[20] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation on polynomials using few processors. *SIAM J. Comput.*, **12**(4):641–644, 1983.
[21] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. Assoc. Comput. Math.*, **23**(4):733–742, 1976.
[22] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *Proc. Third IEEE Symposium on Logic Programming*, 1986 (preliminary version also presented in Workshop on Foundations of Deductive Databases and Logic Programming, Washington, DC, 1986).
[23] M. Vardi. Complexity of relational queries. In *14th ACM Symposium on Theory of Computing*, pp. 137–145, 1982.
[24] M. Y. Vardi. Querying logical databases. In *4th PODS*, pp. 57–65, 1985.

Algorithmica (1988) 3: 43–51

Algorithmica

Simulations Among Concurrent-Write PRAMs[1]

Faith E. Fich,[2] Prabhakar Ragde,[2] and Avi Wigderson[3]

Abstract. This paper is concerned with the relative power of the two most popular concurrent-write models of parallel computation, the PRIORITY PRAM [G], and the COMMON PRAM [K]. Improving the trivial and seemingly optimal $O(\log n)$ simulation, we show that one step of a PRIORITY machine can be simulated by $O(\log n/(\log\log n))$ steps of a COMMON machine with the same number of processors (and more memory). We further prove that this is optimal, if processor communication is restricted in a natural way.

Key Words. Parallel random access machines, Write-conflict resolution, Lower bounds.

1. Introduction. A model of parallel computation that has gained prominence in the theoretical literature is the CRCW PRAM (concurrent-read concurrent-write parallel random access machine). The power of this machine makes it a desirable model for the purposes of algorithm design (see, for example, [Ga], [SV], and [TV]); but this power also makes it difficult to prove lower bounds that do not follow directly from sequential considerations. In addition, several different models of CRCW PRAM have been defined, and the question of their relative power remains open. In what follows we hope to shed some light on this question, and to demonstrate interesting techniques for proving lower bounds on parallel machine models.

A CRCW PRAM consists of a set of n processors $P_1, P_2, \ldots, P_n$, together with a shared memory. Each processor is a random access machine with a local memory. The processors are synchronized; one step of computation consists of three phases. In the first phase each processor may perform local computation. In the second phase each processor may choose a cell of shared memory to write into; all processors write simultaneously. In the third phase each processor may read one shared memory cell. Any number of processors may simultaneously read from or write into the same shared memory cell. For the purposes of lower bounds, we allow an arbitrary amount of local computation in the first phase.

Models that allow simultaneous writes must specify a write-conflict resolution scheme. Two models that have appeared in the literature are the COMMON model and the PRIORITY model. In the COMMON model, write conflicts are

[1] Support for this research was provided by NSF Grants MCS-8402676 and MCS-8120790, DARPA Contract No. N00039-84-C-0089, an IBM Faculty Development Award, and an NSERC postgraduate scholarship.
[2] Department of Computer Science, University of Toronto, Ontario M5S 1A4, Canada.
[3] Hebrew University, Jerusalem, Israel.

Received December 5, 1986; Revised May 13, 1987. Communicated by Jeffrey Scott Vitter.

disallowed by imposing the condition that whenever several processors attempt to write into the same cell simultaneously, they must all be writing the same value [K]. In the PRIORITY model, distinct priorities are assigned to the processors. When several processors attempt to write (possibly different) values into the same cell simultaneously, the processor that succeeds is the one of highest priority [G]. Without loss of generality we can assume that priority is given to the processor of lowest index. When the number of shared memory cells m in a model is important, we place it after the model name in parentheses (e.g., PRIORITY (m)).

Kučera [K] showed that COMMON with n^2 processors can simulate PRIORITY with n processors in constant time. However, existing lower-bound techniques do not appear to be strong enough to separate these two models when each has the same number of processors and the same number of shared-memory cells. It is trivial to simulate one step of PRIORITY(m) by $O(\log n)$ steps of COMMON(m) by using a technique similar to binary search, as explained in Section 2. This method was proved optimal in [FMRW] when $m = O(n^{1-\varepsilon})$ for fixed $\varepsilon > 0$. In the same paper an $\Omega(\log \log \log n)$ separation was shown between PRIORITY(∞) and COMMON(∞), by looking at the problem of element distinctness. This was improved to $\Omega(\sqrt{\log n})$ [RSSW]. These proofs imply the existence of a rapidly growing function $m_0(n)$ such that the separation also holds between PRIORITY(m) and COMMON(m) for $m \geq m_0(n)$. No separation is known for other values of m, when the two models have the same number of memory cells. (Some separations for models with differing numbers of memory cells can be found in [FRW1].)

Here, we demonstrate a surprising improvement in simulation time by increasing the amount of shared memory available to the simulating machine. More precisely, we show how to simulate one step of PRIORITY(m) by $O(\log n/(\log \log n))$ steps of COMMON(nm). In particular, this demonstrates that the separation between PRIORITY(∞) and COMMON(∞) is $O(\log n/(\log \log n))$, rather than the previously conjectured $\Theta(\log n)$. Such problems as element distinctness and set equality thus have sublogarithmic algorithms on COMMON(∞), since they can be solved in constant time on PRIORITY(∞).

In simulating an algorithm designed for PRIORITY on COMMON, each processor in the COMMON model takes the role of one processor in the PRIORITY model. Simulation of the read and compute phases is trivial, since the two models do not differ in this respect. To simulate one write phase of PRIORITY requires the following problem to be solved:

LEFTMOST WRITERS

Before: Each processor P_i ($\leq i \leq n$) has a value x_i ($0 \leq x_i \leq m$) in its local memory. ($x_i = 0$ if P_i does not write at this phase; otherwise x_i represents the index of the cell that P_i wishes to write into.)

After: Each processor P_i with $x_i \neq 0$ will be in one of the states "leftmost" or "not-leftmost". P_i will be in state "leftmost" if and only if it is the processor of lowest index among those with the same value of x_i.

Once the problem is solved, a processor P_i will write if and only if it has the value "leftmost," thus resolving the write conflict in the manner that a PRIORITY machine would.

A natural way of solving this problem is to have each processor work only on resolving the conflict at the cell it wishes to write into. Each processor knows the subproblem it is working on, but does not know which other processors are working on the same problem. We call such a problem a "prisoner" problem. The analogy is with prisoners in cells; each prisoner knows what cell he is in, but does not know which other cells contain prisoners. A different area of shared memory can be dedicated to each subproblem, and all subproblems can be solved simultaneously and independently. To any one processor P, the processors working on different subproblems will appear "dead," as they will not participate in the same computation as P. This leads to the following problem definition:

LEFTMOST PRISONER

Before: Each processor has one of the values "live" or "dead" in its local memory. Processors with the value "dead" do not participate in the computation.

After: Each live processor P_i will be in one of the states "leftmost" or "not-leftmost." P_i will be in state "leftmost" if and only if it is the live processor of lowest index.

Then the simulation problem can be solved by solving m simultaneous leftmost prisoner problems, one for each cell of shared memory in the simulated machine. This is a logical division of labor, and requires no overhead for processor allocation.

In Section 2 of this paper we show how to solve the leftmost prisoner problem on the COMMON model in $O(\log n/(\log \log n))$ steps, using $O(n)$ cells of shared memory. This leads to a simulation of one step of PRIORITY(m) by $O(\log n/(\log \log n))$ steps of COMMON(nm). In Section 3 we prove that the leftmost prisoner problem requires $\Omega(\log n/(\log \log n))$ steps to solve on the COMMON model, regardless of the number of cells of shared memory. Our simulation is thus optimal among "prisoner-style" simulations, in which processors are forbidden to communicate with those attempting to write into different cells. We do not know any method of proving a lower bound for general simulations, but conjecture that the upper bound given here is tight. In [FRW2] evidence is presented that allowing communication between processors working on different subproblems does not help in the COMMON model; however, the same paper also shows that interproblem communication does help for a model whose power lies between COMMON and PRIORITY.

This work adds to the growing body of knowledge about techniques for lower bounds on these powerful models. Other related results may be found in [B], [FRW1], [LY], [MW], [R], and [VW].

2. An Upper Bound for Leftmost Prisoner. We initially assume that all memory cells are initialized to 0. Binary search can be used to solve the leftmost prisoner

problem in $O(\log n)$ steps using only one shared memory cell. At the first step, any live processor whose index is at most $\lfloor n/2 \rfloor$ writes the value 1 into the cell. If the value 1 appears, then the live processor of lowest index is among $P_1, \ldots, P_{\lfloor n/2 \rfloor}$. If not, then it is among $P_{(1+\lfloor n/2 \rfloor)}, \ldots, P_n$. In this fashion the range within which the live processor of lowest index lies can be cut by a factor of 2 at each step. This procedure is easily shown to be optimal if only one cell is used. If memory is not initialized, all processors can keep a count of the step number, and write that value instead of the value 1 in the above procedure.

Let us fix the number of steps used to solve the problem at t, and consider the largest number of processors for which we can specify a t-step algorithm. The previous procedure solved a problem for 2^t processors, using one cell of shared memory. We can increase the number of processors by using more cells of shared memory. Whenever we describe processor actions, it is understood that those actions are taken only if the processor is live.

THEOREM 1. *A leftmost prisoner problem for $(t+1)!$ processors can be solved in t steps on* COMMON(m_t), *where $m_t = \sum_{i=2}^{t+1} (t+1)!/i! \leq (e-2)(t+1)!$. Here, e represents the base of the natural logarithm.*

COROLLARY. *Leftmost prisoner on* COMMON($(e-2)n$) *can be solved in $O(\log n/(\log \log n))$ steps.*

PROOF. The proof of Theorem 1 is by induction on t. We can easily solve a 2-processor problem in one step, using one shared memory cell. In the write phase, P_1 writes the value 1 into the cell; in the read phase, P_2 reads the cell. P_1 is the live processor of lowest index if and only if it is alive; P_2 is the live processor of lowest index if and only if it is alive and it reads 0.

Now assume that we have algorithm A which solves a problem for $t!$ processors in $t-1$ steps, using m_{t-1} memory cells. Given $(t+1)!$ processors and m_t cells, divide the processors into $t+1$ equal groups, where the ith group consists of processors $P_{t!(i-1)+1}$ through $P_{t!i}$ inclusive. Since $m_t = (t+1)m_{t-1} + 1$, we may divide the memory into $t+1$ blocks of size m_{t-1}, with one extra cell remaining (say M_1). Each group of processors is assigned one block of memory cells.

Each group will use $t-1$ of its read-write phases to solve leftmost prisoner within the group, using algorithm A and their own block of memory. The remaining read and write phase will be used to interact with other groups, through the cell M_1. The key to this efficient simulation is in the scheduling of these interactions.

In the write phase of the ith step, every processor in group i writes the value 1 into M_1. (Recall that only live processors are participating.) This informs any processor in a group of higher index that it is not the live processor of lowest index. In the read phase of the ith step, every processor in group $i+1$ reads M_1, thereby ascertaining whether or not there is a live processor in a group of lower index. Each group is busy with M_1 for at most one read phase and the next write phase.

At the end of t steps, a processor in group 1 is the live processor of lowest index if and only if it is the live processor of lowest index within its group. A processor in group $i > 1$ is the live processor of lowest index if and only if it has read the value 0 from M_1 (indicating that no processors in lower groups were live) and it is the live processor of lowest index within its group. This proves Theorem 1. □

3. A Lower Bound for Leftmost Prisoner. In this section we will show that the upper bound given in the corollary to Theorem 1 is optimal to within a constant factor. We demonstrate a lower bound for leftmost prisoner that holds regardless of the number of cells of shared memory available. Suppose we are given an algorithm that solves leftmost prisoner for n processors in t steps. We will go through a series of simplifications that reduce the problem of bounding t as a function of n to a question involving a simple combinatorial structure. First, we consider a related problem.

DETECTION

Before: Each processor has one of the values "live" or "dead" in its local memory; processors with the value "dead" do not participate in the algorithm.

After: Every live processor will be in one of the states "alone" or "not-alone"; there exists a live processor in state "not-alone" if and only if there are at least two live processors.

Note that the detection problem does not define a function. An algorithm for detection will ensure that, for every input in which more than one processor is live, some live processor will be in the state "not-alone." The statement of the problem does not specify which of the live processors should be in that state, or what state any of the other live processors should be in.

Detection is a problem which captures the difficulty of prisoner problems on COMMON. The intuition behind this is that when a processor in the COMMON model writes into a cell, it has no way of finding out which other processors (if any) wrote into that cell at the same time, as they must have written the same value. To solve the detection problem, some processor must read a value that it knows it did not write.

It will be convenient, in what follows, to consider a step of computation to consist of read, compute, and write phases, in that order; this increases the length of an algorithm by at most one step. We can then convert the given leftmost prisoner algorithm to an algorithm that solves the detection problem on a COMMON model with read-compute-write ordering in $t+1$ steps; simply relabel state "not-leftmost" to be state "not-alone." If the leftmost prisoner algorithm is valid, then any processor that is in state "not-leftmost" at the end of that algorithm has detected a live processor of lower index.

We can assume without loss of generality that any detection algorithm is oblivious: that is, that the sequence of cells accessed by any one processor does not depend on any information that the processor did not have at the start of

the computation. This is true because as soon as a processor reads a value that it did not write at a previous step, it can enter the state "not-alone" and halt.

By at most doubling the length of the algorithm, we may also assume that the algorithm is singleminded; that is, each processor accesses exactly one cell at each step. Each step "P_k: read from cell i, write to cell j" is converted to "read from cell i, write [the same value back] into cell i, read from cell j [and ignore the value read], write into cell j." If a processor does not read or write at some step, then we modify it to access a cell that does not appear in the program of any other processor.

Thus the program of any processor can be viewed as a sequence of integers representing cells accessed at steps 1, 2, etc. Let us say that a set of such sequences has the difference property if for any two sequences α and β, there exist positions j and k, where $k > j$, such that $\alpha_j \neq \beta_j$ but either $\alpha_j = \beta_k$ or $\alpha_k = \beta_j$. In this case, we say that j and k are witnesses to the difference of α and β.

LEMMA 1. *The set of access sequences of processors in an oblivious singleminded algorithm for the detection problem has the difference property.*

PROOF. Suppose two processors P_i and P_j had access sequences that did not satisfy the difference property. Then, at each time step, P_i and P_j either access the same cell, or each accesses a cell that the other processor will not access at a subsequent time step. Consider the situation where all processors are dead except P_i and P_j. P_j, if alive, will not be able to tell if P_i is alive or dead, and thus cannot correctly choose a final state. P_i has the same problem. Thus the algorithm is faulty. □

We have shown that a leftmost prisoner algorithm for n processors in t steps leads to a set of n sequences of length $2t+2$ with the difference property. Since we wish to place an upper bound on n in terms of t, it suffices to bound the size of any such set of sequences. The bound that we will derive will hold for sets of sequences having the property that no sequence contains a repeated integer. The following technical lemma ensures that we can make this assumption.

LEMMA 2. *Given a set S of sequences of length t with the difference property, there is a set S' of sequences of length t such that no sequence in S' contains a repeated integer, and $|S| = |S'|$.*

PROOF. The proof of Lemma 2 is achieved by transforming S into S' with the required property. We will construct $S_1, S_2, \ldots, S_t$ such that $|S_i| = |S|$, each S_i has the difference property, and no sequence in S_i contains a repetition when truncated after position i. We can let $S_1 = S$. Given S_i, we construct S_{i+1} as follows: let R be the set of integers that occur for the second time in some sequence of S_i, at position $i+1$. For each $r \in R$, choose a new integer r' that does not appear in any sequence and, for each sequence α in S_i, replace the second and all subsequent occurrences of r in α by occurrences of r'. This defines S_{i+1}.

It is clear that S_{i+1} has no repetitions in any sequence truncated after position $i+1$. We must show that S_{i+1} has the difference property. Let α' and β' be two sequences in S_{i+1}, and let α and β be the corresponding sequences in S_i. We will see that witnesses exist to the difference of α' and β'. There must exist witnesses d and c ($d > c$) to the difference of α and β. Assume without loss of generality that $\alpha_c \neq \beta_c$ and $\alpha_c = \beta_d = r$. Since the transformation never makes unequal quantities equal, $\alpha'_c \neq \beta'_c$. Keep in mind that the only differences between α and α' (or between β and β') arise as a result of repetitions in those sequences. There are four cases:

(1) $\alpha'_c = \alpha_c$ and $\beta'_d = \beta_d$. Then, clearly, c and d are witnesses to the difference of α' and β'.

(2) $\alpha'_c \neq \alpha_c$ and $\beta'_d \neq \beta_d$. Since $\alpha_c = \beta_d = r$, these will both be changed by the transformation to r', and so $\alpha'_c = \beta'_d$. Thus c and d are still witnesses to the difference of α' and β'.

(3) $\alpha'_c \neq \alpha_c$ but $\beta'_d = \beta_d$. Then α_c must have been a repetition. Let a be the position of the first occurrence of the integer r in α. We can prove that positions d and a are witnesses to the difference of α' and β'. Since first occurrences are not affected, $\alpha'_a = r$. It cannot be the case that $\beta_a = r$, otherwise β_d would have been a repetition and would have changed in the transformation. Thus $\alpha'_a \neq \beta'_a$. Also, $\alpha'_a = r = \beta_d = \beta'_d$, and we conclude that $\alpha'_a = \beta'_d$, as required.

(4) $\alpha'_c = \alpha_c$ but $\beta'_d \neq \beta_d$. Then β_d was a repetition. Let a be the position of the first occurrence of the integer r in β. As before, $\beta'_a = \beta_a = r$. We know that a cannot equal c, because $\beta_a = r$ and $\alpha_c = r$ but $\alpha_c \neq \beta_c$. If $a > c$, then we know that $\alpha'_c \neq \beta'_c$ and $\alpha'_c = \beta'_a$, as required. So positions a and c are witnesses to the difference of α' and β'. If $c > a$, then α_a must be different from r, for otherwise α_c would be a repetition and would have been affected by the transformation. Since $\beta'_a = r$, we conclude $\alpha'_a \neq \beta'_a$. Also, $\beta'_a = r = \alpha'_c$, and so positions c and a are witnesses to the difference of β', α'.

Letting $S' = S_t$ finishes the proof. □

If no sequence contains a repeated integer, then the difference property has a simpler formulation: for every pair of sequences, there exists an integer appearing in both, but at different positions in each.

LEMMA 3. *If S is a set of length-t sequences with the difference property, then $|S| \leq t^t$.*

PROOF. We may assume, by Lemma 2, that no sequence in S contains a repeated integer. Let E be the set of integers that are entries in the sequences of S, that is, $E = \{\alpha_j \mid \alpha \in S, 1 \leq j \leq t\}$. Let $U = \{1, 2, \ldots, t\}^E$, that is, the set of tuples indexed by E with entries between 1 and t. Clearly, $|U| = t^{|E|}$.

One way of looking at a tuple $u \in U$ is as a specification of where integers might appear in a sequence; u_i is a possible location for the integer i. For $\beta \in S$, let $U(\beta)$ be the set of all tuples u for which this specification is consistent with

the sequence β. That is, $U(\beta) = \{u \in U \mid$ for all $e \in E$, either e does not occur in β, or $\beta_{u_i} = e\}$. $U(\beta)$ is well defined, since we have assumed that any integer e occurs at most once in β.

Note that in any tuple $u \in U(\beta)$, it is the case that $u_{\beta_j} = j$ for $j = 1, 2, \ldots, t$, but that the remaining $|E| - t$ entries in u can take on any value, since the corresponding element in E does not appear in β. Thus $|U(\beta)| = t^{|E|-t}$ for all $\beta \in S$.

Also note that, for two distinct sequences α and β in S, there exists some integer i appearing in both, but at different locations. Say $\alpha_a = \beta_b = i$ and $a \neq b$. Then for all $u \in U(\alpha)$, $u_i = a$, and for all $u \in U(\beta)$, $u_i = b$. Thus $U(\alpha) \cap U(\beta) = \phi$.

Then, since U contains the disjoint union $\bigcup_{\beta \in S} U(\beta)$,

$$t^{|E|} = |U| \geq \left| \bigcup_{\beta \in S} U(\beta) \right| = |S| t^{|E|-t}.$$

Hence $t^t \geq |S|$. □

Using Lemma 3 we can finally prove the following theorem, which is the main result of this section.

THEOREM 2. *Leftmost prisoner for n processors requires $\Omega(\log n/(\log \log n))$ steps.*

PROOF. Since a leftmost prisoner for n processors running in t steps leads to a set of n sequences of length $2t+2$ with the difference property, $n \leq (2t+2)^{2t+2}$. Thus $t = \Omega(\log n/(\log \log n))$, as required. In fact, our argument that a detection algorithm corresponds to a set of sequences with the difference property only requires that one or two processors be live. Thus leftmost prisoner has the same complexity even if the number of live processors is guaranteed to be at most two. □

We conclude by remarking that this proof can be extended [*R*] to show that leftmost prisoner has complexity $\Theta(\log n/(\log \log n))$ even if at least $n^{1-\varepsilon}$ processors are guaranteed to be alive, for any $\varepsilon > 0$.

References

[B] Beame, P., Limits on the Power of Concurrent-Write Parallel Machines, *Proc. 18th Annual ACM Symposium on Theory of Computing*, 1986, pp. 169-176.

[FMRW] Fich, F. E., Meyer auf der Heide, F., Ragde, P. L., and Wigderson, A., One, Two, Three... Infinity: Lower Bounds for Parallel Computation., *Proc. 17th Annual ACM Symposium on Theory of Computing*, 1985, pp. 48-58.

[FMW] Fich, F. E., Meyer auf der Heide, F., and Wigderson, A., Lower Bounds for Parallel Random Access Machines with Unbounded Shared Memory, in *Advances in Computing*

[FRW1] Fich, F. E., Ragde, P. L., and Wigderson, A., Relations Between Concurrent-Write Models of Parallel Computation, *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing*, 1984, pp. 179-189.

[FRW2] Fich, F. E., Ragde, P. L., and Wigderson, A., Relations Between Concurrent-Write Models of Parallel Computation, *SIAM J. Comput.* (to appear).

[Ga] Galil, Z., Optimal Parallel Algorithms for String Matching, *Proc. 16th Annual ACM Symposium on Theory of Computing*, 1984, pp. 240-248.

[G] Goldschlager, L., A Unified Approach to Models of Synchronous Parallel Machines, *J. Assoc. Comput. Mach.*, **29**(4), 1073-1086.

[K] Kučera, L., Parallel Computation and Conflicts in Memory Access, *Inform. Process. Lett.* **14**(2), 1982, 93-96.

[LY] Li, M., and Yesha, Y., New Lower Bounds for Parallel Computation, *Proc. 18th Annual ACM Symposium on Theory of Computing*, 1986, pp. 177-187.

[MW] Meyer auf der Heide, F., and Wigderson, A., The Complexity of Parallel Sorting. *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 532-540.

[R] Ragde, P. L., Lower Bounds for Parallel Computation, Ph.D. Thesis, University of California at Berkeley, 1986.

[RSSW] Ragde, P. L., Steiger, W., Szemerédi, E., and Wigderson, A., The Parallel Complexity of Element Distinctness is $\Omega(\sqrt{\log n})$, *SIAM J. Disc. Math.* (to appear).

[SV] Shiloach, Y., and Vishkin, U., Finding the Maximum, Merging and Sorting On Parallel Models of Computation, *J. Algorithms*, **2**, 1981, 88-102.

[TV] Tarjan, R. A., and Vishkin, Y., Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time, *Proc. 25th Annual Symposium on Foundations of Computer Science*, 1984, pp. 12-20.

[VW] Vishkin, U., and Wigderson, A., Trade-offs Between Depth and Width in Parallel Computation, *SIAM J. Comput.* **14**(2), 1985, 303-314.