An Optimal Hypercube Algorithm for the All Nearest Smaller Values Problem

Dina Kravets[*] *C. Greg Plaxton*[†]

Abstract

Given a sequence of n elements, the All Nearest Smaller Values (ANSV) problem is to find, for each element in the sequence, the nearest element to the left (right) that is smaller, or to report that no such element exists. Time and work optimal algorithms for this problem are known on all the PRAM models [4, 6], but the running time of the best previous hypercube algorithm [9] is optimal only when the number of processors p satisfies $1 \leq p \leq n/((\lg^3 n)(\lg\lg n)^2)$. In this paper, we prove that any normal hypercube algorithm requires $\Omega(n)$ processors to solve the ANSV problem in $O(\lg n)$ time, and we present the first normal hypercube algorithm for the ANSV problem that is optimal for all values of n and p. We use our ANSV algorithm to give the first $O(\lg n)$-time n-processor normal hypercube algorithms for triangulating a monotone polygon and for constructing a Cartesian tree.

[*]Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102. Supported by NSF Research Initiation Award CCR–9308204 and the New Jersey Institute of Technology SBR under Grant No. 421220. Email: `dina@cis.njit.edu`.

[†]Department of Computer Science, University of Texas, Austin, TX 78712. Supported by the Texas Advanced Research Program under Grant No. 003658–461. Email: `plaxton@cs.utexas.edu`.

1 Introduction

The *All Nearest Smaller Values (ANSV)* problem is defined as follows. Let $W = \langle w_i : 0 \leq i < n \rangle$ be a sequence of n elements. For each w_i, $0 \leq i < n$, we want to find the nearest element to the left of w_i in W and the nearest element to the right of w_i in W that are smaller than w_i, if such elements exist. More formally, the left nearest neighbor of w_i in W (called the *left match* of w_i) is w_ℓ such that $w_\ell < w_i$ and $w_i \leq w_j$ for all $\ell < j < i$. Similarly, the right nearest neighbor of w_i in W (called the *right match* of w_i) is w_r such that $w_r < w_i$ and $w_i \leq w_j$ for all $i < j < r$. Recently, the ANSV problem has been identified as an important sub-problem in the design of efficient parallel algorithms. In particular, a subroutine for the ANSV problem is used by Aggarwal et al. [2] in parallel searching of staircase-Monge arrays, and by Berkman et al. [4] in finding a triangulation of a monotone polygon, preprocessing for answering range minimum queries in constant time, reconstructing a binary tree from its inorder and either preorder or postorder labelings, and matching parenthesis. Furthermore, two fundamental problems can be reduced to ANSV: (i) merging two sorted lists [5, 10], and (ii) finding the maximum of n elements [13].

The ANSV problem is easy to solve sequentially in $O(n)$ time using a stack. Berkman, Schieber, and Vishkin [4] give the following PRAM algorithms for the ANSV problem: (i) an $O(\lg n)$-time $(n/\lg n)$-processor CREW PRAM algorithm, and (ii) an $O(\lg \lg n)$-time $(n/\lg \lg n)$-processor CRCW PRAM algorithm. Chen [6] gives an EREW PRAM algorithm that matches the CREW PRAM bounds of [4]. In this paper, we develop the first $O(\lg n)$-time hypercube algorithm for the ANSV problem. Our algorithm uses n processors and belongs to the class of so-called "normal" hypercube algorithms, and thus achieves the same processor/time bounds on any of the bounded-degree variants of the hypercube (e.g., the butterfly, cube-connected cycles, and shuffle-exchange). Furthermore, we prove that any normal hypercube algorithm requires $\Omega(n)$ processors to solve the ANSV problem in $O(\lg n)$ time.

Our paper is not the first to consider the complexity of normal hypercube algorithms for the ANSV problem. In particular, JáJá and Ryu [9] give a normal hypercube algorithm for the ANSV problem with optimal running time for any number p of processors satisfying $1 \leq p \leq n/((\lg^3 n)(\lg \lg n)^2)$. In contrast, our algorithm is optimal for all values of n and p. (Our $O(\lg n)$-time, n-processor algorithm is easily generalized to obtain a p-processor algorithm running in time $O((n/p)\lg p + \lg n)$.) For the case $n = p$, JáJá and Ryu [9] obtain a time bound of $O(\lg n(\lg \lg n)^2)$ by making use of the Sharesort [7] sorting algorithm as a subroutine. Our $O(\lg n)$-time algorithm, on the other hand, does not make use of a general routing or sorting subroutine; instead, we confine our on-line routing operations to restricted classes of permutations for which optimal-time normal hypercube are known. Most importantly, we make use of the the optimal parentheses routing algorithm of Mayr and Werchner [12].

We use our ANSV algorithm to obtain more efficient hypercube algorithms for the following two problems:

- *Triangulating a monotone polygon.* A simple polygon is *monotone with respect to a line* ℓ if any line orthogonal to ℓ intersects the polygon in at most two points. The triangulation of a simple polygon has numerous applications in computational geometry [15]. The triangulation of a monotone polygon is a subroutine used in all of the known parallel algorithms for triangulating simple polygons. Berkman et al. [4] give CREW and CRCW PRAM algorithms for triangulating a monotone polygon. JáJá and Ryu [9] give a normal hypercube algorithm for this problem with the same asymptotic performance as their ANSV algorithm. We give the first $O(\lg n)$-time n-processor normal hypercube algorithm for monotone polygon triangulation.

- *Building a Cartesian tree.* The Cartesian tree of a sequence $W = \langle w_i : 0 \leq i < n\rangle$ is a binary tree where the root corresponds to the element $w_k = \min_{0 \leq i < n} w_i$, the left child is the Cartesian tree of $\langle w_i : 0 \leq i < k\rangle$, and the right child is the Cartesian tree of $\langle w_i : k < i < n\rangle$ [14]. The Cartesian tree is used in preprocessing algorithms for answering range minimum queries. Berkman et al. [4] show how to find the Cartesian tree in $O(\lg n)$ time using an $(n/\lg n)$-processor CREW PRAM. We obtain the first $O(\lg n)$-time n-processor normal hypercube algorithm for this problem.

The remainder of this paper is organized as follows. Section 2 defines the model of computation. Section 3 presents our normal hypercube algorithm for the ANSV problem. Section 4 presents a lower bound for the ANSV problem. Sections 5 and 6 provide normal hypercube algorithms for triangulating a monotone polygon and building a Cartesian tree.

2 Normal Hypercube Algorithms

A *dimension-d hypercube* may be constructed as follows. First, associate a unique d-bit ID with each of $n = 2^d$ processors. Second, connect (by a two-way channel) all pairs of processors whose IDs differ in a single bit position. A channel (or edge) connecting two processors x and y (that is, with IDs x and y) will be referred to as a *dimension i edge* if and only if x and y differ in bit position i.

In this paper, we make the standard assumption that each processor of a p-processor machine has access to a local memory configured in $O(\lg p)$-bit words. With respect to the ANSV problem, we will further assume that each of the given elements w_i can be represented with a constant number of words. We analyze the complexity of our algorithms in terms of *time steps*. In a single time step each processor can send and/or receive a single word of data from an adjacent processor, and can perform one CPU operation on word-sized operands. In addition, we require that: (i) only one dimension of edges is used at any given time step, and (ii) the dimension used at time step $t+1$ is within 1 (modulo d) of the dimension used at time step t, $t \geq 0$. Algorithms satisfying conditions (i) and (ii) are often referred to as *normal* hypercube algorithms [11, Section 3.1.3]. Normal algorithms are widely regarded as the most interesting class of hypercube algorithms, since they can be executed with constant-factor slowdown on any of the bounded-degree variants of the hypercube (e.g., butterfly, shuffle-exchange, cube-connected cycles).

Note that any normal hypercube algorithm can be executed with constant slowdown on an EREW PRAM with the same number of processors. (The converse does not hold.)

3 Two Hypercube Algorithms for the ANSV Problem

In this section we present two $O(\lg n)$-time n-processor normal hypercube algorithms: (i) an algorithm for solving an $(n/\lg n)$-input ANSV problem (Subsection 3.1), and (ii) an algorithm for the usual n-input ANSV problem (Subsection 3.2).

3.1 An $(n/\lg n)$-input n-processor algorithm

We only describe how to find right matches; left matches are found analogously. The SparseANSV algorithm is based on the CREW PRAM algorithm of Berkman et al. [4]. Before describing the algorithm, we give some definitions and notations. We assume a d-dimensional hypercube containing $2^d = n$ processors, $d \geq 1$. Let d' denote the minimum integer such that $2^{d'} \geq d - d'$. Note that $d' \sim \lg d = \lg\lg n$. For all i and j such that $0 \leq i < 2^{d-d'}$ and $0 \leq j < d - d'$, the processor with ID $i \cdot (d - d') + j$ will be referred to as processor (i, j). Local variable w at processor (i, j) will be referred to as $w[i, j]$.

The input to our algorithm is a set of integer variables $w[i, 0]$, $0 \leq i < 2^{d-d'}$. For each i, $0 \leq i < 2^{d-d'}$, let $r(i)$ denote the minimum integer such that $i < r(i) < 2^{d-d'}$ and $w[i, 0] > w[r(i), 0]$. If no such integer exists, then let $r(i) = +\infty$. Further define the *right-match count* of i, denoted $c(i)$, to be the number of integers j such that $0 \leq j < 2^{d-d'}$ and $r(j) = i$. The output of our algorithm is integer variables $c[i, 0] = c(i)$ and $r[i, 0] = r(i)$, $0 \leq i < 2^{d-d'}$. For all integers α and β such that $0 \leq \alpha < 2^{d-d'-\beta-1}$ and $0 \leq \beta < d - d'$, let $A_{\alpha,\beta}$ denote the subcube consisting of the 2^β processors $(\alpha \cdot 2^{\beta+1} + i, \beta)$, $0 \leq i < 2^\beta$. Similarly, let $B_{\alpha,\beta}$ denote the subcube consisting of the 2^β processors $(\alpha \cdot 2^{\beta+1} + 2^\beta + i, \beta)$, $0 \leq i < 2^\beta$. Note that the subcubes $A_{\alpha,\beta}$ and $B_{\alpha,\beta}$ partition the $(2^{d-d'}) \cdot (d - d')$ processors (i, j) with $0 \leq i < 2^{d-d'}$ and $0 \leq j < d - d'$. Figure 3.1 shows the partition of the processors into $A_{\alpha,\beta}$'s and $B_{\alpha,\beta}$'s. Let $C_{\alpha,\beta} = A_{\alpha,\beta} \cup B_{\alpha,\beta}$.

Define a sequence of records S to be *quasi-sorted* if and only if the subsequence S' of S consisting of all records with key value not equal to $+\infty$ is sorted. For all integers x and i, let x_i denote the ith bit of the binary representation of x (numbering from 0), and let $x^{(i)}$ denote the integer obtained by complementing the ith bit of x. Define the *greedy decreasing subsequence* of a sequence $\langle a_i : 0 \leq i < n \rangle$ as

$$\langle a_i : a_i < a_j \text{ for all } 0 \leq j < i < n \rangle.$$

In the description of our algorithm, each step consists of the general idea of the step (italicized), followed by the details of the step, which include the specification of the routines that implement the step on the hypercube and the running time of the implementation.

$\boxed{\text{Algorithm SparseANSV}}$

1. *For each i, $0 \leq i < 2^{d-d'}$, if $r(i) \neq +\infty$ then set $f[i, 0]$ to the most significant bit position in which the binary representation of i differs from that of $r(i)$. If $r(i) = +\infty$ then set $f[i, 0] = +\infty$.*

   At each processor $(i, 0)$, $0 \leq i < 2^{d-d'}$, we execute the following loop:

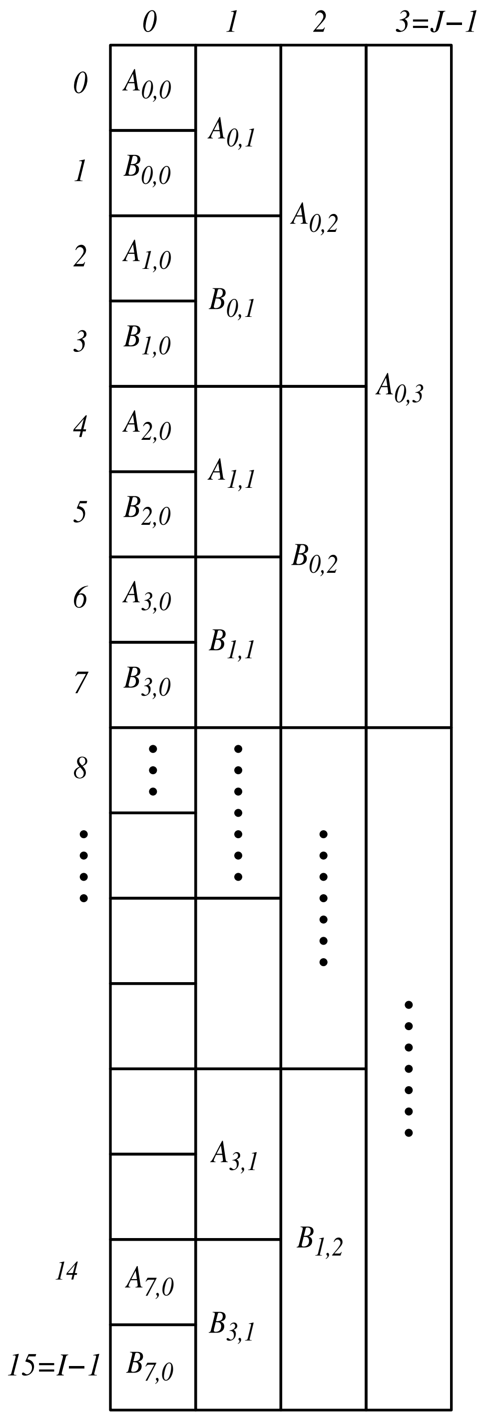

Figure 3.1: The partition of the processors into $A_{\alpha,\beta}$'s and $B_{\alpha,\beta}$'s; $I = 2^{d-d'}$ and $J = d - d'$.

$f[i,0] := +\infty$
$w'[i,0] := +\infty$
for $j = 0$ **to** $d - d' - 1$
  $w'[i,0] := \min\{w'[i,0], w'[i^{(j)},0]\}$
  if $(i_j = 0 \wedge f[i,0] = +\infty$
    $\wedge\, w[i,0] > w'[i,0])$ **then**
   $f[i,0] := j$

Implementation: Single pass over dimensions d' to $d-1$. Running time: $O(d)$. Remarks: See Figure 3.2 for an example.

	input *x*				*f* after step 1				*s* after step 2(a)				*s* after step 2(b)			
	0	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*
0000	*10*				*1*				∞	*10*	∞	∞	∞	*10*	*4*	*7*
0001	*13*				*1*				∞	*13*	∞	∞	∞	*13*	∞	*9*
0010	*5*				*0*				*5*	∞	∞	∞	*5*	∞	∞	∞
0011	*4*				*2*				∞	∞	*4*	∞	∞	∞	∞	∞
0100	*1*				∞				∞	∞	∞	∞	∞	∞	∞	∞
0101	*7*				*3*				∞	∞	∞	*7*	∞	∞	∞	∞
0110	*20*				*0*				*20*	∞	∞	∞	*20*	∞	∞	∞
0111	*9*				*3*				∞	∞	∞	*9*	∞	∞	∞	∞
1000	*17*				*0*				*17*	∞	∞	∞	*17*	*12*	*8*	∞
1001	*12*				*1*				∞	*12*	∞	∞	∞	∞	∞	∞
1010	*15*				*0*				*15*	∞	∞	∞	*15*	∞	∞	∞
1011	*8*				*2*				∞	∞	*8*	∞	∞	∞	∞	∞
1100	*3*				*1*				∞	*3*	∞	∞	∞	*3*	∞	∞
1101	*23*				*0*				∞	*23*	∞	∞	∞	*23*	∞	∞
1110	*2*				∞				∞	∞	∞	∞	∞	∞	∞	∞
1111	*18*				∞				∞	∞	∞	∞	∞	∞	∞	∞

Figure 3.2: The contents of certain variables following the first few steps of algorithm SparseANSV.

2. *Let $\mathcal{S}_{\alpha,\beta}$ denote the subsequence of all w_i values such that $f[i,0] = \beta$ and $\lfloor i \cdot 2^{-\beta-1} \rfloor = \alpha$, $0 \le \alpha < 2^{d-d'-\beta-1}$, $0 \le \beta < d-d'$. Note that each $\mathcal{S}_{\alpha,\beta}$ is an increasing subsequence. In the following substeps, we identify each subsequence $\mathcal{S}_{\alpha,\beta}$, save the addresses of the w_i's constituting $\mathcal{S}_{\alpha,\beta}$, and concentrate $\mathcal{S}_{\alpha,\beta}$ in subcube $A_{\alpha,\beta}$.*

   (a) Set $s[i,j] = +\infty$, $0 \le i < 2^{d-d'}$, $0 \le j < d-d'$. For all i such that $0 \le i < 2^{d-d'}$ and $f[i,0] \ne +\infty$, route $w[i,0]$ to $s[i, f[i,0]]$. Implementation: Greedy routing over dimensions 0 to $d'-1$. Running time: $O(\lg d)$. Remark: Note that the greedy routing is collision-free.

(b) Set $i'[i,j]$ to i, $0 \leq i < 2^{d-d'}$, $0 \leq j < d-d'$. Stably sort the pairs (s,i') within each subcube $A_{\alpha,\beta}$ in ascending order of s. Implementation: Monotone route. Running time: $O(d)$. Remarks: (i) Note that the (s,i') pairs in $A_{\alpha,\beta}$ are already quasi-sorted (sorting a quasi-sorted sequence is also referred to in the literature as concentration routing); (ii) The i' values will be used in Step 5(a) to send information regarding the w_i's back to the appropriate original locations.

3. *Let $\mathcal{T}_{\alpha,\beta}$ denote the greedy decreasing subsequence of those w_i values for which $i_\beta = 1$ and $\lfloor i \cdot 2^{-\beta-1} \rfloor = \alpha$, $0 \leq \alpha < 2^{d-d'-\beta-1}$, $0 \leq \beta < d-d'$. In the following substeps, we identify each subsequence $\mathcal{T}_{\alpha,\beta}$, save the addresses of the w_i's constituting $\mathcal{T}_{\alpha,\beta}$, and concentrate $\mathcal{T}_{\alpha,\beta}$ in subcube $B_{\alpha,\beta}$.*

   (a) For $0 \leq i < 2^{d-d'}$, $0 \leq j < d-d'$, copy $w[i,0]$ to $t[i,j]$. Let $B'_{\alpha,\beta}$ denote the set of all processors in $B_{\alpha,\beta}$ such that some lower-numbered processor in $B_{\alpha,\beta}$ has a strictly smaller value of t. At each processor belonging to some $B'_{\alpha,\beta}$, set t to $+\infty$. Implementation: Prefix operation. Running time: $O(d)$.

   (b) Stably sort the pairs (t,i') within each subcube $B_{\alpha,\beta}$ in descending order of t. Implementation: Monotone route. Running Time: $O(d)$. Remarks: (i) Note that the pairs (t,i') in each subcube $B_{\alpha,\beta}$ are already quasi-sorted; (ii) The i' values will be used in Step 6(a) to send information regarding the w_i's back to the appropriate original locations.

4. *Perform the following operations within each subcube $C_{\alpha,\beta}$. First, merge the sorted sequences $\mathcal{S}_{\alpha,\beta}$ and $\mathcal{T}_{\alpha,\beta}$. Then, use prefix operations to determine the right match of each element in $\mathcal{S}_{\alpha,\beta}$. (As proven in [4], the right match of every element of $\mathcal{S}_{\alpha,\beta}$ is an element of $\mathcal{T}_{\alpha,\beta}$.) Use additional prefix operations to count, for each element y of $\mathcal{T}_{\alpha,\beta}$, the number of elements x in $\mathcal{S}_{\alpha,\beta}$ such that y is the right match of x.*

   (a) At each processor (i,β) in $A_{\alpha,\beta}$ such that $s \neq +\infty$, set $r[i,\beta]$ to j where (j,β) is the lowest-numbered processor of $B_{\alpha,\beta}$ such that $s[i,\beta] > t[j,\beta]$. If no such processor (j,β) exists, then set $r[i,\beta]$ to $+\infty$. Implementation: Bitonic merge and prefix operations within each subcube $C_{\alpha,\beta}$. Running time: $O(d)$.

   (b) For each processor (j,β) in $B_{\alpha,\beta}$, set $c(j,\beta)$ to the number of processors (i,β) in $A_{\alpha,\beta}$ for which (j,β) is the lowest-numbered processor in $B_{\alpha,\beta}$ with $s[i,\beta] > t[j,\beta]$. Implementation: Prefix operations within each subcube $C_{\alpha,\beta}$. Running time: $O(d)$.

5. *Route $r(i)$ to processor $(i,0)$, $0 \leq i < 2^{d-d'}$.*

   (a) Sort the pairs (i',r) within each subcube $A_{\alpha,\beta}$ in ascending order of i'. Implementation: Monotone route. Running time: $O(d)$. Remark: This routing operation is the inverse of that applied in Step 2(b).

   (b) For each i, $0 \leq i < 2^{d-d'}$, at most one processor of the form (i,j), $0 \leq j < d-d'$, contains a value of r that is not equal to $+\infty$. Route this value to $r[i,0]$.

Implementation: Greedy routing over dimensions 0 to $d' - 1$. Running time: $O(\lg d)$.

6. *For each i, $0 \leq i < 2^{d-d'}$, sum the associated partial counts computed in Step 4(b) to obtain $c(i)$, and store the result in processor $(i, 0)$.*

   (a) Sort the pairs (i', c) within each subcube $B_{\alpha,\beta}$ in descending order of i'. Implementation: Monotone route. Running time: $O(d)$. Remark: This routing operation is the inverse of that applied in Step 3(b).

   (b) For each i, $0 \leq i < 2^{d-d'}$, sum the values of $c[i, j]$, $0 \leq j < d - d'$, and store the result in $c[i, 0]$. Implementation: Prefix operation. Running time: $O(\lg d)$.

Since each step of SparseANSV runs in $O(d)$ time, this algorithm solves any $(n/\lg n)$-input ANSV problem in $O(\lg n)$ time using n processors. The central lemma underlying the proof of correctness of algorithm SparseANSV is due to Berkman et al. [4], and is stated below. We omit the proof of correctness of algorithm SparseANSV since it is quite similar to the proof of correctness of the CREW PRAM algorithm of Berkman et al. [4].

Lemma 3.1 ([4]) *The right matches for the elements of $\mathcal{S}_{\alpha,\beta}$ are among the elements of $\mathcal{T}_{\alpha,\beta}$.*

3.2 An n-input n-processor algorithm

We will now show how to solve a larger ANSV problem using the same number of processors. The algorithm presented in this section offers two main advantages over the algorithm of Berkman et al. [4]: (i) the present algorithm runs efficiently on any hypercubic machine, and (ii) the high-level structure of the present algorithm is somewhat simpler. The simplification we achieve is primarily due to Lemma 3.2 (stated at the end of this section), which provides a useful property for streamlining the search for matches.

We assume a d-dimensional hypercube containing $2^d = n$ processors, $d \geq 1$. Let d' denote the minimum integer such that $2^{d'} \geq d - d'$. Note that $d' \sim \lg d = \lg \lg n$. It will be convenient to emulate a $(d+1)$-dimensional hypercube on the given d-dimensional machine. (This can be done with constant slowdown.) In what follows, the term "processor" should be interpreted as "virtual processor". For all i and j such that $0 \leq i < 2^{d-d'+1}$ and $0 \leq j < 2^{d'}$, the processor with ID $k = i \cdot 2^{d'} + j$ will be referred to either as processor k or as processor (i, j). Local variable w at processor k (resp., processor (i, j)) will be referred to as $w[k]$ (resp., $w[i, j]$).

The input to our algorithm is a set of integer variables $w[k]$, $0 \leq k < 2^d$. In order to simplify the presentation, we assume that the $w[k]$'s are distinct. (At the end of this section, we prove that there is no loss of generality inherent in this assumption; see Lemma 3.3.) For each k, $0 \leq k < 2^d$, let $r(k)$ denote the minimum integer such that $k < r(k) < 2^d$ and $w[k] > w[r(k)]$. If no such integer exists, then let $r(k) = +\infty$. Analogously, let $\ell(k)$ denote the maximum integer such that $0 \leq \ell(k) < k$ and $w[\ell(k)] < w[k]$. If no such integer exists, then let $\ell(k) = -\infty$. The output consists of integer variables $\ell[k] = \ell(k)$ and $r[k] = r(k)$, $0 \leq k < 2^d$.

We now present our algorithm, which runs in $O(d)$ steps. For each i, $0 \le i < 2^{d-d'+1}$, let C_i denote the d'-dimensional subcube consisting of processors (i,j), $0 \le j < 2^{d'}$.

Algorithm ANSV

1. *Recursively solve the ANSV problem within each subcube C_i. In addition, find the minimum element in C_i, denoted m_i.*

   (a) If $d = 0$, then set $\ell[0] = -\infty$, $r[0] = +\infty$, and return. Otherwise, recursively apply algorithm ANSV within each d'-dimensional subcube C_i, $0 \le i < 2^{d-d'}$, and then set $\ell'[k] = \ell[k]$ and $r'[k] = r[k]$, $0 \le k < 2^d$. Implementation: Recursive call. Running time: $T(d')$, where $T(d)$ denotes the running time of ANSV on a d-dimensional hypercube. (We will ultimately find that $T(d) = O(d)$, and so the running time of this step is $O(\lg d)$.)

   (b) At each processor $(i,0)$, $0 \le i < 2^{d-d'}$, set $m[i,0]$ to the minimum w value in C_i. Implementation: Prefix operation. Running time: $O(d)$.

2. *Call* SparseANSV *to find the right matches for the sequence $M = \langle m_i : 0 \le i < n/\lg n\rangle$. Let $R(m_i)$ denote the right match of m_i among the elements of M.*

   (a) At each processor $(i,0)$, $0 \le i < 2^{d-d'}$, set $R[i,0]$ to the minimum integer i' such that $i < i' < 2^{d-d'}$ and $m[i,0] > m[i',0]$ (or to $+\infty$ if no such integer i' exists), and set $c[i,0]$ to the number of integers i', $0 \le i' < i$, such that $R[i',0] = i$. Implementation: SparseANSV, prefix operation. Running time: $O(d)$.

3. *Prune the sequence $\mathcal{S}'_i = \langle m_i, \ldots, w_{i\cdot\lg n-1}\rangle$ (resp., $\mathcal{T}'_i = \langle w_{(i-1)\cdot\lg n}, \ldots, m_i\rangle$) within each subcube C_i by eliminating from it any element with right (resp., left) match in $\mathcal{S}'_i$ (resp., $\mathcal{T}'_i$). Let $\mathcal{S}_i$ (resp., $\mathcal{T}_i$) refer to the pruned sequence.*

   (a) For all integers i and j such that $0 \le i < 2^{d-d'+1}$ and $0 \le j < 2^{d'}$, set $k[i,j]$ to $i \cdot 2^{d'} + j$. Route the pair (k,w) (resp., $(-1,+\infty)$) at processor (i,j) to variables (k',w') at processor $(2\cdot i, j)$ if and only if $\ell'[i,j] = -\infty$ (resp., $\ell'[i,j] \ne -\infty$), $0 \le i < 2^{d-d'}$, $0 \le j < 2^{d'}$. Route the pair (k,w) (resp., $(-1,+\infty)$) at processor (i,j) to variables (k',w') at processor $(2\cdot i+1, j)$ if and only if $r'[i,j] = +\infty$ (resp., $r'[i,j] \ne +\infty$), $0 \le i < 2^{d-d'}$, $0 \le j < 2^{d'}$. Route $c[i,0]$ to $c[2\cdot i, 0]$, $0 \le i < 2^{d-d'}$. Set $c[2\cdot i+1, 0]$ to 1 if $R[i,0] \ne +\infty$, and to 0 otherwise, $0 \le i < 2^{d-d'}$. Implementation: Monotone route operations. Running time: $O(d)$.

   (b) Stably sort the pairs (w',k') within each subcube $C_{2\cdot i}$ in descending order of w'. Stably sort the pairs (w',k') within each subcube $C_{2\cdot i+1}$ in ascending order of w'. Implementation: Monotone route. (Note that the (w',k') pairs in each C_i are already quasi-sorted.) Running time: $O(\lg d)$.

4. *Copy each sequence $\mathcal{T}_i$ to a set of $f(i)$ consecutive subcubes, where $f(i)$ denotes the number of integers j such that $R(m_j) = i$.*

(a) Set $a[i,0]$ to $2^{d'} \cdot \sum_{0 \le i' < i} c[i',0]$, $0 \le i < 2^{d-d'+1}$. Set $a[i,j]$ to $a[i,0]+j$, and $c[i,j]$ to $c[i,0]$, $0 \le i < 2^{d-d'+1}$, $0 \le j < 2^{d'}$. Implementation: Prefix operation. Running time: $O(d)$.

(b) If $c \neq 0$, route the tuple $(i \bmod 2, c, k', w')$ at processor (i,j) to variables (b, c, k', w') at processor $a[i,j]$, $0 \le i < 2^{d-d'+1}$, $0 \le j < 2^{d'}$. At any processor that does not receive a tuple in this routing operation, set $(b,c,k',w') = (-1, 0, -1, +\infty)$. Implementation: Monotone route. Running time: $O(d)$.

(c) If $c > 1$, route the tuple $(b, 0, k', w')$ at processor (i,j) to variables (b,c,k',w') at processors $(i+i', j)$, $0 \le i < 2^{d-d'+1}$, $0 \le i' < c$, $0 \le j < 2^{d'}$. Implementation: Prefix operation. Running time: $O(d)$.

5. *Route copies of $\mathcal{S}_i$ and $\mathcal{T}_{R(m_i)}$ to a common subcube.*

Associate a right (resp., left) parenthesis with each processor (i,j) such that $b=0$ (resp., $b=1$). (Do not associate a parenthesis with any processor (i,j) such that $b=-1$.) Note that the following conditions are satisfied: (i) each subcube C_i either contains no parentheses, entirely left parentheses, or entirely right parentheses, (ii) for each subcube C_i of left parentheses there is a "matching" subcube $C_{i'}$ of right parentheses, $0 \le i < i' < 2^{d-d'+1}$. Furthermore, C_i and $C_{i'}$ are matching subcubes if and only if $R[i,0]$ was set to i' in Step 3. We can efficiently exchange information between all pairs of matching subcubes in parallel by making use of the parenthesis routing operation of Mayr and Werchner [12]. In the following two substeps, the indices i and i' range over all pairs of matching subcubes, $0 \le i < i' < 2^{d-d'+1}$.

(a) Route the pair (k', w') stored in processor (i', j) to variables (k'', w'') at processor (i,j), $0 \le j < 2^{d'}$. Implementation: Parenthesis route. Running time: $O(d)$.

(b) Route the pair (k', w') stored in processor (i,j) to variables (k'', w'') at processor (i', j), $0 \le j < 2^{d'}$. Implementation: Parenthesis route. Running time: $O(d)$.

6. *Determine the right match $r''(w)$ for each w in $\mathcal{S}_i$ among the elements of $\mathcal{T}_{R(m_i)}$. Set $r(w)$ to be the leftmost of $r(w)$ and $r''(w)$.*

The following substeps are now performed for each integer i such that C_i is a "left" subcube (i.e., $b=1$ throughout the subcube), $0 \le i < 2^{d-d'+1}$.

(a) At each processor (i,j) such that $0 \le j < 2^{d'}$ and $w'[i,j] \neq +\infty$, set $r''[i,j]$ to $k''[i,j']$ where j' is the maximum integer such that $0 \le j' < 2^{d'}$ and $w'[i,j] > w''[i,j']$. If no such integer j' exists, then set $r''[i,j]$ to $+\infty$. Implementation: Bitonic merge, prefix operation. Running time: $O(d)$.

(b) For all j such that $0 \le j < 2^{d'}$ and $w'[i,j] \neq +\infty$, route $r''[i,j]$ to $r''[k']$. Implementation: Monotone route. Running time: $O(d)$.

(c) At each processor that received an r'' value in the preceding substep, set r to $\min\{r, r''\}$. Running time: $O(1)$.

7. *Determine the left match $\ell''(w,i)$ for each w in $\mathcal{T}_{R(m_i)}$ among the elements of $\mathcal{S}_i$. Set $\ell''(w)$ to be the rightmost of the $\ell(w,i)$'s. Set $\ell(w)$ to be the rightmost of $\ell(w)$ and $\ell''(w)$.*

Note that every "right" subcube C_i (i.e., $b = 0$ throughout the subcube) for which $c \neq 0$ belongs to a contiguous sequence of right subcubes $\langle C_j : i \leq j < i + \Delta \rangle$ such that: (i) $\Delta > 0$, (ii) every processor in C_i has $c = \Delta$, (iii) every other processor in the sequence of subcubes has $c = 0$, and (iv) every subcube in the sequence contains the same sorted sequence of $2^{d'}$ (w', k') pairs. The following substeps are now performed for all such pairs of integers (i, Δ) in parallel.

(a) At each processor $(i+\Delta', j)$ such that $0 \leq \Delta' < \Delta$, $0 \leq j < 2^{d'}$, and $w'[i+\Delta', j] \neq +\infty$, set $\ell''[i + \Delta', j]$ to $k''[i + \Delta', j']$ where j' is the maximum integer such that $0 \leq j' < 2^{d'}$ and $w''[i + \Delta', j'] < w'[i + \Delta', j]$. If no such integer j' exists, then set $\ell''[i + \Delta', j]$ to $-\infty$. Implementation: Bitonic merge, prefix operation. Running time: $O(d)$.

(b) For all j such that $0 \leq j < 2^{d'}$ and $w'[i, j] \neq +\infty$, set $\ell''[i, j]$ to $\max_{0 \leq \Delta' < \Delta} \ell''[i + \Delta', j]$. Implementation: Prefix operation. Running time: $O(d)$.

(c) For all j such that $0 \leq j < 2^{d'}$ and $w'[i, j] \neq +\infty$, route $\ell''[i, j]$ to $\ell''[k']$. Implementation: Monotone route. Running time: $O(d)$.

(d) At each processor that received an ℓ'' value in the preceding substep, set ℓ to $\max\{\ell, \ell''\}$. Running time: $O(1)$.

8. *Note that Steps 2 through 7 above are based on a call to* SparseANSV *that computes the right matches of the subcube minima. (See Step 2.) We now perform an entirely symmetric sequence of steps corresponding to the left matches of the subcube minima.*

For the time complexity, note that the first step of this algorithm takes time $T(d')$ and all the other steps take time $O(d)$. The solution to the recurrence $T(d) = T(d') + O(d)$ is $O(d)$, and thus this algorithm solves the n-input ANSV problem in $O(\lg n)$ time using n processors.

The correctness of algorithm ANSV depends on the following lemma.

Lemma 3.2 *Given k non-empty integer sequences X_i, $0 \leq i < k$, let Y denote the single sequence obtained by concatenating the X_i's in ascending order of i. Let m_i denote the minimum integer in X_i, $0 \leq i < k$. If all elements of Y are distinct, then for all w in X_i such that the right match of w in Y belongs to X_j, $j > i$, either: (i) the right match of m_i belongs to X_j, or (ii) the left match of m_j belongs to X_i.*

Proof: Abusing our notation slightly, let $r(w)$ (resp., $\ell(w)$) denote the right (resp., left) match of w. For w to the left of w' in Y, let $[w, w']$ denote the sequence of integers within Y that starts at w and ends at w'. Similarly, let (w, w') denote the sequence $[w, w']$ excluding w and w'.

If $w = m_i$, then the lemma is trivially satisfied. Thus, assume that $w \neq m_i$. Since $r(w) \in X_j$ and $j > i$, we know that w is to the right of m_i in X_i. Since m_i is the minimum value in X_i, $r(m_i) \notin X_i$. Similarly, $\ell(m_j) \notin X_j$. From the definition of a right match, we know that $r(w)$ is the minimum value in $[w, r(w)]$ and that $w' > w > r(w)$ for all $w' \in (w, r(w))$. Since $w > m_i$ and $r(w) \geq m_j$, $w' > m_i$ and $w' > m_j$ for all $w' \in (w, r(w))$. Hence, $r(m_i)$ is either $r(w)$ or to the right of $r(w)$, and $\ell(m_j)$ is to the left of w. Assuming (as in the statement of the lemma) that all elements of Y are distinct, either $m_i > m_j$ or $m_j > m_i$. In the former case, $r(m_i)$ is at or to the left of m_j, and thus, $r(m_i) \in X_j$. In the latter case, $\ell(m_j)$ is at or to the right of m_i, and thus, $\ell(m_j) \in X_i$. ■

We can now complete the argument for the correctness of the ANSV algorithm. The sequence X_i in Lemma 3.2 corresponds to the elements stored in subcube C_i. Since the right (resp., left) match of every element in $\langle w_{(i-1)\cdot \lg n}, \ldots, m_i \rangle$ (resp., $\langle m_i, \ldots, w_{i \cdot \lg n - 1} \rangle$) lies in subcube C_i, the sequence $\mathcal{S}_i$ (resp., $\mathcal{T}_i$) contains every element in subcube C_i whose right (resp., left) match lies outside subcube C_i. Thus, the right match of each element in $\mathcal{S}_i$ is determined correctly either in Step 6 of the iteration based on the right matches of the subcube minima when we process $\mathcal{S}_i$ and $\mathcal{T}_{R(m_i)}$, or in Step 7 of the iteration based on the left matches of the subcube minima when we process $\mathcal{T}_j$ and $\mathcal{S}_{L(m_j)}$, where $L(m_j) = i$.

Finally, note that the proof of Lemma 3.2, and hence also our proof of correctness of algorithm ANSV, relies on the assumption that all the input elements (i.e., the $w[k]$'s) are distinct. The following lemma shows that this assumption can easily be avoided.

Lemma 3.3 *Let $\mathcal{A}$ be a comparison-based algorithm for a restricted version of the ANSV problem in which all input elements are distinct. Then $\mathcal{A}$ can be used to solve the unrestricted ANSV problem in the same asymptotic time and processor bounds.*

Proof: Let an instance I of the unrestricted ANSV problem be given, and assume that our goal is to calculate right matches only. (A similar approach can be used to calculate left matches.) We modify I to obtain a same-size instance I' such that: (i) all elements of instance I' are distinct, and (ii) the right matches of the elements in I are in correspondonce with those in I'. The modification is straightforward; we simply append (in the "low-order" $\lceil \lg n \rceil$ bit positions) the binary representation of the integer i to each element w_i of I. Put differently, whenever two equal elements w_i and w_j of I are compared by algorithm $\mathcal{A}$, we return $w_i < w_j$ if and only if $i < j$. ■

4 A Lower Bound

This section establishes a lower bound for the ANSV problem on a wide class of fixed-connection networks, including the hypercube and all of its bounded-degree variants. JáJá and Ryu [9] use a separator-based argument to establish the same asymptotic lower bound for the shuffle-exchange and cube-connected cycles. However, the bound they obtain for the hypercube is weaker than ours by a $\sqrt{\lg p}$ factor. The factor of $\sqrt{\lg p}$ arises because the hypercube has $\Theta(p/\sqrt{\lg p})$-size separators, whereas the shuffle-exchange and cube-connected cycles have $\Theta(p/\lg p)$-size separators. Our (non-separator-based) argument establishes the same lower bound not only for the hypercube, shuffle-exchange, and cube-connected cycles, but also for any bounded-degree expander network. (The separator-based approach of JáJá and Ryu [9] does not give a non-trivial lower bound for bounded-degree expander networks.)

The lower bound is proven under the following set of assumptions:

L1 Each input key is provided at exactly one processor.

L2 In a single time step, a processor can send and receive $O(1)$ messages, and perform $O(1)$ local operations. A message can hold $O(1)$ keys. The only local operations allowed on keys are copy and comparison.

L3 Each of the p processors is $\Omega(\lg p)$ hops away from at least $p - p/\lg p$ processors.

The last assumption is satisfied by a wide class of fixed-connection networks. In the case of the hypercube as well as all bounded-degree networks (including, of course, the bounded-degree variants of the hypercube), every processor is $\Omega(\lg p)$ hops away from $p - p^{\varepsilon}$ processors for any constant $\varepsilon > 0$.

Theorem 1 *Under Assumptions L1 to L3, any p-processor algorithm for the ANSV problem requires $\Omega((n/p)\lg p)$ steps. The same bound holds for the problem of merging two sorted lists of length $n/2$.*

Proof: We establish the lower bound for a highly restricted version of the ANSV problem. (It will be apparent that the problem we consider may also be viewed as a restriction of the merging problem; hence, our lower bound also applies to merging.) In particular, we assume that the input consists of a sequence of $n = 4m$ distinct keys (for convenience, we assume that n is a multiple of 4)

$$\langle a_0, \ldots, a_{2m-1}, b_{2m-1}, \ldots, b_0 \rangle$$

and an integer k, $0 \leq k < m$. The following properties are satisfied by the input: (i) The a_i's are in ascending order, that is, $a_{i-1} < a_i$ for $0 < i < 2m$; (ii) The b_i's are in descending order, that is, $b_i > b_{i-1}$ for $0 < i < 2m$; (iii) Key b_{m+i} has rank $3m + i$, $0 \leq i < m$; (iv) Key a_{m+i} has rank $2m + i$, $k \leq i < m$ (v) Key a_i has rank i, $0 \leq i < k$; (vi) The minimum of keys a_{i+k} and b_i has rank $2i + k$, and the maximum of these two keys has rank $2i + k + 1$, $0 \leq i < m$. Thus, to determine the right match of a_{i+k}, $0 \leq i < m$, it is both necessary and sufficient to compare keys a_{i+k} and b_i. If $a_{i+k} > b_i$ then the right match of a_{i+k} is b_i; otherwise, the right match of a_{i+k} is b_{i-1} (unless $i = 0$, in which case the right match of a_{i+k} is undefined).

In accordance with Assumption L1, we assume that each key is input at a single processor. Let $\ell(a_i)$ (resp., $\ell(b_i)$) denote the index of the processor initially holding a_i (resp., b_i), and let $\Delta(i, j)$ denote the length of a shortest path between processors i and j. Assume without loss of generality that input variable k is provided at all processors. Let $\mathcal{A}$ denote an arbitrary algorithm for this restricted version of the ANSV problem, and let M_k denote the maximum total number of messages sent by algorithm $\mathcal{A}$ for each k, $0 \leq k < m$. Note that

$$M_k \geq \sum_{0 \leq i < m} \Delta(\ell(a_{i+k}), \ell(b_i)).$$

Setting $M'_i = \sum_{0 \leq k \leq m} \Delta(\ell(a_{i+k}), \ell(b_i))$, $0 \leq i < m$, we find that

$$\begin{aligned} \sum_{0 \leq i < m} M'_i &= \sum_{0 \leq i < m} \sum_{0 \leq k < m} \Delta(\ell(a_{i+k}), \ell(b_i)) \\ &\leq \sum_{0 \leq k \leq m} M_k. \end{aligned}$$

We now consider two cases. First, assume there exists an index set $I \subseteq \{0, \ldots, m-1\}$ such that $|I| \geq \frac{1}{16}(n/p) \lg p$ and $\ell(b_i)$ is the same for all i in I. In this case the lower bound holds by Assumption L2 since the processor initially holding the set of b_i's indexed by I must examine each of these keys.

Now assume that no such index set I exists. Using Assumption L3, we find that at least $\frac{3}{4}m$ of the b_i's with $0 \leq i < m$ are initially located $\Omega(\lg p)$ hops away from any fixed processor. Hence $M_i' = \Omega(n \lg p)$, $0 \leq i < m$, and

$$\sum_{0 \leq k < m} M_k = \Omega(n^2 \lg p).$$

By averaging, we conclude that for some choice of k, $M_k = \Omega(n \lg p)$. Assumption L2 then implies that $\mathcal{A}$ must run for $\Omega((n/p) \lg p)$ steps in order to generate a total of $\Omega(n \lg p)$ messages. ■

5 Monotone Polygon Triangulation

A polygon is *monotone* if and only if it consists of two polygonal chains that are monotone with respect to the same line. (A polygonal chain C is *monotone* with respect to a line $\mathcal{L}$ if any line orthogonal to $\mathcal{L}$ intersects C in at most one point.) Let $\mathcal{P}$ be a monotone polygon consisting of two chains: the "upper" chain $U = \langle u_i : 0 \leq i < s \rangle$ and the "lower" chain $D = \langle d_i : 0 \leq i < t \rangle$, where $u_0 = d_{t-1}$ and $u_{s-1} = d_0$. A recent result of Atallah and Chen [3] gives an optimal hypercube algorithm for determining the line with respect to which a given polygon is monotone, if such a line exists. Their result allows us to assume that the given polygon is monotone with respect to the x-axis. (If not, a trivial transformation can be applied to ensure that this property holds.) Note that in a polygon that is monotone with respect to the x-axis, all the vertices on the upper chain (resp., lower chain) have distinct x-coordinates. A monotone polygon is *one-sided* if either its upper or lower chain consists of a single edge, called the *distinguished* edge.

Our algorithm for triangulating a monotone polygon follows the approach of Berkman et al. [4], Aggarwal et al. [1], and Goodrich [8]. The algorithm consists of two stages. In the first stage, we decompose the monotone polygon into one-sided monotone polygons. In the second stage, we triangulate the one-sided monotone polygons.

We assume a d-dimensional hypercube containing $2^d = n$ processors, $d \geq 1$. Local variable y at processor i will be referred to as $y[i]$. The input to our algorithm consists of two sets of real variables $x[i]$ and $y[i]$, $0 \leq i < 2^d$. Variables x and y at processors i in the range $0 \leq i < s$ (resp., $s \leq i < 2^d$) are the x- and y-coordinates of vertices $\langle u_i : 0 \leq i < s \rangle$ (resp., $\langle d_i : 0 \leq i < t \rangle$). For each i, $0 \leq i < 2^d$, let $r(i)$ (resp., $\ell(i)$) denote the integer j, $0 \leq j < 2^d$, such that the edge from $(x[i], y[i])$ to $(x[j], y[j])$ is a right (resp., left) edge in the triangulation of $\mathcal{P}$ (see Step 3 for the definitions of right and left edges). If no such integer exists, then let $r(i) = +\infty$ (resp., $\ell(i) = -\infty$). The output consists of integer variables $r[i] = r(i)$ and $\ell[i] = \ell(i)$, $0 \leq i < 2^d$.

Algorithm Triangulate

1. *Decompose $\mathcal{P}$ into one-sided monotone polygons as follows. Merge sequences U and D by the x-coordinates of their vertices. Let U_i (resp., D_i) be the subsequence of U (resp., D) that consists of vertices which lie between vertex d_i and vertex d_{i-1} (resp., u_i and u_{i+1}) in the merged list. Note that $\mathcal{U}_i = U_i \cup \{d_i, d_{i-1}\}$ (resp., $\mathcal{D}_i = D_i \cup \{u_i, u_{i+1}\}$) is a one-sided monotone polygon with (d_i, d_{i-1}) (resp., (u_i, u_{i+1})) as the distinguished edge.*

   Set $i'[i] = \min_{s \le j < 2^d}\{j : x[j] \le x[i] \le x[j-1]\}$, $(x'[i], y'[i]) = (x[i'[i]], y[i'[i]])$, and $(x''[i], y''[i]) = (x[i'[i]-1], y[i'[i]-1])$, $0 \le i < s$. Set $i'[i] = \min_{0 \le j < s}\{j : x[j] \le x[i] \le x[j+1]\}$, $(x'[i], y'[i]) = (x[i'[i]], y[i'[i]])$, and $(x''[i], y''[i]) = (x[i'[i]+1], y[i'[i]+1])$, $s \le i < 2^d$. Implementation: Bitonic merge, prefix operation. Running time: $O(d)$.

2. *Rotate each one-sided monotone polygon $\mathcal{U}_i$, $0 \le i < t$ (resp., $\mathcal{D}_i$, $0 \le i < s$), by the smallest possible angle around the first (smallest x-coordinate) endpoint of its distinguished edge so that the distinguished edge is parallel to x-axis.*

   Set $(x[i], y[i]) = \theta(x[i], y[i])$, where the function θ computes the coordinates of $(x[i], y[i])$ after a rotation around point $(x'[i], y'[i])$ by the smallest angle that makes $y''[i] = y'[i]$. Set $(x''[i], y''[i]) = \theta(x''[i], y''[i])$, $0 \le i < 2^d$. Implementation: Local arithmetic operations. Running time: $O(1)$.

3. *For each vertex q in a one-sided polygon $\mathcal{U}_i$ (resp., $\mathcal{D}_i$) compute "right" and "left" edges, as defined below.*

   - The *right edge* of q is the edge (q, r) where r is the vertex of $\mathcal{U}_i$ (resp., $\mathcal{D}_i$) with the smallest x-coordinate such that $x(q) < x(r)$ and $y(q) \ge y(r)$ (resp., $y(q) \le y(r)$).
   - The *left edge* of q is the edge (q, ℓ) where ℓ is the vertex of $\mathcal{U}_i$ (resp., $\mathcal{D}_i$) with the largest x-coordinate such that $x(r) \le x(q)$ and $y(q) > y(r)$ (resp., $y(q) \le y(r)$).

   (a) Set $r[i] = \min_{i<j}\{j : i'[j] = i'[i] \text{ and } y[i] \ge y[j]\}$, $0 \le i < s$. For any $r[i]$, $0 \le i < s$, that was not set in the previous step, set $r[i] = i'[i] - 1$. Set $\ell[i] = \max_{j<i}\{j : i'[j] = i'[i] \text{ and } y[i] > y[j]\}$, $0 \le i < s$. For any $\ell[i]$, $0 \le i < s$, that was not set in the previous step, set $\ell[i] = i'[i]$. Implementation: Algorithm ANSV run on variable y and segmented according to the variable i'. Running time: $O(d)$.

   (b) Set $r[i] = \max_{j<i}\{j : i'[j] = i'[i] \text{ and } y[i] \le y[j]\}$, $s \le i < 2^d$. For any $r[i]$, $s \le i < 2^d$, that was not set in the previous step, set $r[i] = i'[i] + 1$. Set $\ell[i] = \min_{i<j}\{j : i'[j] = i'[i] \text{ and } y[i] < y[j]\}$, $s \le i < 2^d$. For any $\ell[i]$, $s \le i < 2^d$, that was not set in the previous step, set $\ell[i] = i'[i]$. Implementation: Algorithm ANSV run on variable y and segmented according to the variable i'. Running time: $O(d)$.

Algorithm Triangulate runs in $O(\lg n)$ time on an n-processor hypercube. The correctness of the algorithm follows from the work of Berkman et al. [4]. Furthermore, Berkman et al. [4] show how to reduce the problem of merging two sorted lists of length n to the problem of triangulating a monotone polygon of size n. In view of Theorem 1 (see Section 4), this reduction implies that an $\Omega(n)$-processor hypercube is needed to triangulate a monotone polygon in $O(\lg n)$ time. Thus, our normal hypercube algorithm is processor-optimal as well as time-optimal.

6 Building a Cartesian Tree

Let the Cartesian tree of a sequence $W = \langle w_i : 0 \leq i < n \rangle$ be defined as in Section 1. Berkman et al. [4] prove that the parent of any element w_i in the Cartesian tree is the larger of w_i's left and right matches. Thus, a straightforward application of the ANSV algorithm yields an $O(\lg n)$-time n-processor normal hypercube algorithm for constructing a Cartesian tree.

7 Concluding Remarks

In this paper, we have presented an $O(\lg n)$-time n-processor normal hypercube algorithm for the ANSV problem. It would be interesting to extend our techniques to obtain efficient normal hypercube algorithms for other fundamental problems.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. K. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.

[2] A. Aggarwal, D. Kravets, J. K. Park, and S. Sen. Parallel searching in generalized Monge arrays with applications. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 259–268, 1990.

[3] M. J. Atallah and D. Z. Chen. Optimal parallel hypercube algorithms for polygon problems. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 208–215, 1993.

[4] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

[5] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[6] D. Z. Chen. Efficient geometric algorithms in the EREW-PRAM. In *Proceedings of the 28th Annual Allerton Conference on Communication, Control, and Computing*, pages 818–827, 1990.

[7] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences*, 47:501–548, 1993.

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

[9] J. F. JáJá and K. W. Ryu. Optimal algorithms on the pipelined hypercube and related networks. *IEEE Transactions on Parallel and Distributed Systems*, 4:582–591, 1993.

[10] C. P. Kruskal. Searching, merging, and sorting. *IEEE Transactions on Computers*, C-32(10):942–946, 1983.

[11] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, volume 1. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[12] E. W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–117, 1992.

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

[14] J. Vuillemin. A unified look at data structures. *Communications of the ACM*, 23:229–239, 1980.

[15] C. Yap. Parallel triangulation of a polygon in two calls to the trapezoidal map. *Algorithmica*, 3:279–288, 1988.