Theoretical Computer Science 100 (1992) 1–44
Elsevier

Fundamental Study

Multiplication, division, and shift instructions in parallel random access machines*

Jerry L. Trahan

Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA

Michael C. Loui

Department of Electrical and Computer Engineering, and Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Vijaya Ramachandran

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA

Communicated by A. Schönhage
Received November 1989
Revised May 1991

Abstract

Trahan, J.L., M.C. Loui and V. Ramachandran, Multiplication, division and shift instructions in parallel random access machines, Theoretical Computer Science 100 (1992) 1–44.

We prove that polynomial time on a parallel random access machine (PRAM) with unit-cost multiplication and division or on a PRAM with unit-cost shifts is equivalent to polynomial space on a Turing machine (PSPACE). This extends the result that polynomial time on a basic PRAM is equivalent to PSPACE to hold when the PRAM is allowed multiplication or division or unrestricted shifts. It also extends to the PRAM the results that polynomial time on a random access machine (RAM) with multiplication is equivalent to PSPACE and that polynomial time on a RAM with

* A preliminary version of a portion of this work appeared in the proceedings of the 22nd Conference on Information Sciences and Systems held in 1988 at Princeton, NJ. This work was conducted at the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign with the support of the Joint Services Electronics Program (U.S. Army, U.S. Navy, U.S. Air Force) under Contract N00014-84-C0149. The work of the second author was also supported in part by the National Science Foundation under grant CCR-8922008.

shifts (that is, a vector machine) is equivalent to PSPACE. We obtain simulations of uniform circuits by RAMs with enhanced instruction sets and use the enhanced RAMs to simulate PRAMs with enhanced instruction sets.

Contents

1. Introduction

An important model of parallel computation is the *parallel random access machine* (PRAM), which comprises multiple processors that execute instructions synchronously and share a common memory. Formalized by Fortune and Wyllie [9] and Goldschlager [10], the PRAM is a much more natural model of parallel computation than older models such as combinational circuits and alternating Turing machines [19] because the PRAM abstracts the salient features of a modern multiprocessor computer. The PRAM provides the foundation for the design of highly parallel algorithms [14]. This model permits the exposure of the intrinsic parallelism in a computational problem because it simplifies the communication of data through a shared memory.

In this paper, we study the effect of the instruction set on the performance of the PRAM. The basic PRAM has unit-cost addition, subtraction, Boolean operations, comparisons, and indirect addressing. To quantify differences in computational

performance, we determine the time complexities of simulations between PRAMs with different instruction sets. We focus on the computational complexity of simulations between enhanced PRAMs with the following additional unit-time operations:

multiplication,
division,
arbitrary left shift,
arbitrary right shift.

Further, to better understand the effects of parallelism in the PRAM, it is necessary to view the model in relation to the sequential RAM. We bound the time for an enhanced RAM to simulate a similarly enhanced PRAM. Thus, we study enhanced PRAMs by stripping away one feature at a time, and we are able to observe the gain in time-bounded computational power due to individual features.

Let PRAM[*op*] denote the class of PRAMs with the basic instruction set augmented with the set *op* of instructions. Let PRAM[*op*]-TIME($T(n)$) denote the class of languages recognized by PRAM[*op*]s in time $O(T(n))$ on inputs of length n, PRAM[*op*]-PTIME the union of PRAM[*op*]-TIME($T(n)$) over all polynomials $T(n)$, and PRAM[*op*]-POLYLOGTIME the union of PRAM[*op*]-TIME($T(n)$) over all $T(n)$ that are polynomials in $\log n$. Let RAM[*op*], RAM[*op*]-TIME($T(n)$), RAM[*op*]-PTIME, and RAM[*op*]-POLYLOGTIME denote the analogous classes for the sequential random access machine (RAM) model.

We prove that polynomial time on a PRAM with unit-time multiplication and division or on a PRAM with unit-time unrestricted shifts is equivalent to polynomial space on a Turing machine (TM). Consequently, a PRAM with unit-time multiplication and division and a PRAM with unit-time unrestricted shifts are at most polynomially faster than a standard PRAM, which does not have these powerful instructions. These results are surprising for two reasons. First, for a sequential RAM, adding unit-time multiplication ($*$) or unit-time unrestricted left shift ($\uparrow$) seems to increase its power:

$$\text{RAM-PTIME} = \text{PTIME [6]},$$

$$\text{RAM}[*]\text{-PTIME} = \text{PSPACE [12]},$$

$$\text{RAM}[\uparrow]\text{-PTIME} = \text{PSPACE [15, 22]},$$

whereas adding one of these operations to a PRAM does *not* increase its power by more than a polynomial in time. Second, despite the potential speed offered by massive parallelism, a sequential RAM with unit-cost multiplication or unrestricted shifts is just as powerful, within a polynomial amount of time, as a PRAM with the same additional operation.

We establish the following new facts about PRAMs. Recall that PSPACE = PRAM-PTIME and POLYLOGSPACE = PRAM-POLYLOGTIME

[9]. Let $\downarrow$ denote unrestricted right shift.

$$\begin{aligned}\text{PSPACE} &= \text{PRAM}[*]\text{-PTIME} \\ &= \text{PRAM}[*, \div]\text{-PTIME} \\ &= \text{PRAM}[\div]\text{-PTIME} \\ &= \text{PRAM}[\uparrow, \downarrow]\text{-PTIME},\end{aligned} \tag{1}$$

$$\begin{aligned}\text{POLYLOGSPACE} &= \text{PRAM-POLYLOGTIME} \\ &= \text{PRAM}[*]\text{-POLYLOGTIME} \\ &= \text{PRAM}[*, \div]\text{-POLYLOGTIME} \\ &= \text{PRAM}[\div]\text{-POLYLOGTIME} \\ &= \text{PRAM}[\uparrow, \downarrow]\text{-POLYLOGTIME}.\end{aligned} \tag{2}$$

The results in (1) are the parallel analogues of the results of Hartmanis and Simon [12] and Simon [22] for sequential RAMs. Because of the very long numbers that the RAM$[*]$ and RAM$[\uparrow, \downarrow]$ can generate and because of the equivalence of polynomial time on these models to PSPACE, the RAM$[*]$ and RAM$[\uparrow, \downarrow]$ have sometimes been viewed as "parallel". Thus, the PRAM$[*]$ and PRAM$[\uparrow, \downarrow]$ may be viewed as "doubly parallel". The results in (1) are therefore also significant in that introducing unbridled parallelism to a random access machine with unit-time multiplication or unit-time unrestricted shift decreases the running time by at most a polynomial amount.

The results in (2) are notable because of their possible implications for the robust class NC, which can be characterized by several different models of parallel computation [5]. If we could reduce the number of processors used by the simulation of a PRAM$[*]$, PRAM$[*, \div]$, or PRAM$[\uparrow, \downarrow]$ by a PRAM from an exponential number to a polynomial number, then NC would be the class of languages accepted by PRAM$[*]$s, PRAM$[*, \div]$s, or PRAM$[\uparrow, \downarrow]$s, respectively, in polylog time with a polynomial number of processors.

For the sequential RAMs, we know by the above results that time on the RAM$[op]$ is polynomially related to time on the PRAM$[op]$, since the class of languages accepted in polynomial time on any of these models is equivalent to PSPACE [12, 22]. We obtain tighter time bounds by the simulations given here.

The simulations are performed through uniform, bounded fan-in circuits. We prove that a RAM$[op]$ can efficiently simulate a uniform, bounded fan-in circuit and then show that the circuits that simulate a PRAM$[op]$ meet the uniformity conditions.

In another paper [26], we combined multiplication and shifts in the instruction set and proved NEXPTIME $\subseteq$ PRAM$[*, \uparrow, \downarrow]$-PTIME $\subseteq$ EXPSPACE. Thus, the combination of multiplication and shifts is more powerful, to within a polynomial in time, than either instruction separately.

The thesis by Trahan [25] gives the detailed proofs of lemmas for which only outlines of proofs appear in this paper.

2. Definitions and a key lemma

2.1. PRAM definitions

We study a deterministic PRAM similar to that of Stockmeyer and Vishkin [24]. A PRAM consists of an infinite collection of processors $P_0, P_1, \ldots$, an infinite set of shared memory cells, $c(0), c(1), \ldots$, and a program which is a finite set of instructions labeled consecutively with $1, 2, 3, \ldots$. All processors execute the same program. Each processor has a program counter. Each processor P_m has an infinite number of local registers: $r_m(0), r_m(1), \ldots$. Each cell $c(j)$, whose *address* is j, contains an integer $con(j)$, and each register $r_m(j)$ contains an integer $rcon_m(j)$.

For convenience we use a PRAM with concurrent read and concurrent write (CRCW) in which the lowest numbered processor succeeds in a write conflict. Since we are concerned with at least polylog time, there are no significant differences between the concurrent read/concurrent write (CRCW), the concurrent read/exclusive write (CREW), and the exclusive read/exclusive write (EREW) PRAMs because the EREW model can simulate the CRCW model with a penalty of only a logarithmic factor in time (log of the number of processors attempting to simultaneously read or write) [8, 27]. If one or more processors attempt to read a cell at the same time that a processor is attempting to write the same cell, then all reads are performed before the write.

Initially, the input, a nonnegative integer, is in $c(0)$. For all m, register $r_m(0)$ contains m. All other cells and registers contain 0, and only P_0 is active. A PRAM accepts its input if and only if P_0 halts with its program counter on an ACCEPT instruction.

In time $O(\log n)$, a processor can compute the smallest n such that $con(0) \leqslant 2^n - 1$; the PRAM takes this n as the length of the input. Whenever $con(i)$ is interpreted in two's complement representation, we number the bits of $con(i)$ consecutively with $0, 1, 2, \ldots$, where bit 0 is the rightmost (least significant) bit.

We allow indirect addressing of registers and shared memory cells through register contents. The notation $c(r_m(j))$ refers to the cell of shared memory whose address is $rcon_m(j)$, and $r(r_m(j))$ refers to the register of P_m whose address is $rcon_m(j)$.

The basic PRAM model has the following instructions. When executed by processor P_m, an instruction that refers to register $r(i)$ uses $r_m(i)$.

$r(i) \leftarrow k$ (load a constant)
$r(i) \leftarrow r(j)$ (load the contents of another register)
$r(i) \leftarrow c(r(j))$ (indirect read from shared memory)
$c(r(i)) \leftarrow r(j)$ (indirect write to shared memory)
$r(i) \leftarrow r(r(j))$ (indirect read from local memory)
$r(r(i)) \leftarrow r(j)$ (indirect write to local memory)

ACCEPT (halt and accept)
REJECT (halt and reject)
*FORK label*1, *label*2 (P_m halts and activates P_{2m} and P_{2m+1}, setting their program counters to *label*1 and *label*2, respectively.)
$r(i) \leftarrow BIT(r(j))$ (read the $rcon_m(j)$th bit of $con(0)$ (the input))
CJUMP $r(j)\, comp\ r(k)$, *label* (jump to instruction labeled *label* on condition $rcon_m(j)\, comp\, rcon_m(k)$, where the arithmetic comparison $comp \in \{<, \leqslant, =, \geqslant, >, \neq\}$),
$r(i) \leftarrow r(j) \bigcirc r(k)$ for $\bigcirc \in \{+, -, \wedge, \neg\}$ (addition, subtraction, bitwise Boolean AND and negation)

Processor P_0 can perform a *FORK* operation only once. This restriction is necessary to prevent the activation of multiple processors with identical processor numbers. This is also the reason why P_m halts when it performs a *FORK*. With the *FORK* instruction, at most 2^t processors are active at time t in a computation of a PRAM.

In some variants of the PRAM model, the input is initially located in the first n cells, one bit per cell. We therefore have the instruction "$r(i) \leftarrow BIT(r(j))$" in order to allow the PRAM to transform the input to this format in $O(\log n)$ time. This instruction was also used by Reischuk [18].

For an integer d, define its length $len(d)$ as the minimum integer w such that $-2^{w-1} \leqslant d \leqslant 2^{w-1} - 1$. Thus, d has a two's complement representation with w bits. Let $w = \max\{len(rcon_m(j)), len(rcon_m(k))\}$. Let $\# d$ denote the two's complement representation of d. To perform a Boolean operation on $rcon_m(j)$ and $rcon_m(k)$, the PRAM performs the operation bitwise on the w-bit two's complement representations of $rcon_m(j)$ and $rcon_m(k)$. The PRAM interprets the resulting integer x in w-bit two's complement representation and writes x in $r_m(i)$. We need at least w bits so that the result in correctly positive or negative.

Let us assume that the division instruction returns the quotient. Let $\uparrow$ ($\downarrow$) denote the unrestricted left (right) shift operation: the instruction $r(i) \leftarrow r(j) \uparrow r(k)$ ($r(i) \leftarrow r(j) \downarrow r(k)$) places $rcon_m(j) \cdot 2^{rcon_m(k)}$ (the integer part of $rcon_m(j) \div 2^{rcon_m(k)}$) into $r_m(i)$. The instruction can also be viewed as placing into $r_m(i)$ the result of shifting the binary integer $rcon_m(j)$ to the left (right) by $rcon_m(k)$ bit positions.

At each step, each active processor simultaneously executes the instruction indicated by its program counter in one unit of time, then increments its program counter by one, unless the instruction causes a jump. On an attempt to read a cell at a negative address, the processor reads the value 0; on an attempt to write a cell at a negative address, the processor does nothing.

The assumption of unit-time instruction execution is an essential part of our definition. In a sense, our work is a study of the effects of this unit-cost hypothesis on the computational power of time-bounded PRAMs as the instruction set is varied.

For ease of description, we sometimes allow a PRAM a small constant number of separate memories, which can be interleaved. This allowance entails no loss of generality and only a constant factor time loss.

A PRAM Z has time bound $T(n)$ if for all inputs ω of length n, a computation of Z on ω halts in $T(n)$ steps. Z has processor bound $P(n)$ if for all inputs ω of length n, Z activates at most $P(n)$ processors during a computation on ω. We assume that $T(n)$ and $P(n)$ are both time-constructible in the simulations of a PRAM$[op]$ by a PRAM, so that all processors have values of $T(n)$ and $P(n)$.

Let R be a PRAM$[*]$. By repeated application of the multiplication instruction, R can generate integers of length $O(n2^{T(n)})$ in $T(n)$ steps. By indirect addressing, processors in R can access cells with addresses up to $2^{n2^{T(n)}}$ in $T(n)$ steps, although R can access at most $O(P(n)T(n))$ different cells during its computation. In subsequent sections, these cell addresses will be too long for the simulating machines to write. Therefore, we first construct a PRAM$[*]$ R' that simulates R and uses only short addresses. Similarly, a PRAM$[\uparrow, \downarrow]$ can generate extremely long integers and use them as indirect addresses, so we simulate this by a PRAM$[\uparrow, \downarrow]$ that uses only short addresses.

Associative Memory Lemma. *Let* $op \subseteq \{*, \div, \uparrow, \downarrow\}$. *For all* $T(n)$ *and* $P(n)$, *every language recognized with* $P(n)$ *processors in time* $T(n)$ *by a* PRAM$[op]$ R *can be recognized in time* $O(T(n))$ *by a* PRAM$[op]$ R' *that uses* $O(P^2(n)T(n))$ *processors and accesses only cells with addresses in* $0, \ldots, O(P(n)T(n))$.

Proof. Let R be an arbitrary PRAM$[op]$ with time bound $T(n)$ and processor bound $P(n)$. We construct a PRAM$[op]$ R' that simulates R in time $O(T(n))$ with $P^2(n)T(n)$ processors, but accesses only cells with addresses in $0, \ldots, O(P(n)T(n))$. R' employs seven separate shared memories: $mem_1, \ldots, mem_7$. Let $c_b(k)$ denote the kth cell of mem_b and $con_b(k)$ the contents of that cell. R' organizes the cells of mem_1 and mem_2 in pairs to simulate the memory of R: the first component, $c_1(k)$, holds the address of a cell in R; the second component, $c_2(k)$, holds the contents of that cell. Actually, in order to distinguish address 0 from an unused cell, $c_1(k)$ holds one plus the address. Let $pair(k)$ denote the kth memory pair. R' organizes the cells of mem_3, mem_4, and mem_5 in triples to simulate the local registers of R: the first component, $c_3(k)$, holds the processor number; the second component, $c_4(k)$, holds the address of a register in R; the third component, $c_5(k)$, holds the contents of that register. Let $triple(k)$ denote the kth memory triple. Since R can access at most $O(P(n)T(n))$ cells in $T(n)$ steps, R' can simulate the cells used by R with $O(P(n)T(n))$ memory pairs and triples. R' uses memories mem_6 and mem_7 for communication among the processors.

Let P_m denote processor m of R; let P'_m denote processor m of R'.

We now describe the operation of R'. In $O(\log P(n))$ steps, R' activates $P(n)$ processors, called *primary processors*. In the next $\log(P(n)T(n))$ steps, each primary processor activates $P(n)T(n)$ *secondary processors*, each of which corresponds to a memory pair and a memory triple. Primary processor P'_m corresponds to the processor of R numbered $(m/P(n)T(n)) - P(n)$. The processors numbered $m+k$, for

all k, $0 \leqslant k \leqslant P(n)T(n)-1$, are the secondary processors belonging to primary processor P'_m. Each secondary processor P'_j belonging to P'_m, $j=m+k$, handles *pair*(k) and *triple*(k). We call k the *assignment number* of P'_j. P'_j computes its assignment number in constant time.

Observe that if $i<m$ and P'_i and P'_m are primary processors, then the processor of R to which P'_i corresponds is numbered lower than the processor of R to which P'_m corresponds, and all secondary processors belonging to P'_i are numbered lower than all secondary processors belonging to P'_m. We exploit this ordering to handle concurrent writes by processors in R.

Suppose R' is simulating step t of R in which P_g writes v in $c(f)$. Then the corresponding primary processor P'_m of R' writes $f+1$ into $c_1(P(n)(T(n)-t)+g+1)$ and v into $c_2(P(n)(T(n)-t)+g+1)$. That is, at step t of R, all primary processors of R' write only cells with addresses in $P(n)(T(n)-t)+1, \ldots, P(n)(T(n)-t+1)$, with the lowest-numbered primary processor writing in the lowest-numbered cell in the block. The memory holds a copy every time a processor attempts to write $c(f)$. By this ordering, the copy of a cell in R with the current contents (most recently written by lowest-numbered processor) is in a lower-numbered cell of mem_2 of R' than each of the other copies. The secondary processor that handles this current copy is lower-numbered than each of the secondary processors handling other copies. If at some later step a primary processor P'_m desires to read $con(f)$ of R, then its secondary processors read all copies of $con(f)$ and concurrently write their values in $c_7(m)$. By the write priority rules in which the lowest-numbered processor of those simultaneously attempting to write a cell succeeds, the secondary processor reading the current value of $con(f)$ succeeds in the write.

Similarly, suppose R' is simulating a step of R in which P_g writes v in $r_g(f)$. Then P'_m writes g in $c_3(P(n)(T(n)-t)+g+1)$, $f+1$ in $c_4(P(n)(T(n)-t)+g+1)$, and v in $c_5(P(n)(T(n)-t)+g+1)$. If at some later step P'_m desires to read $rcon_g(f)$, then its secondary processors read all copies of $rcon_g(f)$ and concurrently write their values in $c_7(m)$.

When a processor P_g of R executes an instruction $r(i) \leftarrow r(j) \bigcirc r(k)$, it reads $rcon_g(j)$ and $rcon_g(k)$, computes $v := rcon_g(j) \bigcirc rcon_g(k)$, and writes v in $r_g(i)$. The corresponding processor P'_m of R' simulates this step as follows. Using mem_6 and mem_7 to communicate with its secondary processors and exploiting the write priority rules, P'_m copies $rcon_g(j)$ of R to $r_m(1)$ and $rcon_g(k)$ of R to $r_m(2)$. P'_m then computes $v := rcon_m(1) \bigcirc rcon_m(2)$, writing v in $r_m(1)$. Next, if i is negative, then P'_m does nothing. Otherwise, suppose R' is simulating step t of R. Each primary processor keeps track of t in its local memory. Then P'_m writes g in $c_3(P(n)(T(n)-t)+g+1)$, $i+1$ in $c_4(P(n)(T(n)-t)+g+1)$, and v in $c_5(P(n)(T(n)-t)+g+1)$ to complete the simulation of step t.

Thus, R' uses a constant number of steps to simulate a step of R and only $O(\log P(n)T(n))$ initialization time. Since $P(n) \leqslant 2^{T(n)}$, R' uses $O(T(n))$ steps to simulate $T(n)$ steps of R. □

Observation 1: R' needs only addition and subtraction to construct any address that it uses.

Observation 2: Each processor of R' uses only a constant number of local registers.

Hagerup [11] proved a result for the same problem as the Associative Memory Lemma, but he fixes the number of processors and lets the time grow. Let $S(n)$ denote the highest-numbered memory cell used in a computation of PRAM[*op*] R, and let $B(n)$ be any function such that $B(n) \geq 2$ and is computable with the resources given below. Specifically, Hagerup proved that for all $T(n), P(n)$, and $S(n)$, every language recognized with $P(n)$ processors in time $T(n)$ with memory cells numbered at most $S(n)$ by a PRAM[*op*] R can be recognized in time $O(\lceil \log(S(n)+1)/\log B(n) \rceil T(n))$ by a PRAM[*op*] R' that uses $P(n)$ processors and accesses only cells with addresses in $0, \ldots, O(P(n)T(n)B(n))$.

2.2. *Circuit definitions*

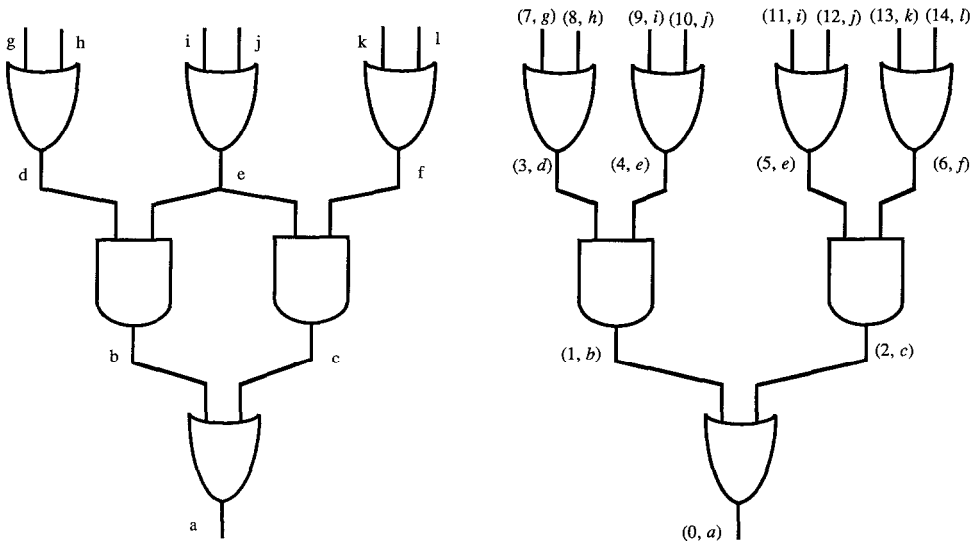
We use the following definitions relating to circuits [19].

- A *circuit* is a directed acyclic graph, where each node (gate) with indegree 0 is labeled by "*inp*" (an *input*), each node with indegree 1 is labeled by the NOT of a variable, and each node with indegree $d > 1$ is labeled by the AND or OR of d variables. Nodes with outdegree 0 are *outputs*.
- A *circuit family* C is a set $\{C_1, C_2, \ldots\}$ of circuits, where C_n has n inputs and one output. We restrict the gate numbering so that the largest gate number is $(Z(n))^{O(1)}$, where $Z(n)$ is the size of C_n. Thus, a gate number coded in binary has length $O(\log Z(n))$.
- A *bounded fan-in circuit* is a circuit where the indegree of all gates is at most 2. For each gate g in C_n, let $g(\lambda)$ denote g, $g(\mathrm{L})$ denote the left input to g, and $g(\mathrm{R})$ denote the right input to g. If C_n has at most $Z(n)$ gates and depth $D(n)$, then the *size complexity* of C is $Z(n)$ and the *depth complexity* is $D(n)$.
- An *unbounded fan-in circuit* is a circuit where indegree is unbounded. For each gate g in C_n, let $g(\lambda)$ denote g, and let $g(p)$, $p = 0, 1, 2, \ldots$, denote the pth input to g. If C_n has at most $Z(n)$ wires and depth $D(n)$, then the *size complexity* of C is $Z(n)$ and the *depth complexity* is $D(n)$.
- The family C *recognizes* $A \subseteq \{0,1\}^*$ if, for each n, C_n recognizes $A^{(n)} = A \cap \{0,1\}^n$; that is, the value of C_n on input $inp_1, inp_2, \ldots, inp_n \in \{0,1\}$ is 1 if and only if $inp_1 \ldots inp_n \in A$. A *language* Q is of simultaneous size and depth complexity $Z(n)$ and $D(n)$ if there is a family of circuits of size complexity $Z(n)$ and depth complexity $D(n)$ that recognizes Q.
- The *bounded direct connection language* of the family $C = \{C_1, C_2, \ldots\}$, L_{BDC}, is the set of strings of the form $\langle n, g, p, h \rangle$, where $n, g \in \{0,1\}^*$, $p \in \{\lambda, L, R\}$, $h \in \{inp, \mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\} \cup \{0,1\}^*$ such that in C_n either (i) $p = \lambda$ and gate g is an h-gate, $h \in \{inp, \mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$, or (ii) $p \neq \lambda$ and gate $g(p)$ is numbered h, $h \in \{0,1\}^*$.

- The *unbounded direct connection language* of the family $C=\{C_1, C_2, \ldots\}$, L_{UDC}, is the set of strings of the form $\langle n, g, p, h\rangle$, where $n, g \in \{0,1\}^*$, $p \in \{\lambda\} \cup \{0, 1, \ldots, Z(n)^{O(1)}\}$, $h \in \{inp, \text{AND}, \text{OR}, \text{NOT}\} \cup \{0,1\}^*$ such that in C_n either (i) $p=\lambda$ and gate g is an h-gate, $h \in \{inp, \text{AND}, \text{OR}, \text{NOT}\}$, or (ii) $p \neq \lambda$ and gate $g(p)$ is numbered h, $h \in \{0,1\}^*$.

Let us now introduce two new definitions of uniformity. Let $I(Z(n))$ be a concatenation of all pairs (g, h), where $g, h \in \{0, 1, \ldots, Z(n)^{O(1)}\}$.

- The family $C=\{C_1, C_2, \ldots\}$ of bounded (unbounded) fan-in circuits of size $Z(n)$ is *VM-uniform* if there is a RAM$[\uparrow, \downarrow]$ that on input $I(Z(n))$ returns an output string in $O(\log Z(n))$ time indicating for each pair (g, h) whether $\langle n, g, L, h\rangle$ is in L_{BDC} and whether $\langle n, g, R, h\rangle$ is in L_{BDC} (indicating for each pair (g, h) the value of p such that $\langle n, g, p, h\rangle$ is in L_{UDC}, for $p=0, 1, \ldots, Z(n)^{O(1)}$, or an indication that no $\langle n, g, p, h\rangle$ is in L_{UDC}). (*Note*: We chose the term VM-uniform because Pratt and Stockmeyer [15] called their restricted RAM$[\uparrow, \downarrow]$ a vector machine.)
- The family $C=\{C_1, C_2, \ldots\}$ of bounded (unbounded) fan-in circuits of size $Z(n)$ is *MRAM-uniform* if there is a RAM$[*]$ that an input $I(Z(n))$ returns an output string in $O(\log Z(n))$ time indicating for each pair (g, h) whether $\langle n, g, L, h\rangle$ or $\langle n, g, R, h\rangle$ is in L_{BDC} (indicating for each pair (g, h) the value of p such that $\langle n, g, p, h\rangle$ is in L_{UDC}, for $p=0, 1, \ldots, Z(n)^{O(1)}$, or an indication that no $\langle n, g, p, h\rangle$ is in L_{UDC}). (*Note*: We chose the term MRAM-uniform because Hartmanis and Simon [12] called their RAM$[*]$ an MRAM.)
- A gate g is at *level* j of C_n if the longest path from any circuit input to g has length j. Gate g is at *height* j of C_n if the longest path from g to the output has length j.
- Let C_n be a bounded fan-in circuit consisting entirely of AND, OR, and *inp* gates with depth $D(n)$. We construct the circuit $CT(C_n)$, the *circuit tree* of C_n, from C_n. Let gate a be the output gate of C_n and let a be of type $\phi \in \{\text{AND}, \text{OR}\}$ with inputs from gates b and c. Then the output gate of $CT(C_n)$ has name $(0, a)$, type ϕ, and inputs from gates named $(1, b)$ and $(2, c)$. Thus, gate $(0, a)$ is the gate at height 0 of $CT(C_n)$ and gates $(1, b)$ and $(2, c)$ are the gates at height 1 of $CT(C_n)$. Now suppose that we have constructed all gates at height j of $CT(C_n)$, and we wish to construct the gates at height $j+1$. Each gate (i, e) at height j corresponds to a gate e in C_n. If e is of type $\phi \in \{\text{AND}, \text{OR}\}$, then gate (i, e) is of type ϕ. Suppose gate e has inputs from gates f and g. Then the inputs to gate (i, e) of $CT(C_n)$ at height $j+1$ are the gates $(2i+1, f)$ and $(2i+2, g)$. If gate e is of type *inp*, that is, an input, and $j < D(n)$, then (i, e) is of type OR (if (i, e) is at an even-numbered level) or type AND (if (i, e) is at an odd-numbered level), and the inputs to gate (i, e) at height $j+1$ are the gates $(2i+1, e)$ and $(2i+2, e)$. If gate e is of type *inp* and $j = D(n)$, then (i, e) is of type *inp* and $CT(C_n)$ has no gates at height $j+1$ connected to gate (i, e). Figure 1 contains an example of a circuit tree.
- In $CT(C_n)$, define path (a, b) to be the path, if one exists, from gate a to gate b.
- In $CT(C_n)$, the *distance* from gate b to gate c is the length of the shortest path from b to c, if such a path exists. We order all gates at distance d from gate b according to the relation *order*(e, f) such that *order*(e, f) is true if *path*(e, b) intersects *path*(f, b)



(a) Circuit C_6 (b) Circuit tree $CT(C_6)$

Fig. 1. Circuit tree example.

at a gate which $path(e, b)$ enters at the left input and $path(f, b)$ enters at the right input. Gate e is the qth *ancestor of gate b at distance d* if gate e is the qth smallest gate in the ordering of the gates at distance d. We say that the smallest gate at distance d is the 0th ancestor. In Fig. 1, gate $(10, j)$ is the 3rd ancestor of gate $(0, a)$ at distance 3. Note that the same gate in C_n can correspond to several ancestors of a gate at distance d in $CT(C_n)$.

- A *double-rail circuit* is a bounded fan-in circuit that is given as input $inp_1, inp_2, \ldots, inp_n$ and their complements $\overline{inp}_1, \overline{inp}_2, \ldots, \overline{inp}_n$ and that contains no NOT-gates. Note that in a double-rail circuit every gate has exactly two inputs, except the input gates.
- A *layered circuit* is a double-rail circuit such that all gates at level i, for all odd i, are AND-gates and all gates at level i, for all even i, are OR-gates, and each input to a gate at level i is connected to an output of a gate at level $i-1$.

Lemma 2.1 (Trahan [25]). *Let* $C=\{C_1, C_2, \ldots\}$ *be a VM-uniform* (*MRAM-uniform*) *family of bounded fan-in circuits of size* $Z(n)$ *and depth* $D(n)$ *recognizing language L. There exists a family of VM-uniform* (*MRAM-uniform*), *bounded fan-in, layered circuits* $F=\{F_1, F_2, \ldots\}$ *of size* $O(Z(n))$ *and depth* $O(D(n))$ *recognizing language L.*

3. Simulation of uniform circuit by RAM[*op*]

In this section, we restructure a uniform, bounded fan-in circuit, then simulate the restructured circuit on a RAM[*op*]. We first describe the simulation of a VM-uniform

circuit on a RAM[$\uparrow,\downarrow$], then modify the algorithm to simulate an MRAM-uniform circuit on a RAM[$*$]. The RAM[op] operates on a circuit tree rather than the original circuit because the RAM[op] can very easily run a circuit tree, once the input bits are properly placed. The RAM[op] also partitions the circuit into slices of depth $O(\log Z(n))$ to balance the time spent generating the circuit with the time spent running the circuit.

3.1. *Simulation of VM-uniform circuit by* RAM[$\uparrow,\downarrow$]

Let $C=\{C_1, C_2, \ldots\}$ be a VM-uniform family of bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. We now describe how a RAM[$\uparrow,\downarrow$] can simulate C.

Simulation. Fix an input length n. Circuit C_n has size $Z(n)$ and depth $D(n)$. We construct a RAM[$\uparrow,\downarrow$] R that recognizes $L^{(n)}$ in time $O(D(n)+\log Z(n)\log\log Z(n))$.

By Lemma 2.1, there exists a VM-uniform layered circuit F_n with size $O(Z(n))$ and depth $O(D(n))$ that recognizes language $L^{(n)}$. Machine R simulates C_n via F_n. For simplicity, let us say that F_n has depth $D(n)$ and size $Z(n)$, and that all gates of F_n are numbered from $\{0, 1, \ldots, Z(n)-1\}$.

Let us first outline the simulation.

Stage 1. R generates a $Z(n)\times Z(n)$ ancestor matrix A in which each entry (g,h) indicates whether gate h is an input of gate g in F_n.

Stage 2. R obtains matrix $A_{\log Z(n)}$, the distance $\log Z(n)$ ancestor matrix.

Stage 3. R extracts from $A_{\log Z(n)}$ a description of the circuit in slices of depth $O(\log Z(n))$ and their circuit trees.

Stage 4. R runs each slice consecutively on the input.

Stage 1: Computation of ancestor matrix. Each entry in ancestor matrix A is a bit vector $Z(n)$ bits long. Entry (g,h) has a 1 in bit position 0 (1) if gate h is the left (right) input to gate g; otherwise, bit position 0 (1) holds 0. All other bit positions hold 0.

R first writes all pairs (g,h), where $g,h\in\{0, 1, \ldots, Z(n)-1\}$, concatenated in a single register in $O(\log Z(n))$ time. We view a register as the concatenation of $Z^2(n)$ *slots*, each slot $Z(n)$ bit positions long. Pair (g,h) is written in slot $gZ(n)+h$ with the least significant bit of $\#g$ in the 0th bit position in the slot and the least significant bit of $\#h$ in the $Z(n)/2$th bit position in the slot. R constructs the first component of every pair one bit position at a time, then the second component of every pair one bit position at a time.

We now describe how R builds the first component of each pair; R builds the second component similarly. We build the first component as a bit vector with $\#g$ in each of slots $gZ(n), \ldots, (g+1)Z(n)-1$, for each g, $0\leqslant g\leqslant Z(n)-1$. Let v denote this bit vector.

R first constructs a bit vector ξ in which each slot $gZ(n)$, $0\leqslant g\leqslant Z(n)-1$, holds $\#g$. Let $q=\log Z(n)$. Each g is q bits long. R constructs ξ in q phases, generating one bit position at a time for all g. Let S_i denote the bit vector where $\#S_i$ equals $\#\xi$ in the ith

bit position of each slot, and is 0's elsewhere. Thus, $\xi = \bigvee_{i=0}^{q-i} S_i$. In phase i, R constructs S_{q-i}, using shifts and Boolean operations on previously constructed S_j's. This takes $O(q)$ time in phase 1 and $O(1)$ time in every other phase.

R next builds υ from ξ by filling in the empty slots in $\log Z(n)$ phases. In this manner, R builds the first component of each pair in $O(\log Z(n))$ time.

R builds the second component of each pair similarly. R now has a register containing the concatenation of all pairs (g, h), $0 \leqslant g, h \leqslant Z(n)-1$. For each pair (g, h), R determines simultaneously whether gate h is an input to gate g. If gate h is the left (right) input to gate g, then R writes a 1 in the first (second) position in the slot. By VM-uniformity, this process takes $O(\log Z(n))$ time. The resulting bit vector constitutes the ancestor matrix A.

Stage 2: *Computation of distance* $\log Z(n)$ *ancestor matrix.* Pratt and Stockmeyer [15] proved that given two $z \times z$ Boolean matrices A and B, a RAM$[\uparrow, \downarrow]$ can compute their Boolean product G, defined by

$$G(i, j) = \bigvee_k (A(i, k) \wedge B(k, j)),$$

in $O(\log z)$ time. The RAM$[\uparrow, \downarrow]$ performs the AND of all triples i, j, k in one step and the OR in $O(\log z)$ steps. Let θ_d be a function, specified below, with two bit vectors as inputs and one bit vector as output. Given two $z \times z$ matrices A and B whose elements are bit vectors m bits long, let us define the function $H_d(A, B) = G$, where

$$G(i, j) = \bigvee_k \theta_d(A(i, k), B(k, j)).$$

We prove that a RAM$[\uparrow, \downarrow]$ can compute the matrix $G = H_d(A, B)$ in $O(\log z + \log m)$ time. The RAM$[\uparrow, \downarrow]$ performs θ_d on all triples i, j, k in $O(\log z + \log m)$ steps and the OR in $O(\log z)$ steps. We prove that a RAM$[\uparrow, \downarrow]$ can compute $\theta_d(A(i, k), B(k, j))$ in $O(\log m)$ time to establish this bound. In our case, $m = z = Z(n)$.

Suppose R has operated $\log d$ times on A. Call the resulting matrix A_d. A 1 in bit position i of $A_d(f, g)$ indicates that gate g is the ith ancestor of gate f at distance d. We want $\theta_d(A_d(f, g), A_d(g, h))$ to return a bit vector $A_{2d}(f, h)$ with a 1 in bit position i if gate h is the ith ancestor of gate f at distance $2d$, and 0 in bit position i otherwise. After $\log \log Z(n)$ operations, $A_{\log Z(n)}(g, h)$ holds 1 in each bit position j if gate h is the jth ancestor of gate g at distance $\log Z(n)$, 0 otherwise.

Without loss of generality, assume that m is a power of 2. Let $x = 2^d$. Let α and β be integers of length x: $\#\alpha = \alpha_{x-1} \ldots \alpha_1 \alpha_0$ and $\#\beta = \beta_{x-1} \ldots \beta_1 \beta_0$. We define the function $\theta_d(\alpha, \beta)$ so that for each 1 in bit position a in $\#\alpha$ and each 1 in bit position b in $\#\beta$, in the result $\gamma = \theta_d(\alpha, \beta)$, $\#\gamma$ has a 1 in position $a \cdot 2^d + b$. If either α or β is 0, then γ is 0.

Before describing how R performs θ_d, we define two functions *SPREAD* and *FILL* that R uses to perform θ_d. The function $SPREAD(\#\alpha, y)$ returns the bit vector $\#\alpha' = \alpha_{x-1} 0 \ldots 0 \alpha_{x-2} 0 \ldots 0 \alpha_1 0 \ldots 0 \alpha_0$ in which α_{i+1} is y bit positions away from α_i, separated by 0's, for all $0 \leqslant i \leqslant x-2$.

Lemma 3.1. *For any y, if $\#\alpha$ is x bits long, then R can perform $SPREAD(\#\alpha, y)$ in time $O(\log x)$.*

Proof. R performs $SPREAD$ in $O(\log x)$ phases of mask and shift operations. Note that the subscript of each bit in $\#\alpha$ specifies its position. In phase i, R masks away all bits of $\#\alpha$ whose subscript has a 0 in the $(\log x - i)$th position. R then shifts the remainder of the string and combines this with the unshifted portion by a Boolean OR. In $O(\log x)$ phases, each taking a constant amount of time, R performs $SPREAD(\#\alpha, y)$. □

Let $\alpha^s = SPREAD(\#\alpha, y)$. The function $FILL(\alpha^s, y)$ returns the value α^f, where $\#\alpha^f = \alpha_{x-1}\alpha_{x-1}\ldots\alpha_{x-1}\alpha_{x-2}\ldots\alpha_1\alpha_1\ldots\alpha_1\alpha_0\alpha_0\ldots\alpha_0$, in which positions $iy,\ldots,(i+1)y-1$ have value α_i. Assume that y is a power of 2. (*Note*: One may think of α^s as α spread out and α^f as α^s filled in.)

Lemma 3.2. *R can perform $FILL(\alpha^s, y)$ in time $O(\log y)$.*

Proof. R performs $FILL(\alpha^s, y)$ in $O(\log y)$ phases of shift and OR operations. Each phase takes constant time, so R performs $FILL(\alpha^s, y) = \alpha^f$ in $O(\log y)$ time. □

Now we describe how a RAM[↑, ↓] R computes $\theta_d(\alpha, \beta)$ in $O(\log x) = O(\log m)$ steps. R first computes $\alpha^s = SPREAD(\#\alpha, 2^d)$ in $O(\log x)$ time. Each 1 in position i in $\#\alpha$ produces a 1 in position $i \cdot 2^d$ of $\#\alpha^s$. R concatenates $x = 2^d$ copies, each m bits long, of $\#\alpha^s$ in $O(\log x)$ time. Let *squid* denote the value of this concatenation. R then computes $\beta^f = FILL(SPREAD(\#\beta, m), m)$ in $O(\log x)$ time, then performs $squid \leftarrow squid \wedge \beta^f$, which blocks out each copy of $\#\alpha^s$ in $\#squid$ that corresponds to a 0 in $\#\beta$.

Next for each nonzero bit β_j of $\#\beta$, R shifts the jth copy of $\#\alpha^s$ to the left by j bits in $O(\log x)$ phases of mask and shift operations. In phase i, R masks away all copies of $\#\alpha^s$ corresponding to nonzero bits β_j for which the $(\log x - i)$th position of $\#j$ is 0, then shifts the remainder of the vector, and combines this with the unshifted portion by an OR.

Let a *block* of size m of $\#\gamma = \gamma_x \ldots \gamma_0$ be a string of bits $\gamma_{(j+1)m-1} \ldots \gamma_{jm}$. Finally, R ORs together all blocks of size m in $O(\log m)$ steps. (Formerly, each block of size m was a copy of $\#\alpha^s$; now some have been shifted.) The resulting bit vector is $\theta_d(\alpha, \beta)$. R has computed $\theta_d(\alpha, \beta)$ in $O(\log m)$ steps.

To compute $\theta_d(A(i,k), B(k,j))$ in $O(\log z + \log m)$ time when the matrices A and B are given as bit vectors, simply allow enough space between elements in the bit vectors A and B so that operations on adjacent pairs do not interfere with each other. We can generate all masks and perform all operations in $O(\log z + \log m)$ time.

Given the $Z(n) \times Z(n)$ ancestor matrix $A_1 = A$ with entries $Z(n)$ bits long, R computes $A_2 = H_1(A_1, A_1)$, then $A_{2j} = H_j(A_j, A_j)$, for each $j = 1, 2, 4, \ldots, \log Z(n) - 1$.

Let $G = A_{\log Z(n)}$. Each H_d operation takes time $O(\log Z(n))$ to compute, and R executes H_d for $\log\log Z(n)$ values of d. Hence, R computes G in $O(\log Z(n) \log\log Z(n))$ time.

Stage 3: *Extraction of slices and circuit trees.* We partition F_n into $D(n)/\log Z(n)$ *slices* of depth $\log Z(n)$ each. Let $v = D(n)/\log Z(n)$. The jth slice comprises levels $j\log Z(n), \ldots, (j+1)\log Z(n)-1$, for $0 \leqslant j \leqslant v-1$. Let $\Sigma(j)$ denote the jth slice.

R extracts circuit tree descriptions of each slice from matrix G, starting with $\Sigma(v-1)$. To extract $CT(\Sigma(i))$, R isolates the portion of matrix G that describes the circuit trees for each output from slice $\Sigma(i)$. We call this matrix a *slice matrix*. Let $S(i)$ denote the slice matrix for $\Sigma(i)$. For each gate g at the output of $\Sigma(i)$, $S(i)$ specifies each ancestor of gate g at the top of $\Sigma(i)$.

Matrix G is stored in a single register in row major order. We view the contents of this register both as a matrix of discrete elements and as a single bit string. We call the portion of a matrix comprising one row a *box*. We call the portion of a box containing one element of a row a *slot*.

We introduce a simple procedure *COLLAPSE*, which R uses to extract information from matrix G. Procedure $COLLAPSE(\alpha, z)$ takes as input the value α, where $\#\alpha = \alpha_{z^2-1} \ldots \alpha_1 \alpha_0$, and returns the value β, where $\#\beta = \beta_{z^2-1} \ldots \beta_1 \beta_0$, and bits $\beta_{kz} = \bigvee_{j=0}^{z-1} \alpha_{kz+j}$, for $0 \leqslant k \leqslant z-1$, and $\beta_i = 0$ if $i \neq kz$. R can perform $COLLAPSE(\alpha, z)$ in $O(\log z)$ time by shifting and ORing, then masking away all bits ψ_i for $i \neq kz$.

Let *out* denote the name of the output gate of F_n. Nonzero entries in row *out* of G correspond to the ancestors of gate *out* at distance $\log Z(n)$; that is, the gates at the boundary between $\Sigma(v-1)$ and $\Sigma(v-2)$. To extract $CT(\Sigma(v-1))$, R masks away all but row *out* of G. Let $S(v-1)$ denote this value.

In general, assume that we have $S(i+1)$, and we want to extract $S(i)$ from G. First, R computes the OR of all boxes of $S(i+1)$. Let $\eta(i)$ denote this value. R computes $\alpha(i) = COLLAPSE(\eta(i), Z(n))$ in $O(\log Z(n))$ time. Bit $jZ^2(n) + kZ(n)$, $0 \leqslant j, k \leqslant Z(n)-1$, of $\#\alpha(i)$ is 0 (1) if slot k of box j of $S(i+1)$ contains all 0's (at least one 1). Let $\alpha_b(i)$ denote bit b of $\#\alpha(i)$. We use $\alpha(i)$ to select the rows of G that correspond to nonzero slots of $S(i+1)$. Next, R computes $\sigma(i) = SPREAD\ (\#\alpha(i), Z(n))$ in $O(\log Z(n))$ steps. This leaves bit $\alpha_{kZ(n)}(i)$ at position $kZ^2(n)$ of $\#\sigma(i)$; that is, it aligns the bit of $\#\alpha(i)$ indicating whether or not slot m contains a 1 with box m of G. Now, R computes $\phi(i) = FILL(\sigma(i), Z^2(n))$, and $\#\phi(i)$ has 1's in the boxes of G that correspond to ancestors of *out* at distance $\log Z(n)$.

R computes $S(i) \leftarrow \phi(i) \wedge G$. Thus, the boxes of G that correspond to slots of $S(i+1)$ that contain all 0's are masked away in $S(i)$. Each nonzero slot of $S(i)$ indicates a gate at the top boundary of $\Sigma(i)$. The extraction of $S(i)$ from G, given $S(i+1)$, takes $O(\log Z(n))$ time. Therefore, R extracts all $D(n)/\log Z(n)$ slice matrices in $O(D(n))$ time.

Stage 4: *Running the slices on the input.* At this point, for $0 \leqslant i \leqslant v-1$, R has computed $S(i)$. Each $S(i)$ contains a description of $CT(\Sigma(i))$. (*Note*: $CT(\Sigma(i))$ is a collection of circuit trees, one for each output of $\Sigma(i)$.) In Stage 4, R runs each $CT(\Sigma(i))$ in sequence. R begins by manipulating the input ω to be in the form

necessary to run on $CT(\Sigma(0))$ by $SPREAD$ $(\#\omega, Z(n))$. The input ω to F_n is $2n$ bits long (n input bits and their complements).

We describe how R runs the circuit by slices. We must take the output $\psi(i)$ from $\Sigma(i)$ and convert it into the form needed for the input to $CT(\Sigma(i+1))$. We let $\omega(i)$ denote this input.

Let us define a function $COMPRESS$, the inverse of $SPREAD$. The function $COMPRESS(\beta, y)$, where $\#\beta = \beta_{xy-1} \ldots \beta_1 \beta_0$, returns the value α, where $\#\alpha = \beta_{(x-1)y}\beta_{(x-2)y} \ldots \beta_y \beta_0$, in $O(\log x)$ time.

In general, we have $S(i)$ and $\psi(i-1)$, and we want to compute $\psi(i)$. The output $\psi(i-1)$ from a slice $\Sigma(i-1)$ is in the form of isolated bits, one for each box corresponding to an output from $\Sigma(i-1)$. R computes $\mu(i-1) = COMPRESS(\psi(i-1), Z(n))$, then builds $\psi^f(i-1) = FILL(\mu(i-1), Z(n))$ in $O(\log Z(n))$ time. The result $\mu(i-1)$ has $Z(n)$ output bits from $\psi(i-1)$. These are $Z(n)$ bits apart, one per slot in a single box. Now R concatenates $Z(n)$ copies of $\psi^f(i-1)$; call this $\psi^c(i-1)$. Each element is $Z^2(n)$ bits long, the length of a box. R computes $S(i)' = S(i) \wedge \psi^c(i-1)$; hence, $\#S(i)'$ has a 1 in position $jZ^2(n) + kZ(n) + l$ if gate j is at the bottom of $\Sigma(i)$, gate k is the lth ancestor of j at the top of $\Sigma(i)$, and the input to gate k is a 1. R ORs all slots in each box together in $O(\log Z(n))$ steps, producing bit vector $\omega(i)$. By our construction, $\omega(i)$ is the input to $CT(\Sigma(i))$. Recall that $CT(\Sigma(i))$ consists of alternating layers of AND and OR gates. We run $CT(\Sigma(i))$ on input $\omega(i)$ in $O(\log Z(n))$ steps by shifting, then ANDing and ORing.

It takes time $O(\log Z(n))$ to manipulate the output from one slice into the form needed for the input to the next slice and time $O(\log Z(n))$ to run a slice. Since there are $D(n)/\log Z(n)$ slices, its takes time $O(D(n))$ to run a circuit on the input, given the distance $\log Z(n)$ ancestor matrix G.

Theorem 3.3. *Let $C = \{C_1, C_2, \ldots\}$ be a family of VM-uniform, bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. There exists a* RAM$[\uparrow, \downarrow]$ *R that recognizes L in time $O(D(n) + \log Z(n) \log\log Z(n))$.*

Proof. We construct R by the method described above. For fixed n, R simulates C_n via F_n in $O(\log Z(n) \log\log Z(n))$ time to create matrix G, then $O(D(n))$ time to run F_n on input ω, given G. Thus, the overall time is $O(D(n) + \log Z(n) \log\log Z(n))$ steps. □

3.2. *Simulation of MRAM-uniform circuit by* RAM$[*]$

In this section, we adapt the simulation of a VM-uniform circuit by a RAM$[\uparrow, \downarrow]$ to the case of a simulation of an MRAM-uniform circuit by a RAM$[*]$.

Theorem 3.4 *Let $MC = \{MC_1, MC_2, \ldots\}$ be a family of MRAM-uniform, bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. There exists a* RAM$[*]$ *R that recognizes L in time $O(D(n) + \log Z(n) \log\log Z(n))$.*

Proof. Without loss of generality, assume that R has two memories: mem_1 and mem_2. R performs the simulation described in Section 3.1, using a precomputed table of shift values in mem_2. To perform a left shift, such as $temp' \leftarrow temp \uparrow j$, R performs $temp' \leftarrow temp \cdot 2^j$. To perform a right shift by j bits, R shifts all other values in mem_1 left by j bits, then notes that the rightmost j bits of all registers are to be ignored [12]. This takes constant time because, by reusing registers, R uses only a constant number of registers in mem_1. In $O(\log Z(n))$ time, R computes the values $2^{Z(n)}$ and $2^{Z^2(n)}$, since $Z(n)$ and $Z^2(n)$ are the basic shift distances. In the course of the computation, R performs shifts by $Z(n) \cdot 2^i$, $0 \leqslant i \leqslant \log Z(n)$, for each value of i. R computes the necessary shift value on each iteration from the previous value.

Thus, the simulation by R takes the same amount of time as the simulation described in Section 3.1: $O(D(n) + \log Z(n) \log\log Z(n))$. □

4. Multiplication

In this section, we simulate a time-bounded PRAM[$*$] by four different models of computation: basic PRAM, bounded fan-in circuit, Turing machine, and RAM[$*$]. We establish that polynomial time on a PRAM[$*$], RAM[$*$], or a PRAM, polynomial depth on a bounded fan-in circuit, and polynomial space on a TM are all equivalent.

4.1. Simulation of PRAM[$*$] *by PRAM*

Let R be a PRAM[$*$] operating in time $T(n)$ on inputs of length n and using at most $P(n)$ processors. Let R' be a PRAM[$*$] that uses only short addresses and simulates R according to the Associative Memory Lemma. Thus, R' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and only addresses in $0, 1, \ldots, O(P(n)T(n))$. Each processor of R' uses only q registers, where q is a constant.

We construct a PRAM Z that simulates R via R' in $O(T^2(n)/\log T(n))$ time, using $O(P^2(n)T^2(n)n^2 4^{T(n)} \log T(n))$ processors. We view Z as having $q+4$ separate shared memories: $mem_0, \ldots, mem_{q+3}$. Our view facilitates description of the algorithm to follow. The idea of the proof is that Z stores the cell contents of R' with one bit per cell and acts as an unbounded fan-in circuit to manipulate the bits.

Initialization. Z partitions mem_q into $O(P(n)T(n))$ *sections* of $n2^{T(n)}$ cells each. Let $S(i)$ denote the ith section. A section is sufficiently long to hold any number generated in $T(n)$ steps by R', one bit per cell, in $n2^{T(n)}$-bit two's complement representation. Section $S(i)$ contains $con(i)$ of R' with one bit of $con(i)$ in each of the first $len(con(i))$ cells of the section. R' writes the more significant bits in cells with larger addresses.

Z partitions each of $mem_0, \ldots, mem_{q-1}$ into $O(P^2(n)T(n))$ *blocks* of $n2^{T(n)}T(n)\log T(n)$ sections each. Let $B_i(m)$ denote the mth block of mem_i. A block is large enough to implement the multiplication algorithm of Schönhage and Strassen

[20]. The first section of $B_i(m)$ contains $rcon_m(i)$ of R' with one bit of $rcon_m(i)$ in each of the first $len(rcon_m(i))$ cells of the section.

Z activates $O(P^2(n)T(n))$ *primary processors*, one for each processor of R', in time $O(\log P(n)T(n))$. Z must quickly access individual cells in each block, so each primary processor activates $O(n^2 4^{T(n)} T(n) \log T(n))$ *secondary processors* in $O(T(n))$ time. For primary processor P_m, secondary processor P_j, $j \in \{m, \ldots, m+n2^{T(n)}-1\}$, assigns itself to the $(j-m)$th cell of the first section of a block. These processors handle comparisons.

Z next spreads the input integer over the first n cells of $S(0)$ of mem_q, that is, Z places the jth bit of the input word in the jth cell of $S(0)$. This process takes constant time for processors $P_0, \ldots, P_{n-1}$, each performing the BIT instruction indexed by their processor number. (Note that without the $r(k) \leftarrow BIT(r(i))$ instruction, where $rcon_m(i)=j$, this process would take time $O(n)$. If $T(n)=o(n)$, then $O(n)$ time is unacceptably high.)

Simulation. We are now prepared to describe the simulation by Z of a general step of R'. Consider a processor P_g of R' and the corresponding primary processor P_m of Z. The actions of P_m and its secondary processors depend on the instruction executed by P_g of R'. P_m notifies its secondary processors of the instruction. The following cases arise.

$r(i) \leftarrow r(j)+r(k)$: Chandra et al. [3] gave an unbounded fan-in circuit of size $O(x(\log^* x)^2)$ and constant depth for adding two integers of length x. Stockmeyer and Vishkin [24] proved that an unbounded fan-in circuit of depth $D(n)$ and size $S(n)$ can be simulated by a CRCW PRAM in time $O(D(n))$ with $O(n+S(n))$ processors. By the combination of these two results, the secondary processors perform addition in constant time with their concurrent write ability. This addition requires $O(n2^{T(n)}(\log^*(n2^{T(n)}))^2)$ processors.

$r(i) \leftarrow r(j) \wedge r(k)$: The secondary processors perform a Boolean AND in one step. Other Boolean operations are performed analogously.

$r(i) \leftarrow r(j)-r(k)$: The secondary processors add $rcon_g(j)$ and the two's complement of $rcon_g(k)$. This takes constant time.

Comparisons ($CJUMP\ r(i)>r(j), label$): For $1 \leqslant k \leqslant n2^{T(n)}$, the secondary processor of the first section that normally handles the kth cell of the section handles the $(n2^{T(n)}-k+1)$th cell. Thus, the lowest-numbered processor reads the most significant bit. Each secondary processor allocated to the first section compares corresponding bits of $B_i(g)$ and $B_j(g)$, then writes the outcome of the comparison only if the bits differ. By the CRCW priority rule, after the secondary processors write concurrently, the value written corresponds to the most significant bits at which the operands differ. Thus, the outcome of $rcon_g(i)>rcon_g(j)$ is determined by employing the concurrent write rules of the PRAM. Other comparisons are performed analogously and all comparisons can be simulated in constant time.

$r(i) \leftarrow r(j) * r(k)$: We use the following lemma.

Lemma 4.1. *A logspace-uniform, unbounded fan-in circuit of depth* $O(\log y/\log\log y)$ *and size* $O(y^2 \log y \log\log y)$ *can compute the product of two operands of length* y.

Proof. For inputs of length y, Schönhage and Strassen [20] gave a multiplication algorithm that may be implemented as a logspace-uniform bounded fan-in circuit with depth $O(\log y)$ and size $O(y \log y \log\log y)$. Chandra et al. [4] proved that for any $\varepsilon > 0$, a bounded fan-in circuit of depth $D(y)$ and size $S(y)$ can be simulated by an unbounded fan-in circuit of depth $O(D(y)/\varepsilon \log\log y)$ and size $O(2^{(\log y)^{\varepsilon}} \cdot S(y))$. If the bounded fan-in circuit is logspace-uniform, then the unbounded fan-in circuit is also logspace-uniform. Setting $\varepsilon = 1$, we establish the lemma. □

R' can generate numbers of length up to $n2^{T(n)}$. By Lemma 4.1 (setting $y = n2^{T(n)}$) and Stockmeyer and Vishkin's [24] simulation of an unbounded fan-in circuit by a PRAM, a CRCW PRAM can simulate a bounded fan-in circuit performing multiplication in time $O(T(n)/\log T(n))$ with $O(n^2 4^{T(n)} T(n) \log T(n))$ processors.

Indirect addressing: Let us now describe a procedure *SQUASH*. A primary processor and its secondary processors perform a *SQUASH* on a set of s cells by placing in a single cell the integer whose two's complement representation is stored in the set of cells with one bit per cell. The processors execute *SQUASH* in $O(\log s)$ time. By the Associative Memory Lemma, R' accesses only addresses of length $O(\log P(n) T(n))$. If P_g wishes to perform an indirect read from $c(r(i))$, then P_m and its associated processors perform a *SQUASH* on $B_i(g)$ in time $O(\log\log P(n) T(n))$.

If processors P_f and P_g of R' simultaneously attempt to write $c(j)$, then the corresponding processors P_l and P_m of Z simultaneously attempt to write $S(j)$ of mem_q. If $f < g$, then $l < m$, and all secondary processors of P_l are numbered less than all secondary processors of P_m. Thus, in R', P_f succeeds in its write, and in Z, P_l and its secondary processors succeed in their writes.

Theorem 4.2. *For all* $T(n) \geqslant \log n$,

$$\text{PRAM}[*]\text{-TIME}(T(n)) \subseteq \text{PRAM-TIME}(T^2(n)/\log T(n)).$$

Proof. According to the above discussion, Z simulates R via R'. Initialization takes $O(\log(P(n)T(n)) + T(n) + \log n) = O(T(n))$ time. Z performs indirect addressing in $O(\log T(n))$ time, multiplication in $O(T(n)/\log T(n))$ time, and all other operations in constant time. Thus, Z uses time $O(T(n)/\log T(n))$ to simulate each step of R'. Z uses $O(P^2(n)T(n))$ primary processors, each with $O(n^2 4^{T(n)} T(n) \log T(n))$ secondary processors. Hence, Z simulates R in $O(T^2(n)/\log T(n))$ time, using $O(P^2(n) T^2(n) n^2 4^{T(n)} \log T(n))$ processors. □

If $T(n) = O(\log n)$, then $P(n)$ is a polynomial in n, and Z simulates R in time $O(\log^2 n/\log\log n)$ with polynomially many processors. Thus, an algorithm running in time $O(\log n)$ on a PRAM[$*$] is in NC^2. If $T(n) = O(\log^k n)$, then Z simulates R in time $O(\log^{2k} n/(2k \log\log n))$ with $O(n^{2+4\log^{k-1} n} \cdot \log^{2k} n \log\log n)$ processors. So, our simulation does *not* show that an algorithm running in time $O(\log^k n), k > 1$, on a PRAM[$*$] is in NC because of the superpolynomial processor count. An interesting open problem is to show either that PRAM[$*$]-POLYLOGTIME = NC by reducing

the processor count to a polynomial or that NC is strictly included in PRAM[$*$]-POLYLOGTIME by proving that the simulation requires a superpolynomial number of processors.

4.2. *Simulations of* PRAM[$*$] *by circuits and Turing machine*

We now describe simulations of a PRAM[$*$] by a logspace-uniform family of unbounded fan-in circuits, a logspace-uniform family of bounded fan-in circuits, and a Turing machine.

Lemma 4.3 (Stockmeyer and Vishkin [24]). *Let Z be a PRAM with time bound $T(n)$, processor bound $P(n)$, and word-length bound $L(n)$. There is an unbounded fan-in circuit C_n that simulates Z in depth $O(T(n))$ and size $O(P(n)T(n)L(n)(L^2(n)+P(n)T(n)))$.*

Note: Minor changes are necessary in the simulation of Stockmeyer and Vishkin to account for differences between their PRAM definition and ours, but these cause no change in the overall depth or size of the simulating circuit. Stockmeyer and Vishkin presented the simulation of a nonuniform PRAM by a nonuniform family of circuits. For our PRAM definition, in which all processors share a constant size program, the simulating circuit is logspace-uniform.

Lemma 4.4. *For each n and $T(n) \geqslant \log n$, every language recognized by a* PRAM[$*$] *R in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform, unbounded fan-in circuit UC_n of depth $O(T^2(n)/\log T(n))$ and size $O(nT^2(n)\, 32^{T(n)}(n^2+T^2(n)))$.*

Proof. The depth bound follows from Theorem 4.2 and Lemma 4.3. We now establish the size bound. Let R' be the PRAM[$*$] described in Theorem 4.2 that simulates R according to the Associative Memory Lemma, using $O(T(n))$ time with $O(P^2(n)T(n))$ processors and word length $O(n2^{T(n)})$. Fix an input length n. Let UC_n be a logspace-uniform, unbounded fan-in circuit that simulates R' by the construction given by Stockmeyer and Vishkin [24] (Lemma 4.3), with one modification. For each time step of R', we add to UC_n a block of depth $O(T(n)/\log T(n))$ and size $O(n^2 4^{T(n)} T(n) \log T(n))$ that handles multiplication (Lemma 4.1). Thus, UC_n has depth $O(T^2(n)/\log T(n))$ and size $O(P(n)T(n)[L(n)(L^2(n)+P(n)T(n))+n^2 4^{T(n)} T(n)\log T(n)]) = O(nT^2(n)\, 32^{T(n)}(n^2+T^2(n)))$, since $P(n) \leqslant 2^{T(n)}$. □

Lemma 4.5. *For each n and $T(n) \geqslant \log n$, every language recognized by a* PRAM[$*$] *R in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform, bounded fan-in circuit BC_n of depth $O(T^2(n))$ and size $O(nT^2(n)\, 32^{T(n)}\, (n^2+T^2(n)))$.*

Proof. Fix an input length n. Let UC_n be the unbounded fan-in circuit described in Lemma 4.4 that simulates R. Except for the circuit blocks implementing

multiplication, the portions of the circuit that simulate a single time step of R' have constant depth and fan-in at most $\max\{O(n2^{T(n)}), O(P^2(n)T^2(n))\} = O(T^2(n)4^{T(n)})$. Hence, these parts of the circuit can be implemented as a logspace-uniform, bounded fan-in circuit of depth $O(T(n))$. The multiplication blocks may be implemented as logspace-uniform, bounded fan-in circuits of depth $O(T(n))$ (Lemma 4.1). Let BC_n be this bounded fan-in implementation of UC_n. Since $P(n) \leqslant 2^{T(n)}$, BC_n simulates each step of R' in depth $O(T(n))$; hence, BC_n simulates R via R' in depth $O(T^2(n))$ and size $O(nT^2(n)\, 32^{T(n)}\, (n^2 + T^2(n)))$. □

Theorem 4.6 *For all* $T(n) \geqslant \log n$, PRAM[*]-TIME$(T(n)) \subseteq$ DSPACE$(T^2(n))$.

Proof. Theorem 4.6 follows from Lemma 4.5 and Borodin's [2] result that a logspace-uniform, bounded fan-in circuit of depth $D(n)$ can be simulated in space $O(D(n))$ on a Turing machine when $D(n) = \Omega(\log n)$. □

4.3. Simulation of PRAM[*] *by* RAM[*]

In this section, we simulate a PRAM[*] by an MRAM-uniform, bounded fan-in circuit family, then simulate this circuit family by a RAM[*]. We also simulate a basic PRAM by a RAM[*].

Let $C = \{C_1, C_2, \ldots\}$ be the family of unbounded fan-in circuits described in Lemma 4.3 that simulates a PRAM that runs in time $T(n)$ with $P(n)$ processors. We construct a family of bounded fan-in circuits $BC' = \{BC'_1, BC'_2, \ldots\}$ from C. The fan-in of any gate in C_n is at most $\max\{O(L(n)), O(P(n)T(n))\} = \max\{O(n + T(n)), O(P(n)T(n))\}$. We replace each gate with fan-in f in C_n by a tree of gates of depth $\log f$ in BC_n. The depth of BC'_n is $O(T(n)(\log P(n)T(n)))$, and the size is $O(P(n)T(n)L(n)(L^2(n) + P(n)T(n)))$, the same size as C_n.

Lemma 4.7. *BC' is MRAM-uniform.*

Proof. We first establish that the unbounded fan-in circuit C is MRAM-uniform, then establish that the bounded fan-in circuit BC' is MRAM-uniform.

Fix a PRAM Y and an input size n. The simulating circuit C_n comprises $T(n)$ identical *time slices*. Each time slice corresponds to a time step of Y. Each time slice comprises $P(n)$ *cartons* of gates, one for each processor, and a *block* of gates, [Update-Common], handling updates to common memory. Each carton comprises 13 *blocks* of gates handling various functions as indicated by their names: [Compute-Operands], [Add], [Sub], [Local-Read], [Common-Read], [=-Compare], [<-Compare], [Compute-Address-of-Result], [Select-Result], [Update-Instruction-Counter], [Local-Change?], [Common-Change?], and [Update-w-Bits-of-Local-Triples]. The size of each time slice of C_n is $O(P(n)L(n)(L^2(n) + P(n)T(n)))$, where $L(n)$ is the word size of Y, and the total size of C_n is $T(n)$ times this amount.

The general form of a gate name is specified in Fig. 2.

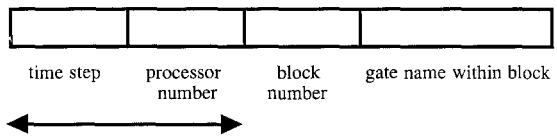


Fig. 2. Gate name in C_n.

Let $Z(n)$ denote the size of C_n. It is clear from the description of the blocks given by Stockmeyer and Vishkin that each block is MRAM-uniform and that the interconnections between blocks are easily computed. Thus, to prove that C is MRAM-uniform, we sketch how a RAM[$*$] R can test the connectivity of all pairs of gates in $O(\log Z(n))$ time.

Let g denote a gate name. Let *slot A* denote the portion of $\#g$ specifying the time step. Let *slot B* denote the portion of $\#g$ specifying the processor number. Let *slot C* denote the portion of $\#g$ specifying the block number. Let *slot D* denote the portion of $\#g$ specifying the gate name within the block. Let $scon_i(g)$ denote the contents of slot i of $\#g$.

The input I is the concatenation of all pairs (g, h), where $g, h \in \{0, 1, \ldots, Z(n)^{O(1)}\}$. To test connectivity, R compares corresponding slots of $\#g$ and $\#h$ for all pairs (g, h) simultaneously. R separates the pairs for which the comparison is true from the pairs for which the comparison is false by building an appropriate mask in time $O(\log Z(n))$. Thus, C is MRAM-uniform.

A gate name in BC'_n is the concatenation of the unbounded fan-in gate name in C_n and the name of the gate within the bounded fan-in tree that replaces the unbounded fan-in gate (Fig. 3).

We prove MRAM-uniformity by the same method as above, with modifications to test slot E, the portion of the gate name giving the gate name within the tree of depth $\log f$. By this algorithm, we see that the family BC' of bounded fan-in circuits is MRAM-uniform since C is MRAM-uniform. □

Theorem 4.8. *For all $T(n) \geq \log n$, every language recognized with $P(n)$ processors in time $T(n)$ by a PRAM can be recognized by a* RAM[$*$] *in time* $O(T(n) \log P(n) T(n))$.

Proof. By Lemma 4.7, BC', the family of bounded fan-in circuits that simulates a PRAM, is MRAM-uniform. By Theorem 3.4, a RAM[$*$] can simulate BC' in time $O(T(n) \log P(n) T(n))$. □

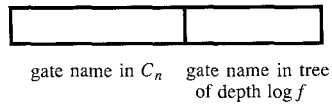


Fig. 3. Gate name in BC'_n.

Let BC denote the family of bounded fan-in circuits described in Lemma 4.5 that simulates a PRAM[$*$] in depth $O(T^2(n))$ and size $O(nT^2(n)\,32^{T(n)}(n^2+T^2(n)))$. We construct a family of bounded fan-in circuits MC from BC. Fix an input size n. The circuit MC_n is exactly the same as the circuit BC_n except that MC_n uses a different multiplication block for reasons of MRAM-uniformity. Insert bounded fan-in circuits performing carry–save multiplication in MC_n. Each block has depth $O(T(n))$ and size $O(n^2 4^{T(n)})$. Thus, MC_n has depth $O(T^2(n))$ and size $O(nT^2(n)32^{T(n)}(n^2+T^2(n)))$.

Lemma 4.9. *For each n, every language recognized by a* PRAM[$*$] R *in time $T(n)$ with $P(n)$ processors can be recognized by bounded fan-in, MRAM-uniform circuit MC_n of depth $O(T^2(n))$ and size $O(nT^2(n)32^{T(n)}(n^2+T^2(n)))$.*

Proof. The proof of the PRAM[$*$] simulation is similar to that given for Lemma 4.5. By an argument similar to the proof of Lemma 4.7, MC is MRAM-uniform. □

Theorem 4.10. *For all $T(n)\geqslant \log n$,*

$$\text{PRAM}[*]\text{-TIME}(T(n)) \subseteq \text{RAM}[*]\text{-TIME}(T^2(n)).$$

Proof. By Lemma 4.9, a PRAM[$*$] running in time $T(n)$ with $P(n)$ processors can be simulated by a bounded fan-in, MRAM-uniform circuit MC_n of depth $O(T^2(n))$ and size $O(nT^2(n)\,32^{T(n)}(n^2+T^2(n)))$. By Theorem 3.4, a RAM[$*$] can simulate MC_n in time $O(T^2(n))$. □

Combining Theorem 4.8 with the simulation of a PRAM[$*$] by a basic PRAM,

$$\text{PRAM}[*]\text{-TIME}(T(n)) \subseteq \text{PRAM-TIME}(T^2(n)/\log T(n)),$$

yields

$$\text{PRAM}[*]\text{-TIME}(T(n)) \subseteq \text{RAM}[*]\text{-TIME}(T^3(n)/\log T(n)).$$

The simulation of Theorem 4.10 is more efficient.

5. Division

In this section, we study the division instruction. We are interested in the division instruction for two reasons. First, division is a natural arithmetic operation. Second, Simon [23] has shown that a RAM with division and left shift ($\uparrow$) can be very powerful. He proved that RAM[$\uparrow, \div$]-PTIME = ER, where ER is the class of languages accepted in time

$$2^{2^{2^{\cdots^{2}}}}\Big\uparrow n$$

by Turing machines. Simon proved ER $\subseteq$ RAM[$\uparrow, \div$]-PTIME by building very long integers with the left-shift operation and then manipulating them as both integers and

binary strings. Division is used to generate a complex set of strings representing all possible TM configurations. (Note that right shift cannot replace division in building these strings.) At first glance, this is a surprising result: for a RAM with left shift and right shift ($\downarrow$), we already know that RAM[$\uparrow, \downarrow$]-PTIME = PSPACE [22]. The division instruction is used to generate a much more complex set of strings than a PRAM[$\uparrow, \downarrow$] can generate.

We consider the power of division paired with multiplication rather than with left shift, as well as the power of only division with our basic instruction set. From Hartmanis and Simon [12], we also know that RAM[$*, \div$]-PTIME = PSPACE.

In the following, let MD be a PRAM[$*, \div$] that uses $T(n)$ time and $P(n)$ processors. Let MD' be a PRAM[$*, \div$] that uses only short addresses and simulates MD according to the Associative Memory Lemma. Thus, MD' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and only addresses in $0, 1, \ldots, O(P(n)T(n))$.

5.1. *Simulation of* PRAM[$*, \div$] *and* PRAM[$\div$] *by PRAM*

We begin by describing the simulation of a PRAM[$*, \div$] by a PRAM. The idea of the proof is that we modify the simulation of a PRAM[$*$] by a PRAM (Section 4.1). Because this simulation depends on the relationship between circuits and PRAMs, we are interested in the Boolean circuit complexity of division. Beame et al. [1] developed a circuit for dividing two n-bit numbers in depth $O(\log n)$. This circuit, however, is polynomial-time uniform, and we need the stronger condition of logspace-uniformity. Reif [17] devised a logspace-uniform, depth $O(\log n \log\log n)$ division circuit, and Shankar and Ramachandran [21] improved the size bound of this circuit.

Lemma 5.1 (Shankar and Ramachandran [21]). *A PRAM can compute the quotient of two x-bit operands in time $O(\log x)$ with $O((1/\delta^4)x^{1+\delta})$ processors, for any $\delta > 0$.*

Simulation. We construct at PRAM Z that simulates MD via MD' in time $O(T^2(n))$. We modify the simulation of a PRAM[$*$] by a PRAM (Section 4.1) to deal with division instructions. By Lemma 5.1 with $x = n2^{T(n)}$ and $\delta = 1$, Z can perform a division in time $O(T(n))$ with the available secondary processors.

Theorem 5.2. *For all $T(n) \geq \log n$, PRAM[$*, \div$]-TIME($T(n)$) $\subseteq$ PRAM-TIME($T^2(n)$).*

Proof. By the simulation above, Z simulates each step of MD' in time $O(T(n))$ with $O(P^2(n)T^2(n)\, n^2 4^{T(n)} \log T(n))$ processors. MD' runs for $O(T(n))$ steps, so Z can simulate MD via MD' in $O(T^2(n))$ steps. □

We now present the simulation of a PRAM[$\div$] by a PRAM. Let D be a PRAM[$\div$] that uses $T(n)$ time and $P(n)$ processors. We construct a PRAM Z that simulates D in time $O(T(n)\log(n+T(n)))$. Z acts as a circuit to simulate the computation of D.

Simulation. We modify the simulation of a PRAM[$*$] by a PRAM from Section 4.1. In $T(n)$ steps, a PRAM[$*$] can build integers of length $n2^{T(n)}$, whereas a PRAM[$\div$] can build only integers of length $O(n+T(n))$. As a result, Z partitions the memory into blocks containing only $O(n+T(n))$ cells each. Z activates $P(n)$ primary processors, each with $O((1/\delta^4)(n+T(n))^{1+\delta})$ secondary processors. The simulation proceeds along the same lines as in Section 4.1 except for division instructions. By Lemma 5.1, Z can perform a division in time $O(\log(n+T(n)))$.

Theorem 5.3. *For all* $T(n) \geqslant \log n$,

$$\text{PRAM}[\div]\text{-TIME}(T(n)) \subseteq \text{PRAM-TIME}(T(n)\log(n+T(n))).$$

Proof. By the simulation above, Z simulates D in time $O(T(n)\log(n+T(n)))$ with $O((P(n)/\delta^4)(n+T(n))^{1+\delta})$ processors. □

5.2. Simulation of PRAM[$*$, $\div$] *and* PRAM[$\div$] *by circuits and Turing machine*

Next, we consider the simulation of a PRAM[$*$, $\div$] by circuits and a Turing machine. We construct a TM M that simulates MD via MD' in $T^2(n)\log T(n)$ space by modifying the simulation of a PRAM[$*$] by a TM (Section 4.2).

We need the following lemmas.

Lemma 5.4 (Shankar and Ramachandran [21]). *A logspace-uniform, bounded fan-in circuit can compute the quotient of two x-bit operands in depth* $O(\log x \log\log x)$ *and size* $O((1/\delta^4)\, x^{1+\delta})$, *for any* $\delta > 0$.

Lemma 5.5. *For each n, every language recognized by a* PRAM[$*$, $\div$] MD *in time* $T(n)$ *can be recognized by a logspace-uniform bounded fan-in circuit* DC_n *of depth* $O(T^2(n)\log T(n))$.

Proof. Fix an input length n. Let BC_n be the logspace-uniform, bounded fan-in circuit described in Lemma 4.5 that simulates a PRAM[$*$]. Let DC_n be BC_n with additional circuit blocks for division. To handle division instructions with operands of length at most $x = n2^{T(n)}$, we used the logspace-uniform $O(\log x \log\log x)$ depth bounded fan-in division circuit specified in Lemma 5.4. Circuit DC_n is at most at constant factor larger in size than BC_n. Hence, DC_n uses depth $O(T(n)\log T(n))$ to simulate each step of MD. □

Lemma 5.6. *A logspace-uniform, unbounded fan-in circuit can compute the quotient of two x-bit operands in depth* $O(\log x)$ *and size* $O((1/\delta^4)\, x^{2+\delta})$, *for any* $\delta > 0$.

Proof. Lemma 5.6 follows from Lemma 5.4 by the transformation due to Chandra et al. [4] from a bounded fan-in circuit to an unbounded fan-in circuit. The transformation preserves logspace-uniformity. □

Lemma 5.7. *For each n, every language recognized by a* PRAM$[*,\div]$ *MD in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform unbounded fan-in circuit UD_n of depth $O(T^2(n))$ and size $O(nT^2(n)32^{T(n)}(n^2+T^2(n)))$.*

Proof. Fix an input size n. Let UD be the logspace-uniform circuit UC_n of Lemma 4.4 with additional circuit blocks for division, using the circuits described in Lemma 5.6. For operands of size at most $x=n2^{T(n)}$, the depth of each division block becomes $O(T(n))$. Overall, the depth of UD_n is $O(T^2(n))$, and the size is the same as that of UC_n. □

Theorem 5.8. *For all $T(n)\geqslant\log n$,*

$$\text{PRAM}[*,\div]\text{-TIME}(T(n))\subseteq\text{DSPACE}(T^2(n)\log T(n)).$$

Proof. Theorem 5.8 follows from Lemma 5.5 and Borodin's result [2] that a bounded fan-in circuit of depth $D(n)$ can be simulated in space $O(D(n))$ on a Turing machine. □

Through Theorem 5.8 and the simulation of DSPACE($T(n)$) in PRAM-TIME($T(n)$) [9], we can obtain an $O(T^2(n)\log T(n))$ time simulation of a PRAM$[*,\div]$ by a PRAM. The direct simulation of Theorem 5.2 is more efficient.

Through Theorem 5.2 and the simulation of PRAM-TIME($T(n)$) in DSPACE($T^2(n)$) [9], we obtain an $O(T^4(n))$ space simulation of a PRAM$[*,\div]$ by a TM. The simulation of Theorem 5.8 is more efficient.

Let $PC=\{PC_1,PC_2,\ldots\}$ be the family of bounded fan-in circuits that simulates the family C of unbounded fan-in circuits described in Lemma 4.3 (from [24]). For a fixed input size n, the depth of PC_n is $O(T(n)\log P(n)T(n))$ and the size is $O(P(n)T(n)L(n)(L^2(n)+P(n)T(n)))$.

Theorem 5.9. *For each n, every language recognized by a* PRAM$[\div]$ *D in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform, bounded fan-in circuit DB_n of depth $O(T(n)\log P(n)+T(n)\log(n+T(n))\log\log(n+T(n)))$.*

Proof. Fix an input length n. Let PC_n be the bounded fan-in circuit described above that simulates a PRAM. Let DB_n be PC_n with additional circuit blocks for division. To handle division instructions with operands of length at most $x=n+T(n)$, we use the log-space uniform, $O(\log x\log\log x)$ depth, bounded fan-in division circuit specified in Lemma 5.4. Circuit DB_n is at most a constant factor larger in size than PC_n. Hence, DB_n uses depth $O(\log P(n)+\log(n+T(n))\log\log(n+T(n)))$ to simulate each step of D. □

Lemma 5.10. *An off-line Turing machine can compute the quotient of two n-bit operands in $O(\log n\log\log n)$ space.*

Proof. Borodin [2] proved that an off-line TM can simulate a logspace-uniform circuit with bounded fan-in and depth $D(n)$ in space $O(D(n))$. Combined with the logspace-uniform $O(\log n \log\log n)$ depth division circuit of Shankar and Ramachandran [21], we have the lemma. □

Theorem 5.11. *For all* $T(n) \geqslant \log n$, $\text{PRAM}[\div]\text{-TIME}(T(n)) \subseteq \text{DSPACE}(T^2(n))$.

Proof. Fortune and Wyllie [9] simulated each PRAM running in time $T(n)$ by a TM running in space $O(T^2(n))$. They used recursive procedures of depth $O(T(n))$ using space $O(T(n))$ at each level of recursion. If we augment the simulated PRAM with division, then by Lemma 5.10, an additional $O(\log(n+T(n))\log\log(n+T(n)))$ space is needed at each level, so $O(T(n))$ space at each level still suffices. Hence, with linear space compression, a TM with space $T^2(n)$ can simulate a $\text{PRAM}[\div]$ running in time $T(n)$. □

5.3. *Simulation of* PRAM[∗, ÷] *by* RAM[∗, ÷]

In this section, we establish the MRAM-uniformity of a bounded fan-in circuit described by Shankar and Ramachandran [21] that performs division in $O(\log n \log\log n)$ depth and $O((1/\delta^4)n^{1+\delta})$, for $\delta > 0$, size. This is the major step leading to a simulation of a PRAM[∗, ÷] by a RAM[∗, ÷].

Given two n-bit inputs, u and v, the division problem is to compute their n-bit quotient, u/v. This reduces to the problem of efficiently computing the n-bit reciprocal of v. Shankar and Ramachandran first normalize v to a number in $[1/2, 1)$, set $x = 1 - v$, and then compute $1/(1-x) = 1 + x + x^2 + x^3 + \cdots + x^{n-1}$. The key portion of their division algorithm, and the only portion for which we explicitly prove the circuit implementation to be MRAM-uniform, is an algorithm for computing the sth power of an r-bit number modulo 2^{r+1}. We must show the following circuit components to be MRAM-uniform: discrete Fourier transform (DFT), DFT^{-1}, square root, $x^s \bmod 2^r + 1$ for restricted r and s, and the circuit component corresponding to the case $r < s^2$.

DFT uniformity

We establish here that a bounded fan-in circuit implementing the DFT is MRAM-uniform. Below we state a DFT algorithm from Cooley and Tukey [7] as described by Quinn [16]. The input to the algorithm is the vector $\boldsymbol{a} = (a(0), a(1), \ldots, a(k))$; the output is the vector $\boldsymbol{b} = (b(0), b(1), \ldots, b(k))$. Let $d = \log_2 k$. For an integer j, let $\#j$ denote its two's complement representation.

Procedure $BIT\_0(j, i)$ returns the integer found by setting the ith bit of $\#j$ to 0.

Procedure $BIT\_1(j, i)$ returns the integer found by setting the ith bit of $\#j$ to 1.

Procedure $OMEGA(i, j, d)$ returns the qth primitive root of unity, where q is the value found by reversing the $i+1$ most significant bits of the d-bit integer j and then padding the result with zeros in the $d-i-1$ least significant positions.

Procedure $REVERSE(i, d)$ returns the integer found by reversing the d-bit binary representation of the integer i. The value of this integer lies between 0 and $2^d - 1$.

DFT Algorithm

begin
  for $i \leftarrow 0$ to $k-1$ **do**
    $r(i) \leftarrow a(i)$
  endfor
  for $i \leftarrow 0$ to $d-1$ **do**
    for $j \leftarrow 0$ to $k-1$ **do**
      $s(j) \leftarrow r(j)$
    endfor
    for $j \leftarrow 0$ to $k-1$ **do**
      $r(j) \leftarrow s(BIT\_0(j, d-i-1)) + OMEGA(i, j, d) * s(BIT\_1(j, d-i-1))$
    endfor
  endfor
  for $i \leftarrow 0$ to $k-1$ **do**
    $b(i) \leftarrow r(REVERSE(i, d))$
  endfor
end

We consider the straightforward implementation of the algorithm as a circuit with the following blocks: $[r \leftarrow a]$, $[s \leftarrow r]$, $[OMEGA]$, $[Mult]$, $[Add]$, and $[REVERSE]$. To establish the MRAM-uniformity of this circuit, we must establish the uniformity of gate connections within each block. The uniformity of the connections between blocks is clear. We name gates such that the binary representation of the name can be partitioned into fields. The gate names are $O(\log k) = O(d)$ bits long.

The $[REVERSE]$ block corresponds to the following steps in the algorithm:

for $i \leftarrow 0$ to $k-1$ **do**
  $b(i) \leftarrow r(REVERSE(i, d))$
endfor

Hence, this block simply routes data from the ***r*** array to the ***b*** array. Let $b_s(i)$ denote the sth bit of $b(i)$. We want to compute on a RAM[*] the gate name corresponding to the input to $b_s(i)$, that is, $r_s(REVERSE(i, d))$. As a first step, compute $REVERSE(i, d)$ in $O(\log d) = O(\log \log k)$ stages, with each stage requiring constant time, given the d-bit value i. In the mth stage, the RAM[*] swaps adjacent blocks of $d/2^m$ bits. The RAM[*] executing the algorithm cannot shift numbers to the right, so it performs only left shifts. (A RAM[*] performs a left shift by multiplying by the appropriate power of two.) The RAM[*] simply keeps track of the number of "insignificant" bits on the right introduced by performing only left shifts.

Assume that we are given a gate name g and that gate g corresponds to $b_s(i)$. Given a gate name h, the RAM[*] must compute whether gate h is the input to gate g. Therefore, the RAM[*] will generate the name of gate g's input (that is,

$r_s(REVERSE(i,d))$) and compare it with h. Also assume that the binary representations of g and h are appropriately masked so that only selected fields of the gate names are nonzero. Thus, the unmasked portions of g are $b\,|\,i\,|\,s$ (variable b, index i, and bit numbers s). From the algorithm, the corresponding fields in the name of g's input are $r\,|\,REVERSE(i,d)\,|\,s$. To test for equality, the RAM[$*$] shifts r and s to appropriate positions on the left and right of $REVERSE(i,d)$, then shifts the corresponding fields from h to the left to test for equality. Thus, if g is a gate in the [$REVERSE$] block, then we can test whether h is its input in $O(\log d)$ time.

The uniformity of the [BIT_0], [BIT_1], [$r \leftarrow a$], [$s \leftarrow r$], and [Add] blocks is clear. The uniformity of the [$OMEGA$] block is clear because ω is a power of two.

In the above discussion, we have shown that a RAM[$*$] can compute the input to a single gate g within the necessary time bounds when $\#g$ is appropriately masked to reveal only certain fields. To establish MRAM-uniformity, however, the RAM[$*$] must take the input I comprising all pairs of integers $O(\log k)$ bits long, mask these integers, and compute the inputs for the gates corresponding to all exposed fields simultaneously. The masking is similar to that done to establish the MRAM-uniformity of the circuit simulating a PRAM[$*$] in Section 4.3. Observe that any of the above procedures can operate on a long integer, comprising a set of fields separated by zeros. Hence, we obtain that the DFT circuit implementing the algorithm described above is MRAM-uniform.

Inverse DFT

For the inverse DFT circuit, the above discussion establishes the uniformity of most of its sections. The following step of the inverse DFT algorithm is one significant exception:

$$b(i) \leftarrow r(REVERSE(i,d))/k.$$

Note that $k = O(\log^{3/4} x)$ since $r = \log x$ and we are in the case $r \geqslant s^2$. Hence, the inverses needed can be obtained by generating a table for all possible relevant values. Since k is very small, this table can be generated with a uniform, polynomial (in r) size circuit. Thus, the inverse DFT circuit is MRAM-uniform.

Square root, $x_i 2^i \bmod 2^k + 1$, *and* $x_i 2^{-i} \bmod 2^k + 1$ *uniformity*

As for the inverse DFT, the values needed to compute the square roots are very small and are obtained by generating a table for all relevant values. Thus, the square root circuit is MRAM-uniform.

Next, we want to establish the MRAM-uniformity of the portions of the circuit that compute $x_i 2^i \bmod 2^k + 1$ and $x_i 2^{-i} \bmod 2^k + 1$. The values of x_i are in the range $0, \ldots, 2^k - 1$; the values of i are in the range $0, \ldots, k-1$. Since $x_i 2^{-i} \bmod 2^k + 1 = x_i 2^{-i}$, we need only be concerned with computing $x_i 2^i \bmod 2^k + 1$.

From the bounds on i and x_i, $x_i 2^i$ is at most $2k$ bits long. Split $x_i 2^i$ into two k-bit portions, denoting the lower-order portion by r_0 and the higher-order portion by r_1. Since $2^k \equiv -1 \bmod 2^k + 1$, we have $x_i 2^i = r_1 2^k + r_0 \equiv (r_0 - r_1) \bmod 2^k + 1$, which can be

computed with a subtraction, a comparison, and an addition. Thus, this portion of the circuit is clearly MRAM-uniform.

Uniformity of the case $r < s^2$

At this point, we consider the case $r < s^2$ of the modular power algorithm. Again, the values needed can be obtained by generating a table for all possible relevant values. This table can be generated by a uniform, polynomial-size circuit. Thus, this portion of the circuit is MRAM-uniform.

Lemma 5.12. *The division circuit of Shankar and Ramachandran* [21] *is MRAM-uniform.*

Proof. From the above discussion, each component of Shankar and Ramachandran's modular power algorithm, implemented as a circuit, is MRAM-uniform. Therefore, the modular power algorithm and, hence, their division algorithm, may be implemented as an MRAM-uniform circuit. □

We next simulate a PRAM[$*, \div$] by an MRAM-uniform, bounded fan-in circuit family, then simulate this circuit family by a RAM[$*$] and, hence, a RAM[$*, \div$].

Let DC denote the family of bounded fan-in circuits described in Lemma 5.5 that simulates a PRAM[$*, \div$] in depth $O(T^2(n)\log T(n))$ and size $O(nT^2(n)\, 32^{T(n)}(n^2 + T^2(n)))$. We construct a family of bounded fan-in circuits DC' from DC. Fix an input size n. The circuit DC'_n is exactly the same as the circuit DC_n except that DC'_n uses carry–save multiplication blocks for reasons of MRAM-uniformity (as in Lemma 4.9). DC'_n has depth $O(T^2(n)\log T(n))$ and size $O(nT^2(n)\, 32^{T(n)}(n^2 + T^2(n)))$.

Lemma 5.13. *For each n, every language recognized by a* PRAM[$*, \div$] *MD in time $T(n)$ with $P(n)$ processors can be recognized by the bounded fan-in, MRAM-uniform circuit DC'_n of depth $O(T^2(n)\log T(n))$ and size $O(nT^2(n)\, 32^{T(n)}(n^2 + T^2(n)))$.*

Proof. The proof of the PRAM[$*, \div$] simulation is similar to that given for Lemma 4.5. By an argument similar to the proof of Lemma 4.7 using Lemma 5.12, DC' is MRAM-uniform. □

Theorem 5.14. *For all* $T(n) \geqslant \log n$, PRAM[$*, \div$]-TIME($T(n)$) $\subseteq$ RAM[$*$]-TIME($T^2(n)\log T(n)$) $\subseteq$ RAM[$*, \div$]-TIME($T^2(n)\log T(n)$).

Proof. By Lemma 5.13, a PRAM[$*, \div$] running in time $T(n)$ with $P(n)$ processors can be simulated by a bounded fan-in, MRAM-uniform circuit DC'_n of depth $O(T^2(n)\log T(n))$ and size $O(nT^2(n)32^{T(n)}(n^2 + T^2(n)))$. By Theorem 3.4, a RAM[$*$] can simulate DC'_n in time $O(T^2(n)\log T(n))$ and, hence, so can a RAM[$*, \div$]. □

Observe that a RAM[$\div$] is unable to generate long integers. Therefore, the gap between the time-bounded power of a PRAM[$\div$] and the time-bounded power of a RAM[$\div$] is much greater than the gap between the power of a RAM[$*$] and the power of a PRAM[$*$].

6. Shift

Pratt and Stockmeyer [15] proved that for a vector machine, that is, a RAM[$\uparrow, \downarrow$] without addition or subtraction in which left-shift ($\uparrow$) and right-shift ($\downarrow$) distances are restricted to a polynomial number of bit positions, RAM[$\uparrow, \downarrow$]-PTIME = PSPACE. Simon [22] proved the same equality for RAMs with *unrestricted* left shift and right shift, addition, and subtraction. We prove that polynomial time on PRAMs with unrestricted shifts is equivalent to polynomial time on basic PRAMs and on RAM[$\uparrow, \downarrow$]s and to polynomial space on Turing machines (TMs).

6.1. *Simulation of* PRAM[$\uparrow, \downarrow$] *by* PRAM

By repeated application of the left-shift instruction, a PRAM[$\uparrow, \downarrow$] can generate numbers of length

$$O(2^{2^{\cdots^{2^n}}}) \Big\} \begin{matrix}\uparrow \\ \downarrow\end{matrix} T(n)$$

in $T(n)$ steps. These extremely large numbers contain very long strings of 0's, however. (If Boolean operations are used, then the numbers have very long strings of 0's and very long strings of 1's.) Since we cannot write such numbers in polynomial space, nor can we address an individual bit of such a number in polynomial space, we encode the numbers and manipulate the encodings. We use the *marked interesting bit* (*MIB*) *encoding*, an enhancement of the interesting bit encoding of Simon [22]. Let d be an integer, and let $len(d) = w$. Let $b_{w-1} \ldots b_0$ be the w-bit two's complement representation of d. An *interesting bit* of d is a bit b_i such that $b_i \neq b_{i+1}$. (Bit b_{w-1} is not an interesting bit.)

If d has an interesting bit at b_i and the next interesting bit is at b_j, $i < j$, then the bits $b_j b_{j-1} \ldots b_{i+1}$ are identical. If these bits are 0's (1's), then we say that d has a *constant interval* of 0's (1's) ending at b_j.

If a constant interval has length 1, then the entire interval is a single bit, which is an interesting bit. We call such an interesting bit a *singleton.* We mark interesting bits that are singletons. We define the MIB encoding as

$$E(0) = 0s,$$

$$E(01) = 1s,$$

$$E(d) = (E(a_t)q_t, \ldots, E(a_2)q_2, E(a_1)q_1; r),$$

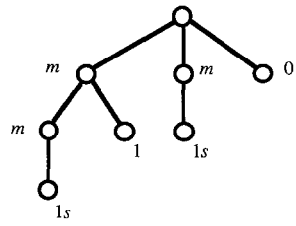


Fig. 4. Encoding tree for E(01100).

where d is an integer, a_j is the position of the jth interesting bit of d; $q_j = s$ if the jth interesting bit is a singleton and $q_j = m$ if the jth interesting bit is not a singleton; and r is the value (0 or 1) of the rightmost bit of d. Call q_j the *mark* of the jth interesting bit. Call r the *start bit*.

The marks permit the simulator to efficiently determine from $E(x)$ and $E(y)$ whether $x+1=y$, using the procedure *PLUS_ONE* given below.

For example, $E(01100) = (E(011)m, E(01)m; 0) = ((E(01)m; 1)m, 1sm; 0) = ((1sm; 1)m, 1sm; 0)$. For all d, define $val(E(d)) = d$.

An encoding can be viewed as a tree. For the encoding $E(d) = (E(a_t)q_t, \ldots, E(a_2)q_2, E(a_1)q_1; r)$, a *node* is associated with each of $E(a_t)q_t, \ldots, E(a_1)q_1, r$. A root node is associated with the entire encoding of $E(d)$ and holds nothing. A nonroot node holds one of: $0s$; $1s$; r, the start bit; or q_j, the mark of the jth interesting bit. If a node holds $0s$, $1s$, or r, then it is a leaf. If a node holds q_j, then it is an internal node, and its children are the nodes of $E(a_j)$. Figure 4 contains a sketch of the encoding tree of E(01100).

For a node α corresponding to $E(a_k)q_k$, the value of the subtree rooted at α, $val(\alpha)$, is a_k, the value of $E(a_k)$. Thus, $val(\alpha)$ is the position of an interesting bit.

We define *level 0* of a tree as the root. We define *level j* of a tree as the set of all children of nodes in level $j-1$ of the tree.

A *pointer* into an encoding specifies a path starting at the root of the tree. For instance, the pointer 7.5.9 specifies a path x_0, x_1, x_2, x_3 in which x_0 is the root, x_1 is the 7th child (from the right) of x_0, x_2 is the 5th child of x_1, and x_3 is the 9th child of x_2. A pointer also specifies the subtree rooted at the last node of the path.

For an integer d, suppose $E(d) = (E(a_t)q_t, \ldots, E(a_1)q_1; r)$. We define $intbits(d) = t$, the number of interesting bits in d. Viewing $E(d)$ as a tree, we refer to $E(a_s)$ as a *subtree* of $E(d)$. We define the kth *subtree at level c* of $E(d)$ as the kth subtree from the right whose root is distance c from the root of $E(d)$. We define $depth(d)$ recursively by

$$depth(0) = depth(01) = 1,$$

$$depth(d) = 1 + \max\{depth(a_t), \ldots, depth(a_1)\}.$$

We now state three lemmas, analogous to those of Simon [22], that bound the size of an encoding. Lemma 6.1 bounds the depth of an encoding and the number of

interesting bits in a number generated by a PRAM$[\uparrow,\downarrow]$. Let *bool* be a set of Boolean operations. The proof of Lemma 6.1 is straightforward.

Lemma 6.1. *Suppose a processor* P_m *executes* $r(i)\leftarrow r(j)\bigcirc r(k)$, $\bigcirc\in\{+,\uparrow,\downarrow,-,bool\}$.

(i) *If* $\bigcirc$ *is* $+$, *then* $depth(rcon_m(i))\leqslant 1+\max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ *and* $intbits(rcon_m(i))\leqslant intbits(rcon_m(j))+intbits(rcon_m(k))$.

(ii) *If* $\bigcirc$ *is a Boolean operation, then* $depth(rcon_m(i))\leqslant\max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ *and* $intbits(rcon_m(i))\leqslant intbits(rcon_m(j))+intbits(rcon_m(k))$.

(iii) *If* $\bigcirc$ *is* $-$, *then* $depth(rcon_m(i))\leqslant 1+\max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ *and* $intbits(rcon_m(i))\leqslant intbits(rcon_m(j))+intbits(rcon_m(k))$.

(iv) *If* $\bigcirc$ *is* $\uparrow$ *or* $\downarrow$, *then* $depth(rcon_m(i))\leqslant 2+\max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ *and* $intbits(rcon_m(i))\leqslant 1+intbits(rcon_m(j))$.

Part (i) of Lemma 6.2 bounds the number of subtrees below first level nodes in an encoding; Part (ii) bounds the number of subtrees below fth level nodes in an encoding, $f>1$. The proof of Lemma 6.2 follows from Lemma 6.1.

Lemma 6.2. *Suppose a processor* P_m *executes* $r(i)\leftarrow r(j)\bigcirc r(k)$, *where* $\bigcirc\in\{+,\uparrow,\downarrow,-,bool\}$, $E(rcon_m(i))=(E(a_r),\ldots,E(a_1);w_i)$, $E(rcon_m(j))=(E(b_s),\ldots,E(b_1);w_j)$, *and* $E(rcon_m(k))=(E(c_t),\ldots,E(c_1);w_k)$, *where* a_v, b_v, *and* c_v *denote the positions of the* v*th interesting bits of* $rcon_m(i), rcon_m(j)$, *and* $rcon_m(k)$, *respectively.*

(i) *For* $E(a_v)$ *(that is, the* v*th subtree at level* 1 *of* $E(rcon_m(i))$),

(a) *if* $\bigcirc$ *is* $+$, *then* $intbits(a_v)\leqslant 1+\max_q\{intbits(b_q), intbits(c_q)\}$,

(b) *if* $\bigcirc$ *is* $\uparrow$ *or* $\downarrow$, *then* $intbits(a_v)\leqslant\max_q\{intbits(b_q)\}+intbits(rcon_m(k))\}$,

(c) *if* $\bigcirc$ *is a Boolean operation, then* $intbits(a_v)\leqslant\max_q\{intbits(b_q), intbits(c_q)\}$, *and*

(d) *if* $\bigcirc$ *is* $-$, *then* $intbits(a_v)\leqslant 1+\max_q\{intbits(b_q), intbits(c_q)\}$.

(ii) *For* $E(\beta)$ *a subtree at level* $f>1$,

(a) *if* $\bigcirc$ *is* $+$, $-$, *or a Boolean operation, then* $intbits(\beta)\leqslant 1+\max_q\{intbits(qth\ subtree\ of\ rcon_m(j)\ at\ level\ f), intbits(qth\ subtree\ of\ rcon_m(k)\ at\ level f)\}$, *and*

(b) *if* $\bigcirc$ *is* $\uparrow$ *or* $\downarrow$, *then* $intbits(\beta)\leqslant 1+\max_q\{intbits\ (qth\ subtree\ of\ rcon_m(j)\ at\ level\ f), intbits(qth\ subtree\ of\ rcon_m(k)\ at\ level\ f-1)\}$.

Lemma 6.3. *An encoding tree for an integer generated by a* PRAM$[\uparrow,\downarrow]$ *in* $T(n)$ *steps contains up to* $O(4^{T^2(n)})$ *nodes.*

Proof. Let d be an integer generated by a PRAM$[\uparrow,\downarrow]$. By Lemma 6.1, $depth(d)\leqslant 2T(n)$. If ω is the input to the PRAM$[\uparrow,\downarrow]$ and ω has length n, then $intbits(\omega)\leqslant n$. Let $E(\beta)$ be either $E(d)$ or a subtree of $E(d)$. By Lemmas 6.1 and 6.2, $intbits(\beta)\leqslant n2^{T(n)}$. The depth of the encoding is at most $2T(n)$, and every internal node has at most $n2^{T(n)}$ children. Therefore, the encoding may have up to $O(4^{T^2(n)})$ nodes, assuming $T(n)\geqslant\log n$. □

We describe here an efficient simulation of a PRAM$[\uparrow,\downarrow]$ by a basic PRAM. Let S be a PRAM$[\uparrow,\downarrow]$ that uses $T(n)$ time and $P(n)$ processors. Let S' be a PRAM$[\uparrow,\downarrow]$ that uses only short addresses and simulates S according to the Associative Memory

Lemma. Thus, S' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and addresses only in $0, 1, \ldots, O(P(n)T(n))$. Let q be a constant such that each processor in S' uses only q registers. By Lemma 6.3, for numbers generated by S (and therefore S'), the encoding may have up to $O(4^{T^2(n)})$ nodes.

We construct a PRAM Z that simulates S via S' in $O(T^2(n))$ time, using $O(P^2(n)T(n)4^{T^2(n)})$ processors. For ease of description, we allow Z to have $q+7$ separate shared memories, $mem_0, \ldots, mem_{q+6}$, which can be interleaved. This entails no loss of generality and only a constant-factor time loss.

Initialization. Z partitions mem_0 into $O(P(n)T(n))$ *blocks* of $4^{T^2(n)}$ cells each. This partitioning allots one block per cell accessed by S', where each block comprises one cell per node of the encoding tree. Z partitions each of $mem_1, \ldots, mem_q$ into $O(P^2(n)T(n))$ blocks. This allots one block per processor of S' and one memory per local register used by a processor. (See Fig. 5.) Let $B_i(m)$ denote the mth block of mem_i. Throughout the simulation, $B_0(j)$ contains $E(con(j))$, and $B_i(m)$, $1 \leqslant i \leqslant q$, contains $E(rcon_m(i))$ of S'.

mem_0	shared memory
$mem_1, \ldots, mem_q$	local memories
mem_{q+1}	address table
mem_{q+2}	communication
mem_{q+3}	rightmost child
mem_{q+4}	parent
mem_{q+5}	rightmost sibling
mem_{q+6}	two's complement representation of cell contents.

Fig. 5. Shared memories of Z.

Z activates $O(P^2(n)T(n))$ *primary* processors, one for each processor used by S'. In mem_{q+1}, these processors construct an address table. The jth entry of this table is $j \cdot 4^{T^2(n)}$, the address of the first cell of the jth block in every memory. The maximum address is $O(P^2(n)T(n)4^{T^2(n)})$, so Z computes this address (and the entire table) in $O(T^2(n))$ time.

Each primary processor now deploys $4^{T^2(n)}$ *secondary* processors, one for each cell in a block, in $O(T^2(n))$ time. To implement a broadcast in constant time, each primary processor P_m uses $c_{q+2}(m)$ as a communication cell. When the secondary processors are not otherwise occupied, they concurrently read this cell at each time step, waiting for a signal from the primary processor to indicate their next tasks.

Consider a complete d-ary tree $\mathcal{A}$ with depth $2T(n)$. We number the nodes of $\mathcal{A}$, starting with the root as node 1, in the order of a right-to-left breadth-first traversal. Node number j has children numbered $dj-(d-2), \ldots, dj, dj+1$; its parent is numbered $\lfloor (j+(d-2))/d \rfloor$.

We view a block as a linear array storing $\mathcal{A}$ with $d=4^{T(n)}$. Node numbers correspond to locations in the array. Let $node(j)$ denote the node whose number is j.

Let $num(\alpha)$ denote the node number of node α. For each primary processor, the jth secondary processor, $1 \leqslant j \leqslant 4^{T^2(n)}$, handles $node(j)$. Let $proc(\alpha)$ denote the secondary processor assigned to node α.

Each encoding is a subtree of A because all encoding nodes have fewer than $4^{T(n)}$ children. Let $p(\alpha)$ denote the parent of node α; let $rc(\alpha)$ denote the rightmost child of node α; let $lc(\alpha)$ denote the leftmost nonempty child of node α. When a primary processor and its secondary processors update $E(con(i))$ or $E(rcon_g(i))$, they also update $num(lc(\alpha))$ for every node α. Let $right(\alpha)$ denote $num(\alpha) - num(rc(p(\alpha)))$. That is, $right(\alpha)$ denotes which child α is of $p(\alpha)$, counting from the right. Similarly, let $left(\alpha)$ denote $num(\alpha) - num(lc(p(\alpha)))$. That is, $left(\alpha)$ denotes which child α is of $p(\alpha)$, counting from the left.

Using mem_{q+2} for communication with primary processor P_h, corresponding to processor P_0 of S', $proc(node(j))$, $1 \leqslant j \leqslant 4^{T^2(n)}$, writes $num(rc(node(j)))$ in $c_{q+3}(j)$, $num(p(node(j)))$ in $c_{q+4}(j)$, and $num(rc(p(node(j))))$ in $c_{q+5}(j)$ in $O(T(n))$ time. Then the processor for each node j can compute $right(j)$.

All the addresses of cells accessed by S' can be constructed using only addition and subtraction. In order to quickly perform indirect addressing, Z generates all cell and register contents in standard two's complement representation, except for results of shifts. The two's complement representation of local register $r_g(i)$ of S', if $rcon_g(i)$ is constructed without shifts, is stored in $c_{q+6}(g(q+1)+i)$. The two's complement representation of shared memory cell $c(j)$ of S', if $con(j)$ is constructed without shifts, is stored in $c_{q+6}((j+1)(q+1))$. If the value v in a register or a shared memory cell is the result of a shift, then S' does not use v as an address, and S' uses no other value computed from v as an address.

As the final initialization step, Z converts the input to the MIB encoding, writing the encoding into $B_0(0)$. Z writes the input integer in $c_{q+6}(q+1)$.

Simulation. In a general step of processor P_g of S', P_g executes instruction *instr*. Assume for now that *instr* has the form $r(i) \leftarrow r(j) \bigcirc r(k)$. To simulate this step, the corresponding primary processor P_m of Z and its secondary processors perform four tasks:

Task 1. If $\bigcirc$ is not a shift, then perform $\bigcirc$ on $con_{q+6}(g(q+1)+j)$ and $con_{q+6}(g(q+1)+k)$, writing the result in $con_{q+6}(g(q+1)+i)$.

Task 2. Merge the first level of the encodings $E(rcon_g(j))$ and $E(rcon_g(k))$.

Task 3. Determine where the interesting bits of $E(rcon_g(i))$ occur in the merged encodings and compute their marks.

Task 4. Compress these marked interesting bits into the proper structure.

Z uses procedures *MERGE* in task 2 and *COMPRESS* in task 4. Depending on the operation $\bigcirc$, Z may also use procedures *BOOL* and *ADD* in task 3. These procedures are described below.

Procedures *MERGE*, *COMPRESS*, *BOOL*, and *ADD* call procedure *COMPARE*, which we now specify. Let j and k be nonnegative integers, and let ψ_1 and ψ_2 be encoding pointers. If $m = \lambda$, the empty string, then $COMPARE(j, \psi_1, k, \psi_2, m)$ compares the value of subtree $E(con(j)).\psi_1$ with the value of subtree $E(con(k)).\psi_2$.

Similarly, if $m \neq \lambda$, then *COMPARE* compares the value of subtree $E(rcon_m(j)).\psi_1$ with the value of subtree $E(rcon_m(k)).\psi_2$.

Suppose $m = \lambda$; the case $m \neq \lambda$ is similar. For each node α in the first level of $E(con(j)).\psi_1$ simultaneously, $proc(\alpha)$ determines $left(\alpha)$. Then $proc(\alpha)$ computes $num(\beta)$ such that node β is in the first level of $E(con(k)).\psi_2$ and $left(\beta) = left(\alpha)$ by reading $num(lc(E(con(k)).\psi_2))$. Next, $proc(\alpha)$ recursively compares the values of the subtrees rooted at α and β. *COMPARE* is recursive in the depth of the encoding, taking constant time at each level. Consequently, $COMPARE(j, \psi_1, k, \psi_2, m)$ takes $O(T(n))$ time.

In task 2, Z merges the first level of the encodings $E(rcon_g(j))$ and $E(rcon_g(k))$. Z does this to compare the positions of interesting bits in $rcon_g(j)$ and $rcon_g(k)$. This comparison is necessary to determine the positions of the interesting bits in $rcon_g(i)$.

The subtrees rooted at the first level of $E(d)$ form a list sorted in increasing order by their values. $MERGE(j, k, i)$ returns, in $B_i(g)$, the list of up to $O(2^{T(n)})$ subtrees resulting from merging the first levels of $E(rcon_g(j))$ and $E(rcon_g(k))$. Each subtree in the merged list retains indications of whether it is from j or k, whether it is the end of a constant interval of 0's or 1's, and its (singleton) mark. By comparing each subtree of the first level of $E(rcon_g(j))$ with each subtree of the first level of $E(rcon_g(k))$ in $O(T(n))$ time, Z can perform a *MERGE* in $O(T(n))$ time. (*Note*: Each subtree in the merged list also indicates whether its value is equal to that of the next subtree in the list.)

We introduce one more procedure before describing the computation of the interesting bits of $rcon_g(i)$. Let $I(d)$ denote the MIB encoding of d without the marks. $PLUS\_ONE(k, \psi_1, i, \psi_2)$ writes $I(val(E(rcon_g(k)).\psi_1) + 1)$ in the location set aside for subtree ψ_2 in $B_i(g)$. That is, given $E(d)$, for d an integer, *PLUS_ONE* writes $I(d+1)$. *PLUS_ONE* does not write singleton marks. Z uses *PLUS_ONE* to generate $I(d+1)$ to test for equality with $E(x)$, x an integer. The processors ignore marks to interpret $E(x)$ as $I(x)$. At most, the two rightmost interesting bits of $d+1$ are different from those of d. Encoding $I(d+1)$ is easily generated by adding or deleting interesting bits and possibly recursively adding 1 by observing whether d starts with a 0 or a 1 and whether the first 0 is a singleton. *PLUS_ONE* is recursive with depth $T(n)$, the depth of the encodings. *PLUS_ONE* uses constant time at each level, so $O(T(n))$ time overall.

We now are ready to describe how Z accomplishes task 3. Assume without loss of generality that i, j, and k are different. Z's actions in task 3 depend on the operation $\bigcirc$ in *instr*. Define an *interval-pair* to be the intersection of a constant interval in $rcon_g(j)$ and a constant interval in $rcon_g(k)$. For example, three interval-pairs, denoted by a, b, and c, are shown below:

	c	c	b	b	b	a
$rcon_g(j)$	1	1	0	0	0	1
$rcon_g(k)$	0	0	1	1	1	1

The *interval-pair length* of interval-pair a is 1, of interval-pair b is 3, and of interval-pair c is 2.

ZERO_ONE(j,k,i) takes as input the merged list from E($rcon_g(j)$) and E($rcon_g(k)$) in $B_i(g)$ and returns as output an indication for each subtree in the list from E($rcon_g(j)$) (E($rcon_g(k)$)) whether $rcon_g(k)$ ($rcon_g(j)$) is in a constant interval of 0's or 1's at the location specified by the value of the subtree. The secondary processors handling the merged list act as a binary computation tree to pass along the desired information in $O(T(n))$ time.

IP_LENGTH(j,k,i) takes as input the merged list from E($rcon_g(j)$) and E($rcon_g(k)$) in $B_i(g)$ and returns as output in $O(T(n))$ time an indication for each subtree in the list whether the interval pair ending at the location specified by the value of the subtree has length 1 or greater than 1. To perform this computation, Z calls *PLUS_ONE*(i,ψ_1,i',ψ_1) in parallel for each subtree in the list, where ψ_1 is the location of the subtree in the list and i' refers to $B_{q+2}(i)$. Suppose the subtree encodes the integer d. Then, in parallel, Z tests $I(d+1)$ for equality with the next subtree in the list. If they have equal value, then the interval-pair length is 1; otherwise, the interval-pair length is greater than 1.

If *instr* is $r(i)\leftarrow r(j)+r(k)$, then Z calls *ADD*$(j,\lambda,k,\lambda,i,\lambda)$. *ADD*$(j,\psi_1,k,\psi_2,i,\psi_3)$ writes E(val(E($rcon_g(j)$).ψ_1)+val(E($rcon_g(k)$).ψ_2)) in the location set aside for subtree ψ_3 in $B_i(g)$. Assume that we have the merged encodings of E($rcon_g(j)$) and E($rcon_g(k)$) in $B_i(g)$. Z must compute the interesting bits and their marks. To accomplish task 3, Z must test four conditions at the bit location specified by the value of the subtree:

(a) whether the $rcon_g(j)$ and $rcon_g(k)$ pairs are both in constant intervals of 0's, both in 1's, or one in 0's and one in 1's;

(b) whether there is a carry-in to the interval-pair;

(c) whether $rcon_g(i)$ is in a constant interval of 0's or 1's prior to the start of the interval-pair; and

(d) whether the interval-pair length is 1 or greater than 1.

Z calls *ZERO_ONE* and *IP_LENGTH* to test conditions (a) and (d) in time $O(T(n))$. For each subtree α in the list, *proc*(α) does the following. To test condition (b), *proc*(α) tests condition (a) at the preceding subtree in the list. To test condition (c), *proc*(α) determines whether $rcon_g(i)$ is in a constant interval of 0's or 1's at position $val(\alpha)$ using the other three conditions, then passes this information to *proc*$(num(\alpha)+1)$. The processors act as a binary computation tree of height $O(T(n))$ in testing all four conditions. Thus, all four conditions can be tested in $O(T(n))$ time.

A subtree is *interesting* if its value is the location of an interesting bit of $rcon_g(i)$; otherwise, the subtree is *boring*. For each subtree α in the list, *proc*(α) tags subtree α as "interesting" or "boring". In doing so, *proc*(α) may call the procedure *PLUS_ONE*. Z next computes the marks of the interesting bits. The entire procedure takes time $O(T(n))$.

If *instr* is $r(i)\leftarrow r(j)\vee r(k)$, then Z calls *BOOL*$(j,k,i,\vee)$. *BOOL*$(j,k,i,\vee)$ writes E($rcon_g(j)\vee rcon_g(k)$) in $B_i(g)$. *BOOL* is similar to, but simpler than, *ADD*, since only

conditions (a) and (d) must be tested. Other Boolean instructions are handled similarly.

If *instr* is $r(i) \leftarrow r(j) - r(k)$, then Z computes $E(rcon_g(j)+1)$ in $B_i(g)$ by a call to *ADD*, then calls *ADD* $(i, \lambda, k', \lambda, i, \lambda)$, where k' indicates that the start bit of $E(rcon_g(k))$ is complemented (thus adding $rcon_g(j)$ and the two's complement of $rcon_g(k)$). This takes $O(T(n))$ time.

If $rcon_g(k) < 0$ and *instr* is $r(i) \leftarrow r(j) \uparrow r(k)$, then Z treats *instr* as $r(i) \leftarrow r(j) \downarrow r(k)$, substituting $|rcon_g(k)|$ for $rcon_g(k)$. Similarly, if $rcon_g(k) < 0$ and *instr* is $r(i) \leftarrow r(j) \downarrow r(k)$, then Z treats *instr* as $r(i) \leftarrow r(j) \uparrow r(k)$, substituting $|rcon_g(k)|$ for $rcon_g(k)$. Thus, for both shift instructions, we shall assume $rcon_g(k) \geqslant 0$.

If *instr* is $r(i) \leftarrow r(j) \uparrow r(k)$, then the dth interesting bit of $rcon_g(i)$ is in the position specified by the sum of $rcon_g(k)$ and the position of the dth (if the least significant bit of $rcon_g(j)$ is 0) or $(d+1)$st (if the least significant bit of $rcon_g(j)$ is 1 and $rcon_g(k) \neq 0$) interesting bit of $rcon_g(j)$. Z adds $rcon_g(k)$ to the value of each subtree from $rcon_g(j)$. Marks stay the same, except perhaps for the first interesting bit of $rcon_g(j)$: if it has mark s and $rcon_g(k) = 0$, then Z marks it s; otherwise, Z marks it m. This procedure takes $O(T(n))$ time, the time to perform *ADD*.

If *instr* is $r(i) \leftarrow r(j) \downarrow r(k)$, then Z subtracts $rcon_g(k)$ from the value of each first-level subtree of $rcon_g(j)$. Z tags as boring those subtrees for which this difference is negative. For the others, this difference is the location of an interesting bit in $rcon_g(i)$. Let γ denote the subtree whose value specifies the location of the first interesting bit in $rcon_g(i)$. Marks stay the same, except perhaps for γ: if $val(\gamma) = 0$, then Z marks it s; otherwise, Z marks it m. The start bit of $rcon_g(i)$ depends on whether $rcon_g(j)$ is in a constant interval of 0's or 1's at the location specified by the subtree that became γ. This procedure takes time $O(T(n))$, the time to perform *ADD*.

Z accomplishes task 4 by calling *COMPRESS*. *COMPRESS*(i) takes the contents of block i, which implicitly stores a tree in which some subtrees rooted at the first level are tagged to be deleted (boring), and rewrites the tree without the boring subtrees. The secondary processors act as a binary computation tree so that the processors associated with the root of each interesting (that is, not boring) first-level subtree can determine the number of interesting subtrees to the right in time $O(T(n))$. This number specifies the location of the subtree in the compressed tree. Then Z copies each subtree into the appropriate location and writes 0's in the unused locations. Overall, *COMPRESS*(i) takes $O(T(n))$ time.

If processors P_f and P_g of S' wish to simultaneously write $c(j)$, then the corresponding processors P_l and P_m of Z simultaneously attempt to write $B_0(j)$. If $f < g$, then $l < m$, and all secondary processors of P_l are numbered less than all secondary processors of P_m. Thus, in S', P_f succeeds in its write, and in Z, P_l and its secondary processors succeed in their writes.

Theorem 6.4. *For all* $T(n) \geqslant \log n$, PRAM$[\uparrow, \downarrow]$-TIME$(T(n)) \subseteq$ PRAM-TIME$(T^2(n))$.

Proof. In the simulation given above, Z takes $O(T(n))$ time per step of S' to merge two encodings, compute new marked interesting bits, and compress the list into the proper MIB form. S' simulates S in $O(T(n))$ time. Hence, Z takes $O(T^2(n))$ time to simulate S via S'. □

6.2. *Simulations of* PRAM$[\uparrow,\downarrow]$ *by circuits and Turing machine*

We now describe simulations of a PRAM$[\uparrow,\downarrow]$ by a logspace-uniform family of unbounded fan-in circuits, a logspace-uniform family of bounded fan-in circuits, and a Turing machine.

Lemma 6.5. *For each n, every language recognized by a* PRAM$[\uparrow,\downarrow]$ *S in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform unbounded fan-in circuit C_n of depth $O(T^2(n))$ and size $O(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))$.*

Proof. Let Z be the PRAM described in Theorem 6.4, simulating S in $O(T^2(n))$ time with $O(P^2(n)T(n)4^{T^2(n)})$ processors. Fix an input length n. We construct an unbounded fan-in circuit C_n that simulates Z by the construction given by Lemma 4.3. □

Lemma 6.6. *For each n, every language recognized by a* PRAM$[\uparrow,\downarrow]$ *S in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform unbounded fan-in circuit UC_n of depth $O(T^2(n))$, size $O(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))$, and maximum fan-in $O(4^{T(n)}T^2(n))$.*

Proof. Fix an input length n. We construct UC_n from C_n of Lemma 6.5. We reduce the fan-in in the portions of the circuit that simulate updates in the shared memory of Z. The circuit described in Lemma 4.3 allows all processors to attempt to simultaneously write the same cell. This does not occur in Z. During the execution of each procedure of Z over $T(n)$ time steps, either $4^{T^2(n)}$ secondary processors concurrently write the same cell once or $4^{T(n)}$ secondary processors concurrently write the same cell at each of $O(T(n))$ levels of recursion. For the cases in which $4^{T^2(n)}$ secondary processors concurrently write the same cell in one time step, let these processors fan in their results by writing in groups of $4^{T(n)}$ processors over $T(n)$ time steps. Thus, we can modify Z such that at most $4^{T(n)}$ processors attempt to write the same cell at each time step, keeping the time for each procedure at $O(T(n))$. By the construction given in Lemma 4.3, this leads to a maximum fan-in for any gate in UC_n of $O(4^{T(n)}T^2(n))$ if $T(n)\geqslant n$ or $O(4^{T(n)}T(n)n)$ if $T(n)<n$. The circuit remains uniform after modifications to Z because the processors concurrently writing are all secondary processors belonging to the same primary processor. UC_n has depth $O(T^2(n))$ and size $O(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))$. □

Lemma 6.7. *For each n, every language recognized by a* PRAM$[\uparrow,\downarrow]$ *S in time $T(n)$ with $P(n)$ processors can be recognized by a logspace-uniform, bounded fan-in circuit BC_n of depth $O(T^3(n))$ and size $O(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))$.*

Proof. Fix an input length n. Let UC_n be the unbounded fan-in circuit described in Lemma 6.6 that simulates S. The gates of UC_n have maximum fan-in of $O(4^{T(n)}T^2(n))$ if $T(n)\geqslant n$ or $O(4^{T(n)}T(n)n)$ if $T(n)<n$. We construct the bounded fan-in circuit BC_n by replacing each gate of UC_n with fan-in f by a tree of gates of depth $\log f$. Since every $f=O(4^{T(n)}(T^2(n)+nT(n)))$, and $T(n)\geqslant\log n$, BC_n can simulate each gate of UC_n in depth $O(T(n))$. Since UC_n has depth $O((T^2(n))$ by Lemma 6.6, BC_n has depth $O(T^3(n))$. □

Theorem 6.8. *For all $T(n)\geqslant\log n$,* PRAM$[\uparrow,\downarrow]$-TIME$(T(n))\subseteq$DSPACE$(T^3(n))$.

Proof. Theorem 6.8 follows from Lemma 6.7 and Borodin's result [2] that a bounded fan-in circuit of depth $D(n)$ can be simulated in space $O(D(n))$ on a Turing machine. □

Theorem 6.8 and a fundamental result of Fortune and Wyllie [9],

$$\text{DSPACE}(T(n))\subseteq\text{PRAM-TIME}(T(n)) \quad \text{for all } T(n)\geqslant\log n,$$

together imply that PRAM$[\uparrow,\downarrow]$-TIME$(T(n))\subseteq$PRAM-TIME$(T^3(n))$. The direct simulation of Theorem 6.4 is more efficient.

Theorem 6.4 and the other fundamental result of Fortune and Wyllie,

$$\text{PRAM-TIME}(T(n))\subseteq\text{DSPACE}(T^2(n)) \quad \text{for all } T(n)\geqslant\log n,$$

together imply that PRAM$[\uparrow,\downarrow]$-TIME$(T(n))\subseteq$DSPACE$(T^4(n))$. The $O(T^3(n))$ space simulation of Theorem 6.8 is more efficient.

6.3. *Simulation of* PRAM$[\uparrow,\downarrow]$ *by* RAM$[\uparrow,\downarrow]$

Using Theorem 3.3, we now simulate a PRAM$[\uparrow,\downarrow]$ by a RAM$[\uparrow,\downarrow]$. Previously, we simulated a PRAM$[\uparrow,\downarrow]$ by a family of logspace-uniform unbounded fan-in circuits UC according to the simulation by Stockmeyer and Vishkin [24] (Lemma 6.6), then simulated this by a family of logspace-uniform bounded fan-in circuits BC (Lemma 6.7). In this manner, we showed that a family BC of bounded fan-in circuits of depth $O(T^3(n))$ and size $O(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))$ can simulate time $T(n)$ on a PRAM$[\uparrow,\downarrow]$. We need only establish that BC is VM-uniform to give an $O(T^3(n))$ time simulation of a PRAM$[\uparrow,\downarrow]$ by a RAM$[\uparrow,\downarrow]$.

Let $C=\{C_1,C_2,\ldots\}$ be the family of unbounded fan-in circuits described in Lemma 4.3 that simulates a uniform PRAM that runs in time $T(n)$ with $P(n)$ processors. Let $BC'=\{BC'_1, BC'_2,\ldots\}$ be the family of bounded fan-in circuits described in Section 4.3 that simulates the family C of unbounded fan-in circuits. The depth of BC'_n is

$O(T(n)(\log P(n)T(n)))$, and the size is $O(P(n)T(n)L(n)(L^2(n)+P(n)T(n)))$, the same size as C_n.

Lemma 6.9. *BC' is VM-uniform.*

Proof. By a proof similar to that of Lemma 4.7, C_n and BC'_n are VM-uniform. □

Theorem 6.10. *For all $T(n) \geqslant \log n$ and $P(n) \leqslant 2^{T(n)}$, every language recognized with $P(n)$ processors in time $T(n)$ by a PRAM can be recognized by a* RAM$[\uparrow, \downarrow]$ *in time $O(T(n) \log P(n)T(n))$.*

Proof. By Lemma 6.9, BC', the family of bounded fan-in circuits that simulates a PRAM, is VM-uniform. By Theorem 3.3, a RAM$[\uparrow, \downarrow]$ can simulate BC' in time $O(T(n) \log P(n)T(n))$. □

Lemma 6.11. *Let $BC = \{BC_1, BC_2, \ldots\}$ be the family of bounded fan-in circuits described in Lemma* 6.7 *that simulates a uniform* PRAM$[\uparrow, \downarrow]$. *BC is VM-uniform.*

Proof. Let $UC = \{UC_1, UC_2, \ldots\}$ be the family of unbounded fan-in circuits described in Lemma 6.6 that simulates a uniform PRAM$[\uparrow, \downarrow]$. UC has the same form as C, except in the blocks labeled [Update-Common], handling updates to common memory. We reduce the inputs to the gates in this block because of restrictions on the processors that may simultaneously write a cell. As noted in the proof of Lemma 6.9, family C of unbounded fan-in circuits is VM-uniform. It is easy to compute the processors that may simultaneously write a cell, so UC is also VM-uniform. Since UC is VM-uniform, by an argument similar to the proof of Lemma 4.7, BC is VM-uniform. □

Theorem 6.12. *For all $T(n) \geqslant \log n$,*

$$\text{PRAM}[\uparrow, \downarrow]\text{-TIME}(T(n)) \subseteq \text{RAM}[\uparrow, \downarrow]\text{-TIME}(T^3(n)).$$

Proof. By Lemma 6.11, for each n, every language recognized by a PRAM$[\uparrow, \downarrow]$ in time $T(n)$ can be recognized by a VM-uniform, bounded fan-in circuit BC_n of depth $O(T^3(n))$ and size $O(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))$. By Theorem 3.3, there exists a RAM$[\uparrow, \downarrow]$ running in time $O(T^3(n)+\log(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n)))\log \log(P^4(n)T^6(n)16^{T^2(n)}(n+T^2(n))))=O(T^3(n))$ that simulates BC_n. □

Combining Theorem 6.10 with the simulation of a PRAM$[\uparrow, \downarrow]$ by a basic PRAM in Theorem 6.4 yields PRAM$[\uparrow, \downarrow]$-TIME$(T(n)) \subseteq$ RAM$[\uparrow, \downarrow]$-TIME$(T^4(n))$. The simulation of Theorem 6.12 is more efficient.

7. Summary and open problems

7.1. Summary

In this paper, we compared the computational power of time-bounded parallel random-access machines (PRAMs) with different instruction sets. We proved that polynomial time on a PRAM[$*$] or on a PRAM[$*, \div$] or on a PRAM[$\uparrow, \downarrow$] is equivalent to polynomial space on a Turing machine (PSPACE). In particular, we showed the following bounds. Let each simulated machine run for $T(n)$ steps on inputs of length n; let T denote $T(n)$ in the table below. The simulating machines are basic PRAM, Turing machine, uniform family of bounded fan-in circuits, and RAM augmented with the same set of instructions. The bounds for the simulating machine are expressed in time, space, or depth, as shown in parentheses by the machine type.

Table 1
Summary of results

Simulating machine	Simulated machine			
	PRAM[$*$]	PRAM[$*, \div$]	PRAM[$\div$]	PRAM[$\uparrow, \downarrow$]
PRAM (time)	$T^2/\log T$	T^2	$T\log(n+T)$	T^2
TM (space)	T^2	$T^2 \log T$	T^2	T^3
Circuit (depth)	T^2	$T^2 \log T$	T^2	T^3
RAM[*op*] (time)	T^2	$T^2 \log T$	–	T^3

7.2. Open problems

(1) As noted in the introduction, if we could reduce the number of processors used by the simulation of a PRAM[$*$] or PRAM[$*, \div$] or PRAM[$\uparrow, \downarrow$] by a PRAM from an exponential number to a polynomial number, then *NC* would be the languages accepted by PRAM[$*$]s, PRAM[$*, \div$]s, or PRAM[$\uparrow, \downarrow$]s, respectively, in polylog time with a polynomial number of processors. Can the number of processors used by the PRAM in simulating the PRAM[$*$] be reduced to a polynomial in $P(n)T(n)$?

(2) Can a logspace-uniform, fan-in 2 $O(\log n)$-depth circuit perform division? Beame et al. [1] developed a polytime-uniform division circuit. We could improve Theorems 5.2, 5.3, and 5.8 with a logspace-uniform, $O(\log n)$-depth division circuit.

(3) What are the corresponding lower bounds on any of these simulations? Are any of the bounds optimal?

(4) As one of the first results of computational complexity theory, the linear speed-up theorem for Turing machines [13] states that for every multitape Turing machine of time complexity $T(n) \gg n$ and every constant $c > 0$, there is a multitape Turing machine that accepts the same language in time $cT(n)$. The linear speed-up

property of Turing machines justifies the widespread use of order-of-magnitude analyses of algorithms. Do PRAMs also enjoy the linear speed-up property?

Acknowledgment

We thank the anonymous referee for the detailed reading given to this paper and for the comments that improved the presentation of the paper.

References

[1] P.B. Beame, S.A. Cook and H.J. Hoover, Log depth circuits for division and related problems, *SIAM J. Comput.* **15** (1986) 994–1003.
[2]. A. Borodin, On relating time and space to size and depth, *SIAM J. Comput.* **6** (1977) 733–744.
[3] A.K. Chandra, S. Fortune and R. Lipton, Unbounded fan-in circuits and associative functions, *J. Comput. System Sci.* **30** (1985) 222–234.
[4] A.K. Chandra, L.J. Stockmeyer and U. Vishkin, Constant depth reducibility, *SIAM J. Comput.* **13** (1984) 423–439.
[5] S.A. Cook, A taxonomy of problems with fast parallel algorithms, *Inform and Control* **64** (1985) 2–22.
[6] S.A. Cook and R.A. Reckhow, Time bounded random access machines, *J. Comput. System Sci.* **7** (1973) 354–375.
[7] J.W. Cooley and T.W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comp.* **19** (1965) 297–301.
[8] D.M. Eckstein, Simultaneous memory accesses, Tech. Report TR-79-6, Computer Science Dept., Iowa State Univ., 1979.
[9] S. Fortune and J. Wyllie, Parallelism in random access machines, in: *Proc. 10th ACM Symp. Theory Comput.* (1978) 114–118.
[10] L.M. Goldschlager, A universal interconnection pattern for parallel computers, *J. ACM* **29** (1982) 1073–1086.
[11] T. Hagerup, On saving space in parallel computation, *Inform Process. Lett.* **29** (1988) 327–329.
[12] J. Hartmanis and J. Simon, On the power of multiplication in random access machines, in: *Proc. 15th Symp. Switching Automata Theory* (1974) 13–23.
[13] J. Hartmanis and R.E. Stearns, On the computational complexity of algorithms, *Trans. Amer. Math. Soc.* **117** (1965) 285–306.
[14] R.M. Karp and V. Ramachandran, Parallel algorithms for shared-memory machines, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A* (Elsevier, Amsterdam, 1990) 869–941.
[15] V. Pratt and L. Stockmeyer, A characterization of the power of vector machines, *J. Comput. System Sci.* **12** (1976) 198–221.
[16] M.J. Quinn, *Designing Efficient Algorithms of Parallel Computers* (McGraw-Hill, New York, 1987).
[17] J.H. Reif, Logarithmic depth circuits for algebraic functions, *SIAM J. Comput.* **15** (1986) 231–242.
[18] R. Reischuk, Simultaneous WRITES of parallel random access machines do not help to compute simple arithmetic functions, *J. ACM* **34** (1987) 163–178.
[19] W.L. Ruzzo, On uniform circuit complexity, *J. Comput. System Sci.* **22** (1981) 365–383.
[20] A. Schönhage and V. Strassen, Schnelle Multiplikation grosser Zahlen, *Computing* **7** (1971) 281–292.
[21] N. Shankar and V. Ramachandran, Efficient parallel circuits and algorithms for division, *Inform Process. Lett.* **29** (1988) 307–313.
[22] J. Simon, On feasible numbers, in: *Proc. 9th ACM Symp. Theory Comput.* (1977) 195–207.
[23] J. Simon, Division in idealized unit cost RAMs, *J. Comput. System Sci.* **22** (1981) 421–441.

[24] L. Stockmeyer and U. Vishkin, Simulation of parallel random access machines by circuits, *SIAM J. Comput.* **13** (1984) 409–422.
[25] J.L. Trahan, Instruction sets for parallel random access machines, Tech. Report UILU-ENG-88-2249 (ACT-99), Coordinated Science Laboratory, Univ. of Illinois (Ph.D. thesis) 1988.
[26] J.L. Trahan, V. Ramachandran and M.C. Loui, The power of parallel random access machines with augmented instruction sets, in: *Proc. 4th Structure in Complexity Theory Conf.*, (1989) 97–103.
[27] U. Vishkin, Implementation of simultaneous memory address access in models that forbid it, *J. Algorithms* **4** (1983) 45–50.