel computers based on unbounded fan-in threshold functions, motivated by connectionist models of the brain used in artifical intelligence. Parallel computers based on threshold functions seem immensely powerful (for example, they can compute the parity function in constant time using linear processors). The well-known lower-bound techniques for unbounded fan-in circuits appear to break down completely in the case of threshold circuits.

The remainder of this paper is divided into seven sections. In Section 2 we examine a machine model from artificial intelligence called the *Boltzmann machine*.

TABLE I

A Summary of the Eleven Simulations in This Paper

	Simulated Machine			Simulating Machine		
Result	Machine type	Parallel time	Hardware	Machine type	Parallel time	Hardware
Theorem 2.1	Boltzmann machine	Time $T(n)$	Size $Z(n)$	Threshold circuit	Depth $O(T(n))$	Size $Z(n)^{O(1)}$
Theorem 6.1	Minimal TRAM	Time $T(n)$	Word-size $W(n)$	k-Tape TTM	Thresholds $O(T(n))$	Time $O(T(n)\,W(n))$
Corollary 6.2	Reasonable TRAM	Time $T(n)$	Word-size $W(n)$	k-Tape TTM	Thresholds $O(T(n))$	Time $W(n)^{O(1)}$
Lemma 7.1	k-Tape DTM	—	Time $T(n)$	Minimal WRAM	Time $O(1)$	Word-size $O(T(n))$
Theorem 7.2	k-Tape TTM	Thresholds $H(n)$	Time $T(n)$	Minimal TRAM	Time $O(H(n))$	Word-size $O(T(n)\,H(n))$
Corollary 7.3	k-Tape TTM	Thresholds $H(n)$	Time $T(n)$	Reasonable TRAM	Time $O(H(n)$	Word-size $O(T(n)^2)$
Theorem 8.1	Minimal WRAM	Time $T(n)$	Word-size $W(n)$	k-Tape ATM	Alternations $O(T(n))$	Time $O(T(n)\,W(n))$
Corollary 8.2	Reasonable WRAM	Time $T(n)$	Word-size $W(n)$	k-Tape ATM	Alternations $O(T(n))$	Time $W(n)^{O(1)}$
Theorem 8.3	k-Tape ATM	Alternations $H(n)$	Time $T(n)$	Minimal WRAM	Time $O(H(n))$	Word-size $O(T(n)\,H(n))$
Corollary 8.4	k-Tape DTM	—	Time $T(n)$	Minimal WRAM	Time $O(1)$	Word-size $O(T(n)/\log^* T(n))$
Corollary 8.5	k-Tape ATM	Alternations $H(n)$	Time $T(n)$	Reasonable WRAM	Time $O(H(n))$	Word-size $O(T(n)^2)$

Note. A result numbered "$x \cdot y$" is the yth result in Section x. The terms "minimal" and "reasonable" applied to TRAMs and WRAMs will be defined in Section 2. The table gives details of the simulating machine and simulated machine in each result, and gives for each the resources corresponding to "parallel time" and "hardware" according to the relevant parallel computation thesis.

We define the *unbounded fan-in threshold circuit*, and show that it is equivalent to a deterministic variant of the Boltzmann machine. In Section 3 we define a variant of the popular shared-memory model which we call the TRAM. This is similar to the standard WRAM, except the multiple-write convention makes use of threshold functions. Section 4 contains some prelimilary lemmas on the power of WRAMs. In Section 5 we define the *threshold Turing machine*, a model similar to the standard alternating Turing machine, based on threshold quantifiers instead of universal and existential quantifiers. In Section 6 we simulate a TRAM on a threshold Turing machine, and in Section 7 we simulate a threshold Turing machine on a TRAM. Finally, in Section 8 we gather together the evidence provided by these simulations into two parallel computation theses, one for the standard unbounded fan-in models, and one for the new unbounded fan-in threshold models.

This paper contains eleven simulations of one resource-bounded parallel machine by another. A brief summary appears in Table I. A preliminary version of this paper has appeared in [26].

2. Boltzmann Machines and Threshold Circuits

Connectionist models of the brain have recently regained popularity amongst researchers in artificial intelligence. The connectionist model is a parallel system which has a large number of simple processing elements. Computation is performed by increasing or decreasing the strength of communications between physically connected processors. One such model is the *Boltzmann machine* [1, 18], an undirected graph in which vertices represent processors, and edges links. Each vertex is labelled with a *threshold value*, and each edge with a *weight*, both of which are integers. Each processor can be in one of two states, which we will call *active* and *inactive*. The computation occurs synchronously as follows. At time t, a processor computes the sum of the weights of the edges connecting it to its active neighbors. That processor is active at time $t+1$ with probability depending on the difference between that sum and its threshold (with probability tending to zero below the threshold, exactly one-half at the threshold, and tending to one above it). At the start of the computation a distinguished set of input vertices is held in either the active or inactive state, to represent the input in binary. The output is similarly encoded in the states of a distinguished set of output vertices on completion of the computation. The computation is completed when the energy of the system is at a local minimum.

The three key properties of this model are that it is probabilistic, it computes using threshold functions, and that the termination condition depends on global energy. We wish to focus on the second property and address the question of how parallel machines which compute using threshold functions differ from previously studied models of parallel computers. This may throw some light on the prodigious computing power of the human brain. We isolate this property by making the

model deterministic and simplifying the termination condition. This enables us to avoid certain problems concerned with the determination of the global energy in the more general model [20, 21].

Formally, a *deterministic Boltzmann machine* (hereafter we will drop the adjective "deterministic") is an infinite family of finite machines, $B_1, B_2, \ldots$, one for each input size. Each B_n consists of a directed graph $G_n = (V_n, E_n)$, distinguished sets of *input* and *output* vertices I_n, $O_n \subseteq V_n$, $|I_n| = n$, a *threshold assignment* $h_n: V_n \to \mathbf{Z}$ ($\mathbf{Z}$ denotes the set of integers) and a *weight assignment* $w_n: E_n \to \mathbf{Z}$. Computation on an input $x \in \{0,1\}^n$ proceeds as follows. At time $t=0$, the input processors are placed in states which encode x. All other processors are placed in the inactive state. A processor $v \in V_n$ is active at time $t>0$ iff $\sum_{u \in S_v} w_n(u) \geqslant h_n(v)$, where

$$S_v = \{u \in V_n \mid (u, v) \in E_n \text{ and } u \text{ is active at time } t-1\}.$$

The computation terminates when no further change of state occurs in the system. At that time, the output is encoded in the states of the output vertices. A Boltzmann machine runs in *time* $T(n)$ if the maximum number of steps taken by B_n on an input of size n is bounded above by $T(n)$, and has *size* $Z(n)$ if $|E_n| \leqslant Z(n)$. We assume that G_n is connected, so that the number of edges is a reasonable measure of size. We also assume that the absolute values of the edge weights (and therefore the thresholds) are bounded above by a polynomial in size, and that $T(n) \leqslant Z(n)$.

A number of features of this model are noncritical; for example, the connection graph can be made acyclic, all weights can be made 1, and all threshold values can be made nonnegative. In addition, the threshold functions can be replaced by "exactly equal" functions. Let $B = \{0,1\}$ denote the Boolean set. Define the function $\#_k: B^n \to B$ as follows: $\#_k(x_1, x_2, \ldots, x_n) = 1$ iff exactly k of the x_i's are equal to 1. Define an *unbounded fan-in threshold circuit* to be similar to the standard unbounded fan-in circuit (see, for example, [12, 37]), except for the fact that gates compute #-functions instead of unbounded fan-in AND and OR. Clearly these are equivalent to circuits built from upper-threshold (true if at least k inputs are true) and lower-threshold (true if at most k inputs are true) gates, with at most a polynomial increase in size and a simultaneous increase in depth of at most a constant multiple. We will refer to a parallel machine based on upper- and lower-threshold functions as a *threshold parallel computer*. Where exactness in resources is not crucial, we will choose the convenient normal form based on #-functions. Unbounded fan-in threshold circuits are more powerful than standard unbounded fan-in circuits, for example

$$x_1 \wedge x_2 \wedge \cdots \wedge x_n = \#_n(x_1, \ldots, x_n)$$

$$x_1 \vee x_2 \vee \cdots \vee x_n = \#_1(\#_1(x_1, \ldots, x_n), \#_2(x_1, \ldots, x_n), \ldots, \#_n(x_1, \ldots, x_n))$$

$$x_1 \oplus x_2 \oplus \cdots \oplus x_n = \#_1(\#_1(x_1, \ldots, x_n), \#_3(x_1, \ldots, x_n), \ldots, \#_{n+(n \bmod 2)-1}(x_1, \ldots, x_n))$$

$$\neg x = \#_0(x).$$

THEOREM 2.1. *Size and time on a Boltzmann machine is simultaneously equivalent to size and depth on an unbounded fan-in threshold circuit, size to within a polynomial, and time/depth to within a constant multiple.*

Proof (Sketch). If $w_1, ..., w_n \in \mathbf{Z}$, define the function $\#_k(w_1, ..., w_n)$: $B^n \to B$ as follows: $\#_k(w_1, ..., w_n)(x_1, ..., x_n) = 1$ provided $\sum_{i=1}^n x_i \cdot w_i = k$ (and 0 otherwise). Similarly define $\leqslant_k(w_1, ..., w_n)$ and $\geqslant_k(w_1, ..., w_n)$: $\mathbf{Z}^n \to B$ to be 1 iff $\sum_{i=1}^n x_i \cdot w_i \leqslant k$ and $\sum_{i=1}^n x_i \cdot w_i \geqslant k$, respectively.

CLAIM 1. *Boltzmann machines based on $\#_k$ are at least as powerful as those based on $\geqslant_k$.*

Proof. For all integers k,

$$\geqslant_k(w_1, ..., w_n)(x_1, ..., x_n) = \#_k(w_1, ..., w_n)(x_1, ..., x_n) \vee \#_{k+1}(w_1, ..., w_n)(x_1, ..., x_n)$$
$$\vee \cdots \vee \#_{\sum_{i=1}^n w_i}(w_1, ..., w_n)(x_1, ..., x_n). \blacksquare$$

CLAIM 2. *Boltzmann machines based on $\geqslant_k$ are at least as powerful as those based on $\#_k$.*

Proof. For all integers k,

$$\#_k(w_1, ..., w_n)(x_1, ..., x_n) = \geqslant_k(w_1, ..., w_n)(x_1, ..., x_n) \wedge \leqslant_k(w_1, ..., w_n)(x_1, ..., x_n)$$

and

$$\leqslant_k(w_1, w_2, ..., w_n)(x_1, x_2, ..., x_n) = \geqslant_{-k}(-w_1, -w_2, ..., -w_n)(x_1, x_2, ..., x_n). \blacksquare$$

Thus #-functions provide a convenient normal-form for Boltzmann machines. It remains to show that these normal-form machines can be "unwound" into circuits.

CLAIM 3. *All edge-weights can be made positive.*

Proof. Consider a vertex computing $\#_k(w_1, w_2, ..., w_n)(x_1, ..., x_2, ..., x_n)$. Without loss of generality assume that $w_1, ..., w_t > 0$ and $w_{t+1}, ..., w_n < 0$. Let $S = \sum_{i=1}^t w_i$. First construct $S - k + 1$ vertices computing

$$\#_r(w_1, w_2, ..., w_t)(x_1, x_2, ..., x_t) \qquad \text{for} \quad k \leqslant r \leqslant S$$

and $S - k$ vertices computing

$$\#_s(-w_{t+1}, -w_{t+2}, ..., -w_n)(x_{t+1}, x_{t+2}, ..., x_n) \qquad \text{for} \quad 1 \leqslant s \leqslant S - k.$$

Finally, AND together all pairs of $\#_r$ and $\#_s$ pairs with $r - s = k$, and OR together these ANDs. $\blacksquare$

If we allow the connection graph to have multiple edges, we have

CLAIM 4. *All edge-weights can be made* 1.

Proof. Replace each edge of weight w from vertex u to vertex v by w edges of weight 1. ∎

CLAIM 5. *The graph can be "unwound" into a circuit.*

Proof. Uses standard techniques similar to those used in [16, 33]. Multiple edges in the graph can easily be handled using additional fan-out. ∎

Each of these transformations increases the size by a polynomial and the depth by a constant multiple. ∎

It is also interesting to observe that combinatorial circuits built from "majority" gates are also equivalent to Boltzmann machines. Let $\text{half}(x_1, ..., x_n) = \#_{\lfloor n/2 \rfloor}(x_1, ..., x_n)$, for $n \geqslant 1$. Then if $k < \lfloor n/2 \rfloor$, $\#_k(x_1, ..., x_n) = \text{half}(x_1, ..., x_n, y_1, ..., y_{n-2k})$, where $y_i = 1$ for $1 \leqslant i \leqslant n-2k$. If $k > \lfloor n/2 \rfloor$, $\#_k(x_1, ..., x_n) = \text{half}(x_1, ..., x_n, y_1, ..., y_{2k-n})$, where $y_i = 0$ for $1 \leqslant i \leqslant 2k-n$. The number of inputs to each gate can also me made a power of 2. Suppose n is even, and $m = 2^{\lceil \log n \rceil}$. Then $\text{half}(x_1, ..., x_n) = \text{half}(x_1, ..., x_n, y_1, ..., y_{m-n})$, $y_{2i-1} = 0$, $y_{2i} = 1$ for $1 \leqslant y \leqslant (m-n)/2$. If n is odd, $\text{half}(x_1, ..., x_n) = \text{half}(x_1, ..., x_n, 1)$. The constants 0 and 1 are readily available, since $\text{half}(x, x) = 0$ for all $x \in B$, and $\text{half}(0) = 1$.

Even polynomial-size uniform threshold circuits appear to be extremely powerful. For example, we have seen that they can compute the parity of n bits in constant time. Chandra, Stockmeyer, and Vishkin [6] observe that they can also sort n polynomial-bit integers, add n polynomial-bit integers, and multiply two n-bit integers in constant time. They can also multiply two $n \times n$ matrices of polynomial-bit integers in constant time. We have chosen to remove the probabilism from uniform Boltzmann machines. It is not known whether this reduces their power. However, it can be shown using a result of Adleman [2] that probabilism can be removed from nonuniform Boltzmann machines without increasing their size by more than a polynomial, and their running time by more than a constant multiple [27].

We have chosen to model the brain as an unbounded fan-in computer. It may be argued that this is an unreasonable low-level model, since the brain appears to have around 10^{10} neurons and degree around 10^3. However, at a higher level, computations in the brain appear to involve *concepts*, which can be modelled using large groups of geographically diverse but strongly connected neurons [15]. These concepts behave much like the processors of the Boltzmann machine (if sufficiently many neurons in the concept become activated, then the whole concept becomes activated) and may be better modelled using unbounded fan-in.

3. THE SHARED-MEMORY MODEL

The shared-memory machine is a popular parallel model used by complexity theorists. Informally, it consists of a large number of powerful processors which

communicate via a shared memory. Each processor possesses an infinite number of general purpose registers $r_0, r_1, r_2, \ldots$, each of which can hold a single integer, and a unique read-only processor identity register PID which is preset to i in the ith processor, $i \in N$ (N denotes the set of nonnegative integers). The shared memory consists of an infinite number of cells $s_0, s_1, s_2, \ldots$, each of which is also capable of holding a single integer. A *program* for this machine consists of a finite list of instructions; each instruction is either a *local instruction* (an internal computation or transfer of control), or a *communication instruction* (a read from, or a write into, shared memory). The local instruction-set has the following form, where "$\circ$" denotes a binary operation defined on integers:

$r_i \leftarrow$ constant	(load register with constant)
$r_i \leftarrow r_j \circ r_k$	(binary operation)
$r_i \leftarrow r_{r_j}$	(indirect load)
$r_{r_i} \leftarrow r_j$	(indirect store)
$r_i \leftarrow$ PID	(store PID)
halt	(end execution)
goto m **if** $r_i \geqslant 0$	(conditional transfer of control).

Communication instructions have the form:

$r_i \leftarrow s_{r_j}$	(read)
$s_{r_i} \leftarrow r_j$	(write).

So far, we have left the binary operation "$\circ$" unspecified. In particular, we will be interested in two types of instruction-sets. The *minimal* instruction-set allows integer addition and subtraction, and logical shifts:

$r_i \leftarrow r_j \pm r_k$	(addition)
$r_i \leftarrow \lfloor r_j * 2^{r_k} \rfloor$	(logical shift).

Note that in the second instruction, r_k may be either positive (corresponding to a left-shift) or negative (corresponding to a right-shift). The *general* instruction-set includes any instruction which can be simulated by a k-tape deterministic Turing machine in polynomial time. (That is, a k-tape deterministic Turing machine can, when given as input the m-bit binary representations of the operands, compute the binary representation of the result in time $m^{O(1)}$.)

More formally, each machine is specified by a program M and a processor bound $P(n)$. A computation proceeds roughly as follows. Suppose $x \in \mathbf{Z}^n$, $x = \langle x_1, x_2, \ldots, x_n \rangle$. Each x_i is called an *input symbol*. The symbol x_i is placed into

shared-memory location i, for $1 \leqslant i \leqslant n$. All other memory locations and general purpose registers are set to zero. Processors $0, 1, ..., P(n)-1$ are activated simultaneously; they synchronously execute the program M. When all processors have halted, the output is to be found in some specified place in the shared memory. In particular, single-integer outputs are found in shared memory location s_0. A shared-memory machine acts as an acceptor if the inputs are restricted to a finite alphabet (encoded as integers in the obvious fashion), and at the end of a computation, shared memory location s_0 contains either 1 or 0, indicating acceptance or rejection of the input, respectively. We will consider several different protocols for dealing with simultaneous memory access, starting with the most straightforward:

1. The PRAM model. Simultaneous memory access is forbidden. In the following models, simultaneous access to individual shared-memory locations is allowed. Any number of processors may simultaneously read from any shared-memory location. We will be interested primarily in two different conventions for dealing with simultaneous writes.

2. The WRAM model. If several processors are attempting to write into a single shared-memory cell, then the smallest-numbered processor succeeds. All other data is lost.

3. The TRAM model. Suppose several processors are attempting to write into a shared-memory location which currently contains the value k. If exactly k of them are attempting to write a nonzero value, then the smallest-numbered processor succeeds. Otherwise the shared-memory location takes on the value zero.

The *processor* bound $P(n)$ is a measure of the number of processors used as a function of input size. The machine is said to have *word-size* $W(n)$ if the maximum value in any register or shared memory location during any computation on an input of size n has absolute value less than $2^{W(n)}$. Note that this includes the inputs, outputs and processor identity registers, so in particular $W(n) = \Omega(\log P(n))$. We will concentrate on parallel machines with $W(n) = \Theta(\log P(n))$, which preludes parallel models such as the Vector Machine [17, 31], whose word-size is much larger than is needed to address its processors. We also assume that $W(n) = \Omega(\log n)$. This is a reasonable assumption, since at least $\lceil \log(n+1) \rceil$ bits are required to address the n shared-memory locations containing the inputs. The *time* bound $T(n)$ is the number of instructions executed before all processors have halted, again as a function of input size. We will call a shared-memory machine with the general instruction-set *reasonable* provided $T(n) = W(n)^{O(1)}$; that is, the running time is exponentially smaller than the number of processors.

Similar parallel machine models have appeared in a large number of papers, the earliest of which include Fortune and Wyllie [11], Goldschlager [13], Schwartz [34], and Shiloach and Vishkin [35]. Our model is not strictly SIMD (see Flynn [10] for nomenclature), since different processors can be at different points in their program at any given time. However, it is easy to show [23] that is equivalent to a

strictly SIMD one. Note that our machines are slightly nonuniform in the sense that information (depending on the input-size) may be encoded in the number of processors $P(n)$. Our simulations will make heavy use of this property.

4. Some Preliminary Results Concerning Shared Memory Machines

WRAMs are powerful parallel machines. In particular, they can compute various useful arithmetic functions in constant time. The results in this section will prove useful during the simulation of threshold Turing machines by TRAMs in Section 7. The reader who wishes to process this paper in a top-down fashion may skip the material contained in this section during the first reading.

Lemma 4.1. *A PRAM with $\lceil \log n \rceil$ processors can compute $\lceil \log n \rceil$ or $\lfloor \log n \rfloor$ in constant time with word-size $O(\log n)$, when given as input a single positive integer n.*

Proof. Suppose $n \geqslant 1$; $\lfloor \log n \rfloor$ is computed as follows. Processor i, $i \geqslant 0$, computes the value $v = \lfloor n/2^i \rfloor$ using a single shift instruction. If it finds that $v > 0$ and $\lfloor v/2 \rfloor = 0$, then $i = \lfloor \log n \rfloor$. Computation of $\lceil \log n \rceil$ from $\lfloor \log n \rfloor$ is simple; if $n = 2^i$ then $\lceil \log n \rceil = i$, otherwise $\lceil \log n \rceil = i + 1$. The required value can then be written into shared-memory cell s_0 by processor i. ∎

Lemma 4.2. *A WRAM with word-size $O(n(b + \log n))$ can add n b-bit numbers in constant time.*

Proof. We use the techniques of [23–25]. Suppose that we are given n b-bit integers $x_1, \ldots, x_n$. Let us assume, for the purposes of this proof, that they are all positive. The modifications necessary for the inclusion of negative integers are simple but tedious. We can assume without loss of generality that n is a power of 2.

First, it is easy to determine n, the number of inputs. We can adopt the convention that zero is never used for an input value (if necessary we can encode nonnegative integers by adding one to them). Processor i reads shared memory cells i and $i + 1$. If the latter contains zero while the former contains a nonzero value, then $i = n$. Processor i can then write its PID into shared-memory location 0, where it can be read simultaneously by all processors. Note that if n is not a power of 2, then we can compute $\lceil \log n \rceil$ using Lemma 4.1, and then using a single processor compute $2^{\lceil \log n \rceil}$, the power of 2 immediately above n, using a shift operation. This will have a negligible effect on our resource bounds.

The sum of n b-bit numbers can have no more than $b + \log n$ bits. Each processor requires this value. The value $\log n$ can be computed using Lemma 4.1; it remains to determine the value of b. We do this by finding the largest input integer. We use the first n^2 processors, divided into n equal-sized teams. Each processor can determine whether it participates in this subcomputation by comparing its PID to n^2. Since n is a power of 2 and both n and $\log n$ are known to all processors, n^2 can be computed with a single shift operation. Next, each processor determines which team

it is in, and its identity number within its team, as follows. Each processor extracts the last $\log n$ bits of its PID using two shifts and a subtraction. It treats this value as its identity number within its team. It treats the remaining leading bits of its PID as its team identity number. That is, it has divided its PID in constant time into two values (i, j), where $0 \leqslant i, j < n$, and acts as the jth member of the ith team.

The ith team uses shared memory location $n+i+1$ for communication (remembering that the cells 1 through n contain the input). The 0th processor in team i (which we will call the *leader* of that team) sets shared-memory $n+i+1$ to zero. The ith team $0 \leqslant i < n$ determines whether the $(i+1)$th input integer x_{i+1} is the largest overall. The jth processor in the ith team compares x_{j+1} with x_{i+1}, for $0 \leqslant i, j < n$. If the former is greater than the latter, then it writes a one into shared-memory location $n+i+1$. When this is finished, the team-leader reads shared-memory location $n+i+1$. If that location contains zero, it knows that x_{i+1} is the largest input. It can then write this value into shared-memory location 0, where it can be read by all processors. This has taken constant time, and can be performed provided the word-size is at least $2 \log n$. The value of b can finally be obtained by finding the logarithm of this largest input value using Lemma 4.1.

Now the value $b + \log n$ is known to all processors. Let us assume for the purposes of this proof that it is a power of 2. If not, then it can easily be rounded up to a power of 2 by again using Lemma 4.1 and a shift operation. The number of bits in this value is also easily obtained using Lemma 4.1. The processors now divide themselves into $2^{O(n(b+\log n))}$ teams of n processors. Each team interprets its team identification number (which has $O(n(b \log n))$ bits) as a sequence of n $(b+\log n)$-bit integers. The ith member of each team, $1 \leqslant i < n$ extracts the ith and $(i+1)$th integer in this sequence, while the 0th processor extracts the first integer. The ith processor will have to do a shift of $i(b+\log n)$ bits. In order to do this, it will have to first compute the shift amount. Since the latter factor is a power of 2, the multiplication can be implemented using a shift operation. The ith processor of each team, $1 \leqslant i \leqslant n$ verifies that this $(i+1)$th integer is equal to the ith plus x_{i+1}, while the 0th processor verifies that the first integer equals x_1. Those processors which find a discrepancy report to their team leaders via the shared-memory as described in the previous paragraph. Exactly one team will find no discrepancy. Its team leader knows that its team identity number represents a valid prefix-sum string for the given inputs. It then extracts the total sum (the last integer in the sequence) which it finally writes into shared-memory location 0 for output.

All of the operations described take place in constant time. The PIDs of the processors have $O(n(b+\log n))$ bits, and are the largest words used in the computation. ∎

LEMMA 4.3. *A WRAM with word-size $O(b^2)$ can multiply two b-bit positive integers in constant time.*

Proof. We will use the standard shift-and-add algorithm. For simplicity, let us assume that the two input integers x_1 and x_2 are both positive. The machine first

finds the value of b by taking the logarithm of the largest input (using Lemma 4.1). Processor i, $0 \leqslant i < b$, does the following.

(a) Extract the ith bit of x_2 (where the bits are numbered from left-to-right starting with 0) using shifts and a subtraction.

(b) If the value obtained in (b) is nonzero, then left-shift x_1 by i places (that is, multiply it by 2^i), and write the result into shared-memory location $i+1$.

The sum of these values is computed in constant time using Lemma 4.2. ∎

LEMMA 4.4. *A WRAM, when given as input a single integer m, can compute $\lceil m^{1/c} \rceil$ in constant time and word-size $O(\log^2 m)$, for any natural number $c > 1$.*

Proof. Use p teams of processors, where $p \geqslant \lceil m^{1/c} \rceil$. Team i, $0 \leqslant i < p$, checks to see whether $i = \lceil m^{1/c} \rceil$. It does this by computing i^c (using $c-1$ multiplications). For this purpose, each team has $2^{O(\log^2 m)}$ processors (by Lemma 4.3). If $i^c \geqslant m$, yet $i^{c-1} < m$, then $i = \lceil m^{1/c} \rceil$. ∎

For our purposes, it will be sufficient to show that a WRAM with linear word-size can add n constant-bit integers in constant time. However, a much stronger result is possible without much extra effort.

LEMMA 4.5. *For every $0 < \lambda \leqslant \frac{1}{2}$ there exists $\mu > 0$ such that a WRAM with word-size $O(n^{1-\mu})$ can add n $n^{1-\lambda}$-bit integers in constant time.*

Proof. Suppose we have as input n positive integers, each of $n^{1-1/c}$ bits, for some positive integer $c \geqslant 2$. Since the technique used is elementary, for a cleaner presentation of this proof we will omit the floor and ceiling operators necessary to ensure that all values are integers. The sum of these integers (and every partial sum) has at most $O(n^{1-1/c})$ bits.

The WRAM first determines n, and computes $m = n^{1/(c+1)}$. After this pre-computation, the summation is performed in two phases.

Phase 1. Divide the input into n/m groups of m numbers, and sum each group. After c iterations of this process, we are left with n/m^c partial sums.

Phase 2. Add the n/m^c partial sums.

By Lemma 4.4, the pre-computation takes constant time and negligible word-size (for large enough n). By Lemma 4.2, Phases 1 and 2 can be performed in constant time. The word-size required for the former is proportional to $mn^{1-1/c} = n^{1-1/(c^2+c)}$ and, for the latter, is proportional to

$$\frac{n}{m^c} n^{1-1/c} = n^{1-1/(c^2+c)}.$$

Thus $n\, n^{1-1/c}$-bit integers can be summed in constant time with word-size $O(n^{1-1/(c^2+c)})$. ∎

Suppose $x_i \in N$, $1 \leqslant x_i \leqslant n$ for $1 \leqslant i \leqslant n$. Define prev: $Z^n \to Z^n$ by $\text{prev}(x_1, ..., x_n) = \langle y_1, ..., y_n \rangle$, where $y_i = j$ if $x_j = x_i$, $j < i$, and $x_k \neq x_i$ for $j < k < i$ (and 0 if no such j exists). Define last: $Z^n \to Z^n$ by $\text{last}(x_1, ..., x_n) = \langle y_1, ..., y_n \rangle$, where $y_i = j$ if $x_j = i$ and $x_k \neq i$ for $j < k \leqslant n$ (and 0 if no such j exists).

LEMMA 4.6. *A WRAM can compute* $\text{prev}(x_1, ..., x_n)$ *and* $\text{last}(x_1, ..., x_n)$ *in constant time with word-size* $O(\log n)$.

Proof. We will prove the result for prev (the algorithm used for last is similar). The values $y_1, ..., y_n$ described above are computed as follows. Divide the processors into n^2 teams, one for each ordered pair $\langle i, j \rangle$, $1 \leqslant i, j \leqslant n$. Each team has n processors, one for each k, $1 \leqslant k \leqslant n$. The kth processor of each team, $j < k < i$, remains active; the rest do not participate in the following. We reserve a shared-memory location for each team, initialized to zero. The kth processor of each team, $j < k < i$, verifies that $x_k \neq x_i$. If it finds that $x_k = x_i$, it writes a one into the shared-memory location reserved for its team. All team members do this simultaneously. The lowest-numbered member of its team then reads that value, verifies that it is still zero, and checks that $x_j = x_i$. If so, then it writes the value j to the ith shared memory cell, for output. ∎

5. THRESHOLD TURING MACHINES

It is possible to define threshold quantifiers based on the threshold functions introduced in Section 2. Let Σ be a finite alphabet, and F: $\Sigma^* \to B$. Then $(\#_k^n w: F(w)) \in B$, denoting the predicate "exactly k strings of size n satisfy F," is defined as follows: $(\#_k^n w: F(w)) = 1$ if $|\{w \in \Sigma^n \mid F(w)\}| = k$ (and 0 otherwise). Threshold quantifiers are at least as powerful as existential and universal quantifiers, since $\#_{2^n}^n w: F(w)$ is true iff $F(w)$ holds for all $w \in \Sigma^n$, and $\#_1((\#_1^n w: F(w)), (\#_2^n w: F(w)), ..., (\#_{2^n}^n w: F(w)))$ is true iff $F(w)$ holds for some $w \in \Sigma^n$. We will write $(\#_k w\ F(w))$ for $(\#_k^n w: F(w))$, where the domain of the quantification is obvious from context.

A k-tape *threshold Turing machine* (abbreviated TTM) is similar to the popular alternating Turing machine (abbreviated ATM) [5, 7, 9, 32]. It has k read/write *work-tapes*, a finite-state control, and random-access to its input via a write-only *index-tape*. The latter device is necessary if we are to discuss sublinear running-time and will be familiar to those acquainted with alternating Turing machines. It also has a read-only *guess-tape* and a write-only *threshold-tape*. All tapes are infinite in one direction and have cell numbered 1, 2, ..., each of which can hold a single symbol. Each tape has a single head, which scans a single tape cell. More formally, a k-tape TTM is a 9-tuple $(Q, \Gamma, \Sigma, \delta, q_0, q_a, q_r, q_t)$, where:

Σ is a finite *input alphabet*. Without loss of generality, we will take $\Sigma = \{0, 1\}$.

Γ is a finite *tape alphabet*, $\Sigma \subset \Gamma$. Without loss of generality, we will take $\Gamma = \{0, 1, b\}$, where b is the distinguished blank symbol.

Q is a finite set of states, including four distinct distinguished states, as follows: q_0 is the *initial state*, q_a the *accept state*, q_r the *reject state*, and q_t the *threshold state*.

δ is the transition function. If $\Delta = \{\text{left, stay, right}\}$ is the set of directions in which a tape-head may move, then δ: $\Sigma \times (Q - \{q_a, q_r, q_t\}) \times \Gamma^{k+1} \rightarrow Q \times (\Gamma \times \Delta)^k \times (\Sigma \times \Delta)^2 \times \Delta$.

We define a *configuration* of a TTM in the normal way, to be a snapshot of the machine at some instant in time. A configuration with state q_t is called a *branching* configuration, all others are called *nonbranching*. A configuration with state q_a or q_r is called a *halting* configuration. A TTM is started with all heads in the first cell of their respective tapes, and all tape cells blank except for the first cell on the threshold and index tapes, which both contain the symbol "1." The finite-state control is in state q_0. This is called the *initial configuration*. Consider an arbitrary configuration of a TTM. The action of the machine on input $x_1, x_2, ..., x_n$ is similar to that of a deterministic Turing machine, except where the threshold state is concerned. The contents of the index tape are interpreted as the binary representation of some nonnegative integer i (with its least-significant bit in the first tape-cell).

(i) Suppose the finite-state control is in state $q \in Q - \{q_a, q_r, q_t\}$, symbol $s_0 \in \Gamma$ is under the guess-head, symbols $s_1, s_2, ..., s_k$ are under the k work-tape heads, and

$$\delta(x_i, q, s_0, s_1, ..., s_k) = (r, (t_1, d_1), ..., (t_k, d_k), (t_{k+1}, d_{k+1}), (t_{k+2}, d_{k+2}), d_{k+3}).$$

Then t_j is written in the cell under the head on the jth work-tape and the head is moved one cell in direction d_j, $1 \leqslant j \leqslant k$, t_{k+1} is written in the cell under the head on the index tape and the index-head is moved one cell in direction d_{k+1}, t_{k+2} is written in the cell under the head on the threshold tape and the threshold-head is moved one cell in direction d_{k+2}, the guess-head is moved one cell in direction d_{k+3}, and the finite-state control moves into state r. The new configuration thus obtained is called the *successor* of the original. A configuration is called *accepting* if its successor is accepting. The *time requirement* of the original configuration is defined to be one plus the time requirement of its successor. The *threshold requirement* of the original configuration is defined to be equal to the threshold requirement of its successor.

(ii) If the finite-state control is in state $q \in \{q_a, q_r\}$ then the TTM halts. If $q = q_a$ then the configuration is called *accepting*. The *time requirement* and *threshold requirement* is defined to be zero in both cases.

(iii) If the finite-state control is in state q_t, then the contents of the threshold tape are interpreted as the binary encoding of a nonnegative integer m. Suppose the guess-head is on cell g of the guess-tape. The TTM is restarted in state q_0, with its work-tape and index-tape unaltered, the threshold-tape returned to its initial contents, a random string of symbols from Σ written on cells one through g of the guess-tape (the remaining cells left blank), and the guess-head returned to the first

cell. Each of these 2^g possible new configurations are called *successors* of the original configuration. The configuration is said to be *accepting* if exactly m of its successors are accepting. The *time requirement* of the original configuration is defined to be one plus the longest time requirement of its successors. The *threshold requirement* is defined to be one plus the largest threshold requirement of its successors.

A TTM is said to *accept* its input if its initial configuration is accepting. The language recognized by a TTM is the set of accepted strings over alphabet Σ. A TTM is said to run in *time* $T(n)$ if, for all input strings of length n, the initial configuration has time requirement bounded above by $T(n)$. It is said to use *thresholds* $H(n)$ if, for all input strings of length n, the initial configuration has threshold requirement bounded above by $H(n)$.

Since existential and universal quantifiers are a special case of the threshold quantifier (see the identities given in the first paragraph of this section), it is clear that the standard alternating Turing machine is a special case of the threshold Turing machine (provided we restrict the former to machines with constructible time-bounds). Furthermore, time on this limited threshold Turing machine is related by a polynomial to time on an alternating Turing machine, and the number of thresholds is related by a constant multiple to the number of alternations, both relations holding simultaneously. Therefore, without loss of generality, when we refer to an "alternating Turing machine" we will henceforth mean the special case of a limited threshold Turing machine.

Suppose, for convenience, we reduce threshold Turing machines using upper-threshold and lower-threshold functions instead of #-functions. As noted in Section 2, this affects the time by a polynomial, and the number of thresholds by at most a constant multiple. Threshold Turing machines can then be used to define a *polynomial-time threshold hierarchy*, analogous to the Meyer–Stockmeyer polynomial-time hierarchy [38]. Let TP be the class of languages recognizable in polynomial time by a TTM using a single threshold, either a $\leqslant$-threshold or a $\geqslant$-threshold (note that the class remains the same in either case). Then define $\Theta_0^p = P$, and for $k \geqslant 0$, $\Theta_{k+1}^p = \mathrm{TP}^{\Theta_k^p}$. By induction on k, Θ_k contains the class of languages recognizable in polynomial time by a TTM in k thresholds. It can also be proved using standard techniques that any language in Θ_k can be recognized in polynomial time by a TTM in $2k$ thresholds. However, it is not apparent that the $2k$ can be reduced to k (as in the case of the polynomial-time hierarchy and alternating Turing machines, see Wrathall [42]) since it appears impossible to combine two consecutive polynomial-bounded threshold quantifiers of the same type. The polynomial-time threshold hierarchy clearly includes the polynomial-time hierarchy; for $k \geqslant 0$, $\sum_k^p$, $\Pi_k^p \subseteq \Theta_k^p$. Note also that Θ_2^p contains the language class corresponding to Valiant's #P [39, 40], since it is possible to verify the number of solutions to a polynomial-time verifiable predicate using two queries to an oracle language in Θ_1^p. The polynomial-time threshold hierarchy has been studied in detail by Wagner [41] under the name of "the counting polynomial-time hierarchy." The polynomial-time threshold hierarchy is contained in PSPACE.

6. Simulation of TRAMs by Threshold Turing Machines

In this section we consider the simulation of a $T(n)$ time-bounded, $W(n)$ word-size bounded TRAM on a threshold Turing machine. We say that $W(n) = \Omega(\log n)$ is *constructible* if a k-tape deterministic Turing machine can, when given the binary representation of n, compute the binary representation of $W(n)$ in time $O(W(n))$. Most useful functions are constructible, for example, $W(n) = \log^{O(1)} n$, $W(n) = n^{O(1)}$.

THEOREM 6.1. *Suppose* $W(n)$ *is constructible. A threshold Turing machine can simulate a* $T(n)$ *time-bounded,* $W(n)$ *word-size TRAM with the minimal instruction-set using* $O(T(n))$ *thresholds and time* $O(T(n) \cdot W(n))$.

Proof (Sketch). Let M be a TRAM which runs in time $T(n)$ and uses word-size $W(n)$. Let $x_1, x_2, ..., x_n$ be an input to M. For the purposes of this proof, we will assume that each $x_i \in B$. In general, each x_i will be an integer of at most $W(n)$ bits. In this case, the input to the TTM will be a binary encoding of this sequence. We will demonstrate a threshold Turing machine which accepts this input string iff M does. On input $x_1, x_2, ..., x_n$, the TTM first computes $w = W(n)$. During the simulation of M, the TTM represents individual registers, memory locations, and time-counts with a sequence of $O(w)$ contiguous work-tape cells. The program of M is stored in the finite-state control. We will write $I[l]$ for the lth instruction of this program, $l \geqslant 1$.

Each of the following mutually recursive Boolean procedures returns the value of the quantified Boolean formula given as its statement part. Quantified variables range over all possible values of length w.

function result(p, t, v) {processor p computes value v at time t}
$\exists l(pc(p, t, l) \wedge$
(**case** $I[l]$ **of**
"$r_i \leftarrow$ constant": const $= v$
"$r_i \leftarrow r_j \circ r_k$": $\exists v_1\, \exists v_2$ (local(j, p, $t-1$, v_1) $\wedge$ local(k, p, $t-1$, v_2) $\wedge$
$(v = v_1 \circ v_2)$)
"$r_i \leftarrow r_{r_j}$": $\exists v_1$(local(j, p, $t-1$, v_1) $\wedge$ local(v_1, p, $t-1$, v))
"$r_{r_i} \leftarrow r_j$": local(j, p, $t-1$, v)
"$r_i \leftarrow$ PID": $v = p$
"$r_i \leftarrow s_{r_j}$": $\exists v_1$(local(j, p, $t-1$, v_1) $\wedge$ shared(v_1, p, $t-1$, v))
"$s_{r_i} \leftarrow r_j$": local(j, p, $t-1$, v)
))

function target(p, t, h) {processor p changes register r_h at time t}
$\exists l$(pc(p, t, l) $\wedge$ ($I[l]$ is of the form "$r_h \leftarrow \cdots$", or "$r_{r_i} \leftarrow r_j$" with local (i, p, $t-1$, h)))

function write (p, t, h) {processor p writes into shared-memory location s_i at time t}
$\exists l$(pc(p, t, l) $\wedge$ ($I[l]$ is "$s_{r_i} \leftarrow r_j$") $\wedge$ local(i, p, $t-1$, h))

function pc(p, t, l) {program-counter of processor p at time t is l}
$(t=0 \wedge l=1) \vee (\text{pc}(p, t-1, 0) \wedge l=0) \vee$
$\exists k(\text{pc}(p, t-1, k) \wedge$
(case $I[k]$ **of**
"**goto** m **if** $r_i \geqslant 0$": $\exists v \geqslant 0\ \text{local}(i, p, t-1, v) \wedge (m=l)$
"halt": $l=0$
others: $l=k+1$
$))$

function local(i, p, t, v) {register r_i of processor p gets value v at time t}
$(t=0 \wedge v=0) \vee (\text{target}(p, t, i) \wedge \text{result}(p, t, v)) \vee (\neg\, \text{target}(p, t, i) \wedge \text{local}(i, p, t-1, v))$

function shared(i, t, v)
{shared-memory cell s_i gets value v at time t}
$(t=0 \wedge 1 \leqslant i \leqslant n \wedge v=x_i) \vee$
$(t=0 \wedge i>n \wedge v=0) \vee$
$(\exists k(\text{shared}(i, t-1, k) \wedge (\#_k p\ \text{write}(i, p, t)) \wedge$
$(\exists p(\text{write}(i, p, t) \wedge \text{result}(p, t, v) \wedge \neg \exists q < p\ \text{write}(i, q, t))))) \vee$
$((\neg\, \exists p\ \text{write}(i, p, t)) \wedge \text{shared}(i, t-1, v))$

The Boolean operations $\wedge$ and $\vee$ are computed by branching universally and existentially, respectively. Negations can be computed directly or pushed back to the final states, much in the same manner as alternating Turing machines [5]. Quantifiers are computed by guessing quantified values. The simulation of individual instructions of M is carried out deterministically. The TTM simulates M by computing $\exists t(\forall p(\text{pc}(p, t, 0)) \wedge \text{shared}(0, t, 1))$.

We claim that any call to procedures result(p, t, v), target(p, t, h), write(i, p, t), pc(p, t, l), local(i, p, t, v), or shared(i, t, v) requires at most $O(t)$ thresholds. Let $r(t)$, $t(t)$, $w(t)$, $p(t)$, $l(t)$, and $s(t)$ denote the number of thresholds required by these procedures, respectively. Then

$$r(0)=t(0)=w(0)=p(0)=l(0)=s(0)=0$$

and

$$r(t) \leqslant \max(p(t)+2,\ l(t-1)+6,\ s(t-1)+6)$$
$$t(t) \leqslant \max(p(t)+2,\ l(t-1)+4)$$
$$w(t) \leqslant \max(p(t)+2,\ t(t)+2)$$
$$p(t) \leqslant \max(p(t-1)+3,\ l(t-1)+7)$$
$$l(t) \leqslant \max(t(t)+3,\ r(t)+2,\ l(t-1)+2)$$
$$s(t) \leqslant \max(w(t)+6,\ r(t)+5,\ s(t-1)+3).$$

Thus, in particular, $s(t) \leqslant 28t+21$, and so the simulation of a $T(n)$ time-bounded TRAM uses $O(T(n))$ thresholds. If M has the minimal instruction-set, then the

computation between each threshold takes time $O(W(n))$ (since it only involves guessing register contents, and simulating local instructions of M). This gives the required result. ∎

Note that if the TRAM has the general instruction-set, then the running-time of the TTM increases by only a polynomial.

COROLLARY 6.2. *If* $W(n)$ *is constructible, a threshold Turing machine can simulate a reasonable* $T(n)$ *time-bounded,* $W(n)$ *word-size TRAM using* $O(T(n))$ *thresholds and time* $W(n)^{O(1)}$.

7. SIMULATION OF THRESHOLD TURING MACHINES BY TRAMS

Before we address the problem of simulating a TTM on a TRAM we first consider a much simpler problem, that of simulating a standard k-tape deterministic Turing machine (DTM) on a WRAM.

LEMMA 7.1. *A WRAM with the minimal instruction-set can simulate a* $T(n)$ *time-bounded* k*-tape DTM in constant time and word-size* $O(T(n))$.

Proof (Sketch). Suppose we number the rules of the DTM in some reasonable fashion. We divide the processors into $2^{O(T(n))}$ teams, one for each sequence of $T(n)$ rules $r_0, r_1, ..., r_{T(n)-1}$. Each team has $2^{O(T(n))}$ processors. Each processor can determine which team it is in and its identity number within that team, in constant time as follows.

First, the WRAM computes n as in the proof of Lemma 4.2. Second, the machine determines the number of active processors as follows. Processor i writes a one into shared-memory location $n+i+1$. The number of processors can then be computed using the technique used to determine n (see the proof of Lemma 4.2). Since the number of processors is $2^{cT(n)}$, with c a small constant dependent on the constants in the Turing machine, the value of $T(n)$ can be found efficiently using Lemma 4.1. This value, along with the number of bits in $T(n)$ (found by using Lemma 4.1 again), can be made available to all processors through the shared memory.

Now that the value of $T(n)$ is known to all processors, each can extract the first $T(n)+\log T(n)$ bits of its PID, which it treats as its identity number within its team. The remaining $O(T(n))$ bits are treated as the identity number of the team. These values can be extracted in constant time using shifts and subtractions, as in the proof of Lemma 4.2.

Once each team has determined its team identity number, it interprets that identity number as a sequence of $T(n)$ rules. The ith processor of that team extracts the ith rule r_i, $0 \leqslant i < T(n)$, and determines for each of the tapes the head direction associated with that rule; $+1$ for a rightward move, -1 for a leftward move, and 0 for no move at all. The head position at each point in time is easily determined for each tape by computing a prefix-sum of these values within the team. Each prefix-

sum can be computed using $T(n)$ separate additions performed in parallel using Lemma 4.5, using $T(n)2^{T(n)}$ processors in each team. (This is the reason for requiring $T(n)+\log T(n)$ bits for the identity number of each processor within its team.) Each team uses a separate part of the shared-memory for communication during this computation; the relevant addresses are found by multiplying the team identity number by the appropriate value. The latter can be taken to be a power of 2, so that the multiplication can be performed using a shift operation.

It then verifies that:

1. The sequence of states determined by the sequence of rules is a valid one. That is, rule r_0 requires that the DTM be in its initial state, and for $1 \leqslant i < T(n)$ if rule r_{i-1} leaves the DTM in state q, then rule r_i requires that the DTM be in state q.

2. For each of the k tapes, and for each time t, $0 \leqslant t < T(n)$:

(i) If this is the first time that the head visits this cell, then the symbol read by rule r_t is the symbol found in that cell in the initial configuration.

(ii) If the last time the head visited this cell was at time $s < t$, then the symbol written by rule r_s is the symbol read by rule r_t.

The information necessary for this verification is provided by using the algorithm of Lemma 4.6. Exactly one team will find that its sequence of rules is valid. It can then determine if the final state is accepting and set the contents of shared memory location 0 to 0 or 1, accordingly. By Lemmas 4.5 and 4.6 the simulation requires constant time and word-size $O(T(n))$. ∎

This result can obviously be extended to the simulation of deterministic Turing machines which compute results, rather than acts as acceptors for a language (the final configuration can be constructed from the valid sequence of rule numbers by use of the algorithm for function last in Lemma 4.6), and to the simulation of nondeterministic Turing machines. Furthermore, it can also be used to simulate threshold Turing machines on TRAMs.

THEOREM 7.2. *Suppose* $T(n)$ *is constructible. A TRAM with the minimal instruction-set can simulate a* $T(n)$ *time-bounded,* $H(n)$ *threshold-bounded k-tape TTM in time* $O(H(n))$ *and word-size* $O(T(n) \cdot H(n))$.

Proof (Sketch). The TRAM first evaluates $T(n)$ in constant time and word-size $O(T(n))$ (by use of Lemma 7.1) and constructs a look-up table showing, for every nonbranching configuration, the configuration which follows by the rules of δ in t steps of the TTM, $1 \leqslant t \leqslant T(n)$ (with the convention that once the TTM enters a halting or branching configuration, then it remains there). The table can be constructed in constant time by utilizing the technique used in the proof of Lemma 7.1. A slight modification is necessary to determine the input pointer. The position of the head on the index-tape can easily be computed from the rule sequence in the same manner that the work-tape head positions are computed in Lemma 7.1. The

input pointers can be computed in constant time using word-size $T(n)^{1-\lambda}$ for any real number $\lambda > 0$, by using a technique similar to Lemma 4.5.

The simulation then proceeds as follows. The TTM is simulated up to the point when a branching or halting configuration is entered for the first time (using the look-up table and Lemma 4.6). Call this new configuration C. If the accept state has been entered, then the computation is accepting, and this can be reported in the appropriate way (similarly if the reject state has been entered). Otherwise C is a branching configuration. The processors divide themselves into as many as $2^{O(T(n))}$ teams, one for each possible guess of the appropriate size (gleaned from the position of the guess-tape head in C), each of which continues the computation from the new configuration. When the teams have (recursively) completed their simulation, they report back by having teams which find that their configurations are accepting attempt to write a 1 into a reserved shared-memory location, which has been preset to the integer whose binary representation was found on the threshold tape in configuration C.

Since the computation between branchings takes constant time, the entire simulation takes time $O(H(n))$. The word-size is dominated by $O(T(n) \cdot H(n))$ for the recursive branching. ∎

COROLLARY 7.3. *A reasonable TRAM can simulate a* $T(n)$ *time-bounded,* $H(n)$ *threshold-bounded k-tape TTM in time* $O(H(n))$ *and word-size* $O(T(n)^2)$.

Proof. The TRAM of Theorem 7.2 is reasonable since the number of thresholds that a TTM can perform is bounded above by its running time. ∎

Thus time and word-size on a reasonable TRAM are simultaneously equivalent to thresholds and time on a threshold Turing machine. The first equivalence holds to within a constant multiple, and the second to within a polynomial.

8. Two Parallel Computation Theses for Unbounded Fan-in Parallelism

Since WRAMs are a weaker form of TRAM it is not necessary to use the full power of threshold Turing machines in order to simulate them efficiently.

THEOREM 8.1. *Suppose* $W(n)$ *is constructible. An alternating Turing machine can simulate a* $T(n)$ *time-bounded,* $W(n)$ *word-size* WRAM *with the minimal instruction-set using* $O(T(n))$ *alternations and time* $O(T)(n) \cdot W(n))$.

Proof. Similar to the proof of Theorem 6.1, replacing function shared by:

function shared (i, t, v)
{shared-memory cell s_i contains v at time t}
$(t = 0 \wedge 1 \leqslant i \leqslant n \wedge v = x_i) \vee$
$(t = 0 \wedge i > n \wedge v = 0) \vee$
$(\exists p(\text{write}(i, p, t) \wedge \text{result}(p, t, v) \wedge \neg\, \exists q < p\ \text{write}(i, q, t))) \vee$
$((\neg\, \exists p\ \text{write}(p, t)) \wedge \text{shared}(i, t-1, v))$. ∎

COROLLARY 8.2. *Suppose* $W(n)$ *is constructible. An alternating Turing machine can simulate a reasonable* $T(n)$ *time-bounded,* $W(n)$ *word-size WRAM using* $O(T(n))$ *alternations and time* $W(n)^{O(1)}$.

Similarly, it is not necessary to use the full power of TRAMs to simulate ATM's efficiently.

THEOREM 8.3. *A WRAM with the minimal instruction-set can simulate a* $T(n)$ *time-bounded,* $H(n)$ *alternation-bounded k-tape ATM in time* $O(H(n))$ *and word-size* $O(T(n) \cdot H(n))$.

It was shown in [24] that a WRAM with minimal instruction-set can simulate a $T(n)$ time-bounded nondeterministic k-tape Turing machine in constant time with word-size $T(n)^{1+\varepsilon}$, for any real number $\varepsilon > 0$. Theorem 8.3 improves the word-size to $O(T(n))$, and extends the result to alternating Turing machines with bounded alternations. A further improvement can be shown for the simulation of k-tape deterministic Turing machines.

COROLLARY 8.4. *Suppose* $T(n) = \Omega(n \cdot \log^* n)$ *is time-constructible. A WRAM with the minimal instruction-set can simulate a* $T(n)$ *time-bounded k-tape deterministic Turing machine in constant time using word-size* $O(T(n)/\log^* T(n))$.

Proof. By Theorem 8.3 above, and Theorem 3.3 of Paul *et al.* [29]. ∎

An improvement of the word-size to $O(\sqrt{T(n)})$ for the simulation of single-tape deterministic Turing machines can be made using the weaker result of Maass [22]. Since on an alternating Turing machine, alternations are bounded above by time, we have

COROLLARY 8.5. *A reasonable WRAM can simulate a* $T(n)$ *time-bounded,* $H(n)$ *alternation-bounded k-tape ATM in time* $O(H(n))$ *and word-size* $O(T(n)^2)$.

Proof. By Theorem 8.3. ∎

Thus time and word-size on a reasonable WRAM are simultaneously equivalent to alternations and time on an alternating Turing machine. The first equivalence holds to within a constant multiple, and the second to within a polynomial. This provides evidence for the *third parallel computation thesis*: "In a parallel machine with unbounded fan-in communication, time and address complexity are simultaneously equivalent to alternations and time on an alternating Turing machine, the former to within a constant, and the latter a polynomial." Our results also show that, for constant parallel time, address complexity is equivalent to *within a constant multiple* to time on a constant-alternation ATM. Thus, for example, constant-time massively parallel computers recognize exactly the languages in the polynomial-time hierarchy, and constant-time polynomial-size parallel computers recognize exactly the languages in Sipser's logarithmic-time hierarchy [36].

The third parallel computation thesis sheds some light on a dilemma raised by

Cook [7]: It is popular to take alternating time as a measure of "parallel time" since alternating time is polynomially related to sequential space. This "space is parallel time" characterization was proposed independently by Chandra *et al.* [5] and Goldschlager [13, 14] (the latter calling it the parallel computation thesis). Unfortunately, as Cook points out, the alternating Turing machine has no resource corresponding to "hardware." This led Dymond to his formulation of the extended parallel computation thesis [8, 9], based on the seminal work by Pippenger [30]. Our results suggest that the reason why the alternating Turing machine appeared to have no resource corresponding to "hardware" was that the wrong resource had been chosen for "parallel time." Instead, *number of alternations* corresponds to "parallel time," and alternating time is related to, not hardware, but "address complexity"; that is, the number of bits necessary to describe an individual unit of hardware.

Corollary 6.2 and Corollary 7.3 provide evidence for the *fourth parallel computation thesis*, which seeks to characterize parallel computation based on threshold functions. "In a parallel machine with unbounded fan-in communication using threshold functions, time and address complexity are simultaneously equivalent to thresholds and time on a threshold Turing machine, the former to within a constant, and the latter a polynomial." Further evidence for the third and fourth parallel computation theses is provided by considering uniform unbounded fan-in circuits. The *connection language* of an unbounded fan-in threshold circuit is the set of 4-tuples $\langle g_1, g_2, i, k, n \rangle$ such that in the finite circuit with n inputs the ith input of gate g_1 is connected to the output of gate g_2, and g_1 is a $\#_k$-gate. An unbounded fan-in threshold circuit is said to be *uniform* if its connection language can be recognized by a polynomial-time k-tape deterministic Turing machine (note that the running-time of the Turing machine is thus polynomial in the address complexity of the circuit). Clearly the depth and address complexity of such a circuit is simultaneously equivalent to time and word-size on a TRAM, the former to within a constant multiple and the latter a polynomial (the techniques of [37] extend equally well to the uniform case, with processors that can shift, and to threshold computations). The corresponding result holds for conventional unbounded fan-in circuits and WRAMs.

9. Conclusion

We have provided evidence for two parallel computation theses for unbounded fan-in parallelism. It appears that, for the standard unbounded fan-in models, parallel time and address complexity are simultaneously equivalent to alternations and time on an alternating Turing machine (the former to within a constant, and the latter a polynomial). A similar result holds for the new class of threshold computations, provided the alternating Turing machine is replaced by a threshold Turing machine, and the resource of alternations is replaced by the analogous resource of thresholds.

Many interesting open problems remain. The standard lower-bound proof techniques for unbounded fan-in circuits appear to break down completely in the case of threshold circuits. Is there a problem in LOGSPACE which requires super-polynomial size to solve in constant time? Can a depth hierarchy be shown for polynomial-size machines, analogous to the result for standard unbounded fan-in circuits [36]? The exact number of processors needed to simulate deterministic Turing machines on a WRAM in constant time remains unresolved. This question is related to the question of sequential space versus time. Since a $W(n)$ word-size, $T(n)$ time-bounded WRAM with the minimal instruction-set can be simulated by a deterministic Turing machine in space $O(T(n) \cdot W(n))$ [14], improved upper-bounds on the word-size required to simulate a $T(n)$ time-bounded deterministic Turing machine in constant time on a WRAM may improve the results of Paterson [28] and Hopcroft *et al.* [19].

Finally, we have simplified the Boltzmann machine by removing probabilism and simplifying the termination condition. The computing power of the more general machine remains an open problem.

Acknowledgments

The authors are grateful to Piotr Berman and Nick Pippenger who showed us how to construct the constant-depth threshold circuit for integer multiplication mentioned in Section 2, to Klaus Wagner for correspondence concerning the polynomial-time threshold hierarchy, and to Richard Ladner for suggestions on improving the readability of this paper.

References

1. D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, A learning algorithm for Boltzmann machines, *Cognit. Sci.* **9** (1985), 147–169.
2. L. Adleman, Two theorems on random polynomial time, *in* "Proceedings, 19th Ann. IEEE Symp. on Foundations of Computer Science," 1978, pp. 75–83.
3. M. Ajtai, Σ_1^1-formulae on finite structures, *Ann. Pure Appl. Logic* **24** (1983), 1–48.
4. N. Blum, A note on the "parallel computation thesis," *Inform. Process. Lett.* **17** (1983), 203–205.
5. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, Alternation, *J. Assoc. Comput. Mach.* **28**, No. 1 (1981), 114–133.
6. A. K. Chandra, L. J. Stockmeyer, and U. Vishkin, Constant depth reducibility, *SIAM J. Comput.* **13**, No. 2 (1984), 423–422.
7. S. A. Cook, Towards a complexity theory of synchronous parallel computation, *L'Enseign. Math.* **30** (1980).
8. P. W. Dymond, "Simultaneous Resource Bounds and Parallel Computations," Ph. D. thesis, Technical Report TR145/80, Dept. of Computer Science, Univ. of Toronto, Aug. 1980.
9. P. W. Dymond and S. A. Cook, Hardware complexity and parallel computation, *in* "Proceedings, 21st Annu. IEEE Symp. on Foundations of Computer Science, Oct. 1980," pp. 360–372.
10. M. Flynn, Very high-speed computing systems, *Proc. IEEE* **54** (1966), 1901-1909.
11. S. Fortune and J. Wyllie, Parallelism in random access machines, *in* "Proceedings, 10th Annu. ACM Symp. on Theory of Computing, 1978," pp. 114–118.

12. M. FURST, J. B. SAXE, AND M. SIPSER, Parity, circuits and the polynomial time hierarchy, *Math. Systems Theory* **17**, No. 1 (1984), 13–27.
13. L. M. GOLDSCHLAGER, "Synchronous Parallel Computation," Ph. D. thesis, Technical Report TR-114, Dept. of Computer Science, Univ. of Toronto, Dec. 1977.
14. L. M. GOLDSCHLAGER, A universal interconnection pattern for parallel computers, *J. Assoc. Comput. Mach.* **29**, No. 4 (1982), 1073–1086.
15. L. M. GOLDSCHLAGER, "A Computational Theory of Higher Brain Function," Technical Report, Stanford Univ., Apr. 1984.
16. L. M. GOLDSCHLAGER AND I. PARBERRY, On the construction of parallel computers from various bases of Boolean functions, *Theoret. Comput. Sci.* **43**, No. 1 (1986), 43–48.
17. J. HARTMANIS AND J. SIMON, On the power of multiplication in random access machines, *in* "Proceedings, 15th Annu. IEEE Symp. on Switching and Automata Theory, 1974," pp. 13–23.
18. G. E. HINTON, T. J. SEJNOWSKI, AND D. H. ACKLEY, "Boltzmann Machines: Constraint Satisfaction Networks That Learn," Technical Report CMU-CS-84-119, Dept. of Computer Science, Carnegie-Mellon Univ., May 1984.
19. J. HOPCROFT, W. PAUL, AND L. VALIANT, On time versus space, *J. Assoc. Comput. Mach.* **24**, No. 2 (1977), 332–337.
20. J. J. HOPFIELD, Neural networks and physical systems with emergent collective computational abilities, *Proc. Nat. Acad. Sci.* **79** (1982), 2554–2558.
21. M. LUBY, A simple parallel algorithm for the maximal independent set problem, *in* "Proceedings, 17th Annu. ACM Symp. on Theory of Computing, Providence, R.I., May 1985," pp. 1–10.
22. W. MAASS, personal communication, 1985.
23. I. PARBERRY, "A Complexity Theory of Parallel Computation," Ph. D. thesis, Dept. of Computer Science, Univ. of Warwick, May 1984.
24. I. PARBERRY, "On the Number of Processors Required to Simulate Turing Machines in Constant Parallel Time," Technical Report CS-85-17, Dept. of Computer Science, Penn. State Univ., Aug. 1985.
25. I. PARBERRY, Parallel speedup of sequential machines: A defense of the parallel computation thesis, *SIGACT News* **18**, No. 1 (1986), 54–67.
26. I. PARBERRY AND G. SCHNITGER, Parallel computation with threshold functions (preliminary version), *in* "Proceedings, Structure in Complexity Theory Conference, Berkeley, California, June 1986," Lecture Notes in Computer Science Vol. 223, pp. 272–290, Springer-Verlag, New York/Berlin, 1986.
27. I. PARBERRY AND G. SCHNITGER, "Relating Boltzmann Machines to Conventional Models of Computation," Technical Report CS-87-07, Dept. of Computer Science, Penn. State Univ., Mar. 1987.
28. M. S. PATERSON, Tape bounds for time-bounded Turing machines, *J. Comput. System Sci.* **6**, No. 2 (1972).
29. W. J. PAUL, N. PIPPENGER, E. SZEMERÉDI, AND W. T. TROTTER, On determinism versus nondeterminism and related problems, *in* "Proceedings, 24th Annu. IEEE Symp. on Foundations of Computer Science, Tucson, Arizona, Nov. 1983," pp. 429–438.
30. N. PIPPENGER, On simultaneous resource bounds, *in* "Proceedings, 20th Annu. IEEE Symp. on Foundations of Computer Science, Oct. 1979," pp. 307–311.
31. V. PRATT AND L. J. STOCKMEYER, "A characterization of the power of vector machines," *J. Comput. System Sci.* **12** (1976), 198–221.
32. W. L. RUZZO, On uniform circuit complexity, *J. Comput. System Sci.* **22**, No. 3 (1981), 365–383.
33. J. E. SAVAGE, Computational work and time on finite machines, *J. Assoc. Comput. Mach.* **19**, No. 4 (1972), 660–674.
34. J. T. SCHWARTZ, Ultracomputers, *ACM TOPLAS* **2**, No. 4 (1980), 484–521.
35. Y. SHILOACH AND U. VISHKIN, Finding the maximum, sorting and merging in a parallel computation model, *J. Algorithms* **2** (1981), 88–102.

36. M. SIPSER, Borel sets and circuit complexity, *in* "Proceedings, 15th Annu. ACM Symp. on Theory of Computing, Boston, Mass., Apr. 1983," pp. 61–69.
37. L. STOCKMEYER AND U. VISHKIN, Simulation of parallel random access machines by circuits, *SIAM J. Comput.* **13**, No. 2 (1984), 409–422.
38. L. J. STOCKMEYER, The polynomial time hierarchy, *Theoret. Comput. Sci.* **3** (1977), 1–22.
39. L. G. VALIANT, The Complexity of enumeration and reliability problems, *SIAM J. Comput.* **8**, No. 3 (1979), 410–421.
40. L. G. VALIANT, The complexity of computing the permanent, *Theoret. Comput. Sci.* **8** (1979), 189–201.
41. K. W. WAGNER, The complexity of combinatorial problems with succinct input representation, *Acta Inform.* **23**, No. 3 (1986), 325–356.
42. C. WRATHALL, Complete sets and the polynomial-time hierarchy, *Theoret. Comput. Sci.* **3** (1976), 23–33.
43. A. C. YAO, Separating the polynomial-time hierarchy by oracles, *in* "Proceedings, 26th Annu. IEEE Symp. on Foundations of Computer Science, Portland, Oregon, Oct. 1985."

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium