Relations Between Concurrent-Write Models of Parallel Computation

Faith E. Fich

University of Washington

Prabhakar L. Ragde
Avi Wigderson

University of California at Berkeley

ABSTRACT

Shared-memory models for parallel computation (e.g. parallel RAMs) are very natural and already widely used for parallel algorithm design. The various models differ from each other mainly in the way they restrict simultaneous processor access to a shared memory cell. Understanding the relative power of these models is important for understanding the power of parallel computation.

Two recent pioneering works shed some light on this question. Cook and Dwork [CD] (resp. Snir [S]) present problems that, for instances of size n, can be solved in O(1) time on an n-processor PRAM that allows simultaneous write (resp. read) access to shared memory, but require $\Omega(\log n)$ time on a PRAM that forbids simultaneous write (resp. read) access, regardless of the number of processors.

When allowing simultaneous write access, the model must include a write-conflict resolution scheme. Three such schemes were suggested in the literature, and in this paper we study their relative power. Here the situation is more sensitive, as a small increase in the number of processors allows constant time simulation of the strongest by the weakest. By fixing the number of processors and parametrizing the number of shared memory cells, we obtain tight separation results between the models, thereby partially answering open questions of Vishkin [V]. New lower bounds techniques are developed for this purpose.

Support for this research was provided by NSF grant DCS-8110584, DARPA Contract No. N00039-82-C-0235 and by an NSERC postgraduate scholarship.

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). Not only have numerous algorithms been designed for the CRCW PRAM, but it has also been shown to be closely related to unbounded fan-in circuits and alternating Turing machines [CSV].

Specifically, a CRCW PRAM consists of a set of processors (i.e., random access machines) $P_1, P_2, \cdots P_n$ together with a shared memory. One cycle of computation 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 to simultaneously write into the same

memory cell. An arbitrary amount of computation will be allowed in each compute phase so that we can concentrate on communication between processors.

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. In the case that the priority is given to the processor of lowest index [G], we call this the MINIMUM 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]. The COMMON model allows simultaneous writes to the same memory cell only if all processors doing so are writing a common value [K]. 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. 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 MINIMUM 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 MINIMUM 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 model is at least as powerful as the CREW PRAM, and the CREW PRAM is at least as powerful as the EREW PRAM.

Our aim is to understand the relative power of these models. Algorithms running on all of 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 and Dwork have shown that the CREW PRAM is strictly less powerful than the CRCW PRAM. In particular, their work [CD] 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 for an element in a sorted list, 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 three CRCW models as a function of the number of shared memory cells m (called the *communication width* [VW]) when the number of processors is held fixed at n. This is an important restriction, since one step on the MINIMUM model is easily simulated by two steps on the COMMON model if the number of processors is squared and sufficient common memory is allowed. When width is restricted, however, the three models are not equivalent. Restricting width has a meaning in a practical as well as theoretical sense; an Ethernet may be considered a type of 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 separations in lines 1 and 2 of the table are demonstrated by the lower bounds in section 2; all upper bounds are given in section 3. The lower bounds on lines 1 and 2 are proved tight in section 3 by providing corresponding upper bounds. The result in the third line of the table gives (when $c = 1$) a simple and general simulation of any of the three models on another of the same width without performing any sorting. The upper bound in the fourth line provides evidence for a $\Theta(\log n)$ separation between the COMMON(m) and ARBITRARY(m) when $m = O(n^{1-\epsilon})$; in section 4 we conjecture this separation, and present some partial results that may lead to proving it. The results in the table leave some questions unanswered; more work and probably other techniques are needed to extend these results for all ranges of m.

Table 1

Simulating Machine	Simulated Machine	Time Bounds
COMMON(m)	ARBITRARY(1)	$\Theta\left(\frac{\log n}{\log(m+1)}\right)$
ARBITRARY(m)	MINIMUM(1)	$\Theta\left(\frac{\log n}{\log(m+1)}\right)$
COMMON(cm) $[m = O(n/c)]$	MINIMUM(m)	$O\left(\frac{\log n}{\log(c+1)}\right)$
ARBITRARY(1)	MINIMUM(m) $[m>1]$	$O\left(\frac{m \log n}{\log m}\right)$

New lower bound techniques are developed to obtain the results above. We consider this work as another step (following [S] [CD] [VW]) in forming a foundation of lower bound techniques for parallel computation. Also, as our lower bounds are on the communication between processors, we believe these techniques may be applied to distributed (asynchronous) computation as well (e.g. in the Ethernet model).

2. Lower Bounds

2.1. Separating ARBITRARY(1) from COMMON(m)

The problem of simulating one write step of an ARBITRARY(m) is equivalent to the following problem:

m-colour REPRESENTATIVE

Before: Each processor P_i $(1 \le i \le n)$ has a colour c_i $(0 \le c_i \le m)$ known only to itself.

After: Each processor P_i knows a 0-1 value s_i, where $s_i = 1$ for exactly one processor among those with each particular nonzero colour c, and 0 otherwise.

In the simulation, c_i represents the memory cell that the simulated P_i wishes to write into; $c_i = 0$ if P_i does not wish to write. Once the problem is solved, P_i will write iff $s_i = 1$, thus avoiding write conflicts.

Clearly, the 1-colour REPRESENTATIVE problem takes only one step on an ARBITRARY(1). The following theorem provides a separation between this model and COMMON(m), by providing a corresponding lower bound.

Theorem 1

The 1-colour REPRESENTATIVE problem requires $\left\lfloor \frac{\log n}{\log(m+1)} \right\rfloor$ steps on a COMMON(m).

Proof:

We give the simple proof for $m=1$. The 'input' to an algorithm A solving this problem is a specification of the values of all the c_i. (Note that now $c_i \in \{0,1\}$). We will use an adversary argument, constructing an input on which A requires $\lfloor \log n \rfloor$ steps.

The write action of a particular processor P_i at step t depends only on c_i and the sequence of contents $(H_0, H_1, \cdots H_{t-1})$ of the common memory cell, where H_i is the contents of M_1 before step $i+1$. We call this sequence the *history* through time t. Given a fixed history, we say P_i *writes on* 0 (resp. 1) if it attempts to write to the single memory cell M_1 when $c_i = 0$ (resp. 1).

At each step we will 'fix' the values of certain c_i. This will be done in a manner such that all inputs whose values at the fixed positions agree with the corresponding fixed values, and which contain at least one 1 (termed the "consistent set" of inputs) will produce the same history up to that step. All the inputs in the consistent set are thus indistinguishable to the algorithm.

Positions not fixed are said to be *free*. Our method of fixing positions will always fix values to 0. Suppose there are at least two free positions i and j

after some step T. Then there exists an input I_i in our consistent set in which c_i is the only bit set to 1. Similarly, there exists an input I_j in which c_j is the only bit set to 1, and an input I_{ij} in which both c_i and c_j are 1. Since these three inputs cannot be distinguished, neither P_i nor P_j can assume that it is the representative. Thus, if there are at least two free positions after some step, the algorithm cannot answer after that step.

Initially all positions are free and all inputs except the all-zero input are possible. Now suppose that after step t, we have a set S of fixed positions, a set F of free positions, and all inputs consistent with S have history $\{H_0, H_1, \cdots H_t\}$. We can assume without loss of generality that no processor whose position is fixed attempts to write.

1) If no processor writes into M_1 on 0 or 1 at step $t+1$, then we fix no positions, and $H_{t+1} = H_t$.

2) If there exist processors writing on 0 at step $t+1$, choose one such processor P_i, fix c_i to 0 (move i from F to S), and set H_{t+1} to be whatever P_i writes. (Because all inputs consistent with S generate the same history, P_i will write the same thing at step $t+1$ on all such inputs.)

3) Otherwise: all processors who are writing are writing on 1. Let W be the set of processors in free positions who write on 1 at step $t+1$.
   i) If $|W| > |F|/2$, then fix all free positions not in W to 0. Since we do not consider the all-zero input, then for every input consistent with the new S, some processor in W receives a 1. Furthermore, there is an input in our consistent set in which all processors in W receive a 1, and thus they must all attempt to write the same value. Let H_{t+1} be this value.
   ii) If $|W| \leq |F|/2$, fix all positions in W to 0. No one writes at step $t+1$, and $H_{t+1} = H_t$.

In all cases the number of free positions is cut by at most a factor of 2. If the number of free positions is greater than one at some step T, then the algorithm must take at least $T+1$ steps to answer on some input consistent with the fixed positions. Since we started with n free positions, this gives us the lower bound of $\lfloor \log n \rfloor$.

This proof generalizes to arbitrary m to prove the theorem as stated. ●

2.3. Separating MINIMUM(1) from ARBITRARY(m)

The preceeding theorem demonstrated a separation between an ARBITRARY(1) and a COMMON(m). We can show a similar separation between a MINIMUM(1) and an ARBITRARY(m) by considering the following problem:

m-colour MINIMIZATION

Before: Each processor $P_i (1 \leq i \leq n)$ has a colour c_i known only to itself.
After: Each processor P_i knows the value a_i, where

$$a_i = \begin{cases} 1, & \text{if for all } j<i,\ c_i \neq c_j \text{ and } c_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

(that is, $a_i = 1$ iff P_i is the processor of lowest index with colour c_i)

Simulation of a MINIMUM(m) is equivalent to solving the m-colour MINIMIZATION problem in the same way that simulating an ARBITRARY(m) was equivalent to solving the m-colour REPRESENTATIVE problem.

Clearly, 1-colour MINIMIZATION requires only one step on a MINIMUM(1). The following theorem provides the separation between this model and an ARBITRARY(m) or COMMON(m).

Theorem 2

1-colour MINIMIZATION requires $\left\lfloor \frac{\log n}{\log(m+1)} \right\rfloor$ steps on an ARBITRARY(m).

Proof:

Let A be an algorithm which solves this problem. As adversary to an algorithm running on an ARBITRARY(m), we are allowed not only to choose the input to A, but to decide who wins each competi-

tion to write.

We will (as before) maintain a set S of fixed positions, a set F of free positions, a "consistent set" of inputs whose definition depends on S, and H_t, a specification of the contents of the m memory cells before step $t+1$, for t=0,1... Since we will fix positions to 1 as well as 0, the properties of these sets will be somewhat different than in the previous proof:

- For each position in S, we also record the value to which that position is fixed.
- Our "consistent set" will be *all* inputs consistent with the fixed positions in S.
- A will produce history $(H_0, H_1, \cdots H_t)$ on every consistent input; thus, these inputs are indistinguishable to A before step $t+1$.
- For each free position j in F , there are no positions of lower index fixed to 1. Thus if there are two free positions i and j after step t, there is a consistent input for which c_i is the 1 of lowest index (set $c_i=1$, all other free positions to 0) and an input for which c_j is the 1 of lowest index. Since A cannot distinguish these two, we can conclude that if F has more than one free position, A cannot solve the problem in t steps.

Initially, $S=\phi$, and H_0 is the initial contents of memory. These satisfy the conditions above. Now suppose we have S and $H_0, H_1, \cdots H_t$ satisfying the conditions above. Assume the number of free positions in F is at least $m+1$.

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

2) Suppose there are r cells not taken care of in step 1 into which processors write only on 1. 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 $\left\lfloor \frac{f}{r+1} \right\rfloor$, and so on. (The last $f \bmod (r+1)$ groups will contain $\left\lceil \frac{f}{r+1} \right\rceil$ free positions). With each such cell M_j, associate the group containing the processor of *highest* index writing into it on 1; call this processor P_{i_j}. Since there are r cells and $r+1$ groups, at least one group will have no cells associated with it; suppose such a group G comprises free positions p through q. The idea is that by either forcing no one to write or forcing the processor of highest index to write, the adversary can provide "no information" about group G.

   i) Fix all free positions with index less than p to 0. Any cells associated with groups containing free positions less than p will have c_i of *all* processors writing into them on 1 set to 0; hence, no one will write into these cells at step t+1 for any input consistent with S, and their contents in H_{t+1} will be their contents in H_t.

   ii) Fix all free positions with index greater than q to 1. For any cell M_j associated with a group containing free positions greater than q, we declare P_{i_j} to win the competition to write into M_j at step t+1 for every input consistent with S, thus fixing the contents of M_j in H_{t+1}.

3) The remaining cells have no processor writing into them on any input in the consistent set at step $t+1$; the contents of these cells in H_{t+1} will be their contents in H_t.

If the number of free positions is less than $m+1$ at this point, say at time T, the algorithm can force us to fix all remaining free positions, and so we

do not continue the construction. Intuitively, the number of free positions is cut down by at most a factor of $m+1$ on each step. More precisely, if f_t is the number of free positions at time t, we see that

$$f_0 = n$$

$$f_{t+1} \geq \left\lfloor \frac{f_t}{m+1} \right\rfloor$$

and $f_T \leq m$. Noting that the algorithm requires T+1 steps on any input in the consistent set at time T, we have the lower bound of $\left\lfloor \frac{\log n}{\log(m+1)} \right\rfloor$, as required. •

3. Upper Bounds

3.1. Simulating MINIMUM(1) by COMMON(m)

Theorem 3

The 1-colour MINIMIZATION problem can be solved in $O\left(\left\lfloor \frac{\log n}{\log(m+1)} \right\rfloor\right)$ steps on a COMMON(m).

Proof:

Throughout the algorithm, memory cells will contain only 1's and 0's. Note that for the 1-colour MINIMIZATION problem, $c_i \in \{0,1\}$. We call the leftmost processor the winner.

One iteration of the algorithm goes as follows: First, all shared memory cells are set to 0. The processors are divided into $m+1$ nearly equal groups, where each group contains a set of consecutively numbered processors. The first $n \bmod (m+1)$ groups contain $\left\lceil \frac{n}{m+1} \right\rceil$ processors and the rest contain $\left\lfloor \frac{n}{m+1} \right\rfloor$. Processor P_i in the j^{th} group $(1 \leq j \leq m)$ will write 1 into M_j iff $c_i=1$. At this point, if all memory cells are unchanged, the winner is in the $(m+1)^{th}$ 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 if his group wins or not. The following subroutines are used to decide which of the above two cases holds, in constant time.

EMPTY MEMORY (m-way OR)
Before: Cells M_i $(1 \leq i \leq m)$ each contain 1 or 0
After: M_1 contains 0 iff all M_i were initially 0.
Procedure: Processor P_i will read M_i and, if it is 1, write 1 into M_1.

LEFTMOST ONE IN MEMORY
(assume $m \leq \sqrt{n}$)
Before: Cells M_i $(1 \leq i \leq m)$ each contain 1 or 0.
After: M_i contains 1 iff all M_j for $j<i$ were initially 0 and M_i was initially 1.
Procedure: Processor i forms the ordered pair (j,k) from its name by setting $j \leftarrow (i \bmod m)+1$ and $k \leftarrow i - m(j-1)$. If $j<k$ and $M_j=1$, it writes 0 into M_k.

After the LEFTMOST ONE IN MEMORY algorithm is applied, and 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 is 1 or 0 respectively). Note that this algorithm requires m^2 processors; if $m > \sqrt{n}$, we only use the first $\sqrt{n}$ cells of memory (i.e., divide into only $\sqrt{n}$ groups). Applying EMPTY MEMORY will then allow the $(m+1)^{st}$ group to decide if it is the winning group or not.

All processors except the ones in the winning group set $c_i=0$ 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 who may still be the winner after the t^{st} read,

$$g_1 = n$$

$$g_t \leq \left\lceil \frac{g_{t-1}}{m+1} \right\rceil$$

If T is the number of iterations the algorithm takes, we have $g_T \leq 1$, giving $T \leq \left\lceil \frac{\log n}{\log(m+1)} \right\rceil$, as required. ●

Corollary: The 1-colour REPRESENTATIVE PROBLEM can be solved in time $O\left(\left\lceil \frac{\log n}{\log(m+1)} \right\rceil\right)$ on a COMMON(1).

Theorem 3 states the upper bounds that appear in lines 1 and 2 of Table 1. The following theorem gives the upper bound that appears in line 3 of Table 1.

Theorem 4

Simulating one step of a MINIMUM(m) can be done on a COMMON(cm) in $O\left(\left\lceil \frac{\log n}{\log(c+1)} \right\rceil\right)$, provided $m = O(n/c)$.

Proof:

Simulating one step of a MINIMUM(m) is equivalent to solving the m-colour MINIMIZATION problem. This can be considered as m simultaneous 1-colour MINIMIZATION problems; devote c cells of the COMMON(cm) to each problem, and solve them in parallel using the algorithm of Theorem 3. The restriction on m ensures that there are enough processors to do this.

3.3. Simulating MINIMUM(m) by ARBITRARY(1)

The problem that was studied in the previous subsection was the simulation of MINIMUM(1) by ARBITRARY(m) and COMMON(m). Here we study the "complementary" problem of simulating MINIMUM(m) by ARBITRARY(1) and COMMON(1) models. This problem yields insight into the strength of the ARBITRARY model relative to the COMMON model, as will be discussed in the next section. In what follows we obtain a nontrivial upper bound for simulating MINIMUM(m) by ARBITRARY(1).

Our goal is to solve the m-colour MINIMIZATION problem in the ARBITRARY(1) model. This can clearly be done in time $m \log_2 n$, by solving the problem one colour at a time and using the algorithm of Theorem 3. We show below how to solve the problem in time $O(\frac{m \log n}{\log m})$, which says that for $m = n^{\epsilon}$ ($\epsilon > 0$ fixed) we need only $O(m)$ time, matching the trivial lower bound. In comparison with the one-at-a-time algorithm, this solution uses an average of $O(1)$ steps per colour.

The idea is to try to solve the m different problems concurrently, although we have only one common memory cell. We show first how this can be done for a simpler problem.

Consider the following special case of m-colour MINIMIZATION, called m-group MINIMIZATION. In this problem, the n processors are partitioned into m groups of sizes $n_1, n_2, \ldots, n_m$ where $n = \sum_{i=1}^{m} n_i$. For $j = 1, \ldots, m$, only processors in group j are allowed to have colour j. Therefore, if a processor P belongs to group j, then its colour c is either 0 or j. The information about c can then be represented by a single bit b, where $b = 1$ iff $c = j$. Formally, the problem can be defined as follows:

m-group MINIMIZATION

Before: There are n processors $P_{i,j}$ ($1 \leq i \leq n_j$, $1 \leq j \leq m$), each with a private bit $b_{i,j}$.
After: Each processor $P_{i,j}$ knows the value $a_{i,j}$, where

$$a_{i,j} = \begin{cases} 1 \text{ if } b_{i,j} = 1 \text{ and } b_{q,j} = 0 \text{ for } 1 \leq q < i \\ 0 \text{ otherwise} \end{cases}$$

($a_{i,j} = 1$ iff $P_{i,j}$ is the processor of lowest index within the j^{th} group which has color j)

Theorem 5

m-group MINIMIZATION can be solved in time $m + \max_{1 \leq i \leq n} n_i$ on an ARBITRARY(1).

Proof:

We use the following simple algorithm. Initially all groups are "unsolved".

```
i ← 1;
repeat until all groups solved
  For all unsolved groups j,
      if b_{i,j} = 1 then P_{i,j} writes j into M_1;
  If M is unchanged (no one wrote) then
      i ← i+ 1;
      any group j with n_j < i is solved
      (a_{i,j} = 0 for all j)
  else if M contains j
      group j is solved
      (a_{i,j} = 1, a_{k,j} = 0 for k ≠ i)
end repeat;
```

Note that, in each step, either one group is solved, or the size of all groups decreases by 1. The claimed running time follows. •

Note that the above algorithm will not solve m-colour MINIMIZATION quickly, because the size of a colour may be $O(n)$. But a similar technique will work: instead of having just one processor from each colour write, we have an initial fraction α of processors from each colour write, where α depends on the number of unsolved colours. If some processor succeeds in writing, all processors of that colour not in the initial fraction drop out; if no one attempts to write, all processors in all initial fractions drop out, and the process is repeated. We can show:

Theorem 6

m-colour MINIMIZATION can be solved on an ARBITRARY(1) in time $O(\frac{m \log n}{\log m})$. •

The proof of this theorem requires substantial analysis, and is deferred to the final version of this paper.

4. Towards Separating ARBITRARY(m) from COMMON(m)

In the last section we gave an upper bound on the simulation of MINIMUM(m) by ARBITRARY(1). In this section we explain how a lower bound on the simulation of MINIMUM(m) by COMMON(1) would lead to separating COMMON(m) from ARBITRARY(m), and give some partial results for this lower bound question.

The result we are after, and believe to be true, is:

Conjecture 1: The m-colour MINIMIZATION problem [which is equivalent to the simulation of MINIMUM(m)] requires $m \log n$ steps on a COMMON(1).

In fact, a weaker conjecture will suffice for our purposes. Consider the m-group MINIMIZATION problem where all groups are of the same size, namely n/m. Call this problem the m-equal-group MINIMIZATION problem.

Conjecture 2: The m-equal group MINIMIZATION problem requires $m \log (m/n)$ steps on a COMMON(1).

Theorem 7

If conjecture 2 holds, then there exists a problem that can be solved in $O(\sqrt{(n/m)})$ steps on an ARBITRARY(m), but requires $\Omega(\sqrt{(n/m)} \log(n/m))$ steps on a COMMON(m). If $m = n^{1-\epsilon}$, ϵ fixed, this implies an $\Omega(\log n)$ separation between COMMON(m) and ARBITRARY(m).

Proof:

The problem we consider is the $\sqrt{nm}$-equal-group MINIMIZATION problem.

Upper bound on ARBITRARY(m):
Partition the problem into m subproblems, each on $\sqrt{(n/m)}$ groups, and devote to each subproblem n/m processors and one shared memory cell from the ARBITRARY(m). All the subproblems can be solved in parallel. The time needed for each subproblem (and therefore for the whole problem) is, by Theorem 6,

$$O\left\{\frac{\sqrt{n/m}\log(n/m)}{\log\sqrt{n/m}}\right\} = O(\sqrt{n/m})$$

Lower bound for COMMON(m):
It follows from Conjecture 2 that a COMMON(1) requires $\Omega\left\{\sqrt{nm}\log(\frac{n}{\sqrt{nm}})\right\}$ steps to solve this problem. Since a COMMON(1) can simulate one step of a COMMON(m) in m steps, a lower bound of $\Omega(\sqrt{n/m}\log\sqrt{n/m}) = \Omega(\sqrt{n/m}\log(n/m))$ follows for a COMMON(m). •

How to prove conjecture 2? We considered two approaches, and below we describe them and the partial results they led to.

The first approach was to consider the structure of an optimal algorithm for the m-equal-group MINIMIZATION problem, and prove the lower bound just for optimal algorithms using the information on their structure. We conjecture (but cannot prove) that there exists an optimal algorithm for the problem that satisfies the following: if at time t a processor in a certain group will write on 1, then so will all other processors in this group that are of lower index. We call an algorithm with this property "restricted". This restriction seems natural, since we are looking for the leftmost 1.

Theorem 8

Any restricted program on a COMMON(1) that solves m-group MINIMIZATION requires at least $\lceil \sum_{j=1}^{m} \log(n_j + 1) \rceil - m + 1$ steps.

For the purposes of the proof, it is necessary to introduce yet another problem. CONSTRAINED m-group MINIMIZATION is m-group MINIMIZATION in which constraints are placed upon the set of allowable inputs (i.e. allowable problem instances). This additional information is known in advance by all processors.

The set OR is used to describe these constraints. Each constraint is specified by an m-tuple $(i_1, \ldots, i_m)$ where $1 \leq i_j \leq n_j$ for $j = 1, \ldots, m$. For notational convenience, we define bits $b_{0,j} = 0$. The semantics associated with the statement $(i_1, \ldots, i_m) \in OR$ is

$$1 = \bigvee_{j=1}^{m} \bigvee_{i=0}^{i_j} b_{i,j} .$$

Thus among the first i_1 processors in group 1, the first i_2 processors in group 2, . . . , and the first i_m processors in group m, there is at least one whose private bit has value 1.

Lemma 1

A CONSTRAINED m-group MINIMIZATION problem with r different possible solutions requires at least $\lceil \log_2 r \rceil - m + 1$ steps on a restricted COMMON(1).

Proof:

By induction on r and m.

If $r = 1$, then $\lceil \log_2 r \rceil - m + 1 = 1 - m \leq 0$ and the claim is trivially true. If $m = 1$, then the problem is equivalent to 1-color MINIMIZATION for $\min\{ i \mid (i) \in OR \}$ processors. The claim follows from Theorem 2. Therefore let $m, r > 1$ and assume that the claim is true when the problem has less than m groups or less than r different possible solutions.

Because the answer is not yet known, every algorithm to solve the problem must perform at least one step. Let us examine the first step of an *optimal* algorithm which solves this problem. Since the algorithm is optimal, the number of solutions for each subproblem remaining after this step has been performed must be less than r, independent of its outcome.

First, suppose that no processor writes on 0. The monotone nature of the constraints implies that the input in which all private bits have value 1 is always allowed. Therefore all processors which write on 1 must write a common value.

For $j = 1, \ldots, m$, let $k_j = \max\{ k \mid P_{k,j}$ writes on $1 \}$; $k = 0$ if no such $P_{k,j}$ exists. By assumption each of the processors $P_{i,j}$, where $1 \leq j \leq m$ and $1 \leq i \leq k_j$, writes the aforementioned common value when its private bit $b_{i,j}$ is 1. If the write does not occur, then the private bits of the processors which write on 1 must all be 0. The remaining problem can then be viewed as m-group MINIMIZATION for processors $P'_{i,j}$ $(1 \leq j \leq m, 1 \leq i \leq n_j - k_j)$ where $P'_{i,j} = P_{k_j+i,j}$ with constraints specified by $OR' = \{ (i'_1, \ldots, i'_m) \mid (i_1, \ldots, i_m) \in OR$ and $i'_j = \max(0, i_j - k_j) \}$. However, if the write occurs, then the remaining problem can be viewed as m-group MINIMIZATION with one additional constraint, namely $(k_1, \ldots, k_m)$.

Notice that each solution for the original problem must correspond to a solution for at least one of these subproblems. Therefore, the number of solutions for at least one of the two subproblems must be $\lceil r/2 \rceil$. By the induction hypothesis, that subproblem requires at least $\lceil \log_2(r/2) \rceil - m + 1$ steps on a res-

tricted COMMON(1). Therefore the original problem requires $\lceil \log_2 r \rceil - m + 1$ steps.

Now suppose there is some processor $P_{k,l}$ that writes on 0. We focus attention on the two subproblems obtained by fixing $b_{k,l} = 0$ and $b_{k,l} = 1$, respectively. Each solution of the original problem is a solution of at least one of these two subproblems. Thus the number of solutions for at least one of the subproblems must be at least $\lceil r/2 \rceil$.

In particular, if $k > 1$, then the number of solutions for the 0 subproblem is at least $\lceil r/2 \rceil$. To see this, consider the two cases where $a_{k,l} = 0$ and $a_{k,l} = 1$. If $a_{k,l} = 0$, every solution to the 1 subproblem is also a solution to the 0 subproblem. If $a_{k,l} = 1$, then to every solution of the 1 subproblem corresponds at least one solution of the 0 subproblem, namely one in which $a_{k-1,l} = 1$. This is because any constraint

$$\bigvee_{j=1}^{m} \bigvee_{i=0}^{i_j} b_{i,j}$$

that is satisfied by having $b_{k,l} = 1$ is also satisfied by having $b_{k-1,l} = 1$.

If the number of solutions for the 0 subproblem is at least $\lceil r/2 \rceil$, then an adversary would force processor $P_{k,l}$ to write (and hence determine the outcome of the step) by setting $b_{k,l} = 0$. The 0 subproblem can be viewed as a CONSTRAINED m-group MINIMIZATION with 1 fewer processor. Formally, the processors are $P'_{i,j}$ $(1 \le j \le m,\ 1 \le i \le n'_j)$ where

$$n'_j = \begin{cases} n_l - 1 & \text{if } j = l \\ n_j & \text{otherwise} \end{cases}$$

and

$$P'_{i,j} = \begin{cases} P_{i+1,l} & \text{if } j = l \text{ and } i \ge k \\ P_{i,j} & \text{otherwise} \end{cases}$$

The constraints are specified by $\{ (i_1, \ldots, i'_l, \ldots, i_m) \mid (i_1, \ldots, i_l, \ldots, i_m) \in OR \}$ where

$$i'_l = \begin{cases} i_l - 1 & \text{if } i_l \ge k \\ i_l & \text{if } i_l < k \end{cases}$$

Then, as above, the lower bound follows from the induction hypothesis.

Otherwise, $k = 1$ and the number of solutions for the 1 subproblem is at least $\lceil r/2 \rceil$. The 1 subproblem can be viewed as CONSTRAINED $(m-1)$-group MINIMIZATION where the processors are $P_{i,j}$ $(j = 1, \ldots, l-1, l+1, \ldots, m, 1 \le i \le n_j)$ and the constraints are specified by $\{ (i_1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_m) \mid (i_1, \ldots, i_{l-1}, 0, i_{l+1}, \ldots, i_m) \in OR \}$. By the induction hypothesis, the 1 subproblem requires $\lceil \log_2 \lceil r/2 \rceil \rceil - (m-1) + 1 = \lceil \log_2 r \rceil - m + 1$ steps on COMMON(1). Hence the original problem also requires at least this many steps. ∎

Since m-group MINIMIZATION is a CONSTRAINED m-group MINIMIZATION with an empty set of constraints and it has $\prod_{j=1}^{m} (n_j + 1)$ solutions, Theorem 8 follows directly from Lemma 1. Applying this result to m-equal-group MINIMIZATION ($n_j = n/m$), we see:

Corollary: Any restricted COMMON(1) algorithm for the m-equal-group MINIMIZATION problem requires $\Omega(m \log(n/m))$ steps, if $m < n/4$.

The second approach towards proving conjecture 2 was to generalize it. Consider any function $f: \{0,1\}^n \to \{0,1\}^n$. A PRAM is said to compute f if before the computation each processor P_i has a private bit c_i, after the computation P_i has an "answer" but a_i, and $f(c = c_1, c_2 \cdots c_n) = a_1 a_2 \cdots a_n$ for all vectors $c \in \{0,1\}^n$. Clearly, the m-group MINIMIZATION problem is defined in this way. Let $R(f)$ be the range of the function f. Then the following implies conjecture 2.

Conjecture 3: Any COMMON(1) algorithm that computes a function $f: \{0,1\}^n \to \{0,1\}^n$ requires $\Omega(\log |R(f)|)$ steps.

This seems like an "information-theoretic" lower bound, and, indeed, would trivially follow if the unique shared memory cell could contain only a Boolean value. We believe it holds in the general case, when the cell can hold arbitrary values, although this ability allows non-binary branching. The best we can prove is:

Theorem 9

Any COMMON(1) algorithm that computes a function $f:\{0,1\}^n \rightarrow \{0,1\}^n$ requires $\Omega\left(\frac{\log|R(f)|}{\log\log|R(f)|}\right)$ steps. ■

Unfortunately, this theorem is not strong enough to prove the separation we seek. The proof of this theorem is omitted due to space considerations, and will appear in the final version of this paper.

References

[CSV] A.K. Chandra, L.J. Stockmeyer and U. Vishkin, *Complexity Theory of Unbounded Fan-in Parallelism*, Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982, pages 1-13.

[CD] S.A. Cook and C. Dwork, *Bounds on the Time for Parallel RAM's to Compute Simple Functions*, Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pages 231-233.

[FW] S. Fortune and J. Wyllie, *Parallelism in Random Access Machines*, Proc. 10th Annual ACM Symposium on Theory of Computing, 1978, pages 114-118.

[G] L. Goldschlager, *A Unified Approach to Models of Synchronous Parallel Machines*, JACM, volume 29, number 4, 1982, pages 1073-1086.

[K] Ludek Kucera, *Parallel Computation and Conflicts in Memory Access*, Information Processing Letters, volume 14, number 2, 1982, pages 93-96. January 1963, pages 589-591.

[SV] Yossi Shiloach and Uzi Vishkin, *An O(log n) Parallel Connectivity Algorithm*, Journal of Algorithms, volume 3, 1982, pages 57-67.

[S] Marc Snir, *On Parallel Searching*, Department of Computer Science, The Hebrew University of Jerusalem, Research Report 83-21, June 1983.

[Sz] J.T. Schwartz, *Ultracomputers*, ACM Trans. on Programming Languages and Systems, vol. 2 (1980), pages 484-521.

[V] Uzi Vishkin, *Implementation of Simultaneous Memory Address Access In Models That Forbid It*, Tech. Rept. 210, Israel Institute of Technology, July 1981. (to appear in J. Alg.)

[VW] Uzi Vishkin and Avi Wigderson, *Trade-offs Between Depth and Width in Parallel Computation*, to appear in SIAM J. Computing.