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 \(O(\log \log n)\) 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.

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 most useful parallel model for bridging the gap between algorithms and architectures is still not settled (see, e.g., Valiant [28]). 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 been widely studied already. 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 $O(\log n)$ per step. Second (to our surprise), this lower bound is tight. We give such a step-by-step simulation using $O(\log n)$ 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. Previously this problem was not known to be solvable by a PPM 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 the 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 $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ 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 of $n$ arguments can be computed by an abstract PRAM in $\log n$ steps (see, e.g., Fich [11]). 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 the more usual arithmetic operations.

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

**CROW PRAMs.** Dymond and Ruzzo [9] 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. Even algorithms that make more subtle use of exclusive write can often be converted to ones obeying the ownership restriction. For example, Lin et al. have recently shown that CROW PRAMs are sufficiently powerful to execute a variant of Cole’s parallel merge sort algorithm in time $O(\log n)$ [22].

Dymond and Ruzzo [9] characterize the power of such 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 $\text{DSPACE}(\log n)$ and (uniform) $\text{AC}^1$.) 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 [15].) For definiteness, the term “CROW PRAM” below will refer to this model unless otherwise qualified.

Similar but not identical notions of “ownership” have proven useful in practice for certain cache coherence protocols (see, e.g., Archibald and Baer [1]) and have appeared in the earlier lower bound work of Cook et al. [4] and of Fich and Wigderson [14].

**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 Ben-Amram and Galil [2], Harel and Tarjan [18], and Tarjan [26]. 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—ordinary unit cost RAMs stripped of all arithmetic capabilities except for the successor, or $+1$ operation [25].

The notion of parallel computation by pointer manipulation is formally captured by the PPM, studied by Dymond and Cook [3, 5, 7]. 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 [5 or 7] for a more complete definition. In the earlier papers, the PPM is called an HMM, or hardware modification machine, by analogy to Schönhage’s SMM. Goodrich and Kosaraju subsequently considered a more complex model, which they also called a PPM, having both pointers and integer arithmetic [16]. Tromp and van Emde Boas have proposed a quite different form of parallel pointer machine [27].)

Lam and Ruzzo [20] 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 PRAM. This characterization is central to our results, since it allows us to cast PPMs and CROW PRAMs in a common framework. Adding 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, Cook and Dymond used the “pointer doubling” technique of Fortune and Wyllie [15] to show that DSPACE(log n) can be simulated by a PPM in time O(log n) [5, 7]. 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 and write constraint and to use only a PPM by a PRAM, the PRAM can be made to obey the double, rather than general addition. Furthermore, for PRAMs satisfying these two restrictions (i.e., rCROW PRAMs, a converse simulation by PPMs is possible. The known relationships among the various complexity classes described above are summarized in Fig. 1. In this figure, the notations CROW(log n), rCROW(log n), and CRCW(log n) denote the classes of languages accepted by CROW, rCROW, and CRCW PRAMs, respectively, in O(log n) time, and similarly for PPM(log n). In this paper, all models and complexity classes are uniform ones, unless otherwise stated. Note that since the uniform versions of these parallel models activate new processors only by explicit “fork” instructions, only nO(1) processors can be activated in O(log n) steps.

Shortly after the appearance of a preliminary version of the present paper [8], Lange and Niedermeier provided additional insight into the relationships among these and other parallel models by casting many of them into a common framework [21]. In particular, they obtained PRAM characterizations of all of the complexity classes listed in Fig. 1 by variously restricting the PRAMs’ instruction sets, data flow, and/or control flow. Some of their restrictions, e.g., restricting the machines to have only unary operations, are similar to those we consider below.

The open problem that motivated the present paper was the question of whether the simulation of rCROW PRAMs by parallel pointer machines could be extended to the more general CROW PRAM model considered by Dymond and Ruzzo [9]. Specifically, could a PPM simulate addition? On the one hand, “adding” two unrelated pointers seems difficult. On the other hand, by [9] it would suffice if one could do DCFL recognition on a PPM, and the DCFL recognition algorithm given in [9] is basically a generalization of the pointer doubling algorithm. Thus, it does not 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 PRAM.

Arithmetically Restricted PRAMs. The essential weakness of the rCROW PRAM does not 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 PRAM 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 unary 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 PRAM (and, hence, the PPM) is a very simple special case of an arithmetically restricted CROW PRAM with a conditional branch.

The following problem is central to our results.

The Pairing Problem. The pairing problem of size n is, given two inputs x and y in the range [1, ..., n], to compute any injective function π(x, y), where

\[ \pi: \{1, ..., n\} \times \{1, ..., n\} \rightarrow \mathbb{N}. \]

One example of a pairing function is the function π(x, y) = (x − 1)n + y with codomain \{1, ..., n²\}, commonly used by compilers to linearize two-dimensional arrays. Another example is the function that concatenates the \[n\] and \[n+1\]-bit binary representation of x and y to form a 2\[n+1\]-bit number. (The latter is the function used by our upper bound algorithm.) Addition is another example of a binary operation that is not a pairing function, since it is not injective.

The pairing problem was motivated by the DCFL recognition algorithm of Dymond and Ruzzo [9], a key component of which was accessing a two-dimensional array via the pairing function in the first example above. It is not

\[ AC^0 \subseteq NC^1 \subseteq \text{DSPACE}(\log n) \subseteq \left\{ \begin{array}{ll} \text{PPM}(\log n) & \subseteq \text{LOGDCFL} \subseteq \text{CROW}(\log n) \subseteq \text{AC}^1 \end{array} \right\} \]

FIG. 1. Relationships among some parallel complexity classes.
difficult to see that bounds on this pairing function apply to all of them, as well as to addition and various other nonpairing functions. Specifically, if an arithmetically restricted CROW PRAM could solve the pairing problem, then it could simulate addition by using the pairing function \( \pi \) to access a precomputed table \( A \) such that \( A[\pi(x, y)] = x + y \). More generally, it could compute any binary operation in this way, including any other pairing function. Thus any pairing problem is “universal” for simulating binary operations by unary ones. Conversely, many binary operation that are not themselves pairing functions can be used to compute pairing functions. For example, it is easy to see that a pairing function can be computed by one processor in constant time, given a simple precomputed table \( A \) such that \( A[\pi(x, y)] = x + y \).

Outline. The rest of this paper is organized as follows. Section 2 defines the arithmetically restricted PRAM more fully. Section 3 gives our lower bounds. Section 4 sketches some basic upper bounds for arithmetically restricted PRAMs, and Section 5 gives our upper bound on the pairing problem, and the consequent simulation of CROW PRAMs by PPMs.

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 \((L[1], L[2], \ldots)\). The number of processors, \( p \), may be a function of the problem size, \( n \). 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. Let \( C \subseteq \mathbb{N} \) be an arbitrary set of constants. Let \( n \in \mathbb{N} \) be a fixed natural number (the problem size). For each \( k \geq 2 \), let \( Q_k \) be an arbitrary set of \( k \)-ary functions \( g: \mathbb{N}^k \rightarrow \{1, \ldots, n\} \), and let \( Q = \bigcup_k Q_k \). 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 I. An arithmetically restricted PRAM program is a finite sequence of such instructions. In a uniform version of the model (as used in our upper bounds), all processors execute the same program. In a nonuniform version (as in our lower bounds), each processor can have a different program.

Throughout this paper, we assume that processors are allowed to simultaneously read from the same global memory cell. Reads happen before writes—if processor \( A \) reads a global memory cell during the same step that processor \( B \) writes it, the value read by \( A \) is the value present at the end of the previous step, not the value written by \( B \). 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—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 Epstein and Galil [10], Fich et al. [12, 13], and Hagerup and Radzik [17].

A concurrent-read, owner-write (CROW) PRAM 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. See Dymond and Ruzzo [9].

Note that several features (besides arithmetic) commonly included in PRAM models, such as conditional branching and indirect addressing into local memory, are excluded from the arithmetically restricted PRAM. Extensions to the basic model arise by allowing one or more of these operations, as listed in Table II and described more fully below.

The branch instruction conditionally 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 more restricted form of branching. Here L[1] must contain either 0 or 1 and, in the latter case, the unary function f ∈ F is applied to the value in L[0].

The k-concatenate instruction concatenates the bit strings held in two cells. It requires that the second argument L[1] contain a number that is at most k bits in length, adding leading zeros if it is shorter. Note that k-concatenate for k = \lceil \log_2(n + 1) \rceil directly solves the pairing problem. We are interested in the case when k is much smaller.

Another extension considered is indirect addressing of local memory. Additionally, we consider k-limited indirect addressing, defined to be the special case where the address argument L[0] of lread or L[1] of lwrite is restricted to be a positive integer no larger than k. In this case, one of the local memory cells L[1], ..., L[k] of the processor will be accessed.

The last extension we consider 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 subroutine in a larger computation whose total cost dominates the cost of precomputing the tables used by the pairing subroutine.

Formally an arithmetically restricted PRAM is defined by specifying n, p, F, C, Q, its program, and the contents of its preinitialized memory cells (if any). We will generally not be this formal, as the specifications of all these parameters will be clear from context.

### 3. LOWER BOUNDS

In this section we prove lower bounds on the pairing problem, defined in Section I. Initially, global memory cells M[1] and M[2] each contain a value in the range \{1, ..., 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.

### TABLE I

**Arithmetically Restricted PRAM: Basic Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>L[0] ↦ M[L[0]]</td>
<td>Indirect read from global memory</td>
</tr>
<tr>
<td>LREAD</td>
<td>L[0] ↦ M[0]</td>
<td>Direct read from local memory</td>
</tr>
<tr>
<td>WRITE</td>
<td>M[L[1]] ↦ L[0]</td>
<td>Indirect write into global memory</td>
</tr>
<tr>
<td>LWRITE</td>
<td>L[i] ↦ L[0]</td>
<td>Direct write into local memory</td>
</tr>
<tr>
<td>LOADC</td>
<td>L[0] ↦ c</td>
<td>Assign a predefined constant c ∈ C</td>
</tr>
<tr>
<td>f</td>
<td>L[0] ↦ f(L[0])</td>
<td>Apply a unary function f ∈ F</td>
</tr>
<tr>
<td>q</td>
<td>L[0] ↦ q(L[0]...M[k-1])</td>
<td>Evaluate a k-ary function q ∈ Q_k</td>
</tr>
</tbody>
</table>

### TABLE II

**Arithmetically Restricted PRAM: Extensions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH</td>
<td>if L[0] &gt; 0 then goto ...</td>
<td>Conditional branch</td>
</tr>
<tr>
<td>CONDITIONAL-f</td>
<td>if L[1] &gt; 0 then L[0] ↦ f(L[0])</td>
<td>Conditional function application (L[1] ∈ {0, 1})</td>
</tr>
<tr>
<td>k-concatenate</td>
<td>L[0] ↦ L[0] · 2^k + L[1]</td>
<td>k-bit concatenation (L[1] &lt; 2^k)</td>
</tr>
<tr>
<td>lread</td>
<td>L[0] ↦ L[L[0]]</td>
<td>Indirect read from local memory</td>
</tr>
<tr>
<td>lwrite</td>
<td>L[L[1]] ↦ L[0]</td>
<td>Indirect write to local memory</td>
</tr>
</tbody>
</table>
The lower bound proof technique was partly inspired by results of Dymond concerning sequential RAMs [6]. Throughout this section, let $V(0, j, x, y, t)$ denote the value in global cell $M[j]$ and, for $t > 0$, let $V(i, j, x, y, t)$ denote the value in cell $L[i][j]$ of processor $P_i$ at the end of step $t$ when $x$ and $y$ are the inputs to the pairing problem. Here $x, y \in \{1, \ldots, n\}$, $t \in \mathbb{N}$, $i \in \{0, \ldots, p\}$, $j \in \mathbb{N}$ and either $i \neq 0$ or $j \neq 0$.

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 and arbitrarily many processors.

**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

$$A_t = \{ V(i, j, x, y, t') \mid i \in \{0, 1, \ldots, p\}, j \in \mathbb{N},$$

$$i \neq 0 \text{ or } j \neq 0, x, y \in \{1, \ldots, n\},$$

$$t' \in \{0, \ldots, t\}, \text{ and } V(i, j, x, y, t') \neq V(i, j, x', y', t')$$

for some $x', y' \in \{1, \ldots, n\}$.

Clearly $A_t \subseteq A_{t+1}$.

Let $a_t$ denote the cardinality of the set $A_t$.

Initially, $A_0 = \{1, \ldots, n\}$; hence $a_0 = n$. To get an upper bound on $a_{t+1}$, we consider the instructions executed at step $t+1$. Note that any given processor at any given step executes the same instruction for all $x, y \in \{1, \ldots, n\}$, since only the basic instructions of an arithmetically restricted PRAM are available, and they exclude conditional execution. For each type of instruction, we determine which values might be in affected cells for the first time at the end of step $t+1$ as a result of instructions of that type.

Direct reads from local memory and direct writes to local memory do not add any new values to $A_{t+1}$, although they may increase the number of affected memory cells.

Suppose $P_i$ performs an indirect read from global memory. At the end of step $t$, its accumulator, $L[0]$, contains the address from which to read. At the end of step $t+1$, $L[0]$ contains the value read. If $L[0]$ is oblivious at time $t$, then its value is affected at time $t+1$ only if the cell read, $M[L[0]]$, is affected at time $t$. In this case, the value $P_i$ reads is in $A_t$. Therefore, assume $L[0]$ is affected at time $t$. Then, for any input, the address of the cell from which $P_i$ is to read is in $A_t$, and, hence, the value it reads is in $A_t = \{ V(0, a, x, y, t) \mid a \in A_t, x, y \in \{1, \ldots, n\} \}$. In other words, all new values added to $A_{t+1}$ as a result of indirect reads from global memory are in $A_t - A_{t-1}$. 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 the cardinality of $A_t - A_{t-1}$ is at most $a_t$.

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 it writes. If $L[1]$ and $L[0]$ are oblivious at time $t$, then the memory cell $M[L[1]]$ to which $P_i$ writes is oblivious at time $t+1$. If $L[1]$ is oblivious at time $t$, but $L[0]$ is affected at time $t$, then $M[L[1]]$ is affected at time $t+1$ and will contain a value in $A_t$, for all $x, y \in \{1, \ldots, n\}$. Now consider the set $B_i$ of processor indices $i \in \{1, \ldots, p\}$ such that $P_i$ performs an indirect write to global memory during step $t+1$ and $P_i$’s local memory cell $L[1]$ is affected at time $t$. The set of possible locations to which $P_i$ writes during step $t+1$ is a subset of $A_t$, for every $i \in B_i$. Because write conflicts are not allowed in a CREW PRAM, it follows from the pigeonhole principle that $B_i$ has cardinality at most $a_t$. The values added to $A_{t+1}$ as a result of indirect writes to global memory are either values $V(0, x, y, t)$ written by processors $P_i$ for $i \in B_i$, or values $V(0, a, x, y, t) \neq a$ already in the cells that processors $P_i$ for $i \in B_i$ might write to. Since $B_i$ has cardinality at most $a_t$, it follows that $\{ V(0, x, y, t) \mid i \in B_i, x, y \in \{1, \ldots, n\} \}$ has cardinality at most $a_t$. Hence, at most $2a_t$ new values are added to $A_{t+1}$ as a result of indirect reads from global memory and indirect writes to global memory.

When a predefined constant is loaded into a processor’s accumulator, the accumulator is oblivious at time $t+1$.

Now, suppose $P_i$ applies a unary function $f \in F$. If $P_i$’s accumulator is oblivious at time $t$, then it is also oblivious at time $t+1$, so assume that $P_i$’s accumulator $L[0]$ is affected at time $t$. For any input, the value it contains at the end of step $t$ is in $A_t$. Thus, at the end of step $t+1$, $L[0]$ contains a value in $\{ f(a) \mid a \in A_t \}$. Hence, at most $a_t |F|$ new values are added to $A_{t+1}$ as a result of applications of unary functions.

Finally, when a processor evaluates a $k$-ary function $g \in Q_k$, the value in the accumulator at the end of step $t+1$ is in $\{1, \ldots, n\} = A_0 \subseteq A_t$.

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

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 $\Omega(\log(p^2/n))$ 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, that is,

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

Clearly \(V_0 \subseteq V_{t+1}\). Let \(v_t\) denote the cardinality of the set \(V_t\). Recall that \(V(i, 0, x, y, 0) = i\), so \(v_0 = p\) and \(\{1, \ldots, n\} \subseteq V_0\). To get an upper bound on \(v_{t+1}\), we consider the instructions executed at step \(t + 1\).

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 \(v_t\) different values appearing in the accumulators during the first \(t\) steps and at most \(|F|\) different values can result from each by applying the functions in \(F\). The functions in \(Q\) have codomain \(\{1, \ldots, n\}\). Thus, after a processor evaluates such a function, 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 the global memory cells \(M[1]\) 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 added to \(V_{t+1}\) for each local or global memory cell that is 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 \(2^p\) states at the end of step \(t\). In each such state, it might read the initial value of a (directly addressed) local memory cell or use a new predefined constant \(c \in \mathbb{C}\) (but not both).

Thus \(v_{t+1} \leq v_t(2^p(F[2] + 2^p)\). It is easy to verify by induction that \(v_t \leq 2^p(F[2] + 2^p)\) 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 of Eppstein and Galil [10] 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, the ability to branch, and arbitrarily many processors.

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

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

Clearly \(K(S, t) \subseteq K(S, t+1)\). Let \(k(s, t)\) denote the maximum cardinality of the set \(K(S, t)\), taken over all \(s\)-processor subsets \(S\).

Initially, each processor’s accumulator contains its number. Thus

\[ K(S, 0) = \{1, \ldots, n\} \cup S \]

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

To get an upper bound on \(k(s, t+1)\), consider the instructions executed at step \(t + 1\) by the processors in some set \(S\) of size \(s\). Writes to local or global memory do not change the values a processor could know. Evaluating functions with codomain \(\{1, \ldots, n\}\) produces values in \(K(S, 0) \subseteq K(S, t)\). As a result of branches, each of the processors in \(S\) can be in one of at most \(2^t\) states at the end of step \(t\). In each such state, it might read the initial value of a local memory cell or use a new predefined constant \(c \in \mathbb{C}\) (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 produce at most \(2^t\) new values (i.e., values not in \(K(S, t)\)) 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. (Recall, this is a CROW
model.) Except for $M[1]$ and $M[2]$, whose initial values are contained in $\{1, ..., n\} \subseteq K(S, t)$, each memory cell has a single initial value. The only values that 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 on $t$ that $k(s, t) \leq (n+s)(3+|F|)^t$. In particular, $k(1, t) = n2^{20t}$. At the end of the computation, the value in the output cell is either the initial value of the cell or a value written by the processor $P$ that owns this cell. Since every value written by $P$ is a value that $P$ could know, there must be at least $n^2-1$ different values that $P$ could know. Thus, the number of steps in the computation must be in $\Omega(\log \log n)$.

We remark that allowing a binary function with a quadratic (or even superlinear) codomain would invalidate the lower bound proofs presented above. This motivates the restriction that the functions in $Q$ all have codomains of linear size.

4. BASIC UPPER BOUNDS

In this section we present upper bounds for the pairing problem using arithmetically restricted PRAMs with different instruction sets. They are mainly important in showing that the lower bounds proved in the previous section are tight.

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. For example,

\[
\text{for } 5 \leq i \leq n \text{ pardo}
\]

\[
P_1 \colon \{M[i] \leftarrow 1\text{-CONCATENATE}(L[1], M[i])\}
\]

is really a shorthand notation hiding a handful of direct and indirect read/write operations required to collect the arguments to, and store the results from, the 1-CONCATENATE operations being performed in parallel by processors $P_5, P_6, ..., P_n$. Additionally, we assume a few local memory cells are available as scratch space for such purposes. As above, we generally indicate parallel execution with an explicit pardo ("parallel do") construct and explicitly indicate which processor(s) execute each statement or block of statements.

Obviously, using $[\log_2(n+1)]\text{-CONCATENATE}$, a single processor can solve the pairing problem in constant time by concatenating $x$ and $y$:

\[
P_1 \colon \{M[1] \leftarrow [\log_2(n+1)]\text{-CONCATENATE}(M[1], M[2])\}
\]

If $[\log_2(n+1)]\text{-CONCATENATE}$ 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:

\[
P_1 \colon \{\text{do } [\log_2(n+1)] \text{ times}
M[3] \leftarrow \text{MOD}_2(M[2])
M[2] \leftarrow \text{DIV}_2(M[2])
M[1] \leftarrow 1\text{-CONCATENATE}(M[1], M[3])
\}
\]

(Here $\text{MOD}_k$ and $\text{DIV}_k$ 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 four unary functions and without preinitialized memory, a single processor arithmetically restricted PRAM can solve the pairing problem in $O(\log n)$ time.

**Proof.** Use the program given in Fig. 2.
There is a unique processor
number into the last step. As before, the desired processor writes its number, as the answer, to $M$
other processors write their numbers to $M$.

Theorem 1. 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 $P_i$ 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 are no longer needed. In short, we execute the following:

\[
\text{for } 1 \leq r \leq n^2 \text{ pardo}
\]
\[
P_i ; \{ M[2] - q(M[1], M[2], r) \} \leftarrow r
\]

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 2. 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 $P_i$ 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 $M[1]$:

\[
L[0] \leftarrow \text{double}(L[0])
\]
\[
\text{if } L[1] = 1 \text{ then } L[0] \leftarrow \text{successor}(L[0])
\]

Shift $L[0]$ left one bit.

Conditionally change low order bit.

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 1-CONCATENATE or 2-limited indirect addressing of local memory (defined in Section 2) is available instead of conditional function application.

Theorem 3. 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 Fig. 3.

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

Finally, 1-COMPACTION can simulate 2-limited indirect addressing of local memory. The idea is for processor $P_i$ to use indirect addressing into global memory to replace indirect addressing into local memory. It temporarily uses the global memory cells $M[2]$ and $M[2i + 1]$ in place of its local memory cells $L[1]$ and $L[2]$. See Fig. 5.

Finally, we note that constant time algorithms for the pairing problem can be obtained easily if the model is modified in various other ways. 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:

\[
L[0] \leftarrow \text{shift}(L[0])
\]
\[
L[1] \leftarrow L[0]
\]
\[
L[2] \leftarrow \text{add}(L[0])
\]


FIG. 3. Conditional function application simulates 1-CONCATENATE.

FIG. 4. 2-limited indirect addressing simulates conditional function application.
In constant time, each processor nodes contain the function values for this related problem.

\[ \text{single integer in the range } F \]

\[ P_i : \{ L[1] \leftarrow M[1] \} \]

\[ \text{for } 1 \leq i \leq n \text{ pardo} \]

\[ P_i : \{ M[i] \leftarrow f_i(M[2]) \} \]

\[ P_i : \{ M[1] \leftarrow M[L[0]] \} \]

After the second step, \( M[i] = (i - 1) n + y \) for all \( i \in \{ 1, \ldots, n \} \), and so \( M[1] = (x - 1) n + y \) at the end. Thus, the restriction that the set of unary operations \( F \) has constant size is necessary for the lower bound in Section 3. It is not even sufficient that each processor only use one different unary operation. Similarly, if unlimited indirect addressing of preinitialized local memory is allowed, 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 \( f_i \). Thus, the restrictions on the model as defined in Section 2 are necessary for the lower bounds given in Section 3.

5. PAIRING UPPER BOUND ON ARITHMETICALLY RESTRICTED CROW PRAMs

We now turn to our upper bound for the pairing problem on arithmetically restricted 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 constituent bits. Specifically, the \( k \)-join problem is to concatenate \( k \) input bits, producing a single 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 the \( k \) input bits together with a high-order bit with value 1, 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:

\[ \text{for } 1 \leq i \leq 2^k - 1 \text{ pardo} \]

\[ P_i : \{ M[i] \leftarrow 1\text{-concatenate}(i, M[d + 1]) \} \]

Recall that we are using a nonuniform model, so \( d \), which depends only on the processor index \( i \), is assumed to be “built in.” We will consider a uniform model below.

It turns out that the function value to be stored in the leaf at address \( i \) is \( i \) itself. Thus, the leaves can be initialized as follows:

\[ \text{for } 2^k \leq i \leq 2^k + 1 - 1 \text{ pardo} \]

\[ P_i : \{ M[i] \leftarrow i \} \]

Pointer jumping can then be used to determine the answer in \( 1 + \lceil \log_2 k \rceil \) steps. Specifically, processors corresponding to internal nodes perform the following operations:

\[ \text{for } 1 \leq i \leq 2^k - 1 \text{ pardo} \]

\[ \text{do } 1 + \lceil \log_2 k \rceil \text{ times} \]

\[ P_i : \{ M[i] \leftarrow M[M[i]] \} \]

Finally, to solve the \( k \)-join problem, processor 1 applies the \text{MOD}_{2^k} \text{ function to remove the unwanted high-order bit from the answer constructed above:}

\[ P_1 : \{ M[1] \leftarrow \text{MOD}_{2^k}(M[1]) \} \].

(Again, we are assuming a nonuniform model, so the available unary functions like \text{MOD}_{2^k}, are allowed to depend on \( k \).)

Note that the only processor to write into \( M[i] \) is \( P_i \), so the algorithm obeys the owner write restriction with \( P_i \) owning \( M[i] \). \( \square \)

Using standard techniques (e.g., see Karp and Ramachandran [19]), 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
for \( l \leq i \leq l2^{k+1} - 1 \) pardo
\[
P_l: \begin{align*}
\text{If EQUAL}(d(i)) = 0 & \text{ then } M[i] \leftarrow 1-CONCATENATE(i, M[b(i)]) \\
\text{else } M[i] & \leftarrow 1 \\
\text{do } 1 + \lceil \log_2 k \rceil \text{ times} & \\
M[i] & \leftarrow M[M[i]] \\
\text{if } d(i) = 0 & \text{ then } M[t(i)] \leftarrow \text{MOD}_2(M[i]) \\
\text{Initialize free: } & \\
\text{Internal node: } & \\
\text{Leaf: } & \\
\text{Pointier jumping: } & \\
\text{Extract and move answer: } & \\
\end{align*}
\]

FIG. 6. The algorithm for \( l \) instances of \( k \)-JOIN.

**Lemma 10.** An \( l2^{k+1} - 1 \) processor arithmetically restricted CROW PRAM can solve \( l \) independent instances of the \( k \)-JOIN problem in time \( O(\log k) \) time.

**Proof.** Root \( l \) binary trees at locations \( M[1], ..., M[2^l - 1] \), again viewing \( M[2^l] \) and \( M[2^l + 1] \) as the children of \( M[i] \). Thus, for \( d = 0, ..., k \), the \( l2^d \) nodes of depth \( d \) are located at cells \( M[2^d], ..., M[2^{d+1} - 1] \). In other words, location \( M[i] \), for \( l \leq i \leq l2^{k+1} - 1 \), contains a node at depth \( d(i) = \lceil \log_2 (i/l) \rceil \) in the tree numbered \( t(i) = \lfloor i/2^{d(i)} \rfloor + 1 - l \). To determine where it should point, processor \( P_l \), \( l \leq i \leq l2^k - 1 \), has to read the input bit numbered \( b(i) = k(t(i) - 1) + 1 + d(i) \). Note that for \( l \leq i \leq l2^{k+1} - 1 \), we have \( 0 \leq d(i) \leq k \), \( 1 \leq t(i) \leq k \), and for \( l \leq i \leq l2^k - 1 \), we have \( 1 \leq b(i) \leq kl \). The unary functions \( \text{EQUAL}_x \), which has value 1 when its argument is \( k \) and 0 otherwise, and \( \text{MOD}_2 \), which computes the remainder when its argument is divided by \( 2^k \), are also used in the program, which is executed by all processors \( P_l \), \( l \leq i \leq l2^{k+1} - 1 \). See Fig. 6. Again, note that the algorithm obeys the order write restriction since the only processor to write into \( M[i] \) is \( P_l \).

The \( k \)-SPLIT problem is the inverse of \( k \)-JOIN, namely, to break an integer in the range \( 0 \) to \( 2^k - 1 \) into a sequence of \( k \) bits.
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 $n^2$ 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 $0, ..., 2^k - 1$. Then perform $2n$ simultaneous instances of $k$-JOIN. 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 done. Some of the techniques are borrowed from the ideas used there to simulate rCROWs by parallel models. That is, there is only one initially active processor. (The PPM, as defined by Cook and Dymond [3, 5, 7], 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 PRAM 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 precomputation can be done. Some of the techniques are borrowed from Lam and Ruzzo [20] and, incidentally, illustrate a few of the ideas used there to simulate rCROWs by parallel pointer machines.

**Lemma 13.** Let $k$ be a fixed positive integer, and let $l = 2^l$ be a fixed integer power of two with $c \geq 0$. Let $K = 2^{c \log k}$, and let $h = K + c + 1$. Given $k$ and $l$, an rCROW with $2^k = O(2^h)$ processors can compute in time $O(h) = O(k + \log l)$ the information needed for the foregoing algorithms to solve $l$ simultaneous instances of the pairing problem on $k$-bit values (i.e., pairs $(x_i, y_i)$, $1 \leq i \leq l$, with $x_i, y_i < 2^k$ for all $i$). This includes calculating the values of parameters such as $e, h, K, lK, \{ \log_i k \}$, as well as tables of the values $f(i)$ of the unary functions $f \in F_{k, l}$ for all arguments $i$, $0 \leq i \leq 2^k - 1$.

**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 $P_i$, $0 \leq i \leq 2^b - 1$, will own a block of $2^b$ words in global memory, beginning at address $2^b i$. It will store into its block the values of $f(i)$ for the various unary functions $f \in F_{k, l}$, 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 process-\( sor \( P_0 \) is active. Each active processor \( P_i, i < 2^h - 1 \), will fork two others, \( P_{2i} \) and \( P_{2i+1} \), passing them its own index \( i \), and a flag indicating which child they are. (Again, \( i = 0 \) is an exception.) A newly forked processor \( P_j \) stores its parent’s index as \( \text{DIV}_j(f) \), and the flag as \( \text{MOD}_j(f) \).

Various unary functions are now easy to compute. For example, the depth of any node in the tree is easily com-\( puited as the successor of its parent’s depth, where node 0 has depth 0. \text{PREDECESSOR}(i) \) 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} \((\text{DIV}_i(f)) \) if \( \text{MOD}_i(f) = 1 \); otherwise, it is \text{SUCCESSOR} \((\text{DOUBLE} \((\text{PREDECESSOR}(\text{DIV}_i(f))) \))\). For any \( i > 0 \), \( \lceil \log_2 i \rceil \) is easily calculated as the depth of node \( \text{PREDECESSOR}(i) \). The parameter \( e = \log_2 i \) can be calculated in this way. Furthermore, the algorithm can tell when the tree has been grown to this depth by watching for global memory block \( l \) to be initialized.

\[
K = 2^{\lceil \log_2 k \rceil}
\]

is the left child of the leftmost node on the same level as \( \text{PREDECESSOR}(k) \). It can be found by walking a pointer from node \( \text{PREDECESSOR}(k) \) towards the root, while simultaneously walking another pointer from the root down the left side of the tree. Once both \( K \) and \( e \) are known, \( h = K + e + 1 \) can be determined by walking a pointer downwards from \( K \) while walking another upwards from \( e \). This process can be synchronized with growth of the tree so that construction of further levels of the tree is terminated when level \( h \) is built. Similar techniques can be used to con-\( trol which processors are active in the various \text{parado} loops.

Since \( l = 2^k \), the function \( \text{DIV}(i) \) is simply the depth of \( i \)'s left ancestor in the tree. It is easily found by following the parent pointers up \( e \) levels and then copying the depth value stored there into \( i \)'s \text{DIV} field.

Next, observe that the \( \text{MOD}_2 \) bits along the path from \( i \) up to, but not including 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 \( \text{REVERSE}(i) \). We say the integer \( i \) encodes a string \( x \in \{0, 1\}^* \) if the binary numeral for \( i \) is \( 1x \). \text{REVERSE}(i) \) is the encoded value of the reversal of the bit string encoded by \( i \). For example, \( \text{REVERSE}(111010_2) = 101011_2 \). 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 \( \text{MOD}_2 \) bits seen along the upward path. \( \text{REVERSE}(\text{ENCODE}_2(\text{MOD}_2(f))) \) can be found by carrying out a similar process for \( k \) steps. \( \text{MOD}_2(f) \) is found by reversing this, except that the down-\( going \) pointer starts at node 0 instead of node 1 (which effectively deletes the high-order 1 bit inserted by \( \text{ENCODE}_2 \)) \( \text{LEFT}(i) \) for a node \( i \) at depth \( 2^d + 1 \), 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 \( \text{EQUAL}_k \). Recall that \( rCROW \) PRAMs can compare to zero, but lack a general compare instruc-\( tion. Given a \text{K} \) in a known location in global memory, each process \( i \) walks two pointers in parallel towards the root, one from \( i \) and the other from \( k \), comparing the bit sequen-\( ces \) comprising the two integers. The unique processor finding them all equal will set its \( \text{EQUAL}_k \) 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.

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

**Theorem 14.** A \((\text{uniform}) \) CROW PRAM with \( n \) Boolean inputs, \( n^{O(1)} \) processors, and running in time \( O(\log n) \) can be simulated by an \( rCROW \) PRAM with \( n^{O(1)} \) processors in time \( O(\log n \log \log n) \).

**Proof (Sketch).** By standard arguments, there is a \( k = O(\log n) \) such that the all integers constructed by the CROW PRAM have at most \( k \) bits. Similarly, there is an \( e = O(\log n) \) such that it activates at most \( l = 2^e \) processors. For these parameters, precompute tables of the unary functions needed by the pairing algorithm as sketched above in Lemma 13. Similarly, precompute tables for addition of \( k \)-bit values. This all takes \( O(\log n) \) time with \( n^{O(1)} \) pro-\( cessors. 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 with polynomially many pro-\( cessors.

Two important features of the simulation presented in Theorem 14 are that it is uniform and that it uses only poly-\( nomially many more processors. It is possible to obtain faster \( rCROW \) PRAM algorithms, computing any function to within a constant factor as fast as on a nonuniform CREW PRAM, by exploiting both nonuniformity and sub-\( stantially more processors. This relies on the following characteriza-\( tion. 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 \( f \) (Fich [11], Nisan [24]).

With normal arithmetic capabilities, a nonuniform CROW PRAM can evaluate any decision tree of height \( h \)
and size \(s\) in \([\log_2 h]+O(1)\) steps using \(s\) processors, by pointer jumping [Prabhakar Ragde, personal communication; see also Fich [11], Nisan [24]). 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 \(\text{cROW}\) (with preinitialized memory) can evaluate a Boolean decision tree using the same resources: since the out-degree of each internal node is two, \(D_i\) edges. More generally, if the domain \(D_i\) of every input variable \(x_j\) has cardinality at most \(2^k\), then \([\log_2 h]+O(\log k)\) steps suffice, even on an \(\text{cROW}\) PRAM. The idea is to use table lookup to replace each input variable \(x_j\) 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 \(h \log k \in [\log_2 h k]+O(1)\) 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 \(O(\log \log n)\) 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 \(T\).

ACKNOWLEDGMENTS

We thank the anonymous referees for careful reading and helpful suggestions.

REFERENCES


