JOURNAL OF COMPUTER AND SYSTEM SCIENCES 53, 104–111 (1996)
ARTICLE NO. 0052

Weak Forms of Write Conflict Resolution

Faith E. Fich*

Department of Computer Science, University of Toronto, Toronto, Canada M5S 3G4

Russell Impagliazzo†

Department of Computer Science, University of California, San Diego, California 92093

Bruce Kapron‡ and Valerie King§

Department of Computer Science, University of Victoria, Victoria, Canada V8W 3P6

and

Mirosław Kutyłowski″

Heinz Nixdorf Institute and Department of Mathematics and Computer Science, University of Paderborn, D-33095 Paderborn, Germany

Received October 10, 1995; revised April 26, 1996

The ROBUST PRAM is a concurrent-read concurrent-write (CRCW) parallel random access machine in which any value might appear in a memory cell as a result of a write conflict. This paper addresses the question of whether a PRAM with such a weak form of write conflict resolution can compute functions faster than the concurrent-read exclusive-write (CREW) PRAM. We prove a lower bound on the time required by the ROBUST PRAM to compute Boolean functions in terms of the number of different values each memory cell of the PRAM can contain and the degree of the function when expressed as a polynomial over a finite field. In the case of 1-bit memory cells, our lower bound for the problem of computing the OR of n Boolean variables exactly matches Cook, Dwork, and Reischuk's upper bound on the CREW PRAM. We extend our result to obtain a lower bound, depending on the number of processors, for computing Boolean functions on the ROBUST PRAM, even with memory cells of unbounded size. A particular consequence is that the ROBUST PRAM with $2^{2^{O(\sqrt{\log n})}}$ processors requires $\Omega(\sqrt{\log n})$ steps to compute OR. These results are obtained by defining a class of CRCW PRAMs, the fixed adversary PRAMs, all of which are at least as powerful as the ROBUST PRAM. We prove our lower bounds using carefully chosen PRAMs from this class. We also show the limitations of this technique by describing how, with n-bit memory cells, any fixed adversary PRAM can compute OR and, more generally, simulate a PRIORITY PRAM in constant time. Finally, we consider the effect of adding randomization to the ROBUST PRAM. For any algorithm that computes OR without error, its expected running time on its worst input is no better than the worst case deterministic time complexity of computing OR. However, allowing a small probability of error enables the ROBUST PRAM with single bit memory cells to compute OR in almost constant time.

* E-mail: fich@cs.toronto.edu.
† E-mail: russell@cs.ucsd.edu.
‡ E-mail: bmkapron-csr.uvic.ca.
§ E-mail: val@csr.uvic.ca.
″ E-mail: mirekk@uni-paderborn.de.

1. INTRODUCTION

A parallel random access machine (PRAM) that allows concurrent writes must specify how to resolve write conflicts when they occur. One method is to let an adversary determine what value appears [1]. In other words, an algorithm must compute the correct answer no matter what values appear when write conflicts occur. Hagerup and Radzik [15] call this model the ROBUST PRAM. Such a model is very weak, since the way an adversary resolves write conflicts may depend on the entire algorithm and the values of all the inputs.

We define a *fixed adversary* PRAM to be a concurrent-read concurrent-write PRAM in which the value that appears in a given cell after a given time step can be expressed as a function of only the value already in the cell, the processors attempting to write to the cell, and the values they are attempting to write. This function may be different for different memory cells or different time steps. However, the function does not depend on what algorithm will be run. Essentially, the write conflict resolution mechanism is algorithm oblivious. By definition, any fixed adversary PRAM is at least as powerful as the ROBUST PRAM.

Many different write conflict resolution schemes for the PRAM have been studied. In the PRIORITY PRAM [8, 11], the processor of lowest index that attempts to write into a given cell at a given time step succeeds. When two or more processors simultaneously attempt to write into the same cell in the COLLISION PRAM [8], a special collision symbol appears. In the TOLERANT PRAM [12], the value of the cell remains unchanged in case of a write

0022-0000/96 $18.00

conflict. Another example is the MAXIMUM PRAM [4], in which the largest value written to a memory cell at a given time step is the value that appears there. These are all examples of fixed adversary PRAMs.

In the ARBITRARY PRAM [8, 10], an adversary is allowed to determine which one of the values that is written to a given cell will appear. Unlike the ROBUST PRAM, the adversary cannot leave the cell contents unchanged nor write some unrelated value when a write conflict occurs. The ARBITRARY PRAM is at least as powerful as the ROBUST PRAM. However, it is not an example of a fixed adversary PRAM, because the value chosen by the adversary may depend on the algorithm being executed.

The COMMON PRAM [8, 16] is a fixed adversary PRAM that can only run a restricted class of algorithms. In this model, many processors may simultaneously attempt to write to the same cell, provided they all attempt to write the same value. This value appears as the result of the write. The relationship between the computational powers of the COMMON and ROBUST PRAMs is not well understood. Specifically, it is unknown whether there are problems that can be solved substantially faster by the ROBUST PRAM than by the COMMON PRAM or vice versa.

Except for the ROBUST PRAM, every previously studied concurrent-read concurrent-write (CRCW) PRAM can easily compute the OR of n Boolean variables in constant time using n processors, whereas $\Omega(\log n)$ steps are necessary for the concurrent-read exclusive-write (CREW) PRAM, even with an unbounded number of processors [2]. It is unknown whether the ROBUST PRAM can compute OR any faster than the CREW PRAM. More generally, it is not known whether this very weak form of concurrent-write is useful for deterministically computing any function over a complete domain.

In contrast, over fractured domains, there are examples where the ROBUST PRAM can compute functions faster than the CREW PRAM. In particular, consider the problem of computing the OR of n input bits, when the input is known to contain at most k bits with value 1. The time complexity of this problem on the CREW PRAM is $\Theta(k)$ for $k \in O(\log n)$ [2, 5]. However, the ROBUST PRAM with n processors can compute this function in $O(\log k)$ steps [7]. This separation may be less significant than it first appears: the complexity of this restricted version of OR is $\Theta(\log n)$ on the CROW PRAM for all $k \geqslant 1$ [2], but the time complexity on the CROW PRAM of any problem over a complete domain differs by at most a constant from its time complexity on the CREW PRAM [17].

We address the question of the relative power of the ROBUST PRAM and the CREW PRAM for computing functions over complete domains. In Section 2, we prove a lower bound of $\Omega(\log d/\log kq)$ on the time required by the ROBUST PRAM to compute Boolean functions in terms of q^k, the number of different values each memory cell of the PRAM can contain, and the degree d of the function when expressed as a polynomial over the finite field of q elements. For almost all Boolean functions, including OR, this implies that the ROBUST PRAM with 1-bit memory cells requires $\Omega(\log n)$ steps. This is significant, since every Boolean function can be computed by the CREW PRAM with 1-bit memory cells in $O(\log n)$ steps [2]. We also derive a lower bound of $\Omega(\min\{\sqrt{\log d}, (\log d)/(\log q + \log\log p)\})$ steps for computing any Boolean function of degree d over $\mathbb{F}_q$ by the ROBUST PRAM with p processors, even if memory cells can contain arbitrarily large values. In particular, the ROBUST PRAM with $2^{2^{O(\sqrt{\log n})}}$ processors requires $\Omega(\sqrt{\log n})$ steps to compute OR. These lower bounds are obtained using carefully chosen fixed adversary PRAMs.

In Section 3, we show the limitations of these techniques, by describing how any fixed adversary PRAM with p processors and $O(\log(p/n))$-bit memory cells can compute OR in $O(\log n/\log\log(p/n))$ steps. In particular, with $2^{2^{\Omega(\sqrt{\log n})}}$ processors, $O(\sqrt{\log n})$ steps suffice. With $2^n - 1$ processors, only two steps and one n-bit memory cell suffice. This result gives rise to a simulation of the p-processor PRIORITY PRAM by any fixed adversary PRAM using only a constant factor more time and a factor of p more memory cells, but with an exponential increase in the number of processors.

Adding randomization to the ROBUST PRAM increases its computational power. Borodin, Hopcroft, Paterson, Ruzzo, and Tompa [1] showed that the randomized ROBUST PRAM can compute OR in $O(\log\log n)$ time, with any constant probability of error. Hagerup and Radzik [15] also showed that $O(\log\log n)$ time is sufficient to compute OR, using only n processors and with error probability $2^{-(\log n)^{O(1)}}$. In Section 4, we present a new randomized ROBUST PRAM algorithm that uses only $O(\log^* n)$ time and $n/(\log^* n)$ processors and has error probability $2^{-O(2^{(\log^* n)/4})}$. Together with our lower bounds, these results prove a separation between the deterministic and randomized ROBUST PRAM. For the CREW PRAM, no such separation exists, since randomization provides no more than a factor of 8 in speedup over the deterministic model [3].

Throughout this paper, we use $P_1, ..., P_p$ to denote the p processors of a PRAM and $M_1, ..., M_m$ to denote its m memory cells. If the input consists of n variables, $x_1, ..., x_n$, we assume that they are initially located in memory cells $M_1, ..., M_n$, respectively. The other memory cells are initialized to 0. Each step of a PRAM computation is assumed to consist of three phases. In the first phase, each processor can read from a cell of shared memory; in the second phase, processors are allowed to do an arbitrary amount of local computation; and in the third phase, each processor can attempt to write to a cell of shared memory. Formal definitions of the PRAM model can be found in [2, 3].

2. LOWER BOUNDS

Given any field F and any function $f: D \to R$, where $D \subseteq F^n$ and $R \subseteq F$, the polynomial $g: F^n \to F$ *represents* f *over* F if $g(x) = f(x)$ for all $x \in D$. In particular, every Boolean function can be uniquely expressed as a multilinear polynomial (i.e., as a sum of monomials) over any field [18]. The *degree* of f over F is defined to be the minimum degree of any polynomial that represents f over F. The degree of f over F is undefined if there is no polynomial that represents f over F.

In [3], it was shown that, on the CREW PRAM, the time complexity of computing any Boolean function of degree d over the field of real numbers is $\Theta(\log d)$. Here, using a similar approach, we derive lower bounds for the time to compute a Boolean function on a particular fixed adversary PRAM (and hence the ROBUST PRAM) in terms of its degree $\deg_q$ over the finite field $\mathbb{F}_q$ of q elements.

THEOREM 2.1. *On the ROBUST PRAM with memory cells that can hold at most q^k different values, $\Omega(\log d/\log qk)$ steps are required to compute any Boolean function of degree d over $\mathbb{F}_q$.*

To prove this result, it suffices to prove the lower bound on any fixed adversary PRAM, each of whose memory cells holds a k-tuple of elements in $\mathbb{F}_q$. The key is to choose a fixed adversary PRAM with nice properties.

When memory cells can contain only single bits (i.e., when $q = 2$ and $k = 1$), we use the *$\mathbb{F}_2$-adversary PRAM*. In this model, the value of a shared memory cell changes exactly when an odd number of processors simultaneously attempt to write the negated value. In other words, if a memory cell contains the value 0 (1) before a given step, then it will contain the value 1 (0) after the step, if there are an odd number of processors that attempt to write the value 1 (0) there during the step, and it will remain 0 (1) otherwise.

For larger values of q, the write conflict resolution rule can be generalized. Specifically, in the *$\mathbb{F}_q$-adversary PRAM*, if a memory cell M contains the value $v \in \mathbb{F}_q$ before a given step, then, after that step, M will contain the value

$$v + \sum_{u \in \mathbb{F}_q} (u - v) \cdot (\text{the number of processors that attempt to write the value } u \text{ to } M),$$

where all operations are performed in the field $\mathbb{F}_q$. Note that, if no processors write to M, then the sum is empty and M will retain its old value v. If all the processors that write to M attempt to write v, M also retains the value v. However, if exactly one processor writes to M, then M will contain the value that processor wrote.

The $\mathbb{F}_q$-adversary PRAM works component-wise for $k > 1$, treating each component of the k-tuple in each memory cell separately. Short tuples can be padded with leading zeros when necessary, so the actual number of components k does not need to be specified. However, when k is bounded, it is possible to derive lower bounds for the time to compute Boolean functions on the $\mathbb{F}_q$-adversary PRAM.

LEMMA 2.2. *On the $\mathbb{F}_q$-adversary PRAM with memory cells that hold k-tuples of elements in $\mathbb{F}_q$, $\Omega((\log d)/\log(qk))$ steps are required to compute any Boolean function of degree d over $\mathbb{F}_q$.*

Proof. Consider any $\mathbb{F}_q$-adversary PRAM algorithm for memory cells that hold k-tuples of elements in $\mathbb{F}_q$. Without loss of generality, we may assume that a processor's state is merely (an encoding of) the sequence of values it has read at each step (so a processor never forgets information). Given an input $x = (x_1, ..., x_n) \in \{0, 1\}^n$, let

$$\mathscr{S}_{P,t,w}(x) = \begin{cases} 1 & \text{if processor } P \text{ is in state } w \text{ immediately after step } t, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\mathscr{S}_{P,t,w}(x)$ is the characteristic function describing those inputs x for which processor P is in state w at time t. Note that for different states w and w', the set of inputs for which processor P is in state w at time t is disjoint from the set of inputs for which processor P is in state w' at time t. Thus for any set of states W, $\sum_{w \in W} \mathscr{S}_{P,t,w}(x)$ is the characteristic function of the set of inputs for which processor P is in a state of W at time t.

Let $\mathscr{C}_{M,t,i}(x)$ denote the contents of the ith component of memory cell M immediately after step t on input x and let

$$\mathscr{B}_{M,t,i,u}(x) = \begin{cases} 1 & \text{if } \mathscr{C}_{M,t,i}(x) = u, \\ 0 & \text{otherwise} \end{cases}$$

be the characteristic function of the set of inputs for which the ith component of memory cell M has value u at time t. Define

$$s(t) = \max_{P,w} \deg_q(\mathscr{S}_{P,t,w}),$$

$$c(t) = \max_{M,i} \deg_q(\mathscr{C}_{M,t,i}),$$

$$b(t) = \max_{M,i,u} \deg_q(\mathscr{B}_{M,t,i,u}),$$

to be the maximum degrees of these functions over $\mathbb{F}_q$.

Since each processor is in its initial state at time 0 for all inputs, the associated characteristic functions are all

constant, so $s(0)=0$. The initial contents of the least significant components of the memory cells $M_1, ..., M_n$ are described by the linear functions $x_1, ..., x_n$, respectively. The other components of memory cells $M_1, ..., M_n$ and all k components of any other memory cells are all initially 0 and their contents are described by the constant 0 function. Furthermore, since the inputs are binary, it follows that, for $i=1, ..., k$,

$$\mathscr{B}_{M,0,i,1}(x)=\mathscr{C}_{M,0,i}(x),$$
$$\mathscr{B}_{M,0,i,0}(x)=1-\mathscr{C}_{M,0,i}(x),$$
$$\mathscr{B}_{M,0,i,u}(x)=0 \qquad \text{if} \quad u\in \mathbb{F}_q-\{0,1\}.$$

Therefore, $c(0)=b(0)=1$.

Processor P is in state $\langle v^{(1)}, ..., v^{(t-1)}, v\rangle$ immediately after step t if and only if it is in state $\langle v^{(1)}, ..., v^{(t-1)}\rangle$ immediately after step $t-1$ and the memory cell M that it reads during step t contains value $v=(v_k, ..., v_1)$. Hence

$$\mathscr{S}_{P,t,\langle v^{(1)}, ..., v^{(t-1)}, v\rangle}(x) = \mathscr{S}_{P,t-1,\langle v^{(1)}, ..., v^{(t-1)}\rangle}(x)\cdot\prod_{i=1}^{k}\mathscr{B}_{M,t-1,i,v_i}(x)$$

and $s(t)\leqslant s(t-1)+kb(t-1)$ for $t>0$.

Now consider the writing phase of step t. Let $W(P, M, t, i, u)$ denote the set of states in which processor P writes the value u to the ith component of memory cell M at step t. Then $\sum_{w\in W(P,M,t,i,u)}\mathscr{S}_{P,t,w}(x)$ has value 1, if processor P attempts to write the value u to the ith component of memory cell M at step t on input x, and has value 0, otherwise. It follows from the definition of the write conflict resolution rule that the ith component of the contents of memory cell M at the end of step t can be expressed as

$$\mathscr{C}_{M,t,i}(x)=\mathscr{C}_{M,t-1,i}(x)+\sum_{u\in\mathbb{F}_q}(u-\mathscr{C}_{M,t-1,i}(x))\times\sum_{P}\left(\sum_{w\in W(P,M,t,i,u)}\mathscr{S}_{P,t,w}(x)\right),$$

when viewed as a polynomial over $\mathbb{F}_q$. Thus $c(t)\leqslant c(t-1)+s(t)$ for $t>0$.

Finally, over $\mathbb{F}_q$,

$$\mathscr{B}_{M,t,i,u}(x)=1-(\mathscr{C}_{M,t,i}(x)-u)^{q-1}=\begin{cases}1 & \text{if} \quad \mathscr{C}_{M,t,i}(x)=u\\ 0 & \text{otherwise,}\end{cases}$$

so $b(t)\leqslant(q-1)\,c(t)$.

It is easily shown by induction that

$$s(t)\leqslant\frac{k(q-1)}{\Delta}\left[\left(\frac{k(q-1)+2+\Delta}{2}\right)^t-\left(\frac{k(q-1)+2-\Delta}{2}\right)^t\right],$$
$$c(t)\leqslant\frac{k(q-1)+\Delta}{2\Delta}\left(\frac{k(q-1)+2+\Delta}{2}\right)^t-\frac{k(q-1)-\Delta}{2\Delta}\left(\frac{k(q-1)+2-\Delta}{2}\right)^t,$$

where $\Delta=\sqrt{k(q-1)[k(q-1)+4]}$.

If an algorithm computes a Boolean function f, then the contents of the least significant component of the output cell at the end of the computation is described by the function f. Hence the number of steps t taken by the algorithm satisfies $c(t)\geqslant\deg_q(f)$, which implies that $t\geqslant\log(2\Delta\deg_q(f)/k(q-1)+\Delta)/\log(k(q-1)+2+\Delta/2)\in\Omega(\log(\deg_q(f))/\log(qk))$. ∎

In the special case when the memory cells can contain only single bits and f is a Boolean function, $c(t)\leqslant F_{2j+1}$, where $F_0, F_1, F_2, ...$ is the Fibonacci sequence defined by the recurrence $F_0=1, F_1=1$, and $F_j=F_{j-1}+F_{j-2}$ for $j\geqslant 2$. Thus it follows that at least $\varphi(\deg_2(f))$ steps are required to compute the Boolean function f, where $\varphi(d)=\min\{j|F_{2j+1}\geqslant d\}$. In particular, since the OR of n Boolean variables has degree n over $\mathbb{F}_2$, the $\mathbb{F}_2$-adversary PRAM and the ROBUST PRAM both require $\varphi(n)$ steps to compute this function with single bit memory cells. This lower bound exactly matches the upper bound for computing OR on the CREW PRAM with single bit memory cells [2].

The threshold k function of n Boolean variables and the exactly k out of n function both have degree at least $n/2$ over $\mathbb{F}_2$. In fact, over $\mathbb{F}_2$, only a very tiny fraction (at most $1/2^{2n-1}$) of all polynomials of n variables have degree less than $n/2$. Hence, most Boolean functions of n arguments require at least $\varphi(n)-1$ steps to be computed by the ROBUST PRAM with single bit memory cells.

Any nonconstant Boolean function f satisfies $|f^{-1}(0)|, |f^{-1}(1)|\geqslant 2^{n-\deg_2(f)}$ [9]. Thus, if $0<|f^{-1}(0)|\leqslant 2^{n-d}$ or $0<|f^{-1}(1)|\leqslant 2^{n-d}$, then $\deg_2(f)\geqslant d$, so the ROBUST PRAM with 1-bit memory cells requires at least $\varphi(d)$ steps to compute f.

The PARITY function of n variables has degree 1 over $\mathbb{F}_2$. However, over $\mathbb{F}_3$, it has degree n. It follows from the bound derived in the proof of Lemma 2.2 that a ROBUST PRAM whose memory cells can hold at most three different values requires at least $0.45\log_2 n$ steps to compute PARITY.

It is also possible to apply Lemma 2.2 to obtain lower bounds on the ROBUST PRAM with memory cells that

can hold arbitrarily large values. This involves showing that, for each algorithm, a variant of the $\mathbb{F}_q$-adversary PRAM can be constructed which does not need to introduce too many new values. Initially, each memory cell contains a single bit. However, as a computation on the $\mathbb{F}_q$-adversary PRAM proceeds, the number of different values that may appear in a memory cell can increase. Part of this increase may be due to the program itself; part may be due to the actions of the adversary. We show that, for each algorithm, a variant of the $\mathbb{F}_q$-adversary PRAM can be constructed which does not introduce too many new values.

For the ROBUST PRAM, the actual values written to and read from shared memory are not important. They can be renamed essentially arbitrarily without affecting the number of steps performed. Specifically, let A be any ROBUST PRAM algorithm and let h: $\mathbb{N} \to \mathbb{N}$ be any bijection of the natural numbers that maps 0, each possible input value, and each possible output value to itself. Then, construct a new algorithm $h(A)$ in which every processor applies the function h to each value it attempts to write into shared memory and applies the function h^{-1} to each value it reads from shared memory. Because the input and output values are not affected by the renaming, the algorithms A and $h(A)$ compute the same function. Furthermore, since processors are allowed to perform an unlimited amount of local computation at each step, $h(A)$ uses no more steps than A.

LEMMA 2.3. *Consider any ROBUST PRAM algorithm A computing a Boolean function using p processors. There is a renaming function h such that, during the first t steps of $h(A)$ on the $\mathbb{F}_q$-adversary PRAM, all except the $\lfloor 2^t \log_q(2pq) \rfloor$ least significant components of every memory cell are* 0 *(when integers are represented in q-ary notation) and, after step t, every processor can be in at most $(2pq)^{2^t-1}$ different states.*

Proof. The proof is by induction on t. Initially, the only values in shared memory are 0 and 1 and every processor is in its one initial state. Since $1 \leqslant \lfloor 2^0 \log_q(2pq) \rfloor$ and $1 = (2pq)^{2^0-1}$, the claim is true for $t = 0$ with h as the identity function.

Let $t \geqslant 0$ and assume the claim is true for t with renaming function h. Consider step $t+1$ of $h(A)$ on the $\mathbb{F}_q$-adversary PRAM. At the beginning of this step, each processor is in at most $(2pq)^{2^t-1}$ states and, in each state, it can read one of at most q^e different values, where $e = \lfloor 2^t \log_q(2pq) \rfloor$. Therefore, at the end of step $t+1$, every processor is in at most $(2pq)^{2^t-1} \cdot q^e \leqslant (2pq)^{2^t-1} \cdot (2pq)^{2^t} = (2pq)^{2^{t+1}-1}$ states.

Let V denote the set of those values greater than or equal to q^e that processors attempt to write to shared memory during this step, over all possible inputs. Since each processor can write at most $(2pq)^{2^t-1} \cdot q^e$ different values during time step $t+1$, it follows that $|V| \leqslant p \cdot q^e \cdot (2pq)^{2^t-1}$. Let h' be any bijection that maps the same element as h does to each of $0, ..., q^e - 1$ and maps each element in V to the range $\{q^e, ..., q^e + |V| - 1\}$.

Note that every execution of $h'(A)$ on the $\mathbb{F}_q$-adversary PRAM is identical to an execution of $h(A)$ on the $\mathbb{F}_q$-adversary PRAM for the first t steps. Furthermore, each value a processor attempts to write during step $t+1$ of $h'(A)$ on the $\mathbb{F}_q$-adversary PRAM has value less than $q^e + |V|$, so all except its $\lceil \log_q(q^e + |V|) \rceil \leqslant e + 1 + \lfloor \log_q(p(2pq)^{2^t-1}) \rfloor \leqslant \lfloor 2^{t+1} \log_q(2pq) \rfloor$ least significant q-ary digits are 0. It follows from the definition of the $\mathbb{F}_q$-adversary PRAM that all except the $\lfloor 2^{t+1} \log_q(2pq) \rfloor$ least significant components of every memory cell are 0 at the end of step $t+1$. Thus the claim is true for $t+1$ with renaming function h'. ∎

THEOREM 2.4. *With p processors (and memory cells that can contain arbitrarily large values), the ROBUST PRAM requires $\Omega(\min\{\sqrt{\log d}, (\log d)/(\log q + \log\log p)\})$ steps to compute any Boolean function of degree d over $\mathbb{F}_q$.*

Proof. Let f be a Boolean function of degree d over $\mathbb{F}_q$ that can be computed by a ROBUST PRAM algorithm A in T steps. By Lemma 2.3, there is a renaming function h such that when $h(A)$ is run on the $\mathbb{F}_q$-adversary PRAM, all except the $\lfloor 2^T \log_q(2pq) \rfloor$ least significant components of every memory cell are 0. Since $h(A)$ computes f, it follows from Lemma 2.2 that $T \in \Omega((\log d)/\log(q2^T \log_q(2pq)))$ or, equivalently, $T^2 + T(\log q + \log\log p) \in \Omega(\log d)$. Hence $T \in \Omega(\min\{\sqrt{\log d}, (\log d)/(\log q + \log\log p)\})$. ∎

In particular, since $\deg_2(\mathrm{OR}) = n$, the ROBUST PRAM with $2^{2^{O(\sqrt{\log n})}}$ processors requires $\Omega(\sqrt{\log n})$ steps to compute the OR of n bits. To compute OR in constant time, the ROBUST PRAM requires at least $2^{n^{\Omega(1)}}$ processors. Moreover, from Theorem 2.1, its shared memory cells must be capable of holding $n^{\Omega(1)}$-bit numbers.

3. CONSTANT TIME UPPER BOUNDS FOR FIXED ADVERSARY PRAMS

It is not known whether the ROBUST PRAM can compute the OR of n bits faster than the CREW PRAM, even if it is allowed an unlimited number of processors and an unlimited number of memory cells that can contain arbitrarily large integers. In this section, we show that any fixed adversary PRAM can even simulate the PRIORITY PRAM with only a constant factor increase in time, given sufficient resources. In particular, this indicates that new proof techniques will be needed to prove that the ROBUST PRAM is no more powerful than the CREW PRAM.

The important resources are the number of processors and memory wordsize. The *wordsize* of a memory cell is the number of bits needed to represent the values it may

contain. Specifically, if a memory cell can hold w different values, then its wordsize is at least $\lceil \log_2 w \rceil$.

An algorithm for computing OR is the key to the simulation.

THEOREM 3.1. *The OR of n bits can be computed in* 2 *steps on any fired adversary PRAM with* $2^n - 1$ *processors and one memory cell of wordsize n.*

Proof. Let $p = 2^n - 1$. For any subset $W \subseteq \{1, 2, ..., p\}$, let $v(W)$ denote the value that appears in the memory cell after step 1 when each processor whose index is in W attempts to write its index to the memory cell and the remaining processors do not attempt to write. The memory cell is assumed to be initialized to 0.

If no processors attempt to write, the contents of the memory cell remain unchanged and if exactly one processor attempts to write, the memory cell will contain the index of that processor. Thus the range of the function v has size at least $p + 1$.

We will partition the processors into three sets, A, N, and D, with $|D| = n$, and show that there exists a subset $S \subseteq D$ such that $v(A \cup S) \neq v(A \cup S')$ for all subsets $S' \subseteq D$ that differ from S. Using these sets, the following algorithm computes the OR of n Boolean variables. In the read phase of the first step, each processor in D is assigned a different bit of input to read. Processors in A will *always* attempt to write their indices, processors in N will *never* write, and each processor in D will write its index *depending* on the value of the input bit it read. A processor in S attempts to write its index if it reads the value 0 and a processor in $D - S$ attempts to write its index if it reads the value 1.

If the input bits are all 0, then the processors in $A \cup S$ are exactly those that attempt to write and, as a result, the memory cell will contain the value $v(A \cup S)$. Otherwise, the set of processors attempting to write is $A \cup S'$ for some subset $S' \subseteq D$ that differs from S. In this case the memory cell will contain a value other than $v(A \cup S)$.

Therefore, in the next step, any predetermined processor can complete the computation by reading the contents of the memory cell and writing the value 0 if it saw the value $v(A \cup S)$ and writing the value 1 if it saw anything else.

Now we show how to construct sets A, N, and S satisfying the desired properties. Since the function v has a range of size at least $p + 1$, there is at least one value r in the range that is the image of at most $2^p/(p+1)$ different subsets in the domain. Let $\mathscr{C}$ initially be the collection of subsets that map to r (i.e., $\mathscr{C} = v^{-1}(r)$).

While there are at least two subsets remaining in $\mathscr{C}$, choosy a processor P that occurs in some but not all of the subsets in $\mathscr{C}$. If the majority of subsets in $\mathscr{C}$ contain P, add P to N and remove any subset that contains P from $\mathscr{C}$. Otherwise, add P to A, and remove any subset that does not contain P from $\mathscr{C}$.

At each step, $\mathscr{C}$ is reduced in size by at least a factor of 2. After no more than $\log(2^p/(p+1)) = p - \log(p+1) = p - n$ iterations, only one subset remains and $|A \cup N| \leqslant p - n$. The subsets remaining in $\mathscr{C}$ are those that map to r, contain A, and are disjoint from N.

Let L be the last subset of processors in $\mathscr{C}$. Let $S = L - A$. If S contains at most n processors, let D be any set of n processors that contains S and is disjoint from $A \cup N$. Otherwise move processors from S to A until S has size n and then let $D = S$. Since $A \cup S = L \in \mathscr{C}$, $v(A \cup S) = r$.

Now suppose $S' \subseteq D$ and $v(A \cup S') = r$. Then $A \cup S'$ was originally in $\mathscr{C}$. But $A \cup S'$ contains no processors in N and every processor in A; therefore it was never removed from $\mathscr{C}$. Since L is the only subset in $\mathscr{C}$ at the end of the construction, $A \cup S' = L$ and, hence, $S' = S$. Therefore $v(A \cup S') \neq r$ for all $S' \subseteq D$ such that $S' \neq S$.

COROLLARY 3.2. *The OR of n bits can be computed in* $O(\log n / \log\log(p/n))$ *steps on any fixed adversary PRAM with p processors and wordsize* $\log(p/n) + \log\log(p/n)$.

Proof. Divide the bits into groups of size $s = \log(p/n) + \log\log(p/n)$ and assign $2^s - 1$ processors to each group. Since there are at most n/s groups and $(2^s - 1)\, n/s < n(p/n)\log(p/n)/(\log(p/n) + \log\log(p/n)) < p$, there is a sufficient number of processors available. By Theorem 3.1, the OR of each group can be computed in $O(1)$ steps, leaving a problem of size n/s.

If this process is repeated $t = \log(n)/\log\log, (p/n)$ times, a problem of size $n/s^t = 1$ remains, which is the solution to the original problem. ▮

In particular, the OR of n bits can be computed in $O(\sqrt{\log n})$ steps using $2^{2^{\Omega(\sqrt{\log n})}}$ processors.

THEOREM 3.3. *The PRIORITY PRAM with p processors and m memory cells can be simulated for t steps by any fixed adversary PRAM with* $2^p - 1$ *processors and* $m(p+1)$ *memory cells in* $O(t)$ *steps.*

Proof. Processor P_i is simulated by a team of 2^{i-1} processors, one of which is designated P'_i. Each memory cell M_j is simulated using one memory cell M'_j and p auxiliary memory cells $M'_{j,1}, ..., M'_{j,p}$ that are initialized to 0.

When processor P_i reads M_j, the ith team of processors all read M'_j and they all perform the same local computations. When processor P_i attempts to write to M_j, processor P'_i writes the value 1 in location $M'_{j,i}$. Then the ith team computes the OR of the values in locations $M'_{j,1}, ..., M'_{ji-1}$ (using the algorithm in Theorem 3.1) and writes the answer in $M'_{j,i}$. Processor P'_i reads $M'_{j,i}$ and, if it contains the value 0, writes the value P_i attempted to write to location M'_j. Otherwise P'_i does not write anything. Each $M'_{j,i}$ is then reset to 0. ▮

4. RANDOMIZED ALGORITHMS FOR COMPUTING OR ON THE ROBUST PRAM

In this section, we describe a randomized algorithm for computing OR in $O(\log^* n)$ time on the ROBUST PRAM with single bit shared memory cells. The algorithm has one-sided error: when the value 1 is output it is always correct; when 0 is output, it errs with probability $o(1)$. Virtually the same algorithm may be used to simulate a step of the ARBITRARY PRAM with the same error and time bounds, provided memory cells can contain at least p different values.

Hagerup and Radzik [15] describe a randomized algorithm to simulate a step of the ARBITRARY PRAM in time $O(\log\log n)$ with error at most $1/n$, using $O(n \log n/\log\log n)$ processors. Our algorithm uses a routine REDUCE which is similar to theirs, but replaces their randomized "scattering" technique with a simple deterministic method. To achieve the better time bounds, our algorithm involves the repeated application of REDUCE, which compounds the error.

The idea of our algorithm is to sample varying amounts of the input in parallel, in such a way that, when the input contains a 1, we find a sample in which a small number of input bits are 1.

Let $X = \{x_1, x_2, ..., x_n\}$ be the set of input bits. Using at most $n(1+\log_2 n)^4$ processors and a constant number of parallel steps, REDUCE(X) reduces the input set to a set Y containing at most $4(1+\log_2 n)^4$ bits, so that if $\mathrm{OR(X)} = 0$ then $\mathrm{OR(Y)} = 0$, and if $\mathrm{OR(X)} = 1$ then, with probability less than $1/n$, $\mathrm{OR(Y)} = 0$.

REDUCE (X). Perform the following $\lceil \log_2 n \rceil$ times, independently and in parallel:

1. For each of the n input bits x_i, associate a group of $(\lfloor \log_2 n \rfloor + 1)^3$ processors $\{P_{i,j,k,l} | 0 \leqslant j, k, l \leqslant \lfloor \log_2 n \rfloor\}$, all of which read x_i.

2. Consider a matrix A with n rows and $\lfloor \log_2 n \rfloor + 1$ columns. For each input bit x_i, one processor in its group, say $P_{i,j,1,1}$, writes the value 1 to $A(i, j)$ with probability $1/2^j$, provided $x_i = 1$; otherwise, it writes the value 0. Note that if $x_i = 0$, then the entire ith row of A is 0.

3. For each column j, initialize a $(\lfloor \log_2 n \rfloor + 1) \times 2 \times (\lfloor \log_2 n \rfloor + 1) \times 2$ array B_j to 0. This can be accomplished by having processor $P_{1,j,k,l}$ write the value 0 to $B_j(k, 0, l, 0)$, $B_j(k, 0, l, 1)$, $B_j(k, 1, l, 0)$, and $B_j(k, 1, l, 1)$.

4. Each processor $P_{i,j,k,l}$ reads $A(i, j)$ and, if $A(i, j) = 1$, writes the value 1 to $B_j(k, i_k, l, i_l)$, where $i_{\lfloor \log_2 n \rfloor + 1} \cdots i_0$ is the binary representation of i.

The $4(\lfloor \log_2 n \rfloor + 1)^3 \lceil \log_2 n \rceil$ element output array Y is created by concatenating the entries in all the B arrays created during all of the parallel executions.

If $\mathrm{OR}(X) = 0$, then the value 1 is never written and the output Y is entirely 0. If $\mathrm{OR}(X) = 1$, the output Y should contain at least one 1. We show that this happens with probability at least $1 - 1/n$.

LEMMA 4.1. *If $OR(X) = 1$, then Y fails to contain a 1 with probability at most $1/n$.*

Proof. Assume $\mathrm{OR}(X) = 1$. Let r be the number of ones contained in X. Consider any one of the parallel executions and let z be the number of ones in the $\lfloor \log_2 r \rfloor$th column of A.

The probability that $z = 0$ is $(1 - 1/2^{\lfloor \log_2 r \rfloor})^r \leqslant 1/e$. The expected value of z is $r(1/2^{\lfloor \log_2 r \rfloor}) < 2$. Chernoff bounds show that the probability of z being at least three times its expected value is less than $1/e^4$. Hence $Pr[1 \leqslant z \leqslant 5] > 1 - 1/e - 1/e^4 > 1/2$.

Suppose that $1 \leqslant z \leqslant 5$. In this case, some cell of $B_{\lfloor \log_2 r \rfloor}$ will have the value 1 written to it by exactly one processor. To see this, consider the indices i such that $A(i, \lfloor \log_2 r \rfloor) = 1$. Then there exist two bits which distinguish one of these indices from the others.

The output Y fails to contain a 1 only when all the $\lceil \log_2 n \rceil$ parallel executions fail to produce z within the required range. If $\mathrm{OR}(X) = 1$, the probability of these $\lceil \log_2 n \rceil$ independent events all occurring is less than $1/n$. ∎

Independently, Hagerup [13] developed a similar technique, *graduated conditional scattering*, that can be used to reduce the problem of computing the OR from n bits to $O((\log n)^2)$ bits with n processors and error probability $O(1/\log n)$.

After a sufficient number of applications of REDUCE, the original problem is reduced to computing the OR of a very small number of bits, which may be done deterministically. The resulting algorithm is always correct when it outputs 1, although it may err when it outputs 0.

LEMMA 4.2. *The OR of n bits can be computed by a ROBUST PRAM with $O(n(\log n)^4)$ processors and 1-bit shared memory cells in $O(\log^* n)$ steps using a randomized algorithm that errs with probability $O(1/2^{4(\log^* n)})$.*

Proof. Let $h(n)$ denote the minimum integer h such that $\log^{(h)} n \leqslant 2^{\log^* n}$. Then $h(n) \leqslant \log^* n$. Iterate REDUCE $h(n)$ times. The size of the remaining problem is in $\Theta((\log^{(h(n))} n)^4) \subseteq O(2^{4 \log^* n})$. Solve this problem using a deterministic CREW PRAM algorithm in $O(\log^* n)$ steps [2]. The failure probability of this algorithm is at most $1/n + \sum_{h=1}^{h(n)-1} 1/\Omega((\log^{(h)} n)^4) = O(1/(\log^{(h(n)-1)} n)^4) \subseteq O(1/2^{4 \log^* n})$. ∎

Using the standard technique of partitioning a problem into successively larger subproblems, the number of processors can be reduced to n and the error probability can be improved.

LEMMA 4.3. *The OR of n bits can be computed by a ROBUST PRAM with n processors and* 1*-bit shared memory cells in* $O(\log^* n)$ *steps using a randomized algorithm that errs with probability* $O(1/2^{2^{(\log^* n)/4}})$.

Proof. First partition the bits into $n/2^{\log^* n}$ sets of size $2^{\log^* n}$ and use a deterministic CREW PRAM algorithm to compute the OR of each set in $O(\log^* n)$ steps. This leaves a subproblem of size $n/2^{\log^* n}$.

Let $z_0 = 2^{\log^* n}$, and for $i \geqslant 0$, let $z_{i+1} = 2^{z_i^{1/4}-3}$. In the $(i+1)$th iteration, divide the remaining n/z_i bits into $n/z_i s_i$ subproblems each of size $s_i = 2^{z_i^{1/4}-1}$ and allocate $s_i(1+\log_2 s_i)^4 = s_i z_i$ processors to each. Apply REDUCE to each subproblem in parallel. This reduces each subproblem from size s_i to size $4(1+\log_2 s_i)^4 = 4z_i$. Therefore, at the end of the ith iteration, $4n/s_i = n/z_{i+1}$ bits remain. After $t = \min\{i | z_i \geqslant n\}$ iterations, the algorithm terminates.

Provided n is sufficiently large, $z_{i+2} \geqslant 2^{z_i}$ for all $i \geqslant 0$. Therefore $t \in O(\log^* n)$. Each iteration takes constant time. Hence, the time complexity of this algorithm is in $O(\log^* n)$.

The probability of error is the probability that some 1 in the input fails to appear in any of the reductions. From the Lemma 4.1, it follows that the total probability of error is at most $\sum_{i=1}^{k} 1/s_{i-1} \in O(1/2^{z_0^{1/4}}) = O(1/2^{2^{(\log^* n)/4}})$. ∎

There is another way to obtain an algorithm for computing OR that runs in $O(\log^* n)$ time and uses n processors: repeatedly apply graduated conditional scattering until the problem size is $2^{\Theta(\log^* n)}$ and then solve the remaining problem deterministically, as in the proof of Lemma 4.2 [14]. The resulting algorithm has error probability $2^{-O(\log^* n)}$.

By first partitioning the bits into groups of size $\Theta(\log^* n)$ and computing the OR of each group sequentially, the number of processors can be reduced from n to $O(n/\log^* n)$ without increasing the time by more than a constant factor. The two resulting algorithms are efficient in the sense that their time-processor products are $\Theta(n)$, the sequential complexity of computing OR.

THEOREM 4.4. *The OR of n bits can be computed by a ROBUST PRAM with* $O(n/\log^* n)$ *processors and* 1*-bit shared memory cells in* $O(\log^* n)$ *steps using a randomized algorithm that errs with probability* $2^{-O(2^{(\log^* n)/4})}$.

ACKNOWLEDGMENTS

We thank Martin Dietzfelbinger, Torben Hagerup, and Charlie Rackoff for helpful discussion. This work was supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, the Defense Advanced Research Projects Agency of the United States, NEC, and the Deutsche Forschungsgemeinschaft.

REFERENCES

1. A. Borodin, J. Hopcroft, M. Paterson, L. Ruzzo, and M. Tompa, Observations concerning synchronous parallel models of computation, manuscript, 1980.
2. S. Cook, C. Dwork, and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **15** (1986), 87–97.
3. M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk, Exact time bounds for computing boolean functions without simultaneous writes, *in* "Proceedings, Second Annual ACM Symposium on Parallel Algorithms and Architectures, 1990," pp. 125–135.
4. D. Eppstein and Z. Galil, Parallel algorithmic techniques for combinatorial computing, *Annu. Rev. Comput. Sci.* **3** (1988), 233–283.
5. F. Fich, The complexity of computation on the parallel random access machine, *in* "Synthesis of Parallel Algorithms" (J. Reif, Ed.), Morgan Kaufmann, San Mateo, CA, 1993.
6. F. 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* "Proceedings, 10th Annual Symposium on Theoretical Aspects of Computer Science, 1993," pp. 386–397.
7. F. Fich, M. Kowaluk, M. Kutyłowski, K. Loryś, and P. Ragde, Retrieval of scattered information by EREW, CREW and CRCW PRAMs, *in* "Proceedings, Third Scandinavian Workshop on Algorithm Theory, 1992," pp. 30–41.
8. F. Fich, P. Ragde, and A. Wigderson, Relations between concurrent-write models of parallel computation, *SIAM J. Comput.* **17** (1988), 606–627.
9. P. Fischer and H. U. Simon, On learning ring-sum-expansions, *in* "Proceedings, Third Workshop on Computational Learning Theory, 1990," *SIAM J. Comput.* **21** (1992), 181–192.
10. S. Fortune and J. Wyllie, Parallelism in random access machines, *in* "Proceedings, Annual Symposium on Theory of Computing, 1978," pp. 114–118.
11. L. Goldschlager, A unified approach to models of synchronous parallel machines, *J. Assoc. Comput. Mach.* **29** (1982), 1073–1086.
12. V. Grolmusz and P. Ragde, Incomparability in parallel computation, *Discrete Appl. Math.* **29** (1990), 63–78.
13. T. Hagerup, Fast and optimal simulations between CRCW PRAMs, *in* "Proceedings, Annual Symposium on Theoretical Aspects of Computer Science, 1992," pp. 45–48.
14. T. Hagerup, Personal communication.
15. T. Hagerup and T. Radzik, Every robust CRCW PRAM can efficiently simulate a PRIORITY PRAM, *in* "Proceedings, Second Annual ACM Symposium on Parallel Algorithms and Architectures, 1990," pp. 117–124.
16. L. Kucera, Parallel computation and conflicts in memory access, *Inform. Process Lett.* **14** (1982), 93–96.
17. N. Nisan, CREW PRAMs and decision trees, *in* "Proceedings, 21st Annual ACM Symposium on Theory of Computing, 1989," pp. 327–335.
18. R. Smolensky, Algebraic methods in the theory of lower bounds for Boolean circuit complexity, *in* "Proceedings, 19th Annual ACM Symposium on Theory of Computing, 1987," pp. 77–82.