Pointers versus Arithmetic in PRAMs

Patrick W. Dymond Faith E. Fich
Naomi Nishimura
Prabhakar Ragde Walter L. Ruzzo

Technical Report 93-03-06
March, 1993

A preliminary version of this paper will appear in *Proceedings of the* 8^{th} *Annual IEEE Structure in Complexity Theory Conference*, San Diego, CA, May 1993.

Also available as:

- University of Waterloo Department of Computer Science Technical Report CS-93-21,
- York University Department of Computer Science Technical Report CS-93-01,

and via anonymous FTP from `cs.washington.edu` (`128.95.1.4`), file `tr/1993/03/UW-CSE-93-03-06.PS.Z`.

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Pointers versus Arithmetic in PRAMs*

Patrick W. Dymond
York University
Toronto, Ontario, Canada
dymond@cs.yorku.ca

Faith E. Fich
University of Toronto
Toronto, Ontario, Canada
fich@cs.toronto.edu

Naomi Nishimura
University of Waterloo
Waterloo, Ontario, Canada
nishi@maytag.waterloo.edu

Prabhakar Ragde
University of Waterloo
Waterloo, Ontario, Canada
plragde@maytag.waterloo.edu

Walter L. Ruzzo
University of Washington
Seattle, Washington, USA
ruzzo@cs.washington.edu

May 11, 1993

Abstract

Manipulation of pointers in shared data structures is an important communication mechanism used in many parallel algorithms. Indeed, many fundamental algorithms do essentially nothing else. A *Parallel Pointer Machine*, (or *PPM*) is a parallel model having pointers as its principal data type. PPMs have been characterized as PRAMs obeying two restrictions — first, restricted arithmetic capabilities, and second, the CROW memory access restriction (Concurrent Read, Owner Write, a commonly occurring special case of CREW). We present results concerning the relative power of PPMs (and other arithmetically restricted PRAMs) versus CROW PRAMs having ordinary arithmetic capabilities. First, we prove lower bounds separating PPMs from CROW PRAMs. For example, any step-by-step simulation of an n-processor CROW PRAM by a PPM requires time $\Omega(\log\log n)$ per step. Second, we show that this lower bound is tight — we give such a step-by-step simulation using $O(\log\log n)$ time per step. As a corollary, we obtain sharply improved PPM algorithms for a variety of problems, including deterministic context-free language recognition.

*Research supported by NSERC, the Information Technology Research Centre of Ontario, NSF Grant CCR-9002891 and NSF/DARPA Grant CCR-8907960. A portion of this work was performed while the first and last authors were visiting the University of Toronto, whose hospitality is gratefully acknowledged.

1 Introduction

Many sequential algorithms spend the bulk of their time doing pointer manipulation, as opposed to, say, arithmetic operations. Like their sequential counterparts, many PRAM algorithms spend a considerable proportion of their time manipulating pointers in global memory. Indeed, since interprocessor communication is so fundamental to most parallel algorithms, pointer manipulation in PRAMs may be even more pervasive than in RAMs. Despite the widespread use of pointer-based parallel data structures and algorithms, there has been little formal study of the power of this fundamental computing paradigm. Our paper addresses this issue.

The PRAM model in its various forms has achieved wide acceptance for use in expressing parallel algorithms. Nevertheless, the model is often criticized for being too powerful to correspond to realistic computer architectures. At this point the right (i.e., most useful) parallel model for bridging the gap between algorithms and architectures is still not settled [24]. This motivates further study of restrictions on the PRAM model, and the power of its arithmetic and addressing instructions.

Memory restrictions (e.g., CRCW versus CREW) have already been widely studied. One somewhat less well-known restriction is the CROW PRAM model, which further restricts CREW memory access by permitting only the *owner* of a global memory location to write there. Another class of restrictions focuses attention on pointer and addressing capabilities of the model, removing arithmetic.

We present two main results concerning the relative power of *Parallel Pointer Machines* (*PPMs*) or, equivalently, arithmetically restricted PRAMs, versus PRAMs having ordinary arithmetic capabilities. First, we prove lower bounds separating PPMs from CROW PRAMs. In particular, any step-by-step simulation of a CROW PRAM by a PPM requires time $\Omega(\log\log n)$ per step. Second, (to our surprise) this lower bound is tight. We give such a step-by-step simulation using $O(\log\log n)$ time per step. The lower bound holds even for strong, nonuniform PPMs, while the upper bound proof yields a simple uniform PPM algorithm. As a corollary, any problem solvable by a CROW PRAM in time $O(\log n)$ is also solvable by a PPM in time $O(\log n \log\log n)$ with a polynomial number of processors. Deterministic context-free language recognition is an example of such a problem. These problems were not previously known to be solvable by PPMs in less than $O(\log^2 n)$ time. Other results give tight upper and lower bounds on variants of the models, and prove a separation between CROW and CREW versions with otherwise identical features.

An additional reason for interest in our results lies in the novelty of the proof techniques. Many lower bounds for PRAMs are proved for "abstract" PRAMs, where there is no limit placed on the computation performed by a single PRAM instruction — any function $q : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ can be computed in one step. Since computation is "free" in this model, a lower bound of this form is a lower bound on the *communication* requirement of the problem, not its *computational* requirement. Such lower bounds certainly have the virtue of generality, but they are limited, *a priori*, by the fact that any function can be computed by an abstract PRAM in $\log_2 n$ steps. Understanding the interplay between computation and communication

is essential for obtaining better lower bounds. In this paper we take a modest step towards this difficult goal by proving a separation between PRAMs with restricted arithmetic capabilities, and ones with more normal arithmetic operations.

Below we outline prior work and our results in more detail.

CROW PRAMs: Dymond and Ruzzo [8] observe that most known Concurrent Read, Exclusive Write (CREW) PRAM algorithms guarantee write-exclusion by the simple stratagem of assigning an *owner* to each global memory cell and requiring that the owner of a memory cell be the only processor allowed to write into the cell. Furthermore, they characterize the power of such Concurrent Read, Owner Write, or CROW, PRAMs, showing that languages recognizable by CROW PRAMs in time $O(\log n)$ are precisely the languages that are logspace reducible to deterministic context-free languages (LOGDCFL). (This language class is known to lie somewhere between the better known classes DSPACE($\log n$) and AC^1.) It is important to note that these results apply to CROW PRAMs having a simple instruction set, basically including only indirect addressing, conditional branching, and addition. (To be precise, it is exactly the instruction set of the CREW PRAM of Fortune and Wyllie [14].) For definiteness, the term "CROW PRAM" below will refer to this model unless otherwise qualified.

It is interesting to note that similar but not identical notions of "ownership" have proven useful in practice in certain cache coherence protocols [1], and have appeared in earlier lower bound work [4, 13].

Pointer Machines: Pointer-based data structures are ubiquitous in sequential algorithms. One reason to study pointer-based computation is that useful lower bounds may be more easily obtained in such a structured model. For examples, see [23, 17, 2]. The *Storage Modification Machine (SMM)* or *Pointer Machine* is a formal model that captures the notion of sequential computation by pointer manipulation. Deep insight into the power of such machines is provided by Schönhage's demonstration of the equivalence of SMMs and *unit-cost successor RAMs*, i.e., ordinary unit cost RAMs stripped of all arithmetic capabilities except for the SUCCESSOR, or $+1$ operation [22].

The notion of parallel computation by pointer manipulation is formally captured by the PPM[1], studied by Dymond and Cook [3, 7, 5]. In brief, the model consists of a collection of finite state units, each with a fixed number of pointers to other units. Each unit can read the state of, and/or copy the pointers of, the units to which it points. Also, in each step, a unit may create and initialize a new unit. (See [7] or [5] for a more complete definition.)

Lam and Ruzzo [19] proved the equivalence of PPMs and a restricted version of the CROW PRAM, namely one stripped of arithmetic capabilities except for the SUCCESSOR ($+1$) and DOUBLE ($\times 2$) operations. Time and hardware resources of the two models, simultaneously, are the same to within a constant factor (for time bounds at least $\log n$). For definiteness, we refer to this restricted CROW PRAM as an rCROW. This characterization is central to our results, since it allows us to cast PPMs and CROW PRAMs in a common framework. Adding

[1] In the earlier papers, the PPM is called an HMM, or *Hardware Modification Machine*, by analogy to Schönhage's SMM. The PPM considered subsequently by Goodrich and Kosaraju [15] is a more complex model having both pointers and integer arithmetic.

$$AC^0 \subsetneq NC^1 \subseteq \text{DSPACE}(\log n) \subseteq \left\{ \begin{array}{c} \text{PPM}(\log n) \\ \| \\ \text{rCROW}(\log n) \end{array} \right\} \subseteq \left\{ \begin{array}{c} \text{LOGDCFL} \\ \| \\ \text{CROW}(\log n) \end{array} \right\} \subseteq \left\{ \begin{array}{c} AC^1 \\ \| \\ \text{CRCW}(\log n) \end{array} \right\}$$

Figure 1: Relationships among some parallel complexity classes

certain other simple unary functions such as those used in Section 4 to the set of arithmetic operations does not change the characterization.

Parallel Pointer Machines versus PRAMs: How powerful are Parallel Pointer Machines? A variety of parallel algorithms have been adapted to PPMs. As one important example, the "pointer doubling" technique of Fortune and Wyllie [14] has been used to show that DSPACE($\log n$) can be simulated by a PPM in time $O(\log n)$ [7, 5]. To relate pointer machines to PRAMs, it is not hard to see that a PRAM can perform a step-by-step simulation of a PPM, by maintaining the PPM's pointer structure in global memory. Of course, the characterization results cited above make the relationship between PPMs and PRAMs more concrete. Namely, for the simulation of a PPM by a PRAM, the PRAM can be made to obey the owner write constraint, and to use only SUCCESSOR and DOUBLE, rather than general addition. Furthermore, for PRAMs satisfying these two restrictions (i.e., rCROWs), a converse simulation by PPMs is possible.

The known relationships among the various complexity classes described above are summarized in Figure 1.

The open problem that motivated the present paper was the question of whether the simulation of rCROWs by Parallel Pointer Machines could be extended to the more general CROW PRAM model considered in [8]. Specifically, could a PPM simulate addition? On the one hand, "adding" two unrelated pointers seems difficult. On the other hand, by [8] it would suffice if one could do DCFL recognition on a PPM, and the DCFL recognition algorithm given in [8] is basically a generalization of the pointer doubling algorithm. Thus, it doesn't seem out of the question that one could show equality between PPMs and CROW PRAMs. However, our lower bounds show that this is impossible for time bounds below $\log \log n$, and also render it much less likely for larger time bounds — in particular, we show that it is impossible to obtain a linear time step-by-step simulation of general CROW PRAMs by several reasonably strong variants of the rCROW.

Arithmetically Restricted PRAMs: The essential weakness of the rCROW doesn't seem to lie in particular properties of SUCCESSOR and DOUBLE, but rather in the generic property that they are *unary* functions. Hence, for our lower bounds we generalize the rCROW model to allow computation of an *arbitrary* finite set of unary functions (of unbounded codomain). We also allow computation of arbitrary k-ary functions, provided their codomains are of size at most n. In particular, the latter subsumes arbitrary Boolean predicates. In addition, we allow the processors' local and global memory to be arbitrarily preinitialized. Finally, our model is nonuniform. To distinguish this model from the others we consider, we refer to it as an Arithmetically Restricted PRAM. CROW, CREW and CRCW

variants of it will be discussed. Thus, the rCROW (and hence the PPM) is a very simple special case of an Arithmetically Restricted CROW PRAM with conditional branch.

Our Lower Bounds: We consider a simple problem, the *pairing problem*, defined below. It is a key component of the DCFL recognition algorithm, and is easily solved in constant time by one processor on a CROW PRAM with addition. However, we show it is *not* solvable in constant time by variants of the Arithmetically Restricted PRAM, including those with preinitialized memory. Thus there can be no linear time step-by-step simulation of a general CROW PRAM, even when arbitrary precomputed tables are provided "for free." Specifically, we show the following three lower bounds.

- Without branch instructions, but with all the other facilities discussed above, an Arithmetically Restricted CREW PRAM requires time $\Omega(\log n)$ to solve the pairing problem, even with an unlimited number of processors.
- With branch instructions as well as the other facilities discussed above, an Arithmetically Restricted PRIORITY PRAM with p processors requires time $\Omega(\log(n^2/p))$ to solve the pairing problem. In particular, $\Omega(n^2)$ processors are necessary to solve it in constant time. Thus, even a strong form of concurrent write can't cheaply compensate for restricted arithmetic capabilities.
- With branch instructions as well as the other facilities discussed above, an Arithmetically Restricted CROW PRAM requires time $\Omega(\log\log n)$ to solve the pairing problem, even with an unlimited number of processors.

Our Upper Bounds: On the positive side, although we have given strong evidence that PPMs are not as strong as general CROW PRAMs, we can also show that PPMs are unexpectedly powerful. We show, using our upper bound for the pairing function, that they can do step-by-step simulations of general CROW PRAMs at a cost of $O(\log\log n)$ PPM steps per simulated step, while using only polynomially many processors. This implies our lower bound is tight for step-by-step simulations. Note that this upper bound holds for (uniform) PPMs, not just for (nonuniform) Arithmetically Restricted CROW PRAMs.

The Pairing Problem: The pairing problem is to compute any injective function

$$\pi : \{1, \ldots, n\} \times \{1, \ldots, n\} \to \mathbb{N}.$$

The natural pairing function $\pi(x, y) = (x-1)n + y$ with codomain $\{1, \ldots, n^2\}$ is an example, as is the function that concatenates the $\lceil \log_2 n \rceil$-bit binary representations of x and y. (The latter is the function used by our upper bound algorithm.) The pairing problem was motivated by the DCFL recognition algorithm of [8], a key component of which was accessing a two dimensional array. It is easy to see that a pairing function can be computed by one processor in constant time, given a simple precomputed table of multiples of n and an addition instruction. Conversely, if an Arithmetically Restricted CROW PRAM could solve the pairing problem, then it could simulate addition, by using the pairing function to access a precomputed table A such that $A[\pi(x, y)] = x + y$. Thus the pairing problem is "universal" for simulating binary operations by unary ones.

Outline: The rest of this paper is organized as follows. Section 2 defines the Arithmetically Restricted PRAM more fully. Sections 3 and 4 sketch our lower and upper bounds, respectively.

2 The Arithmetically Restricted PRAM

We consider PRAMs with an infinite shared global memory ($M[1], M[2], \ldots$) and p processors $P_1, \ldots, P_p$ that each have an infinite private local memory ($L[1], L[2], \ldots$). Each (global and local) memory cell can hold one nonnegative integer of arbitrary size. Each processor also has an *accumulator* that initially contains the processor's number. For convenience, we call the accumulator $L[0]$. The inputs defining the problem instance being solved are initially located in the first appropriately many global memory cells. Unless otherwise indicated, all other local and global memory cells are assumed to be initialized to 0. When the computation terminates, the outputs for the problem are the values contained in the first appropriately many global memory cells.

Let F be a fixed, but arbitrary, finite set of unary functions and let $C \subseteq \mathbb{N}$ be an arbitrary set of constants. Let Q_k be an arbitrary set of k-ary functions $q : \mathbb{N}^k \to \{1, \ldots, n\}$. The essential feature of these functions is that their ranges are not too large; the specific choice of $\{1, \ldots, n\}$ for the codomain is unimportant, as function values can be renamed by subsequently applying a unary function. At each step of the computation, each processor can perform one of the operations listed in Table 1. A program is a finite sequence of such instructions.

Throughout this paper, we assume that processors are allowed to simultaneously read from the same global memory cell. If two or more processors are allowed to simultaneously write to the same global memory cell, then the PRAM is concurrent-read, concurrent-write (CRCW); otherwise it is concurrent-read, exclusive-write (CREW). When concurrent writes are allowed, a method for resolving write conflicts must be specified. A PRIORITY PRAM resolves conflicts in favor of the lowest numbered processor attempting to simultaneously write into a cell. It is at least as powerful as most other CRCW PRAMs. A ROBUST PRAM resolves write conflicts in a completely arbitrary way — i.e., no assumption may be made about the final value in a cell at which a write conflict occurred. It is weaker than most other CRCW PRAMs. For more details, see [12], [9], [16], [11].

A concurrent-read, owner-write (CROW) PRAM [8] is a CREW PRAM in which each global memory cell is *owned* by a single processor; only the owner of a global memory cell may write to it. Note that processors may own many different global memory cells.

In addition to the various concurrent write mechanisms, we also will consider certain extensions to the basic model. The first extension is *preinitialized memory*. In a PRAM with preinitialized memory, except for each processor's accumulator and those global memory cells that contain the input values, programs may specify initial values for local and global memory cells. These values can be arbitrary, but cannot depend on the input values. This is an interesting extension to consider since the pairing problem may be a frequently executed

READ	$L[0]$	$\leftarrow$	$M[L[0]]$	indirect read from global memory
LREADi	$L[0]$	$\leftarrow$	$L[i]$	direct read from local memory
WRITE	$M[L[1]]$	$\leftarrow$	$L[0]$	indirect write into global memory
LWRITEi	$L[i]$	$\leftarrow$	$L[0]$	direct write into local memory
LOAD-c	$L[0]$	$\leftarrow$	c	assign a predefined constant $c \in C$
f	$L[0]$	$\leftarrow$	$f(L[0])$	apply a unary function $f \in F$
q	$L[0]$	$\leftarrow$	$q(L[0], \ldots, L[k-1])$	evaluate a k-ary function $q \in Q_k$

Table 1: Arithmetically restricted PRAM: Basic instructions

BRANCH	if $L[0] > 0$ then goto ...			conditional branch
CONDITIONAL-f	if $L[1] > 0$ then $L[0] \leftarrow f(L[0])$			conditional function application $(L[1] \in \{0,1\})$
k-CONCATENATE	$L[0]$	$\leftarrow$	$L[0] \cdot 2^k + L[1]$	k-bit concatenation $(L[1] < 2^k)$
LREAD	$L[0]$	$\leftarrow$	$L[L[0]]$	indirect read from local memory
LWRITE	$L[L[1]]$	$\leftarrow$	$L[0]$	indirect write to local memory

Table 2: Arithmetically restricted PRAM: Extensions

subroutine in a larger computation whose total cost dominates the cost of precomputing tables used by the pairing subroutine.

Additional extensions arise by allowing one or more of the operations listed in Table 2, and described below.

The BRANCH instruction changes flow of control. If this operation is allowed, a processor's program can be viewed as a computation tree with at most 2^t nodes at distance t from the root. Conditional function application provides a much more restricted form of branching. Here $L[1]$ must contain either 0 or 1 and, in the latter case, the unary function $f \in F$ is applied to the value in $L[0]$.

The k-CONCATENATE instruction requires that the second argument $L[1]$ contain a number that is at most k bits in length. In this case, the concatenation is performed with the second argument treated as a k-bit number by adding leading zeros, if necessary.

The last extension considered is indirect addressing of local memory. When the address argument $L[0]$ of LREAD or $L[1]$ of LWRITE is restricted to be a positive integer no larger than k, we say that the indirect addressing of local memory is *k-limited*. In this case, one of the local memory cells $L[1], \ldots, L[k]$ of the processor will be accessed.

3 Lower Bounds

In this section we prove lower bounds on the pairing problem, defined in Section 1. Initially, $M[1]$ and $M[2]$ each contain a value in the range $\{1, \ldots, n\}$. Call these values x and y,

respectively. At the end of the computation, the value in $M[1]$ must be an injective function of x and y. All three of our lower bounds are tight, as will be shown in Section 4.

The lower bound proof technique was partly inspired by results of Dymond concerning sequential RAMs [6], but the PRAM case is substantially more difficult. Throughout this section, let $V(0, j, x, y, t)$ denote the value in global cell $M[j]$ and let $V(i, j, x, y, t)$ denote the value in cell $L[j]$ of processor P_i at the end of step t when x and y are the inputs to the pairing problem. Here $x, y \in \{1, \ldots, n\}$, $t \in \mathbb{N}$, $i \in \{1, \ldots, p\}$, $j \in \mathbb{N}$ and $j \neq 0$ if $i = 0$. (For simplicity, the latter condition is usually omitted from statements below.)

Without instructions that change the flow of control (BRANCH and CONDITIONAL-f), the instruction that a processor performs at each step does not depend on the values of the inputs. We exploit this in the next theorem.

Theorem 1 *An Arithmetically Restricted CREW PRAM requires $\Omega(\log n)$ steps to solve the pairing problem, even with preinitialized memory.*

Proof: A global memory cell, local memory cell, or accumulator is *oblivious* at time t if it contains the same value at the end of step t for all $x, y \in \{1, \ldots, n\}$. Otherwise it is *affected* at time t. The set of values appearing in affected cells during the first t steps of the computation is

$$A_t = \{V(i, j, x, y, t') \mid i \in \{0, 1, \ldots, p\},\ j \in \mathbb{N},\ x, y \in \{1, \ldots, n\},\ t' \in \{0, \ldots, t\},\ \text{and}\ V(i, j, x, y, t') \neq V(i, j, x', y', t') \text{ for some } x', y' \in \{1, \ldots, n\}\}.$$

Let a_t denote the size of this set.

Initially, $A_0 = \{1, \ldots, n\}$, hence $a_t = n$.

Next we wish to show $a_{t+1} \leq (|F| + 3)a_t$.

Clearly $A_t \subseteq A_{t+1}$. Consider the instructions executed by the processors at step $t + 1$.

If a predefined constant is loaded into a processor's accumulator, then the accumulator is oblivious at time $t + 1$. Similarly, if the processor applies a unary function $f \in F$ and the accumulator is oblivious at time t, then it is also oblivious at time $t + 1$. Now suppose that the accumulator is affected at time t, so, for any input, at time t, it contains a value in A_t at time t. Then, at time $t + 1$, it contains a value in $\{f(a) \mid a \in A_t\}$ if the processor applied the unary function $f \in F$. Furthermore, if a processor applies a function whose codomain is a subset of $\{1, \ldots, n\}$, then the value in the accumulator at time $t + 1$ is in A_0.

A direct write to or read from local memory does not add any new values to A_{t+1}, although it may increase the number of affected memory cells.

When an indirect read from global memory is performed by processor P_i, its accumulator, $L[0]$, contains, at time t, the address from which to read and, at time $t + 1$, the value read. If the accumulator is oblivious at time t, then it is affected at time $t + 1$ if and only if the cell read, $M[L[0]]$, is affected at time t and, if so, the value read is in A_t. However, if the accumulator is affected at time t, then, for any input, the address of the cell from which to read is in A_t and, hence, the value read is in $\{V(0, a, x, y, t) \mid a \in A_t, x, y \in \{1, \ldots, n\}\}$.

When an indirect write to global memory is performed by processor P_i, its accumulator, $L[0]$, contains the value to be written and its local memory cell $L[1]$ contains the address to which to write. If $L[1]$ is oblivious at time t, then $M[L[1]]$ is affected at time $t+1$ if and only if $L[0]$ is affected at time t and, if so, the value written is in A_t. Now suppose that $L[1]$ is affected at time t. Then the set of locations to which P_i writes during step $t+1$ is a subset of A_t. Note that there can be at most a_t such processors; otherwise, by the pigeonhole principle, a write conflict will occur. If $L[0]$ is oblivious at time t, then P_i writes the same value during step $t+1$ for all values of x and y, whereas, if $L[0]$ is affected at time t, the value written is in A_t. Let B_t be the set of indices $i \in \{1, \ldots p\}$ of processors P_i such that P_i writes to global memory during step $t+1$, its accumulator $L[0]$ is oblivious at time t, and its local memory cell $L[1]$ is affected at time t. Then, at time $t+1$, the set of values in affected global memory cells is a subset of

$$\begin{aligned} A_t \quad &\cup \quad \{V(0,a,x,y,t) \mid a \in A_t,\ x,y \in \{1,\ldots,n\}\} \\ &\cup \quad \{V(i,0,x,y,t) \mid i \in B_t,\ x,y \in \{1,\ldots,n\}\}. \end{aligned}$$

Furthermore, the last of these three sets has cardinality at most a_t.

Note that if $M[a]$ is affected at time t, then $V(0,a,x,y,t) \in A_t$ for all $x,y \in \{1,\ldots,n\}$ and if $M[a]$ is oblivious at time t, then $V(0,a,x,y,t)$ has the same value for all $x,y \in \{1,\ldots,n\}$. Thus

$$\begin{aligned} &\{V(0,a,x,y,t) \mid a \in A_t,\ x,y \in \{1,\ldots,n\}\} \quad \subseteq \\ &\qquad A_t \cup \{V(0,a,x,y,t) \mid a \in A_t,\ x,y \in \{1,\ldots,n\} \text{ and } M[a] \text{ is oblivious at time } t\,\}. \end{aligned}$$

The latter set has cardinality at most a_t. It follows that

$$\begin{aligned} A_{t+1} \quad &\subseteq \quad A_t \cup \{f(a) \mid a \in A_t,\ f \in F\} \\ &\cup \quad \{V(0,a,x,y,t) \mid a \in A_t,\ x,y \in \{1,\ldots,n\} \text{ and } M[a] \text{ is oblivious at time } t\,\} \\ &\cup \quad \{V(i,0,x,y,t) \mid i \in B_t,\ x,y \in \{1,\ldots,n\}\,\}, \end{aligned}$$

so $a_{t+1} \leq (|F|+3)a_t$.

It is easy to verify by induction that $a_t \leq n(|F|+3)^t$, for all $t \geq 0$. The cell containing the answer at the end of the computation has a different answer for each of the n^2 different pairs of inputs and thus the number of different values appearing in affected memory cells during the computation must be at least n^2. Hence, the number of steps in the computation must be in $\Omega(\log n)$. □

Theorem 2 *An Arithmetically Restricted PRIORITY PRAM with p processors requires $\Omega(\log(n^2/p))$ steps to solve the pairing problem, even with preinitialized memory and the ability to branch.*

Proof: If $1 \leq p \leq n$, then $2 \log n \geq \log(n^2/p) \geq \log n$. Thus it suffices to prove the result when $p \geq n$.

Let V_t be the set of values that appear in the processors' accumulators during the first t steps of the computation, i.e.,

$$V_t \;=\; \{V(i,0,x,y,t') \mid i \in \{1,\ldots,p\},\ x,y \in \{1,\ldots,n\},\ t' \in \{0,\ldots,t\}\}.$$

Let v_t denote the cardinality of this set. Recall that $V(i,0,x,y,0) = i$, so $v_0 = p$ and $\{1,\ldots,n\} \subseteq V_0$.

Next we argue that $v_{t+1} \leq v_t(1 + |F| + 1) + p2^t$.

The values in a processor's accumulator can change only as a result of the evaluation of a function, a read, or the assignment of a predefined constant.

There are at most v_t different values that can appear in the accumulators during the first t steps and at most $|F|$ different values can result from each by applying the functions in F. After a processor evaluates a function with codomain $\{1,\ldots,n\}$, its accumulator contains a value in $V_0 \subseteq V_t$.

When a processor performs a write to local or global memory, the value in its accumulator is written. Hence, at the end of step t, the value in each memory cell is either its initial value or a value in V_t. Except for $M[1]$ and $M[2]$, whose initial values are contained in $\{1,\ldots,n\} \subseteq V_0$, each memory cell has the same initial value for all inputs. Thus at most one new value is obtained from each local or global memory cell that can be read during step $t+1$. Furthermore, since the global memory locations from which processors read are specified by the contents of their accumulators, there are at most v_t different global memory cells that can be read during step $t+1$.

As a result of branches, each of the p processors can be in one of at most 2^t states. In each such state, it might read the initial value of a (directly addressed) local memory cell or use a new predefined constant $c \in C$ (but not both).

Thus $v_{t+1} \leq v_t(1 + |F| + 1) + p2^t$. It is easy to verify by induction that $v_t \leq 2p(|F| + 2)^t$ for all $t \geq 0$. Since the PRAM must give a different answer for each of the n^2 different pairs of inputs, and a value cannot be written to global memory unless it appears in an accumulator, it follows that n^2 different values must appear in the accumulators during the course of the computation. Hence, the number of steps in the computation must be in $\Omega(\log(n^2/p))$. □

In fact, this proof works for any CRCW PRAM in which the result of a write conflict leaves the cell unchanged or causes one of the values being written there to appear. The MAXIMUM PRAM [9] is an example of such a model. Clearly, the lower bound does not apply to a PRAM in which the value that appears as the result of a write conflict is the sum of the values written.

Theorem 3 *An Arithmetically Restricted CROW PRAM requires* $\Omega(\log\log n)$ *steps to solve the pairing problem, even with preinitialized memory and the ability to branch.*

Proof: We say that a processor P_i *could know* a value at time t if the value is an input or the value appears in the processor's accumulator $L[0]$ during the first t steps of computation,

for some choices of the inputs. Then for any subset of processors $\{P_i \mid i \in S\}$, the set of values that processors in S could know at time t is

$$K(S,t) \quad = \quad \{V(i,0,x,y,t') \mid i \in S,\ x,y \in \{1,\ldots,n\},\ t' \in \{0,\ldots,t\}\} \cup \{1,\ldots,n\}.$$

Let $k(s,t)$ denote the maximum cardinality of this set, taken over all s-processor subsets S.

Initially, each processor's accumulator contains its number; thus

$$K(S,0) \subseteq \{1,\ldots,n\} \cup S$$

so $k(s,0) \leq n+s$.

Consider the instructions executed at step $t+1$ by the processors in some set S. Writes to local or global memory do not change the values a processor could know. Evaluating functions with codomain $\{1,\ldots,n\}$ produces values that have already been accounted for. As a result of branches, each of the processors in S can be in one of at most 2^t states. In each such state, it might read the initial value of a local memory cell or use a new predefined constant $c \in C$ (but not both). Note that any value in a processor's local memory cell after step t is either the initial value of that cell or was in its accumulator at some earlier time and, hence, could be known by the processor. Thus assignment of predefined constants and direct reads of local memory account for at most $s2^t$ new values that processors in S could know at time $t+1$.

There are at most $k(|S|,t)$ different values that could be known by processors in S at time t and hence that could be in those processors' accumulators. At most $|F|$ different values can result from each by applying the functions in F, for a total of $|F| \cdot k(|S|,t)$ new values.

Furthermore, there are at most $k(|S|,t)$ different global memory cells that can be read during step $t+1$ by processors in S. Any value in a global memory cell is either the initial value of that cell or a value that was written there by the processor that owns the cell. Except for $M[1]$ and $M[2]$, whose initial values are contained in $\{1,\ldots,n\}$, each memory cell has a single initial value. The only values that could have been written to these global memory cells during the first t steps are the at most $k(k(|S|,t),t)$ different values that could have been known at time t by the set of at most $k(|S|,t)$ processors that own these cells. Therefore, altogether, these global memory cells could contain at most $k(|S|,t) + k(k(|S|,t),t)$ different values at the end of step t.

Thus $k(s,t+1) \leq k(s,t) + s2^t + |F| \cdot k(s,t) + k(s,t) + k(k(s,t),t)$ for $t > 0$. It is easy to verify by induction that $k(s,t) \leq (n+s)(3+|F|)^{3^t}$. In particular, $k(1,t) \in n2^{2^{O(t)}}$. At the end of the computation, the value in the output cell is either the value of the input x or a value written by the processor P that owns this cell and, hence, a value that P could know. Since there must be at least n^2 different values that P could know, the number of steps in the computation must be in $\Omega(\log\log n)$. □

4 Upper Bounds

In this section we present upper bounds for the pairing problem using Arithmetically Restricted PRAMs with different instruction sets. With the exception of Theorem 14, they are mainly important in showing that the lower bounds proved in the previous section are tight.

```
L[0] ← M[2]                                  Get y.
do ⌈log n⌉ times
   M[1] ← DOUBLE(M[1])                       Concatenate each of 0 and 1 to x.
   M[2] ← SUCCESSOR(M[1])
   L[1] ← SUCCESSOR(MOD_2(L[0]))             Use the least significant bit of y to
   M[1] ← M[L[1]]                            choose between these two alternatives.
   L[0] ← DIV_2(L[0])                        Delete the least significant bit of y.
```

Figure 2: Pairing on an Arithmetically Restricted PRAM with one processor

In the interest of simplicity, the code fragments presented in this section are not given in full detail. In particular, we often omit motion of constants and data to or from the accumulator, especially via direct addressing.

Obviously, using $\lceil \log n \rceil$-CONCATENATE, a single processor can solve the pairing problem in constant time by concatenating x and y.

$M[1] \leftarrow \lceil \log n \rceil\text{-CONCATENATE}(M[1], M[2])$.

If $\lceil \log n \rceil$-CONCATENATE is not available, it can be replaced by 1-CONCATENATE using the following (slower) sequence of code. The idea is that the bits of the second argument are pulled off one at a time and concatenated to the end of the first argument. The resulting program solves the pairing problem in $O(\log n)$ time using one processor.

```
do ⌈log n⌉ times
   L[2] ← MOD_2(L[1])
   L[1] ← DIV_2(L[1])
   L[0] ← 1-CONCATENATE(L[0], L[2])
```

(Here MOD_k and DIV_k are the unary functions that return the remainder and quotient, respectively, when their arguments are divided by k. Note, for use later, that this *reverses* the bits of the second argument.)

More interestingly, none of the extended features of the Arithmetically Restricted PRAM are necessary to achieve this result — indirect addressing into global memory can be used instead of 1-CONCATENATE, as shown in the next theorem.

Theorem 4 *Using only a small finite set of unary functions and without preinitialized memory, one processor can solve the pairing problem in $O(\log n)$ time.*

Proof: See Figure 2. □

One implication of this result is that the $\Omega(\log n)$ lower bound in Theorem 1 is the best possible, as is the $\Omega(\log(n^2/p))$ lower bound in Theorem 2 for $p = O(n)$. For $p = \Theta(n^2)$, the lower bound in Theorem 2 is also tight, as shown in Theorem 5.

Theorem 5 *An Arithmetically Restricted ROBUST PRAM with n^2 processors can solve the pairing problem in constant time.*

Proof: The idea is to view each processor number in $\{1, \ldots, n^2\}$ as a distinct ordered pair $\langle i, j \rangle \in \{1, \ldots, n\} \times \{1, \ldots, n\}$. Processors compare the two parts of their processor numbers with x and y using the ternary predicate $q(x, y, \langle i, j \rangle)$ which equals 1 if and only if $x = i$ and $y = j$. There is a unique processor P_r for which $q(x, y, r) = 1$. This processor writes its number, as the answer, to $M[1]$. All other processors write their numbers to $M[2]$, a location whose contents we do not care about. In short, each processor P_r executes the following.

$M[2 - q(x, y, r)] \leftarrow r$

□

The same result holds on an Arithmetically Restricted ROBUST PRAM having only a binary Boolean operation such as AND in place of the ternary predicate q used above, although the details are more complex.

Using branching, or even conditional function application, an Arithmetically Restricted CREW PRAM can avoid the concurrent write used in the algorithm presented in the proof above.

Theorem 6 *An Arithmetically Restricted CREW PRAM with n^2 processors can solve the pairing problem in constant time using conditional function application.*

Proof: The code used in the previous proof is replaced by the following, which causes every processor P_r for which $q(x, y, r) \neq 1$ to write to a distinct location, namely $r + 1$, in the last step. As before, the desired processor writes its number into $M[1]$.

$L[0] \leftarrow 1 - q(x, y, r)$
if $L[0] > 0$ then $L[0] \leftarrow r$
$L[0] \leftarrow \text{SUCCESSOR}(L[0])$
$M[L[0]] \leftarrow r$

□

By Theorem 3, the result of Theorem 6 cannot be strengthened from CREW to CROW. Thus, CREW and CROW PRAMs with this instruction set are provably different in power.

Using the following result, the previous upper bound also holds when either 2-limited indirect addressing of local memory or 1-CONCATENATE is available instead of conditional function application.

Theorem 7 1-CONCATENATE, *conditional function application, and 2-limited indirect addressing of local memory are equivalent instructions, to within constant factors.*

Proof: The 1-CONCATENATE instruction can easily be simulated in constant time using conditional function application, as shown in Figure 3.

Conditional function application can be simulated in constant time using 2-limited indirect addressing of local memory, as demonstrated in Figure 4. The idea is to apply the function unconditionally and then choose between the original and resulting values.

$L[0]$	$\leftarrow$ DOUBLE$(L[0])$	Shift $L[0]$ one bit
if $L[1] = 1$ then $L[0]$	$\leftarrow$ SUCCESSOR$(L[0])$	Conditionally change low order bit from 0 to 1

Figure 3: Simulating 1-CONCATENATE

$L[3]$	$\leftarrow$ SUCCESSOR$(L[1])$	$L[3]$ has value 1 or 2
$L[1]$	$\leftarrow L[0]$	
$L[2]$	$\leftarrow f(L[0])$	
$L[0]$	$\leftarrow L[L[3]]$	Choose between $L[1]$ and $L[2]$

Figure 4: Simulating conditional function application

$L[3]$	$\leftarrow M[2i]$	Temporarily save the values
$L[4]$	$\leftarrow M[2i+1]$	in the global memory cells.
$M[2i]$	$\leftarrow L[1]$	Move the necessary values from
$M[2i+1]$	$\leftarrow L[2]$	local to global memory.
$L[0]$	$\leftarrow$ 1-CONCATENATE$(i, L[0])$	Concatenate the first argument to the end of the processor number, i.
$L[0]$	$\leftarrow M[L[0]]$	Determine the answer using (indirect) read from global memory.
$M[2i]$	$\leftarrow L[3]$	Restore the global memory cells.
$M[2i+1]$	$\leftarrow L[4]$	

Figure 5: Simulating limited indirect addressing

Furthermore, 1-CONCATENATE can simulate 2-limited indirect addressing of local memory. The idea is for processor P_i to temporarily use the global memory cells $M[2i]$ and $M[2i+1]$ in place of its local memory cells $L[1]$ and $L[2]$. See Figure 5.

□

Finally, we note that none of the three lower bounds holds when other restrictions on the model mentioned in Section 2 are relaxed. Clearly, allowing a binary function with a quadratic (or even superlinear) range would cause problems. The restriction that the set of unary operations F has constant size is also necessary to obtain our lower bounds. It is not even sufficient that each processor only use one different unary operation. For example, suppose processor P_i, $i \in \{1, \ldots, n\}$, is given the unary function f_i that adds $n(i-1)$ to its argument. Then the following Arithmetically Restricted CROW PRAM program solves the pairing problem in constant time using only n processors.

	P_1	$P_i,\ i \neq 1$
Step 1	$L[0] \leftarrow M[2]$	$M[i] \leftarrow f_i(M[1])$
Step 2	$M[1] \leftarrow M[L[0]]$	

Note that, after the first step, $M[i] = x + n(i-1)$ for all $i \in \{1, \ldots, n\}$. Similarly, if unlimited indirect addressing of local memory is allowed together with preinitialized local memory, an Arithmetically Restricted CROW PRAM can solve the pairing problem in constant time using only n processors, by giving processor P_i a preinitialized table of the unary function f_i.

We now turn to our upper bound for the pairing problem on CROW PRAMs.

The key idea for solving the pairing problem in $O(\log\log n)$ time comes from solving a different problem: forming an integer from its bits. Specifically, the k-JOIN problem is to concatenate k bits into an integer in the range $\{0, \ldots, 2^k - 1\}$.

Lemma 8 *An Arithmetically Restricted CROW PRAM with $2^{k+1} - 1$ processors can solve the k-JOIN problem in $O(\log k)$ time.*

Proof: We first solve a related problem, that of concatenating a high-order 1-bit together with the k input bits, producing an integer in the range $\{2^k, \ldots, 2^{k+1} - 1\}$. The idea is to view the first $2^{k+1} - 1$ global memory cells as an implicit balanced binary decision tree such that, for $d = 0, \ldots, k-1$, all of the nodes at depth d are labeled with the $(d+1)$st input variable, and the leaf nodes contain the function values for this related problem. In constant time, each processor P_i, $1 \leq i \leq 2^k - 1$, creates a pointer from the i^{th} internal node to either its left child or its right child, depending on whether the input variable labeling the i^{th} node is 0 or 1. If the i^{th} node is at depth d, this is done as follows.

$M[i] \leftarrow 1\text{-CONCATENATE}(i, M[d+1])$

It turns out that the function value to be stored in the leaf at address i is i itself. Thus, each processor P_i, $2^k \leq i \leq 2^{k+1} - 1$ can initialize its leaf as follows.

$M[i] \leftarrow i$

Pointer jumping can then be used to determine the answer in $1 + \lceil \log_2 k \rceil$ steps. Specifically, each processor P_i corresponding to an internal node performs the operation

$M[i] \leftarrow M[M[i]]$

$1 + \lceil \log_2 k \rceil$ times. Finally, to solve the k-JOIN problem, processor 1 applies the MOD_{2^k} function to remove the unwanted high-order bit from the answer constructed above:

$M[1] \leftarrow \text{MOD}_{2^k}(M[1])$

(Recall that we are assuming a nonuniform model, so the available unary functions, e.g. MOD_{2^k}, are allowed to depend on k. We will consider uniform versions below.)

Note that the only processor to write into $M[i]$ is P_i, i.e., the algorithm obeys the owner write restriction with P_i owning $M[i]$. □

Using standard techniques [18], the number of processors can be improved to $2^k/k^{O(1)}$, while only increasing the time by a constant factor. The idea is to apply the foregoing algorithm to

For all P_i, $l \leq i \leq l2^{k+1} - 1$

if $\text{EQUAL}_k(\text{D}(i)) = 0$	Initialize tree:
then $M[i] \leftarrow 1\text{-CONCATENATE}(i, M[\text{B}(i)])$	Internal node;
else $M[i] \leftarrow i$	Leaf.
do $1 + \lceil \log_2 k \rceil$ times	Pointer jumping
$M[i] \leftarrow M[M[i]]$	
if $\text{D}(i) = 0$ then $M[\text{T}(i)] \leftarrow \text{MOD}_{2^k}(M[i])$	Extract and move answer.

Figure 6: The k-JOIN algorithm

only the high order $k - O(\log k)$ bits, then sequentially concatenate the remaining $O(\log k)$ bits.

Any function with domain $\{0,1\}^k$ can be expressed as the composition of a unary function and k-JOIN. Thus we have the following result.

Corollary 9 *Any function with domain $\{0,1\}^k$ can be computed by an Arithmetically Restricted CROW PRAM with $2^{k+1} - 1$ processors in $O(\log k)$ time.*

This result is within a constant factor of optimal, since even on a CREW PRAM with an unlimited number of processors and an arbitrarily powerful instruction set, computing the OR of n Boolean values requires $\Omega(\log n)$ steps [4].

It is also possible to solve multiple instances of the k-JOIN problem in parallel, although determining which cell a processor should access is somewhat more complicated.

Lemma 10 *Any l independent instances of the k-JOIN problem can be solved by an $l2^{k+1} - 1$ processor Arithmetically Restricted CROW PRAM in time $O(\log k)$.*

Proof: Root l binary trees at locations $M[l], \ldots, M[2l-1]$, again viewing $M[2i]$ and $M[2i+1]$ as the children of $M[i]$. Thus, for $d = 0, \ldots, k$, the $l2^d$ nodes of depth d are located at cells $M[l2^d], \ldots, M[l2^{d+1} - 1]$. In other words, location $M[i]$, for $l \leq i \leq l2^{k+1} - 1$, contains a node at depth $\text{D}(i) = \lfloor \log_2(i/l) \rfloor$ in the tree numbered $\text{T}(i) = \lfloor i/2^{\text{D}(i)} \rfloor + 1 - l$. To determine where it should point, processor P_i, $l \leq i \leq l2^k - 1$, has to read the input bit numbered $\text{B}(i) = k(\text{T}(i) - 1) + 1 + \text{D}(i)$. Note that for $l \leq i \leq l2^{k+1} - 1$, we have $0 \leq \text{D}(i) \leq k$, $1 \leq \text{T}(i) \leq l$, and for $l \leq i \leq l2^k - 1$, we have $1 \leq \text{B}(i) \leq kl$. The unary functions EQUAL_k, which has value 1 when its argument is k and 0 otherwise, and MOD_{2^k}, which computes the remainder when its argument is divided by 2^k, are also used in the program, which is executed by all processors P_i with $l \leq i \leq l2^{k+1} - 1$. See Figure 6. Again, note that the algorithm obeys the owner write restriction since the only processor to write into $M[i]$ is P_i. □

The k-SPLIT problem is the inverse of k-JOIN, i.e., to break an integer in the range 0 to $2^k - 1$ into a sequence of k bits.

```
M[i] ← ENCODE_K(M[i])                                   P_i, 1 ≤ i ≤ l
for j ← 1, ..., log_2 K do                              P_i, 1 ≤ i ≤ l · 2^j
  if MOD_2(i) = 1
    then M[i] ← LEFT(M[⌊(i − 1)/2⌋ + 1])
    else M[i] ← RIGHT(M[⌊(i − 1)/2⌋ + 1])
M[i] ← MOD_2(M[i])                                      P_i, 1 ≤ i ≤ lK
```

Figure 7: The k-SPLIT algorithm

Lemma 11 *An Arithmetically Restricted CROW PRAM with $O(lk)$ processors can solve l instances of the k-SPLIT problem simultaneously, in time $O(\log k)$.*

Proof: We use a number of unary functions, including ENCODE_K, LEFT, and RIGHT. ENCODE_K adds 2^K to its argument, where $K = 2^{\lceil \log_2 k \rceil}$. This leading 1 is necessary to keep track of the number of leading 0's in the original argument, when viewed as a K-bit string. LEFT returns the left half of its argument (when viewed as a bit string) and RIGHT returns the right half of its argument prepended by a 1. Specifically, if $|z_1| = |z_2|$, then $\text{LEFT}(1z_1z_2) = 1z_1$ and $\text{RIGHT}(1z_1z_2) = 1z_2$.

The method for solving one instance of k-SPLIT is simple. First we apply ENCODE_K to the input. Then in each step $j = 1, \ldots, \log_2 K$, we have 2^j processors use LEFT or RIGHT to replace the first 2^{j-1} global memory words by 2^j words of half the length. Finally, we apply MOD_2 to remove the leading 1 from each word. The method easily generalizes so that l integers in the range 0 to $2^k - 1$ can be broken into l sequences of k bits, in $O(\log k)$ steps, using lK processors. See Figure 7.

Note that if k is not a power of 2, there will be extraneous zeros between the bit strings, namely the $K - k$ leftmost bits within each block of K. These are easy to remove if desired: the i^{th} bit of the final answer, $1 \le i \le lk$, is the j^{th} bit computed by the procedure above, where

$$j = \left\lfloor \frac{i-1}{k} \right\rfloor \cdot K + \text{MOD}_k(i-1) + K - k + 1.$$

Thus, the extraneous zeros can be removed by having processor P_i, $1 \le i \le lk$, execute $M[i] \leftarrow M[j]$.

Again, note that the algorithm obeys the owner write restriction since the only processor to write into $M[i]$ is P_i. □

Combining Lemmas 10 and 11 gives us a fast way to solve n instances of the pairing problem simultaneously in $O(\log \log n)$ time on an Arithmetically Restricted CROW PRAM.

Lemma 12 *An Arithmetically Restricted CROW PRAM with n^3 processors can solve n instances of the pairing problem simultaneously in $O(\log \log n)$ time.*

Proof: Let $k = \lceil \log_2 n \rceil$. First, subtract 1 from each input so that it is an integer in the range $0, \ldots, 2^k - 1$. Then perform $2n$ simultaneous instances of k-SPLIT. Finally, perform n simultaneous instances of $2k$-JOIN. □

It may seem counterintuitive that the best way to concatenate two bit strings is by first breaking each into a sequence of bits, particularly since concatenation can be performed by a decision tree of depth two. The difficulty is that each internal node of this decision tree has fanout n (corresponding to the n different potential values x and y). Selecting the appropriate edge out of each internal node is as hard as our original problem. When there are only two (or any constant number of) choices at each node, the selection is easy to perform.

Recall that in contrast to the nonuniform Arithmetically Restricted CROW PRAM considered throughout most of the foregoing, the rCROW is a uniform model with a specific, limited instruction set, mainly having the SUCCESSOR and DOUBLE instructions. It is natural to ask whether the CROW k-SPLIT, k-JOIN, and pairing algorithms developed in Lemmas 8, 10, 11, and 12 can be made uniform, or even more strongly, can be made to run on an rCROW. The answer is a qualified "yes" — both are possible, with the qualification that uniformity comes at the expense of some precomputation, as we explain below.

First, note that several aspects of the algorithms above are already uniform. Namely, all processors execute the same program, no preinitialized memory is required, and many of the unary functions available to each processor, specifically the set

$$F_1 = \{\text{MOD}_2,\ \text{DIV}_2,\ \text{LEFT},\ \text{RIGHT},\ \text{PREDECESSOR}\},$$

are simple, and independent of the input. However, other aspects are nonuniform. In particular, the functions in the set

$$F_2 = \{\text{MOD}_{2^k},\ \text{EQUAL}_k,\ \text{D},\ \text{T},\ \text{B},\ \text{ENCODE}_K\}$$

all depend on the parameters l, k, or n defining the problem being solved. An additional mild source of nonuniformity is that each algorithm begins execution with a number of active processors that is a function of the input size, e.g., n^3 in Lemma 12.

To construct uniform versions of the algorithms, we replace the set of unary operations $F = F_1 \cup F_2$ above by indirect addressing into suitable (uniformly) precomputed tables stored in global memory. Note that the algorithms in Lemmas 8, 10, 11, and 12 use no multivariate functions or predicates, so indirect addressing suffices to simulate all operations (except, of course, SUCCESSOR and DOUBLE, which are necessary for constructing the tables). In the course of constructing these tables, we will coincidentally activate the correct number of processors. (The rCROW, as defined in [19], as well as the PPM, as defined in [3, 7, 5], are *forking* models. That is, there is only one initially active processor; others are activated by FORK instructions. At most 2^t processors can be active within the first t steps.)

Thus, the algorithms described in Lemmas 8, 10, 11, and 12 can be (repeatedly) executed by an rCROW in the time bounds quoted above, after once paying the cost of precomputing the tables, and activating the appropriate number of processors. We sketch below how this can be done. Some of the techniques are borrowed from [19], and, incidentally, illustrate a few of the ideas used there to simulate rCROWs by Parallel Pointer Machines.

Lemma 13 *For fixed integers n, k, l, c, e, and h, where $k, e = O(\log n)$, $l = 2^e$, $c > 0$, and $h = c\lceil \log_2 n \rceil$, an rCROW can compute tables of the values of the unary functions in F (the set of function used to solve the pairing problem) for all arguments i, $0 \leq i \leq 2^h - 1$, in time $O(\log n)$ using $2^h - 1$ processors,.*

Proof: (Sketch.) It is convenient to assume that both processor indices and global memory addresses start at zero, rather than one as used everywhere else in this paper. For some fixed integer $b > 0$, processor P_i, $0 \leq i \leq 2^h - 1$ will own a block of 2^b words in global memory, beginning at address $i2^b$, and will store into the block the values of $f(i)$ for the various unary functions $f \in F$, plus a few others. It is convenient to view the blocks as forming a balanced binary tree, with $2i$ and $2i + 1$ being the children of i. (Note, however, that $i = 0$ has only a right child, or is its own left child, depending on one's viewpoint.) Initially, only processor P_0 is active. Each active processor $i < 2^{h-1}$ will fork 2 others, $2i$ and $2i + 1$, passing them its own index, and a flag indicating which child they are. (Again, $i = 0$ is an exception.) The newly forked processors j store their parent's index as $\text{DIV}_2(j)$, and the flag as $\text{MOD}_2(j)$.

The various unary functions are now easy to compute. For example, the depth of any node in the tree is easily computed as the successor of its parent's depth, where 0 has depth 0. Since $l = 2^e$, the function $\text{D}(i)$ is simply the depth of i's e^{th} ancestor in the tree. It is easily found by following the parent pointers up e levels, then copying the depth value stored there into i's D field. $\text{PREDECESSOR}(i)$ is i's left sibling if i is a right child; otherwise it is i's parent's predecessor's right child. That is, $\text{PREDECESSOR}(i)$ is $\text{DOUBLE}(\text{DIV}_2(i))$ if $\text{MOD}_2(i) = 1$; otherwise it is $\text{SUCCESSOR}(\text{DOUBLE}(\text{PREDECESSOR}(\text{DIV}_2(i))))$.

Next, observe that the MOD_2 bits along the upward path from i to the root comprise a list of i's bits, least significant first. Using this observation, a wide variety of functions can be efficiently precomputed. A useful example is the function $\text{REVERSE}(i)$. $\text{REVERSE}(i)$ is the reversal of the bit string ENCODE'd by i, or more precisely (to preserve low order zeros, and strip off the high order bit added by ENCODE) the floor of one half of the reversal of $2i + 1$. E.g., $\text{REVERSE}(111010_2) = 101011_2$. This can be constructed by using the procedure in the example immediately preceding Theorem 4, or in terms of the tree, by walking one pointer up the tree from i to node 1 while walking another down from 1 according to the MOD_2 bits seen along the upward path. $\text{REVERSE}(\text{MOD}_{2^k}(i))$ can be found by carrying out a similar process for k steps; $\text{MOD}_{2^k}(i)$ is found by reversing this. $\text{LEFT}(i)$ for a node i at depth $2d + 1$, which encodes a bit string of length $2d$, is i's ancestor at depth $d + 1$. This node can be found by walking two pointers up from i, with the first making two steps for each step made by the second; the second will reach $\text{LEFT}(i)$ when the first reaches 1. $\text{RIGHT}(i)$ is now easily found as $\text{RIGHT}(i) = \text{REVERSE}(\text{LEFT}(\text{REVERSE}(i)))$.

Another example is EQUAL_k. Recall that rCROWs can compare to zero, but lack a general compare instruction. Given k in a known location in global memory, each processor i walks two pointers in parallel towards the root, one from i and the other from k, comparing the bit sequences comprising the two integers. The unique processor finding them all equal will set its EQUAL_k field to one; all others store zero.

The remaining functions can be computed similarly. All of these operations can be completed in time proportional to the height h of the tree, which is $O(\log n)$ for our application to pairing. □

Thus, Lemmas 8, 10, 11, and 12 apply to rCROWs, provided $O(\log n)$ time for precomputation is allowed. In particular, we obtain the following result.

Theorem 14 *A CROW PRAM with n processors running in time $O(\log n)$ can be simulated by an rCROW with polynomially many processors in time $O(\log n \log\log n)$.*

Proof: (Sketch.) Precompute tables of the unary functions needed by the pairing algorithm as sketched above in Lemma 13. Also, precompute tables for addition (and/or other binary operations on $O(\log n)$ bit quantities used by the simulated CROW PRAM). This all takes $O(\log n)$ time. Finally, do a step-by-step simulation of the CROW PRAM, using Lemma 12 and the precomputed addition table to simulate addition steps. □

It follows from this that deterministic context-free language recognition and many other problems solvable in $O(\log n)$ time on CREW PRAMs are solvable in time $O(\log n \log\log n)$ by PPMs.

Two important features of the simulation presented in Theorem 14 are that it is uniform and that it uses only polynomially many more processors. It is possible to obtain faster rCROW algorithms, computing any function to within a constant factor as fast as on a nonuniform CREW PRAM, by exploiting both nonuniformity and substantially more processors. This relies on the following characterization. Let f be any n-ary function $f : D_1 \times \cdots \times D_n \to \mathbb{N}$, where $D_1, \ldots, D_n \subseteq \mathbb{N}$ are finite sets. Then the logarithm of f's decision tree complexity characterizes to within a constant factor the time for a (nonuniform) CREW PRAM with an arbitrarily powerful instruction set to compute f [21, 10].

With normal arithmetic capabilities, a nonuniform CROW PRAM can evaluate any decision tree of height h and size s in $\lceil \log_2 h \rceil + O(1)$ steps using s processors, by pointer jumping (Ragde, personal communication; see also [21, 10]). Preinitialized memory is used to specify the decision tree, naming the input variable to be tested at each internal node, the out-edges from each, and the function value at each leaf. Addition is used to index into the list of out-edges at each internal node in constant time. As in the proof of Lemma 8, even an rCROW (with preinitialized memory) can evaluate a *Boolean* decision tree using the same resources: since the out-degree of each internal node is two, SUCCESSOR can replace general addition for indexing into the list of out-edges. More generally, if the domain D_i of every input variable x_i has cardinality at most 2^k, then $\lceil \log_2 h \rceil + O(\log k)$ steps suffice, even on an rCROW. The idea is to use table lookup to replace each input variable x_i by its rank in D_i in $O(1)$ steps, to use k-SPLIT to convert these values to sequences of Booleans in $O(\log k)$ steps, then to evaluate the associated Boolean decision tree of height at most hk in $\lceil \log_2 hk \rceil + O(1)$ steps.

Note that the additive $\log k$ term above is best possible, since the pairing problem with domain $\{1, \ldots, n\} \times \{1, \ldots, n\}$ can be solved by a decision tree of height two, but requires time $\Omega(\log\log n)$ on an Arithmetically Restricted CROW PRAM by Theorem 3. We also remark that applying this result to convert a CREW PRAM algorithm running in time T

into a CROW or rCROW algorithm, in addition to introducing nonuniformity, may require a number of processors that is double-exponential in T.

References

[1] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[2] A. M. Ben-Amram and Z. Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[3] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique*, XXVII(1–2):99–124, Jan.-June 1981. Also in [20, pages 75–100].

[4] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, Feb. 1986.

[5] S. A. Cook and P. W. Dymond. Parallel pointer machines, 1991. Submitted.

[6] P. W. Dymond. Indirect addressing and the time relationships of some models of sequential computation. *Int. J. of Computers and Math. with Applications*, 5:195–209, 1979.

[7] P. W. Dymond and S. A. Cook. Hardware complexity and parallel computation. In *21st Annual Symposium on Foundations of Computer Science*, pages 360–372, Syracuse, NY, Oct. 1980. IEEE.

[8] P. W. Dymond and W. L. Ruzzo. Parallel random access machines with owned global memory and deterministic context-free language recognition. In L. Kott, editor, *Automata, Languages, and Programming: 13th International Colloquium*, volume 226 of *Lecture Notes in Computer Science*, pages 95–104, Rennes, France, July 1986. Springer-Verlag.

[9] D. Eppstein and Z. Galil. *Parallel Algorithmic Techniques for Combinatorial Computation*, pages 233–283. Annual Reviews in Computer Science. Annual Reviews, Inc., 1988.

[10] F. E. Fich. The complexity of computation on the parallel random access machine. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufman, San Mateo, CA, 1993. To appear.

[11] F. E. Fich, R. Impagliazzo, B. Kapron, V. King, and M. Kutyłowski. Limits on the power of parallel random access machines with weak forms of write conflict resolution. In *10th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[12] F. E. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM Journal on Computing*, 17:606–627, 1988.

[13] F. E. Fich and A. Wigderson. Towards understanding exclusive read. *SIAM Journal on Computing*, 19(4):717–727, 1990.

[14] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, May 1978.

[15] M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *30th Annual Symposium on Foundations of Computer Science*, pages 190–195, Research Triangle Park, NC, Oct. 1989. IEEE. Preliminary version.

[16] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 117–124, Crete, Greece, July 1990.

[17] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[18] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwan, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 869–941. M.I.T. Press/Elsevier, 1990.

[19] T. W. Lam and W. L. Ruzzo. The power of parallel pointer manipulation. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 92–102, Santa Fe, NM, June 1989.

[20] *Logic and Algorithmic,* An International Symposium Held in Honor of Ernst Specker, Zürich, Feb. 5–11, 1980. Monographie No. 30 de L'Enseignement Mathématique, Université de Genève, 1982.

[21] N. Nisan. CREW PRAMs and decision trees In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 327–335, Seattle, WA, May 1989.

[22] A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, Aug. 1980.

[23] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18:110–127, 1979.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

(RCS Revision: 1.56 Date: 1993/05/12 06:04:07)