Triply-Logarithmic Parallel Upper and Lower Bounds for Minimum and Range Minima over Small Domains *

Omer Berkman † *Yossi Matias* ‡ *Prabhakar Ragde* §

Abstract

We consider the problem of computing the minimum of n values, and several well-known generalizations (prefix minima, range minima, and all-nearest-smaller-values, or ANSV) for input elements drawn from the integer domain $[1..s]$ where $s \geq n$. In this paper we give simple and efficient algorithms for all of the above problems. These algorithms all take $O(\log\log\log s)$ time using an optimal number of processors and $O(ns^{\epsilon})$ space (for constant $\epsilon < 1$) on the COMMON CRCW PRAM. The best known upper bounds for the range minima and ANSV problems were previously $O(\log\log n)$ (using algorithms for unbounded domains). For the prefix minima and for the minimum problems, the improvement is with regard to the model of computation.
We also prove a lower bound of $\Omega(\log\log n)$ for domain size $s = 2^{2^{\Omega(\log n \log\log n)}}$. Since, for s at the lower end of this range, $\log\log n = \Omega(\log\log\log s)$, this demonstrates that any algorithm running in $o(\log\log\log s)$ time must restrict the range of s on which it works.

*A preliminary version of this paper was presented at the *Third Workshop on Algorithms and Data Structures (WADS)*, Montréal, Canada, August 1993.

†Tel Aviv Academic College 4 Antokolsky St., Tel Aviv 64044, Israel. omer@math.tau.ac.il Part of this work was carried out at the Dept. of Computer Science, King's College London, The Strand, London WC2R 2LS, England.

‡AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974-0636, USA. matias@research.att.com.

§Dep. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. plragde@plg.uwaterloo.ca.

1 Introduction

Let $A = (a_1, \ldots, a_n)$ be an array of input elements. Denote by $MIN(i,j)$ the minimum over $a_i, \ldots, a_j$. We consider the following problems:

- The *minimum* problem: find $MIN(1,n)$.
- The *prefix minima* problem: find $MIN(1,i)$ for all i, $1 \le i \le n$.
- The *range minima* problem: build a data structure that will permit a constant-time answer to any query $MIN(i,j)$ for any $1 \le i < j \le n$.
- The *all nearest smaller values (*ANSV*)* problem: find for all i, $1 \le i \le n$, the maximum j, $j < i$, such that $a_j < a_i$ (the "left match" of a_i) and the minimum k, $k > i$, such that $a_k < a_i$ (the "right match" of a_i).

Clearly, an algorithm for range-minima also solves minimum and prefix-minima, and a lower bound for minimum also applies to the other problems in the list.

In this paper we consider the above problems when the elements of A are drawn from the integer domain $[1..s]$ where $s \ge n$. We show:

Theorem 1 (upper bounds) *Each of the above problems can be solved on the* COMMON CRCW PRAM *in $O(\log\log\log s)$ time using $n/\log\log\log s$ processors and $O(ns^\epsilon)$ space (for constant $\epsilon < 1$).*

Theorem 2 (lower bounds) *Any n-processor* PRIORITY CRCW PRAM *algorithm for computing the minimum, and hence any algorithm for the other three problems, takes $\Omega(\log\log n)$ time for any s, $s \ge 2^{2^{\Omega(\log n \log\log n)}}$.*

Corollary 3 *Any n-processor* PRIORITY CRCW PRAM *algorithm for computing the minimum, and hence any algorithm for the other three problems, cannot run in $o(\log\log\log s)$ time for all values of s.*

1.1 The model of computation

The model of parallel computation used in this paper is the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). See [12, 23, 24, 31, 38] for introductions and surveys of results concerning PRAM. The CRCW PRAM model employs synchronous processors, all having access to a shared memory with concurrent access permitted. There are several variants of the CRCW PRAM regarding the conflict resolution rule in case of a concurrent writing. In the COMMON model, several processors may attempt to write simultaneously at the same location only if they write the *same* value; COMMON thus forbids write conflicts. Following [16, 29], Boppana [10] gave a lower bound of $\Omega(\log n/\log\log n)$ for computing the *Element Distinctness* problem on an

n-processor COMMON. This problem can be solved in constant time on models that allow write conflicts. Such models include: (i) TOLERANT, where if two or more processors attempt to write to the same cell in a given step then the content of that cell does not change; (ii) COLLISION, where a concurrent write results in a special "collision" symbol appearing in the target cell; (iii) ARBITRARY (stronger than the previous two), in which a concurrent write results in one arbitrary processor succeeding, among those wishing to write; and (iii) the yet stronger PRIORITY in which a write conflict is resolved by having the processor with highest priority succeed. The results of [16, 29, 10] indicate that algorithms running on PRIORITY or ARBITRARY might not be transferable to COMMON without a significant slowdown or loss of efficiency.

A parallel algorithm is said to be *optimal* if its time-processor product is (asymptotically) equal to the lower bound on the time complexity of any sequential algorithm for the problem. A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible.

1.2 Related Work

We review below previous and related results for the four problems considered in this paper.

Sequential algorithms Gabow, Bentley, and Tarjan [17] gave a linear-time preprocessing algorithm for range minima that results in constant-time query retrieval. The ANSV problem has a simple linear-time algorithm using a stack: Push a_1 to the stack. For $2 \leq i \leq n$: as long as a_i is smaller than the element at the top of the stack, pop an element from the stack and set its right match to be a_i. Finally push a_i to the stack and set its left match to be the element at the top of the stack (unless the stack is empty). This algorithm is mentioned in [8] but not described.

Bounds for unbounded-domain input Using n processors in the parallel comparison tree model, the minimum-finding problem, and hence all four problems have an $\Omega(\log\log n)$ time lower bound [37]. This lower bound is matched for each of the four problems by optimal COMMON CRCW PRAM $O(\log\log n)$ time using linear space algorithms: [35] for minimum, [33] for prefix minima, and [8] for range minima and ANSV.

Input from restricted domains [16] gave an optimal constant-time, linear-space, algorithm on COMMON for finding the minimum for integers in the domain $[1..n^k]$ for a constant k. [5] gives a prefix-minima algorithm on PRIORITY (and thus also a minimum finding algorithm) that runs in $O(\log\log\log s)$ time and $O(n)$ operations, using $O(ns^\epsilon)$ space for input restricted to integers in the domain $[1..s]$ where $s \geq n$. For the case $s = n$, [5] gives a PRIORITY algorithm that takes $O(\log^* n)$ time and $O(n)$ operations, using $O(n)$ space.

Randomized algorithms Reischuk [32] gave a randomized algorithm for the minimum problem that takes constant time with high probability, using $O(n)$ space, on an n-processor ARBITRARY.

Using a parallel hashing algorithm [27, 19, 26], the integer-prefix-minima algorithm of [5] can be implemented using only linear space; the time then increases by a factor of $O(\log^* n)$ with high probability (but the number of operations remains linear with high probability). Recently, a randomized algorithm for the range-minima problem with unrestricted input (and hence for the prefix-minima and the minimum problems) was given by [7]. Its running time is $O(\log^* n)$ with high probability, using $O(n)$ space on an $(n/\log^* n)$-processor TOLERANT. Comparison-based randomized algorithms for the ANSV problem with unrestricted input cannot do better than $\Omega(\log\log n)$ expected time, as implied by the $\Omega(\log\log n)$ expected time lower bound for merging [18] and the $o(\log\log n)$ reduction of merging to the ANSV problem [8].

Lower bounds The following lower bounds have been proved using Ramsey-theoretic arguments. [36] gave an $\Omega(\sqrt{\log n})$ lower bound on searching in a sorted table of size n with an EREW PRAM. An $\Omega(\sqrt{\log n})$ lower bound on sorting n items with an n-processor PRIORITY CRCW PRAM is given in [28]. This paper also gives an $\Omega(\log\log n)$ lower bound for finding the minimum among n numbers on PRIORITY assuming that the numbers are drawn from a domain of size at least doubly exponential in n. An $\Omega(\sqrt{\log n})$ lower bound on deciding element distinctness of n items with an n-processor COMMON CRCW PRAM is given in [29]. This was improved in [10] to the best-possible result $\Omega(\log n/\log\log n)$. [34] gave a best-possible $\Omega(\log\log n)$ lower bound on merging two sequences of length n with an $n\log^{O(1)} n$-processor PRIORITY CRCW PRAM.

1.3 Discussion

1.3.1 Upper bounds

Our triply-logarithmic time algorithms for the range-minima and ANSV problems should be compared with the doubly-logarithmic time algorithms given in [8]. Those algorithms all take $O(n)$ work on the COMMON. The new algorithms are faster for, say, $s = n^{(\log n)^{(\log\log n)^{O(1)}}}$, for which $\log\log\log s = O(\log\log\log n)$. On the other hand, the new algorithms requite super-linear space.

For the minimum and prefix-minima problems, the new algorithms improve in that they run on COMMON, whereas the previous triply-logarithmic algorithms of [5] assume the (stronger) PRIORITY model.

Applications Our new ANSV algorithm implies an optimal $O(\log\log\log s)$ time, $O(ns^\epsilon)$ space algorithm for triangulating a monotone polygon whose coordinates are taken from the domain $[1..s]$, $s \geq n$. Previous optimal parallel algorithms for triangulating a monotone polygon are those of [8] and [20]. Their running times are $O(\log\log n)$ and $O(\log n)$ using the COMMON CRCW PRAM and the CREW PRAM respectively. Both assume that coordinates have unrestricted domain.

1.3.2 Lower bounds

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 behavior 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, each of the above mentioned lower bounds were proved.

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 doubly exponential in n, then algorithms may exist which beat these lower bounds. As an analogy, consider the case of sequential sorting; 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. [2] 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. [11] has obtained the same lower bound as [10] for element distinctness but with a domain size that is doubly exponential in n.

Our lower bound of Theorem 2 can be interpreted in two ways. First, as reflected in Corollary 3, it implies that any algorithm that takes $o(\log\log\log s)$ time, using an n processor PRIORITY, must assume a restriction on the value of s as a function of n. Second, it extends the $\Omega(\log\log n)$ lower bound for computing the minimum problem, and hence for any of the other problems mentioned above, on an n-processor PRIORITY from the range $s \geq 2^{2^{\Omega(n)}}$ (as shown by [28]) to the range $s \geq 2^{2^{\Omega(\log n \log\log n)}}$.

The domain-sensitive lower bound implied by Theorem 2 above 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.

The rest of this paper is organized as follows. In Section 2 we present constant-time non-optimal algorithms for each of the four problems. In Section 3 we present the optimal algorithms. The lower bound is given in Section 4. Concluding remarks and open problems are given in Section 5.

2 Constant-Time Non-Optimal Algorithms

We begin with a COMMON PRAM algorithm for finding the minimum. This algorithm is then used as a subroutine for an algorithm that solves both the prefix-minima and ANSV problems. Finally, the prefix minima algorithm is used to get an algorithm for the range minima problem.

It is assumed that all input elements are distinct. If this is not the case than we can replace the value of each input element a_i, $i = 1, \ldots, n$, with the value $a_i \cdot n + i$. In addition, s is assumed to be a power of 2. If this is not the case then s can be modified to be the nearest power of 2 greater than s. These modifications do not change the complexity of the algorithms by more than a constant factor.

2.1 Minimum

The following lemma and algorithm demonstrate a basic step which appears (in different forms) in some of the algorithms below.

Lemma 2.1 *Let $A = (a_1, \ldots, a_n)$ be an array of elements drawn from the domain $[1..s]$, $s \geq n$. The algorithm below finds the minimum in A in $O(1)$ time using $n \log s$ processors and $O(s)$ space.*

We first describe the algorithm and then discuss some implementation details.

Step 1 (Data Structure). Build a complete binary tree T_s whose leaves are the numbers $[1..s]$. We assume that the space allocated for the tree is initialized to zero. It will be shown at the end of the algorithm how to get rid of this assumption.

Step 2 (Processor Allocation). Allocate $\log s$ processors to each element a_i, $1 \leq i \leq n$: a processor for each ancestor of the leaf in T_s whose value is a_i.

Step 3 (Marking). Each processor assigned to the ancestor v of a leaf a_i writes '1' in a variable attached to v, for $i = 1, \ldots, n$.

Step 4 (Information Gathering). The $\log s$ processors of each a_i are assigned to the ancestors of the leaf a_i as in Step 3. A processor that is assigned to ancestor v of a_i which is a right sibling reads the variable of the left sibling of v. Element a_i is the minimum in A if and only if none of its processors has read a '1'. The minimum can therefore be found in constant time by simple OR computations.

Implementation. To implement *Steps 2–4*, it is possible to use an algorithm in [22], which for a node v in a complete binary tree and some ℓ, computes the ancestor of v in level ℓ of the tree in constant time. To handle the case that the input is not initialized to zero we add a step between Step 2 and Step 3 that initializes to zero only those locations which are being read in Step 4. The complexity of such a step is the same as that of Step 4.

Lemma 2.1 follows.

Lemma 2.2 *There is an algorithm for finding the minimum that runs in $O(1)$ (more precisely, $O(\frac{1}{\epsilon})$) time using $n \log s$ processors and $O(ns^{\epsilon})$ space for any constant ϵ, $0 < \epsilon < 1$.*

The algorithm that realizes the lemma is based on adding a variant of the radix sort idea, where the most significant bits are handled first, to the algorithm above. Such a variant was given for the PRIORITY algorithm for prefix minima and can be found in the appendix of [5] or in [4]. It uses only the assumptions of COMMON and can thus be adapted to prove the lemma. (More precisely, the variant in [5] solves a problem which is shown to be reducible to the prefix minima problem within our complexity bounds on the COMMON.)

2.2 Prefix Minima and All Nearest Smaller Values

Lemma 2.3 *Let $A = (a_1, \ldots, a_n)$ be an array of elements drawn from the domain $[1..s]$, $s \geq n$. The algorithm below solves both the prefix-minima and the* ANSV *problems with respect to A in $O(1)$ time using $n \log^3 s$ processors and $O(ns^{\epsilon} \log s)$ space for any constant ϵ, $0 < \epsilon < 1$.*

Step 1 (Data Structure). Build a full binary tree T_A whose leaves are the elements of A.

Step 2 (Processor Allocation). Allocate $\log^3 s$ processors to each leaf a_i of T_A: $\log^2 s$ processors for each ancestor of leaf a_i in T_A (note that a_i has $\log n \leq \log s$ ancestors).

Step 3 (Minima Computation). Find the minimum over the leaves of the subtree rooted at each internal node v of T_A, using the algorithm of Lemma 2.2. Given v, let r denote the number of such leaves. Then this step with respect to v uses $r \log s$ processors, which is less than the number of processors allocated to v; it takes $O(1)$ time and uses $O(rs^{\epsilon})$ space. The total space used is $O(ns^{\epsilon} \log s)$.

Step 4 (Prefix- and Suffix-Minima Computation). For each internal node v compute prefix minima and suffix minima over an array $L(v)$ that contains the leaves of v:
Consider a leaf l of (the subtree rooted at) v, and the path of nodes from l to v. Let $LS_v(l)$ be the set of left siblings of the nodes on the path. The leaves in arrays $L(u)$ of nodes u in $LS_v(l)$, together with l itself, represent exactly all the leaves in the prefix of l in $L(v)$. Therefore, the minimum over the prefix of l in $L(v)$ is the minimum over $\{\min(L(u)) : u \in LS_v(l) \cup l\}$. Since $|LS_v(l)| \leq \log n$ we can find this minimum in constant time with $\log^2 n$ processors (out of the $\log^2 s$ processors allocated to l at node u) using the algorithm of [35]. The space used with respect to the leaf l at node u for this $\log n$-size minimum-finding problem is $O(\log n)$. Over all the leaves and all the levels of the tree the space needed is $O(n \log^2 n)$ which is $O(ns^{\epsilon} \log s)$.

Note that the prefix minima computed with respect to the root is actually prefix minima with respect to A. This concludes the computation of prefix minima.
The next steps complete the computation of ANSV.

Step 5 (Find the Nodes whose Subtrees Contain the Matches). Each leaf a_i finds its lowest ancestor that has the left match of a_i among its descendants. Finding the lowest ancestor that has the right match of a_i is similar. For this consider the (at most) $\log n$ nodes which are left siblings of the ancestors of a_i. Among these $\log n$ nodes, we find the lowest node whose minimum is smaller than a_i. This problem can be restated as the problem of finding the leftmost '1' in an array of $\log n$ 0's and 1's and can therefore be done in $O(1)$ time with $\log n$ processors and $O(\log n)$ space using the algorithm of [16]. The parent of this node is the lowest ancestor of a_i that has a_i's left match among its descendants. The overall space used in this step is $O(n \log n)$ which is $O(ns^\epsilon \log s)$.

Step 6 (Merge Left Child's Suffix Minima with Right Child's Prefix Minima). For each node v do the following. Let u and w be the left and right children of v, respectively, and let S_u and P_w denote their respective suffix minima and prefix minima (computed in Step 4 above). We merge S_u (which is a non-decreasing array) with the reverse of array P_w (P_w itself is non-increasing) into an array $A(u, w)$. Denoting by r the number of leaves of the subtree rooted at v, this can be done in $O(1)$ time with $r \log s$ processors (which is the number of processors allocated to v) and $O(rs^\epsilon)$ space, using the integer merging algorithm of [9]. The overall space used is $O(ns^\epsilon \log s)$.

Step 7 (Find Left and Right Matches for all Elements). Let v be the lowest ancestor of a_i that has a_i's left match among its descendants; let u and w be its left and right children, respectively, and let r_1 be the number of leaves in each of the subtrees rooted at u and w. The element a_i must be a leaf of w since otherwise both a_i and its left match are in u and v is not the lowest ancestor containing them. Let j be the index of a_i in P_w (which is also its index in $L(w)$), and let k be the index of a_i in $A(u, w)$. Then, out of the first $k-1$ elements of $A(u, w)$ (these $k-1$ elements constitute the elements of $L(u)$ and $P(w)$ that are smaller than a_i), $r_1 - j$ are elements of w, and thus $(k-1) - (r_1 - j) = k - r_1 + j - 1$ are elements of u. It follows that the $(k - r_1 + j - 1)$th element of u is the left match of a_i. Finding the right match of a_i is similar.

Lemma 2.3 follows.

2.3 Range Minima

Lemma 2.4 *Let $A = (a_1, \ldots, a_n)$ be an array of elements drawn from the domain $[1..s]$. The preprocessing algorithm below solves the range minima problem with respect to A in $O(1)$ time using $n \log^3 s$ processors and $O(ns^\epsilon \log s)$ space for any given ϵ, $0 < \epsilon < 1$. Following this preprocessing, each range minimum query can be answered in constant time by one processor.*

2.3.1 Preprocessing

Step 1 (Data Structure). Build a full binary tree T_A whose leaves are the elements of A.

Step 2 (Processor Allocation). Allocate $\log^3 s$ processors to each leaf a_i of T_A: $\log^2 s$ processors for each ancestor of a_i in T_A (note that a_i has $\log n \leq \log s$ ancestors).

Step 3 (Prefix- and Suffix-Minima Computation). For each internal node v compute prefix minima and suffix minima over an array $L(v)$ that contains the leaves of the subtree rooted at v. This is done using steps 3 and 4 in the algorithm of Lemma 2.3. The processor and space complexities are $O(n \log^3 s)$ and $O(ns^\epsilon \log s)$ respectively.

2.3.2 Query Retrieval

To answer a query $MIN(i,j)$ we find the lowest common ancestor v of a_i and a_j. $MIN(i,j)$ is then the minimum between the following two minima: (1) the minimum over the suffix of a_i in the array of leaves of the subtree rooted at the left child of v; and (2) the minimum over the prefix of a_j in the array of leaves of the subtree rooted at the right child of v. These two minima are computed in the preprocessing algorithm above. We note that since T_A is a full binary tree, the computation of the lowest common ancestor of a_i and a_j can be done in $O(1)$ time using a single processor (see [22]).

Lemma 2.4 follows.

3 Optimal Algorithms

We present optimal algorithms for the ANSV problem and the range-minima problem. Since range minima is a generalization of prefix minima, this also implies optimal algorithms for the problems of finding the minimum and prefix minima.

3.1 All Nearest Smaller Values

We divide the input into $n/\log^3 s$ subarrays of size $\log^3 s$ each and apply the optimal doubly logarithmic ANSV algorithm of [8] to each subarray. This takes $O(\log\log\log s)$ time using $n/\log\log\log s$ processors. We now solve the ANSV problem with respect to an array of $n/\log^3 s$ minima, one minimum from each subarray. This is done in $O(1)$ time using n processors and $O(ns^\epsilon/\log^2 s)$ space using Lemma 2.3 and can thus be implemented in $O(\log\log\log s)$ time using $n/\log\log\log s$ processors and $O(ns^\epsilon)$ space. Finally, using this data we reduce, in $O(1)$ time, the problem of finding nearest smaller values for all elements into (at most) $2n/\log^3 s$ merging problems each of size $2\log^3 s$. The details of this reduction are given in [8] (pages 351–354) and are thus omitted from this manuscript. (In [8] the subarrays are of size $\log n$, but the same details work also for subarrays of size $2\log^3 s$.) We solve each such merging problem using the optimal doubly-logarithmic algorithm for merging of [25]. This takes $O(\log\log\log s)$ time using $\log^3 s/\log\log\log s$ processors for each merging problem and $O(\log\log\log s)$ time using $n/\log\log\log s$ processors overall.

3.1.1 Triangulating a monotone polygon

A *mononote polygonal chain* consists of a series of vertices $Q = (q_1, \ldots, q_m)$, so that for all i, $i = 1, \ldots m-1$, there is an edge between q_i and q_{i+1}, and $q_1, \ldots, q_m$ are in increasing (or decreasing) order by the x-coordinate. A *monotone polygon* is a (closed) non-intersecting polygon composed of two monotone polygonal chains: the *upper* and *lower* chains. We assume without loss of generality that the upper chain goes from the vertex with minimum x-coordinate to the vertex with maximum x-coordinate. A *one-sided monotone polygon* (OSMP) is a monotone polygon whose upper (or lower) chain is a straight line.

In [8] an optimal $O(\log\log n)$ time algorithm is given for triangulating a monotone polygon. The algorithm has two stages:

(I) Merge the upper and lower chains of the polygon. This reduces the problem to that of triangulating (possibly many) OSMPs.

(II) Triangulate each OSMP using an algorithm for ANSV.

We perform Stage (I) using the optimal triply-logarithmic merging algorithm of [9]. Stage (II) is performed using the optimal triply-logarithmic ANSV algorithm above.

We conclude:

Theorem 4 *A monotone polygon whose coordinates are taken from the domain* $[1..s]$*,* $s \geq n$*, can be triangulated in* $O(\log\log\log s)$ *time using an optimal number of processors and* $O(ns^{\epsilon})$ *space.*

3.2 Range Minima

We divide the input into $n/\log^3 s$ subarrays of size $\log^3 s$ each and preprocess each subarray so that range-minima queries within the subarray can be answered in $O(1)$ time. This can be done using the optimal doubly logarithmic range-minima algorithm of [8] and takes $O(\log\log\log s)$ time using $n/\log\log\log s$ processors and linear space for all subarrays. Next we apply the algorithm of Lemma 2.4 to an array of $n/\log^3 s$ minima, a minimum from each subarray. This takes $O(1)$ time using n processors and $O(ns^{\epsilon})$ space and enables answering a range minimum query with respect to this array in $O(1)$ time. It is easy to see that using this data each range minimum query can be answered in constant time: A range minimum query within a subarray can be answered using the preprocessing done with respect to the subarray. A range minimum between subarrays can be answered using the preprocessing done to the $n\log^3 s$ minima.

4 The Lower Bound

The lower bound given here follows the general outlines of other PRAM lower bounds [14, 15, 21, 29]. 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. Each processor of the PRAM is an unbounded state machine; the actions (where to write, what to write, where to read) of that processor during step t are functions of the state σ the processor is in at the beginning of step t, and the state of the processor at the beginning of step $t+1$ is a function of σ and the value 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 [14]) 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 n-processor* PRIORITY CRCW PRAM *requires $\Omega(\log\log n)$ steps to find the maximum of n numbers in the domain $[1..s]$, when $s \geq 2^{2^{c_1 \log n \log\log n}}$, for a constant c_1 specified by the proof of the theorem.*

Proof. Given an n-processor 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 behavior 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 several invariants which will be shown to hold by construction.

Invariant 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 Invariant 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.

Invariant 2.1: For every step $t' \leq t$ and over all allowable inputs, a processor either does not write at step t' or always writes.

Invariant 2.2: Any read or write access function at step t', considered as a function over S_t, is either constant or 1–1.

Invariant 2.3: Any two access functions (read or write) used before or at step t are either identical, or have disjoint ranges.

Given these invariants, 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 invariants are trivially satisfied. For the inductive step, suppose the situation as described above holds through step t. We describe how to maintain the invariants 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 [21]. Lemmas 4.1 and 4.2 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 [14].

Lemma 4.1 *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 ([13]) 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 access function to be 0 if the processor does not wish to write, and apply Lemma 4.1 to the set of all read and write access functions used at step $t + 1$. This restricts S_{t+1} and ensures invariant 2.3 holds after step $t + 1$. 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 4.1 by assuming an initial domain size of $(kq)^{k+1}$ reduced to q. Once we have applied Lemma 4.1 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 and that invariant 2.1 holds. At this point, then, the size of S_{t+1} is $\dfrac{s_t^{1/(n(t+1)+1)}}{n(t+1)} - n(t+1)$.

Lemma 4.2 *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 4.2 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 invariant 2.2 holds after step $t+1$.

It remains to ensure that invariant 1 holds after step $t+1$. There are two ways in which it can be violated: if a cell that "knows" one free variable (whose state is a non-constant function of that 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 [3] states that in any graph with v vertices and e edges, there exists an independent set of size $\dfrac{v^2}{v+2e}$. Hence in our graph there is an independent set of size $\dfrac{v_t^2}{v_t+2n(t+1)} \geq \dfrac{v_t^2}{3n(t+1)}$.

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 invariant 1. All invariants are now satisfied. The resulting recurrence equations (slightly simplified) are:

$$\begin{aligned} v_{t+1} &\geq \frac{v_t^2}{3n(t+1)} \\ s_{t+1} &\geq \frac{s_t^{1/n(t+1)+1}}{n(t+1)2^{(t+1)^2(t+2)n^2}} - n(t+1) \end{aligned}$$

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

$$\begin{aligned} v_t &\geq \frac{n}{2^{2^{3t}}} \\ s_t &\geq \frac{s^{n^{-3t}}}{2^{n^{2t}}} \end{aligned}$$

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 $2^{n^{4\log\log n}} = 2^{2^{\Omega(\log n \log\log n)}}$. ∎

Corollary 6 *No n-processor* PRIORITY CRCW PRAM *algorithm for finding the minimum of n numbers drawn from the range $[1\ldots s]$ can run in time less than $\frac{1}{3}\log\log\log s$ for all values of s.*

Proof. The previous theorem showed a lower bound of $T \geq \frac{1}{3}\log\log n$ for $s = 2^{n^{4\log\log n}}$. A simple calculation shows that $T \geq \frac{1}{3}\log\log\log s$ for n sufficiently large. ∎

5 Conclusions

We have shown that the minima, prefix-minima, range-minima, and ANSV problems, with input elements taken from the integer domain $[1..s]$, $s \geq n$, can all be solved in $O(\log\log\log s)$ time using $n/\log\log\log s$ processors (optimal speedup) on the COMMON CRCW PRAM. As an application, we obtain an algorithm with the same bounds for the problem of triangulating a monotone polygon whose coordinates are taken from the integer domain $[1..s]$. Our results were recently used by [6] to obtain $O(\log\log\log s)$ time algorithms for values of s smaller than n.

We also gave a matching lower bound of $\Omega(\log\log\log s)$ for $2^{2^{c_1 \log n \log\log n}} \leq s \leq 2^{2^{\log^{c_2} n}}$, where c_1 is a specific constant and c_2 an arbitrary constant. Thus, our algorithms cannot be improved when expressed solely in terms of the domain size. This result is somewhat unsatisfying, however, since for the given range of s, $\log\log\log s = \Omega(\log\log n)$. The lower bound is an advance over the previously known bounds [28], which required larger domain sizes, but it would be preferable to show our algorithms are tight for all values of s, particularly those below n. There is evidence, however, that this is not the case. [15] gave a technique which could be applied to find the minimum of n integers from the range $[1..n^k]$ in $O(k)$ time on a COMMON CRCW PRAM; [5] gave an $O(\log^* n)$ time algorithm on PRIORITY for computing the prefix-minima when $s = O(n)$. 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 minimum computation that are tight for all s.

[9] 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 [11] 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.

Acknowledgments

We would like to thank the anonymous referees for their careful reading of the paper and their many comments, which helped to improve the presentation. We would also like to thank Dany Breslauer for useful comments on an earlier version of the paper.

References

[1] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 344–357, 1990.

[2] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 83–93, 1987.

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

[4] O. Berkman. *Paradigms for very fast parallel algorithms.* PhD thesis, Tel Aviv University, Tel Aviv 69978, Israel, August 1991.

[5] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella, and U. Vishkin. Top-bottom routing around a rectangle is as easy as computing prefix minima. *SIAM J. of Computing*, 23(3):229–465, 1994.

[6] O. Berkman and Y. Matias. Fast parallel algorithms for minimum and related problems with small integer inputs. *Parallel Processing Letters*, 5(2):223–230, 1995.

[7] O. Berkman, Y. Matias, and U. Vishkin. Randomized range-maxima in nearly-constant parallel time. *Computational Complexity* 2:350–373, 1992.

[8] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14:344–370, 1993.

[9] O. Berkman and U. Vishkin. On parallel integer merging. *Information and Computation*, 106,2:266–285, 1993.

[10] R.B. Boppana. Optimal separations between concurrent-write parallel machines. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 320–326, 1989.

[11] J. Edmonds. Lower bounds with smaller domain size on concurrent write parallel machines. In *Proc. 6th Annual IEEE Conference on Structure in Complexity Theory*, 1991.

[12] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

[13] P. Erdős and R. Rado. Intersection theorems for systems of sets. *J. London Math. Soc.*, 35:85–90, 1960.

[14] F.E. Fich, F. Meyer auf der Heide, and A. Wigderson. Lower bounds for parallel random-access machines with unbounded shared memory. In *Advances in Computing Research.* JAI Press, 1986.

[15] F.E. Fich, P.L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation (preliminary version). In *Proceedings 3rd ACM Symp. on Principles of Distributed Computing*, pages 179–189, 1984.

[16] F.E. Fich, P.L. Ragde, and A. Wigderson. Simulations among concurrent-write PRAMs. *Algorithmica*, 3:43–51, 1988.

[17] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 135–143, 1984.

[18] M. Gereb-Graus and D. Krizanc. The complexity of parallel comparison merging. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 195–201, 1987. Also *SIAM J. Comput.*, to appear.

[19] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 698–710, October 1991.

[20] M.T. Goodrich. Triangulating a polygon in parallel. *Journal of Algorithms*, 10:327–351, 1989.

[21] V. Grolmusz and P.L. Ragde. Incomparability in parallel computation. *Discrete Applied Mathematics*, 29:63–78, 1990.

[22] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[23] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Inc., 1992.

[24] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–941. North-Holland, Amsterdam, 1990.

[25] C.P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. on Comp*, C-32:942–946, 1983.

[26] Y. Matias. *Highly Parallel Randomized Algorithmics.* PhD thesis, Tel Aviv University, Israel, 1992.

[27] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 307–316, May 1991.

[28] F. Meyer auf der Heide and A. Wigderson. The complexity of parallel sorting. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 532–540, 1985.

[29] P.L. Ragde, W.L. Steiger, E. Szemerédi, and A. Wigderson. The parallel complexity of element distinctness is $\Omega(\sqrt{\log n})$. *SIAM Journal on Disceret Mathematics*, 1(3):399–410, August 1988.

[30] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic parallel time. In *Proc. 3rd Aegean Workshop on Parallel Computing, Springer LNCS 319*, pages 33–42, 1988.

[31] J.H. Reif, editor. *A Synthesis of Parallel Algorithms.* Morgan-Kaufmann, San Mateo, CA, 1993.

[32] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, May 1985.

[33] B. Schieber. *Design and analysis of some parallel algorithms.* PhD thesis, Dept. of Computer Science, Tel Aviv Univ., 1987.

[34] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[35] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[36] M. Snir. On parallel searching. *SIAM Journal on Computing*, 14:688–707, 1985.

[37] L.G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4:348–355, 1975.

[38] U. Vishkin. Structural parallel algorithmics. In *Proc. 18th Int. Colloquium on Automata Languages and Programming, Springer LNCS 510*, pages 363–380, 1991.