A Small-Domain Lower Bound For Parallel Maximum Computation

Prabhakar Ragde*

0. Abstract. Recent work [BJKTV] has shown that parallel algorithms that are sensitive to the size of the input domain can improve on more general parallel algorithms. The cited paper demonstrates an $O(\log\log\log s)$-step algorithm on an n-processor CRCW PRAM for finding the prefix-maxima of n numbers in the range $[1..s]$. This paper proves a lower bound demonstrating that no algorithm is asymptotically faster as a function of s, by showing that for $s = 2^{2^{\Omega(\log n \log\log n)}}$ the upper bound is tight.

1. Introduction. Few techniques exist to show general lower bounds for parallel computation. One of the most useful ones has been the application of powerful methods from Ramsey theory. Intuitively, a Ramsey-like theorem states that in some large and possibly complex universe, there exists a subuniverse with some simpler or more regular structure. To prove a lower bound on the complexity of a problem, it is often possible to take an arbitrary program which may exhibit complex behaviour when considered over all inputs, and apply Ramsey theory to show that there exists a subdomain of inputs on which the program behaves in very simple ways. In effect, the program is reduced to operating in a structured fashion, or with a restricted set of operations. Ad-hoc techniques can then be used to prove a lower bound on the running time of the program on this subdomain. In this fashion, the following lower bounds have been proved:

- An $\Omega(\sqrt{\log n})$ lower bound on searching in a sorted table of size n with an EREW PRAM [S];
- An $\Omega(\sqrt{\log n})$ lower bound on sorting n items with an n-processor PRIORITY CRCW PRAM [MW];
- An $\Omega(\sqrt{\log n})$ lower bound on deciding element distinctness of n items with an n-processor COMMON CRCW PRAM [RSSW]. This was improved in [Bo] to the optimal result $\Omega(\log n/\log\log n)$;
- An optimal $\Omega(\log\log n)$ lower bound on merging two sequences of length n with an $n\log^{O(1)} n$-processor PRIORITY CRCW PRAM [BBGSU]
- An $\Omega(\log\log\log n)$ lower bound on simulating a n-processor ARBITRARY PRAM on an n-processor COLLISION PRAM [GR]. This was improved in [C] to $\Omega(\log\log n)$.

One of the drawbacks of these uses of Ramsey theory is the fact that, in order to show that the subdomain exists, the domain size must be a very rapidly growing function of n. The possibility thus exists that, if inputs are taken from the domain $[1..s]$, where s may be polynomial or even singly or double exponential in n, then algorithms may exist which

* Author's address: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. Electronic mail: plragde@maytag.waterloo.edu.

beat these lower bounds. As an analogy, consider the case of sequential sorting, which has an $\Omega(n \log n)$ lower bound on the RAM model. Radix sort will, for suitably restricted domains, give an $O(n)$ algorithm.

The challenge, then, is to either reduce the domain size required in the lower bounds, or to produce algorithms with better running times on moderate sized domains. [BH] improves both the asymptotic result and the domain size for the sorting bound mentioned above by proving an $\Omega(\log n / \log\log n)$ lower bound on computing parity with a PRIORITY CRCW PRAM. This implies the same lower bound for sorting with domain size 2. [E] has obtained the same lower bound as [Bo] for element distinctness but with a domain size that is doubly exponential in n.

When improved algorithms can be found, new impetus is given to the lower bound effort. This was the case when [BJKTV] reported two interesting algorithms, for the problems of merging and maximum mentioned above. In the case of merging two sorted lists drawn from the domain $[1..s]$, they give an $O(\log\log\log s)$ algorithm on a CREW PRAM. (This is remarkable, given that even computing the OR of n bits on a CREW PRAM requires $\Omega(\log n)$ time.) In the case of maximum finding, they are able to find the maximum of n numbers from domain $[1..s]$ in time $O(\log\log\log s)$ on a PRIORITY CRCW PRAM; in fact, the prefix maxima can be computed in this time bound.

In this paper, we show a value of s for which $\Omega(\log\log\log s)$ time is required to compute the maximum of n numbers on an n-processor PRIORITY CRCW PRAM, thus demonstrating that the domain-sensitive result cannot be improved without further restriction on s. This represents a modest beginning to the search for lower-bound techniques that work on problems defined over small domains.

2. The Upper Bound. For completeness, we briefly sketch the domain-sensitive upper bound for finding the maximum which is claimed (but not elaborated upon) in [BJKTV]. First, we give a fast domain-sensitive algorithm that works with more than n processors – in fact, with a number of processors that is also a function of the domain size.

Theorem 1. *An $(n \log s)$-processor CRCW PRAM can find the maximum of n numbers in the domain $[1..s]$ in constant time.*

Proof: For ease of presentation, consider s to be a power of 2. The input numbers $x_1, x_2, \ldots x_n$ are $\log s$ bits long; let x_i^k be the first (high-order) k bits of x_i. We label the processors $P_{i,j}$, where i ranges from 1 to n and j from 1 to $\log s$. We label $s-1$ locations in memory A_α, where α is a string of bits of length at most $\log s$. Location A_α will be used to indicate whether there exists an input x_i such that α followed by the bit 1 is x_i^k for $k = |\alpha| + 1$. Finally, we label n locations in memory B_i, where i varies between 1 and n. Location B_i will be used to indicate whether there is an input with higher value than x_i.

In the first step, each processor $P_{i,j}$ reads x_i. Then processor $P_{i,j}$ writes 1 to $A_{x_i^{j-1}}$ if the jth bit of x_i is 1. This sets the A's as stated above. In the second step, $P_{i,j}$ reads $A_{x_i^{j-1}}$ if the jth bit of x_i is 0. There is a input of value greater than x_i if and only if no processor $P_{i,j}$ read 1 at this step, since any greater value will have some common prefix with x_i and then have a 1 where x_i has a 0. Consequently, B_i can be set by having any

processor $P_{i,j}$ that read 1 in the second step write the value 1 to B_i. The maximum input value x_i is the only value for which $B_i = 0$. ■

Next, we need a fast algorithm that is not domain-sensitive, but uses more than n processors.

Theorem 2 [K]. *An $n^2/2$-processor CRCW PRAM can find the maximum of n numbers in $O(1)$ time.*

Proof: Let the processors be labelled $P_{i,j}$, for $1 \leq i < j \leq n$. Processor $P_{i,j}$ reads the cells containing the ith number and the jth number, and writes 0 over whichever one is smaller. The only number not overwritten by 0 is the maximum. ■

We use this second algorithm to design a fast algorithm that is not domain-sensitive, but uses only n processors.

Theorem 3 [SV]. *An n-processor CRCW PRAM can find the maximum of n numbers (from an unrestricted domain) in $O(\log\log n)$ time.*

Proof: The algorithm proceeds in phases, starting with phase 1. At the beginning of phase i, there are $n/2^{2^{i-1}-1}$ candidates remaining that could be the maximum. The candidates are divided into $n/2^{2^i-1}$ groups of size $2^{2^{i-1}}$. Each group is assigned a number of processors equal to half the square of its size (that is, each group gets 2^{2^i-1} processors). The maximum of each group is found in constant time using the algorithm of Theorem 2. Each group contributes this one candidate to the next phase; this leaves the claimed number of candidates at the beginning of phase $i+1$, as required. It is easy to see that $O(\log\log n)$ phases are needed. ■

Finally, we can describe the domain-sensitive algorithm that uses only n processors.

Theorem 4. *An n-processor CRCW PRAM can find the maximum of n numbers in the range $[1..s]$ in $O(\log\log\log s)$ time.*

Proof: The numbers can be divided into groups of size $\log s$, and $\log s$ processors assigned to each group. The maximum of each group can be found in $O(\log\log\log s)$ time using the algorithm of Theorem 3. This leaves $n/\log s$ candidates for the global maximum, and using n processors and the algorithm of Theorem 1, the maximum can be found in constant time. ■

Note that the algorithm claimed in [BJKTV] is actually more general than this, as it finds prefix-maxima.

3. The Lower Bound. The lower bound given here follows the general outlines of other PRAM lower bounds ([FMW],[FRW],[GR],[RSSW]). The input to a PRAM will be an n-tuple of positive integers $(x_1, x_2, \ldots, x_n)$, where x_i is drawn from the domain $[1..s]$ and is initially stored in the local memory of processor P_i. (Since memory is unbounded, this is equivalent to the situation where the input variables are stored in shared memory, one to a cell.) The output of the PRAM will be in the local memory of processor P_1 at time T. One step of a PRAM consists of a parallel write followed by a parallel read.

It is useful to slightly modify the PRIORITY PRAM. We disallow overwriting of memory – that is, a cell may be written into only once. To compensate, we allow each processor to simultaneously read $t-1$ cells at step t, providing that those cells, if they were written into at all, were written into at steps $1, 2, \ldots, t-1$ respectively. One can prove easily (see [FMW]) that for infinite memory, this does not decrease the power of the PRAM. This is a technical convenience that makes the proof slightly easier.

Theorem 5. *Any* PRIORITY *CRCW PRAM requires* $\Omega(\log\log\log s)$ *steps to find the maximum of* n *numbers in the domain* $[1..s]$, *when* $s = 2^{2^{\Omega(\log n \log\log n)}}$.

Proof:

Given a PRIORITY CRCW PRAM algorithm that claims to solve the maximum problem, we proceed to construct a set of "allowable" inputs for each step. This set is chosen to restrict the behaviour of the machine so that its state of knowledge can be easily described. As long as the set of allowable inputs for step t is sufficiently rich, we can show (based on our characterization of the state of knowledge of the machine) that there exists an allowable input on which the machine cannot answer correctly after t steps. In order to fully describe the set of allowable inputs after step t, we will require some additional sets, which are described below.

- A set $\mathcal{U}_t$ of *free* variables. These are variables to which no fixed value has been assigned. We denote the total number of variables in $\mathcal{U}_t$ as v_t. Intuitively, after t steps the algorithm has succeeded in determining only that the maximum is one of the free variables. In other words, the free variables are the candidates that the algorithm has to work with (whether or not the algorithm is explicitly structured in this fashion).
- A set S_t of positive integers. In any allowable input, the values given to the free variables will have distinct values chosen from S_t.
- A set $\mathcal{M}_t$ of *fixed* variables. Any variable that is not free will be fixed. A fixed variable has the same value in any allowable input. It is set to some value that is smaller than any value in S_t. Intuitively, either the algorithm has determined that the variables in $\mathcal{M}_t$ are not the maximum, or we as adversary have given that information away.

Any input for which all the variables in $\mathcal{M}_t$ have their assigned fixed values and all the variables in $\mathcal{U}_t$ have values in S_t is an allowable input for step t. We can now state two inductive hypotheses which will be shown to hold by construction.

Hypothesis 1: The state of each processor and each memory cell at each step up to and including step t, considered over the domain of allowable inputs for step t, is a function of at most one free variable. For a given processor or memory cell, this variable, if it exists, is the same over all allowable inputs. We say that the processor or memory cell *knows* that variable.

Because of Hypothesis 1, the choice of which cell processor P_i reads at a given step t (again, considered over the domain of allowable inputs for step t) is also a function of the one free variable that P_i knows. This is called the *read access function* of P_i. A read access function should be considered as a function of some variable z that can take on values from S_t; a processor uses the read access function by substituting as an argument the value of

the free variable it knows. Similarly, the write access function of P_i (the choice of where the processor writes) is a function of that one free variable.

Hypothesis 2: For every step $t' \leq t$ and over all allowable inputs, a processor either does not write at step t' or always writes. Any read or write access function at step t', considered as a function over S_t, is either constant or 1–1; any two such functions used before or at step t are either identical, or have disjoint ranges.

Given these hypotheses, if at any time there are at least two free variables in $\mathcal{U}_t$ and at least $v_t + 1$ values in S_t, then the algorithm cannot answer after step t. This is because processor 1 cannot distinguish two cases: the case when the variable it knows is set to the second highest value in S_t and all other free variables have lower values and the case when one other free variable is set to the highest value in S_t. We must attempt to carry out the construction so as to keep the set of free variables and the domain size as large as possible. When we can no longer maintain two free variables, the construction will stop, yielding a lower bound on T; we can then extract an initial value for s which allows the construction to continue for that many steps.

The proof proceeds by induction on t. For the base case, we set $S_0 = \{1, 2, \ldots s\}$, $\mathcal{U}_0 = \{x_1, \ldots, x_n\}$, $\mathcal{M}_o = \phi$, and $v_0 = n$; the hypotheses are trivially satisfied. For the inductive step, suppose the situation as described above holds through step t. We describe how to maintain the inductive hypotheses by defining $\mathcal{U}_{t+1}$, $\mathcal{M}_{t+1}$, and $\mathcal{S}_{t+1}$. Initially, let $S_{t+1} = S_t$; we will change S_{t+1} by removing values, based on what the PRAM algorithm does at step $t + 1$.

We will find it useful to borrow a technique from [GR]. Lemmas 1 and 2 were used there to restrict the manner in which processors may communicate with each other by restricting the domain S_{t+1}. The importance of the lemmas lies in the relatively small reduction in domain size. Similar lemmas with greater reduction were given in [FMW].

Lemma 1. *If $f_1, f_2, \ldots, f_k$ are functions with common domain S, where $|S| = k!q^{k+1}$, then there exists a subdomain S' of size q such that when $f_1, \ldots, f_k$ are restricted to S', each function is either constant or 1–1.*

Proof: A theorem of Erdős and Rado ([ER]) states that in any family of at least $\ell!k^{\ell+1}$ (not necessarily different) sets of size at most ℓ, there is a *sunflower* formed by k sets; that is, a collection of k sets whose pairwise intersection is equal to its intersection. With each element $e \in S$, associate the set of ordered pairs $A_e = \{(r, f) | f \in \{f_1, \ldots, f_k\}, f(e) = r\}$. There are $k!q^{k+1}$ such sets, and so there exists a sunflower of size q among them.

Let the elements corresponding to the sets in the sunflower be $e_1, e_2, \ldots e_q$. If we set $S' = \{e_1, e_2, \ldots e_q\}$, the desired property is obtained. Consider an ordered pair (r, f_i) in the sunflower. If this pair is in the center of the sunflower (that is, in all the sets A_e, $e \in S'$), it follows that $f_i(e) = r$ for all $e \in S'$, and f_i is constant over S'. If (r, f_i) is in a petal (that is, it is in the set A_{e_j} and in no other set), then $f_i(e_j) = r$ but for no other e_k does $f_i(e_k) = r$. Since there was nothing special about our choice of r, we conclude that f_i is 1–1 over S'. ■

Let us define the value of a write function to be 0 if the processor does not wish to write, and apply Lemma 1 to the set of all read and write access functions used at

step $t+1$. This further restricts S_{t+1}. Remember that each processor uses t read access functions and one write access function at step $t+1$; this is a total of $k = n(t+1)$ functions. We overestimate the domain reduction necessitated by Lemma 1 by assuming an initial domain size of $(kq)^{k+1}$ reduced to q. Once we have applied Lemma 1 to a given f, if it is 1–1, then there is at most one value in S_{t+1} on which it does not write. We can remove that value from S_{t+1}, thereby ensuring that processors using f always write. At this point, then, the size of S_{t+1} is $\frac{s_t^{1/(n(t+1)-1)}}{n(t+1)} - n(t+1)$.

Lemma 2. *If f, g are two 1–1 functions with common domain S, $|S| = 4q$, then there exists a subdomain S' of size q such that f and g, restricted to S', are either identical or have disjoint ranges.*

Proof: If f, g have the same value for q elements in S, then let S' be those elements. As a result, f and g are identical when restricted to S'. Otherwise, remove all such elements from S. Form a graph whose nodes are the elements of S; there is an edge between a and b if $f(a) = g(b)$. This graph consists of disjoint cycles and thus is 3-colourable; choose any independent set of size q and let S' be this set. It follows that f and g have disjoint ranges when restricted to S'. ■

We apply Lemma 2 to all pairs consisting of one read or write access function used before step $t+1$ and one function used at step $t+1$. There are $n(t+1)(t+2)/2$ functions in the first category and $n(t+1)$ functions in the second category; each application reduces the size of S_{t+1} by a factor of 4. This ensures that Hypothesis 2 holds after step $t+1$.

It remains to ensure that Hypothesis 1 holds after step $t+1$. There are two ways in which it can be violated: if a cell that knows one free variable is written into by a processor knowing another free variable, the state of that cell after step $t+1$ may be a function of two free variables. Also, if a processor knowing one free variable reads a cell knowing another free variable, the state of that processor may be a function of two free variables.

Let us construct a graph whose nodes are the free variables; there is an edge between x_i and x_j if a processor knowing x_j learns something about x_i (in the sense described above). Each processor can contribute at most $t+1$ edges to this graph, since it reads at most t cells and writes into at most one cell at step $t+1$. Turán's theorem [Be] states that in any graph with v vertices and e edges, there exists an independent set of size $\frac{v^2}{v+2e}$. Hence in our graph there is an independent set of size $\frac{v_t^2}{v_t + 2n(t+1)} \geq \frac{v_t^2}{3n(t+1)}$.

If there are j variables not in this independent set, then we choose the j smallest values of S_{t+1}, fix the variables to those values in an arbitrary fashion, and remove those values from S_{t+1}, thus ensuring Hypothesis 1. All three inductive hypotheses are now satisfied. The resulting recurrence equations (slightly simplified) are:

$$v_{t+1} \geq \frac{v_t^2}{3n(t+1)}$$

$$s_{t+1} \geq \frac{s_t^{1/n(t+1)+1}}{n(t+1)2^{(t+1)^2(t+2)n^2}} - n(t+1)$$

It is now not difficult to obtain the following inequalities by estimation, and to prove them using induction on t (for n sufficiently large).

$$v_t \geq \frac{n}{2^{2^{3t}}}$$
$$s_t \geq \frac{s^{n^{-3t}}}{2^{n^{2t}}}$$

Since the process can continue as long as there are at least two free variables, the bound on v_t ensures $T \geq \frac{1}{3}\log\log n$. If the domain size after step T is to be at least n, then s need only be as large as $2^{n^{4\log\log n}}$. A simple calculation shows that $T \geq \frac{1}{3}\log\log\log s$ for n sufficiently large. ■

4. Future Research. The lower bound proved above shows that the results of [BJKTV] cannot be improved when expressed solely in terms of the domain size. If the range of the domain is further restricted, however, improvements are possible. [FRW] gave an technique which could be applied to find the maximum of n integers from the range $[1..n^k]$ in $O(k)$ time on a COMMON CRCW PRAM; this shows that $t = \Theta(\log\log\log s)$ does not give the correct tradeoff between domain size and computation time for all values of s. More work is needed to discover upper and lower bounds for parallel maximum computation that are tight for all s.

[BJKTV] gives an algorithm for merging sorted lists of length n from the domain $[1..2n]$ in time $\alpha(n)$, where $\alpha(n)$ is the very slowly growing functional inverse of Ackermann's function. The technique presented here does not seem to be powerful enough to deal with the problem of merging, since fixing values very quickly constrains the adversary. The technique in [E] allows processors to learn more than one variable, but is only good for moderately large (doubly exponential in n) domains, and its applicability to other problems remains unclear.

5. References.

[Be] C. Berge, *Graphs and Hypergraphs*. North-Holland, 1973.

[Bo] R. Boppana, *Optimal Separations Between Concurrent-Write Parallel Machines*, Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 320-326.

[BH] P. Beame and J. Hastad, *Optimal Bounds for Decision Problems on the CRCW PRAM*, Proc. 19^{th} ACM Symposium on Theory of Computing, 1987, pp. 83-93.

[BBGSU] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, "Highly parallelizable problems", Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 309-319.

[BJKTV] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella, and U. Vishkin, "Some triply-logarithmic parallel algorithms", Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 871-881.

[C] S. Chaudhuri, "Lower Bounds for Parallel Computation", Ph.D thesis, Rutgers University, 1991.

[CDR] S.A. Cook, C. Dwork, and R. Reischuk, *Upper and Lower Bounds for Parallel Random Access Machines without Simultaneous Writes*, SIAM J. Computing, vol. 15, no. 1, 1986, pp. 87-97.

[E] J. Edmonds, "Lower Bounds with Smaller Domain Size on Concurrent Write Parallel Machines", Proc. 6th Annual IEEE Conference on Structure in Complexity Theory, 1991.

[ER] P. Erdős and R. Rado, *Intersection Theorems for Systems of Sets*, J. London Math. Soc. , vol. 35, 1960, pp. 85-90.

[GR] V. Grolmusz and P. Ragde, *Incomparability in Parallel Computation (Preliminary Version)*, Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 89-98.

[FMW] F.E. Fich, F. Meyer auf der Heide, and A. Wigderson, *Lower Bounds for Parallel Random-Access Machines with Unbounded Shared Memory*, Advances In Computing Research, 1986.

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

[K] L. Kucera, *Parallel computation and conflicts in memory access*, Information Processing Letters, vol. 14, 1982, pp.93-96.

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

[R] F.P. Ramsey, *On A Problem of Formal Logic*, Proc. London Math. Soc., ser. 2, vol. 30, 1930, pp. 264-286.

[RSSW] P.L. Ragde, A. Szemerédi, W. Steiger, and A. Wigderson, *The Parallel Complexity of Element Distinctness is* $\Omega(\sqrt{\log n})$, SIAM Journal of Discrete Mathematics 1, 1988, pp. 399-410.

[S] M. Snir, *On Parallel Searching*, SIAM J. Computing, vol. 14, no. 2, 1985, pp. 688-709.

[SV1] Y. Shiloach and U. Vishkin, *Finding the Maximum, Merging, and Sorting on Parallel Models of Computation*, J. Algorithms, vol. 2, 1981, pp. 88-102.