September 1993

UILU-ENG-93-2239
ACT-129

Applied Computation Theory

PARALLELISM ALWAYS HELPS

Louis Mak

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-93-2239 ACT-129	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Washington, DC 20050
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Washington, DC	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Parallelism Always Helps

12. PERSONAL AUTHOR(S)
Mak, Louis

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) October 1993	15. PAGE COUNT 39

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
random access machine, parallel random access machine, parallelization, simulation, computational complexity, time complexity

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

It is shown that every unit-cost random access machine (RAM) that runs in time T can be simulated by a concurrent-read exclusive-write parallel random access machine (CREW PRAM) in time $O(T^{1/2}logT)$. The proof is constructive; thus, it gives a mechanical way to translate any sequential algorithm designed to run on a unit-cost RAM into a parallel algorithm that runs on a CREW PRAM and obtain a nearly quadratic speedup. One implication is that there does not exist any recursive function that is "inherently not parallesizable."

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Parallelism Always Helps

Louis Mak*
Department of Computer Science and Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801
email: mak@grinch.csl.uiuc.edu

September 23, 1993

Abstract

It is shown that every unit-cost random access machine (RAM) that runs in time T can be simulated by a concurrent-read exclusive-write parallel random access machine (CREW PRAM) in time $O(T^{1/2} \log T)$. The proof is constructive; thus, it gives a mechanical way to translate any sequential algorithm designed to run on a unit-cost RAM into a parallel algorithm that runs on a CREW PRAM and obtain a nearly quadratic speedup. One implication is that there does not exist any recursive function that is "inherently not parallelizable."

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation — *bounded-action devices (e.g., Turing machines, random access machines); relations among models;* F.1.2 [**Computation by Abstract Devices**]: Modes of Computation — *parallelism and concurrency; relations among modes;* F.1.3 [**Computation by Abstract Devices**]: Complexity Classes — *relations among complexity measures*

General Terms: Theory

Additional Key Words and Phrases: Parallel random access machines, parallelization, complexity theory

---

*Supported by the National Science Foundation under Grant CCR-8922008. Author's address: Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 West Main St., Urbana, IL 61801.

1 Introduction

1.1 Motivation

For some problems, the direct parallelization of a sequential algorithm gives a faster parallel algorithm. An example is matrix multiplication. The brute-force sequential algorithm for matrix multiplication runs in $O(n^3)$ time for $n \times n$ matrices. It is straightforward to parallelize this sequential algorithm to get an $O(\log n)$ time parallel algorithm using $O(n^3/\log n)$ processors. On the other hand, some problems are very difficult to parallelize. For example, depth-first search does not seem to admit itself to parallelization [6]. In this paper, we address the following question: Are all sequential algorithms parallelizable?

Cook and Reckhow [2] defined the unit-cost random access machine (RAM). Fortune and Wyllie [5] introduced the parallel random access machine (PRAM). These two models are respectively the most commonly used machine models for analyzing sequential and parallel algorithms. Thus, the above question can be rephrased as follows: Given any unit-cost RAM R that runs in time T, is it always possible to construct a PRAM that simulates R in time $T' = o(T)$? We answer this question affirmatively by exhibiting such a construction with $T' = O(T^{1/2} \log T)$. Several variants of the PRAM have appeared in the literature since it was first introduced. The original model of Fortune and Wyllie has become known as the concurrent-read exclusive-write (CREW) PRAM, which is the model we used in our construction.

Parberry and Schnitger [14] considered the WRAM, a powerful variant of the PRAM. The WRAM differs from the CREW PRAM in three respects:

1. The WRAM is a concurrent-read concurrent-write (CRCW) priority PRAM [4].

2. The WRAM has a richer instruction set for arithmetic operations. The CREW PRAM supports

only addition and subtraction, whereas the WRAM also allows unit-time unrestricted right shifts and modulus operations.

3. The WRAM and the CREW PRAM differ in the manner in which the processors are activated. In the WRAM, an arbitrary number of processors are self-activated at the beginning of the computation. In the CREW PRAM, only one processor is active initially. An active processor activate an idle processor explicitly by executing a FORK instruction. Consequently, in t steps, a PRAM can activate at most 2^t processors.

Parberry and Schnitger showed that every Turing machine that runs in time T can be simulated in constant time by a WRAM with $2^{O(T)}$ processors. The best known simulation of unit-cost RAM's by Turing machines incurs a cubic overhead in the running time [2]. It follows that every unit-cost RAM with time complexity T can be simulated in constant time by a WRAM with $2^{O(T^3)}$ processors. It is desirable to reduce this huge number of processors used in the simulation for two reasons. The first reason is, obviously, to reduce the hardware requirement.

The second and more important reason is that the ability of the WRAM to use an arbitrary number of processors renders this model unreasonably powerful. The above result of Parberry and Schnitger essentially says that every decidable problem can be decided in constant time by a WRAM. This anomaly arises mainly from allowing self-activated processors. The *parallel computation thesis* [7, 13] asserts that the class of languages accepted by any reasonable parallel machine model in polynomial time is equivalent to PSPACE, where PSPACE, as usual, denotes the class of languages that can be accepted by deterministic Turing machines in polynomial space. The WRAM violates the parallel computation thesis and is considered unreasonably powerful [12]. In contrast, the PRAM is considered reasonable because it obeys the parallel computation thesis [5]. So the challenge is to speed up a unit-cost RAM by a PRAM with a reasonable number of processors; the number

of processors should be small enough so that all processors can be activated explicitly within the simulation time.

1.2 Comparison with Previous Results

Dymond and Tompa [3] showed that every deterministic Turing machine running in time T can be simulated by a CREW PRAM in time $O(T^{1/2})$. However, the random access memory of the PRAM is much more flexible than the linear tapes of the Turing machine, which forbid random access into individual tape cells. It was unclear whether it is the parallelism, or the more flexible storage structure, or the combination of both that realizes such a quadratic speedup. Our result demonstrates that parallelism alone suffices to achieve an almost quadratic speedup.

To the best of our knowledge, in all previous speedup results [3, 9, 12, 14, 15, 18, 20], the machine being simulated is limited to the Turing machine. All these results depend on the fact that the changes in the configuration of a Turing machine in t steps are localized to the $2t-1$ cells around each tape head. In contrast, the random access memory of a unit-cost RAM allows the RAM to change the contents of registers with widely different addresses in consecutive steps. The versatility of the random access memory of a unit-cost RAM has defied all prior attempts to speed up a unit-cost RAM by a PRAM. This paper presents the first speedup theorem of unit-cost RAM's by PRAM's.

Reif [16] demonstrated that every probabilistic unit-cost RAM that runs in time T can be simulated by a probabilistic CREW PRAM in time $t(T,L) = O((T \log T \log(LT))^{1/2})$, where L is the largest integer manipulated by the probabilistic RAM during its computation. It is straightforward to modify Reif's proof to show that every unit-cost RAM running in time T can be simulated by a CREW PRAM in time $t(T,L)$. With unit-time addition, however, a RAM can generate integers as

large as $2^{O(T)}$ in time T. Reif's result does not guarantee a speedup, since $t(L,T) = O(T(\log T)^{1/2})$ when $L = 2^{O(T)}$. Our result gives a definite speedup of unit-cost RAM's by PRAM's, regardless of the value of L. It is routine to generalize our proof to establish a speedup theorem of probabilistic unit-cost RAM's by probabilistic CREW PRAM's. This paper subsumes the above result of Reif. Thus, all algorithms (deterministic and probabilistic) are parallelizable.

In summary, all previous simulation results suffer from one or more of the following drawbacks:

1. No definite speedup is guaranteed (Reif [16]).

2. The machine being sped up is limited to the Turing machine [3, 9, 12, 14, 15, 18, 20].

3. The speedup result fails to isolate the effect of parallelism; that is, apart from the parallelism, the simulator enjoys some additional advantage over the machine being simulated — for example, a more flexible storage structure (Dymond and Tompa [3]).

4. The simulator is too strong to be called reasonable, because it violates the parallel computation thesis (Parberry and Schnitger [14]).

Our result does not suffer from any of the above drawbacks.

The rest of this paper is organized as follows. Section 2 defines the RAM and the PRAM models precisely. In Section 3, we build up a repertoire of techniques for programming a PRAM efficiently. We use these techniques in Section 4 to establish our main result: for every unit-cost RAM R with time complexity T, we construct a CREW PRAM that simulates R in time $O(T^{1/2} \log T)$. We conclude with a few comments in Section 5. All logarithms are taken to base 2.

Instruction	Meaning
$r(i) \leftarrow r(j)$	$r(i)$ gets $\langle r(j) \rangle$.
$r(i) \leftarrow (r(0))$	$r(i)$ gets $\langle r(j) \rangle$, where $j = \lvert\langle r(0) \rangle\rvert$.
$(r(0)) \leftarrow r(j)$	$r(i)$ gets $\langle r(j) \rangle$, where $i = \lvert\langle r(0) \rangle\rvert$.
$r(i) \leftarrow r(j) + r(k)$	$r(i)$ gets $\langle r(j) \rangle + \langle r(k) \rangle$.
$r(i) \leftarrow r(j) - r(k)$	$r(i)$ gets $\langle r(j) \rangle - \langle r(k) \rangle$.
JUMP q	If $\langle r(0) \rangle \leq \langle r(1) \rangle$, then jump to statement q.
ACCEPT	Accept and halt.
REJECT	Reject and halt.

Table 1: Instructions of a RAM

2 Definitions

2.1 The Unit-Cost RAM

A RAM R consists of a *memory*, and a *program*. The memory is an infinite sequence of registers $(r(i)), i = 0, 1, \ldots$. The *address* of $r(i)$ is the integer i. Each register can hold an integer. Let $\langle r(i) \rangle$ denote the content of $r(i)$ and $\lvert\langle r(i) \rangle\rvert$ denote the absolute value of $\langle r(i) \rangle$. The program consists of a finite number of *statements*, numbered $1, 2, \ldots, Q$. Each statement contains one instruction. The allowed instructions are shown in Table 1. The input of R is a binary number $\alpha = \alpha_0 \alpha_1 \cdots \alpha_{n-1}$, where each $\alpha_i \in \{0, 1\}$. Initially, $r(0), r(1), \ldots, r(K-1)$ hold some constant values required in the computation of R, where K is a constant that depends on R; $r(K+i)$ holds α_i for $0 \leq i < n$, and $r(K+n)$ holds -1 to mark the end of the input. All other registers contain 0. A unit-cost RAM executes each instruction in one step. Each step takes unit time. Thus, step t takes a unit-cost RAM from time $t-1$ to time t. The running time of a unit-cost RAM is the number steps performed.

2.2 The PRAM

A PRAM P comprises a collection of processors $P(0), P(1), \ldots$, which communicate via a global memory $(g(i))$. The initial contents of the global memory are as follows: the first K' global registers

hold some constants, where K' is another constant that depends on P; the next $n+1$ global registers hold the n input bits, followed by the end-of-input marker, and all other global registers contain 0. Every processor is a unit-cost RAM. Each $P(p)$ has its own local memory ($r_p(i)$) and can use every global memory register in the same manner as it uses a local memory register. In addition, each processor has an extra FORK q instruction for processor activation. Initially, only $P(0)$ is active. Whenever a processor executes a FORK q instruction, a new processor is activated and starts running at statement q. When $P(p)$ executes the FORK instruction the t^{th} time, processor $P(2^{t-1}(2p+1))$ is activated. The *processor id* (PID) of $P(p)$ is the integer p. When $P(p)$ is activated, its local register $r_p(0)$ is initialized with its PID p, and all other local registers of $P(p)$ contain 0. The PRAM P accepts if and only if $P(0)$ executes an ACCEPT instruction.

In a PRAM, several processors may attempt to access the same memory cell at the same time. A PRAM may allow concurrent-read and concurrent-write (CRCW) operations, concurrent-read and exclusive-write (CREW) operations, or exclusive-read and exclusive-write (EREW) operations [1, 22, 25]. In a CRCW PRAM, some mechanism is necessary to resolve the simultaneous write conflicts [1, 7, 21]. Fich et al. [4] studied the relationships between CRCW PRAM's with different conflict-resolution mechanisms.

In the sequel, we restrict our attention to CREW PRAM's. Unless otherwise stated, our results also hold for CRCW PRAM's.

3 Techniques for Programming PRAM's

In this section, we present several techniques for programming the PRAM. First, we show how to perform the following operations quickly on a PRAM: logical AND, summation, and multiplication on "small" integers. Secondly, we describe a fast implementation of multidimensional memory on

a PRAM. Thirdly, we explain how every processor can extract useful information from its PID efficiently.

3.1 Logical AND, Summation, and Multiple Memories

It is convenient to interpret integers as logical values. We interpret a nonzero integer as true and 0 as false.

Lemma 1 [folklore] *Suppose in a PRAM P, the global memory registers $g(1), g(2), \ldots, g(n)$ store n integers $k_1, k_2, \ldots, k_n$. Then P can find the sum and logical AND of these n integers in $O(\log n)$ time.*

By interleaving memory registers, Cook and Reckhow [2] demonstrated that a unit-cost RAM with a single memory can simulate a unit-cost RAM with multiple memories with merely a constant factor overhead in the running time. By applying the same technique to the PRAM, it is easy to prove the following lemma.

Lemma 2 [folklore] *Let $\gamma > 1$. A PRAM with time complexity T and γ global memories $(g_1(i)), (g_2(i)), \ldots, (g_\gamma(i))$ can be simulated in time $O(T)$ by a PRAM with one global memory.*

3.2 Multiplication on Small Integers

Trahan et al. [24] studied PRAM's with unit-time multiplication. By the following lemma, we may assume that ordinary PRAM's can perform unit-time multiplication on "small" integers.

Lemma 3 *Let P be a PRAM that (i) runs in time T, and (ii) can perform unit-time multiplication on T-bit integers. Then P can be simulated by an ordinary PRAM in time $O(T)$.*

Proof We use a PRAM P' with multiple memories to simulate P. Lemma 3 then follows from Lemma 2. P' simulates P step by step. We only need to show that P' can multiply in $O(1)$ time two T-bit integers. Multiplication reduces to squaring and halving via the identity $xy = ((x+y)^2 - x^2 - y^2)/2$. For two T-bit integers x and y, $x+y$ and $2xy$ are respectively at most $T+1$ and $2T+1$ bits long. It suffices to show that P' can perform in $O(1)$ time (i) squaring on $(T+1)$-bit positive integers and (ii) halving (right shift) on $(2T+1)$-bit positive integers. Before simulating P, P' precomputes a Square Table of size 2^{T+1} and a Right Shift Table of size 2^{2T+1}. Then during the simulation, P' can perform squaring and halving in $O(1)$ time by table lookup. It remains to demonstrate that the Square Table and the Right Shift Table can be precomputed in $O(T)$ time.

P' uses four global memories $(ls(i))$, $(rs(i))$, $(lsb(i))$, and $(sq(i))$ to implement four tables:

1. Left Shift Table — $\langle ls(i) \rangle = 2i$
2. Right Shift Table — $\langle rs(i) \rangle = \lfloor i/2 \rfloor$
3. Least Significant Bit Table — $\langle lsb(i) \rangle =$ least significant bit of i
4. Square Table — $\langle sq(i) \rangle = i^2$

P' initializes the first three tables for $0 \leq i < 2^{2T+1}$ as follows. In $O(T)$ time, P' activates processors $P(0), P(1), \ldots, P(2^{2T+1} - 1)$. Each processor does the following:

1. Store PID + PID in ls(PID).
2. Store PID in rs(PID + PID) and rs(PID + PID + 1).
3. Store PID $-$ $ls(rs$(PID)) in lsb(PID).

Obviously, Steps (1), (2), and (3) take $O(1)$ time. After the Left Shift Table, Right Shift Table, and Least Significant Bit Table are initialized, then for $0 \leq i < 2^{T+1}$, each $P(i)$ compute the square of

its PID using the paper-pencil multiplication method (repeated shift and add) and stores the result in $sq(i)$. This takes $O(T)$ time. Hence, all four tables can be precomputed in $O(T)$ time.

We have assumed that P' knows the value of T a priori. This assumption can be removed easily; P' just tries successive powers of 2 as an estimate of T. This modification does not increase the asymptotic running time of P'. □

3.3 Multidimensional Memory

A d-dimensional RAM is one with memory $(r(i_1, i_2, \ldots, i_d))$, where $i_1, i_2, \ldots, i_d \geq 0$. A d-dimensional PRAM is one with global memory $(g(i_1, i_2, \ldots, i_d))$; each processor of a d-dimensional PRAM is a d-dimensional RAM. Robson [19] showed that ordinary RAM's can simulate multidimensional RAM's with only a constant factor overhead in the running time. However, the proof of Robson cannot be adapted directly to prove the analogous result for PRAM's. Briefly, the reason is as follows. To simulate a RAM R with 2-dimensional memory $(r(i,j))$ by an ordinary RAM R' with memory $(r'(i))$, Robson devised a mapping from the $r(i,j)$'s to the $r'(i)$'s. This mapping depends on the sequence of $r(i,j)$'s accessed during the computation of R, and R' constructs this mapping incrementally as it simulates R step by step. Consider applying the same idea to simulate a PRAM P with 2-dimensional global memory $(g(i,j))$ by an ordinary PRAM P' with global memory $(g'(i))$. If we simulate each processor of P by a corresponding processor of P' as in the proof of Robson, then different processors of P may access the $g(i,j)$'s in different ways, and hence, different processors of P' may have different mappings. Thus, some processor of P' may think that the value of $g(0,0)$ is stored in $g'(0)$, whereas another processor of P' thinks that the same value is stored in $g'(1)$. Obviously, such a simulation of P by P' does not work.

All in all, the analogous result for PRAM's does hold, as shown by the next lemma.

Lemma 4 *A d-dimensional PRAM P running in time T can be simulated by an ordinary PRAM P' in time $O(T)$.*

Proof P' uses processor $P'(i)$ to simulate the corresponding processor $P(i)$ of P. Every $P'(i)$ simulates $P(i)$ step by step. It suffices to explain how to emulate d-dimensional memories by 1-dimensional memories. We demonstrate how $P'(i)$ emulates an access of $P(i)$ to the d-dimensional global memory of P by an access to the 1-dimensional global memory of P'. $P'(i)$ uses its 1-dimensional local memory to emulate the d-dimensional local memory of $P(i)$ in a similar fashion.

P has global memory $(g(i_1, i_2, \ldots, i_d))$; P' has global memory $(g'(i))$. In time T, $P(i)$ can produce integers no longer than CT bits for some constant C. Define $b = 2^{CT}$ and $\eta(i_1, i_2, \ldots, i_d) = \sum_{j=1}^{d} i_j b^{j-1}$. We map $g(i_1, i_2, \ldots, i_d)$ of P to $g'(\eta(i_1, i_2, \ldots, i_d))$ of P'. It is easy to verify that the $\eta(i_1, i_2, \ldots, i_d)$'s are distinct for $0 \leq i_1, i_2, \ldots, i_d < 2^{CT}$. To emulate an access to $g(i_1, i_2, \ldots, i_d)$ by $P(i)$, $P'(i)$ computes $\eta(i_1, i_2, \ldots, i_d)$ and accesses $g'(\eta(i_1, i_2, \ldots, i_d))$.

It remains to show that computing η takes $O(1)$ time. By repeated doubling, $b = 2^{CT}$ can be precomputed in $O(T)$ time. For $i_1, i_2, \ldots, i_d < 2^{CT}$, η is at most dCT bits long. By Lemma 3, we may assume that $P'(i)$ can perform multiplication on dCT-bit integers in $O(1)$ time. Thus, $P'(i)$ can compute η in $O(1)$ time.

Again, we have presumed that the value of T is available. This assumption can be removed in the same way as in the proof of Lemma 3. □

Lemma 4 shows that without loss of generality, we may assume that CREW PRAM's have multidimensional memories. Apparently, some authors have used this fact without proof [3, 16].

3.4 Extracting Information from PID

The advantage of a PRAM over a RAM is that in a PRAM, many processors can work together in parallel. Clearly, this advantage is defeated if all processors just do the same thing on the same data, in which case one processor is as good as many. To take advantage of the parallelism, therefore, different processors have to operate differently. This is easily achieved by exploiting the distinctness of the PID's; each processor consults its PID to determine its operation. For our later purpose, we require each processor to be able to look at successive single bits and successive $O(\log T)$ bits of its PID in order to determine its operation. Next, we demonstrate that every PRAM can be modified to fulfill this requirement.

Let P be a PRAM with time complexity T. In time T, P can activate at most 2^T processors. The PID of every processor is at most T bits long. We modify P as follows.

1. P activates all 2^T processors before any actual computation.

2. P starts its computation by initializing in $O(1)$ time a Least Significant Bit Table, a Right Shift Table, and a Left Shift Table, all of size 2^T, as described in the proof of Lemma 3. Using the first two tables, each processor can extract successive single bits of its PID, spending $O(1)$ time per bit.

3. P implements two additional tables with global memories $(lsb'(i))$ and $(rs'(i))$:

   (i) $lsb'(i) =$ the least significant $\lfloor \log T \rfloor$ bits of i

   (ii) $rs'(i) = \left\lfloor i/2^{\lfloor \log T \rfloor} \right\rfloor$, i.e., i right shifted $\lfloor \log T \rfloor$ bits

   These two tables can be precomputed in $O(\log T)$ time as follows. We presume the availability of the three tables mentioned in modification 2. For $0 \leq i < 2^T$, processor $P(i)$ does the following:

(i) Right shift its PID $\lfloor \log T \rfloor$ times and store the result in $rs'(i)$.

(ii) Left shift $rs'(i)$ $\lfloor \log T \rfloor$ times, subtract the result from its PID, and store the difference in $lsb'(i)$.

Then each processor can extract successive $\lfloor \log T \rfloor$ bits of its PID by table lookup, spending $O(1)$ time per $\lfloor \log T \rfloor$ bits. We have assumed that P knows a priori the values of T and $\lfloor \log T \rfloor$. The knowledge of T is justifiable, as argued in the proof of Lemma 3, and $\lfloor \log T \rfloor$ is simply the number of bits in the binary representation of T.

These modifications increase the running time of P by at most a constant factor.

4 Speedup of RAM's by PRAM's

We now prove that the PRAM is always faster than the RAM.

Theorem 5 *Every unit-cost RAM running in time T can be simulated by a CREW PRAM in time $O(T^{1/2} \log T)$ with $T^{O(T^{1/2})}$ processors.*

Let R be a unit-cost RAM with memory $(r(i))$ and time complexity $T = T(n)$. We devise a CREW PRAM P with multiple multidimensional memories that simulates R in time $O(T^{1/2} \log T)$. Theorem 5 then follows from Lemmas 2 and 4. Let M be a large enough constant so that every address in the program of R can be encoded in M bits; we choose M to be at least $3 \log 3 + 1$ to suit our later purpose. As the input length n tends to infinity, so does T, since $T(n) \geq n$. Consequently, if n exceeds some constant n_0, then $MT^{1/2} > \log(2(T + K + n + 1))$, where K is a constant that depends on R as explained in Section 2.1. It suffices to argue that P runs in $O(T^{1/2} \log T)$ time for $n > n_0$, since we can modify P to handle inputs of length less than n_0 by table lookup. We assume

that P knows in advance the value of $T^{1/2}$. Otherwise, P tries successive powers of 2 as an estimate of $T^{1/2}$.

4.1 Overview of Simulation

Fix an input $\alpha = \alpha_0\alpha_1 \cdots \alpha_{n-1}$ and consider the computation of R on α. The *configuration* of R at time t consists of the statement number of R at time t and the contents of all registers at time t. Denote by $config(t)$ the configuration of R at time t.

The simulation comprises two phases. In phase I, P uses $O(T^{1/2}\log T)$ time to activate $T^{1/2}$ groups of $T^{O(T^{1/2})}$ processors. For $1 \leq m \leq T^{1/2}$, the processors in group m perform some preprocessing such that after the preprocessing, $config(mT^{1/2})$ can be computed from $config((m-1)T^{1/2})$ in $O(\log T)$ time. All groups do the preprocessing simultaneously. In phase II, P finds $config(T)$ as follows. The initial configuration of R, $config(0)$, can be determined trivially. For $m = 1, 2, \ldots, T^{1/2}$, P computes $config(mT^{1/2})$ from $config((m-1)T^{1/2})$ in $O(\log T)$ time. Let q^* be the statement number in $config(T)$. P accepts if and only if statement q^* contains an ACCEPT instruction. Both phases take $O(T^{1/2}\log T)$ time. Next, we present an efficient representation of the configuration of R and then provide the details of phases I and II.

4.2 Representing the Configuration of R

P uses a data structure CONFIG to represent the configuration of R. One difficulty is that P cannot use a single register to store the content of a corresponding register of R. This is because R can generate integers as large as $2^{O(T)}$ in time T, but P can produce integers no larger than $T^{O(T^{1/2})}$ within the intended $O(T^{1/2}\log T)$ time bound. To overcome this difficulty, P divides every $O(T)$-bit integer into $N = O(T^{1/2})$ blocks, each $B = MT^{1/2}$ bits long. Without loss of generality, we assume

that R represents negative integers using sign-and-magnitude representation; thus, R works with nonnegative integers exclusively. With this simplifying assumption, every block in our blockwise representation is a nonnegative integer.

Initially, all registers of R contain 0, except for $r(0), r(1), \ldots, r(K+n)$. By convention, the first step of R is step 1, and $r(0), r(1), \ldots, r(K+n)$ are first written to in step 0 (i.e., they are initialized at time 0). For $0 \leq i \leq K+n$, let u_i denote $r(i)$. For $i > K+n$, if a new register of R is written to in step $i-K-n$, then let u_i denote this register; otherwise, u_i is undefined. To describe the configuration of R at time t, it suffices to specify the statement number of R at time t and the address and content at time t of u_i for $0 \leq i \leq t+K+n$.

CONFIG consists of three global memories $(a(i,j))$, $(c(i,j))$, and $(b(i))$. To represent the configuration of R at time t, register $b(0)$ holds the statement number of R at time t. If $0 \leq i \leq t+K+n$ and u_i is defined, then $c(i,j)$ holds the j^{th} block of B bits in $\langle u_i \rangle$. Thus, $\langle u_i \rangle = \sum_{j=0}^{N-1} \langle c(i,j) \rangle 2^{jB}$. For brevity, we say $c(i)$ holds $\langle u_i \rangle$, or $\langle c(i) \rangle = \langle u_i \rangle$, implying the blockwise representation. Register $a(i)$ holds the address of u_i in the same blockwise format. If $t < i-K-n \leq T$ or u_i is undefined, then $a(i)$ holds -1, and $c(i)$ is not used. The number of registers that CONFIG uses is therefore $O(NT) = O(T^{3/2})$.

Henceforth, when we mention *config*(t), we imply the above representation.

4.3 Phase I

4.3.1 Static, Dynamic, and Effective Instructions

Due to conditional jumps, a program statement may be executed more than once. For clarity, we distinguish between a *static instruction* and a *dynamic instruction.* The former is a static entity in a program statement of R. The latter is an executed instruction — an instance of a static

instruction during the computation of R. A static instruction may correspond to none or many dynamic instructions.

Divide the dynamic instructions of R into three types.

1. ACCEPT, REJECT, and JUMP

2. direct and indirect load and store instructions ($r(i) \leftarrow r(j)$, $r(i) \leftarrow (r(0))$, and $(r(0)) \leftarrow r(j)$)

3. arithmetic operations ($r(i) \leftarrow r(j) + r(k)$ and $r(i) \leftarrow r(j) - r(k)$)

Consider the effect of each dynamic instruction on the memory of R. A type 1 instruction does not change the content of any register. As far as the effect on the memory is concerned, a type 1 instruction is equivalent to a $u_0 \leftarrow u_0$ instruction. An instruction of type 2 copies the content of one register to another. Without loss of generality, we assume that $r(K-1)$ always holds 0. Reading from an uninitialized register is the same as reading from $r(K-1) = u_{K-1}$, since all uninitialized registers contain 0. The effect of a type 2 instruction is thus $u_{i'} \leftarrow u_{j'}$ for some $0 \leq i', j' \leq T+K+n$. A type 3 instruction finds the sum or difference of the contents of two registers and stores the result in a third register. The effect of a type 3 instruction is either $u_{i'} \leftarrow u_{j'} + u_{k'}$ or $u_{i'} \leftarrow u_{j'} - u_{k'}$ for some $0 \leq i', j', k' \leq T + K + n$. Therefore, each dynamic instruction is, in effect, of the form $u_{i'} \leftarrow u_{j'}$, $u_{i'} \leftarrow u_{j'} + u_{k'}$, or $u_{i'} \leftarrow u_{j'} - u_{k'}$ for some $0 \leq i', j', k' \leq T+K+n$. Since $T = T(n) \geq n$, there are $O(T^3)$ *effective instructions* of the above forms. Note that one static instruction may correspond to several dynamic instructions, each equivalent to a different effective instruction.

4.3.2 The Preprocessing

We fix m and describe the processors in group m. Let π_m be the triple (q_m, β_m, σ_m), where q_m is the statement number of R at time $(m-1)T^{1/2}$, σ_m is the sequence of $T^{1/2}$ effective instructions from time

$(m-1)T^{1/2}$ to $mT^{1/2}$, and β_m is a binary string that encodes the outcomes of all conditional jumps between time $(m-1)T^{1/2}$ and $mT^{1/2}$. For uniformity, we view every static and dynamic instruction as a conditional jump. An ACCEPT instruction, for example, may be viewed as a conditional jump where the condition is always **false**, and the destination of the jump is statement 1. In this way, β_m is always of length $T^{1/2}$. The triple π_m specifies the behavior of R between time $(m-1)T^{1/2}$ and $mT^{1/2}$. Using the information contained in π_m, group m performs some preprocessing that enables $config(mT^{1/2})$ to be computed from $config((m-1)T^{1/2})$ quickly. One problem is that group m does not know π_m in advance. To surmount this problem, group m uses enough processors to try all possible triples. Let Q be the number of statements in the program of R. The number of possible triples is thus $Q \times 2^{T^{1/2}} \times O(T^3)^{T^{1/2}} = T^{O(T^{1/2})}$. Group m uses $T^{O(T^{1/2})}$ processors, which can be activated in $O(T^{1/2} \log T)$ time. Each processor is responsible for a distinct triple, which is encoded in the processor's PID. All processors in all groups carry out their preprocessing simultaneously in parallel.

We focus on one specific processor P_π of group m, which is responsible for one particular triple $\pi = (q, \beta, \sigma)$. Notice the difference in notation: the PID of $P(p)$ is p, whereas the PID of P_π is not π, but the triple π is encoded in the PID of P_π. P_π decodes its PID to obtain q, β, and σ. To obtain the sequence of $T^{1/2}$ effective instructions σ, P_π extracts the least significant $O(T^{1/2} \log T)$ bits from its PID, $O(\log T)$ bits at a time. Every effective instruction can be encoded in $O(\log T)$ bits, since there are $O(T^3)$ different effective instructions. To recover the individual bits of β, P_π extracts the next $T^{1/2}$ bits from its PID, one bit at a time. The next $\lfloor \log Q \rfloor + 1$ bits of the PID constitute q. Using the techniques prescribed in Section 3.4, P_π can decode its PID in $O(T^{1/2})$ time. P_π saves all decoded information in tables so that it can access each bit of β and each effective instruction in $O(1)$ time by table lookup.

To facilitate our discussion, we say the triple π "happens" if the actual behavior of R conforms with the information contained in π. Now π may or may not happen. In phase I, P_π performs some preprocessing so that in phase II, once $config((m-1)T^{1/2})$ has been computed, P_π is able to decide in $O(\log T)$ time whether π actually happens and if so, computes $config(mT^{1/2})$ from $config((m-1)T^{1/2})$ in $O(\log T)$ time. Because of the way we represent the configuration of R (Section 4.2), to compute $config(mT^{1/2})$ from $config((m-1)T^{1/2})$, it suffices to determine the following:

1. The statement number of R at time $mT^{1/2}$.

2. For $(m-1)T^{1/2} < i - K - n \leq mT^{1/2}$, the address of u_i if u_i is defined.

3. For $0 \leq i \leq mT^{1/2} + K + n$, the content of u_i at time $mT^{1/2}$ if u_i is defined.

Below we explain the preprocessing that enables P_π to determine each of the above three items efficiently in phase II, assuming π actually happens.

4.3.3 The Line Number

Starting from statement q, P_π steps through the program of R statement by statement, following the flow of control defined by β. Meanwhile, P_π keeps track of the statement number of R. After $T^{1/2}$ steps, P_π obtains the statement number of R at time $mT^{1/2}$. This preprocessing takes $O(T^{1/2})$ time.

4.3.4 The Addresses of the u_i's

In $O(\log T)$ time, P_π activates $T^{1/2}$ processors P_i, where $(m-1)T^{1/2} < i - K - n \leq mT^{1/2}$. Each P_i is responsible for finding the address of u_i.

We fix i and describe P_i. P_i considers the effective instruction in step $s = i - K - n$ given by σ. Suppose this effective instruction is of the form $u_{i'} \leftarrow u_{j'}$. Other cases are handled similarly. If

$i' \neq i$, then no new register is written to in step s, and u_i is undefined. Otherwise, u_i is the register first written to in step s. P_i steps through the program of R in the same manner as described in Section 4.3.3 and finds the dynamic instruction in step s. If this dynamic instruction is of the form $r(j) \leftarrow r(k)$ or $r(j) \leftarrow (r(0))$, then the address of u_i is j. The blockwise representation of j is readily obtained, since all addresses in the program of R are at most $M \leq B$ bits long; the least significant B-bit block of j is just j itself, and all other blocks are 0. If the dynamic instruction in step s is of the form $(r(0)) \leftarrow r(k)$, then the address of u_i is the content of $r(0)$ at time $s-1$. Denote by $\langle u_i, t\rangle$ the content of u_i at time t. In Section 4.3.5, we explain the preprocessing for finding $\langle u_i, mT^{1/2}\rangle$. P_i performs the preprocessing for finding $\langle r(0), s-1\rangle = \langle u_0, s-1\rangle$ in a similar fashion.

4.3.5 The Contents of the u_i's

Since the sole arithmetic operations permitted are addition and subtraction, it follows that for a fixed π, $\langle u_i, mT^{1/2}\rangle$ is a linear combination of the $\langle u_i, (m-1)T^{1/2}\rangle$'s. Let

$$\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle u_j, (m-1)T^{1/2}\rangle,$$

where the C_{ij}'s are integer coefficients which depend only on π. In $O(\log T)$ time, P_π deploys $O(T^2)$ processors P_{ij}, where $0 \leq i, j \leq T+K+n$. Each P_{ij} finds C_{ij} in phase I.

We fix i and j and describe P_{ij}. P_{ij} creates an empty directed multigraph G and then processes the effective instructions specified by σ one by one. As P_{ij} considers each effective instruction, it inserts nodes and edges into G. P_{ij} marks each edge either "positive" or "negative." Node w is a positive child of node v if the edge (v, w) is positive. A negative child is defined analogously. Let v^+ and v^- respectively be the set of positive and negative children of v.

P_{ij} maintains a counter τ to keep track of the step corresponding to the effective instruction

currently under consideration. P_{ij} initializes τ to $(m-1)T^{1/2}+1$ and increments τ after every effective instruction. P_{ij} names each node either $[u_k, (m-1)T^{1/2}]$ or $[u_k, \tau]$ for some $0 \leq k \leq T+K+n$. Intuitively, node $[u_k, t]$ represents $\langle u_k, t\rangle$. For a node $v = [u_k, t]$, we write $\langle v\rangle$ for $\langle u_k, t\rangle$. The edges are marked such that

$$\langle v\rangle = \sum_{w \in v^+} \langle w\rangle - \sum_{w \in v^-} \langle w\rangle \quad \text{for each node } v \tag{1}$$

This will become clear after we explain how P_{ij} constructs G. After processing all $T^{1/2}$ effective instructions, P_{ij} uses G to obtain C_{ij}.

4.3.6 Constructing G

P_{ij} considers the $T^{1/2}$ effective instructions specified by σ one by one and constructs G as follows. For a $u_{i'} \leftarrow u_{j'}$ instruction, P_{ij} does the following. Create node $[u_{i'}, \tau]$. If G does not contain a node $[u_{j'}, \tau']$ for some $\tau' < \tau$, then create node $[u_{j'}, (m-1)T^{1/2}]$. Let $\tau_{j'} < \tau$ be maximum such that G contains node $[u_{j'}, \tau_{j'}]$. Insert edge $([u_{i'}, \tau], [u_{j'}, \tau_{j'}])$ and mark it positive.

P_{ij} processes a $u_{i'} \leftarrow u_{j'} - u_{k'}$ instruction as follows. Create node $[u_{i'}, \tau]$. If G does not contain a node $[u_{j'}, \tau']$ for some $\tau' < \tau$, then create node $[u_{j'}, (m-1)T^{1/2}]$. Similarly, if G does not contain a node $[u_{k'}, \tau']$ for some $\tau' < \tau$, then create node $[u_{k'}, (m-1)T^{1/2}]$. Let $\tau_{j'}, \tau_{k'} < \tau$ be maximum such that G contains nodes $[u_{j'}, \tau_{j'}]$ and $[u_{k'}, \tau_{k'}]$. Insert a positive edge $([u_{i'}, \tau], [u_{j'}, \tau_{j'}])$ and a negative edge $([u_{i'}, \tau], [u_{k'}, \tau_{k'}])$. A $u_{i'} \leftarrow u_{j'} + u_{k'}$ instruction is processed in the same way except that both of the inserted edges are positive.

It is mechanical to verify that the above construction yields a graph which satisfies (1), and every node of the graph has outdegree at most two. We illustrate the above construction with an example for P_π of group $m = 1$ with $T^{1/2} = 5$. Figure 1 shows the effective instructions specified by π. The graph constructed by P_{ij} appears in Figure 2.

Step	Instruction
1	$u_0 \leftarrow u_1$
2	$u_0 \leftarrow u_0 + u_0$
3	$u_0 \leftarrow u_0 - u_2$
4	$u_1 \leftarrow u_1 + u_0$
5	$u_0 \leftarrow u_0 + u_1$

Effect on Memory: $\langle u_0, 5\rangle = 5\langle u_1, 0\rangle - 2\langle u_2, 0\rangle$
$\langle u_1, 5\rangle = 3\langle u_1, 0\rangle - \langle u_2, 0\rangle$

In this example, $C_{01} = 5$, $C_{02} = -2$, $C_{11} = 3$, and $C_{12} = -1$.

Figure 1: The $T^{1/2} = 5$ effective instructions specified by π and their effect on the memory (example).

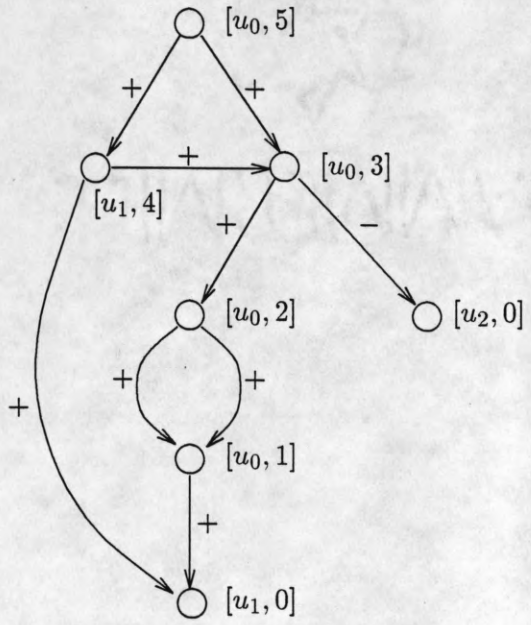

Figure 2: The graph G constructed by P_{ij} after processing the effective instructions in Figure 1.

4.3.7 Computing C_{ij}

We explain how P_{ij} uses G to compute C_{ij}. P_{ij} checks whether G contains a node $[u_i, \tau']$ for some $\tau' > (m-1)T^{1/2}$.

Case I (No such node exists) By construction of G, P_{ij} will create node $[u_i, \tau']$ if R writes to u_i in step τ'. The hypothesis thus implies R does not write to u_i between time $(m-1)T^{1/2}$ and $mT^{1/2}$. It follows that $\langle u_i, mT^{1/2}\rangle = \langle u_i, (m-1)T^{1/2}\rangle$. Ergo, $C_{ij} = 0$ for $j \neq i$, and $C_{ii} = 1$.

Case II (Otherwise) Let τ_i be maximum such that G contains node $[u_i, \tau_i]$. Similar arguments as in Case I give $\langle u_i, mT^{1/2}\rangle = \langle u_i, \tau_i\rangle$.

Consider the subgraph H of G induced by node $[u_i, \tau_i]$ and all its descendants. By construction, G (and hence H) is a directed acyclic multigraph. P_{ij} sorts the nodes in H topologically and labels each edge in H with an integer as follows. P_{ij} considers the nodes in H in topological order. For

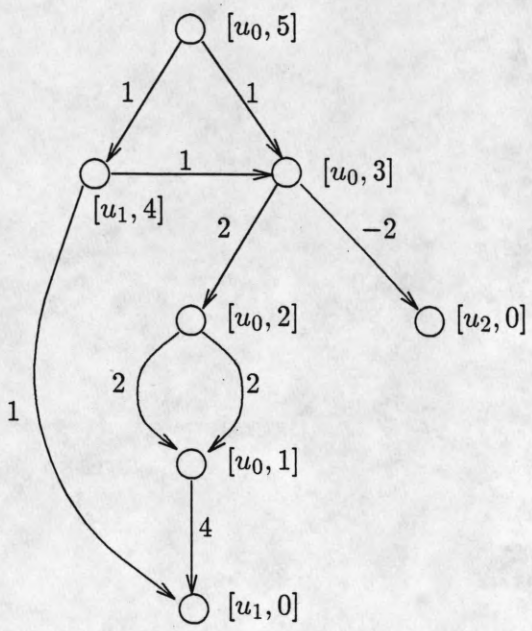

Figure 3: The result of applying the labeling algorithm of Section 4.3.7 to the graph in Figure 2.

each node v, P_{ij} labels the outgoing edges of v. When P_{ij} considers node v, all incoming edges of v are labeled, since P_{ij} considers the nodes in topological order. Evidently, the first node considered is $[u_i, \tau_i]$. P_{ij} labels every positive and negative outgoing edge of $[u_i, \tau_i]$ with 1 and -1 respectively. For each remaining node v in topological order, let $\lambda(H, v)$ be the sum of the labels on the incoming edges of v in H. P_{ij} labels each positive and negative outgoing edge of v with $\lambda(H, v)$ and $-\lambda(H, v)$ respectively. Figure 3 shows the result of applying the above labeling algorithm to the graph in Figure 2.

For $s > 0$, let H_s be the subgraph of H induced by all labeled edges after s nodes are considered. The *leaves* of H_s, denoted by $L(H_s)$, are the nodes in H_s with no outgoing edges. The following invariant is a consequence of (1): After s nodes are considered, $\langle u_i, \tau_i \rangle = \sum_{v \in L(H_s)} \lambda(H_s, v) \langle v \rangle$. Therefore, after all edges in H are labeled, $\langle u_i, mT^{1/2} \rangle = \sum_{v \in L(H)} \lambda(H, v) \langle v \rangle$. By construction,

every leaf of H is named $[u_j, (m-1)T^{1/2}]$ for some j. Hence, $C_{ij} = \lambda(H, [u_j, (m-1)T^{1/2}])$ if $[u_j, (m-1)T^{1/2}]$ is a leaf of H; otherwise, $C_{ij} = 0$. For the example in Figure 3, P_{01} determines that $C_{01} = 4 + 1 = 5$, and P_{02} concludes that $C_{02} = -2$.

In the above labeling algorithm, the sum of the absolute values of all the labels on the edges of H_s is at most triple that of H_{s-1}, since every node in H has outdegree at most two. The number of nodes in H is $|H| \leq |G| \leq 3T^{1/2}$, because at most three nodes are created for each of the $T^{1/2}$ effective instructions. Therefore, $|C_{ij}| \leq 3^{3T^{1/2}}$ for all i, j. Each C_{ij} is at most $B = MT^{1/2}$ bits long, since $M \geq 3 \log 3 + 1$.

The number of edges in G is $O(|G|)$, since each node has bounded outdegree. Constructing G and H, topologically sorting H, labeling the edges of H, and computing $\lambda(H, [u_j, (m-1)T^{1/2}])$ all take $O(|G|) = O(T^{1/2})$ time. Thus, the bottleneck in phase I is the activation of enough processors to try all possible π, which takes $O(T^{1/2} \log T)$ time.

4.3.8 Table Precomputation

In phase II, P has to extract efficiently the most and least significant B bits of a $2B$-bit integer. In phase I, P precomputes two tables $(h1(i))$ and $(h2(i))$ so that the first and second half of i can be extracted in $O(1)$ time by table lookup. P uses $O(T^{1/2})$ time to activate processors $P(0), P(1), \ldots, P(2^{2B} - 1)$ and builds up a Left Shift Table and a Right Shift Table of size 2^{2B} as in the proof of Lemma 3. Next, for $0 \leq i < 2^{2B}$, each $P(i)$ extracts in $O(T^{1/2})$ time the first and second halves of its PID as follows. The first half is obtained by shifting the PID right B times using the Right Shift Table. The second half is obtained by shifting the first half left B times and subtracting the first half from the PID. $P(i)$ stores the first and second halves in $h1(i)$ and $h2(i)$ respectively. Hence, the two tables $(h1(i))$ and $(h2(i))$ can be precomputed in $O(B) = O(T^{1/2})$ time.

4.4 Phase II

The data structure CONFIG has $O(T^{3/2})$ registers. In phase II, P initializes CONFIG in parallel using $O(\log T)$ time so that CONFIG contains *config*(0). For $m = 1, 2, \ldots, T^{1/2}$, the $T^{O(T^{1/2})}$ processors in group m do the following:

1. Each processor P_π in group m checks in $O(\log T)$ time whether π actually happens.

2. If so, compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ (stored in CONFIG) in $O(\log T)$ time and update CONFIG accordingly.

After $T^{1/2}$ updates, CONFIG contains *config*(T). P accepts if and only if statement q^* contains an ACCEPT instruction, where q^* is the statement number in *config*(T). Notice that in Step (1), exactly one P_π determines that π happens. So in Step (2), no write conflicts arise when updating CONFIG.

Next, we demonstrate that P_π can compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time, provided that π actually happens. In Section 4.5, we prove that $O(\log T)$ time suffices to verify whether π actually happens. The preprocessing of Section 4.3.3 yields the statement number at time $mT^{1/2}$ directly. For $(m-1)T^{1/2} < i - K - n \leq mT^{1/2}$, the preprocessing of Section 4.3.4 either gives the address of u_i directly or reduces the problem of finding the address of u_i to that of finding the content of u_0. It remains to explain how to determine the contents of the u_i's.

4.4.1 Computing Contents of Registers

We now explain how P_π computes the contents of the u_i's at time $mT^{1/2}$ from the contents of the u_i's at time $(m-1)T^{1/2}$. Suppose *config*$((m-1)T^{1/2})$ is available in CONFIG as described in Section 4.2. Then $c(j,k)$ holds the k^{th} B-bit block of $\langle u_j, (m-1)T^{1/2}\rangle$. Recall that

1. $\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle u_j, (m-1)T^{1/2}\rangle$.

2. In phase I, P_π dispatches processor P_{ij} to calculate C_{ij}.

3. C_{ij} is a B-bit integer.

The product $C_{ij}\langle c(j,k)\rangle$ is thus a $2B$-bit integer. In phase II, the P_{ij}'s cooperate to compute $\langle u_i, mT^{1/2}\rangle$ in $O(\log T)$ time as follows. The P_{ij}'s use four multidimensional global memories $(g(i_1,i_2,i_3,i_4))$, $(g'(i_1,i_2,i_3,i_4))$, $(h(i_1,i_2,i_3))$, and $(h'(i_1,i_2,i_3))$. Let p be the PID of P_π. In $O(\log T)$ time, every P_{ij} activates $O(T)$ processors P'_k, where $0 \le k \le T+K+n$. Each P'_k multiplies C_{ij} with $\langle c(j,k)\rangle$ and puts the most and least significant B bits of the product in $g'(p,i,j,(k+1))$ and $g(p,i,j,k)$ respectively. By Lemma 3, we may assume that the multiplication requires $O(1)$ time. Extracting the most and least significant B bits also takes $O(1)$ time as discussed in Section 4.3.8. Then

$$\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle c(j)\rangle \tag{2}$$

$$\langle c(j)\rangle = \sum_{k=0}^{N-1} \langle c(j,k)\rangle 2^{kB} \tag{3}$$

$$C_{ij}\langle c(j,k)\rangle = \langle g'(p,i,j,(k+1))\rangle 2^B + \langle g(p,i,j,k)\rangle \tag{4}$$

From (2), (3), and (4),

$$\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} \sum_{k=0}^{N} (\langle g'(p,i,j,k)\rangle + \langle g(p,i,j,k)\rangle) 2^{kB} \tag{5}$$

Next, P_π uses $O(\log T)$ time to deploy $O(T^{3/2})$ processors P'_{ik}, where $0 \le i \le T+K+n$ and $0 \le k \le N$. Each P'_{ik} computes the sum

$$\phi_{ik} = \sum_{j=0}^{T+K+n} (\langle g'(p,i,j,k)\rangle + \langle g(p,i,j,k)\rangle)$$

in $O(\log T)$ time (Lemma 1). The sum of $2(T+K+n+1)$ integers, each B bits long, is at most $B+\log(2(T+K+n+1)) \leq 2B$ bits long. P'_{ik} extracts the most and least significant B bits of ϕ_{ik} and places them in $h'(p,i,(k+1))$ and $h(p,i,k)$ respectively. Therefore,

$$\sum_{j=0}^{T+K+n} (\langle g'(p,i,j,k)\rangle + \langle g(p,i,j,k)\rangle) = h'(p,i,(k+1))2^B + h(p,i,k) \tag{6}$$

Let $\psi_i = \sum_{k=0}^{N+1} \langle h(p,i,k)\rangle 2^{kB}$ and $\psi'_i = \sum_{k=0}^{N+1} \langle h'(p,i,k)\rangle 2^{kB}$. From (5) and (6),

$$\langle u_i, mT^{1/2}\rangle = \sum_{k=0}^{N+1} (\langle h'(p,i,k)\rangle + \langle h(p,i,k)\rangle)2^{kB} = \psi'_i + \psi_i \tag{7}$$

Consider the carries into and out of the k^{th} B-bit block when we add ψ and ψ' together. By (7), the k^{th} block of $\langle u_i, mT^{1/2}\rangle$ is (roughly) $\langle h'(p,i,k)\rangle + \langle h(p,i,k)\rangle$, except that we have to adjust for the carries into and out of the k^{th} block. A carry into the block amounts to an increment by 1, whereas a carry out of the block is offset by subtracting 2^B. The value 2^B is precomputed during phase I in $O(B) = O(T^{1/2})$ time by repeated doubling. In Section 4.4.2, we show that all block-to-block carries can be determined in $O(\log T)$ time. To update $c(i)$ with $\langle u_i, mT^{1/2}\rangle$, every P'_{ik} finds the k^{th} block of $\langle u_i, mT^{1/2}\rangle$ (by adding $\langle h'(p,i,k)\rangle$ and $\langle h(p,i,k)\rangle$ and adjusting for the carries) and updates $c(i,k)$ accordingly. Hence, P_π is able to compute $config(mT^{1/2})$ from $config((m-1)T^{1/2})$ in $O(\log T)$ time during phase II.

In the above discussion, we have presumed that all C_{ij}'s are positive. Strictly speaking, to calculate $\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle c(j)\rangle$, we have to sum up the positive and the negative components separately using the above method, do a blockwise subtraction, and adjust for the block-to-block borrows. The calculation of the borrows is analogous to that of the carries.

4.4.2 Computing the Carries

Consider adding two $O(T)$-bit integers together. By parallel prefix computation [10, 17], it is possible to determine all the bit-to-bit carries in $O(\log T)$ time, provided that the individual bits of the integers are immediately accessible. In our case, however, the integers are represented in a blockwise instead of bitwise format. To apply the parallel prefix technique, we formulate the computation of the block-to-block carries as a prefix sum problem, in a way slightly different from that in the bitwise case. The idea is to let a block take the place of a bit. Define a binary operation $\otimes$ on $\{\bar{g}, \bar{s}, \bar{p}\}$ as follows:

$$x \otimes y = \begin{cases} y & \text{if } y \neq \bar{p} \\ x & \text{otherwise} \end{cases} \tag{8}$$

It is routine to check that $\otimes$ is associative. For $0 \leq k \leq N+1$, let

$$x_k = \begin{cases} \bar{g} & \text{if } \langle h'(p,i,k) \rangle + \langle h(p,i,k) \rangle \geq 2^B \\ \bar{p} & \text{if } \langle h'(p,i,k) \rangle + \langle h(p,i,k) \rangle = 2^B - 1 \\ \bar{s} & \text{otherwise} \end{cases}$$

Intuitively, $x_k = \bar{g}$ if a carry is "generated" in the k^{th} block; $x_k = \bar{p}$ if a carry is "propagated" through the k^{th} block (i.e., there is a carry out of the k^{th} block if and only if there is a carry into the k^{th} block), and $x_k = \bar{s}$ if a carry is "stopped" in the k^{th} block (i.e., no carry out of the k^{th} block regardless whether there is a carry into the k^{th} block). Let $x_{-1} = \bar{s}$, and for $-1 \leq k \leq N+1$, let $y_k = x_{-1} \otimes x_0 \otimes x_1 \cdots \otimes x_k$. By (8), $y_k = x_{k'}$, where $k' \leq k$ is maximum such that $x_{k'} \neq \bar{p}$. This implies $y_k = \bar{g}$ if and only if there is a carry out of the k^{th} block. By parallel prefix computation, we can determine all the y_k's, and hence all block-to-block carries, in $O(\log N) = O(\log T)$ time.

4.5 Verifying π

During phase I, P_π performs some additional preprocessing so that during phase II, P_π can decide in $O(\log T)$ time whether π actually happens. We first outline the verification process and then supply the details.

4.5.1 The Outline

Now π specifies the behavior of R between time $(m-1)T^{1/2}$ and $mT^{1/2}$. We say π "happens up to time t" if the behavior of R from time $(m-1)T^{1/2}$ to time t agrees with π. Similarly, we say π "happens in step t" if the behavior of R from time $t-1$ to time t agrees with π. P_π uses $T^{1/2}$ processors P_t^*, where $(m-1)T^{1/2} \leq t < mT^{1/2}$. Each P_t^* checks whether π happens in step $t+1$, assuming that π happens up to time t. Each P_t^* obtains a **true** or **false** answer. Clearly, π happens if and only if all these answers are **true**. P_π calculates the logical AND of these $T^{1/2}$ answers in $O(\log T)$ time (Lemma 1) and decides whether π actually happens.

4.5.2 Preprocessing for Verification

P_π activates all P_t^*'s in phase I using $O(\log T)$ time. Recall that in phase I, P_π performs some preprocessing based on the triple $\pi = (q, \beta, \sigma)$; if π happens, then this preprocessing enables P_π to compute $config(mT^{1/2})$ from $config((m-1)T^{1/2})$ in $O(\log T)$ time. During phase I, every P_t^* performs the analogous preprocessing using the triple $(q, \beta(t), \sigma(t))$, where $\beta(t)$ and $\sigma(t)$ are prefixes of β and σ respectively that define the behavior of R between time $(m-1)T^{1/2}$ and t. If π happens up to time t, then this preprocessing enables P_t^* to compute $config(t)$ from $config((m-1)T^{1/2})$ in $O(\log T)$ time. As argued in Section 4.3, this preprocessing takes $O(T^{1/2})$ time.

4.5.3 The Actual Verification

In Section 4.4, we discussed how P_π computes $config(mT^{1/2})$ from $config((m-1)T^{1/2})$, provided that π actually happens. In an analogous manner, each P_t^* computes $config(t)$ from $config((m-1)T^{1/2})$ during phase II, assuming that π actually happens up to time t. Using $config(t)$, P_t^* verifies whether π happens in step $t+1$ in $O(\log T)$ time.

If π actually happens, then every P_t^* obtains a positive answer (**true**), and P_π deduces that π actually happens. Otherwise, let t' be maximum such that π happens up to time t'. Then $P_{t'}^*$ determines $config(t')$ correctly and discovers that π does not happen in step $t'+1$. $P_{t'}^*$ answers **false**, and P_π infers that π does not happen. Note that for $t > t'$, the preprocessing of P_t^* yields nothing useful, and P_t^* cannot compute $config(t)$ correctly. This does not concern us, however, since the negative answer of $P_{t'}^*$ renders other answers immaterial. We merely need to guarantee that P_t^* finishes its preprocessing within $O(T^{1/2})$ time and produces some answer within $O(\log T)$ time. This is readily accomplished by having each P_t^* count the number of steps it executes.

4.5.4 Verifying a Single Step

It remains to explain how P_t^* uses $config(t)$ to check whether $\pi = (q, \beta, \sigma)$ happens in step $t+1$. Recall the following facts:

1. The integer q specifies the statement number of R at time $(m-1)T^{1/2}$.

2. We treat every static and dynamic instruction as a conditional jump, and β is a binary string of length $T^{1/2}$. For each dynamic instruction from step $(m-1)T^{1/2}+1$ to step $mT^{1/2}$, β stipulates whether the condition of the jump is **true** or **false**.

3. Every dynamic instruction is equivalent to an effective instruction, and σ gives the sequence of effective instruction from step $(m-1)T^{1/2}+1$ to step $mT^{1/2}$.

4. Every uninitialized register of R contains 0, and $r(K-1) = u_{K-1}$ contains 0 throughout the computation of R.

To decide whether π happens in step $t+1$, P_t^* performs the following checks:

1. **Check for** q: Let q' be the statement number in $config(t)$. If $t = (m-1)T^{1/2}$, then P_t^* checks that $q = q'$.

2. **Check for** β: Let $s = t-(m-1)T^{1/2}+1$. The s^{th} bit of β specifies whether the condition is true or false for the dynamic instruction in step $t+1$. This dynamic instruction corresponds to the static instruction in statement q'. P_t^* checks that the s^{th} bit of β is 1 if and only if statement q' indeed contains a JUMP instruction, and the condition of the jump is true, i.e., $\langle u_0, t\rangle \leq \langle u_1, t\rangle$ according to $config(t)$.

3. **Check for** σ: P_t^* checks that the effective instruction in step $t+1$ specified by σ is equivalent to the dynamic instruction in step $t+1$.

The first two checks need no further explanation. We supply the details of the third check below. In Section 4.3.1, we showed that every dynamic instruction is equivalent to an effective instruction of the form $u_{i'} \leftarrow u_{j'}$, $u_{i'} \leftarrow u_{j'} + u_{k'}$, or $u_{i'} \leftarrow u_{j'} - u_{k'}$ for some $0 \leq i', j', k' \leq T+K+n$. The dynamic instruction in step $t+1$ corresponds to the static instruction in statement q'. Consider the effective instruction in step $t+1$ specified by σ. P_t^* checks that the form of this effective instruction is "compatible" with the static instruction in statement q'. Table 2 shows the four categories of compatible instruction pairs. P_t^* performs some further checks according to the category of the compatible pair.

Category	Effective Instruction	Static Instruction
1	$u_0 \leftarrow u_0$	ACCEPT, REJECT, and JUMP
2	$u_{i'} \leftarrow u_{j'}$	$r(i) \leftarrow r(j)$, $r(i) \leftarrow (r(0))$, and $(r(0)) \leftarrow r(j)$
3	$u_{i'} \leftarrow u_{j'} + u_{k'}$	$r(i) \leftarrow r(j) + r(k)$
4	$u_{i'} \leftarrow u_{j'} - u_{k'}$	$r(i) \leftarrow r(j) - r(k)$

Table 2: Compatible effective and static instruction pairs.

Category 1 No further check is necessary.

Category 2 The static instruction in statement q' is either $r(i) \leftarrow r(j)$, $r(i) \leftarrow (r(0))$, or $(r(0)) \leftarrow r(j)$; σ stipulates that the effective instruction in step $t+1$ is $u_{i'} \leftarrow u_{j'}$. Let a_w be the address of the register that is written to in step $t+1$, and a_r be the address of the register that is read from in step $t+1$. More precisely,

$$a_w = \begin{cases} i & \text{if the static instruction in statement } q' \text{ is } r(i) \leftarrow r(j) \text{ or } r(i) \leftarrow (r(0)) \\ \langle r(0), t \rangle & \text{if the static instruction in statement } q' \text{ is } (r(0)) \leftarrow r(j) \end{cases}$$

$$a_r = \begin{cases} j & \text{if the static instruction in statement } q' \text{ is } r(i) \leftarrow r(j) \text{ or } (r(0)) \leftarrow r(j) \\ \langle r(0), t \rangle & \text{if the static instruction in statement } q' \text{ is } r(i) \leftarrow (r(0)) \end{cases}$$

According to σ, R reads from $u_{j'}$ and writes to $u_{i'}$ in step $t+1$. P_t^* compares a_r with the addresses of all u_k's in $config(t)$ in parallel. By definition, the addresses of all u_k's are distinct. If a_r equals the address of u_k for some k, then P_t^* checks that $j' = k$. Otherwise, R reads from an uninitialized register in step $t+1$; P_t^* checks that $j' = K - 1$.

If $i' = t + 1 + K + n$, then according to σ, a new register is written to in step $t+1$; P_t^* checks that a_w is different from the addresses of all u_k's in $config(t)$. If $i' < t + 1 + K + n$, then according to σ, R writes to $u_{i'}$ in step $t+1$, but not for the first time; P_t^* checks that a_w is the address of $u_{i'}$ in $config(t)$. If $i' > t + 1 + K + n$, then σ stipulates that in step $t+1$, R writes to $u_{i'}$, which by

definition is the register first written to in step $i' - K - n > t + 1$. The information contained in σ contradicts itself; P_t^* simply answers **false**.

P_t^* uses N processors (comparators) to compare two addresses in the blockwise format for equality. Each comparator checks for equality in a corresponding block and obtains a **true** or **false** answer; P_t^* calculates the logical AND of these answers in $O(\log N) = O(\log T)$ time (Lemma 1). To compare a_r and a_w against the addresses of all u_k's in parallel, P_t^* requires $O(NT) = O(T^{3/2})$ comparators. Observe that although these comparators are used in phase II, all comparators can be pre-activated in phase I using $O(\log T)$ time. We will need this observation in Section 4.6.

Categories 3 and 4 These cases are similar to those in Category 2.

Hence, P_π can verify whether π actually happens in $O(\log T)$ time. This concludes the proof of Theorem 5. □

4.6 Time-Processor Tradeoffs

In this section, we discuss how to reduce the number of processors used in the simulation at the expense of increasing the simulation time. The following theorem is a generalization of Theorem 5.

Theorem 6 *Let $\rho > 0$. Every unit-cost RAM that runs in time T can be simulated by a CREW PRAM in time $O(\rho \log T + (T \log \rho)/\rho)$ with $T^{O(\rho)}$ processors.*

Proof The proof of Theorem 6 is similar to that of Theorem 5. We explain how to modify the proof of Theorem 5 to establish Theorem 6. Instead of dividing each integer into $N = O(T^{1/2})$ blocks, We divide every $O(T)$-bit integer into $N' = O(\rho)$ blocks, each $O(T/\rho)$ bits long. We use T/ρ groups of processors. During phase I, group m performs the preprocessing based on the triple $(q'_m, \beta'_m, \sigma'_m)$, where q'_m is the statement number at time $(m-1)\rho$, β'_m is a binary string that encodes the outcomes of all condition jumps from time $(m-1)\rho$ to time $m\rho$, and σ'_m is the sequence of ρ

effective instructions between time $(m-1)\rho$ and $m\rho$. In phase I, group m uses $O(\rho \log T)$ time to activate $T^{O(\rho)}$ processors to try all possible triples. Similar analysis as in Section 4.3 reveals that the preprocessing takes $O(\rho)$ time. Again, the bottleneck in phase I is the activation of enough processors to try all triples, which takes $O(\rho \log T)$ time.

The content of each u_i at time $m\rho$ is a linear combination of the $\langle u_i, (m-1)\rho \rangle$'s. Let $\langle u_i, m\rho \rangle = \sum_j C'_{ij} \langle u_j, (m-1)\rho \rangle$. Observe that such a linear combination has at most $O(\rho)$ nonzero coefficients, since all arithmetic operations between time $(m-1)\rho$ and $m\rho$ involve at most $O(\rho)$ registers. Let $J = \{j \mid C'_{ij} \neq 0\}$. In Section 4.4.1, we described how to compute $\sum_{j=0}^{T+K+n} C_{ij} \langle u_j, (m-1)T^{1/2} \rangle$ in $O(\log N + \log(T+K+n)) = O(\log T)$ time. Using the same method, we can compute $\langle u_i, m\rho \rangle = \sum_{j \in J} C'_{ij} \langle u_j, (m-1)\rho \rangle$ in $O(\log N' + \log |J|) = O(\log \rho)$ time. Verification of the triple also takes $O(\log \rho)$ time. Note that the verification of the triple requires $O(T\rho)$ comparators, which can be pre-activated in $O(\log T + \log \rho)$ time during phase I. The preprocessing thus enables group m to compute *config*$(m\rho)$ from *config*$((m-1)\rho)$ in $O(\log \rho)$ time during phase II.

In phase II, the PRAM P computes *config*(T) in $O((T \log \rho)/\rho)$ time as follows. For $m = 1, 2, \ldots, T/\rho$, group m computes *config*$(m\rho)$ from *config*$((m-1)\rho)$ in $O(\log \rho)$ time. Let q^* be the statement number in *config*(T). P accepts if and only if statement q^* contains an ACCEPT instruction. This simulation takes $O(\rho \log T + (T \log \rho)/\rho)$ time and uses $T^{O(\rho)}$ processors. □

5 Discussion

5.1 Parallelism Always Helps

We have shown that we can always speed up a sequential computation on a unit-cost RAM by a CREW PRAM. We mentioned in Section 1.1 that the unit-cost RAM is the most commonly used

machine model for analyzing sequential algorithms. There are, however, other machine models of sequential computation, for example, Turing machine, tree Turing machine, multidimensional Turing machine, and log-cost RAM. In a separate paper [11], we show that a sequential computation on each of these other models can also be sped up by a corresponding parallel machine model:

1. Every tree Turing machine that runs in time T can be simulated by an alternating Turing machine in time $O(T/\log T)$.

2. Every d-dimensional Turing machine that runs in time T can be simulated by an alternating Turing machine in time $O(T5^{d\log^* T}/\log T)$.

3. Every log-cost RAM that runs in time T can be simulated by an alternating log-cost RAM in time $O(T\log\log T/\log T)$.

We conclude that parallelism always helps us speed up a sequential computation.

5.2 Speedup Using a Polynomial Number of Processors

It is well-known that the Turing machine enjoys the constant speedup theorem [26]: Let $\epsilon > 0$ and M be a Turing machine with time complexity T; then M can be simulated by another Turing machine in time $\epsilon T + n$. Hence, efforts on speeding up the Turing machine have focused on asymptotic speedup [3, 9, 15]. The unit-cost RAM, however, does not enjoy the constant speedup theorem [23]; that is, there exist an $\epsilon > 0$ and a unit-cost RAM R with time complexity T such that R cannot be simulated by any unit-cost RAM in time $\epsilon T + n$. Thus, it is not trivial to speed up the computation of a unit-cost RAM by a constant factor. Theorem 6 shows that it is possible to speed up a unit-cost RAM by an arbitrary constant factor with a CREW PRAM using a polynomial number of processors.

5.3 Is Result Optimal?

We have constructed a simulator that runs in $O(T^{1/2} \log T)$ time. We do not know whether our result is optimal, but we believe that it is difficult to reduce the simulation time by more than a $\log T$ factor, because this would imply improvements over some best known results, as explained below. We would like to call the reader's attention to the following previously established results:

1. Every CREW PRAM that runs in time T can be simulated by a Turing machine in space $O(T^2)$ (Fortune and Wyllie, 1978 [5]).

2. Every Turing machine that runs in time T can be simulated by a unit-cost RAM in time $O(T/\log T)$ (Hopcroft, Paul, and Valiant, 1975 [9]).

3. Every Turing machine that runs in time T can be simulated by another Turing machine in space $O(T/\log T)$ (Hopcroft, Paul, and Valiant, 1977 [8]).

4. Every Turing machine that runs in time T can be simulated by a CREW PRAM in time $O(T^{1/2})$ (Dymond and Tompa, 1985 [3]).

These are the best known results for the respective simulations. For our problem, namely, simulation of unit-cost RAM's by CREW PRAM's, reducing the simulation time to $o((T \log T)^{1/2})$, together with the first result of Hopcroft et al. above, implies an improvement over the result of Dymond and Tompa. By the same reasoning, if we manage to reduce the simulation time to $o(T^{1/2})$, then we can simulate every Turing machine with time complexity T by a CREW PRAM in time $o((T/\log T)^{1/2})$. It then follows from the above result of Fortune and Wyllie that for Turing machines, time T can be simulated in space $o(T/\log T)$, improving the second result of Hopcroft et al. above. This would be a significant breakthrough in simulating time by space for Turing machines.

References

[1] Borodin, A., and Hopcroft, J. E. Routing, merging, and sorting on parallel models of computation. *J. Comput. System Sci. 30* (1985), 130–145.

[2] Cook, S. A., and Reckhow, R. A. Time bounded random access machines. *J. Comput. System Sci. 7* (1973), 354–375.

[3] Dymond, P. W., and Tompa, M. Speedups of deterministic machines by synchronous parallel machines. *J. Comput. System Sci. 30* (1985), 149–161.

[4] Fich, F. E., Ragde, P., and Wigderson, A. Relations between concurrent-write models of parallel computation. *SIAM J. Comput. 17* (1988), 606–627.

[5] Fortune, S., and Wyllie, J. Parallelism in random access machines. In *Proc. 10th Ann. ACM Symp. on Theory of Computing* (1978), pp. 114–118.

[6] Gibbons, A., and Rytter, W. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, England, 1988.

[7] Goldschlager, L. M. A universal interconnection pattern for parallel computers. *J. Assoc. Comput. Mach. 29* (1982), 1073–1086.

[8] Hopcroft, J., Paul, W., and Valiant, L. On time versus space. *J. Assoc. Comput. Mach. 24*, 2 (1977), 332–337.

[9] Hopcroft, J. E., Paul, W. J., and Valiant, L. G. On time versus space and related questions. In *Proc. 16th Ann. IEEE Symp. on Foundations of Computer Science* (1975), pp. 57–64.

[10] LEIGHTON, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[11] MAK, L. Are parallel machines always faster than sequential machines? Submitted to *SIAM J. Comput.* for publication (1993).

[12] PARBERRY, I. Parallel speedup of sequential machines: a defense of the parallel computation thesis. *ACM SIGACT News 18* (1986), 54–67.

[13] PARBERRY, I. *Parallel Complexity Theory.* Wiley, New York, 1987.

[14] PARBERRY, I., AND SCHNITGER, G. Parallel computation with threshold functions. *J. Comput. System Sci. 36* (1988), 278–302.

[15] PAUL, W., AND REISCHUK, R. On alternation II. *Acta Inform. 14* (1980), 391–403.

[16] REIF, J. H. On synchronous parallel computations with independent probabilistic choice. *SIAM J. Comput. 13* (1984), 46–56.

[17] REIF, J. H., Ed. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, California, 1993.

[18] ROBSON, J. M. Fast probabilistic RAM simulation of single tape Turing machine computations. *Inform. and Control 63* (1984), 67–87.

[19] ROBSON, J. M. Random access machines with multi-dimensional memories. *Inform. Process. Lett. 34* (1990), 265–266.

[20] ROBSON, J. M. Deterministic simulation of a single tape Turing machine by a random access machine in sub-linear time. *Inform. and Comput. 99* (1992), 109–121.

[21] SHILOACH, Y., AND VISHKIN, U. Finding the maximum, merging, and sorting in parallel computation model. *J. Algorithms 2* (1981), 88–102.

[22] SNIR, M. On parallel searching. *SIAM J. Comput. 14* (1985), 688–708.

[23] SUDBOROUGH, I. H., AND ZALCBERG, A. On families of languages defined by time-bounded random access machines. *SIAM J. Comput. 5* (1976), 217–230.

[24] TRAHAN, J. L., LOUI, M. C., AND RAMACHANDRAN, V. Multiplication, division, and shift instructions in parallel random access machines. *Theoret. Comput. Sci. 100* (1992), 1–44.

[25] VISHKIN, U. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms 4* (1983), 45–50.

[26] YAP, C. K. *Theory of Complexity Classes.* Oxford University Press, Oxford, England, to appear.