Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem [*]

Xin He [†]

Department of Computer Science and Engineering

State University of New York at Buffalo

Buffalo, NY, U.S.A.

Chun-Hsi Huang [‡]

Department of Computer Science and Engineering

State University of New York at Buffalo

Buffalo, NY, U.S.A.

March 26, 2001

[*]Research supported in part by NSF Grant CCR-9912418. Computational resources and technical support provided by Center for Computational Research at SUNY-Buffalo (UB CCR).

[†]Email: xinhe@cse.buffalo.edu

[‡]Email: ch25@cse.buffalo.edu

Title: Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem

Corresponding Author:

XIN HE

221 Bell Hall

Department of Computer Science and Engineering

State University of New York at Buffalo

Buffalo, NY 14260-2000

U.S.A.

PHONE: (716) 645-3180 x 128

FAX: (716) 645-3464

EMAIL: xinhe@cse.buffalo.edu

Abstract

We present a BSP (Bulk Synchronous Parallel) algorithm for solving the All Nearest Smaller Values Problem (ANSVP), a fundamental problem in both graph theory and computational geometry. Our algorithm achieves optimal sequential computation time and uses only three communication supersteps. In the worst case, each communication phase takes no more than an $(\frac{n}{p} + p)$-relation, where p is the number of the processors. In addition, our average-case analysis shows that, on random inputs, the expected communication requirements for all three steps are bounded above by a p-relation, which is independent of the problem size n. Experiments have been carried out on an SGI Origin 2000 with 32 R10000 processors and a SUN Enterprise 4000 multiprocessing server supporting 8 UltraSPARC processors, using the MPI libraries. The results clearly demonstrate the communication efficiency and load balancing for computation.

Keywords: BSP, ANSVP, Parallel Algorithm

1 Introduction

1.1 The BSP Model

Interprocessor communication has been shown to become a major bottleneck for parallel algorithm performance. Parallel algorithms should seek to minimize both computation and communication time to be considered practical. For PRAM model, one fundamental assumption is that communication between processors are via common memory, and each memory access consumes one unit computation time. For practical purposes, this assumption does not lead to accurate performance prediction.

The Bulk Synchronous Parallel (BSP) model is a model for general-purpose, architecture-independent parallel programming, introduced by Valiant [17] and McColl [13]. The BSP model consists of three components: a set of processors, each with a local memory, a communication network, and a mechanism for globally synchronizing the processors. All existing parallel computers are BSP computers in this sense, but they have very different performance characteristics. These differences are captured by three parameters (in addition to a fourth, the processor speed of the computer), p: the number of processors, g: the ratio of communication throughput to processor throughput, and L: the time required to barrier synchronize all or part of the processors. A BSP program proceeds as a series of *supersteps*. In each superstep, a processor may operate only on values stored in local memory. Values sent through the communication network are not guaranteed to arrive until the end of the current superstep. Parameters p, g, and L can be used to estimate the running time of a BSP program whose communication behavior is known. Similarly, a BSP program that has access to these parameters for the machine it is running on can use them to choose between different algorithms. Our purpose is to design a scalable BSP algorithm, minimizing the number of communication supersteps as well as the local computation time. If the best possible sequential algorithm for a given problem takes $T_s(n)$ time, then ideally we would like to design a BSP algorithm using $O(1)$ communication supersteps, preferably with total message size bounded by $O(\frac{n}{p})$ in each superstep, and $O(\frac{T_s(n)}{p})$ total local computation time.

Throughout the paper, we assume $\frac{n}{p} = \Omega(p)$, a condition true for all commercially available parallel machines and practical problem sizes.

1.2 All Nearest Smaller Values Problem (ANSVP) and its Applications

The all nearest smaller values problem is defined as follows: Let $A = (a_1, a_2, \cdots, a_n)$ be an array of elements from a totally ordered domain. For each a_j, $1 \leq j \leq n$, find the nearest element to the left of a_j and the nearest element to the right of a_j that are less than a_j.

A typical application of the ANSVP is the merging of two sorted lists [3, 12]. Let $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_n)$ be two increasing arrays to be merged. The reduction to the ANSVP is done by constructing an array $C = (a_1, a_2, \cdots, a_n, b_n, b_{n-1}, \cdots, b_1)$ and then solving the ANSVP with respect to C. If b_y is the right match of a_x, the location of a_x in the merged list is $x + y$. The locations of b_x's can be found similarly. In addition, the ANSVP is fundamental in that all the following problems have been shown to be reducible to it [1]:

Monotone Polygon Triangulation

A polygon P is considered monotone with respect to a line L if it can be split into two polygonal chains A and B, both of which are monotone with respect to L. The two chains share the highest and lowest vertices. We are asked to triangulate such polygons.

Finding Range Minimum

Preprocess an array of real numbers $A = (a_1, \cdots, a_n)$ so that given x, y, $1 \leq x \leq y \leq n$, it takes constant time to find the minimum element in the subarray $A_{x,y} = (a_x, \cdots, a_y)$.

Binary Tree Reconstruction

Given the inorder and preorder (postorder) traversals of a binary tree, we are asked to reconstruct the binary tree from these traversals. In other words, the problem is to determine each node's parent.

Parenthesis Matching

A sequence of parentheses is *balanced* if every left (right, resp.) parenthesis has a matching right (left, resp.) parenthesis. Let a balanced sequence of parentheses be represented by $h_1, h_2, \cdots, h_n$, where h_k represents the kth parenthesis. It is required to determine the matching parenthesis of each parenthesis.

Parallel ANSV algorithm also plays an important role in solving the following problems: *string matching* [4], *forest matching* [10], *computing all nearest neighbors in convex polygons* [15], and *computing the convex hull of a sorted point set* [1].

A work-optimal CRCW PRAM algorithm for the ANSVP using $O(\log \log n)$ time was proposed in [1]. Katajainen [9] explored the parallel complexity of the ANSVP on a variation of PRAM, DMM (Distributed Memory Machine, also referred to as Local Memory PRAM). Kravets and Plaxton [11] presented a parallel algorithm on the hypercube. These algorithms have not been implemented [1, 9, 11]. Our main contribution is to provide the first practical and portable general purpose parallel algorithm for solving the ANSVP with provable communication efficiency in three BSP supersteps and optimal sequential computation time. The portability and scalability have been experimentally justified in Section 4.

The rest of this paper is organized as follows: In Section 2, we present the BSP algorithm. The worst-case and average-case BSP cost analyses are given in Section 3. Section 4 provides the experimental results on the SGI Origin 2000 and Sun Enterprise 4000 parallel machines. Section 5 concludes this paper.

2 The BSP Algorithm

Given a sequence $A = (a_1, a_2, \cdots, a_n)$ and a p-processor BSP computer with any communication media (shared memory or any interconnection network), each P_i $(1 \leq i \leq p)$ stores $(a_{\frac{n}{p}(i-1)+1}, a_{\frac{n}{p}(i-1)+2}, \cdots, a_{\frac{n}{p}i})$. For simplicity, we assume that the elements in the sequence are distinct. We define the nearest smaller value to the right of an element to be its *right match*. The ANSVP can be solved sequentially with linear time using a straightforward stack approach. To find the right matches of the elements, we scan the input, keep the elements for which no right match has been found on a stack, and report the current element as a right match for those elements on the top of the stack that are larger than the current element. The left matches can be found similarly. For brevity and without loss of generality, we will focus on finding the right matches.

Some definitions are given below. Throughout the rest of this paper, we use i $(1 \leq i \leq p)$ for processor related indexing and j $(1 \leq j \leq n)$ for array element related indexing.

- For any j:
  - $\text{rm}(j)$ ($\text{lm}(j)$, resp.) $\overset{\text{def}}{=}$ the index of the right (left, resp.) match of a_j.
  - $\text{rmp}(j)$ ($\text{lmp}(j)$, resp.) $\overset{\text{def}}{=}$ the index of the processor containing $a_{\text{rm}(j)}$ ($a_{\text{lm}(j)}$, resp.).
- For any i:
  - $\min(i) \overset{\text{def}}{=}$ the index (in A) of the smallest element in P_i.
  - $\text{rm\_min}(i)$ ($\text{lm\_min}(i)$, resp.) $\overset{\text{def}}{=}$ the index of the right (left, resp.) match of $a_{\min(i)}$ with respect to the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$.
  - $\wp_i \overset{\text{def}}{=} \{P_x |\ \text{rm\_min}(x) = i\}$; $\varphi_i \overset{\text{def}}{=} \{P_x |\ \text{lm\_min}(x) = i\}$.

Based on the above definitions, we observe that $P_{\text{rmp}(\min(i))} = P_{\text{rm\_min}(i)}$ ($P_{\text{lmp}(\min(i))} = P_{\text{lm\_min}(i)}$, resp.) Next we prove a lemma used in our BSP algorithm.

Lemma 2.1 *On a p-processor BSP computer, for any i, if $a_{\mathrm{rm}(\min(i))}$ exists and $\mathrm{rmp}(\min(i)) \neq i+1$, then there exists a unique processor $P_{k(i)}$, $i < k(i) < \mathrm{rmp}(\min(i))$, such that $\mathrm{lmp}(\min(k(i))) = i$ and $\mathrm{rmp}(\min(k(i))) = \mathrm{rmp}(\min(i))$. (Symmetrically, for any i, if $a_{\mathrm{lm}(\min(i))}$ exists and $\mathrm{lmp}(\min(i)) \neq i-1$, then there exists a unique processor $P_{k'(i)}$, $\mathrm{lmp}(\min(i)) < k'(i) < i$, such that $\mathrm{lmp}(\min(k'(i))) = \mathrm{lmp}(\min(i))$ and $\mathrm{rmp}(\min(k'(i))) = i$).*

Proof: We show that P_s, where $a_{\min(s)} = \min\{a_{\min(i+1)}, a_{\min(i+2)}, \cdots, a_{\min(\mathrm{rmp}(\min(i))-1)}\}$, is the unique processor described in the lemma. For any s' with $i+1 \leq s' < s$, $\mathrm{rmp}(\min(s'))$ must be $\leq s$. Similarly, for any s' with $s+1 \leq s' < \mathrm{rmp}(\min(i))$, $\mathrm{lmp}(\min(s'))$ must be $\geq s$. This leaves P_s the only candidate processor. We can easily infer that $a_{\min(s)} > a_{\min(i)} > a_{\min(\mathrm{rmp}(\min(i)))}$. Since $a_{\min(s)}$ is the smallest element among those in $P_{i+1}, \cdots, P_{\mathrm{rmp}(\min(i))-1}$, we conclude that P_s is the unique processor $P_{k(i)}$ specified in Lemma 2.1. (The symmetric part can be proved similarly.) $\square$

We next outline our algorithm. To begin with, all processors sequentially find the right matches for their local elements, using the stack approach. Those matched elements require no interprocessor communication. We therefore focus on those elements which are not yet matched. The general idea is to find the right matches for those not-yet-matched elements by reducing the original ANSVP to $2p$ smaller "special" ANSVPs, and solve them in parallel.

Next we compute the right and left matches for all $a_{\min(i)}$'s. To do this, we first solve the ANSVP with respect to the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$. Then, for each processor P_i, we define four sequences, $\mathrm{Seq1}_i$, $\mathrm{Seq2}_i$, $\mathrm{Seq3}_i$ and $\mathrm{Seq4}_i$ as follows:

- If $a_{\mathrm{rm}(\min(i))}$ does not exist, then $\mathrm{Seq1}_i$ and $\mathrm{Seq2}_i$ are undefined.
- If $a_{\mathrm{rm}(\min(i))}$ exists and $\mathrm{rmp}(\min(i)) = i+1$, then:

  $\mathrm{Seq1}_i = (a_{\min(i)}, \cdots, a_{\frac{n}{p}i})$, $\mathrm{Seq2}_i = (a_{\frac{n}{p}i+1}, \cdots, a_{\mathrm{rm}(\min(i))})$.
- If $a_{\mathrm{rm}(\min(i))}$ exists and $\mathrm{rmp}(\min(i)) > i+1$, let $P_{k(i)}$ be the unique processor specified in

Lemma 2.1. Then: $\mathrm{Seq1}_i = (a_{\min(i)}, \cdots, a_{\mathrm{lm}(\min(k(i)))})$, $\mathrm{Seq2}_i = (a_{\mathrm{rm}(\min(k(i)))}, \cdots, a_{\mathrm{rm}(\min(i))})$.

- If $a_{\mathrm{lm}(\min(i))}$ does not exist, then $\mathrm{Seq3}_i$ and $\mathrm{Seq4}_i$ are undefined.

- If $a_{\mathrm{lm}(\min(i))}$ exists and $\mathrm{lmp}(\min(i)) = i - 1$, then:

  $\mathrm{Seq3}_i = (a_{\frac{n}{p}(i-1)+1}, \cdots, a_{\min(i)})$, $\mathrm{Seq4}_i = (a_{\mathrm{lm}(\min(i))}, \cdots, a_{\frac{n}{p}(i-1)})$.

- If $a_{\mathrm{lm}(\min(i))}$ exists and $\mathrm{lmp}(\min(i)) < i - 1$, let $P_{k'(i)}$ be the unique processor specified in Lemma 2.1. Then: $\mathrm{Seq3}_i = (a_{\mathrm{rm}(\min(k'(i)))}, \cdots, a_{\min(i)})$, $\mathrm{Seq4}_i = (a_{\mathrm{lm}(\min(i))}, \cdots, a_{\mathrm{lm}(\min(k'(i)))})$.

Note that $\mathrm{Seq1}_i$ and $\mathrm{Seq3}_i$, if they exist, always reside on P_i, $\mathrm{Seq2}_i$, if it exists, always resides on $P_{\mathrm{rmp}(\min(i))}$, and $\mathrm{Seq4}_i$, if it exists, always resides on $P_{\mathrm{lmp}(\min(i))}$.

The following two lemmas 2.2 and 2.3 specify how to find the right matches for all unmatched elements. Detailed proofs can be found in [1].

Lemma 2.2 *The right matches of all not-yet-matched elements in* $\mathrm{Seq1}_i$ *lie in* $\mathrm{Seq2}_i$. *The right matches of all not-yet-matched elements in* $\mathrm{Seq4}_i$*, except its first element, lie in* $\mathrm{Seq3}_i$.

Each processor P_i therefore is responsible for identifying right matches for not-yet-matched elements in $\mathrm{Seq1}_i$ and $\mathrm{Seq4}_i$. Again, we apply the sequential algorithm at each processor P_i with respect to the two concatenated sequences, $\mathrm{Seq1}_i \| \mathrm{Seq2}_i$ and $\mathrm{Seq4}_i \| \mathrm{Seq3}_i$.

Lemma 2.3 *All elements will be right-matched after the above-mentioned 2p special ANSVPs are solved in parallel.*

We need the following lemma:

Lemma 2.4 *1. Suppose that* $\wp_i = \{P_{x_1}, P_{x_2}, \cdots, P_{x_t}\}$ *where* $x_1 < x_2 < \cdots < x_t$. *Then:*

$\mathrm{Seq2}_{x_1} = (a_{\mathrm{rm}(\min(x_2))}, \cdots, a_{\mathrm{rm}(\min(x_1))})$, $\mathrm{Seq2}_{x_2} = (a_{\mathrm{rm}(\min(x_3))}, \cdots, a_{\mathrm{rm}(\min(x_2))})$,

$\cdots$, $\mathrm{Seq2}_{x_t} = (a_{\frac{n}{p}(i-1)+1}, \cdots, a_{\mathrm{rm}(\min(x_t))})$.

2. *Suppose that* $\varphi_i = \{P_{y_1}, P_{y_2}, \cdots, P_{y_s}\}$ *where* $y_1 < y_2 < \cdots < y_s$. *Then:*

   $\text{Seq4}_{y_1} = (a_{\text{lm}(\min(y_1))}, \cdots, a_{\text{lm}(\min(y_2))})$, $\text{Seq4}_{y_2} = (a_{\text{lm}(\min(y_2))}, \cdots, a_{\text{lm}(\min(y_3))})$,

   $\cdots$, $\text{Seq4}_{y_s} = (a_{\text{lm}(\min(y_s))}, \cdots, a_{\frac{n}{p}i})$.

Proof: We only prove Statement 1. The proof of Statement 2 is similar. First observe that, for any $P_x, P_y \in \wp_i$, $x < y$ implies $a_{\min(x)} < a_{\min(y)}$ and $k(x) \leq y$. Based on these observations, we have $k(x_l) = x_{l+1}$ for $1 \leq l < t$ and $x_t = i - 1$. The lemma follows from the definition of Seq2. $\square$

The algorithm below finds the right matches and is therefore denoted as Algorithm $ANSV_r$. It is described following BSP programming style. In each step, we also mention the standard MPI libraries we use. (MPI is a standard specification for message passing and will be described in more details in Section 4.)

Algorithm $ANSV_r$:

Input: A partitioned into p subsets of contiguous elements. Each processor stores one subset.

(MPI_Scatter)

Output: The right match of each a_i is computed and stored in the processor containing a_i.

1. Each P_i sequentially solves the $ANSV_r$ problem with respect to its local subset.

2. (a) Each P_i computes its local minimum $a_{\min(i)}$.

   (b) All $a_{\min(i)}$'s are globally communicated. (Hence each P_i has the array $A_{\min}$.)

   (MPI_Allgather, MPI_Barrier)

3. Each P_i solves the $ANSV_r$ and the $ANSV_l$ problems with respect to $A_{\min}$; and identify the sets $\wp_i$ and φ_i.

4. Each P_i computes $a_{\text{rm}(\min(x))}$ for every $P_x \in \wp_i$ and $a_{\text{lm}(\min(y))}$ for every $P_y \in \varphi_i$.

5. Each P_i determines Seq1_i, Seq3_i and receives Seq2_i, Seq4_i as follows:

(a) Each P_i computes the unique $k(i)$ and $k'(i)$ (as in Lemma 2.1), if exist, and determines Seq1_i and Seq3_i.

(b) Each P_i determines Seq2_x for every $P_x \in \wp_i$, and Seq4_y for every $P_y \in \varphi_i$ (as in Lemma 2.4).

(c) Each P_i sends Seq2_x, for every $P_x \in \wp_i$, to P_x and Seq4_y, for every $P_y \in \varphi_i$, to P_y.

(MPI_Send, MPI_Recv, MPI_Barrier)

6. (a) Each P_i finds the right matches for the unmatched elements in Seq1_i and Seq4_i

(b) Each P_i collects the matched Seq4_y's from all P_y's$\in \varphi_i$.

(MPI_Send, MPI_Recv, MPI_Barrier)

3 Complexity Analysis

3.1 Computation and Communication Complexities

We use the term *h-relation* to denote a routing problem where each processor has at most h words of data to send to other processors and each processor is also due to receive at most h words of data from other processors. In each BSP superstep, if at most w arithmetic operations are performed by each processor and the data communicated forms an *h-relation*, then the cost of this superstep is $w + h * g + L$ (the parameters g and L are as defined in Section 1.1). The cost of a BSP algorithm using S supersteps is simply the sum of the costs of all S supersteps:

$$BSP\ cost = comp.\ cost + comm.\ cost + \ synch.\ cost = W + H * g + L * S$$

where H is the sum of the maxima of the *h-relation*s in each superstep and W is the sum of the maxima of the local computations in each superstep.

Table 1: BSP Cost Breakdown of the *Algorithm* $ANSV_r$

	Cost		
Step	comp.	comm.	synch.
1	$T_s(\frac{n}{p})$		
2(a)	$T_\theta(\frac{n}{p})$		
2(b)		pg	L
3	$2T_s(p)$		
4	$\max_i\{T_\theta(\frac{n}{p}+\lvert\varphi_i\rvert)+T_\theta(\frac{n}{p}+\lvert\wp_i\rvert)\}$		
5(a)	$\max_i\{T_\theta(\text{rm\_min}(i)-\text{lm\_min}(i))\}$		
5(b)	$O(\frac{n}{p})$		
5(c)		$\max_i\{(\Sigma_{P_x\in\wp_i}\lvert\text{Seq}2_x\rvert + \Sigma_{P_y\in\varphi_i}\lvert\text{Seq}4_y\rvert)g\}$	L
6(a)	$\max_i\{T_s(\lvert\text{Seq}1_i\rvert + \lvert\text{Seq}2_i\rvert) + T_s(\lvert\text{Seq}4_i\rvert+\lvert\text{Seq}3_i\rvert)\ \}$		
6(b)		$\max_i\{\Sigma_{P_x\in\varphi_i}\lvert\text{Seq}4_x\rvert g\}$	L

Here we assume the sequential computation time for the $ANSV_r$ problem of input size n is $T_s(n)$, and the sequential time for finding the minimum of n elements is $T_\theta(n)$. Then the BSP cost breakdown of Algorithm $ANSV_r$ can be derived as in Table 1.

Since $\frac{n}{p}=\Omega(p)$ and $T_s(n)=T_\theta(n)=O(n)$, the computation time in each step is obviously linear in the local data size, namely $O(\frac{n}{p})$. Steps 2 (b), 5 (c) and 6 (b) involve communication. Thus the algorithm takes three supersteps. Based on Lemma 2.4 and the fact that $|\varphi_i|+|\wp_i|\le p$, the communication steps 5 (c) and 6 (b) can each be implemented by an $(\frac{n}{p}+p)$-relation. Therefore we have:

Theorem 3.1 *The ANSVP with input size n can be solved on a p-processor BSP machine in three supersteps using linear local computation time and at most an $(\frac{n}{p}+p)$-relation in each communication phase, provided $p\le n/p$.*

3.2 Average-Case Communication Complexity

The communication complexity analysis given in the last subsection is for the worst case. In our experiments, we have observed that, for some fixed p, the communication costs do not seem to depend on the input size n. Since applications using the ANSVP tend to have randomly generated inputs, finding the average-case communication complexity actually gives more precise communication time estimates. Although the communication cost of step 2 (b) of Algorithm $ANSV_r$ is always $\Theta(pg)$, we will show in this section that, on random inputs, the expected communication costs of steps 5 (c) and 6 (b) are actually both $O(g)$, much smaller than the communication cost $O(\frac{n}{p}g)$ as indicated in the worst case analysis. Thus the total average communication cost of Algorithm $ANSV_r$ is $\Theta(pg)$. Since p is normally far smaller than n/p, the average communication cost of our algorithm is much better than that indicated in the worst-case analysis.

We focus on step 5 (c). The analysis of step 6 (b) is similar. In step 5 (c), processor P_i sends out Seq2_x for each $P_x \in \wp_i$. We will show that the expected size of Seq2_x and the expected size of $\wp_i$ are both $O(1)$. This will establish our claim.

Consider the $ANSV_r$ problem on the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$. For convenience, we define $\text{rm\_min}(i) = p+1$ if $a_{\min(i)}$ has no right match. Note that, for $i+1 \leq u \leq p$, $\text{rm\_min}(i) = u$ if and only if $a_{\min(i+1)}, \cdots, a_{\min(u-1)}$ are all greater than $a_{\min(i)}$, and $a_{\min(u)} < a_{\min(i)}$. Since the input is random, we have:

$$P_r[\text{rm\_min}(i) = u] = \frac{1}{2^{u-i}}, \text{ for } i+1 \leq u \leq p \quad (1)$$

$$P_r[\text{rm\_min}(i) = p+1] = \frac{1}{2^{p-i}} \quad (2)$$

Suppose that $\text{rm\_min}(i) = u$ for some $i < u \leq p$. Then $a_{\min(k(i))}$ is the minimum value among

$a_{\min(i+1)}, \cdots, a_{\min(u-1)}$. Since the input is random, the probability of all possibilities are equal. Thus, for $i < v < u \leq p$, we have:

$$P_r[k(i) = v \mid \text{rm\_min}(i) = u] = \frac{1}{u-i-1} \ .$$

The expected value of $|\text{Seq2}_i|$ can be described as:

$$E(|\text{Seq2}_i|) = \sum_{u=i+1}^{p+1} P_r[\text{rm\_min}(i) = u] \cdot E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u) \tag{3}$$

Assume that $\text{rm\_min}(i) = u$ and $k(i) = v$. Suppose that the right match of $a_{\min(u)}$ is the t-th ($1 \leq t \leq n/p$) element in the subarray contained in P_u, and the right match of $a_{\min(v)}$ is the x-th ($1 \leq x \leq t$) element in the subarray contained in P_u. Then $|\text{Seq2}_i| = t - x + 1$. Thus:

$$\begin{aligned} E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u \mid k(i) = v) &= \sum_{t=1}^{\frac{n}{p}} \frac{1}{2^t}(\sum_{x=1}^{t-1} \frac{1}{2^x}(t-x+1) + \frac{1}{2^{t-1}} \cdot 1) \\ &= \sum_{t=1}^{\frac{n}{p}} \frac{1}{2^t}(t - 1 + \frac{1}{2^{t-1}}) = \frac{5}{3} - \frac{\frac{n}{p}+1}{2^{\frac{n}{p}}} - \frac{1}{3}\frac{1}{2^{\frac{2n}{p}-1}} < \frac{5}{3} \end{aligned}$$

This gives:

1. $E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = p+1) = 0$,

2. $E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = i+1)$

   $= \sum_{s=1}^{\frac{n}{p}} s \cdot P_r[$ the right match of $a_{\min(i)}$ is the sth element in the subarray of $P_{i+1}]$

   $= \sum_{s=1}^{\frac{n}{p}} s \cdot \frac{1}{2^s} = 2(1 - \frac{1}{2^{\frac{n}{p}}} - \frac{\frac{n}{p}}{2^{\frac{n}{p}+1}}) < 2$, and

3. $E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u)$, $i + 2 \leq u \leq p$, can be derived as follows:

$$\begin{aligned} E(|\mathrm{Seq2}_i| \mid \mathrm{rm\_min}(i)=u) &= \sum_{s=i+1}^{u-1} \frac{1}{u-i-1} \cdot E(|\mathrm{Seq2}_i| \mid \mathrm{rm\_min}(i)=u \mid k(i)=v) \\ &\leq \sum_{s=i+1}^{u-1} \frac{1}{u-i-1} \cdot \frac{5}{3} = \frac{5}{3} \end{aligned}$$

Therefore,

(1) $E(|\mathrm{Seq2}_p|) = 0$.

(2) $E(|\mathrm{Seq2}_{p-1}|) \leq \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2 = 1$.

(3) $E(|\mathrm{Seq2}_i|)$, $1 \leq i \leq p-2$, can be derived as follows:

$$\begin{aligned} E(|\mathrm{Seq2}_i|) &= \sum_{u=i+1}^{p+1} P_r[\mathrm{rm\_min}(i)=u] \cdot E(|\mathrm{Seq2}_i| \mid \mathrm{rm\_min}(i)=u) \\ &\leq \frac{1}{2^{p-i}} \cdot 0 + \frac{1}{2} \cdot 2 + \sum_{u=i+2}^{p} \frac{1}{2^{u-i}} \cdot \frac{5}{3} = 1 + (\frac{1}{2} - \frac{1}{2^{p-i}}) \cdot \frac{5}{3} \leq 1 + \frac{1}{2} \cdot \frac{5}{3} = \frac{11}{6} \end{aligned}$$

Next, we compute the expected size of $\wp_i$. Note that, for $1 \leq x < i$, processor $P_x \in \wp_i$ if and only if $a_{\min(x)} > a_{\min(i)}$ and $a_{\min(l)} > a_{\min(x)}$ for all $x+1 \leq l \leq i-1$. Thus:

$E(|\wp_i|) = \sum_{x=1}^{i-1} = P_r[P_x \in \wp_i] = \sum_{x=1}^{i-1} \frac{1}{2^{i-x}} = 1 - \frac{1}{2^i} \leq 1$.

Hence, the expected communication requirements for steps 5 (c) and 6 (b) are both bounded above by $\frac{11}{6} \times 1 = \frac{11}{6}$. Step 2 (b) therefore becomes the dominant step in communication requirement, which takes simply a p-relation.

This concludes the following theorem :

Theorem 3.2 *The ANSVP with input size n can be solved on a p-processor BSP machine in 3 supersteps using linear local computation time, with the average-case communication requirement* *bounded above by a p-relation for each communication phase.*

4 Experimental Results

To demonstrate the practical relevance, we have implemented Algorithm $ANSV_r$ on UB CSE department's Sun Enterprise 4000 multiprocessing server supporting 8 UltraSPARC processors and UB CCR's SGI Origin 2000 with 32 R10000 processors.

Our program is about 350 lines, written in C, using the MPI (Message Passing Interface) library [14] for interprocessor communication. MPI is a standard specification for message passing libraries. MPT (on SGI Origin 2000) and MPICH (on Sun Enterprise 4000) are both portable implementations of the full MPI specification for a wide variety of parallel and distributed environments, including parallel computers, clusters of workstations, integrated distributed environments (computational grids) and shared-memory symmetric multiprocessors. Yet both MPT and MPICH provide unified system calls for communication between processes via the various communication media. Programming with MPI libraries makes it possible for developing portable and efficient parallel programs. In our experiment, the codes on SGI Origin 2000 and Sun Enterprise 4000 are exactly the same. Since MPI interface provides wide support for most commercially available parallel machines, and current systems that conform to the BSP computer model include networks of workstations, distributed memory processor arrays, and shared memory multiprocessors, the Algorithm $ANSV_r$ can easily be ported to any commercially available machines.

MPT, MPICH and MPICH-GM (used on CCR's Sun Myrinet Cluster) are all compatible with MPICH-G2, another portable implementation of the full MPI specification, supported by Globus [6]. (The Globus project [6] is developing fundamental technologies needed to build computational grids.) To facilitate future expansion, our code is thus based on the MPI. Another portable communication system, BSPLib [7], has been popularly used for BSP programming model [2, 16].

Algorithm $ANSV_r$ can be implemented with BSPLib in a similar manner.

We use MPI_Bcast for broadcasting. Some collective communication routines, such as MPI_Allreduce, MPI_Allgather, are also used for global communication. MPI_Scatter and MPI_Gather are used to distribute input data and collect results. Due to the property of Algorithm $ANSV_r$ that data items routed in each superstep are always in contiguous locations, only MPI_Send and MPI_Recv are used for message passing.

Recently, another bridging model, CGM (Coarse Grained Multicomputer), by Dehne [5], argued that in a message passing multiprogramming environment, sending many smaller messages in one communication round requires a lot more enveloping overhead than sending a single, packed, maximal-size message [5]. In our implementation, the actual message passing is carried out by MPI_Send and MPI_Recv since all data elements to be routed are actually in contiguous locations. MPI also provides MPI_Pack and MPI_Unpack for packing elements, but we don't find them useful in our implementation. Timings were done by invoking MPI_WTime routine, which is intended to be a high-resolution, elapsed (or wall) clock. Throughout this section, we use $ANSV(n, p)$ to denote the parallel running time of the Algorithm $ANSV_r$, where n and p stand for the input size and the number of processors respectively.

4.1 On SGI Origin 2000

We investigated the algorithm's behavior on UB CCR's SGI Origin 2000 machine, using up to 32 processors. The experiment was done on input sizes from 0.1 to 8 millions. Each data point presented was obtained as the average of 5 test runs, each on a different randomly generated array.

Figures 1 and 2 depict the parallel running times when input sizes range from 0.1 to 8 millions.

The speedups achieved when the input size is above 1 million are almost linear as shown in Fig 1. When input size is between 0.1 and 0.8 millions, synchronization overhead starts to take up a

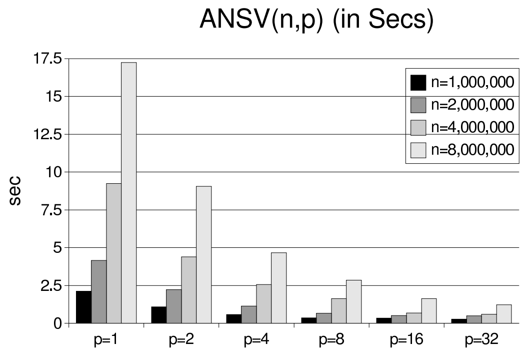

Figure 1: SGI Origin 2000 running times of Algorithm $ANSV_r$ (1)

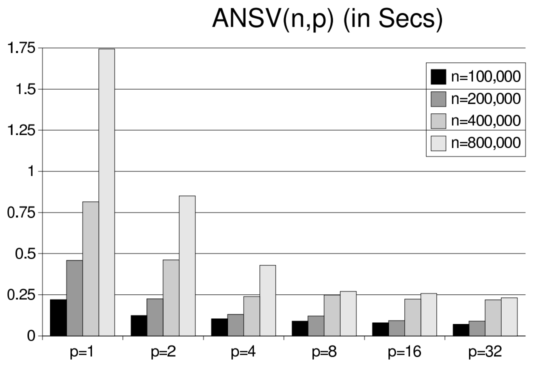

Figure 2: SGI Origin 2000 running times of Algorithm $ANSV_r$ (2)

larger portion of total running time. As the number of processors increases to 32 and the input size is 0.1 million, almost 37.5% (0.036 secs/0.096 secs) of parallel running times are contributed by barrier synchronization. This is partly because of the high cost of barrier synchronization for this machine [8], and partly because of the extremely small, actually < 2, hidden constant of the linear-time sequential stack algorithm. When 16 processors are used on 0.1 million data items, barrier synchronization uses about 27.2% (0.018 secs/0.066 secs) of the total execution time. However, when only 4 processors are used, this ratio significantly reduces to about 4.6% (0.00468 secs/0.09988 secs).

4.2 On Sun Enterprise 4000

The timings on Sun Enterprise 4000 multiprocessing server were made under time sharing. However, all experiments were carried out when system CPU idle time was at least 80%. Fig. 3 shows the average running times for input sizes of 0.84 and 4.2 millions on up to 7 processors. When two processors are used, the speedup is about 1.6, approximately 80% of the optimal speedup. When the number of processors scales to 7 (during our experiment, the 8th processor had been constantly occupied by some CPU-intensive jobs, thus was not included in our experiment), the speedup is about 4, which is about 55% of optimal speed-up. (Here the p-processor speedup is measured based on ANSV$(n,1)$/ANSV(n,p)).

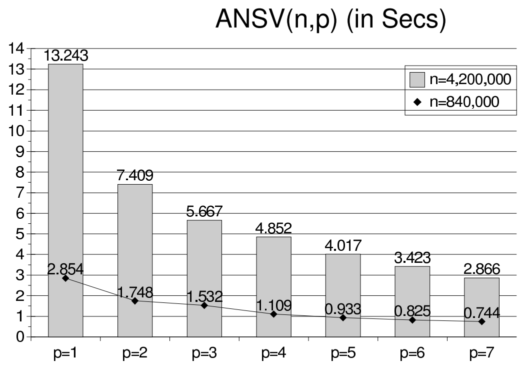

Figure 3: Sun Enterprise 4000 running times of Algorithm $ANSV_r$

5 Conclusion

Long communication latency and the overhead of synchronization tend to be the two critical factors that greatly affect the speedup of a parallel algorithm on present multiprocessor architectures. The Algorithm $ANSV_r$ is intended to minimize both. The parallel algorithm design is based on BSP programming model, which conforms to most of the commercially available parallel machines. Experiments on SGI Origin 2000 and Sun Enterprise 4000 demonstrate good speedups, which makes it clear that practically efficient parallel algorithms for various problems in both graph theory and computational geometry, as described in this paper, are indeed achievable. Two portable MPI implementations, MPICH and MPT, are used in our experiments to justify the portability of our algorithm.

6 Acknowledgement

The authors would like to thank the referees for their suggestions that considerably improved the presentation.

References

[1] Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms*, 14:344–370, 1993.

[2] Rob H. Bisseling. Basic Techniques for Numerical Linear Algebra on Bulk Synchronous Parallel Computers. In Lubin Vulkov, Jerzy Waśniewski, and Plamen Yalamov, editors, *Workshop Numerical Analysis and its Applications 1996*, volume 1196 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, Berlin, 1997.

[3] A. Borodin and J.E. Hopcroft. Routing, Merging and Sorting on Parallel Models of Computation. *J. Comput. System Sci.*, 30:130–145, 1985.

[4] D. Breslauer and Z. Galil. An Optimal $O(\log \log n)$ Time Parallel String Matching Algorithm. *SIAM J. Computing*, 19:1050–1058, 1990.

[5] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. In *Proc. 9th ACM Annual Computational Geometry*, 1993, 298-307.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[7] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[8] Jonathan M. D. Hill and D. B. Skillicorn. Practical Barrier Synchronisation. In *Proceedings Sixth EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pages 438–444. IEEE Computer Society Press, January 1998.

[9] J. Katajainen. Finding All Nearest Smaller Values on a Distributed Memory Machine. In *Proc. of Conference on Computing: The Australian Theory Symposium*, pages 100–107, 1996.

[10] Z.M. Kedem, G.M. Landau, and K.V. Palem. Optimal Parallel Prefix-Suffix Matching Algorithm And Applications. In *Proceedings 1st ACM Symposium on Parallel Algorithms and Architectures*, 1989, 388-398.

[11] D. Kravets and C. G. Plaxton. All Nearest Smaller Values on the Hypercube. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):456–462, 1996.

[12] C.P. Kruskal. Searching, Merging and Sorting in Parallel Computation. *IEEE Trans. Comput.*, C-32:942–946, 1983.

[13] W. F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, Berlin, 1995.

[14] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[15] B. Schieber and U. Vishkin. Finding All Nearest Neighbors for Convex Polygons in Parallel: A New Lower Bound Technique and A Matching Algorithm. *Discrete App. Math.*, 29:97–111, 1990.

[16] D.B. Skillicorn, J. MD Hill, and W.F. McColl. Questions and Answers About BSP. *Oxford University Computing Laboratory*, 1996.

[17] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

XIN HE received the Ph.D. degree in computer and information science in 1987, the M.S. degree in computer and information science in 1984, and the M.S. degree in mathematics in 1981, all from Ohio State University. He attended the Graduate School of the Academia Sinica, Beijing, China, from 1978 to 1980, majoring in mathematics. He attended Lanzhow University, Lanzhow, China, in 1978, majoring in chemistry. In the fall of 1987, Dr. He joined the faculty of the Department of Computer Science at the State University of New York at Buffalo, where he is currently an associate professor. His primary research interests are graph algorithms, parallel algorithms, and combinatorics. Dr. He is a member of the Association for Computing Machinery.

CHUN-HSI HUANG received his B.S. in computer and information science in 1989 from National Chiao-Tung University in Taiwan. He received his M.S. in computer science from University of Southern California in 1993. Since 1996, he has been a doctoral student in computer science and engineering department of SUNY at Buffalo. Prior to attending UB, he was a software engineer at Multimedia B.D. of Acer Inc., Taipei, Taiwan. His current research interest is on parallel graph algorithms.