RELATIONS BETWEEN CONCURRENT-WRITE MODELS OF PARALLEL COMPUTATION*

FAITH E. FICH, PRABHAKAR RAGDE, AND AVI WIGDERSON†

Abstract. Shared memory models of parallel computation (e.g., parallel RAMs) that allow simultaneous read/write access are very natural and already widely used for parallel algorithm design. The various models differ from each other in the mechanism by which they resolve write conflicts. To understand the effect of these communication primitives on the power of parallelism, we extensively study the relationship between four such models that appear in the literature, and prove nontrivial separations and simulation results among them.

Key words. parallel computation, lower bounds, parallel random access machines

AMS(MOS) subject classification. 68Q10

1. Introduction. Parallel computation has been the object of intensive study in recent years. Many models of synchronous parallel computation have been proposed. One important model is the CRCW PRAM (concurrent-read concurrent-write parallel random access machine, sometimes denoted WRAM). Not only have numerous algorithms been designed for the CRCW PRAM (examples include [Ga], [KMB], [SV], and [TV]), but it has also been shown to be closely related to unbounded fan-in circuits and alternating Turing machines ([CSV], [LY2]).

Specifically, a CRCW PRAM consists of a set of processors (i.e., random access machines) \( P_1, P_2, \ldots, P_n \) together with a shared memory. One step consists of three phases. In the read phase, every processor may read one shared memory cell. In the compute phase, every processor may perform computation. In the write phase, every processor may write into one shared memory cell. Any number of processors can simultaneously read from the same memory cell, and any number may attempt simultaneously write into the same memory cell.

An arbitrary amount of computation will be allowed in each compute phase. Although this is unrealistic, it enables us to concentrate on communication between processors. For all the problems we consider, communication rather than computation is the limiting factor. In fact, the algorithms presented in this paper actually perform very little computation at each step. Furthermore, the powerfulness of the model makes the lower bounds we present very strong.

A fundamental question concerning CRCW PRAMs is how to resolve write conflicts. One method is to assign priorities to processors and, if more than one processor attempts to write to the same memory cell, then the one with the highest priority will succeed. Without loss of generality (by reordering processors), we can assume that priorities are assigned in order of processor index, with highest priority given to the processor of lowest index [Go]. We call this the PRIORITY model.

Other mechanisms for conflict resolution appear in the literature. In the ARBITRARY model, if more than one processor attempts to write to the same memory cell, an arbitrary one will succeed [V]. Algorithms for the ARBITRARY model must work regardless of who wins the conflict, and so are writing a common value [Ku].

When more than one process is in the COLLISION model, a special conflict resolution primitive is given about which processors will be writing to the data structure; in this model, the processors are trying to write. This write-commit primitive is also used by Ethernet and other networks.

Write conflicts can also be avoided by using exclusive-write (CREW) PRAM, in which each given memory cell at each time step is read by at most one processor. Exclusive-read exclusive-write (EREW) PRAM is at least as powerful as the CREW PRAM and is restricted in this manner.

Any algorithm that runs on the PRIORITY model; if an algorithm runs on the ARBITRARY model, then it will certainly work if the ARBITARY model is at least as powerful as the PRIORITY model is at least as powerful as the ARBITARY model is at least as powerful as the COLLISION model is at least as powerful as the CREW PRAM is at least as powerful as the PRIORITY model is at least as powerful as the ARBITARY model is at least as powerful as the COLLISION model.

One step of the COLLISION model, using the same number of processors as in the ARBITARY model, requires \( \Omega(n^2) \) steps using a CREW PRAM, in which each processor is assigned a unique index. In the COLLISION model, the number of processors is at least as powerful as the CREW PRAM is at least as powerful as the PRIORITY model is at least as powerful as the ARBITARY model is at least as powerful as the COLLISION model.

Our aim is to understand the power of these models and to extend our results to other models of parallel computation.

Cook, Dwork, and Reischuk [CR] have shown that the CREW PRAM is at least as powerful as the CRCW PRAM. In this paper, we consider the problem of the number of processors required to solve a problem in parallel. We show that if the number of processors is \( n \) and the problem is solvable in \( K \) time steps using a CREW PRAM, then the problem can be solved in \( \Omega(n^k) \) steps using a CRCW PRAM. In general, the number of processors required to solve a problem in \( k \) time steps using a CRCW PRAM is at least as powerful as the number of processors required to solve the same problem in \( \Omega(n^k) \) steps using a CREW PRAM.

In this paper, we obtain separation results for the number of processors required to solve a problem in parallel. We show that if the problem is solvable in \( n \) time steps using a CREW PRAM, then the problem can be solved in \( \Omega(n^k) \) steps using a CRCW PRAM. In general, the number of processors required to solve a problem in \( k \) time steps using a CRCW PRAM is at least as powerful as the number of processors required to solve the same problem in \( \Omega(n^k) \) steps using a CREW PRAM.

Table 1 summarizes our results. The CREW model is denoted by its name for clarity (e.g., COMMON(1)). The weaker machine required to solve a problem is the weaker machine required to solve a problem with logarithms to the base 2. The

* Received by the editors October 7, 1986; accepted for publication (in revised form) April 22, 1987. This work was supported by National Science Foundation grants MCS-8120790, MCS-842676, and ECS-8110684, Defense Advanced Research Projects Agency contract N00014-82-C-0235, an IBM Faculty Development Award, the University of Washington Graduate School Research Fund, and a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship.

†Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4.

‡The Hebrew University, Jerusalem, Israel.
RELATIONS BETWEEN CONCURRENT-WRITE MODELS

work regardless of who wins the competition to write at each step. The COMMON model allows simultaneous writes to the same memory cell only if all processors doing so are writing a common value [Ku].

When more than one processor attempts to write to the same memory cell in the COLLISION model, a special collision symbol will appear in that cell. No information is given about which processors were involved in the collision nor what values they were trying to write. This write-conflict resolution scheme is a synchronous version of that used by Ethernet and multiple access channels [Gr].

Write conflicts can also be avoided by not allowing them; in the concurrent-read exclusive-write (CREW) PRAM, at most one processor can attempt to write to a given memory cell at each time step [FW]. An even more restrictive model is the exclusive-read exclusive-write (EREW) PRAM, in which both reads and writes are restricted in this manner.

Any algorithm that runs on the ARBITRARY model will run unchanged on the PRIORITY model; if an algorithm works regardless of who wins a competition to write, then it will certainly work if the processor of lowest index always wins. Thus the PRIORITY model is at least as powerful as the ARBITRARY model. Similarly, the ARBITRARY model is at least as powerful as the COMMON model, the COMMON and COLLISION models are at least as powerful as the CREW PRAM, and the CREW PRAM is at least as powerful as the EREW PRAM.

One step of the COLLISION model can be simulated by two steps on the ARBITRARY model, using the same number of processors and shared memory cells. First, each processor in the ARBITRARY model writes the corresponding processor in the COLLISION model wrote. However, it writes its index in addition to the value originally written. Then each processor reads the cell from which it has just written. If the index written there is not its own, a collision must have occurred during the previous write step. In this case, the processor writes the collision symbol to the memory cell.

Our aim is to understand the relative power of these models. Algorithms running on these models have appeared in the literature, and their expositions often include attempts to implement them on the most restrictive model possible. Such attempts are of little value without knowing which of the inclusions described above are strict.

Cook, Dwork, and Reischuk have shown that the CREW PRAM is strictly less powerful than the CRCW PRAM. In particular, their work [CDR] shows that the n-way OR function, which can be computed in one step on the COMMON model, requires \( \Omega(\log n) \) steps using a CREW PRAM. By considering the problem of searching in a sorted list of distinct elements, Snir [S] has shown that the EREW PRAM is strictly less powerful than the CREW PRAM.

In this paper, we obtain separation results for the four CRCW models as a function of the number of shared memory cells \( m \) (called the communication width [VV]) when the number of processors is held fixed at \( n \). This is an important restriction, since one step on the PRIORITY model is easily simulated by two steps on the COMMON or COLLISION model if the number of processors is squared and sufficient common memory is allowed [Ku]. When width is restricted, however, the four models are not equivalent. Restricting width has a meaning in a practical as well as theoretical sense; a bus or a satellite relay may be considered to be a CRCW PRAM with width 1.

Table 1 summarizes our results on simulations and separations. A particular model is denoted by its name followed by the number of shared memory cells in parentheses (e.g., COMMON(1)). The time bound given is the number of steps on the weaker machine required to simulate one step on the more powerful machine. All logarithms are to the base 2. The results in \( \S 2 \) and \( \S 3 \) are, for the most part, easy
adversary arguments; those in the remaining sections are harder and more revealing. Among the results we consider particularly significant is an information-theoretic lower bound for computation on COMMON(1) which is applicable in a more general setting (Theorem 6). The characterization of the global state of information proven in that theorem also allows us to prove a surprising constant time simulation of COMMON(1) by COLLISION(1) (Theorem 10).

**Table 1**

<table>
<thead>
<tr>
<th>Simulated Machines</th>
<th>Simulating Machines</th>
<th>Time Bounds</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIORITY(1)</td>
<td>ARBITRARY(m)</td>
<td>( \Theta \left( \frac{\log n}{\log(m+1)} \right) )</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>COLLISION(m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMON(m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIORITY(m) m = O(n/c)</td>
<td>ARBITRARY(cm)</td>
<td>O(( \frac{\log n}{\log(c+1)} ))</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>COLLISION(cm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMON(cm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARBITRARY(1)</td>
<td>COLLISION(m)</td>
<td>( \Theta \left( \frac{\log n}{\log(m+1)} \right) )</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>COMMON(m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIORITY(km) ARBITRARY(km) COLLISION(km)</td>
<td>COMMON(m)</td>
<td>( \Omega(k \log(n/km)) )</td>
<td>4</td>
</tr>
<tr>
<td>PRIORITY(km)</td>
<td>ARBITRARY(m)</td>
<td>( O \left( \frac{k \log n}{\log(k+1)} \right) )</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Omega \left( \frac{k \log(n/km)}{\log(k+1)} \right) )</td>
<td></td>
</tr>
<tr>
<td>COLLISION(1)</td>
<td>COMMON(m)</td>
<td>( \Theta \left( \frac{\log n}{\log(m+1)} \right) )</td>
<td>2.5</td>
</tr>
<tr>
<td>COMMON(1) on domain ( {0,1}^n )</td>
<td>COLLISION(1)</td>
<td>O(1)</td>
<td>5</td>
</tr>
</tbody>
</table>

New lower bound techniques are developed to obtain the results below. We consider this work as another step (following [S], [CDR], and [VW]) in forming a foundation of lower bound techniques for parallel computation. Also, as our lower bounds concern the communication between processors, we believe these techniques may be applied to distributed (asynchronous) computation as well (e.g., in the Ethernet model). Recent results have been obtained using more powerful techniques for models with infinite shared memory ([FMW], [MW]) and an infinite number of processors [B]. Li and Yesha ([LY1],[LY2]) have extended many of these results to models with the input in read-only memory (ROM) and have proved other results on this related model.

2. Simulating PRIORITY(1) by weaker models. Let us consider how to simulate one step of an algorithm for PRIORITY(m) on a machine with a weaker write

**m-colour MINIMIZATION**

Before: Each processor \( P_i \) for \( i \) writes to the processor of lowest index \( x_i = \min \{ x | x_i = 0 \} \).

After: Each processor \( P_i \) knows \( a_i \).

Thus \( a_i = 1 \) if and only if \( x_i = \min \{ x | x_i = 0 \} \).

In the simulation, \( x_i \) represents the processor \( P_i \) wishes to write; \( x_i = 0 \) if \( P_i \) writes to the processor of lowest index \( x_i \).

Clearly, the \( m \)-colour MINIMIZATION is equivalent to PRIORITY(m).

**THEOREM 1.** On COMMON solved in \( O(\frac{\log n}{\log(m+1)}) \) steps.

**Proof.** Without loss of generality, we may use the first \( \sqrt{n} \) cells of memory as the \( k \) running time.

Throughout the algorithm, \( x_i \) for the 1-colour MINIMIZATION in the \( (m+1) \)st group contains the processor of lowest index whose colour is 1 the winner.

The algorithm repeatedly picks out \( k \) shared memory cells are set to \( \text{set} \) into cell \( M_i \). The processors are in groups; each group contains a set of consecutively \( \frac{n}{m+1} \) processors. The \( j \)th group, \( 1 \leq j \leq m \), contains the \( m \) processors of the \( (m+1) \)st group; otherwise it contains the processor of lowest index containing a 1.

We note that a processor does not need to know its own colour, only whether its group contains a 1; we do not care which of the above two cases holds.

**LEFTMOST ONE IN MEMORY**

Before: Cells \( M_i \), for \( i = 1, \ldots, n \).

After: \( M_i \) contains 1 if and only if \( i = 1 \) and initially 1.
conflict resolution method, but with at least as much shared memory. Each processor in the PRIORITY(m) machine will be simulated by one processor in the simulating machine. Likewise, the contents of each shared memory cell in the PRIORITY(m) will appear in a specific shared memory cell. Simulation of the read phase is trivial. However, in the write phase on the simulating machine, processors must know if they are the processor of lowest index writing into the cell that they wish to write into. This requires some extra computation and leads to the definition of the following problem.

**m-colour MINIMIZATION.**

Before: Each processor \( P_i \), for \( i = 1, \ldots, n \), has a colour \( x_i \in \{0, \ldots, m\} \) known only to itself.

After: Each processor \( P_i \) knows the value \( a_i \), where

\[
a_i = \begin{cases} 
1 & \text{if for all } j < i, x_i \neq x_j \text{ and } x_i > 0, \\ 
0 & \text{otherwise.}
\end{cases}
\]

Thus \( a_i = 1 \) if and only if \( P_i \) is the processor of lowest index with colour \( x_i \) and \( x_i \neq 0 \).

In the simulation, \( x_i \) represents the memory cell into which the simulated processor \( P_i \) wishes to write: \( x_i = 0 \) if \( P_i \) does not wish to write. Once the problem is solved, \( P_i \) will write if and only if \( a_i = 1 \), thus resolving the write conflict in the fashion that PRIORITY machine would.

Clearly, the \( m \)-colour MINIMIZATION problem takes only one step to solve on PRIORITY(m).

**Theorem 1.** On COMMON(m), the 1-colour MINIMIZATION problem can be solved in \( O(\log m + 1) \) steps.

**Proof.** Without loss of generality, we may assume \( m \leq \sqrt{n} \). If \( m > \sqrt{n} \) we only use the first \( \sqrt{n} \) cells of memory. This is because with \( m = \sqrt{n} \) we already achieve \( O(1) \) running time.

Throughout the algorithm, memory cells will contain only 1's and 0's. Note that for the 1-colour MINIMIZATION problem, \( x_i \in \{0, 1\} \). We call the processor of lowest index whose colour is 1 the **winner**.

The algorithm repeatedly performs the following sequence of steps. First, all shared memory cells are set to 0 by having processor \( P_i \), for \( i = 1, \ldots, m \), write 0 into cell \( M_i \). The processors are divided into \( m + 1 \) nearly equal groups, where each group contains a set of consecutively numbered processors. The first \( n \mod (m+1) \) groups contain \( \lfloor \frac{n}{m+1} \rfloor \) processors and the rest contain \( \lfloor \frac{n}{m+1} \rfloor \). A processor \( P_i \) in the \( j \)th group, where \( 1 \leq j \leq m \), will write 1 into \( M_j \) if and only if \( x_i = 1 \). At this point, if all memory cells are unchanged (i.e., contain the value 0), the winner is in the \( (m+1) \)st group; otherwise it is in the group corresponding to the memory cell of lowest index containing a 1.

We note that a processor does not have to know which group contains the eventual winner, only whether its group wins. The following subroutines are used to decide which of the above two cases holds, in constant time.

**LEFTMOST ONE IN MEMORY.**

Before: Cells \( M_i \), for \( i = 1, \ldots, m \), each contain 1 or 0.

After: \( M_i \) contains 1 if and only if all \( M_j \) for \( j < i \) were initially 0, and \( M_i \) was initially 1.
Procedure: Processor $P_i$ forms the ordered pair $(j,k)$ from its name by setting $j = (i \mod m) + 1$ and $k = i - m(j-1)$. If $j < k \leq m$ and $M_j$ contains 1, $P_i$ writes 0 into $M_k$.

**EMPTY MEMORY (m-way OR).**

Before: Cells $M_i$, for $i = 1, \ldots, m$, each contain 1 or 0.
After: $M_i$ contains 0 if and only if all $M_i$ were initially 0.
Procedure: Processor $P_i$ will read $M_i$ and, if it contains 1, $P_i$ writes 1 into $M_1$.

After the **LEFTMOST ONE IN MEMORY** algorithm is applied, the processors in group $i$ look at $M_i$ to see if they are in the winning group or not (depending on whether $M_i$ contains 1 or 0, respectively). Note that this algorithm uses $m^2$ processors; the assumption $m \leq \sqrt{n}$ ensures that sufficient processors are available. Application of **EMPTY MEMORY** will then allow the $(m+1)$st group to decide if it is the winning group or not, by looking at $M_1$.

All processors except the ones in the winning group set $a_i = 0$ and stop; the ones in the winning group repeat the above procedure with $n$ replaced by the size of the group. This continues until the size of the winning group is equal to 1; at this point, the winner is determined.

Intuitively, the algorithm cuts the size of the winning group by a factor of $m+1$ each time. More precisely, if $g_t$ is the size of the set of processors that may still be the winner after the $t$th step, then

$$g_0 = n \quad \text{and} \quad g_t \leq \left\lfloor \frac{g_{t-1}}{m+1} \right\rfloor.$$

If $T \geq \log n / \log(m+1)$, then $g_T \leq \left\lfloor \frac{n}{(m+1)^T} \right\rfloor \leq 1$. Thus the algorithm takes at most $\left\lfloor \frac{\log n}{\log(m+1)} \right\rfloor$ iterations. Since each iteration takes a constant number of steps, we have the desired upper bound. □

**Corollary 1.1.** On **ARBITRARY**($m$) and **COLLISION**($m$), the 1-colour MINIMIZATION problem can be solved in $O\left(\frac{\log n}{\log(m+1)}\right)$ steps.

**Proof.** The algorithm described in the proof of Theorem 1 will run on **ARBITRARY**($m$). It will also run on **COLLISION**($m$) provided that, before performing **LEFTMOST ONE IN MEMORY**, each processor $P_i$ for $i = 1, \ldots, m$, reads memory cell $M_i$ and, if $M_i$ contains the collision symbol, writes 1 into $M_i$. □

It follows that **ARBITRARY**($m$), **COLLISION**($m$), and **COMMON**($m$) can simulate one step of **PRIORITY**($1$) in $O\left(\frac{\log n}{\log(m+1)}\right)$ steps. One cell of the simulating machine (say $M_1$) is designated as being equivalent to the single cell of **PRIORITY**($1$). Processor $P_i$ then sets $x_i = 1$ if it wishes to write at that step, and sets $x_i = 0$ otherwise. The algorithm for 1-colour MINIMIZATION is then followed, with the following change: the value 0 is replaced by the value presently in $M_1$, and the value 1 is replaced by some other value. At the end, the winner writes into $M_1$ the value that the corresponding processor would have written in the **PRIORITY**($1$) algorithm.

**Corollary 1.2.** **COMMON**($m$), **COLLISION**($m$), and **ARBITRARY**($m$) can simulate one step of **PRIORITY**($m$) in $O(\log n)$ steps.

**Proof.** The write conflict resolution problems for each of the $m$ memory cells of the **PRIORITY** machine can be treated as separate 1-colour MINIMIZATION problems that are solved simultaneously.

In the write conflict resolution problems of the simulated processor $P_i$ wishes to write 1 into $M_j$.

This subproblem is solved via the algorithm described above, using the single cell $M_j$.

In the algorithm, a processor $P_i$ is allowed to write 1 into $M_j$ if at most one of the subproblems has been solved, to where the processor it is simulating is supposed to have written 0 into $M_j$. We are solving subproblems that the processor of the simulated machine is supposed to be solving the subproblems. Although many processors are required to participate, each processor is allowed to write 1 into $M_j$ if at most one of the subproblems has been solved, to where the processor it is simulating is supposed to have written 0 into $M_j$.

**Corollary 1.3.** If $m = O(n^{1-\epsilon})$, one can simulate **ARBITRARY**($m$) in $O(n^{1-\epsilon})$ steps.

**Proof.** As in the proof of Corollary 1.1, for each of the $m$ memory cells we simulate the 1-colour MINIMIZATION problems simultaneously, with $c$ memory cells per processor.

Essentially, the algorithm described above allows $m$ processors to simulate 1-colour MINIMIZATION problems associated with $c$ memory cells per processor. We want to use the 1-colour MINIMIZATION problem associated with the lowest index cell containing the value 1 as a function of the number of memory cells and $c$.

Unlike the proof of Corollary 1.1, this proof does not allow us to write into memory cells that are not associated with 1-colour MINIMIZATION problems. Instead, we wish to write in some memory cells, but we are not allowed to write in some memory cells. The algorithm takes $O(n^{1-\epsilon})$ steps to write into all memory cells.

When $m = O(n^{1-\epsilon})$ for some $\epsilon > 0$, the algorithm takes $O(n^{1-\epsilon})$ steps to write into all memory cells.

The following lower bound holds for $m = O(n^{1-\epsilon})$.

**Theorem 2.** The 1-colour MINIMIZATION problem cannot be solved in $O(n^{1-\epsilon})$ steps to write into all memory cells.

Consider an algorithm solving the 1-colour MINIMIZATION problem. The proof of Theorem 2 is divided into two parts: a lower bound on the number of steps and a specification of the values of all memory cells at each step.

**Theorem 2.** The 1-colour MINIMIZATION problem cannot be solved in $O(n^{1-\epsilon})$ steps to write into all memory cells.

Consider an algorithm solving the 1-colour MINIMIZATION problem. The proof of Theorem 2 is divided into two parts: a lower bound on the number of steps and a specification of the values of all memory cells at each step.

The write action of a processor $P_i$ is a specification of the contents of memory cell $M_j$ at step $i+1$. We call this sequence a step of the processor $P_i$.

**Theorem 2.** The 1-colour MINIMIZATION problem cannot be solved in $O(n^{1-\epsilon})$ steps to write into all memory cells.

Consider an algorithm solving the 1-colour MINIMIZATION problem. The proof of Theorem 2 is divided into two parts: a lower bound on the number of steps and a specification of the values of all memory cells at each step.

The write action of a processor $P_i$ is a specification of the contents of memory cell $M_j$ at step $i+1$. We call this sequence a step of the processor $P_i$.

**Theorem 2.** The 1-colour MINIMIZATION problem cannot be solved in $O(n^{1-\epsilon})$ steps to write into all memory cells.

Consider an algorithm solving the 1-colour MINIMIZATION problem. The proof of Theorem 2 is divided into two parts: a lower bound on the number of steps and a specification of the values of all memory cells at each step.

The write action of a processor $P_i$ is a specification of the contents of memory cell $M_j$ at step $i+1$. We call this sequence a step of the processor $P_i$.

**Theorem 2.** The 1-colour MINIMIZATION problem cannot be solved in $O(n^{1-\epsilon})$ steps to write into all memory cells.

Consider an algorithm solving the 1-colour MINIMIZATION problem. The proof of Theorem 2 is divided into two parts: a lower bound on the number of steps and a specification of the values of all memory cells at each step.

The write action of a processor $P_i$ is a specification of the contents of memory cell $M_j$ at step $i+1$. We call this sequence a step of the processor $P_i$.
In the write conflict resolution problem for memory cell \( M_j \), let \( x_i = 1 \) if the simulated processor \( P_i \) wishes to write into memory cell \( M_j \); otherwise, let \( x_i = 0 \). This subproblem is solved via the algorithm described in Theorem 1 or Corollary 1.1, using the single cell \( M_j \).

In the algorithm, a processor only writes if its colour is 1. Each processor will have colour 1 for at most one of the subproblems, namely the subproblem corresponding to where the processor it is simulating wished to write. In fact, this is the only subproblem that the processor can win. Thus the processor can ignore the rest of the subproblems. Although many subproblems are being solved simultaneously, each processor is required to participate in the solution of at most one.

**Corollary 1.3.** If \( m = O(n/c) \), then \( \text{COMMON}(cm) \), \( \text{COLLISION}(cm) \), and \( \text{ARBITRARY}(cm) \) can simulate one step of \( \text{PRIORITy}(m) \) in \( O(\frac{\log n}{\log(c+1)}) \) steps.

**Proof.** As in the proof of Corollary 1.2, the write conflict resolution problems for each of the \( m \) memory cells of the \( \text{PRIORITy} \) machine can be treated as separate 1-colour \( \text{MINIMIZATION} \) problems. Each of these \( m \) problems will be solved simultaneously, with \( c \) memory cells devoted to each problem.

Essentially, the algorithm described in Theorem 1 or Corollary 1.1 is used. However, the allocation of processors is done somewhat differently for the computation of \( \text{LEFTMOST ONE IN MEMORY} \) and \( \text{EMPTY MEMORY} \). The processors are divided into groups of \( c \) consecutively numbered processors. Since \( m = O(n/c) \), each group is responsible for the solution of a constant number of subproblems.

Unfortunately, the \( \text{LEFTMOST ONE IN MEMORY} \) algorithm requires \( c^2 \) processors. Instead, each processor reads one of the \( c \) cells of memory and then they use the 1-colour \( \text{MINIMIZATION} \) algorithm again to decide which processor read the lowest indexed cell containing the value 1. This takes a constant number of steps, since the number of memory cells and the number of processors are the same.

Unlike the proof of Corollary 1.2, it is not sufficient to have only those processors that wish to write into memory cell \( M_j \) participate in the solution of the 1-colour \( \text{MINIMIZATION} \) problem associated with \( M_j \). There may be far fewer than \( c \) processors wanting to write into some cells. Even if there were a sufficient number of processors wanting to write into each shared memory cell, a processor would not necessarily know the identities of the other processors working with it. Therefore, it would not know what to do during the execution of the \( \text{LEFTMOST ONE IN MEMORY} \) algorithm.

When \( m = O(n^{1-\epsilon}) \) for some constant \( \epsilon > 0 \), choosing \( c = n' \) in Corollary 1.3 provides us with constant time simulations of \( \text{PRIORITy}(m) \), without increasing the number of processors.

The following lower bound shows that the algorithm in Theorem 1 is optimal, to within a constant factor. The proof uses a fairly simple adversary argument, which we present in detail, as it serves as a paradigm for subsequent proofs.

**Theorem 2.** The 1-colour \( \text{MINIMIZATION} \) problem requires at least \( \frac{\log(n+1)-1}{\log(m+1)} \) steps to solve on \( \text{ARBITRARY}(m) \).

Consider an algorithm solving this problem. An input to the algorithm is a specification of the values of all the colours \( z_i \). (Recall that \( z_i \in \{0, 1\} \).) We will use an adversary argument, constructing an input on which the algorithm requires many steps.

The write action of a particular processor \( P_i \) at step \( t \) depends only on \( z_i \) and the sequence of contents \( (H_0, H_1, \ldots, H_{t-1}) \) of the \( m \) shared memory cells. Here \( H_t \) is a specification of the contents of memory cells \( M_1, M_2, \ldots, M_m \) immediately before step \( t + 1 \). We call this sequence the history through time \( t \). Given a fixed history, we say \( P_i \) writes into \( M_j \) on value \( v \) if it attempts to write to memory cell \( M_j \) when \( z_i = v \).

Thus the algorithm takes at most a constant number of steps, we have

\[ \text{COLLISION}(m), \text{the 1-colour MIN-} \]

Thus Theorem 1 will run on ARBI-

\[ (m), \text{and COMMON}(m) \) can each of the \( m \) memory cells of the colour \( \text{MINIMIZATION} \) problems
At each step we will "fix" the values of certain input variables and maintain a set of allowable inputs. The set of allowable inputs will consist of those inputs in which each fixed variable has the value to which it was fixed. This will be done in a manner such that all allowable inputs will produce the same history up to that step.

The term position is used to denote the index of an input variable. A position $i$ is said to be fixed to a certain value if the corresponding variable $x_i$ is fixed to that value. If the variable $x_i$ is not fixed, the position $i$ is said to be free. We will maintain a set $S$ of fixed positions and a set $F$ of free positions such that $S \cup F = \{1, \ldots, n\}$. Associated with each position in $S$ will be the value to which that position is fixed. Furthermore, we will ensure that, for each free position in $F$, there will be no lower numbered positions fixed to 1.

Suppose there are at least two free positions $i$ and $j$ after some step $T$, and $i < j$. Consider the following two inputs: $I_{ij}$, in which $x_i$ and $x_j$ are the only variables in free positions that equal 1, and $I_j$, in which $x_j$ is the only variable in a free position that equals 1. Both these inputs put $P_j$ in the same state at the end of step $T$, since in both $P_j$ sees the same input value and the same history. But, for $I_{ij}$, $a_j = 1$, and for $I_j$, $a_j = 0$. Thus $P_j$ cannot know the value $a_j$ after the $T$th step. We can conclude that, when the algorithm terminates, there is at most one free position.

Initially, $S = \emptyset$, $H_0$ is the initial contents of memory, and the conditions stated above are satisfied. Now suppose that, after the $t$th step, we have fixed a set of positions $S$ so that all allowable inputs produce the same history $(H_0, H_1, \ldots, H_t)$ through step $t$. Furthermore, for any free position in $F$, there is no lower numbered position fixed to 1. Let $f_t$ denote the number of free positions remaining after step $t$.

We determine $H_{t+1}$ by fixing certain free positions, as follows.

1. If position $i$ is fixed before step $t+1$ and processor $P_i$ writes to some memory cell $M_j$ at step $t+1$, then the contents of $M_j$ in $H_{t+1}$ can be fixed by declaring $P_i$ to win the competition to write into $M_j$. Notice that we do not need to fix any additional free positions in this case.

2. For all cells $M_j$ into which some processor writes on 0 at step $t+1$ given history $(H_0, H_1, \ldots, H_t)$, choose one processor $P_i$ doing so, fix position $i_j$ to 0, and declare $P_i$ to win the competition to write into $M_j$ at step $t+1$ for all inputs consistent with $S$. This fixes the contents of $M_j$ in $H_{t+1}$ to a unique value.

3. Suppose there are $r$ memory cells not taken care of above. Processors only write on value 1 into these remaining cells. Let $f$ be the number of remaining free positions. Divide these free positions into $r+1$ nearly equal groups; the first group will contain the lowest $\left\lfloor \frac{f}{r+1} \right\rfloor$ positions, the second the next $\left\lfloor \frac{f}{r+1} \right\rfloor$ positions, and so on. The last $f \mod (r+1)$ groups will contain $\left\lceil \frac{f}{r+1} \right\rceil$ free positions. For each such cell $M_j$, let $P_j$ denote the processor of highest index writing on 1 into $M_j$ and associate this cell with the group containing $P_j$. Since there are $r$ cells and $r+1$ groups, at least one group will have no cells associated with it. Let $G$ be such a group and suppose that it comprises free positions $k$ through $l$. The idea is that, for each memory cell, by either forcing no processor to write to it or forcing the highest index processor wishing to write to it to do so, we can provide no information about group $G$.

i) Fix all free positions with index less than $k$ to 0. Consider any cell $M_j$ associated with a group consisting of free positions less than $k$. All processors that write into $M_j$ at step $t+1$ only do so on 1. However, they will have all had their colours $x_i$ set to 0. Hence, no processor will write into these cells at step $t+1$ for any allowable input and the contents of these cells in $H_{t+1}$ will remain as they were in $H_t$.

ii) Fix all free positions with index greater than $l$ to 1. Consider any group $G$ consisting of free positions greater than $l$. All processors that write into a cell in $G$ at step $t+1$ only do so on 0. However, they will have had their colours $x_i$ set to 0. Hence, no processor will write into these cells at step $t+1$ for any allowable input and the contents of these cells in $H_{t+1}$ will remain as they were in $H_t$.

Let $T$ be the total number of steps required to reach a history $H_T$ on which all processors have written. It follows that $T \geq \left\lceil \frac{\log(n+1)}{\log(m+1)} \right\rceil$.

It has been shown [Ha] that the problem of MINIMIZATION on ARBITRARY can be solved in $O(\log n \log \log n)$ steps by means of arbitrary choices to determine their behaviour.

3. Simulating ARBITRARY

This demonstrates a separation between the problems of minimizing nodes and arbiter nodes, which show a similar separation between the problems of ARBITRARY and COLLISION.

m-colour REPRESENTATION

Before: Each processor $P_i$, for $i \neq 0$, sets itself to 0.

After: Each processor $P_i$ knows whether it is the m-colour processor among those who set themselves to 0.

Notice that the m-colour REPRESENTATION problem can be solved in $O(\log n \log \log n)$ steps.

Theorem 3. On COMMONLY TATTIVE problem can be solved in $O(\log n \log \log n)$ steps.

Theorem 3, in fact, follows from the fact that the 1-colour MINIMIZATION problem reduces to the COMMONLY TATTIVE problem. The following theorem states this:

Theorem 4. On COMMONLY TATTIVE problem, each processor requires at least $\log n \log \log n$ steps to set itself.

Proof. The proof is similar to that of Theorem 2. As before, we maintain $H_t$ for each $t$, and $H_t$, a specification of the current history, at the end of step $t$. In this proof, we only fix positions to 1, and not to 0, as in the other consistent inputs are allowed.

Suppose there are at least two inputs $I_1$ and $I_2$, in which $x_j$ is the only variable that is
input variables and maintain a set $F$ of all these variables that will consist of those inputs in which $x_i$ has been fixed. This will be done in a manner similar to the one described by the history up to that step.

2) Fix all free positions of an input variable. A position $i$ of an input variable $x_i$ is fixed to $1$ if there are no input variables that are fixed to $0$ at the same position of the history. We will maintain a set $F$ of positions such that $S \cup F = \{1, \ldots, n\}$, where $S$ is the set of positions to which an input variable is fixed. For any position $i$ of input variable $x_i$ in $F$, there will be no lower numbered position $j$ where $x_j$ has been fixed to $1$ after some step $T$, and $i < j$.

3) Fix all free positions of an input variable. A position $i$ of an input variable $x_i$ is fixed to $0$ if there are no input variables that are fixed to $1$ at the same position of the history. But, for any position $i$ of input variable $x_i$ in $F$, there is no lower numbered position $j$ where $x_j$ has been fixed to $0$ after some step $T$, and $i > j$.

4) Fix all free positions of a memory cell. We will maintain a set $M$ of memory cells that are fixed to $1$ and $0$. For any memory cell $M_i$ at step $t + 1$ where $x_i$ is fixed to $1$ after some step $T$, we declare $P_i$ to win the competition to write into $M_i$ at step $t + 1$ for every allowable input. This fixes the contents of $M_i$ in $H_{t+1}$ to a unique value.

Intuitively, the number of free positions is cut down by at most a factor of $m + 1$ at each step. More precisely, if $f_t$ is the number of free positions at time $t$, then $f_{t+1} \geq \min_{0 \leq r \leq m} \left[ \frac{f_t - m + r}{r + 1} \right] = \left[ \frac{f_t}{m + 1} \right]$.

Let $T$ be the total number of steps taken by the algorithm. Since $f_0 = n$ and $f_T \leq 1$, it follows that $T \geq \left( \frac{n}{\log(n+1)} \right)$ steps.

It has been shown [Ra] that $\Omega \left( \frac{n}{\log(n+1)} \right)$ steps are required to solve 1-colour MINIMIZATION on ARBITRARY $(m)$ even if processors are allowed to make random choices to determine their behaviour at each step.

3. Simulating ARBITRARY $(1)$ by weaker models. The preceding section demonstrated a separation between PRIORITY $(1)$ and ARBITRARY $(m)$. We can show a similar separation between ARBITRARY $(1)$ and COMMON $(m)$ and between ARBITRARY $(1)$ and COLLISION $(m)$ by considering the following problem.

m-colour REPRESENTATIVE.

Before: Each processor $P_i$, for $i = 1, \ldots, n$ has a colour $x_i \in \{0, \ldots, m\}$ known only to itself.

After: Each processor $P_i$ knows the value $a_i \in \{0, 1\}$, where $a_i = 1$ for exactly one processor among those with each particular nonzero colour $c$.

Notice that the $m$-colour REPRESENTATIVE problem requires only one step on ARBITRARY $(m)$.

Theorem 3. On COMMON $(m)$ or COLLISION $(m)$, the 1-colour REPRESENTATIVE problem can be solved in $O \left( \frac{n}{\log(n+1)} \right)$ steps.

Theorem 3, in fact, follows easily from Corollary 1.1, as any solution to the 1-colour MINIMIZATION problem is also a solution to the 1-colour REPRESENTATIVE problem. The following theorem provides the separation between ARBITRARY $(1)$ and COMMON $(m)$.

Theorem 4. On COMMON $(m)$, the 1-colour REPRESENTATIVE problem requires at least $\Omega \left( \frac{n}{\log(n+1)} \right)$ steps to solve.

Proof. The proof is similar to that of Theorem 2; here we merely sketch the differences. As before, we maintain a set $S$ of fixed positions, a set $F$ of free positions, and $H_t$, a specification of the contents of the $m$ memory cells after step $t$. In this proof, we only fix positions to the value 0, but we do not allow the all-zero input. All other consistent inputs are allowed. The number of free positions is initially $n$.

Suppose there are at least two free positions $i$ and $j$ after some step $T$. Consider the following three inputs: $I_i$, in which $x_i$ is the only variable that has value 1, $I_j$, in which $x_j$ is the only variable that has value 1, and $I_{ij}$, in which $x_i$ and $x_j$ are the
only variables that have value 1. Both \(I_i\) and \(I_{ij}\) put \(P_i\) in the same state at the end of step \(T\), since in both \(P_i\) sees the same input value and the same history. Similarly, \(I_i\) and \(I_{ij}\) both put \(P_i\) in the same state at the end of step \(T\). For \(I_i\), \(a_i = 1\) and for \(I_{ij}\), \(a_j = 1\). If step \(T\) is the last step of the algorithm, then, for \(I_{ij}\), \(a_i = a_j = 1\). This contradicts the fact that the algorithm solves the 1-colour REPRESENTATIVE problem. We can conclude that, when the algorithm terminates, there is at most one free position.

Given the set of fixed positions \(S\) and the history \((H_0, H_1, \ldots, H_t)\), through time \(t\) we need to show how to fix some free positions in a way that defines \(H_{t+1}\). Those memory cells into which no processors write on either 0 or 1 retain their values. Memory cells that are written into by some processor on \(0\) (including any processors in a fixed position) are handled as in the proof of Theorem 2.

The difficulty occurs for those cells into which processors write only on 1. Let the set of indices of such cells be denoted \(C\). For \(j \in C\), let \(W_j\) be the set of processors in the remaining free positions that, at step \(t+1\), write on 1 into \(M_j\) and let \(W_0\) be those that do not write on 1 at this step. Note that none of these processors write into \(M_j\) on 0 for any \(j \in C\).

Now, the number of remaining free positions is at least \(|F| - (m - |C|)\). Thus, for some \(j \in C \cup \{0\}\), it follows that

\[
|W_j| \geq \frac{|F| - m + |C|}{|C| + 1} \geq \frac{|F|}{m + 1}.
\]

For each such \(j\), we fix the colours of all processors not in \(W_j\) to 0. This defines the contents of all memory cells \(M_k\) with \(k \in C \setminus \{j\}\). Specifically, their contents in \(H_{t+1}\) remain as they were in \(H_t\).

Finally, when \(j \in C\), consider the processors in \(W_j\). There is an allowable input in which all these processors have colour 1 and, thus, they must all write the same value on 1. Since we prohibited the all 0 input, all allowable inputs result in at least one processor in \(W_j\) receiving an input value that causes it to write at this step. Thus \(H_{t+1}\) is determined, at the cost of cutting down the number of free positions by at most a factor of \(m + 1\). As before, we can define a recurrence bounding the number of free positions at step \(t\) and conclude that Theorem 4 is true.

A lower bound of \(\Omega\left(\frac{\log n}{\log (m+1)}\right)\) steps holds for the 1-colour REPRESENTATIVE problem on COMMON\((m)\) even if processors are allowed to make random choices [Ra]. The following theorem provides an analogous separation between ARBITRARY\((1)\) and COLLISION\((m)\).

**Theorem 5.** On COLLISION\((m)\), the 1-colour REPRESENTATIVE problem requires at least \(\frac{\log (n+1) - \log 3}{\log (m+1)}\) steps to solve.

**Proof.** The proof of this theorem is very similar to that of Theorems 2 and 4. We must define the set of consistent inputs to be those with at least two 1's in free positions, in order to enable us to fix collisions in a history. It is not difficult to reason that, as long as there are at least three positions left free by the adversary at the end of the algorithm, there is an input on which the algorithm answers incorrectly. (The conclusion follows easily from the fact that any processor \(P_i\) cannot distinguish consistent inputs that agree on the private bit of \(P_i\).)

In the course of fixing the history in a cell after a particular step, there are three cases to consider: where no processors write into that cell, where exactly one processor writes, and where two or more processors write. The precise details of how to fix positions are thus slightly more complicated than those of Theorem 4, but no new techniques are involved. A recurrence for the number of free positions left after \(t\) steps can be defined, and the result obtained.

4. A lower bound for COMMON\((m)\):

PRAM with one cell \(M\) of shared memory of size \(R\) for some range set \(R\) if, at the beginning (denoted by \(x_i\)) in its local memory, \(f(x_1, x_2, \ldots, x_n)\) appears in \(M\).

Our definition can be thought of as allowing required to compute a private answer, and this can lead to a good lower bound for a useful public in a small number of steps using a unique processor with \(a_i = 1\) can appear in shared memory. The RRM defined earlier can be thought of as required to compute a private answer, and this can lead to a good lower bound for a useful public in a small number of steps using a unique processor with \(a_i = 1\) can appear in shared memory.

The following theorem gives an upper bound on the number of function values that can appear in shared memory.

**Theorem 6.** On COMMON\((m)\), \(f : \{0, 1\}^n \rightarrow R\) required to compute any function.

It is important to notice that an upper bound on the number of function values that can appear in shared memory.

**Theorem 6.** On COMMON\((m)\), \(f : \{0, 1\}^n \rightarrow R\) required to compute any function.

For example, no communication \((a_1, \ldots, a_n)\) given the input bits \((y_1, \ldots, y_n)\) by Theorem 6, a linear number of steps is required to compute any function id : \(\{0, 1\}^n \rightarrow \{0, 1\}^n\).

Furthermore, Theorem 6 does not hold for a subset of \(\{0, 1\}^n\) \(\{y\}^n\). Consider the function \(f(x) = y\) defined on the input, exactly one input variable is the index of that variable. This function can only be computed once in each step to publicly compute on COMMON\((1)\).

Although the theorem, as stated above, is powerful enough to use, we have:

**Lemma 6.1.** \(T\) steps of COMMON\((1)\).

**Proof.** Each processor in COMMON\((m)\) must wait for a shared memory that holds the simulation of each step of COMMON\((m)\). Assume a single cell in the simulating machine that each processor will write the state of the ordered pair \((v, i)\) into \(M_i\) of the simulating machine on that processor, update the contents of \(M_i\) in the simulating machine.

Each phase takes one step and is a function of COMMON\((1)\).

**Corollary 6.2.** On COMMON\((m)\), \(f : \{0, 1\}^n \rightarrow R\) required to compute any function that appears in shared memory.

When COMMON\((m)\) publicizes a function \(f\) that appears in shared memory, \(f\) is distributed among the \(m\) shared memory locations. Commonality is particularly weak when \(m = 1\), and it is just an \(n\)-tuple consisting of functions that are computed in one step under our model. A function \(f\) must appear in a single cell, \(\Omega(\log n)\).

By specifying a particular function for ARBITRARY and COMMON models.
4. A lower bound for COMMON(1) and its applications. We say that a PRAM with one cell $M$ of shared memory computes a surjective function $f : \{0,1\}^n \rightarrow R$ for some range set $R$ if, at the beginning of the computation, $P_i$ has the $i$th argument (denoted by $x_i$) in its local memory and, at the end of the computation, the value of $f(x_1, x_2, \ldots, x_n)$ appears in $M$.

Our definition can be thought of as public computation, since the answer must appear in shared memory. The REPRESENTATIVE and MINIMIZATION problems defined earlier can be thought of as private computation, since each processor is required to compute a private answer bit $a_i$. A good lower bound for public computation can lead to a good lower bound for private computation if $(a_1, a_2, \ldots, a_n)$ can be made public in a small number of steps. For example, in 1-colour MINIMIZATION, the unique processor with $a_1 = 1$ can take one more step and write $i$ into $M$.

The following theorem gives a lower bound on the number of steps required to publicly compute any function on COMMON(1). The lower bound depends only on the number of function values that are possible.

**Theorem 6.** On COMMON(1), any algorithm that publicly computes a surjective function $f : \{0,1\}^n \rightarrow R$ requires at least $\left\lceil \frac{\log_2|R|}{m} \right\rceil$ steps for some input.

It is important to notice that Theorem 6 does not apply to private computation. For example, no communication is required to privately compute the answer bits $(a_1, \ldots, a_n)$ given the input bits $(x_1, \ldots, x_n)$, where $a_i = x_i$ for $i = 1, \ldots, n$. However, by Theorem 6, a linear number of steps must be performed to publicly compute the identity function $id : \{0,1\}^n \rightarrow \{0,1\}^n$ on COMMON(1).

Furthermore, Theorem 6 does not apply when the domain is left, that is, a proper subset of $(0,1)^n$ is required. Consider the case where inputs are restricted so that in all valid inputs, exactly one input variable has value 1 and the function to be computed is the index of that variable. This function has a range of size $n$, but only requires a single step to publicly compute on COMMON(1).

Although the theorem, as stated, applies to the case of a single shared memory cell, it is powerful enough to use in a more general setting.

**Lemma 6.1.** $T$ steps of COMMON(m) can be simulated by $mT$ steps of COMMON(1).

**Proof.** Each processor in COMMON(1) keeps a picture in its local memory of what shared memory on COMMON(m) would contain at the corresponding step. The simulation of each step of COMMON(m) proceeds in phases: in the $i$th phase, the single cell in the simulating machine takes the role of $M_i$ in the simulated machine. All processors that would write the value $v$ into $M_i$ on COMMON(m) at this step write the ordered pair $(v, i)$ into $M_i$ on COMMON(1). Then all processors read $M_i$ and update the contents of $M_i$ in their picture of the shared memory of COMMON(m). Each phase takes one step and so one step of COMMON(m) is simulated by $m$ steps of COMMON(1). □

**Corollary 6.2.** On COMMON(m), any algorithm that publicly computes a surjective function $f : \{0,1\}^n \rightarrow R$ requires at least $\left\lceil \frac{\log_2|R|}{m} \right\rceil$ steps for some input.

When COMMON(m) publicly computes a function, we require the value of the function to appear in shared memory at the end of the computation, but it may be distributed among the $m$ shared memory cells. Note that this definition of computation is particularly weak when $m$ is large; consider the identity function, whose value is just an $n$-tuple consisting of the input bits. When $m = n$, this can be publicly computed in one step under our definition; but Beame [B] has shown that if the tuple must appear in a single cell, $\Omega(\log n)$ steps are required on PRIORITY($n^{O(1)}$).

By specifying a particular function to be computed, we can separate the ARBITRARY and COMMON models, with the separation varying as a function of the size.
of shared memory.

**Corollary 6.3.** Simulating one step of $\text{COLLISION}(km)$ on $\text{COMMON}(1)$ requires $\Omega(km \log(n/km))$ steps: on $\text{COMMON}(m)$, it requires $\Omega(k \log(n/km))$ steps.

Proof. Partition the input positions into $m$ groups of size $[n/km]$ or $[n/km] + 1$, and consider the function $f$ defined on domain $\{0, 1\}^n$ whose value is an $km$-tuple $(a_1, a_2, \ldots, a_{km})$ such that $a_i = j$ if $x_j$ is the only variable in group $i$ with value 1 and $a_i = 0$ if either no variable or more than one variable in group $i$ has value 1.

This function can be publicly computed in two steps on $\text{COLLISION}(km)$, assuming each shared memory cell is initialized to 0. In the first step, each processor $P_i$ in group $j$ writes 1 into $M_j$ on value 1. Each processor $P_i$, for $i = 1, \ldots, m$, then reads cell $M_j$ and, if it sees the collision symbol, writes 0 into $M_j$ at the second step.

The function $f$ has at least $\left(\frac{n}{km}\right)^{km}$ possible values. Applying Theorem 6 and Corollary 6.2 give lower bounds of $\Omega(km \log(n/km))$ and $\Omega(k \log(n/km))$ for $\text{COMMON}(1)$ and $\text{COMMON}(m)$, respectively. □

By letting $k = 1$, Corollary 6.3 proves a separation between $\text{COLLISION}(m)$ and $\text{COMMON}(m)$ when $m = o(n)$. In particular, when $m = O(n^{1-\epsilon})$ for some constant $\epsilon > 0$, $\Omega(\log n)$ steps are required.

We introduce some terminology to be used in the proof of Theorem 6. The *tree of possible computations* has nodes that intuitively correspond to the different states that the PRAM can be in during the course of the computation. Formally, with each node $v$ at depth $t$, we associate a history $(H_0, H_1, \ldots, H_t)$ and a set $I_v$ consisting of all inputs that generate this history through step $t$. An input is said to reach node $v$ if it is a member of $I_v$. The children of $v$ correspond to all possible extensions to the history at $v$; each child is labelled with a different extension $(H_0, H_1, \ldots, H_t, H_{t+1})$. The last entry in the history associated with a leaf of the tree will be the function value for all inputs that reach that leaf.

The statement of the theorem has an “information theory” flavour, and if we could show that the degree of fanout in the computation tree is bounded by a constant, the result would follow easily. Unfortunately, arbitrarily high fanout is possible, as the example with clef domain showed. The intuition behind this theorem is that a node of high fanout corresponds to a step at which many different values can be written, depending on the input. Since for a particular input, two processors may not attempt to write different values, this implies that some knowledge of “mutual exclusion” can be inferred from the history. In the example with clef domain, the knowledge that only one processor had an input variable with value 1 allowed $n$ different values to be written at step 1, depending on the input. We will show that this “mutual exclusion” takes time to set up and is not reusable.

With each node $v$ in the computation tree, we can associate a formula $f_v$ in conjunctive normal form, whose variables are the private input bits $x_i$. This formula will have the property that the set of inputs $I_v$ associated with this node is exactly the set of inputs that satisfy the formula $f_v$. The construction of these formulas will proceed by induction on the depth of a node. Formulas will have two types of clauses: trivial clauses will contain exactly one literal, and nontrivial clauses will contain more than one literal.

For the root $r$ of the computation tree, we define $f_r$ to be the empty formula. Now suppose we have a node $w$ with associated history $(H_0, H_1, \ldots, H_{t-1})$ and associated formula $f_w$. Suppose, furthermore, that $w$ has a child $v$ and that the history at $v$ is the history at $w$ extended by the value $H_t$. This means that for some inputs in $I_w$, the content of $M$ after the $t$th step is the value $H_t$. The action of any processor at step $t$ for an input in $I_w$ is completely determined by the history through step $t-1$ (the history associated with $w$, which is the same for all inputs in $I_w$) and by the processor’s private input bit $x_i$. The values would cause processors to write $H_t$.

For inputs that reach $v$, at least $x_i$ will write $H_t$. Thus inputs with history $H_t$ and clause consisting of the OR of the other literals. For example, if $P_1$ writes $H_t$ when clause $x_1$ is added clause would be $(x_1 \lor \bar{x}_2)$. We add. In two cases we do not change the history regardless of what its private bit is. Since there is only one memory cell of phase. Therefore, we can assume, after the memory cell $M$ only to change.

All possible bit values that written will result in additional tree $H'_t$ (different from $H_t$) if $x_3 = 1$. $H_t$ and not $H'_t$ was written implies that literal $x_3$ is removed; a clause containing that literal removed: a clause containing that literal removed: a clause containing that literal.

**Lemma 6.4.** If a node $w$ writes a clause, the formula at each child of $w$ will contain the clause.

Proof. If $q \leq 2$ this follows trivially from the way clauses are added. Thus we may assume $q = 2$. The tree of computation history at this node is a union of two trees, both containing no ones (if $H_t = H_{t-1}$), but the formula is written at the next step.

No processor may write more than one bit; we assume that we arbitrarily select one processor to write. If $q = 1$ and for $i = 1, \ldots, q-1$, value 1 is written (Note that $l_i$ is either $x_i$ or $\bar{x}_i$). The formula $f_w$ implies that $l_i$ is written. Otherwise, there would exist an input attempt to simultaneously write different values.

Now consider the formula $f_v$ written. This is created by first adding trivial clauses as a result of the mutual exclusion knowledge also results in some sum form $l_i$ which makes $l_{q-1}$ true. Since $l_i$ makes $l_i$ false, $C_j$ must be nontrivial clause containing the $l_i$ value of the $j$th bit. Thus $C_j$ must be a literal in $C_j$ that $\beta$ makes true.

Note that $C_j$ contains the literal $l_i$, but not $C_i$. Hence, for $1 \leq i < j \leq q$, $l_i$ is false.

Consider the creation of $f_v$. If $l_1, l_2, \ldots, l_{q-2}$ are false will result...
processor's private input bit $x_i$. Thus, it is possible to determine which private bit values would cause processors to write $H_1$.

For inputs that reach $v$, at least one processor must have a value that causes it to write $H_1$. Thus inputs with history $(H_0, H_1, \ldots, H_{t-1}, H_t)$ must satisfy $f_w$ and also a clause consisting of the OR of the literals corresponding to these possible bit values. For example, if $P_1$ writes $H_t$ when $x_1 = 1$, and $P_2$ does so when $x_2 = 0$, then the added clause would be $(x_1 \lor \overline{x_2})$. This is the only (possibly) nontrivial clause that we add. In two cases we do not add such a clause: when one processor $P_1$ writes regardless of what its private bit is, and, when no processor writes, i.e., $H_t = H_{t-1}$. Since there is only one memory cell, each processor reads its content during every read phase. Therefore, we can assume, without loss of generality, that processors write into the memory cell $M$ only to change its value.

All possible bit values that would have resulted in something other than $H_t$ being written would result in additional trivial clauses. For example, if $P_3$ would have written $H_1$ (different from $H_1$) if $x_3 = 1$, we need the trivial clause $(\overline{x_3})$, because the fact that $H_1$ and not $H_1'$ was written implies that $x_3 = 0$. We can also substitute these known values into other clauses. In our example, a clause containing the literal $x_3$ would have that literal removed; a clause containing the literal $\overline{x_3}$ would be entirely removed. This simplification is crucial to our proof, as it removes nontrivial clauses.

**Lemma 6.4.** If a node $w$ with $q$ children has a formula $f_w$ with $c$ nontrivial clauses, the formula at each child of $w$ has at most $c + 3 - q$ nontrivial clauses.

**Proof.** If $q \geq 2$ this follows from the construction, as at most one nontrivial clause is added. Thus we may assume $q > 2$. There are $q$ possible extensions of the computation history at this node. One of them could correspond to the case where no one writes $(H_t = H_{t-1})$, but there are at least $q - 1$ different values that could be written at the next step.

No processor may write more than one of these values, for otherwise that processor would always write, and $w$ would have exactly two children. For each value written, we arbitrarily select one processor that writes it; assume without loss of generality that for $i = 1, 2, \ldots, q - 1$, value $V_i$ is written by $P_i$ at this step if literal $i$ is true. (Note that $i$ is either $x_i$ or \overline{x_i}.)

The formula $f_w$ implies that at most one of the literals $l_1, l_2, \ldots, l_q$ is true. Otherwise, there would exist an input in $I_w$ for which two different processors would attempt to simultaneously write different values, a violation of the COMMON model.

Now consider the formula $f_v$ at the child $v$ of $w$ that corresponds to $V_{q-1}$ being written. This is created by first adjoining one nontrivial clause to $f_1$, and also some trivial clauses as a result of the knowledge that $l_1, l_2, \ldots, l_{q-2}$ are all false. This knowledge also results in some substitutions. Let $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ be an input in $I_v$, which makes $l_{q-1}$ true. Since $I_v$ is a subset of $I_w$, the input $\beta$ satisfies $f_w$ and makes each of the literals $l_1, l_2, \ldots, l_{q-2}$ false.

For $j = 1, \ldots, q - 2$, let $\beta^j$ be the input obtained from $\beta$ by complementing $\beta_j$ (i.e., $\beta^j$ makes both $l_j$ and $l_{q-1}$ true). The input $\beta^j$ cannot satisfy $f_w$, because it makes two literals in $(l_1, l_2, \ldots, l_{q-1})$ true. Let $C_j$ be some clause in $f_w$ that $\beta^j$ does not satisfy. Since there exists an input in $I_w$ which makes $l_j$ true, and another that makes $l_j$ false, $C_j$ must be nontrivial. The only difference between $\beta$ and $\beta^j$ is in the value of the $j$th bit. Thus $C_j$ must contain the literal $l_{\overline{j}}$. Furthermore, $l_{\overline{j}}$ is the only literal in $C_j$ that $\beta$ makes true.

Note that $C_j$ contains the literal $l_{\overline{j}}$ and $\beta^j$ makes $l_j$ false. Thus $\beta^j$ satisfies $C_j$, but not $C_j$. Hence, for $1 \leq i < j \leq q - 2$, the clauses $C_i$ and $C_j$ are distinct.

Consider the creation of $f_v$. The substitutions that follow from the knowledge that $l_1, l_2, \ldots, l_{q-2}$ are false will remove the nontrivial clauses $C_i$, for $i = 1, \ldots, q - 2$. The tree $T_v$ may correspond to the different states of the computation. Formally, with each processor $P_i$, for $i = 1, \ldots, m$, then writes 0 into $M_i$ at the second step. The possible values. Applying Theorem 6 to $\log(n/km)$ and $\Omega(k \log(n/km))$ for

by $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$.

In the proof of Theorem 6. The tree $T_v$ may correspond to the different states of the computation. Formally, with each processor $P_i$, for $i = 1, \ldots, m$, then writes 0 into $M_i$ at the second step. The possible values. Applying Theorem 6 to $\log(n/km)$ and $\Omega(k \log(n/km))$ for

by $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$.
Thus \( f_\nu \) can have at most \( c - (q - 2) + 1 \) nontrivial clauses, as required. A similar argument works for the other children of \( \nu \); in fact, the child that corresponds to the case when no process writes will have at most \( c + 2 - q \) nontrivial clauses. \( \square \)

The importance of Lemma 6.4 is that, although we cannot bound the degree of a node in the computation tree, high degree requires accumulating and then destroying nontrivial clauses, and only one nontrivial clause is accumulated per level. We use this idea to prove the following lemma.

**Lemma 6.5.** The maximum number of leaves in a computation tree of height \( h \) is \( 3^h \).

**Proof.** Let \( L(s, h) \) be the maximum number of leaves in a subtree of height \( h \) whose root formula has \( s \) nontrivial clauses. By Lemma 6.4, we have

\[
L(s, 1) \leq s + 3 \quad \text{and} \quad L(s, h) \leq \max_{2 \leq q \leq s + 3} \{ q \cdot L(s + 3 - q, h - 1) \}.
\]

This can be shown by induction on \( h \) to satisfy the inequality \( L(s, h) \leq (3 + s/h)^h \).

The base case is obvious; suppose the statement is true for \( h < k \). Then

\[
L(s, k) \leq \max_{2 \leq q \leq s + 3} \{ q \cdot L(s + 3 - q, k - 1) \}
\]

\[\leq \max_{2 \leq q \leq s + 3} \left\{ q \left( 3 + \frac{s + 3 - q}{k - 1} \right)^{k-1} \right\}.\]

The quantity inside the curly brackets, considered as a function over real \( q \), can be shown by elementary calculus to reach its maximum at \( q = 3 + s/k \). This yields \( L(s, k) \leq (3 + s/k)^k \), as required. \( \square \)

Theorem 6 then follows from the fact that each leaf of the computation tree can be labelled with at most one function value. All function values must appear, so the tree has at least \( |R| \) leaves. By Lemma 6.5, the computation tree must have height at least \( \lceil \log_3 |R| \rceil \).

We can extend this result and obtain a theorem similar to Theorem 6 for probabilistic algorithms. In the probabilistic COMMON model, each processor is allowed to make random choices to determine its behaviour at each step. We insist that no sequence of choices results in two processors attempting to write different values into the same cell at the same time. Theorem 7 gives a bound on the expected number of steps to compute a function in terms of the size of its range.

**Theorem 7.** In the probabilistic COMMON(1) model, any algorithm that publicly computes a surjective function \( f : \{0, 1\}^n \rightarrow R \) has an expected running time of at least \( \lceil \log_3 |R| \rceil \) steps on some input.

As in Corollary 6.3, we obtain a logarithmic separation between the probabilistic COMMON(m) model and the deterministic ARBITRARY(m) model, for \( m = O(n^{1-\epsilon}) \) where \( \epsilon \) is a positive constant. In this case, randomization does not help the COMMON model to simulate the more powerful model.

Theorem 7 is proved using the following two lemmas.

**Lemma 7.1.** The sum of the root-leaf distances to any set \( S \) of leaves in a tree of possible computations is at least \( |S| \cdot \lceil \log_3 |S| \rceil \).

**Proof.** Let us define a tree skeleton to be a tree whose nodes can be labelled with nonnegative integers, such that the root is labelled with zero, and any node labelled with \( s \) that has \( q \) children has each child labelled no higher than \( s + 3 - q \). Lemma 6.5 is actually a statement about tree skeletons; any computation tree leads in a natural way to a tree skeleton, where the label of a node is just the number of nontrivial clauses in its formula. Let \( S \) be a set of root-leaf distances. We can prune \( S \) in \( S \) by removing those leaves whose root-leaf distances to leaves in \( S \) are less than the largest of the root-leaf distances to leaves in \( S \). This still leaves a tree skeleton.

We can then transform the tree skeleton of the root-leaf distances to leaves in \( S \) with \( v \) at depth \( t \) and \( v_2 - t \) with the same number as \( v_1 \) and \( v_2 \).

Continuing in this fashion, we will prove that the depth of each distance to leaves in \( S \) is less than \( \lfloor \log_3 |S| \rfloor \).

**Lemma 7.2.** Let \( T_1 \) be the time of an algorithm solving problem \( P \), maximized over all inputs, running time for a given input data algorithms to solve \( P \). Then \( T_1 \geq \lceil \log_3 |R| \rceil \).

Lemma 7.2 was stated by Yannakakis for problems that can be proved in a few lines. We can extend this result to the distribution of deterministic algorithms in \( I \) and \( J \) of our set of inputs. Let \( r(A_t) \).

Suppose our given probabilistic algorithm \( \mathcal{A} \) in the above, with probability \( p_t \), and that on input \( \sum \mathbb{E} T_1 \).

Then

\[ T_1 = \sum \mathbb{E} T_1 \]

\[ \geq \sum \mathbb{E} T_1 \]

\[ \geq \sum \mathbb{E} T_1 \]

\[ = \sum \mathbb{E} T_1 \]

We wish to bound \( T_1 \) from below. To do this, we must specify the running time for any algorithm \( \mathcal{A} \) on \( f_t \), but not on a particular problem \( P \) that results in that value. This step is to give probability \( 1/|R| \) to each of these.

To bound \( T_2 \) from below for the distributions associated with some deterministic algorithm \( \mathcal{A} \) and set of \( |R| \) leaves. Then the expected value of the average depth of these leaves proves Theorem 7.

By a more careful analysis of Theorems 6 and 7 can be reduced the complexity of the family of functions \( \{ f_t \} \) such that

\[ \log_3 |R_t| + O(1) \]
clausal clauses, as required. A similar argument shows that the child that corresponds to the root of the tree is the only nontrivial clause. \( \Box \)

Although we cannot bound the degree of a subtree only by the number of nontrivial clauses accumulating and then destroying them, we can bound it by the number of nontrivial clauses accumulating per level. We use the following lemma:

**Lemma 6.4.** Let \( g(k) \) be the cost of leaves in a subtree of height \( k \). Then \( g(k) \) is at most \( 3 \cdot 3^{k-1} \).

**Lemma 6.5.** Let \( L(s, h) \) be the number of leaves in a subtree of height \( h \). Then \( L(s, h) \) is at most \( (3 + s) h^k \).

Proof. Let \( L(s, h) \) be the number of leaves in a subtree of height \( h \). Then we prove that \( L(s, h) \) is at most \( (3 + s) h^k \).

We use the following lemma:

**Lemma 6.6.** Let \( T \) be a computation tree of height \( h \). Then \( T \) has at most \( (3 + s) h^k \) leaves.

Proof. Let \( T \) be a computation tree of height \( h \). Then \( T \) has at most \( (3 + s) h^k \) leaves.

We can then transform the tree skeleton in a way that will never increase the sum of the root-leaf distances to leaves in \( S \). Suppose we can find two leaves \( v_1 \) and \( v_2 \), where \( v_1 \) is at depth \( t_1 \) and \( t_2 - t_1 \geq 2 \). We add two children \( v'_1, v'_2 \) to \( v_1 \), label them with the same number as \( v_1 \), and delete \( v_2 \). We remove \( v_1, v_2 \) from \( S \) and add \( v'_1, v'_2 \).

Continuing in this fashion, we can obtain a tree skeleton and a set \( S' \) of leaves, where \( |S'| = |S| \), and the depths of all leaves in \( S' \) differ by no more than 1. Lemma 6.5 implies that the depth of each leaf is at least \( \log_3 |S'| \). Since the sum of root-leaf distances to leaves in \( S' \) is less than or equal to \( Q \), the result follows. \( \Box \)

**Lemma 7.2.** Let \( T_1 \) be the expected running time for a given probabilistic algorithm solving problem \( P \), maximized over all possible inputs. Let \( T_2 \) be the average running time for a given input distribution, minimized over all possible deterministic algorithms to solve \( P \). Then \( T_1 \leq T_2 \).

**Lemma 7.2** was stated by Yao [Y] in a stronger form; the weak form here can be proved in a few lines. We can consider a probabilistic algorithm as a probabilistic distribution of deterministic algorithms. Let \( A \) be the set of deterministic algorithms, and \( I \) our set of inputs. Let \( t_i[A, I] \) be the running time of algorithm \( A \) on input \( I \).

Suppose our given probabilistic algorithm chooses to run deterministic algorithm \( A_i \) with probability \( p_i \), and that our given input distribution gives probability \( q_j \) to \( I_j \). Then

\[
T_1 = \max_{I_j \in I} \left\{ \sum_{A_i \in A} p_i t_i[A_i, I_j] \right\}
\]
\[
= \sum_{I_j \in I} q_j \sum_{A_i \in A} p_i t_i[A_i, I_j]
\]
\[
= \sum_{A_i \in A} \sum_{I_j \in I} q_j t_i[A_i, I_j]
\]
\[
\geq \min_{A_i \in A} \left\{ \sum_{I_j \in I} q_j t_i[A_i, I_j] \right\}
\]
\[
= T_2. \quad \Box
\]

We wish to bound \( T_1 \) from below. By Lemma 7.2, it suffices to bound \( T_2 \) from below. To do this, we must specify an input distribution that results in a large average running time for any algorithm to compute \( f \). This input distribution must depend on \( f \), but not on a particular program. For each possible value of \( f \), choose one input that results in that value. This selects a set of \( |R| \) inputs; our chosen distribution will give probability \( 1/|R| \) to each of these.

To bound \( T_2 \) from below for this distribution, consider the tree of possible computations associated with some deterministic algorithm. Our set of inputs reaches some set of \( |R| \) leaves. Then the expected running time on the given input distribution is the average depth of these leaves which by, Lemma 7.1, is at least \( \log_3 |R| \). This proves Theorem 7.

By a more careful analysis, the base of the logarithm in the lower bounds of Theorems 6 and 7 can be reduced to \( 1 + \sqrt{2} \). It is possible to define a somewhat artificial family of functions \( \{f_i\} \) such that \( f_i \) has range \( R_i \) and can be publicly computed in \( \log_3 |R_i| + O(1) \) steps on COMMON(1), where \( \beta = \frac{1+\sqrt{13}}{2} \) [Ra].
5. The relationship between COMMON and COLLISION. It is not immediately obvious whether COMMON is at least as powerful as COLLISION, whether COLLISION is at least as powerful as COMMON, or whether the power of these two models are incomparable.

To begin with, consider the following problem.

EXACTLY ONE.

Before: Each processor $P_i$, for $i = 1, \ldots, n$, has a bit $x_i$ known only to itself.

After: $M_1$ contains the value 1 if and only if exactly one bit $x_i$ is initially one.

If $M_1$ is initialized to 0, the EXACTLY ONE problem can be solved on COLLISION(1) by having each processor $P_i$ write the value 1 into $M_1$ when $x_i = 1$. However, in a model without the ability to detect collisions, it can take significantly longer.

THEOREM 8. The EXACTLY ONE problem requires at least $\frac{\log n}{\log(m+1)}$ steps to solve on COMMON($m$).

The proof of this result is essentially identical to the proof of Theorem 4. It then follows that $\Omega \left( \frac{\log n}{\log(m+1)} \right)$ steps on COMMON($m$) are needed, in the worst case, to simulate one step of COLLISION(1). The lower bound comes from Theorem 8, and the upper bound follows from the simulation of ARBITRARY by COMMON given in Theorem 1 and the fact that ARBITRARY can simulate COLLISION in constant time (as observed in §1).

Conversely, consider the following problem which can be solved in a constant number of steps on COMMON(1).

k-GROUP IDENTIFICATION.

Before: The processors are divided into $k$ groups of size $s$ or $s+1$, where $s = \lfloor n/k \rfloor$.

Each processor knows the indices of all other processors in its group. Each processor $P_i$, for $i = 1, \ldots, n$, has a private bit $x_i \in \{0, 1\}$ known only to itself. Furthermore, for $1 \leq i, j \leq n$, if $x_i = x_j = 1$, then $P_i$ and $P_j$ are in the same group.

After: $M_1$ contains the value $a \neq 0$ if and only if $x_i = 1$ for some processor $P_i$ in group $a$. $M_1$ contains the value 0 if and only if all input variables are 0.

THEOREM 9. On COLLISION($m$), the $\lfloor \sqrt{n} \rfloor$-GROUP IDENTIFICATION problem requires $\Omega \left( \frac{\log n}{\log(m+1)} \right)$ steps to solve.

Proof. The proof is similar to that of Theorems 2, 4, and 5. In addition to maintaining a set $S$ of fixed positions, a set $F$ of free positions, and the history $(H_1, H_2, \ldots, H_t)$ through time $t$, we maintain a set $A \subseteq \{1, \ldots, \lfloor \sqrt{n} \rfloor \}$ of groups whose positions have not yet been completely fixed. Positions are fixed only to 0. Allowable inputs are consistent with the fixed positions and also with the restriction mentioned in the statement of the problem.

At each step of the construction, we not only fix individual positions within groups to 0, but fix whole groups to 0 as well. It is possible to do this in such a way that if $|A| = s$ before step $t$, then it has size $\frac{s}{m+1}$ after step $t$. Furthermore, if $b$ is a lower bound, for each $a \in A$, on the number of free positions in group $a$, then $\frac{b-1}{m+1}$ is a lower bound afterwards. This is essentially done by associating with each group the cell into which the most processors in that group write on 1, and then finding the cell that is associated with the most groups. All groups not associated with that cell are fixed to 0 and removed from $A$; all processors on 1 into different cells have their positions fixed to 0. If groups remain in two different cells, then we just ignore them.

The above result is somewhat dependent on the fact that the $k$-GROUP IDENTIFICATION problem has a lower bound on the number of processors, so the following result is not quite as strong as COMMON(1) for computing a similar problem.

THEOREM 10. The public common problem requires at most twice as many steps on COMMON($m$) as on $f : \{0, 1\}^n \rightarrow R$ on COMMON(1).

Proof. We show how to compute $f$ on $H_{2t}$ through step $2t$, with the problem solved for step $t$ produced by the COMMON($m$) algorithm. Consider the set of processors $I_v$ that have values of $v$. We simulate $f$ on COMMON(1), the problem in a step by step fashion, using two COMMON(1) algorithms.

For any input, the processor $H_{2t}$ of index $v$ writes its index $i$, provided processor $P_i$ is in group $I_v$ that have values of $v$. If processor $P_i$ is not in group $I_v$, processor $P_i$ writes nothing. If the value $v'$ that processor $P_i$ writes is the only processor $P_i$ that wrote the value it would have written in COMMON(1), then processor $P_i$ writes the value $v'$ to a shared memory cell in the COMMON($m$) algorithm.

The COMMON(1) algorithm for $f$ proceeds as follows. At any even depth node $v$, if processor $P_i$ writes its index $i$, provided processor $P_i$ is in group $I_v$ that have values of $v$. If processor $P_i$ is not in group $I_v$, processor $P_i$ writes nothing. If the value $v'$ that processor $P_i$ writes is the only processor $P_i$ that wrote the value it would have written in COMMON(1), then processor $P_i$ writes the value $v'$ to a shared memory cell in the COMMON($m$) algorithm.

As in the proof of Theorem 9, COMMON(1) tree can be described in the form. For each even depth node $I_v$, associate with the corresponding in $I_v$. This is because $I_v \subseteq I_v'$. To convert the COMMON(1) algorithm above into a COMMON($m$) algorithm, we use the following claim.

CLAIM. For each (nontrivial) tree, at least two literals in that clause (and their inputs) or there is a particular literal.

Proof. The proof of this claim is similar to the one in COMMON(1).

If $v$ is the root of the COMMON($m$) algorithm and $v'$ is its corresponding node. Its associated with $v'$, then in this case, the claim is satisfied.

Now suppose $v$ has depth 2.
and COLLISION. It is not immediately obvious whether or whether the power of these two problems can be solved on COLLISION 1 into M1 when x_i = 1. However, it can take significantly longer.

The proof of Theorem 4. It then follows from the proof of Theorem 5 and the fact that the k-GROUP IDENTIFICATION problem can be solved on COMMON 1 into M1 when x_i = 1. However, it can take significantly longer.

The above result is somewhat unsatisfying. In an essential way, the proof depends on the fact that the k-GROUP IDENTIFICATION problem has a cleft domain. Recall, this means that the input is a proper subset of \( \{0,1\}^n \). Such an unrealistic assumption might never arise in the computation of a function defined on the domain \( \{0,1\}^n \). Indeed, the following theorem shows that COLLISION(1) is at least as powerful as COMMON(1) for computing functions on the domain \( \{0,1\}^n \). This simulation result uses the same structure that was used in the lower bound result of Theorem 6.

**Theorem 10.** The public computation of any function \( f : \{0,1\}^n \rightarrow R \) requires at most twice as many steps on COLLISION(1) as it does on COMMON(1).

**Proof.** We show how to convert any algorithm that publicly computes a function \( f : \{0,1\}^n \rightarrow R \) on COMMON(1) into an algorithm, using at most twice as many steps, that works on COLLISION(1). This new algorithm simulates the original algorithm in a step by step fashion, using two steps to simulate each step of the COMMON(1) algorithm.

For any input, the COLLISION(1) algorithm will produce a history \( (h_0, h_1, \ldots, h_2t) \) through step \( 2t \), with the property that \( (h_0, h_2, \ldots, h_2t) \) is the history through step \( t \) produced by the COMMON(1) algorithm on that input. Thus, the set of inputs \( I_v \) that reach any even depth node \( v \) in the tree of possible computations for the COLLISION(1) algorithm is a subset of \( I_v' \) for some node \( v' \) occurring at the half the depth in the tree of possible computations for the COMMON(1) algorithm. The node \( v' \) will be called the COMMON(1) node corresponding to \( v \).

The COMMON(1) algorithm is obtained from the COMMON(1) algorithm as follows. At any even depth node in the COMMON(1) algorithm, each processor \( P_i \) writes its index \( i \), provided processor \( P_i \) writes on the input \( x_i \) at the corresponding COMMON(1) node. Without loss of generality, we will assume that all the values written in every possible execution of the COMMON(1) algorithm are distinct from the processor indices \( 1, \ldots, n \). If no write takes place, then, at the next step, the processors do nothing. If the value \( i \) appears in the shared memory cell, indicating that processor \( P_i \) was the only processor attempting to write, then processor \( P_i \) writes the value it would have written in the COMMON(1) algorithm. Finally, if a collision occurred, then processor \( P_i \) writes the value that would have been written into the shared memory cell in the COMMON(1) algorithm. It remains to show that there is enough information for processor \( P_i \) to determine this value.

As in the proof of Theorem 6, the set of inputs \( I_v' \) that reach a node \( v' \) in the COMMON(1) tree can be described by a Boolean formula \( f_v' \) in conjunctive normal form. For each even depth node \( v \) in the COMMON(1) tree, the Boolean formula \( f_v' \) associated with the corresponding COMMON(1) node \( v' \) is satisfied by each input in \( I_v \). This is because \( I_v \subseteq I_v' \). There is additional information about the input that the COLLISION(1) algorithm also accumulates during the course of the simulation.

**Claim.** For each (nontrivial) clause in \( f_v' \), either every input in \( I_v \) satisfies at least two literals in that clause (although not necessarily the same literals for different inputs) or there is a particular literal in that clause which is true for all inputs in \( I_v \).

**Proof.** The proof of this claim proceeds by induction on the depth of the node \( v \). If \( v \) is the root of the COLLISION(1) tree, then the root of the COMMON(1) tree is its corresponding node. Its associated formula, the empty formula, has no clauses. In this case, the claim is satisfied.

Now suppose \( v \) has depth \( 2t \), where \( t \geq 1 \), and assume that the claim is true for
the grandparent \( w \) of \( v \). Let \( w' \) and \( v' \) be the COMMON(1) nodes corresponding to \( w \) and \( v \), respectively.

Consider the history \((H_0, H_1, H_2, \ldots, H_{2t-1}, H_{2t})\) through step \(2t\) produced by the COLLISION(1) algorithm on inputs in \( I_v \). By construction, no write occurs at step \(2t-1\) in the COLLISION(1) algorithm (i.e., \( H_{2t-1} = H_{2t-2} \)) if and only if no write would have occurred at step \( t \) in the COMMON(1) algorithm. In this case, each nontrivial clause in \( f_{w'} \) is also a nontrivial clause in \( f_{wv'} \). Since \( I_v \subseteq I_w \), there is nothing to prove.

Otherwise, node \( v' \) is the child of \( w' \) arising when the value \( H_{2t} \) is written at the \( t\)th step in the COMMON(1) algorithm. Compared to \( f_{wv'} \), the formula \( f_{wv'} \) contains at most one additional nontrivial clause, consisting of the literals corresponding to those private bit values that cause processors to write the value \( H_{2t} \).

If \( H_{2t-1} \) is the index of processor \( P_1 \), then, for every input in \( I_v \), \( P_1 \) is the only processor that writes at step \(2t-1\) in the COLLISION(1) algorithm. Thus, the literal (either \( x_i \) or \( \overline{x_i} \)) causing \( P_1 \) to write the value \( H_{2t} \) at step \( t \) in the COMMON(1) algorithm is true for all inputs in \( I_v \). Notice that the additional nontrivial clause in \( f_{wv'} \), if there is one, contains this literal, thereby satisfying the conditions of the claim.

Finally, if \( H_{2t-1} \) is the collision symbol, then, for every input in \( I_v \), at least two processors write at step \(2t-1\) in the COLLISION(1) algorithm and, therefore, would have written at step \( t \) in the COMMON(1) algorithm. Hence at least two literals in the additional clause are true. This concludes the proof of the claim. \( \square \)

We are now ready to complete the proof of Theorem 10. Let \( v \) be a node of depth \(2t\) in the COLLISION(1) tree and let \( w \) be the child corresponding to a collision occurring at step \(2t+1\). We will show that the values written by any processor at the \((t+1)\)st step of the COMMON(1) algorithm, for any input in \( I_v \), are all the same. Hence, \( P_j \) can determine this value from its knowledge of \( I_v \) and the programs of the other processors.

Let \( a \) be an input in \( I_v \) and assume that there is another input in \( I_v \) for which, at the \((t+1)\)st step in the COMMON(1) algorithm, a different value is written. Suppose that \( P_j \) is a processor writing this other value and that it does so because literal \( l_j = 1 \).

Consider the input \( b \) obtained from \( a \) by making \( l_j = 1 \). Then \( b \notin I_w \); otherwise the COMMON model would be violated. Since a collision occurs, two or more true literals occurring in \( a \) cause a particular value to be written. Thus at least one processor would write that value at step \( t+1 \) on input \( b \). Since \( l_j = 1 \) in \( b \), \( P_j \) would simultaneously write another value.

Because \( a \in I_v \) and \( b \notin I_v \), \( f_w(a) = 1 \) and \( f_w(b) = 0 \). Now \( f_w \) is a formula in conjunctive normal form. Therefore \( f_w \) contains a clause \( q \) such that \( g(a) = 1 \) and \( g(b) = 0 \). Since \( b \) is obtained from \( a \) by changing \( l_j \) from 0 to 1, \( l_j \) is the only literal in \( g \) satisfied by \( a \). By the claim, \( l_j = 1 \) is true for all inputs in \( I_v \). This is a contradiction. \( \square \)

6. Simulating PRIORITY\((km)\) by ARBITRARY\((m)\). Section 2 considered the simulation of PRIORITY machines by ARBITRARY machines with more memory. Here we study the “complementary” problem of simulating PRIORITY machines by ARBITRARY machines with less memory. As a corollary, we obtain a separation between PRIORITY and ARBITRARY machines with the same amount of memory.

Our goal is to solve the \( km \)-colour MINIMIZATION problem in the ARBITRARY\((m)\) model. This can clearly be done in time \( O(k \log n) \), by dividing the colours into \( k \) groups of \( m \) colours and solving one group at a time by the algorithm of Theorem 1. A proof similar to that of Corollary 6.3 shows that the \( km \)-colour MIN-
IMIZATION problem requires $\Omega(km \log(n/km))$ steps on COMMON(1), and thus $\Omega(k \log(n/km))$ steps on COMMON(m). In fact, it is possible to do considerably better on ARBITRARY(m), as the following theorem and corollary demonstrate. Hence ARBITRARY is another example of a computational model in which "mixing" the computation of several functions on disjoint sets of inputs enhances efficiency. Boolean and arithmetic circuits also exhibit this property ([P], [U], [AHU]).

**Theorem 11.** On ARBITRARY(1), the $m$-colour MINIMIZATION problem can be solved in $O\left(\frac{m \log n}{\log m}\right)$ steps.

When $m = O(n^\epsilon)$ for some constant $\epsilon > 0$, this upper bound $O(m)$ matches the trivial lower bound. In comparison with algorithms that solve the problem one colour at a time, this solution uses an average of $O(1)$ steps per colour. This disproves a conjecture of Vishkin [V].

The idea is to try to solve the $m$ different problems concurrently, although we have only one common memory cell. We say that a processor $P_i$ has colour $c$ if $x_i = c$. For each colour $c$, processors maintain an ordered set $S_c \subset \{1, \ldots, n\}$ and a "current winner" $w_c \in \{1, \ldots, n, \infty\} = S_c$. If $w_c < \infty$, then $w_c$ is the smallest index of any processor globally known to have colour $c$. When there is no processor that is globally known to have colour $c$, as is the case initially, $w_c = \infty$. The set $S_c$ consists of the indices of those processors that may replace the current winner. In particular, $i \in S_c$ implies that $i < w_c$. Initially, $S_c = \{1, 2, \ldots, n\}$.

The algorithm proceeds in phases. In a single phase, each set $S_c$ is shrunk by approximately a factor of $m$. When $S_c = \emptyset$ for all $c$, the algorithm can halt.

At the beginning of a phase, each set $S_c$ is divided into $m$ pieces $S^1_c, S^2_c, \ldots, S^m_c$ of size at most $\left[\frac{|S_c|}{m}\right]$, where the processors in $S^k_c$ have lower indices than those in $S^{k+1}_c$ for $a < b$. The goal in a phase is to publicly determine, for each colour $c$, the first group among $S^1_c, S^2_c, \ldots, S^m_c$ containing the index of some processor having colour $c$. Then $S_c$ is updated appropriately.

Conceptually, these sets are arranged in an $m \times m$ array; the entry in row $c$ and column $i$ is $S^{j}_c$. At each step of the phase, we either eliminate a row or the leftmost column of the array. The set $C$ will consist of those colours whose rows have not yet been eliminated in this phase.

If processor $P_j$ has colour $c$ and $j$ belongs to the group in the leftmost column of the row corresponding to colour $c$, then it attempts to write its index and colour into memory cell $M_j$. Throughout the phase, the invariant is maintained that for any colour $c \in C$, no processor with that colour lies in $S^k_c$ for any eliminated column $k$. Hence, when $(j, c)$ appears in $M_j$, any processor having colour $c$ and with index lower than $j$ must be in the group currently in the leftmost column of row $c$. In this case, row $c$ is eliminated from the array. If no write occurs at a given step, then none of the groups in the leftmost column contain the winner for their row and the column can be eliminated. More formally, the phase proceeds as follows.

- $C \leftarrow \{1, 2, \ldots, m\}$
- $i \leftarrow 1$

While $C \neq \emptyset$ and $i \leq m$ do
- If $j \in S^i_j$ and $x_j \in C$, processor $P_j$ will attempt to write $(j, x_j)$ into $M_j$.
- If $(j, c)$ appears in $M_j$, remove $c$ from $C$, set $w_c \leftarrow P_j$, and shrink $S_c$ to $\{k \in S^i_c : k < j\}$
- Otherwise set $i \leftarrow i + 1$.

At each step, either $|C|$ is decreased by one or $i$ is increased by one. A phase thus takes at most $2m - 1$ steps, and any set $S_c$ which was of size $s$ before the phase is of
size at most $\left\lceil \frac{a}{m} \right\rceil - 1 < \frac{a}{m}$ at the end of the phase. Thus $O\left(\frac{\log n}{\log m}\right)$ phases suffice. \(\square\)

**Corollary 11.1.** One step of $\text{PRIORITY}(km)$ can be simulated by $O\left(\frac{k \log n}{\log k}\right)$ steps of $\text{ARBITRARY}(m)$, for $k > 1$.

**Proof.** Simulating one step of $\text{PRIORITY}(km)$ is equivalent to solving the $km$-colour MINIMIZATION problem; divide the colours into $m$ groups of $k$ colours each, and use the algorithm above to solve each group in parallel. \(\square\)

The following lower bound proves this procedure optimal for $km = O(n^{1-\epsilon})$, and proves a separation between $\text{PRIORITY}(m)$ and $\text{ARBITRARY}(m)$.

**Theorem 12.** The $km$-colour MINIMIZATION requires $\Omega\left(\frac{k \log n (km)}{\log (k+1)}\right)$ steps to solve on $\text{ARBITRARY}(m)$.

**Corollary 12.1.** $\text{ARBITRARY}(m)$ requires at least $\Omega(\log(n/m))$ steps to simulate one step of $\text{PRIORITY}(m)$.

To simplify the proof of Theorem 12, we divide the processors into $km$ groups of size at least $\lceil n/km \rceil$, and declare that the processors in group $i$ will have colour either $i$ or 0. Note that we can define this restricted problem over domain $\{0, 1\}^n$.

We maintain a history $H_0, H_1, \ldots, H_t$, a set $S$ of fixed positions, and a set $F$ of free positions. Initially, the processor of highest index in each group has its colour fixed to 1, and all other positions are free. We maintain the invariant that for any free position, there are no positions of lower index that are within the group and fixed to 1.

Our measure of the algorithm's progress against the adversary strategy will be by means of a potential function. If there are $s_i$ fixed positions in group $i$, then this function has value $\sum_{i=1}^{km} \log_{k+1}(s_i + 1)$. Initially, then, the function has value at least $km \log_{k+1}(n/km)$. We shall show that the adversary can fix positions in such a fashion so that the history is determined through step $t$ and the value of this function decreases by at most $ctm$, for some absolute constant $c$.

As long as there is at least one free position (that is, the value of the potential function is nonzero), the algorithm has not solved all colours, thus establishing the lower bound.

Given history $H_0, H_1, \ldots, H_t$, and the fact that group $i$ contains $s_i$ free positions, we show how to fix $H_{t+1}$ and cause a drop in the potential function of at most $cm$. Initially, the contents of each cell in $H_{t+1}$ are unfixed. Suppose the free positions in group $i$ at any point are $j_1, j_2, \ldots, j_s$. We define $lower_i$ to be the lowest $\frac{1}{k+1}$st of the free positions in group $i$, that is, positions $j_1$ through $j_{\lfloor s/(k+1)\rfloor}$. $upper_i$ is defined to be all free positions in group $i$ not in $lower_i$.

1) If a processor in any fixed position writes at step $t+1$ into an unfixed cell, declare one such processor to win the competition to write into that cell for all allowable inputs. The value of the potential function does not change. Repeat this step until all cells are fixed or no such processor exists.

2) If any processor in any free position writes on into an unfixed cell, choose one such processor, fix its colour to 0, and declare it to win the competition to write at time $t+1$. If it is in group $i$, then the potential function drops by

$$\log_{k+1}(s_i + 1) - \log_{k+1} s_i = \log_{k+1} \left(1 + \frac{1}{s_i}\right) \leq \log_{k+1}(2) \leq 1.$$  

Repeat this step until all cells are fixed or no such processor exists.

3) Once the first two cases are taken care of, then processors write into unfixed cells only if they receive their colour. If there is a processor $P_j$ in a free position in group $i$ such that $P_j$ writes into an unfixed cell on colour $i$, and $j \in upper_i$, then fix $upper_i$ to colour $i$ and delete the cell for all consistent inputs.

$$\log_{k+1}(s_i + 1) - \log_{k+1} s_i = \log_{k+1} \left(1 + \frac{1}{s_i}\right) \leq \log_{k+1}(2) \leq 1.$$  

Repeat this step until all cells are fixed or no such processor exists.

4) Fix $lower_i$ to 0 for all groups, unfixed cells at this step, for all consistent inputs, this function is at most

$$\sum_{j=1}^{km} \log_{k+1}(s_j + 1) - \sum_{j=1}^{km} \log_{k+1}(r_j).$$

Steps 2 and 3 of the adversary strategy for each step fixes one cell. Hence there is a total potential drop is thus at most $ctm$.

The proof of Theorem 12 is when $m = 1$ in $[1, \ldots, n]$, and of the similar bound for the case $k = 1$, when in constant time.

**7. Conclusions.** In this paper we have shown how to simulate one step of a machine using shared memory cells. If the domain of the function is then either the programs for the machine, or the results are, for the most part, then $\text{COMMON}(1)$ can be simulated by $\Theta\left(\frac{n \log_{k+1}(n/km)}{\log_{k+1}(2)}\right)$ steps to $\text{PRIORITY}(km)$, $\text{ARBITRARY}(m)$ can be improved to $\Omega(m \log n/m)$.

The upper bound for $\text{COLLISION}(m)$, or $\text{COMMON}(m)$ is the function $\Omega(\log n/m^2)$. The proof is based on Lemma 6.1 and Theorem 10. The more direct for simulating $\text{PRIORITY}(m)$ on $\text{COMMON}(m)$ or $\text{PRIORITY}(km)$ by $\text{COLLISION}(m)$ is implied by Corollary 11.1 and $O(1)$ time, but this simulation in $O(1)$ time?

$$\log_{k+1}(s_i + 1) - \log_{k+1} s_i = \log_{k+1} \left(1 + \frac{1}{s_i}\right) \leq \log_{k+1}(2) \leq 1.$$  

Repeat this step until all cells are fixed or no such processor exists.
Thus $O\left(\frac{\log n}{\log m}\right)$ phases suffice. \hfill $\square$

$(n)$ is equivalent to solving the $km$-ary problem into $m$ groups of $k$ colours each, in parallel. \hfill $\square$

4) If $\phi(km) = O(n^{1-\epsilon})$, and $\phi(n)$ is not \textit{ARBITRARY}($m$), then $\phi(n/km) \in \Omega\left(\frac{k \log(n/m)}{\log(k+1)}\right)$ steps to simulate.

Thus, we have $\phi(n/km) \in \Omega(\log(n/m))$ steps to simulate.

Since the processors into $km$ groups of processors, process $i$ will have colour either even or in each group has its colour fixed.

Therefore, the processors in group $i$ will have colour either even or odd, and we have the invariant that for any free position $j$, $s_j$ is within the group and fixed to be even.

The adversary strategy will be as follows: in each free position in group $i$, then this $j$ is odd, then the function has value at most $c$.

The adversary can fix positions in such a way that the function value of the function is at most $c$.

Let $s_i$ be the value of the potential function for the group $i$.

Let $s_i$ be the value of the potential function for the group $i$.

Let $s_i$ be the value of the potential function for the group $i$.

The converse problem of simulating one step of a machine with $m$ shared memory cells by another with one shared memory cell is not as well understood. The complexity of simulating $\text{PRIM}	ext{ORITY}(m)$ by \text{ARBITRARY}(1) is $\Theta\left(\frac{m \log n}{\log m}\right)$. For $\text{COMMON}(1)$ simulating $\text{PRIM}	ext{ORITY}(m)$, \text{ARBITRARY}(m), or $\text{COMMON}(m)$ the lower bound can be improved to $\Omega(m \log(n/m))$. However, our upper bound is $O(m \log n)$.

The upper bound for $\text{COMMON}(1)$ simulating $\text{PRIM}	ext{ORITY}(m)$, \text{ARBITRARY}(m), or $\text{COMMON}(m)$ is the same, except when $\text{COMMON}(m)$ is computing a function with domain $\{0, 1\}^m$. In this case, an $O(m)$ upper bound follows from Lemma 6.1 and Theorem 10. This is clearly optimal. The $\Omega\left(\frac{m \log n}{\log m}\right)$ lower bound for simulating $\text{PRIM}	ext{ORITY}(m)$ on $\text{COMMON}(1)$ is a direct consequence of the lower bound on $\text{ARBITRARY}(1)$.

The case of simulating a model with $m$ cells by another with $m$ cells is also not as well understood. We have shown that the complexity of simulating $\text{COMMON}(m)$ by $\text{COMMON}(m)$ or simulating $\text{PRIM}	ext{ORITY}(m)$ by $\text{ARBITRARY}(m)$ is $O(\log n - \log m)$. For small values of $m$, i.e., $m = O(n^{1-\epsilon})$ for some constant $\epsilon > 0$, these results are tight to within a constant factor. Nothing is known about the complexity of simulating $\text{COMMON}(m)$ by $\text{COMMON}(m)$ (for $m > 1$) except the upper bounds of $O(\log n)$ (implied by Corollary 1.1) and $O(m)$ (implied by Theorem 10). Is it possible to do this simulation in $O(1)$ time?
No other separation between models with the same (finite) amount of shared memory is known for higher values of $m$. More work and probably other techniques are needed to extend these results for all ranges of $m$. To improve these results, we must understand in a more fundamental way how processors can use larger amounts of memory to communicate. Li and Yesha [LY1], [LY2] have made a first step in this direction by considering concurrent-read concurrent-write PRAM's with a small shared memory plus $n$ cells of read-only memory containing the input.

Another relevant result concerns the problem of element distinctness on machines with an infinite amount of shared memory. In this problem, $n$ integers are stored in the first $n$ cells of shared memory, and the machine must decide whether or not there exist two which are equal. In [FMW] it is shown that element distinctness requires $\Omega(\log \log \log n)$ steps on $\text{COMMON}(\infty)$ but $O(1)$ steps on $\text{COLLISION}(\infty)$. The lower bound has been improved to $\Omega(\sqrt{\log n})$ steps [RSSW]. Both these results imply the existence of a function $f(n)$ such that the corresponding separation holds between $\text{COMMON}(f(n))$ and $\text{COLLISION}(f(n))$. For the $\Omega(\log \log \log n)$ result, $f(n) = 2^{\log^2 n}$, but for the $\Omega(\sqrt{\log n})$ result, $f(n)$ grows much more rapidly with $n$.

We conclude by mentioning a surprising recent result [FRW2], which shows that one step of $\text{PRIORITY}(m)$ can be simulated by $O\left(\frac{\log n}{\log \log n}\right)$ steps of $\text{COMMON}(nm)$. This shows that allowing more memory can lead to improved simulations even if the number of processors is held fixed, and also that the separation between $\text{PRIORITY}(\infty)$ and $\text{COMMON}(\infty)$ is not $\Theta(\log n)$.

REFERENCES


the same (finite) amount of shared work and probably other techniques of $m$. To improve these results, we show processors can use larger amounts [1],[LY2] have made a first step in concurrent-write PRAM's with a small number containing the input.

of element distinctness on machines is a problem, $n$ integers are stored in a machine must decide whether or not they are distinct in $O(1)$ steps on COLLISION($\infty$).

steps [RSSW]. Both these results (the corresponding separation holds [U]). For the $\Omega(\log \log \log n)$ result, it grows much more rapidly with $n$. The result [FRW2], which shows that $(\log n)^2$ steps of COMMON($nm$) to improved simulations even if the separation between PRIORI-

---

**REFERENCES**


R. Reischuk, Simultaneous writes of parallel random access machines do not help to compute simple arithmetic functions, manuscript, 1984.


