Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems

Jehoshua Bruck, *Senior Member*, *IEEE*, Ching-Tien Ho, *Member*, *IEEE*, Shlomo Kipnis, *Member*, *IEEE*, Eli Upfal, *Senior Member*, *IEEE*, and Derrick Weathersby

Abstract—We present efficient algorithms for two all-to-all communication operations in message-passing systems: *index* (or all-to-all personalized communication) and *concatenation* (or all-to-all broadcast). We assume a model of a fully connected message-passing system, in which the performance of any point-to-point communication is independent of the sender-receiver pair. We also assume that each processor has $k \geq 1$ ports, through which it can send and receive k messages in every communication round. The complexity measures we use are independent of the particular system topology and are based on the communication start-up time, and on the communication bandwidth.

In the index operation among n processors, initially, each processor has n blocks of data, and the goal is to exchange the ith block of processor j with the jth block of processor i. We present a class of index algorithms that is designed for all values of n and that features a trade-off between the communication start-up time and the data transfer time. This class of algorithms includes two special cases: an algorithm that is optimal with respect to the measure of the start-up time, and an algorithm that is optimal with respect to the measure of the data transfer time. We also present experimental results featuring the performance tuneability of our index algorithms on the IBM SP-1 parallel system.

In the concatenation operation, among n processors, initially, each processor has one block of data, and the goal is to concatenate the n blocks of data from the n processors, and to make the concatenation result known to all the processors. We present a concatenation algorithm that is optimal, for most values of n, in the number of communication rounds and in the amount of data transferred.

Index Terms—All-to-all broadcast, all-to-all personalized communication, complete exchange, concatenation operation, distributed-memory system, index operation, message-passing system, multiscatter/gather, parallel system.

1 Introduction

Collective communication operations [2] are communication operations that generally involve more than two processors, as opposed to the point-to-point communication between two processors. Examples of collective communication operations include: (one-to-all) broadcast, scatter, gather, index (all-to-all personalized communication), and concatenation (all-to-all broadcast). See [13], [16] for a survey of collective communication algorithms on various networks with various communication models.

The need for collective communication arises frequently in parallel computation. Collective communication operations simplify the programming of applications for parallel computers, facilitate the implementation of efficient communication schemes on various machines, promote the portability of applications across different architectures, and reflect conceptual grouping of processes. In particular, collective communication is used extensively in many scientific applications for which the interleaving of stages of local computation with stages of global communication is possible (see [12]).

This paper studies the design of all-to-all communication algorithms, namely, collective operations in which every processor both sends data to and receives data from every other processor. In particular, we focus on two widely used operations: *index* (or all-to-all personalized communication) and *concatenation* (or all-to-all broadcast).

The algorithms described here are incorporated into the Collective Communication Library (CCL) [2], which was designed and developed for the new IBM line of scalable parallel computers. The first computer in this line, the IBM 9076 Scalable POWERparallel System 1 (SP1), was announced in February 1994.

1.1 Definitions and Applications

INDEX: The system consists of n processors $p_0, p_1, \ldots, p_{n-1}$. Initially, each processor p_i has n blocks of data $B[i, 0], B[i, 1], \ldots, B[i, n-1]$, where every block $B[i, j]$ is of size b. The goal is to exchange block $B[i, j]$ (the jth data block of processor p_i) with block $B[j, i]$ (the ith data block of processor p_j), for all $0 \leq i, j \leq n-1$. The final

- *J. Bruck is with the California Institute of Technology, Mail Code 136-93, Pasadena, CA 91125. E-mail: bruck@paradise.caltech.edu.*
- *C.-T. Ho and E. Upfal are with IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120. E-mail: {ho, upfal}@almaden.ibm.com.*
- *S. Kipnis is with News Datacom Research Ltd., 14 Wedgewood St., Haifa 34635, Israel. E-mail: skipnis@ndc.co.il.*
- *D. Weathersby is with the Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. E-mail: derrick@cs.washington.edu.*

Manuscript received 6 Apr. 1994; revised 27 Apr. 1997.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100822.

result is that each processor p_i, for $0 \le i \le n-1$, holds blocks $B[0, i], B[1, i], \ldots, B[n-1, i]$.

CONCATENATION: The system consists of n processors $p_0, p_1, \ldots, p_{n-1}$. Initially, each processor p_i has a block of data $B[i]$ of size b. The goal is to make the concatenation of the n data blocks, namely, $B[0]\ B[1] \cdots B[n-1]$, known to all the n processors.

Both the index and concatenation operations are used extensively in distributed-memory parallel computers and are included in the Message-Passing Interface (MPI) standard proposal [24]. (The index operation is referred to as MPI_Alltoall in MPI, while the concatenation is referred to as MPI_Allgather in MPI.) For example, the index operation can be used for computing the transpose of a matrix, when the matrix is partitioned into blocks of rows (or columns) with different blocks residing on different processors. Thus, the index operation can be used to support the remapping of arrays in HPF compilers, such as remapping the data layout of a two-dimensional array from (block, *) to (cyclic, *), or from (block, *) to (*, block). The index operation is also used in FFT algorithms [22], in Ascend and Descend algorithms [26], in the Alternating Direction Implicit (ADI) method [21], and in the solution of Poisson's problem by the Fourier Analysis Cyclic Reduction (FACR) method [28], [23], or the two-dimensional FFT method [8]. The concatenation operation can be used in matrix multiplication [19] and in basic linear algebra operations [12].

1.2 Communication Model

We assume a model of a multiport fully connected message-passing system. The assumption of full connectivity means that each processor can communicate directly with any other processor and that every pair of processors are equally distant. The assumption of multiple ports means that, in every communication step (or round), each processor can send k distinct messages to k processors and simultaneously receive k messages from k other processors, for some $k \ge 1$. Throughout the paper, we assume $1 \le k \le n-1$, where n is the number of processors in the system. The multiport model generalizes the one-port model that has been widely investigated. There are examples of parallel systems with k-port capabilities for $k > 1$, such as the nCUBE/2, the CM-2 (where k is the dimension of the hypercube in both machines), and transputer-based machines.

Such a fully connected model addresses emerging trends in many modern distributed-memory parallel computers and message-passing communication environments. These trends are evident in systems such as IBM's Vulcan [6], MIT's J-Machine [10], NCUBE's nCUBE/2 [25], Thinking Machines' CM-5 [29], and IBM's 9076 Scalable POWERparallel System 1, and in environments such as IBM EUI [1], PICL [14], PARMACS [17], Zipcode [27], and Express [31]. These systems and environments generally ignore the specific structure and topology of the communication network and assume a fully connected collection of processors, in which each processor can communicate directly with any other processor by sending and receiving messages. The fact that this model does not assume any single topology makes it general and flexible. For instance, this model allows the development of algorithms that are portable between different machines, that can operate within arbitrary and dynamic subsets of processors, and that can operate in the presence of faults (assuming connectivity is maintained). In addition, algorithms developed for this model can also be helpful in designing algorithms for specific topologies.

We use the linear model [13] to estimate the communication complexity of our algorithms. In the linear model, the time to send an m-byte message from one processor to another, without congestion, can be modeled as $T = \beta + m\tau$, where β is the overhead (start-up time) associated with each send or receive operation, and τ is the communication time for sending each additional byte (or any appropriate data unit).

For convenience, we define the following two terms in order to estimate the time complexities of our communication algorithms in the linear model:

- C_1: the number of communication steps (or rounds) required by an algorithm. C_1 is an important measure when the communication start-up time is high, relative to the transfer time, of one unit of data, and the message size per send/receive operation is relatively small.
- C_2: the amount of data (in the appropriate unit of communication: bytes, flits, or packets) transferred in a sequence. Specifically, let m_i be the largest size of a message (over all ports of all processors) sent in round i. Then, C_2 is the sum of all the m_is over all rounds i. C_2 is an important measure when the start-up time is small compared to the message size.

Thus, in our fully connected, linear model, an algorithm has an estimated communication time complexity of $T = C_1\beta + C_2\tau$. It should be noted that there are more detailed communication models, such as the BSP model [30], the Postal model [3], and the LogP model [9], which further take into account that a receiving processor generally completes its `receive` operation later than the corresponding sending processor finishes its `send` operation. However, designing practical and efficient algorithms in these models is substantially more complicated. Another important issue is the uniformity of the implementation. For example, in the LogP model, the design of collective communication algorithms is based on P, the number of processors. Optimal algorithms for two distinct values of P may be very different. This presents a challenge when the goal is to support collective communication algorithms for processor groups with various sizes while using one collective communication library.

1.3 Main Contributions and Organization

We study the complexity of the index and concatenation operations in the k-port fully connected message-passing model. We derive lower bounds and develop algorithms for these operations. The following is a description of our main results:

- **Lower bounds:** Section 2 provides lower bounds on the complexity measures C_1 and C_2 for both the concatenation and the index operations.

For the concatenation operation, we show that any algorithm requires $C_1 \geq \lceil \log_{k+1} n \rceil$ communication rounds and sends $C_2 \geq \left\lceil \frac{b(n-1)}{k} \right\rceil$ units of data.

For the index operation, we show that any algorithm requires $C_1 \geq \lceil \log_{k+1} n \rceil$ communication rounds and sends $C_2 \geq \left\lceil \frac{b(n-1)}{k} \right\rceil$ units of data. We also show that, when n is a power of $k + 1$, any index algorithm that uses the minimal number of communication rounds (i.e., $C_1 = \log_{k+1} n$) must transfer $C_2 \geq \frac{bn}{k+1} \log_{k+1} n$ units of data. Finally, we show that, in the one-port model, if the number of communication rounds C_1 is $O(\log n)$, then C_2 must be $\Omega(bn \log n)$.

- **Index algorithms**: Section 3 describes a class of efficient algorithms for the index operation among n processors. This class of algorithms is designed for arbitrary values of n and features a trade-off between the start-up time (measure C_1) and the data transfer time (measure C_2). Using a parameter r, where $2 \leq r \leq n$, the communication complexity measures of the algorithms are $C_1 = \left\lceil \frac{r-1}{k} \right\rceil \lceil \log_r n \rceil$ and $C_2 \leq b \left\lceil \frac{r-1}{k} \right\rceil \left\lceil \frac{n}{r} \right\rceil \lceil \log_r n \rceil$. Note that, following our lower bound results, optimal C_1 and C_2 cannot be obtained simultaneously. To increase the performance of the index operation, the parameter r can be carefully chosen as a function of the start-up time β, the data transfer rate τ, the message size b, the number of processors n, and the number of ports k. Two special cases of this class are of particular interest: One case exhibits the minimal number of communication rounds (i.e., C_1 is minimized to $\lceil \log_{k+1} n \rceil$ by choosing $r = k + 1$), and another case features the minimal amount of data transferred (i.e., C_2 is minimized to $b \left\lceil \frac{n-1}{k} \right\rceil$ by choosing $r = n$). The one-port version of the index algorithm was implemented on the IBM's SP-1 to confirm the existence of the trade-off between C_1 and C_2. It should be noted that, when n is a power of two, there are known algorithms for the index operation which are based on the structure of a hypercube (see [5], [20], [18]). However, none of these algorithms can be easily generalized to values of n that are not powers of two without losing efficiency. The idea of a trade-off between C_1 and C_2 is not new and has been applied to hypercubes in [5], [18].
- **Concatenation algorithms**: Section 4 presents algorithms for the concatenation operation in the k-port model. These algorithms are optimal for any values of n, b, and k, except for the following range: $b \geq 3$, $k \geq 3$, and $(k + 1)^d - k < n < (k + 1)^d$, for some d. (Thus, if $b = 1$ **or** $k = 1$, which covers most practical cases, our algorithm is optimal.) In this special range, we achieve either optimal C_2 and suboptimal C_1 (one more than the lower bound $\lceil \log_{k+1} n \rceil$), or optimal C_1 and suboptimal C_2 (at most $b - 1$ more than the lower bound $\left\lceil \frac{b(n-1)}{k} \right\rceil$).
- **Pseudocode**: Appendices A and B provide pseudocode for the index and concatenation algorithms, respectively, in the one-port model. Both the index and concatenation operations were included in the Collective Communication Library [2] of the External User Interface (EUI) [1] for the 9076 Scalable POWERparallel System (SP1) by IBM. In addition, these one-port versions of the algorithms have been implemented on various additional software platforms including PVM [15], and Express [31].

2 Lower Bounds

This section provides lower bounds on the complexity measures C_1 and C_2 for algorithms that perform the concatenation and index operations. Proposition 2.1 was shown in [13]. We include it here for completeness.

2.1 Lower Bounds for the Concatenation Operation

PROPOSITION 2.1. *In the k-port model, for $k \geq 1$, any concatenation algorithm requires $C_1 \geq \lceil \log_{k+1} n \rceil$ communication rounds.*

PROOF. Focus on one particular processor, say, processor p_0. The concatenation operation requires, among other things, that the data block $B[0]$ of processor p_0 be broadcast among the n processors. With k communication ports per processor, data block $B[0]$ can reach at most $(k + 1)^d$ processors in d communication rounds. For $(k + 1)^d$ to be at least n, we must have $d \geq \lceil \log_{k+1} n \rceil$ communication rounds. □

PROPOSITION 2.2. *In the k-port model, for $k \geq 1$, any concatenation algorithm transfers $C_2 \geq \left\lceil \frac{b(n-1)}{k} \right\rceil$ units of data.*

PROOF. Each processor must receive the $n - 1$ data blocks of the other $n - 1$ processors, the combined size of which is $b(n - 1)$ units of data. Since each processor can use its k input ports simultaneously, the amount of data transferred through one of the input ports must be at least $\left\lceil \frac{b(n-1)}{k} \right\rceil$. □

2.2 Lower Bounds for the Index Operation

PROPOSITION 2.3. *In the k-port model, for $k \geq 1$, any index algorithm requires $C_1 \geq \lceil \log_{k+1} n \rceil$ communication rounds.*

PROOF. Any concatenation operation on an array $B[i]$, $0 \leq i < n$, can be reduced to an index operation on $B[i, j]$, $0 \leq i, j < n$, by letting $B[i, j] = B[i]$ for all i and j. Thus, the proposition follows from Proposition 2.1. □

PROPOSITION 2.4. *In the k-port model, for $k \geq 1$, any index algorithm transfers $C_2 \geq \left\lceil \frac{b(n-1)}{k} \right\rceil$ units of data.*

PROOF. Similar to the proof of Proposition 2.3, the proposition follows from Proposition 2.2. □

2.3 Compound Lower Bounds for the Index Operation

Here, we provide additional lower bounds for the index operation. These lower bounds characterize the measure C_1 as a function of C_2 and vice versa. Theorems 2.5 and 2.7 show that when C_1 is optimized first, the lower bound on C_2 becomes an order of $O(\log_{k+1} n)$ higher than the "stand-alone" lower bound given in Proposition 2.4. Then, Theorem 2.6 shows that when C_2 is optimized first, the lower bound on C_1 becomes $(n-1)/k$ as opposed to $\lceil \log_{k+1} n \rceil$. Finally, Theorem 2.9 gives a more general lower bound for the one-port case.

THEOREM 2.5. *If $n = (k+1)^d$, for some integer $d \geq 0$, then any index algorithm that uses exactly $C_1 = \log_{k+1} n$ communication rounds must transfer at least $C_2 = \frac{bn}{k+1}\log_{k+1} n$ units of data.*

PROOF. Let $n = (k+1)^d$. In order to finish the algorithm in exactly $\log_{k+1} n = d$ rounds, the number of processors having received data from a given processor, say p_i, must grow by a factor of $k+1$ in every round. This defines a unique structure of the spanning tree T_i, which is rooted at p_i, that is a generalized version of the binomial tree used to distribute the $n-1$ data blocks of processor p_i among the other $n-1$ processors. Denote by ℓ_j the number of processors at level j in tree T_i rooted at processor p_i. One may use induction to show that $\ell_j = \binom{d}{j}k^j$. Now, the total amount of data D_i that is injected into the network over the edges of the binomial tree T_i rooted at p_i is given by

$$D_i = b\sum_{j=0}^{d} j\ell_j = b\sum_{j=0}^{d} j\binom{d}{j}k^j = b\frac{k}{k+1}dn,$$

where the last equality step can be derived by differentiating both sides of

$$\sum_{j=0}^{d}\binom{d}{j}k^j = (1+k)^d$$

and then multiplying both sides by bk. Now, clearly,

$$C_2 \geq \sum_{i=0}^{n-1}\frac{D_i}{nk} = \frac{bn}{k+1}d = \frac{bn}{k+1}\log_{k+1} n\ . \qquad \square$$

THEOREM 2.6. *Any algorithm for the index operation that transfers exactly $C_2 = \frac{b(n-1)}{k}$ units of data from each processor requires $C_1 \geq \frac{n-1}{k}$ communication rounds.*

PROOF. In the index operation, each processor has $n-1$ data blocks that it needs to send to the other $n-1$ processors. If each processor is allowed to transfer at most $\frac{b(n-1)}{k}$ units of data per port over all rounds, then it must be the case that the jth data block of processor p_i is sent directly from processor p_i to processor p_j. (That is, each data block is sent exactly once from its source to its destination, and no processor can forward data blocks of other processors.) In this case, each processor must send $n-1$ distinct messages to the other $n-1$ processors. Any such algorithm must require $C_1 \geq \frac{n-1}{k}$ rounds. $\square$

THEOREM 2.7. *Any index algorithm that uses $C_1 = \lceil \log_{k+1} n \rceil$ communication rounds must transfer at least $C_2 = \Omega\left(\frac{bn}{k+1}\log_{k+1} n\right)$ units of data.*

PROOF. It is sufficient to prove the theorem for $b = 1$. Consider any algorithm that finishes the index operation in $d = C_1$ (minimum) rounds. We show that the algorithm executed a total of $\Omega(n^2 \log_{k+1} n)$ data transmissions (over all nodes), thus, there is a port that transmitted $\Omega\left(\frac{n}{k+1}\log_{k+1} n\right)$ units of data.

We first concentrate on the data distribution from a given source node v to all other $n-1$ nodes. Any such algorithm can be characterized by a sequence of $d+1$ sets, $S_0, S_1, \cdots, S_d$, where S_i is the set of nodes that have received their respective data by the end of communication round i. Thus, $S_0 = \{v\}$, $|S_d| = n$, and S_i contains S_{i-1}, plus nodes that received data from nodes in S_{i-1} in the ith communication rounds. Let $x_i = |S_i|$. Clearly, $x_i \leq x_{i+1} \leq (k+1)x_i$, because each node in S_i can send data to at most k other nodes under the k-port model.

Next, we assign weights to the nodes in the sets, S_is, where the weight of a node u in S_i represents the path length (or the number of communication rounds incurred) from v to u in achieving the data distribution. The weights can be assigned based on the following rule. If a node u appears first in S_i due to a data transmission from node w in S_{i-1}, then the weight of u is the weight of w plus one. Note that, once a node is assigned a weight, it holds the same weight in all subsequent sets.

By Lemma 2.8, we know that there are at most $\binom{j}{f}k^f$ nodes of weight f in S_j. Our goal is to give a lower bound for the sum of the weights of the n nodes in S_d. Without loss of generality, we can assume that the sum of the weights is the minimum possible.

Let $X = \sum_{f=0}^{d}\binom{d}{f}k^f = (1+k)^d$. By the choice of d, $n \leq X < n(1+k)$.

Let $Y = \sum_{f=0}^{\lceil \frac{d}{2} \rceil - 1}\binom{d}{f}k^f$. Since, for

$$f \leq \left\lceil \frac{d}{2} \right\rceil - 1, \binom{d}{f}k^f \leq \frac{1}{k}\binom{d}{d-f}k^{d-f},$$

$$Y \leq \frac{1}{1+k}X = (1+k)^{d-1} < n.$$

Thus, the algorithm must use all the possible nodes with weights less than $\lceil d/2 \rceil$.

To bound the sum of the weights ,we need a lower bound on

$$Z = \sum_{f=0}^{\lceil \frac{d}{2} \rceil - 1} f \binom{d}{f} k^f .$$

For $f \le \lceil d/2 \rceil - 1$, $\binom{d}{f} k^f$ is monoton in f. Thus, at least $n/2$ of the nodes have weight at least $\left(\lceil d/2 \rceil - 1\right)/2$. That is, $Z = \Omega(nd)$.

Summing over all origins, the total number of transmissions is at least $nZ = \Omega(n^2 d)$. Thus, at least one port has a sequence of

$$C_2 = \Omega\left(\frac{n}{k} d\right) = \Omega\left(\frac{n}{k+1} \log_{k+1} n\right)$$

data transmissions. □

LEMMA 2.8. *There are no more than* $\binom{j}{f} k^f$ *nodes of weight f in* S_j (*defined in the proof of Theorem* 2.7).

PROOF. We prove by induction on j. There is clearly no more than one node of weight zero and k nodes of weight one in S_1. Assume that the hypothesis holds for $j - 1$. Note that S_j contains up to $\binom{j-1}{f} k^f$ nodes of weight f that appeared with the same weight in S_{j-1}, plus up to $k\binom{j-1}{f-1} k^{f-1}$ nodes that receive data at communication round j from nodes with weight $f - 1$ in S_{j-1}. The claim holds for j since

$$\binom{j-1}{f} k^f + k\binom{j-1}{f-1} k^{f-1} = \binom{j}{f} k^f .$$ □

THEOREM 2.9. *When* $k = 1$, *any algorithm for the index operation that uses* $C_1 = O(\log n)$ *communication rounds must transfer* $C_2 = \Omega(bn \log n)$ *units of data.*

PROOF. Assume that there is an algorithm with $C_1 \le c \log n$ for some constant $c \ge 1$. Consider the binomial distribution $\binom{c \log n}{j}$. Let h be the minimal ℓ, such that $\sum_{j=0}^{\ell+1} \binom{c \log n}{j} \ge n$. One can show that any algorithm that finishes in $c \log n$ rounds must have the following property. For every j such that $1 \le j \le h$, there exist $\binom{c \log n}{j}$ messages from each node that travel at least j hops in the network. Notice that, in this property, each message can only be counted once for a given j. Therefore, the average number of hops a message has to travel for each node is $h/2$, if $h \le \log n$, or $\log n/2$, if $h \ge \log n$. Since h must be $\Omega(\log n)$ from Lemma C.1 in Appendix C, we have $C_2 = \Omega(bn \log n)$. □

3 INDEX ALGORITHMS

This section presents a class of efficient algorithms for the index operation. First, we provide an overview of the algorithms. Then, we focus on the communication phase of the algorithms for the one-port model. Next, we describe two special cases of this class of algorithms. Then, we generalize the algorithms to the k-port model. And finally, we comment on the implementation and performance of this class of algorithms.

3.1 Overview

The class of algorithms for the index operation among n processors can be represented as a sequence of processor-memory configurations. Each processor-memory configuration has n columns of n blocks each. Columns are labeled from 0 through $n - 1$ (from left to right in the figures) and blocks are labeled from 0 through $n - 1$ (from top to bottom in the figures). Column i represents processor p_i, and block j represents the jth data block in the memory offset. The objective of the index operation, then, is to transpose these columns of blocks. Fig. 1 shows an example of the processor-memory configurations before and after the index operation for $n = 5$ processors. The notation "ij" in each box represents the jth data block initially allocated to processor p_i. The label j is referred to as the block-id.

All the algorithms in the class consist of three phases. Phases 1 and 3 require only local data rearrangement on each processor, while Phase 2 involves interprocessor communication.

PHASE 1. Each processor p_i independently rotates its n data blocks i steps upwards in a cyclical manner.

PHASE 2. Each processor p_i rotates its jth data block j steps to the right in a cyclical manner. This rotation is im-

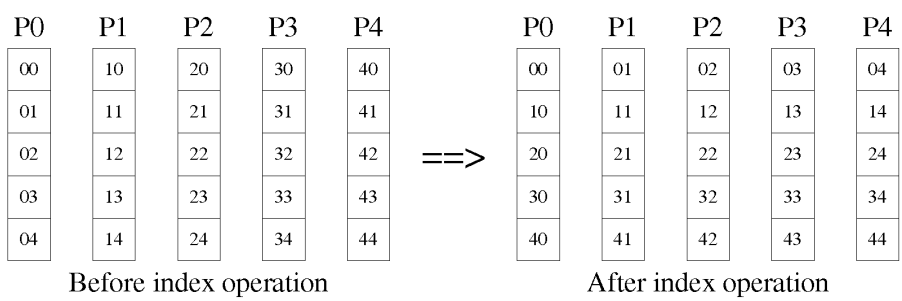


Fig. 1. Memory-processor configurations before and after an index operation on five processors.

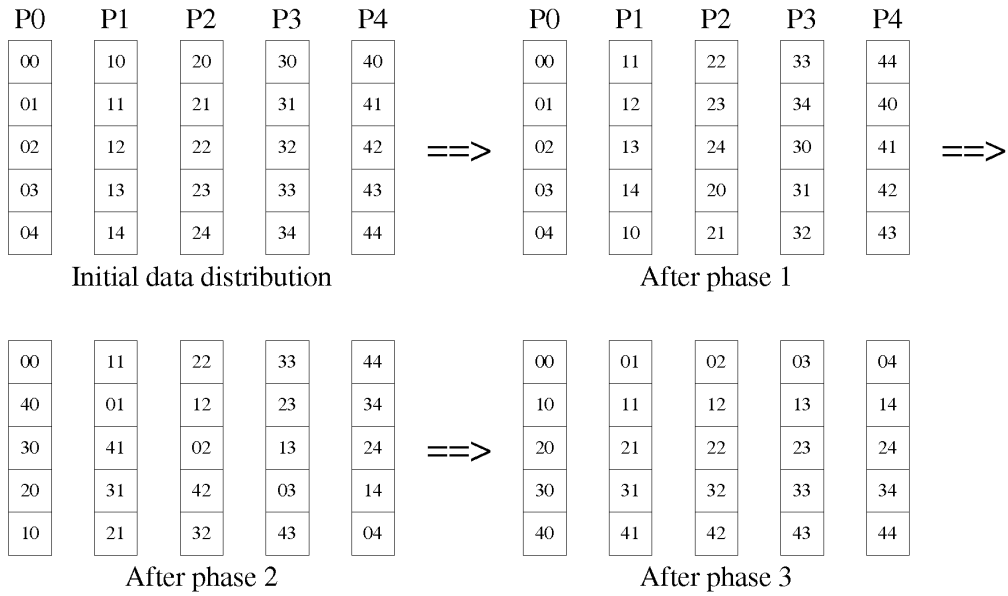


Fig. 2. An example of memory-processor configurations for the three phases of the index operation on five processors.

plemented by interprocessor communication.

PHASE 3. Each processor p_i independently rotates its n data blocks i steps downwards in a cyclical manner.

Fig. 2 presents an example of these three phases of the algorithm for performing an index operation among $n = 5$ processors.

The implementation of Phases 1 and 3 on each processor involves only local data movements and is straightforward. In the sequel, we focus only on the implementation of Phase 2. Different algorithms are derived depending on how the communication pattern of Phase 2 is decomposed into a sequence of point-to-point communication rounds.

3.2 The Interprocessor Communication Phase

We present the decomposition of Phase 2 into a sequence of point-to-point communication rounds, assuming the one-port model and using a parameter r (for *radix*) in the range $2 \le r \le n$.

For convenience, we say that the *block-id* of the jth data block in each processor after Phase 1 is j. Consider the rotation required in Phase 2. Each block with a block-id j in processor i needs to be rotated to processor $(i + j) \bmod n$. The block-id j, where $0 \le j \le n - 1$, can be encoded using radix-r representation using $w = \lceil \log_r n \rceil$ digits. For convenience, we refer to these w digits from zero through $w - 1$ starting with the least significant digit. Our algorithm for Phase 2 consists of w subphases corresponding to the w digits. Each subphase consists of at most $r - 1$ steps, corresponding to the (up to) $r - 1$ different non-zero values of a given digit. In subphase x, for $0 \le x \le w - 1$, we iterate Step 1 through Step $r - 1$, as follows:

- During Step z of subphase x, where $1 \le z \le r - 1$ and $0 \le x \le w - 1$, all data blocks, for which the xth digit of their block-id is z, are rotated $z \cdot r^x$ steps to the right. This is accomplished in a communication round by a direct point-to-point communications between processor i and processor $(i + z \cdot r^x) \bmod n$, for each $0 \le i \le n - 1$.

For example, when r is chosen to be 3, the fifth block will be rotated two steps to the right during Step 2 of Subphase 0, and later rotated again three steps to the right during Step 1 of Subphase 1. This follows from the fact that 5 is encoded into "12" using radix-3 representation.

Note that, after w subphases, all data blocks have been rotated to the correct destination processor as specified by the processor id. However, data blocks are not necessarily in their correct memory locations. Phase 3 of the algorithm fixes this problem.

The following points are made regarding the performance of this algorithm.

- Each step can be realized by a single communication round by packing all the outgoing blocks to the same destination into a temporary array and sending them together in one message. Hence, each subphase can be realized in at most $r - 1$ communication rounds.
- The size of each message involved in a communication round is at most $b\lceil \frac{n}{r} \rceil$ data.
- Hence, the class of the index algorithms has complexity measures $C_1 \le (r - 1)\lceil \log_r n \rceil$ and

$$C_2 \le b(r - 1)\lceil \tfrac{n}{r} \rceil \lceil \log_r n \rceil,$$

where r is chosen in the range $2 \le r \le n$.

3.3 Two Special Cases

The class of algorithms for the index operation in the one-port model contains two interesting special cases:

1) When $r = 2$, the derived algorithm requires

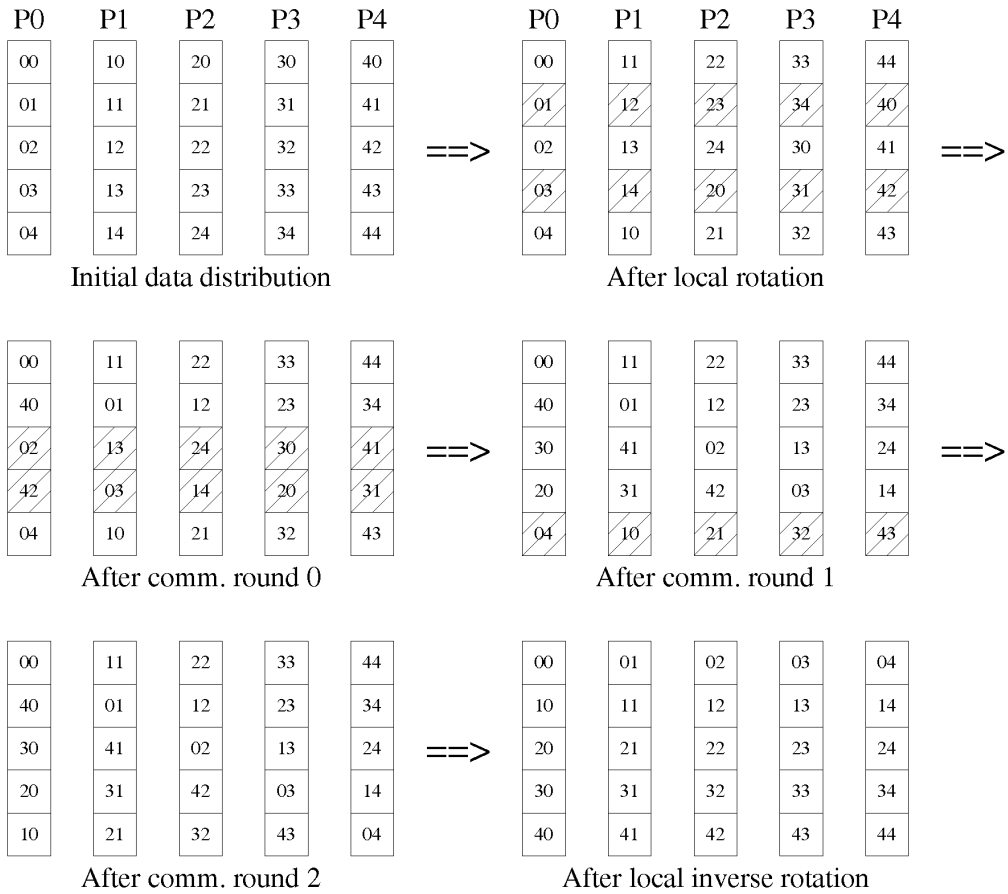


Fig. 3. An example of memory-processor configurations for the index algorithm on five processors, which has an optimal C_1 measure.

$$C_1 = \lceil \log_2 n \rceil$$

communication rounds, which is optimal with respect to the measure C_1. Also, in this case,

$$C_2 \le b\lceil \tfrac{n}{2} \rceil \lceil \log_2 n \rceil,$$

which is optimal (to within a multiplicative factor) for the case when $C_1 = \lceil \log_2 n \rceil$. Fig. 3 shows such an example with $r = 2$ and $n = 5$. The shaded data blocks are the ones subject to rotation during the next subphase.

2) When $r = n$, the derived algorithm transfers $C_2 = b(n - 1)$ units of data from each node, which is optimal with respect to the measure C_2. The value of C_1 in this case is $C_1 = n - 1$, which is optimal for the case when $C_2 = b(n - 1)$.

Hence, $r = 2$ should be chosen when the start-up time of the underlying machine is relatively significant, and the product of the block size b and the per-element transfer time is relatively small. On the other hand, $r = n$ should be chosen when the start-up time is negligible. In general, r can be fine-tuned according to the parameters of the underlying machines to balance between the start-up time and the data transfer time.

3.4 Generalization to the *k*-Port Model

We now present a modification to the index algorithm above for the k-port model. Phase 1 and Phase 3 of the algorithm remain the same. In Phase 2, we still have $w = \lceil \log_r n \rceil$ subphases as before, corresponding to the w digits in radix-r representation of any block-id j, where $0 \le j \le n - 1$. In each subphase, there are, at most, $r - 1$ "independent" point-to-point communication steps that need to be performed. Since these point-to-point communication steps are independent, they can be performed in parallel, subject to the constraint on the number of parallel input/output ports k. Thus, every k of these communication steps can be grouped together and performed concurrently. Therefore, each subphase consists of at most $\lceil \frac{r-1}{k} \rceil$ communication steps. The complexity measures for the index algorithm under the k-port model, therefore, are $C_1 \le \lceil \frac{r-1}{k} \rceil \lceil \log_r n \rceil$ and $C_2 \le b\lceil \frac{r-1}{k} \rceil \lceil \frac{n}{r} \rceil \lceil \log_r n \rceil$, where r can be chosen in the range $2 \le r \le n$. To minimize both C_1 and C_2, one clearly needs to choose r, such that $(r - 1) \bmod k = 0$.

3.5 Implementation

We have implemented the one-port version ($k = 1$) of the index algorithm on an IBM SP-1 parallel system. (The IBM SP-1 is closer to the one-port model in the domain of the multiport model.) The implementation is done on top of the point-to-point message-passing external user interface (EUI), running on the EUIH environment. At this level, the communication start-up, β, measures about 29 μsec, and the

sustained point-to-point communication bandwidth is about 8.5 Mbytes / sec, i.e., $\tau \approx 0.12\ \mu$sec / byte.

Fig. 4 shows the measured times of the index algorithm as a function of message size with various power-of-two radices r on a 64 node SP-1. As can be seen, the smaller radix tends to perform better for smaller message sizes, and vice versa.

Fig. 5 compares the measured times of the index algorithm with $r = 2$, $r = n = 64$, and optimal r among all power-of-two radices, respectively, on a 64 node SP-1. The break-even point of the message size between the two special cases of the index algorithms (i.e., $r = 2$ and $r = n$) occurs at about 100 to 200 bytes. The index algorithm with optimal power-of-two radix, as expected, is the best overall choice.

Fig. 6 shows the measured times of the index algorithm as a function of radix for three different message sizes: 32 bytes, 64 bytes, and 128 bytes. As the message size increases, the minimal time of the curve tends to occur at a higher radix.

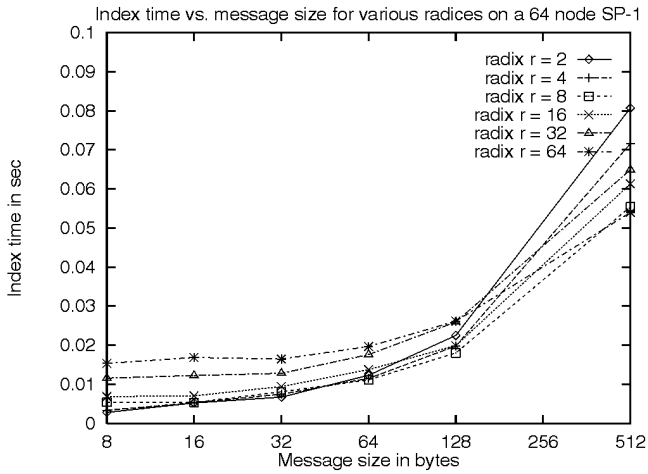


Fig. 4. The measured time of the index algorithm as a function of message sizes on a 64 node SP-1.

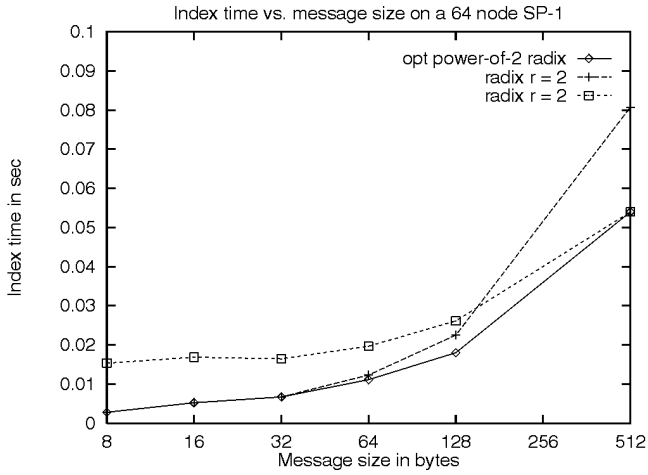


Fig. 5. The measured times of the index algorithm with $r = 2$, $r = n = 64$, and optimal r among all power-of-two radices, respectively, on a 64 node SP-1.

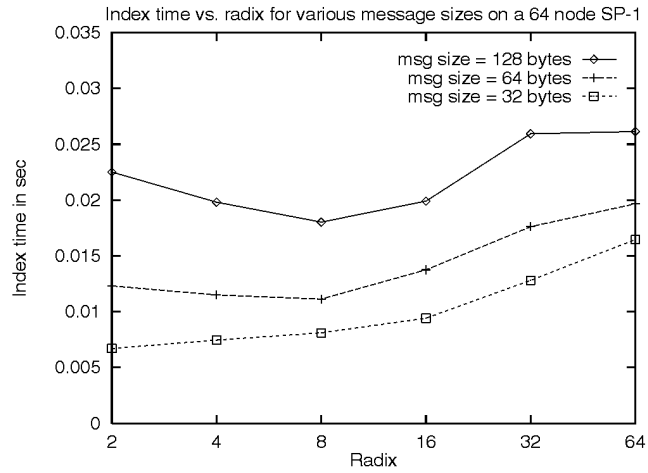


Fig. 6. The measured times of the index algorithm as a function of radix for various message sizes on a 64 node SP-1.

When comparing these measured times with our predicted times based on the linear model, we find big discrepancies quantitatively, but relatively consistent qualitatively. Note that we are mainly interested in the qualitatively behavior of the index algorithm on a general message-passing system. We believe the quantitative differences between the measured times and the predicted times are due to the following factors:

1) There are various system routines running in the background that have a higher priority than the user processes.
2) We do not model the copy time incurred by the function **copy**, **pack,** and **unpack** (see the pseudocode in Appendix A).
3) We do not model the congestion behavior of the SP-1.
4) There is a slowdown factor, somewhere between one and two, from the linear model to the send_and_receive model.

If we model the congestion behavior as a fixed multiplicative factor of t_c and assume the system routines have a fixed slowdown factor of the overall time, then the total time for the index operation can be modeled as

$$T = \gamma_1 C_1 t_s + \gamma_2 C_2 t_c + \gamma_3.$$

4 Concatenation Algorithms

There are two known algorithms for the concatenation operation in the one-port model. The first is a simple folklore algorithm which consists of two phases. In the first phase, the n blocks of data from the n processors are accumulated to a designated processor, say processor p_0. This can be done using a binomial tree (or a subtree of it when n is not a power of two). In the second phase, the concatenation result from processor p_0 is broadcast to the n processors using the same binomial tree. This algorithm is not optimal since it consists of $C_1 = 2\lceil \log n \rceil$ communication rounds and

transfers $C_2 = 2b(n-1)$ units of data. The second known concatenation algorithm is for the case when n is a power of two and $k = 1$ (see [20]). This algorithm is based on the structure of a binary hypercube and is optimal in both C_1 and C_2. For a given $k \geq 1$, this algorithm can be generalized to the case where n is a power of $k+1$ by using the structure of a generalized hypercube [4]. However, for general values of n, we do not know of any existing concatenation algorithm that is optimal in both C_1 and in C_2, even when $b = k = 1$.

In this section, we present efficient concatenation algorithms for the k-port communication model that, in most cases of n and k, are optimal in both C_1 and C_2. Throughout this section, we assume that k is in the range $1 \leq k \leq n-2$. Notice that, for $k \geq n-1$, the trivial algorithm that takes a single round is optimal.

The main structure that we use for deriving the algorithms is that of circulant graphs. We note here that circulant graphs are also useful in constructing fault-tolerant networks [7].

DEFINITION. A circulant graph $G(n, S)$ is characterized by two parameters: the number of nodes n, and a set of offsets S. In $G(n, S)$, the n nodes are labeled from 0 through $n-1$, and each node i is connected to node $((i-s) \bmod n)$ and to node $((i+s) \bmod n)$ for all $s \in S$ (see [11]).

The concatenation algorithm consists of d rounds. Let $d = \lceil \log_{k+1} n \rceil$, that is, $(k+1)^{d-1} < n \leq (k+1)^d$. Also let $n = n_1 + n_2$, where $n_1 = (k+1)^{d-1}$ and $1 \leq n_2 \leq kn_1$. The rounds of the algorithm can be divided into two phases. The first phase consists of $d-1$ rounds, at the end of which every node has the concatenation result of the $n_1 - 1$ nodes that precede it in the numbering (in a circulant sense). The second phase consists of a single round and completes the concatenation operation among the n nodes.

4.1 The First $d-1$ Rounds

For the first $d-1$ rounds, we use a circulant graph $G(n, S)$, where

$$S = S_0 \cup S_1 \cup \cdots \cup S_{d-2},$$

$$S_i = \{(k+1)^i, 2(k+1)^i, \ldots, k(k+1)^i\}.$$

We identify the n processors with the n nodes of $G(n, S)$, which are labeled from 0 through $n-1$.

The communication pattern related to broadcasting the data item of each node can be described by a spanning tree. Let T_i denote the spanning tree associated with the data item $B[i]$ of node i (namely, T_i is rooted at node i). We describe the spanning tree associated with each node by specifying the edges that are used in every communication round. The edges associated with round i are called round-i-edges. First, we describe the tree T_0, and then we show how tree T_i, for $1 \leq i \leq n-1$, can be derived from tree T_0.

We start with an initial tree T_0 which consists only of node 0. In round 0, we add edges with offsets in S_0 to T_0 to form a partial spanning tree; the added edges are the round-0-edges. (That is, in round 0, we add the set of edges $\{(0, 1), (0, 2), \cdots, (0, k)\}$.) In general, in round r, where $0 \leq r \leq d-2$, we add edges with offsets in S_r to the current partial spanning tree to form a new larger partial spanning tree. It is easy to verify that, after $d-1$ rounds, the resulting tree spans the first n_1 nodes starting from node 0, namely, nodes 0 through $n_1 - 1$. Fig. 7 illustrates the process of constructing T_0 for the case of $k = 2$ and $n = 9$.

Next, we use tree T_0 to construct the spanning trees T_i, for $1 \leq i \leq n-1$. We do this by translating each node j in T_0 to node $(j+i) \bmod n$ in T_i. Also, the round id associated with each tree edge in T_i (which represents the round during which the corresponding communication is performed) is the same as that of the corresponding tree edge in T_0. Fig. 8 illustrates tree T_1 for the case of $k = 2$ and $n = 9$. It is easy to see that T_1 was obtained from T_0 by adding one (modulo nine) to the labels of the nodes in T_0.

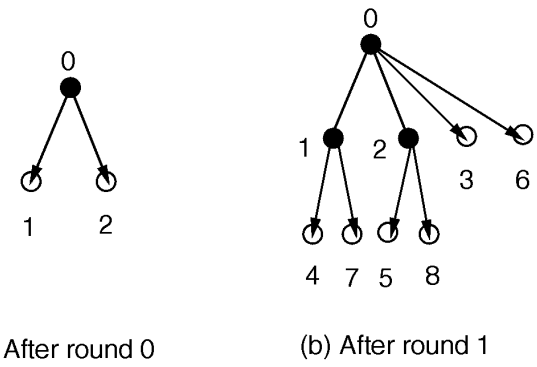


Fig. 7. The two rounds in constructing the spanning tree rooted at node 0 for $n = 9$ and $k = 2$.

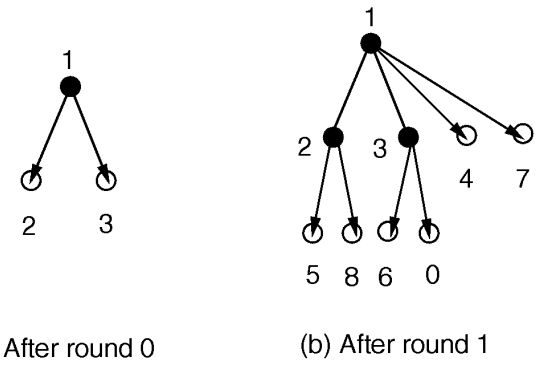


Fig. 8. The two rounds in constructing the spanning tree rooted at node 1 for $n = 9$ and $k = 2$. They can be derived by translating node addresses of the spanning tree rooted at node 0 in Fig. 7.

The concatenation algorithm in each node is specified by the trees T_i, for $0 \leq i \leq n-1$, as follows:

In round i, for $0 \leq i \leq d-2$, do:

- For all $0 \leq j \leq n-1$, if data item $B[j]$ is present at the node, then send it on all round-i-edges of tree T_j.
- Receive the corresponding data items on the round-i-edges of all the n trees.

THEOREM 4.1. *After $d-1$ rounds of the above algorithm, every node i, for $0 \leq i \leq n-1$, has the n_1 data items $B[j]$, where $i \geq j \geq i - n_1 + j + 1 \pmod n$. Also, during these d–1 rounds, the measure C_2 is optimal*:

$$C_2 = \frac{b}{k}(n_1 - 1).$$

PROOF. The spanning trees T_i, for $1 \le T_i \le n-1$, are derived from T_0 by shifting the indices in a cyclic manner. Hence, it suffices to focus on the spanning tree T_0. Notice that the algorithm can be implemented in a k-port model, since, in every round i, we use only the set of offsets S_i, which consists of k offsets. Also, the tree T_0 is a spanning tree for the nodes p_i, where $0 \le i \le n_1 - 1$, because every i in this range can be represented using a set of distinct offsets from S. Hence, after $d-1$ rounds of the algorithm, the data items are distributed according to the claim of the theorem.

Next, we need to prove that C_2 associated with the $d-1$ rounds is as claimed. By induction on i, it follows that, before round i, any node has at most $(k+1)^i$ distinct data items. Hence, in round i, any node sends at most $(k+1)^i$ data items on any given edge. Thus,

$$C_2 \le b\left[1 + (k+1) + (k+1)^2 + \cdots + (k+1)^{d-2}\right] = \frac{b}{k}(n_1 - 1).$$

However, by the lower bound argument, we have $C_2 \ge \frac{b}{k}(n_1 - 1)$, and the claim follows. □

4.2 The Last Round

Before round $d-1$, the last round of the algorithm, we have the following situation: Every node i had broadcast its message to the $n_1 - 1$ nodes succeeding it in the circular graph and had received the broadcast message from the $n_1 - 1$ nodes preceding it in the circular graph. Consider tree T_0 just before the last round. The first n_1 nodes (nodes 0 through $n_1 - 1$) are included in the current tree, and the remaining n_2 nodes still need to be spanned. We bring the following proposition.

PROPOSITION 4.2. *The last round can be performed with* $C_2 = \left\lceil \frac{bn_2}{k} \right\rceil$, *for any combination of n, b, and k, except for the following range:* $b \ge 3$, $k \ge 3$, *and* $(k+1)^d - k < n < (k+1)^d$, *for some d.*

The proof of this proposition is somewhat complicated, and we only give the main ideas here. The basic idea is to transform the scheduling problem for the last round of the algorithm into a table partitioning problem. (In the sense that, if the table partitioning problem can be solved, then we have an optimal algorithm by deriving an optimal schedule for the last round.) The table partitioning problem is defined as follows. Let $\alpha = \left\lceil \frac{bn_2}{k} \right\rceil$. Given a table of b rows and n_2 columns, we would like to partition the table into disjoint k areas, denoted by $A_1, A_2, \ldots, A_k$, such that

- the column-span of A_i, for all $1 \le i \le k$, is at most n_1, where the *column-span* of A_i is defined as $R_i - L_i + 1$ if R_i and L_i are the rightmost and leftmost columns, respectively, touched by A_i; and
- the number of table entries in A_i, for all $1 \le i \le k$, is at most α.

If a solution can be found to the table-partitioning problem, then a schedule for the last round can be derived as follows. Each of the n_2 table columns corresponds to one of the n_2 nodes yet to be spanned, and each of the b table rows represents one byte. Table elements in the same area, say A_i, will use the same offset, which is determined by the index of the leftmost column touched by A_i.

It can be shown that a straightforward algorithm for partitioning the table satisfies the above two conditions for any combination of n, b, and k, except for the following range: $b \ge 3$, $k \ge 3$ and $(k+1)^d - k < n < (k+1)^d$, for some d. For instance, Table 1 presents a partitioning example for $n_1 = 3$, $n_2 = 7$, $b = 3$, and $k = 3$, which fall in the optimal range of n. The area covered by A_i is marked by the number i. From this table, one can derive the following scheduling for the last round:

- The sum of the weighted edges with offset 3 (in area A_1) is 7. Thus, node p_3 receives three bytes from p_0, node p_4 receives three bytes from p_1, and node p_5 receives one byte from p_2.
- The sum of the weighted edges with offset 5 (in area A_2) is 7. Thus, node p_5 receives two bytes from p_0, node p_6 receives three bytes from p_1, and node p_7 receives two bytes from p_3.
- The sum of the weighted edges with offset 7 (in area A_3) is 7. Thus, node p_7 receives one byte from p_0, node p_8 receives three bytes from p_1, and node p_9 receives three bytes from p_2.

After rotation, to generate n spanning trees, each of which is rooted at a different node, each node i needs to send seven bytes to nodes $(i+3) \bmod n$, $(i+5) \bmod n$, and $(i+7) \bmod n$, and receive seven bytes from nodes $(i-3) \bmod n$, $(i-5) \bmod n$, and $(i-7) \bmod n$.

THEOREM 4.3. *The above concatenation algorithm attains optimal* $C_1 = \lceil \log_{k+1} n \rceil$ *and* $C_2 = \left\lceil \frac{b(n-1)}{k} \right\rceil$ *for any combination of n, b, and k, except for the following range:* $b \ge 3$, $k \ge 3$, *and* $(k+1)^d - k < n < (k+1)^d$, *for some integer d.*

PROOF. By combining Theorem 4.1 and Proposition 4.2, we have $C_2 = \frac{b}{k}(n_1 - 1) + \left\lceil \frac{bn_2}{k} \right\rceil = \left\lceil \frac{b(n-1)}{k} \right\rceil$, which matches the lower bound of C_2 in Proposition 2.2. □

Fig. 9 presents an example of the concatenation algorithm for $k = 1$ and $n = 5$. Note that, to simplify the pseudocode included in Appendix A, we actually grow the spanning tree T_i using negative offsets. That is, in both the figure

TABLE 1
AN EXAMPLE OF THE TRANSFORMED PROBLEM FOR $n_1 = 3$ (p_0 THROUGH p_2), $n_2 = 7$ (p_3 THROUGH p_9), $b = 3$ (BYTES), AND $k = 3$ (PORTS)

	p_3	p_4	p_5	p_6	p_7	p_8	p_9
first byte	1	1	1	2	2	3	3
second byte	1	1	2	2	2	3	3
third byte	1	1	2	2	3	3	3

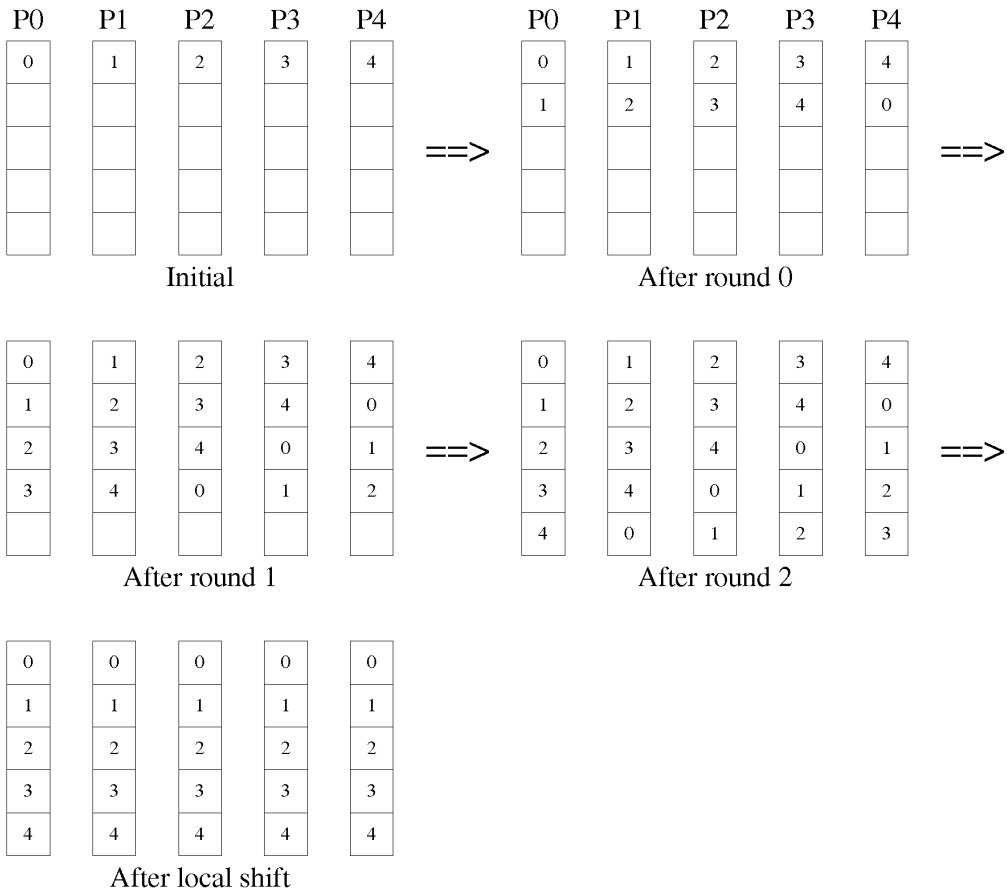


Fig. 9. An example of the one-port concatenation algorithm with five processors.

and in the pseudocode, left-rotations are performed instead of right-rotations.

REMARK. *For the nonoptimal range of n, it is easy to achieve optimal C_2 at the expense of increasing C_1 by one round over the lower bound. It is also easy to achieve optimal C_1 and suboptimal C_2, where C_2 is at most $b - 1$ more than the lower bound.*

APPENDIX A
PSEUDOCODE FOR THE INDEX ALGORITHM

This appendix presents pseudocode for the index algorithm of Section 3 when $k = 1$. This pseudocode sketches the implementation of the index operation in the Collective Communication Library of the EUI [1] by IBM. In the pseudocode, the function **index** takes six arguments: *outmsg* is an array for the outgoing message; *blklen* is the length in bytes of each data block; *inmsg* is an array for the incoming message; *n* is the number of processors involved; *A* is the array of the *n* different processor ids, such that, $A[i] = p_i$; and *r* is the *radix* used to tune the algorithm. Arrays *outmsg* and *inmsg* are each of length *blklen* * *n* bytes. Other routines that appear in the code are as follows: Routine **copy**(*A*, *B*, *len*) copies array *A* of size *len* bytes into array *B*. Routine **getrank**(*id*, *n*, *A*) returns the index *i* that satisfies $A[i] = id$. The routine **mod**(*x*, *y*) returns the value *x* mod *y* in the range of 0 through $y - 1$, even for negative *x*. The function **send_and_recv** takes six arguments: the outgoing message; the size of the outgoing message; the destination of the outgoing message; the incoming message; the size of the incoming message; and the source of the incoming message. The function **send_and_recv** is supported by IBM's Message Passing Library (MPL) [1] on SP-1 and SP-2, and the recent MPI standard [24]. It can also be implemented as a combination of blocking send and nonblocking receive.

In the following pseudocode, lines 3 and 4 correspond to Phase 1, lines 5 through 20 correspond to Phase 2, and lines 21 through 23 correspond to Phase 3. In Phase 2, there are *w* subphases, which are indexed by *i*. During each subphase, each processor needs to perform the **send_and_recv** operation $r - 1$ times, except for the last subphase, where each processor performs the **send_and_recv** operation only $\lceil n/r^{w-1} \rceil - 1$ times. Lines 7 through 11 take into account the special case for the last subphase. The routine **pack** is used to pack those blocks that need to be rotated to the same intermediate destination into a consecutive array. Specifically, **pack**(*A*, *B*, *blklen*, *n*, *r*, *i*, *j*, *nblocks*) packs some selected blocks of array *A* into array *B*; each block is of size *blklen* in bytes; those blocks, for which the *i*th digit of the radix-*r* representation of their block ids are equal to *j*, are selected for packing; and value of the number of selected blocks is written to the argument *nblocks*. The routine **unpack**(*A*, *B*, *blklen*, *n*, *r*, *i*, *j*, *nblocks*) is defined as the inverse function of

pack where *B* becomes the input array to be unpacked and *A* becomes the output array.

Function index (*outmsg, blklen, inmsg, n, A, r*)

```
(1)  w = ⌈log_r n⌉
(2)  my_rank = getrank (my_pid, n, A)
(3)  copy (outmsg, tmp[(n − my_rank) * blklen], my_rank *
         blklen)
(4)  copy (outmsg [my_rank * blklen], tmp, (n − my_rank) *
         blklen)
(5)  dist = 1
(6)  for i = 0 to w − 1 do
(7)      if (i == w − 1) then
(8)          h = ⌈n/dist⌉
(9)      else
(10)         h = r
(11)     endif
(12)     for j = 1 to h − 1 do
(13)         dest_rank = mod (my_rank + j ∗ dist, n)
(14)         src_rank = mod (my_rank − j ∗ dist, n)
(15)         pack (tmp, packed_msg, blklen, n, r, i, j,
                 nblocks)
(16)         send_and_recv (packed_msg, blklen ∗
                 nblocks, A [dest_rank],
                 packed_msg, blklen ∗
                 nblocks, A [src_rank])
(17)         unpack (tmp, packed_msg, blklen, n, r, i, j,
                 nblocks)
(18)     endfor
(19)     dist = dist ∗ r
(20) endfor
(21) for i = 0 to n − 1 do
(22)     copy (tmp [mod (my_rank − i, n) ∗ blklen], inmsg
             [i ∗ blklen], blklen)
(23) endfor
(24) return
```

Appendix B
Pseudocode for the Concatenation Algorithm

This appendix presents pseudocode for the concatenation algorithm of Section 4 when $k = 1$. This pseudocode sketches the implementation of the concatenation operation in the Collective Communication Library of the EUI [1] by IBM. In this pseudocode, the function **concat** takes five arguments: *outmsg* is an array for the outgoing message; *len* is the length in bytes of array *outmsg*; *inmsg* is an array for the incoming message; *n* is the number of processors involved; and *A* is the array of the *n* different processor ids, such that, $A[i] = p_i$. Array *inmsg* is of length *len* ∗ *n* bytes. The function **concat** sends and receives messages using the **send_and_recv** routine. The routines **copy**, **getrank**, **send_and_recv,** and **mod** were defined in Appendix A.

In the following pseudocode, each processor first initializes some variables and copies its *outmsg* array into a temporary array *temp* (lines 1 through 5). Then, each processor performs the first $d - 1$ rounds of the algorithm (lines 6 through 12). Then, each processor performs the last round of the algorithm (lines 13 and 16). Finally, each processor performs a local circular shift of the data such that all data blocks in its *inmsg* array begin with the block *B*[0] (lines 17 and 18).

Function concat (*outmsg, len, inmsg, n, A*)

```
(1)  d = ⌈log_2 n⌉
(2)  my_rank = getrank (my_pid, n, A)
(3)  copy (outmsg, temp, len)
(4)  nblk = 1
(5)  current_len = len
(6)  for r = 0 to d − 1 do
(7)      dest_rank = mod (my_rank − nblk, n)
(8)      src_rank = mod (my_rank + nblk, n)
(9)      send_and_recv (temp, current_len,
                 A [dest_rank],temp [current_len],
                 current_len, A [src_rank])
(10)     nblk = nblk ∗ 2
(11)     current_len = current_len ∗ 2
(12) endfor
(13) current_len = len ∗ (n − nblk)
(14)     dest_rank = mod (my_rank − nblk, n)
(15)     src_rank = mod (my_rank + nblk, n)
(16)     send_and_recv (temp, current_len,
                 A [dest_rank], temp [current_len],
                 current_len, A [src_rank])
(17) copy (temp, inmsg [len ∗ my_rank],
         len ∗ (n − my_rank))
(18) copy (temp [len ∗ (n − my_rank)], inmsg,
         len ∗ my_rank)
(19) return
```

Appendix C
Proof of a Lemma

LEMMA C.1. *Let c and m be integers such that* $2 \le c \le m$. *Then, if* $\sum_{j=0}^{h} \binom{cm}{j} \ge 2^m$, *then* $h \ge \min(m/64, m/8 \log c)$.

PROOF. Assume, for the sake of contradiction, that the lemma does not hold. First, note that the lemma holds if $h \ge m/64$, so it must be the case that $h < m/64$. Also, note that $\binom{cm}{0} = 1 < 2^m$, so $h \ge 1$ and $m > 64$. Therefore, $h + 1 \le 2m \le cm$. Because $h < m/64 \le cm/128$, the terms in the summation $\sum_{j=0}^{h} \binom{cm}{j} \ge 2^m$ are monotonically increasing, so

$$2^m \le \sum_{j=0}^{h} \binom{cm}{j} \le (h+1)\binom{cm}{j} \le (h+1)(cm)^h / h!$$

Note that $h! \ge h^{h+1/2}/e^h$, so

$$m \le \log(h+1) + h \log(cm) + h \log e - (h + 1/2) \log h$$

$$\le (h+1) \log(cm) + h \log e - h \log h.$$

Because $h < m/64 \le m/(2 \log e)$,

$$m/2 \le h(\log(cm) - \log h) + \log(cm).$$

Because $\log(cm) \le 2 \log m \le m/4$, it follows that $m/4 \le h(\log(cm) - \log h)$. Let $h = m/x$ and note that $x > 64$, so

$m/4 \le (m/x)(\log c + \log x)$, which implies that $x \le 4 \log c + 4 \log x$ and $x - 4 \log x \le 4 \log c$. Note that $x \ge 8\log x$, so $x/2 \le x - 4 \log x \le 4 \log c$. Therefore, $x \le 8\log c$ and $h = m/x \ge m/8 \log c$, which is a contradiction. □

ACKNOWLEDGMENTS

We thank Robert Cypher for his help in deriving Lemma C.1. Jehoshua Bruck was supported in part by U.S. National Science Foundation Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, and by DARPA and BMDO through an agreement with NASA / OSAT.

REFERENCES

[1] V. Bala, J. Bruck, R. Bryant, R. Cypher, P. deJong, P. Elustondo, D. Frye, A. Ho, C.-T. Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir, "The IBM External User Interface for Scalable Parallel Systems," *Parallel Computing*, vol. 20, no. 4, pp. 445–462, Apr. 1994.

[2] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir, "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 2, pp. 154–164, Feb. 1995.

[3] A. Bar-Noy and S. Kipnis, "Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems," *Mathematical Systems Theory*, vol. 27, no. 5, pp. 431-452, Sept. / Oct. 1994.

[4] L. Bhuyan and D. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network," *IEEE Trans. Computers*, vol. 33, no. 4, pp. 323–333, Apr. 1984.

[5] S. Bokhari, "Multiphase Complete Exchange on a Circuit-Switched Hypercube," *Proc. 1991 Int'l Conf. Parallel Processing*, vol. I, pp. 525–528, Aug. 1991.

[6] J. Bruck, R. Cypher, L. Gravano, A. Ho, C.-T. Ho, S. Kipnis, S. Konstantinidou, M. Snir, and E. Upfal, "Survey of Routing Issues for the Vulcan Parallel Computer," IBM Research Report, RJ-8839, June 1992.

[7] J. Bruck, R. Cypher, and C.-T. Ho, "Fault-Tolerant Meshes and Hypercubes with Minimal Numbers of Spares," *IEEE Trans. Computers*, vol. 42, no. 9, pp. 1,089–1,104, Sept. 1993.

[8] C.Y. Chu, "Comparison of Two-dimensional FFT Methods on the Hypercubes," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, pp. 1,430–1,437, 1988.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. Fourth SIGPLAN Symp. Principles and Practices Parallel Programming*, ACM, May 1993.

[10] W.J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler, "The J-Machine: a Fine-Grain Concurrent Computer," *Proc. Information Processing '89*, pp. 1,147–1,153, 1989.

[11] B. Elspas and J. Turner, "Graphs with Circulant Adjacency Matrices," *J. Combinatorial Theory*, no. 9, pp. 297–307, 1970.

[12] G. Fox, M. Johnsson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors, Vol. I.* Prentice Hall, 1988.

[13] P. Fraigniaud and E. Lazard, "Methods and Problems of Communication in Usual Networks," *Discrete Applied Math.*, vol. 53, pp. 79–133, 1994.

[14] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley, "A User's Guide to PICL: A Portable Instrumented Communication Library," ORNL Technical Report no. ORNL / TM-11616, Oct. 1990.

[15] G.A. Geist and V.S. Sunderam, "Network Based Concurrent Computing on the PVM System," ORNL Technical Report no. ORNL / TM-11760, June 1991.

[16] S.M. Hedetniemi, S.T. Hedetniemi, and A.L. Liestman, "A Survey of Gossiping and Broadcasting in Communication Networks," *Networks*, vol. 18, pp. 319-349, 1988.

[17] R. Hempel, "The ANL / GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming Using the Message Passing Programming Model, User's Guide and Reference Manual," technical memorandum, Gesellschaft für Mathematik und Datenverabeitung mbH, West Germany.

[18] C.-T. Ho and M.T. Raghunath, "Efficient Communication Primitives on Hypercubes," *Concurrency: Practice and Experience*, vol. 4, no. 6, pp. 427–458, Sept. 1992.

[19] S.L Johnsson and C.-T. Ho, "Matrix Multiplication on Boolean Cubes Using Generic Communication Primitives," *Parallel Processing and Medium-Scale Multiprocessors*, A. Wouk, ed., pp. 108–156. SIAM, 1989.

[20] S.L. Johnsson and C.-T. Ho, "Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes," *IEEE Trans. Computers*, vol. 38, no. 9, pp. 1,249–1,268, Sept. 1989.

[21] S.L. Johnsson and C.-T. Ho, "Optimizing Tridiagonal Solvers for Alternating Direction Methods on Boolean Cube Multiprocessors," *SIAM J. Scientific and Statistical Computing*, vol. 11, no. 3, pp. 563–592, 1990.

[22] S.L. Johnsson, C.-T. Ho, M. Jacquemin, and A. Ruttenberg, "Computing Fast Fourier Transforms on Boolean Cubes and Related Networks," *Advanced Algorithms and Architectures for Signal Processing II*, vol. 826, pp. 223–231. Soc. Photo-Optical Instrumentation Engineers, 1987.

[23] O.A. McBryan and E.F. Van de Velde, "Hypercube Algorithms and Implementations," *SIAM J. Scientific and Statistical Computing*, vol. 8, no. 2, pp. 227–287, Mar. 1987.

[24] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, May 1994.

[25] J.F. Palmer "The NCUBE Family of Parallel Supercomputers," *Proc. Int'l Conf. Computer Design*, 1986.

[26] F.P. Preparata and J.E. Vuillemin, "The Cube Connected Cycles: A Versatile Network for Parallel Computation," *Comm. ACM*, vol. 24, no. 5, pp. 300–309, May 1981.

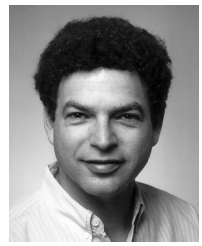
[27] A. Skjellum and A.P. Leung, "Zipcode: A Portable Multicomputer Communication Library Atop the Reactive Kernel," *Proc. Fifth Distributed Memory Computing Conf.*, pp. 328–337, Apr. 1990.

[28] P.N. Swarztrauber, "The Methods of Cyclic Reduction, Fourier Analysis, and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle," *SIAM Rev.*, vol. 19, pp. 490–501, 1977.

[29] *Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, 1991.

[30] L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[31] *Express 3.0 Introductory Guide*. Parasoft Corporation, 1990.



Jehoshua Bruck received the BSc and MSc degrees in electrical engineering from the Technion, Israel Institute of Technology, in 1982 and 1985, respectively, and the PhD degree in electrical engineering from Stanford University in 1989.

He is an associate professor of computation and neural systems and electrical engineering at the California Institute of Technology. His research interests include parallel and distributed computing, fault-tolerant computing, error-correcting codes, computation theory, and neural and biological systems. Dr. Bruck has extensive industrial experience including, serving as manager of the Foundations of Massively Parallel Computing Group at the IBM Almaden Research Center from 1990-1994, a research staff member at the IBM Almaden Research Center from 1989-1990, and a researcher at the IBM Haifa Science center from 1982-1985.

Dr. Bruck is the recipient of a 1995 Sloan Research Fellowship, a 1994 National Science Foundation Young Investigator Award, six IBM Plateau Invention Achievement Awards, a 1992 IBM Outstanding Innovation Award for his work on "Harmonic Analysis of Neural Networks," and a 1994 IBM Outstanding Technical Achievement Award for his contributions to the design and implementation of the SP-1, the first IBM scalable parallel computer. He has published more than 120 journal and conference papers in his areas of interests and he holds 20 patents. Dr. Bruck is a senior member of the IEEE and a member of the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*.

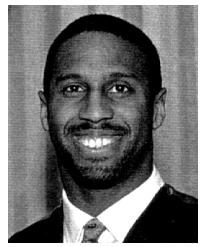
Ching-Tien Ho received a BS degree in electrical engineering from National Taiwan University in 1979 and the MS, MPhil, and PhD degrees in computer science from Yale University in 1985, 1986, and 1990, respectively.

He joined IBM Almaden Research Center as a research staff member in 1989. He was manager of the Foundations of Massively Parallel Computing group from 1994-1996, where he led the development of collective communication, as part of IBM MPL and MPI, for IBM SP-1 and SP-2 parallel systems. His primary research interests include communication issues for interconnection networks, algorithms for collective communications, graph embeddings, fault tolerance, and parallel algorithms and architectures. His current interests are data mining and on-line analytical processing. He has published more than 80 journal and conference papers in these areas.

Dr. Ho is a corecipient of the 1986 "Outstanding Paper Award" of the International Conference on Parallel Processing. He has received an IBM Outstanding Innovation Award, two IBM Outstanding Technical Achievement Awards, and four IBM Plateau Invention Achievement Awards. He has 10 patents granted or pending. He is on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*. He will be one of the program vice chairs for the 1998 International Conference on Parallel Processing. He has served on program committees of many parallel processing conferences and workshops. He is a member of the ACM, the IEEE, and the IEEE Computer Society.

Shlomo Kipnis (M'87) received a BSc in mathematics and physics in 1983 and an MSc in computer science in 1985, both from the Hebrew University of Jerusalem, Israel. He received a PhD in electrical engineering and computer science in 1990 from the Massachusetts Institute of Technology. From 1990-1993, he worked as a research staff member at the IBM T. J. Watson Research Center in Yorktown Heights, New York. From 1993-1995, he worked as a research staff member at the IBM Haifa Research Laboratory in Israel. Currently, he is working as manager of new technologies at NDS Technologies Israel. In addition, since 1994, Dr. Kipnis has been an adjunct professor of computer science at Bar Ilan University and at Tel Aviv University. His research interests include parallel and distributed processing, efficient communication structures and algorithms, and system security. Dr. Kipnis is a member of the IEEE, the IEEE Computer Society, ACM, and ILA. He has published in numerous journals and presented his work in many conferences and workshops. He is also an inventor and coinventor of two U.S. patents.

Eli Upfal received a BSc in mathematics from the Hebrew University in 1978, an MSc in computer science from the Weizmann Institute in 1980, and a PhD in computer science from the Hebrew University in 1983. During 1983-1984, he was a research fellow at the University of California at Berkeley, and, in 1984-1985, a postdoctoral fellow at Stanford University. In 1985, Dr. Upfal joined the IBM Almaden Research Center, where he is currently a research staff member in the Foundations of Computer Science Group. In 1988, he also joined the Faculty of Applied Mathematics and Computer Science at the Weizmann Institute, where he is currently the Norman D. Cohen Professor of Computer Science. Dr. Upfal's research interest include theory of algorithms, randomized computing, probabilistic analysis of algorithms, communication networks, and parallel and distributed computing. He is a senior member of the IEEE.



W. Derrick Weathersby is a PhD candidate in the Department of Computer Science at the University of Washington, Seattle, Washington. His current research involves compiler optimizations for collective communication primitives, portable software support for efficient collective communication libraries, and parallel programming language design.