Journal of Parallel and Distributed Computing 62, 1493–1526 (2002)
doi:10.1006/jpdc.2002.1862

Portable and Scalable Algorithm for Irregular All-to-All Communication[1]

Wenheng Liu, Cho-Li Wang,[2] and Viktor K. Prasanna[3]

Department of EE-Systems, University of Southern California, Los Angeles, California 90089-2562
E-mail: liu@halcyon.usc.edu; clwang@cs.hku.hk; prasanna@halcyon.usc.edu

Received July 27, 1998; accepted April 11, 2002

In irregular all-to-all communication, messages are exchanged between every pair of processors. The message sizes vary from processor to processor and are known only at run time. This is a fundamental communication primitive in parallelizing irregularly structured scientific computations. Our algorithm reduces the total number of message start-ups. It also reduces node contention by smoothing out the lengths of the messages communicated. As compared to the earlier approaches, our algorithm provides deterministic performance and also reduces the buffer space at the nodes during message passing. The performance of the algorithm is characterised using a simple communication model of high-performance computing (HPC) platforms. We show the implementation on T3D and SP2 using C and the message passing interface standard. These can be easily ported to other HPC platforms. The results show the effectiveness of the proposed technique as well as the interplay among the machine size, the variance in message length, and the network interface. © 2002 Elsevier Science (USA)

1. INTRODUCTION

Irregular structured algorithms are widely used to solve problems in scientific computations, computer vision and database applications [7–9, 12, 25, 30, 31]. While parallelizing these algorithms, irregular all-to-all communication arises. In irregular all-to-all communication, each node sends a distinct message to every other node in which the size of the message to be sent varies from node to node. This paper proposes a portable multi-stage algorithm to perform frequent irregular all-to-all communication on various high-performance computing (HPC) platforms. Our initial work in designing the algorithm appears in [31]. That work was motivated by

[1]This research was supported in part by ARPA under Contract DABT63-95-C0092.

[2]Supported in part by NSF under Grant CCR-9317301. Current address: Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong.

[3]To whom correspondence to be addressed.

0743-7315/02 $35.00
© 2002 Elsevier Science (USA)
All rights reserved.

the need to perform irregular and data-dependent communication operations in parallelizing intermediate and high-level vision problems.

A straightforward approach to perform irregular all-to-all communication is to send a message to each node one by one. However, this simple approach is inefficient due to the large start-up latency and possible node contention. Start-up latency is the major source of overhead in message passing particularly on loosely coupled platforms. The variance in the message size causes node contention and, thus results in the serialization of the message start-ups.

Several algorithms have been proposed to perform all-to-all communication [5, 16, 20, 24] motivated by particular topologies, e.g., mesh, hypercube, etc. Besides, the variance of the message size is not considered in the design of these algorithms. In this paper, we design and analyze an efficient algorithm for irregular all-to-all communication. The algorithm is suitable for state-of-the-art HPC platforms.

Our algorithm consists of four stages. The first two stages perform message distribution, while the rest of the stages perform message collection. In each stage, the nodes are partitioned into several groups. Within each group, an all-to-all communication is performed. At each node, some local memory accesses are performed to compose the messages to be communicated. The algorithm reduces the serialization of associated message passing start-ups (or node contention) by balancing the length of the messages communicated in each stage. It also reduces the overall communication latency by reducing the number of message start-ups. For comparison purpose, we estimate the performance of the algorithm based on "flat" communication model. This model captures the features of the interconnection networks of HPC platforms in which the software overheads dominate the hardware latencies [22, 30]. In this model, let L_{max} denote the maximum traffic (in bytes) at a node. Let T_s be the start-up overhead for the main processor to traverse through the software layers to send and receive a message, t_d denotes the data transfer time per byte, t_c the data copy latency per byte between local memory and interface, and t_m the latency of local memory access per byte. Given P nodes, the total time to perform the algorithm is bounded by $(4\lceil\sqrt{P}\rceil + 2)T_s + 4(\lceil\sqrt{P})\lceil\sqrt{P}\rceil \frac{L\text{max}}{P}(t_m + t_c + t_d)$. Also, our algorithm significantly reduces the buffer space required at each node during the communication. The size of each of the send buffer and the receive buffer is $\frac{(\lceil\sqrt{P}\rceil)^2}{P}L_{max}$ bytes.

Recently, related problems have been studied by Ranka *et al.* [26] and by Bader *et al.* [2]. Both algorithms reduce node contention and provide deterministic performance. In these algorithms, the number of message passing start-ups is doubled; therefore, the start-up latency dominates the overall communication time when the total traffic is light.

Portability has been an important consideration in implementing our algorithm. The message passing interface (MPI/MPI-2) provides a framework for portable algorithm design and is supported on most state-of-the-art HPC platforms. The current version of MPI provides the functionality to perform all-to-all communication. However, there is no Specific optimizations proposed to improve the performance when message size varies from node to node. The algorithms were

implemented using C and the MPI point-to-point communication primitives. Our techniques, nonetheless, can be exploited to optimize performance of MPI primitives. They can also be exploited at the lower level (operating system and network interface level) using machine-specific features primitives to further improve the performance of communication primitives.

Some platforms support nonblocking communication mode. On these platforms, message composing and message communication can be performed in a pipelined fashion. We can, therefore, overlap communication with local memory access. The implementation of our approach is shown using such nonblocking primitives. These designs are advantageous in realizing throughput-oriented implementation in using HPC technology for image and signal processing applications.

To verify the effectiveness of our approach, we compare the performance of our algorithm against the straightforward single-stage approach (denoted as A1) and two prior algorithms in [2, 26] (denoted as A2) and in [17] (denoted as A3). All the algorithms were implemented on SP2 and T3D. The experimental results can be summarized as follows: Compared with A1 and A2, our algorithm reduces the number of messages communicated. This improves the performance when blocking communication mode is used. Compared with A1 and A3, our algorithm reduces node contention as well as minimizes the buffer space needed at each node. Overlapping communication with local memory access using nonblocking communication mode further improves the performance of our algorithm on SP2.

The rest of the paper is organized as follows. Section 2 describes the latencies in performing irregular communication. Section 3 describes the algorithms and implementation details. Section 4 summarizes our experimental results on T3D and SP2 and compares it with those obtained by earlier approaches. Section 5 concludes the paper.

2. IRREGULAR ALL-TO-ALL COMMUNICATION

In this section, we define a model for the cost of performing irregular all-to-all communication on state-of-the-art HPC platforms. Section 2.1 discusses a general communication system of HPC platforms and defines a "flat" communication model for performance analysis. Section 2.2 defines blocking and nonblocking communication modes. Section 2.3 models the communication time to perform irregular communication and identifies the latencies induced by node contention.

2.1. Communication Latencies

In the state-of-the-art HPC platforms, the communication bottleneck is usually not the bandwidth of the network fabrics. Instead, latencies induced at the sender and the receiver cause the bottleneck [10, 18]. These include latencies introduced by message start-ups and memory copying. Figure 1 illustrates the steps in sending and receiving a message in a typical state-of-the-art HPC platform. m denotes the message size in bytes. The latency of a message passing Operation is the total time spent by the message to traverse through the communication path from the sender to the receiver. First, data stored in noncontiguous memory locations are coalesced to a

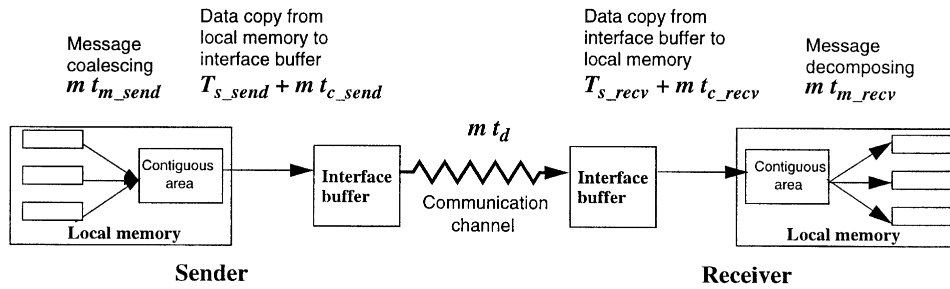


FIG. 1. Major steps in message passing in a typical state-of-the-art HPC platform.

message in the local memory of the sender. The message is then copied to the network interface buffer at the sender, transferred through the network, and copied from the interface buffer to the local memory at the receiver. Then, the received message is decomposed and stored into the specified locations in the local memory of the receiver. The latencies illustrated in Fig. 1 are defined as follows.

- t_{m_send} (t_{m_recv}): message coalescing (decomposing) latency per byte. This time is spent to perform memory copy in local memory to compose (decompose) the message.
- T_{s_send} (T_{s_recv}): start-up overhead incurred by the main processor to traverse through the software layers to initiate a message passing operation at the sender (receiver).
- t_{c_send} (t_{c_recv}): memory copy time per byte between the local memory and the interface buffer at the sender (receiver).
- t_d: data transfer time per byte from the interface buffer at the sender, through the network to the interface buffer at the receiver.

For the sake of convenience, we define $T_s = T_{s_send} + T_{s_recv}$, $t_c = t_{c_send} + t_{c_recv}$, and $t_m = t_{m_send} + t_{m_recv}$.

In our definition, T_{s_send} and T_{s_recv} are independent of the message size. These parameters depend on the architectural features of the platform as well as the network protocol used. Generally, these start-up overheads consist of the transfer request and acknowledgment latencies, context switching latency, and the time to generate the header of the message to be sent, t_{c_send} and t_{c_recv} are the unit data copy time between the user memory and the interface buffer at the sender and the receiver respectively. Some network protocols first copy the data from the user memory to the system space and then copy the data from the system space to the interface buffer. The additional memory copy cause larger t_{c_send} and t_{c_recv}. t_d is the inverse of the transfer rate between the interface buffers of the sender and the receiver. We assume that the network is lightly loaded with a low probability of channel contention at the intermediate nodes or the network switches. This assumption is reasonable in state-of-the-art HPC platforms in which the start-up overhead is in the range of tens of micro-seconds and the switch latencies are in the range of tens of nano-seconds. Besides, since modern HPC platforms use techniques such as worm-hole routing and virtual channels to achieve a constant data transfer rate, t_d is

independent of the number of hops the message traverses. Thus, as a first-level approximation, t_d is not affected by the network topology.

The overall latency to send a message can be modeled by two components: the message size invariant part (T_s) and the message size dependent part ($m \times (t_m + t_c + t_d)$). This model is referred as a "flat" communication model throughout the paper. Note that this model is used to estimate the performance of the communication algorithms described in this paper for comparison purposes. The exact communication times are shown based on the experimental results (see Section 4). The flat model is widely used in analyzing communication time on state-of-the-art HPC platforms [3, 15, 21, 22, 30]. Note that, when a collection of messages is transferred between a pair of nodes, the throughput can be increased by pipelining the communication steps.

On some platforms such as CM5, the processors at the sender and the receiver participate in the entire message passing operation; the processor at the sender injects the packets from the local memory into the network, and the processor at the receiver extracts the packets into the local memory. Overlapping of message passing steps is impossible. On some other platforms such as T3D, SP2, and Myrinet-based multicomputers [13], part of the message passing operations are off-loaded from the main processor. This is achieved by using a message passing processor (usually a micro-processor along with some customized circuitry embedded in the network interface). SP2 and Myrinet also provide direct memory access (DMA) engines to copy the data from the local memory to the interface buffer or vice versa. These further reduce the time the main processor is engaged in message passing.

The communication features of state-of-the-art HPC platforms, typical values of the latencies, and transfer rates are shown in Table 1. The table Is based on [10, 11, 18, 29] and our own measurements. It should be noted that the numbers vary depending on the version of the software environment used for message passing.

2.2. *Blocking and Nonblocking Communication Mode*

To utilize the various architectural features of state-of-the-art HPC platforms, MPI standard [14] defines several communication modes. We briefly define the

TABLE 1

Communication Features of HPC Platforms

Platform	DMA engine	Message passing processor	Program interface	T_d (μs)	$t_c + t_d$ (μs/byte)
CM5	No	Main processor	CMMD	85	0.12
SP2	Yes	Co-processor	MPICH	44	0.035
T3D	Yes	Support circuitry	MPICH	38	0.043
T3E	Yes	Support circuitry	MPI-EPCC	28	0.006
Sparc station + Myrinet + LANai interface	Yes	Microprocessor in LANai board	Myrinet API	68	0.027

blocking and nonblocking communication modes. Additional details can be found in [14].

As defined in MPI [14], in blocking send operation, the main processor of the sender is not free until the message data are safely stored in the interface buffer of the sender so that the send buffer in the local memory is available, if needed. In blocking receive, the main processor of the receiver is not free after the receive command is posted and until the receive buffer contains the message.

Nonblocking communication permits overlap of communication with computation [14]. In nonblocking send, the main processor at the sender starts the communication and resumes the user computation as early as possible. In nonblocking receive, the main processor is free after it posts the receive command. A corresponding waiting command is required to complete the nonblocking send or receive. It indicates when the send or receive buffer is available for further computation. For consecutive message passing operations, the main processor can be used to perform message coalescing or decomposing concurrently with the message copy operation performed by the DMA engine and the message transfer performed by the message passing processor.

2.3. Latencies in Performing Irregular All-to-All Communication

In this section, we analyze the latencies in performing irregular all-to-all communication using the parameters introduced in Section 2.1 and using the communication modes introduced in Section 2.2. In addition, we introduce an aggregate node contention factor. This framework motivates the design of our algorithm as well as analyzes the performance of various algorithms. These are discussed in Section 3.

We first estimate the time to perform an all-to-all communication among P nodes. The size of each message is m bytes. We assume that the P messages to be communicated are composed first. A barrier synchronization is performed after message composition, followed by the communication. For the sake of modeling the algorithms developed in Section 3, the received messages are not decomposed. Therefore, $t_{m_recv} = 0$. Using a simple model of blocking and nonblocking modes, the performance can be analyzed as follows:

(1) *Using blocking mode*. The communication is performed in P steps. In each step, a node sends a message and then receives a message. Each step of message passing takes $T_s + m(t_c + t_d)$ time. The send command in the next step cannot be issued until the completion of receive in the current step. The composition of messages takes Pmt_m time. So, the total execution time can be estimated as

$$T_{block} = P[T_s + m(t_c + t_m + t_d)]. \tag{1}$$

(2) *Using nonblocking mode*. Nonblocking communication mode permits pipelined operations in message passing. To simplify the analysis, each node sends the p outgoing messages first and then receives the p incoming messages. We assume that, at each node, the main processor spends PT_s, time to send and receive the messages, the DMA engine spends Pmt_c time to copy data between the local memory and interface buffer, and the message passing processor spends Pmt_d time to transfer

the messages between the interface buffers of the sender and the receivers. We also assume that the largest latency among these three hides the latencies introduced by the operations in the other stages of the pipeline. A simple approximation to the total communication time is

$$T_{nonblock} = (Pmt_m) + P\, Max(T_s, mt_c, mt_d). \quad (2)$$

When irregular all-to-all communication is performed, additional communication latencies occur due to node contention [18, 19, 23, 27, 28]. Node contention occurs when message compete for the buffer space and the input/output ports. This scenario causes serialization of communication operations at the node. Figure 2 illustrates a scenario depicting node contention in irregular communication.

At time $T1$, nodes P_0, P_1, P_2, P_3 start sending a message to P_1, P_2, P_3, P_0 respectively. Assume that P_0 completes its transfer at time $T2$ and starts sending its next message to P_2. At this time, there is a potential node contention at P_2. Similarly, at time $T3$, another potential node contention occurs at P_0 since both P_2 and P_3 concurrently attempt to send messages to P_0.

Figure 3 illustrates the time line for irregular communication using blocking mode. There is a node contention at P_2 since P_0 and P_1 send messages to P_2 at the same time. Possible delays due to node contention are:

1. The incoming messages R_{02} and R_{12} pass through the limited bandwidth of the input port(s) at the receiver. Thus, t_d's for R_{02} and R_{12} increase. These cause the delays d_1 and d_2 respectively.

2. Since the completion of sending S_{12} is delayed, P_1 postpones the send of the next outgoing message S_{13}. This causes delay d_3 at P_3.

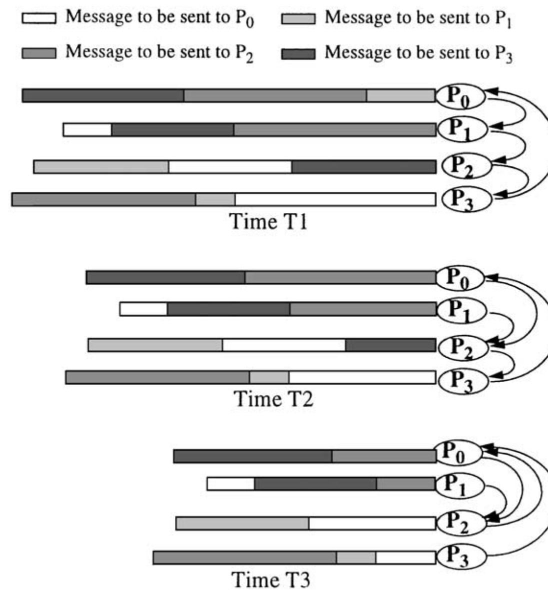


FIG. 2. A scenario depicting possible node contentions in performing irregular communication.

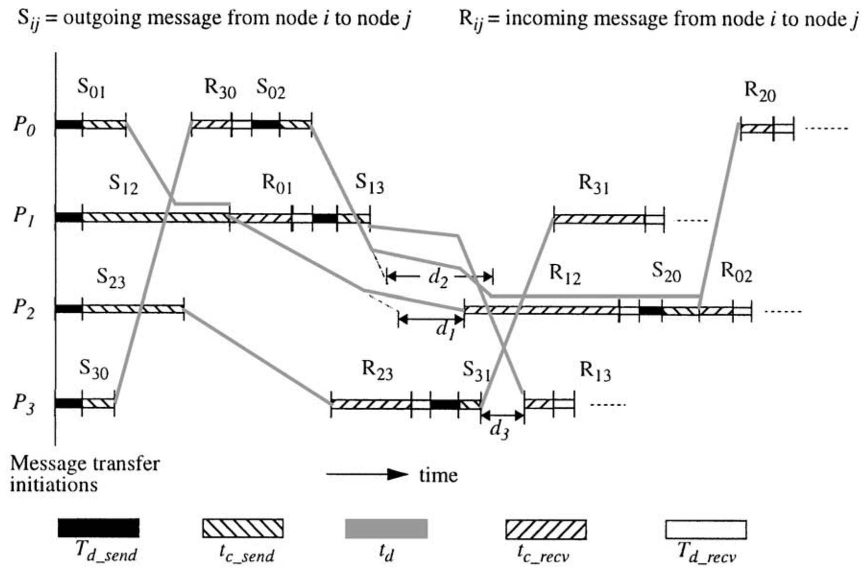


FIG. 3. A possible scenario illustrating latencies in performing an irregular all-to-all communication among four nodes.

Using the parameters introduced in Section 2.1, a simple approximation to the total time to perform all-to-all communication among P nodes is

- *blocking mode*:

$$T_{block} = PT_{s_send} + L_{max}(t_c + t_m + f_{nc_b}t_d); \tag{3}$$

- *nonblocking mode*:

$$T_{nonblock} = L_{max}t_m + Max(PT_s, L_{max}t_c, L_{max}f_{nc_nb}t_d), \tag{4}$$

where L_{max} is the maximum traffic to be communicated at a node. f_{nc_b} and f_{nc_nb} are the aggregate node contention factors in blocking and nonblocking communications respectively. f_{nc_b} and f_{nc_nb} are generally greater than 1. They are equal to 1 if there is no node contention. As shown in Fig. 3, node contention delays the following input or output, and causes serialization of communication operations at a node.

Note that the above estimate is an approximation to the actual time. The above equations, even though simplistic, illustrate two problems that can arise in performing irregular all-to-all communication: latencies due to message start-ups and node contention. Equations (1)–(4) can be used to understand the performance of known algorithms as well in deriving the proposed solution. Indeed, the performance of the proposed algorithms can be fairly accurately predicted by using the parameters defined in this section.

3. ALGORITHMS AND THEIR PERFORMANCE ANALYSIS

In this section, we develop a four-stage algorithm to perform irregular all-to-all communication and compare its performance with those of three earlier algorithms based on the model discussed in Section 2.3.

The algorithms are denoted A1–A4. A1 is a straightforward algorithm. A2 is the algorithm proposed by Ranka [26] and Bader [2]. A3 is a well-known algorithm motivated by the two-dimensional mesh architecture. A3 and its implementation was proposed in [16, 20]. A4 is our four-stage algorithm.

Our goal is to develop an efficient algorithm that is portable onto various state-of-the-art HPC platforms. As discussed later in this section, the algorithm is designed to suit an arbitrary number of processors. The previous approaches assume that the number of nodes required to satisfy a particular property (e.g., a perfect square, a power of 2, etc.) matches the specific architectural configuration and network topology. Our approach does not make this assumption. This makes it attractive particularly on some platforms, such as networks of workstations (NOWs) and switch-based multicomputers that allow a variety of system configurations. These systems consist of an arbitrary number of nodes embedded on a platform. Also, in solving many problems, the nodes on HPC platforms are partitioned into groups. Each group is assigned to perform a particular task. The size of a group depends on the computational power required and the degree of parallelism of the task being performed.

The nodes participating in communication are numbered 0 through $P-1$. The size of the messages to be communicated is represented using a $P \times P$ matrix M (see Fig. 4). At each node, the data to be communicated are initially stored in an arbitrary order. We assume that all the nodes are synchronized at the beginning of the communication.

In Fig. 4, M_{ij} denotes the size of the message to be sent from node i to node j. Let M_{max} be the size of the longest message, $Lout_i$ and Lin_j the total outgoing traffic at node i and the total incoming traffic at node j respectively. We define L_{max} to be the maximum of all the incoming and outgoing traffic among the P processors. Thus,

$$M_{max} = \max_{0 \leqslant i,j < P} M_{ij},$$

$$Lout_i = \sum_{k=0}^{P-1} M_{ik},$$

$$Lin_j = \sum_{k=0}^{P-1} M_{kj},$$

$$L_{max} = \max_{0 \leqslant i,j < P} (Lout_i, Lin_j).$$

To simplify the explanation, we assume that each processor sends a message to itself from its send buffer to its receive buffer.

3.1. Previous Algorithms

For the sake of completeness, we describe A1–A3 before describing A4.

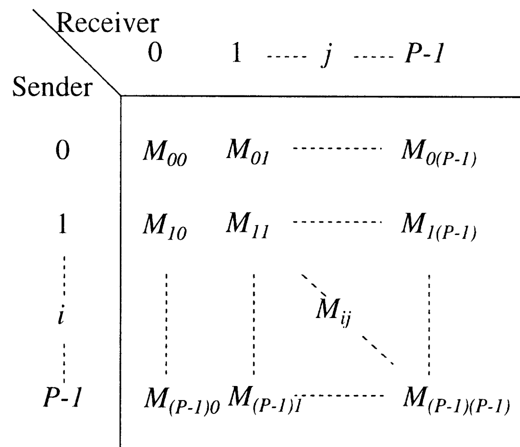


FIG. 4. Table-based formalism representing all-to-all communication.

3.1.1. Algorithm A1. A straightforward algorithm is a one-stage algorithm. Each node sorts its data by their destination to compose P messages and routes the messages to their destinations in P steps. A simple communication schedule is used: during the ith step, $1 \leqslant i \leqslant P$, node j sends the message destined for node $(i+j)\%P$, $0 \leqslant j < P$, where % denotes the *modulo* operator. As discussed in Section 2.3, the communication schedule alone does not solve the node contention problem.

3.1.2. Algorithm A2. Ranka *et al.* proposed a two-stage algorithm to reduce node contention [26]. The algorithm decomposes many-to-many communication with a high message size variance into two many-to-many communication stages with low message size variance.

At each P_i, $0 \leqslant i < P$, the elements to be sent to the receiver P_j, $0 \leqslant j < P$, are evenly divided into P slices. At each node, for each j, the P slices destined for P_j are distributed to P outgoing messages.

Then, an all-to-all communication is performed among the P nodes. In the second stage, an all-to-all communication is performed to send the data to their destination. The variance of the message size is reduced in both the stages. This algorithm was implemented on CM5 using Active Message. Hence the portability of the code is limited.

To perform the algorithm, the data need to be sorted by destination. After the sort operation, the messages to be communicated are generated by first marking the start and the length of each slice of sorted data. Then, the data are copied slice by slice to compose the messages. It takes $O(P^2)$ time to mark the slices in each stage.

A similar algorithm was proposed by Bader *et al.* [2] and was implemented using SPLIT-C on a variety of platforms. However, they assume that, initially, the data are stored in a buffer in an arbitrary order. They developed an algorithm to scan the buffer and evenly distribute the data elements to be sent to the same destination into P outgoing messages. The elements are moved to the send buffers one by one during the scan operation. Compared with the approach in [26], this algorithm reduces the

overall communication latency by avoiding sort of the data and identifying the slices before composing the messages. This algorithm also reduces the buffer space as compared to the approach in [26].

In some applications, data to be transferred are initially available in a sorted order by destination. If a platform provides a fast mechanism for block transfer, the approach to move data slice by slice may be more efficient than the one to scan the data and move data element by element to compose the outgoing messages. Marked slices of data can be exploited using MPI. MPI supports pack (unpack) primitives and derived data-type constructors. These primitives allow the users to specify the patterns to compose (decompose) a message from (to) noncontiguous location. However, we have observed that the current implementation of these primitives is inefficient. Thus, the scan-based composition of messages leads to faster implementation.

3.1.3. Algorithm A3. A3 [16] is a two-stage communication algorithm that combines messages to reduce the overall number of messages transferred. It is motivated by the use of mesh interconnected parallel machines. The algorithm combines the actual messages to form longer messages and redistributes the elements before sending them to their final destination. Hambrusch *et al.* [16] have implemented this algorithm for all-to-all communication. Their goal is to achieve better performance for short messages.

In this algorithm, the nodes are assumed to be arranged in a square mesh of size $\sqrt{P} \times \sqrt{P}$. Note that P is assumed to be a perfect square. P_{ij}, $0 < i,j < \sqrt{P}$, denotes the node at the ith row and the, jth column. The message to be sent from node $P_{i_1 j_1}$ to $P_{i_2 j_2}$ is delivered to an intermediate node $P_{i_1 j_2}$. It is then combined with other messages sent from $P_{i_1,j}$, $0 \leqslant j < \sqrt{P}$, $j \neq j_2$, and then sent to the final destination $P_{i_2 j_2}$. Since each node only communicates with the nodes in its own row in Stage 1 and in its own column in Stage 2, the total number of messages sent by any node is $2\sqrt{P}$.

Since the algorithm does not balance the traffic during an irregular all-to-all communication, the network bandwidth may not be fully utilized. Also, the algorithm may result in hot spots; each message is buffered in a pre-determined intermediate node. These nodes can become bottlenecks. This scenario not only increases the overall communication time but also results in poor utilization of buffer space.

Furthermore, when P is a not perfect square, the algorithm in its current plan does not work. Some intermediate nodes do not exist. In this case, some messages cannot be sent to their final destination. Straightforward extensions can lead to additional hot spots.

3.2. A Four-Stage Algorithm

We develop a four-stage algorithm to perform irregular all-to-all communication on a system having an arbitrary number of nodes. The algorithm balances the traffic by reducing the variance of the message size. It also reduces the number of messages communicated by each node by sending them to a smaller number of intermediate nodes.

To explain the algorithm, the nodes are assumed to be arranged in a two-dimensional array as shown in Fig. 5. Row-major ordering is used throughout the paper. $C = \lceil \sqrt{P} \rceil$ denotes the number of columns and $R = \left\lceil \frac{P}{\lceil \sqrt{P} \rceil} \right\rceil$ denotes the number of rows. When C divides P, each column is defined to be a complete column. When C does not divide P, let $r = P\% \lceil \sqrt{P} \rceil$. We define the first r columns to be complete columns while the last $C - r$ columns are defined to be incomplete columns. We also define the first $R - 1$ rows to be complete rows. The last row is defined as incomplete row if C does not divide P.

At each node, the send buffer, denoted $sBUF$, is partitioned into several buckets. Data stored in each bucket are sent to a receiver. The receiver stores the received data in its receive buffer, denoted $rBUF$. At each node, the data to be communicated are initially stored in $rBUF$ in an arbitrary order.

3.2.1. Overview of the four-stage algorithm. The first two stages perform message distribution, while the rest of the two stages perform message collection. To introduce the key ideas of the algorithm, we first consider the, case when P is divisible by C. The general case when P is an arbitrary integer is discussed in Section 3.2.2. Our four-stage algorithm is described as follows.

- *Stages I and II—message distribution.* The node array is partitioned into sets. In Stage I (II), all the nodes in a row (column) belong to the same set. Let N denote the number of nodes in each set; N equals C (R) in Stage I (II). Each node classifies its local data into P groups based on their destinations. It evenly divides each group into N segments, and collects segment i from each group into bucket i. There are N buckets in $sBUF$. The nodes in each set perform an all-to-all communication to exchange the corresponding data in the buckets.

During the first two stages, the variance of the message size is reduced. This reduces potential node contention. At the end of Stage II, the total data to be sent to the same destination are evenly distributed among all the P nodes.

- *Stages III and IV—message collection.* In Stage III, the node array is partitioned such that all the nodes in a row belong to a set. $sBUF$ in each node is

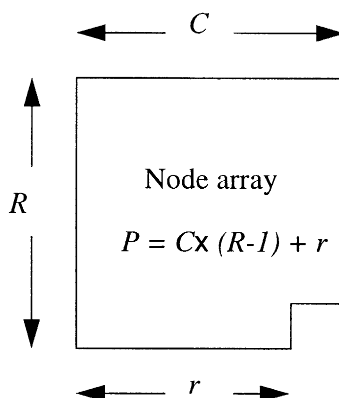
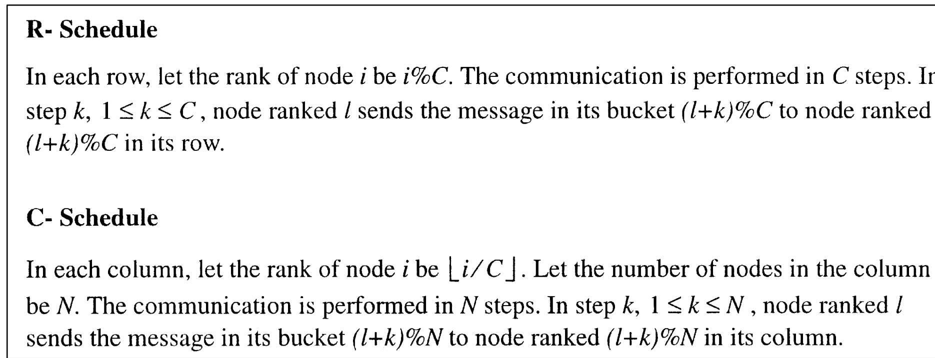


FIG. 5. Logical view of the nodes as a two-dimensional node array.



R- Schedule

In each row, let the rank of node i be $i\%C$. The communication is performed in C steps. In step k, $1 \leq k \leq C$, node ranked l sends the message in its bucket $(l+k)\%C$ to node ranked $(l+k)\%C$ in its row.

C- Schedule

In each column, let the rank of node i be $\lfloor i/C \rfloor$. Let the number of nodes in the column be N. The communication is performed in N steps. In step k, $1 \leq k \leq N$, node ranked l sends the message in its bucket $(l+k)\%N$ to node ranked $(l+k)\%N$ in its column.

FIG. 6. Communication schedules.

partitioned into C buckets. In each node, the data destined for the same column are stored in a bucket. An all-to-all communication is performed in each row. During the communication, the message in a bucket is sent to its destination column along its row. In Stage IV, the node array is partitioned such that all the nodes in a column belong to the same set. Data elements are classified according to their destination and stored into R buckets. Each column performs an all-to-all communication so that each datum is sent to its final destination. Since the data to be sent to the same destination are evenly distributed at the end of Stage II, the variance of the message size communicated in each step in Stages III and IV is reduced.

We develop an efficient algorithm to compose the messages to be communicated in each stage. It avoids sorting of elements at the beginning of each stage. The algorithm is presented in Section 3.2.2.

The communication in each stage can be scheduled so that each node receives exactly one message in any step. Figure 6 shows these schedules. **C-Schedule** is used to schedule the communication in each row in Stages I and III, and **R-Schedule** is used in each column in Stages II and IV. Note that the total number of communication steps performed is $2R + 2C$.

Figure 7 illustrates the message distribution and message collection between a pair of nodes. In Fig. 7(a), each message is sliced and evenly distributed into two buckets. Then, the communication step is performed to exchange the data such that the data

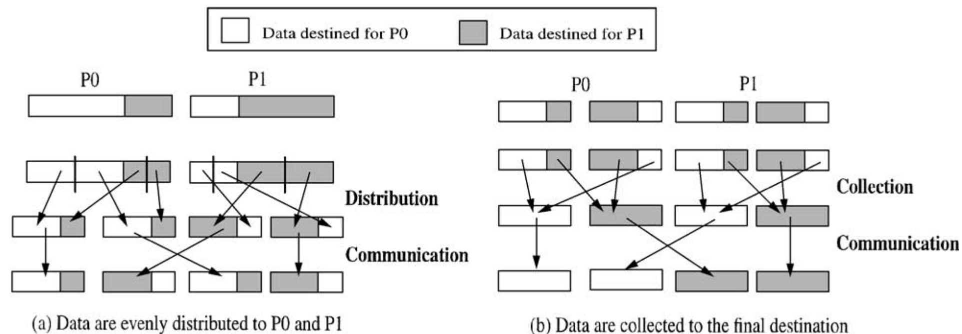


FIG. 7. Illustration of message distribution and message collection. (a) Data are evenly distributed to P0 and P1. (b) Data are collected to the final destination.

destined for each node are evenly distributed between the nodes. In Fig. 7(b), the data destined for the same node are collected and sent to their final destination.

3.2.2. Generalization of the algorithm.. As discussed at the beginning of Section 3, there can be an arbitrary number of nodes in a platform, or, a subset of the array participating in an irregular communication. In general, if P is not divisible by C, the algorithm introduced in Section 3.2.1 is incomplete; those nodes in the incomplete row cannot communicate with the nodes in the incomplete column(s).

A brute-force solution is to choose r nodes in a complete row and move all the data from each node in the incomplete row to a chosen node. The $(R-1)C$ node array, then, performs the four-stage algorithm as discussed in Section 3.2.1. After the four-stage communication, the data destined for the nodes in the incomplete row are transferred using an additional communication step. In the worst case, this approach doubles the maximum traffic (L_{max}). This not only degrades the performance but also increases the buffer size at each node. In addition, two extra steps of communication are required. In each of these steps, r messages are sent in parallel. These messages can be as long as L_{max}.

In this section, we generalize our algorithm by modifying Stages I and III. As compared to the brute-force approach, our algorithm is efficient and saves buffer space.

We modify Stages I and III by introducing additional communication steps. These communication steps are performed in concert with the all-to-all communication in each row so that the incomplete row can communicate with the incomplete column(s). In Stages II and IV, an all-to-all communication is performed among the R nodes in each complete column and among the $R-1$ nodes in each incomplete column.

Note that the node array is rearranged if needed to ensure that the number of nodes in the incomplete row (r) is less than or equal to the number of complete rows ($R-1$). To perform the additional communication without node contention, we design a priority communication schedule so that only $\lceil \sqrt{P} \rceil + 1$ communication steps are required in each of Stages I and III. Also, we balance the load to reduce the variance of the message size. The details of the generalized algorithm are shown below:

(A) *Additional messages for all-to-all communication in an incomplete array.* We define $C-r$ pseudo-nodes to fill the incomplete row of the node array. While performing all-to-all communication among the r nodes in Stages I and III, each node in the incomplete row sends a distinct message to each pseudo-node. The additional messages destined for each pseudo-node, however, are distributed to the nodes in the complete rows of an incomplete column. A total of $r(C-r)$ additional messages are communicated.

For each node in the incomplete row, the message sent from the node in the ith column, $0 \leqslant i < r$, to the pseudo-node in the jth column, $r \leqslant j < C$, is delivered to the node located in the ith row and the jth column. These additional messages and the mapping onto the incomplete columns are illustrated in Fig. 8 for the case of $P = 18$.

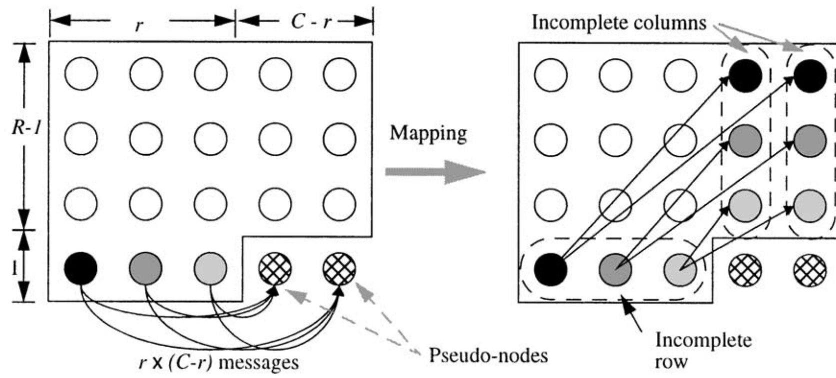


FIG. 8. Additional messages communicated when P is not divisible by C.

We need at least r complete rows to receive these additional messages, i.e., $R-1 \geqslant r$. Since $r \leqslant \lceil \sqrt{P} \rceil - 1$ and $R-1 = \left\lceil \frac{P}{\lceil \sqrt{P} \rceil} \right\rceil - 1 \geqslant \lceil \sqrt{P} \rceil - 2$, this condition is not valid when $P = \lceil \sqrt{P} \rceil \lfloor \sqrt{P} \rfloor - 1$. In this case, $r = \lceil \sqrt{P} \rceil - 1$ and $R - 1 = \lceil \sqrt{P} \rceil - 2$. We rearrange the array by defining C to be $\lfloor \sqrt{P} \rfloor$. This yields $r = \lfloor \sqrt{P} \rfloor - 1$ and $R - 1 = \lceil \sqrt{P} \rceil - 1$. Thus, there are sufficient number of complete rows to receive the additional messages.

(B) *Priority scheduling*. The $r(C-r)$ additional messages discussed above interfere with the all-to-all communication in each row performed in Stages I and III, resulting in node contention.

Figure 9 shows the communication schedule for the third row of the node array when $P = 61$ ($R = 8$, $C = 8$, $r = 5$). The nodes in the third row of the array perform an all-to-all communication, also node 58 sends additional message to each of the nodes 21, 22 and 23. The shaded area indicates the node contention. In the worst case (when $r = 1$), node contention occurs during $C - 1$ steps (in each of Stages I and III). To avoid this node contention, we can perform the all-to-all communication in each row first and then perform the additional communication. However, this causes $C - 1$ more steps in the worst case in each of Stages I and III.

We reschedule the communication steps using a priority schedule. The additional messages sent from the incomplete row have a higher priority. Figure 10 illustrates such a priority schedule.

In Fig. 10, we schedule the communication so that each node receives at most one message in a step. The additional messages to be sent from node 58 are scheduled first. Then, the communication from node 18 is scheduled. Nodes 18 and 58 send messages to node 21 in Step 3. The contention at node 18 is avoided by stalling the send in Step 3. Stalling the data from node 18 causes secondary contention at node 17. The contention at node 17 is eliminated by stalling the send in Step 4. A chain of contentions occurs among the rest of the nodes with lower priorities. These nodes are scheduled to eliminate the contention as shown in Fig. 10. The communication schedule is defined as follows:

Column no.	0	1	2	3	4	5	6	7	
Receiver / Sender / Step	16	17	18	19	20	21	22	23	58
Step 1	17	18	19	20	21	22	23	16	59
Step 2	18	19	20	21	22	23	16	17	60
Step 3	19	20	21	22	23	16	17	18	21
Step 4	20	21	22	23	16	17	18	19	22
Step 5	21	22	23	16	17	18	19	20	23
Step 6	22	23	16	17	18	19	20	21	56
Step 7	23	16	17	18	19	20	21	22	57
Step 8	16	17	18	19	20	21	22	23	58

FIG. 9. Communication schedule illustrating node contention.

Revised R-Schedule

- $C - r$ pseudo-nodes are defined to fill the incomplete row. **R-Schedule** shown in Fig. 6 is used to schedule each row. Note that the pseudo-nodes do not send any message.
- The messages destined for each pseudo-node are distributed to some nodes in the complete rows as discussed in (A).
- In the mth complete row, $0 \leqslant m < r$, for $0 \leqslant k \leqslant C - r + m$, the node ranked $(m + C - k)\%C$ in that row inserts a stall in Step $(r - m + 1 + k)$.

The above schedule results in an additional communication step in Stages I and III. However, no node contention occurs. Thus, Stages I and III are each completed in $\lceil \sqrt{P} \rceil + 1$ steps.

Column no.	0	1	2	3	4	5	6	7	
Receiver / Sender / Step	16	17	18	19	20	21	22	23	58
Step 1	17	18	19	20	21	22	23	16	59
Step 2	18	19	20	21	22	23	16	17	60
Step 3	19	20	X	22	23	16	17	18	21
Step 4	20	X	21	23	16	17	18	19	22
Step 5	X	21	22	16	17	18	19	20	23
Step 6	21	22	23	17	18	19	20	X	56
Step 7	22	23	16	18	19	20	X	21	57
Step 8	23	16	17	19	20	X	21	22	58
Step 9	16	17	18			21	22	23	

X: inserted stall

FIG. 10. Revised communication schedule.

(C) *Load balancing*. Associated with the modification in (A), we need to balance the load at the end of Stage II so that the variance of the message size can be reduced in the last two stages.

It should be noted that the complete columns have R nodes each, while each incomplete column has $R-1$ nodes. In Stage I, in each node, $\frac{R}{P}$ of the data destined for node j, $0 \leqslant j < P$, is distributed to each node in its row in the complete column and $\frac{R-1}{P}$ of this data is distributed to each node in its row in the incomplete columns. Then the messages in the nodes are evenly distributed along each column in Stage II to balance the load.

This distribution can be implemented by using a scan-based algorithm. This algorithm takes $O(L_{max})$ time. It composes the messages to be communicated on the fly as the elements are scanned and avoids sorting of the elements before balancing the load. A similar algorithm was performed in [2]. That algorithm, however, was developed to evenly distribute the load to each bucket.

The distribution primitive, **R-Distribute**, used in Stage I is defined in Fig. 11. In each node, data are distributed from *rBUF* to *sBUF*. *sBUF* at each node is partitioned into C buckets. We define a counter array, *count*[.], with P entries. The jth entry is used to count the number of elements destined for node j during the scan operation. $F(x) = (x\%P)\%C$ is a distribution function that specifies the bucket number to be used.

R-Distribute ensures that, in each node, for every P elements to be sent to node j, R of them are stored in each of the first r buckets, and $R-1$ of them are stored in each of the last $C-r$ buckets. The entries of the counter array are initialized to 0 through $C-1$. Thus, the first element destined for node j is stored in bucket $j\%C$. The purpose of the initialization of the counter array is to smooth out the message size in the buckets. The effectiveness of the initialization is proven in [2].

The distribution primitive, **C-Distribute** used in Stage II can be similarly defined. At each node in the complete (incomplete) columns, *sBUF* is partitioned into $R(R-1)$ buckets. The message stored in each bucket is sent to a node in its column, *count*[i], $0 \leqslant i < R$ ($0 \leqslant i < (R-1)$), is set to $i\%R$ ($i\%(R-1)$). The elements in each node are evenly distributed to the $R(R-1)$ buckets by using the distribution function $F(x)$. $F(x) = x\%R$ ($F(x) = x\%(R-1)$).

3.2.3. The complete four-stage algorithm. To describe the complete algorithm, we first define all-to-all communication primitives, **C-Comm** and **R-Comm**, and collection primitives, **C-Collect** and **R-Collect**.

```
R-Distribute

The following code is executed in each node.
  Set count[i] to i%C, 0 ≤ i < P , i denotes the destination index.
  Repeat
    If an element is to be sent to node j, it is stored in
        bucket F(count[j])=[(count[j]%P)%C] of sBUF.
    Increment count[j].
  Until rBUF is completely scanned
```

FIG. 11. Message distribution in Stage I.

C-Comm. The nodes are partitioned such that all the nodes in a column belong to a set. An all-to-all communication among the nodes in each column is performed. This all-to-all communication is performed in parallel for all the columns. **C-Schedule** shown in Fig. 6 is used to schedule the communication in each column.

R-Comm. The nodes are partitioned such.that all the nodes in a row belong to a set. An all-to-all communication with the nodes in each row along with the additional communication due to pseudo-nodes is performed in parallel for all the rows. **Revised R-Schedule** is used to schedule the communication.

C-Collect. In each node, *sBUF* is partitioned into C buckets. In each node, the elements to be sent to the kth column are collected into the kth bucket, $0 \leqslant k < C$.

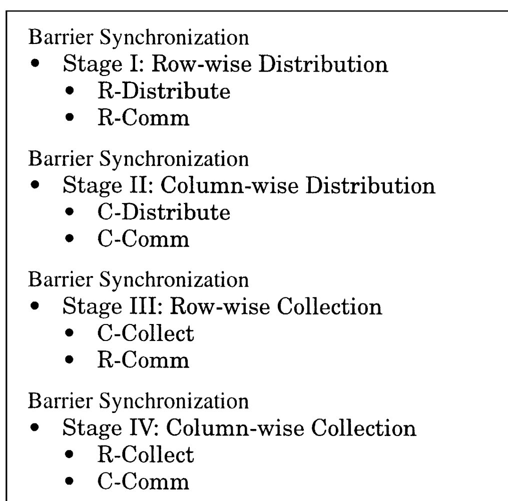
R-Collect. In each node in a complete (an incomplete) column, *sBUF* is partitioned into $R(R-1)$ buckets. The elements to be sent to the kth row are collected into the kth bucket, $0 \leqslant k < R$ $(0 \leqslant k < R-1)$.

Based on the primitives introduced in Section 3.2.2 and in this section, the four-stage algorithm is shown in Fig. 12. A barrier synchronization is performed at the beginning of each stage.

3.2.4. Performance analysis. We analyze first the number of messages received at each node and the maximum length of the messages communicated in each stage, then the buffer space requirements. An expression for the execution time is also derived based on the model described in Section 2.

We consider the case when $r > 0$. To simplify the presentation of our proofs, we assume that M_{ij} is divisible by P, $0 \leqslant i,j < P$.

LEMMA 1. *Any node in a complete column receives at most* $\lceil \sqrt{P} \rceil$ *messages in each stage. Any node in an incomplete column receives at most* $\lceil \sqrt{P} \rceil + 1$ *messages in Stages* I *and* III *and at most* $\lceil \sqrt{P} \rceil - 1$ *messages in Stages* II *and* IV.


Barrier Synchronization
- Stage I: Row-wise Distribution
  - R-Distribute
  - R-Comm

Barrier Synchronization
- Stage II: Column-wise Distribution
  - C-Distribute
  - C-Comm

Barrier Synchronization
- Stage III: Row-wise Collection
  - C-Collect
  - R-Comm

Barrier Synchronization
- Stage IV: Column-wise Collection
  - R-Collect
  - C-Comm

FIG. 12. The complete version of the four-stage algorithm.

Proof. Note that the messages are communicated when **C-Comm** or **R-Comm** is performed. The other steps are local operations.

When **C-Comm** is performed in Stages II and IV, there are at most $\lceil \sqrt{P} \rceil$ nodes in each column. So, any node in a complete column receives at most $\lceil \sqrt{P} \rceil$ messages including a message from itself.. There are at most $\lceil \sqrt{P} \rceil - 1$ nodes in an incomplete column. So, any node in an incomplete column receives at most $\lceil \sqrt{P} \rceil - 1$ messages.

When **R-Comm** is performed in Stages I and III, any node receives at most $\lceil \sqrt{P} \rceil$ messages from the nodes in its row. In addition, a node in an incomplete column receives an additional message if the node is in one of the first r rows. Therefore, a node in an incomplete column receives at most $\lceil \sqrt{P} \rceil + 1$ messages. ∎

Now, we show bounds on the length of the messages communicated during the execution of the algorithm. First, we show the property of data distribution at the end of Stage II in Lemma 2. This is used to derive the maximum length of the messages communicated in the last two stages.

LEMMA 2. *At the end of Stage II, the data elements to be sent to the same destination are evenly distributed among the P nodes.*

Proof. Consider the distribution of data elements destined for an arbitrary node j, $0 \leqslant j < P$, at the end of Stage II.

Let e_j be the data destined for node j. At any node k, $0 \leqslant k < P$, after **R-Distribute** is performed in Stage I, each of the first r buckets has $\frac{R}{P} M_{kj}$ of e_j. Each of the last $C - r$ buckets has $\frac{R-1}{P} M_{kj}$ of e_j. After **R-Comm** is performed:

(1) Any node is in a complete column has

- $\sum_{i=r_c C}^{(r_c+1)C-1} \frac{R}{P} M_{ij}$ of e_j, if the node is in a complete row r_c, for $0 \leqslant r_c < R-1$.
- $\sum_{i=(R-1)C}^{P-1} \frac{R}{P} M_{ij}$ of e_j, if the node is in the incomplete row.

(2) Any node in an incomplete column has

- $\sum_{i=r_{ic} C}^{(r_{ic}+1)C-1} \frac{R-1}{P} M_{ij} + \frac{R-1}{P} M_{sj}$ of e_j, if the node is in row r_{ic} for $0 \leqslant r_{ic} < r$. The second term is due to the additional message received from node s, where s is the r_{ic}th node in the incomplete row.
- $\sum_{i=r_{ic} C}^{(r_{ic}+1)C-1} \frac{R-1}{P} M_{ij}$ of e_j, if the node is in row r_{ic}, for $r \leqslant r_{ic} < R-1$. At the end of Stage I, each complete column has

$$\sum_{r_c=0}^{R-2} \sum_{i=r_c C}^{(r_c+1)C-1} \tfrac{R}{P} M_{ij} + \sum_{i=(R-1)C}^{P-1} \tfrac{R}{P} M_{ij} = \sum_{i=0}^{P-1} \tfrac{R}{P} M_{ij} \text{of} e_j.$$

Each incomplete column has

$$\sum_{r_{ic}=0}^{R-2} \sum_{i=r_{ic} C}^{(r_{ic}+1)C-1} \tfrac{R-1}{P} M_{ij} + \sum_{i=(R-1)C}^{P-1} \tfrac{R-1}{P} M_{ij} = \sum_{i=0}^{P-1} \tfrac{R-1}{P} M_{ij} \text{of} e_j.$$

There are R nodes in a complete column and $R-1$ nodes in an incomplete column. Since the elements are evenly distributed along each column in Stage II, at the end of Stage II, each node has $\sum_{i=0}^{P-1} \frac{M_{ij}}{P}$ elements destined for node j, $0 \leqslant j < P-1$. Thus, at the end of Stage II, the data elements to be sent to the same destination are evenly distributed among the P nodes. ∎

LEMMA 3. *The length of any message received by a node in a complete column is bounded by* $\lceil \sqrt{P} \rceil \frac{L_{max}}{P}$. *The length of any message received by a node in an incomplete column is bounded by* $(\lceil \sqrt{P} \rceil + 1) \frac{L_{max}}{P}$ *in Stages II and IV and* $(\lceil \sqrt{P} \rceil - 1) \frac{L_{max}}{P}$ *in Stages I and III.*

Proof. Let node $A(B)$ be an arbitrary node in a complete (an incomplete) column. In Stage I, the total length of the messages sent from any node i to node $A(B)$ is $\sum_{j=0}^{P-1} \frac{R}{P} M_{ij} (\sum_{j=0}^{P-1} \frac{R-1}{P} M_{ij})$, which is less than or equal to $R \frac{L_{max}}{P} ((R-1) \frac{L_{max}}{P})$.

Node $A(B)$ receives at most $C(C+1)$ messages in Stage I. In Stage II, the received data in node $A(B)$ are evenly distributed to $R(R-1)$ buckets. So the length of the messages transferred along a complete (an incomplete) column is bounded by $C \frac{L_{max}}{P} ((C+1) \frac{L_{max}}{P})$ in Stage II.

As proven in Lemma 2, each node contains $\sum_{i=0}^{P-1} \sum_{i=0}^{P-1} \frac{M_{ij}}{P}$ elements at the end of Stage II. In Stage III, the message sent from each node to a column is $\sum_j \sum_{i=0}^{P-1} \frac{M_{ij}}{P}$ in length, for every node j in that column. Since there are $R(R-1)$ nodes in each complete (incomplete) column; the length of the messages received by node $A(B)$ is bounded by the maximum of $R \sum_{i=0}^{P-1} \frac{M_{ij}}{P} ((R-1) \sum_{i=0}^{P-1} \frac{M_{ij}}{P})$, $0 \leqslant j < P$, which is less than or equal to $R \frac{L_{max}}{P} ((R-1) \frac{L_{max}}{P})$.

In Stage IV, at node $A(B)$, elements destined for the same node are collected into $R(R-1)$ buckets. Since there are at most $C(C+1)$ messages received by node $A(B)$, each bucket has $C \sum_{i=0}^{P-1} \frac{M_{ij}}{P} ((C+1) \sum_{i=0}^{P-1} \frac{M_{ij}}{P})$ elements to be sent to node j. So the length of the messages sent from node $A(B)$ is bounded by $C \frac{L_{max}}{P} ((C+1) \frac{L_{max}}{P})$.

Since $R \leqslant \lceil \sqrt{P} \rceil$ and $C = \lceil \sqrt{P} \rceil$, the length of the messages received by any node in a complete column is bounded by $\lceil \sqrt{P} \rceil \frac{L_{max}}{P}$. The length of the messages received by any node in an incomplete column is bounded by $(\lceil \sqrt{P} \rceil + 1) \frac{L_{max}}{P}$ in Stages II and IV and $(\lceil \sqrt{P} \rceil - 1) \frac{L_{max}}{P}$ in Stages I and III. ∎

Table 2 summarizes the message distribution in the system.

Based on Lemmas 1 and 3, we show a bound on the buffer space requirement in the following theorem.

THEOREM 1. *The total size of the buffer needed at any node is bounded by* $2(\lceil \sqrt{P} \rceil^2 \frac{L_{max}}{P})$.

Proof. From Lemmas 1 and 3, any node in a complete column receives at most $\lceil \sqrt{P} \rceil$ messages, and the length of each is bounded by $\lceil \sqrt{P} \rceil \frac{L_{max}}{P}$.

TABLE 2

Number of Messages Received at a Node and the Size of the Longest Message

Stage	Node in a complete column		Node in a incomplete column	
	No. of messages	Longest message	No. of messages	Longest messages
I	$\lceil\sqrt{P}\rceil$	$\frac{\lceil\sqrt{P}\rceil}{P}L_{max}$	$\lceil\sqrt{P}\rceil+1$	$\frac{\lceil\sqrt{P}\rceil-1}{P}L_{max}$
II	$\lceil\sqrt{P}\rceil$	$\frac{\lceil\sqrt{P}\rceil}{P}L_{max}$	$\lceil\sqrt{P}\rceil-1$	$\frac{\lceil\sqrt{P}\rceil+1}{P}L_{max}$
III	$\lceil\sqrt{P}\rceil$	$\frac{\lceil\sqrt{P}\rceil}{P}L_{max}$	$\lceil\sqrt{P}\rceil+1$	$\frac{\lceil\sqrt{P}\rceil-1}{P}L_{max}$
IV	$\lceil\sqrt{P}\rceil$	$\frac{\lceil\sqrt{P}\rceil}{P}L_{max}$	$\lceil\sqrt{P}\rceil-1$	$\frac{\lceil\sqrt{P}\rceil+1}{P}L_{max}$

On the other hand, in Stages II and IV, any node in an incomplete column receives at most $\lceil\sqrt{P}\rceil+1$ messages. The length of each message is bounded by $\lfloor\sqrt{P}\rfloor\frac{L_{max}}{P}$. In Stages I and III, each of these nodes receives at most $\lceil\sqrt{P}\rceil-1$ messages. The length of each message is bounded by $(\lceil\sqrt{P}\rceil+1)\frac{L_{max}}{P}$.

Since $(\lceil\sqrt{P})(\lceil\sqrt{P}\rceil+1)<\lceil\sqrt{P}\rceil^2$, the size of $rBUF$ is bounded by $\lceil\sqrt{P}\rceil^2\frac{L_{max}}{P}$. The data in $rBUF$ are copied to $sBUF$ in each stage. Thus, the size of $sBUF$ is bounded by $\lceil\sqrt{P}\rceil^2\frac{L_{max}}{P}$. Therefore, the total buffer size at any node is bounded by $2(\lceil\sqrt{P}\rceil^2\frac{L_{max}}{P})$. ∎

Based on the model introduced in Section 2, we derive an expression for the total execution time of the algorithm in terms of the start-up latency (T_d) and the latency proportional to the traffic size ($t_m+t_c+t_d$).

THEOREM 2. *The total execution time of the four-stage algorithm is* $(4\lceil\sqrt{P}\rceil+2)T_d+4(\lceil\sqrt{P}\rceil+1)\lceil\sqrt{P}\rceil\frac{L_{max}}{P}(t_m+t_c+t_d)$.

Proof. Note that a barrier synchronization is performed at the beginning of each stage. Thus, we can derive the execution time for each stage and sum up these times. The algorithm employs a conflict-free schedule in each stage. The execution time of a stage is bounded by the product of the number of communication steps and the time taken by in the longest communication step in the stage. As shown in Table 2, in each of Stages I and III, the length of the longest message is $\lceil\sqrt{P}\rceil\frac{L_{max}}{P}$. There are at most $\lceil\sqrt{P}$ steps as discussed in Section 3.2.2(B). Similarly, in each of Stages II and IV, the length of the longest message is $(\lceil\sqrt{P}\rceil+1)\frac{L_{max}}{P}$. There are at most $\lceil\sqrt{P}\rceil$

TABLE 3

Communication Characteristics of the Algorithms

Algorithm	Number of start-ups	Total traffic	Reduction of number of messages	Reduction of message irregularity	Comments
A1	P	L_{max}	No	No	Node contention occurs
A2	$2P$	$2L_{max}$	No	Yes	$2P$ messages are communicated
A3	$2\sqrt{P}$	$2L_{max}$	Yes	No	P is restricted to be a perfect square. Node contention and hot spots occur
A4	$4\lceil\sqrt{P}\rceil + 2$	$4\frac{\lceil\sqrt{P}\rceil(\lceil\sqrt{P})}{P}L_{max}$	Yes	Yes	—

communication steps. Thus, there are $\lceil\sqrt{P}\rceil + 2$ start-ups and the total traffic at each node is bounded by $4(\lceil\sqrt{P}\rceil + 1)\lceil\sqrt{P}\rceil\frac{L_{max}}{P}$ Thus, the total time to execute the algorithm is $(4\lceil\sqrt{P}\rceil + 2)T_d + 4(\lceil\sqrt{P}\rceil + 1)\lceil\sqrt{P}\rceil\frac{L_{max}}{P}(t_m + t_c + t_d)$. ∎

3.3. Comparison of the Algorithms

Table 3 summarizes the techniques employed by the four algorithms and their communication complexities. Table 4 lists the buffer requirements of the algorithms. The algorithms are compared in the following:

- Based on Table 3, our algorithm requires $O(\sqrt{P})$ number of message start-ups, while A1 and A2 require $O(P)$ number of start-ups. Thus, our algorithm is scalable over a wide range of processors compared with A1 and A2.
- As shown in Table 3, our algorithm is suitable for an arbitrary number of processors. However, A3 is restricted since it is only suitable for a perfect square number of processors.

TABLE 4

Send/Receive Buffer Requirements of the Algorithms

	A1	A2	A3	A4
sBUF	$M_{max}P$	L_{max}	$M_{max}P$	$\frac{\lceil\sqrt{P}\rceil^2}{P}L_{max}$
rBUF	L_{max}	L_{max}	$M_{max}P$	$\frac{\lceil\sqrt{P}\rceil^2}{P}L_{max}$

$M_{max} = \max_{0 \leqslant i,j < P} M_{ij}$,
$L_{max} = \max_{0 \leqslant i,j < P} (\sum_{m=0}^{P-1} M_{im}, \sum_{k=0}^{P-1} M_{kj})$.

• As proven in Theorem 2 in Section 3.2.4 and shown in Table 4, the size of the buffer needed at each node is $2(\lceil\sqrt{P}\rceil^2 \frac{L_{max}}{P})$. Compared with A1 and A3, A4 results in significant savings in buffer space. For instance, if $M_{max} = L_{max}$, then A3 needs P times the buffer space required by A4. The extremely large buffer size requirement at some nodes restricts the feasibility of A1 and A3.

• Our algorithm reduces the variance of size of the messages. Therefore, node contention is reduced. Node contention causes performance degradation in the case of A1 and A3.

3.4. A Nonblocking Version of the Four-Stage Algorithm

When nonblocking communication mode is used, the algorithms can be performed in a pipelined fashion to reduce the overall communication time. Figure 13 depicts the communication in Stages I and II of the four-stage algorithm. In Fig. 13(a), blocking mode is used. Overlap of the communication steps is not possible. In Fig. 13(b), nonblocking mode is used. The communication steps can be partially overlapped as discussed in Section 2. In Fig. 13(b), Stage II cannot start until the last send in Stage I is initiated. If Stage II starts before the initiation of any outgoing message in Stage I, **C-Distribute** moves the received data to the buckets of *sBUF*. However, some of these buckets still contain the outgoing messages to be transferred in Stage I.

The communication steps in the consecutive stages can be further overlapped if dual send buffers are available in each node. In Fig. 13(c), the data are first distributed from *rBUF* to one send buffer. Then **R-Comm** is performed and the messages are received in *rBUF*. Whenever a messages received, the data in that

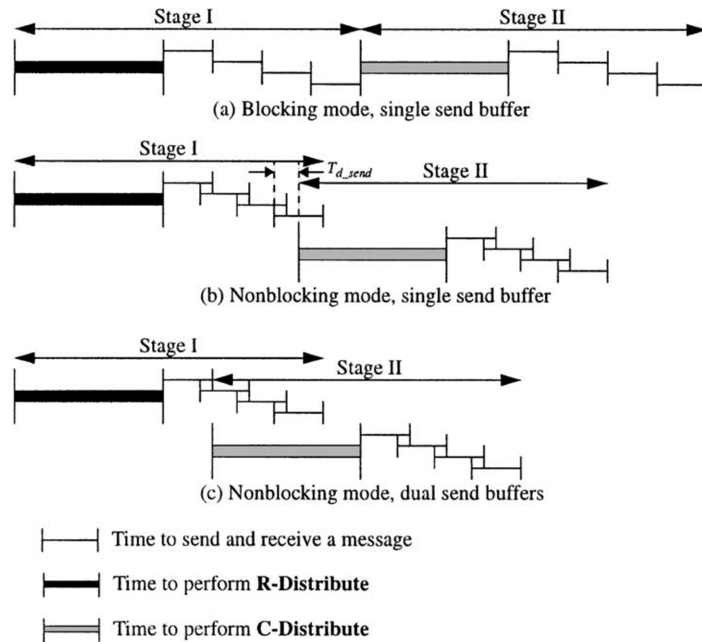


FIG. 13. Implementation of A4 using blocking and nonblocking communication modes.

message can be scanned and distributed immediately into the buckets of the other send buffer. The dual buffers can be used alternately during the stages to further improve the performance. The performance gain achieved by using dual send buffers is shown in Section 4.

4. EXPERIMENTAL RESULTS

A4, the proposed algorithm, as well as A1–A3 were implemented in C using the MPI primitives, *MPI_Send*( ), and *MPI_Recv*( ) for blocking communication, *MPI_Isend*( ), *MPI_Irecv*( ), and *MPI_Wait*( ) for nonblocking communication. The, algorithms were implemented on the CRAY T3D and the IBM SP2. Two communication patterns *Pattern*1 and *Pattern*2 were used in the experiments, and are defined in Fig. 14. Additional communication patterns and corresponding experimental results can be found in [23]. In these experiments, the number of data elements sent from a node to another node is bounded by M_{max}. An element is 48 bytes on T3D and 22 bytes on SP2. The experiments were conducted varying five parameters: (1) the number of nodes (P), (2) the longest message from a node to another node (M_{max}), (3) the communication pattern, (4) the communication mode, and (5) the use of single or dual send buffer (s).

We first compare the performance of A4 with those of A1 and A2 in Section 4.1. Comparison between A4 and A3 is shown in Section 4.2.

4.1. *Comparison of* A4 *with* A1 *and* A2

Figures 15 and 17 show the results of the experiments conducted on T3D and SP2. We assume $M_{max} = 1024$ and varied P. Figures 16 and 18 show the results when $P = 64$ and M_{max} is varied.

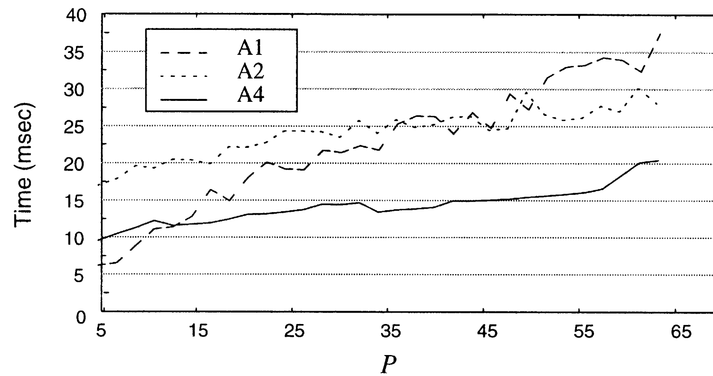
When blocking communication mode is used, A4 results in lower latency than that of A1 and A2. These results are shown in Figs. 15(a), 16, 17(a) and 18. Our algorithm saves a significant number of message start-ups and therefore, reduces the software overheads.

When nonblocking communication mode is used, there are two factors that contribute to performance gains:

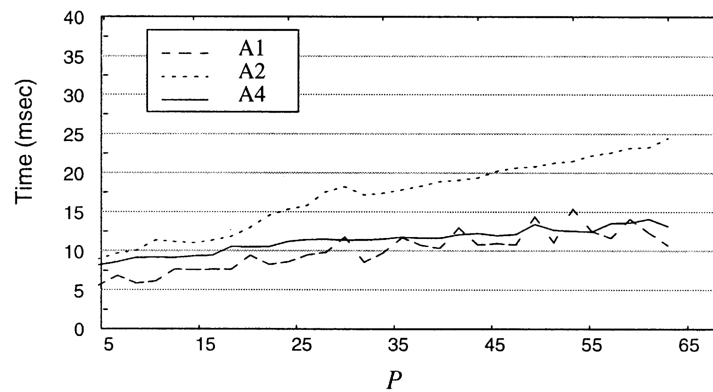
1. Message start-up serialization due to node contention can be reduced.
2. Message start-up and message composing (e.g., **R-Distribute** or **C-Collect**) can be overlapped with actual data transfer.

Pattern1 - One Spike	*Pattern2* - 2D Array Transpose
For each *node i*	For each *node i*
Send M_{max} elements to *node j*.	Send M_{max} elements to *node j*, $j=[(i\%c)C+\lfloor i/C \rfloor]$.
(Node j is randomly chosen at each node i)	*(Node i and node j are symmetric with respect to the diagonal of the logical 2D node array)*
Send a small number of elements to the rest of the nodes.	Send a small number of elements to the rest of the nodes.

FIG. 14. Pseudo-code illustrating the communication patterns.



(a) Blocking mode , *Pattern1*, M_{max}=1024



(b) Nonblocking mode , *Pattern1*, M_{max}=1024

FIG. 15. Comparison of A4 with A1 and A2 on T3D. (a) Blocking mode, *Pattern*1, $M_{max} = 1024$. (b) Nonblocking mode, *Pattern*1, $M_{max} = 1024$.

On T3D, as shown in Fig. 15, A1 results in a large performance gain when nonblocking mode is used. But when A2 and A4 are performed using nonblocking communication, the gains are relatively small. Since A2 and A4 reduce the variance of the size of the messages, the performance gain using nonblocking mode by reducing node contention is limited. The performance of nonblocking version of A2 or A4 is slightly better as compared to the corresponding blocking version. This implies that overlapping of message start-up and message composing with actual data transfer is not effective on T3D. The performance gain achieved by using the nonblocking version of A1 is mainly due to the reduction in start-up serialization.

On T3D, the message queue at each node is located in a reserved portion of the local memory. While the main processor performs message composition and the message passing processor performs actual data transfer, they use the same physical path to access the memory. It is not effective to overlap start-up and message composing latency with actual data transfer on T3D due to the limited memory bandwidth at each node.

On SP2, as shown in Fig. 17, the performance of the nonblocking version of A1 shows the effectiveness of reducing node contention. In Figs. 17 and 18, A2 and A4

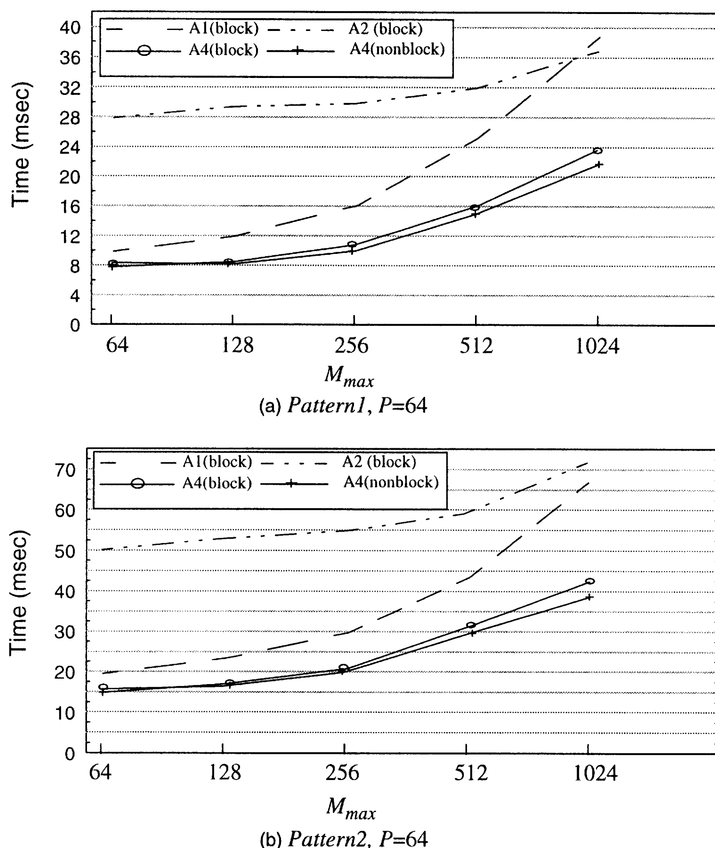


FIG. 16. Comparison of A4 with A1 and A2 on T3D. (a) *Pattern*1, $P = 64$. (b) *Pattern*2, $P = 64$.

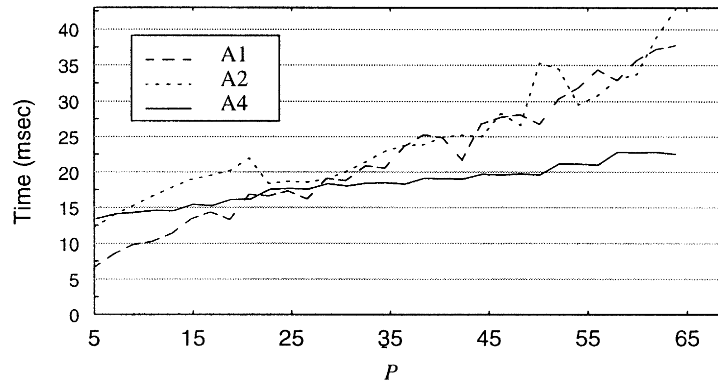
achieve significant performance gain when nonblocking mode is used. This shows the effectiveness in overlapping the message composition with communication on SP2.

4.2. Comparison of A4 *with* A3

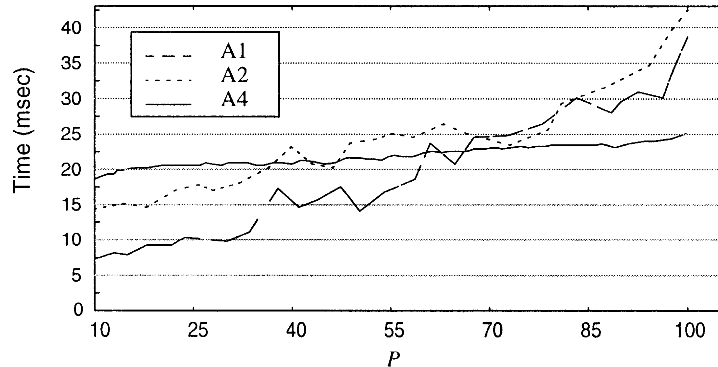
As shown in Table 3, among the four algorithms, A3 transfers fewest number of messages. This section compares A4 with A3 and shows the effectiveness of our algorithm in reducing the variance of the message size.

Figures 19 and 20 show the experimental results on T3D. Figures 21 and 22 show the experimental results on SP2. In addition to the blocking version, a nonblocking version of A3 with a single send buffer was implemented. Nonblocking versions of A4 with single and dual buffers were also implemented. Various latencies in performing A3 and A4 are shown in Figs. 20 and 22 to illustrate the time spent in message distribution (**R-Distribute** and **C-Distribute**), message collection (**R-Collect** and **C-Collect**), and communication (**R-Comm** and **C-Comm**).

On T3D, when 64 processors are used and $M_{max} \leqslant 1k$, A3 performs better than A4 for both *Pattern*1 and *Pattern*2. However, the communication time of A3 increases faster than that of A4 as the message size increases in *Pattern*2 (see Fig. 19(b)). As shown in Figs. 19(a) and 19(b), the implementation with dual buffers does not



(a) Blocking , *Pattern 1*, M_{max}=1024



(b) Nonblocking , *Pattern 1*, M_{max}=1024

FIG. 17. Comparison of A4 with A1 and A2 on SP2. (a) Blocking, *Pattern*1, $M_{max} = 1024$. (b) Nonblocking, *Pattern*1, $M_{max} = 1024$.

improve significantly the performance due to. limited memory bandwidth available at each node of T3D.

To further compare the performance of A4 with that of A3, we conduct the results for *Pattern*2, when $M_{max} \geqslant 1k$. Blocking communication mode was used. The results are shown in Fig. 20.

Although A4 requires two more stages, it has superior performance as compared to A3. The hot-spots encountered in A3 cause heavy traffic at the diagonal nodes and as well as node contention which result in much larger latencies when **C-Comm** or **R-Comm** is performed.

Experimental results on SP2 are shown in Figs. 21 and 22. A3-b, A3-n1, A4-b,A4-n1, A4-n2 denote various implementations of A3 and A4, where "b" denotes blocking, "n1" and "n2" denote nonblocking mode with single buffer and dual buffers, respectively. As discussed in Section 3.4, the operations between consecutive stages are partially overlapped in A4-n2, e.g., **R-Comm** in Stage I can be overlapped with **C-Distribute** in Stage II. In Fig. 22, each component defined in the second column of the legend represents the latency in performing a pair of overlapped operations.

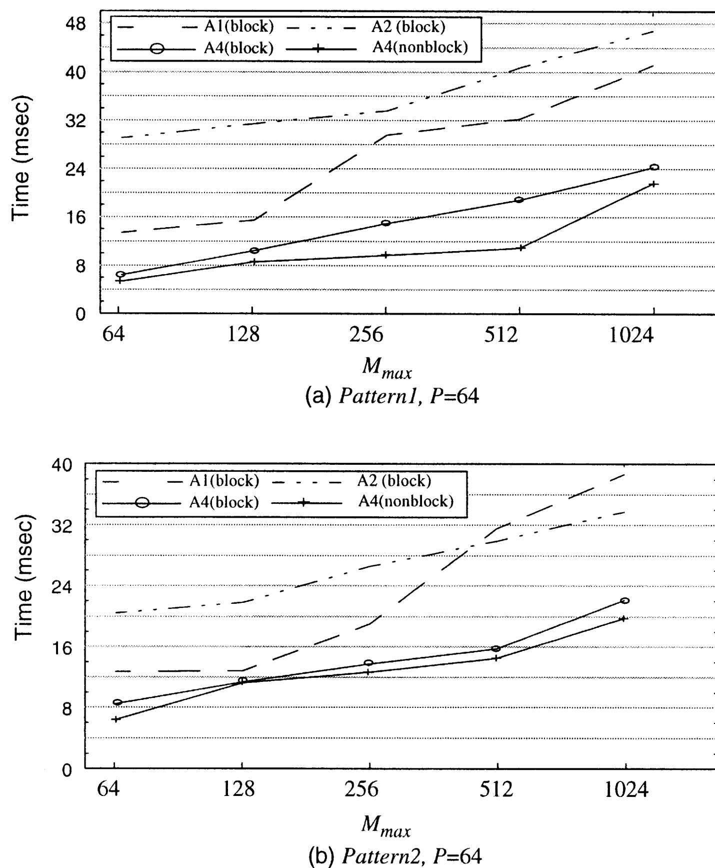


FIG. 18. Comparison of A4 with A1 and A2 on SP2. (a) *Pattern*1, $P = 64$. (b) *Pattern*2, $P = 64$.

As shown in Fig. 21 and 22, A3-n1 has superior performance for *Pattern*1, while A4-n2 has superior performance for *Pattern*2 as compared to the other algorithms. Nonblocking communication mode results in significant performance gains. The nonblocking version using dual end buffers performs better than that using a single send buffer as shown in Fig. 22. This shows the effectiveness of dual buffers in overlapping consecutive communication stages and hide the message start-up, the message coalescing, and decomposing latencies.

Fig. 23 shows the comparison of A4-n2 and A3-n1 when P varies from 25 to 100. Note that P needs to be a perfect square. Otherwise, A3 is not applicable. In this figure, A4-n2 is always superior to A3-n1. This experiment, again, shows the effectiveness of the balance of message sizes used in A4 when various number of processors are involved in irregular all-to-all communication.

5. CONCLUDING REMARKS

In this paper, we have shown an efficient algorithm for irregular all-to-all communication. This algorithm is suitable for state-of-the-art HPC platforms whose

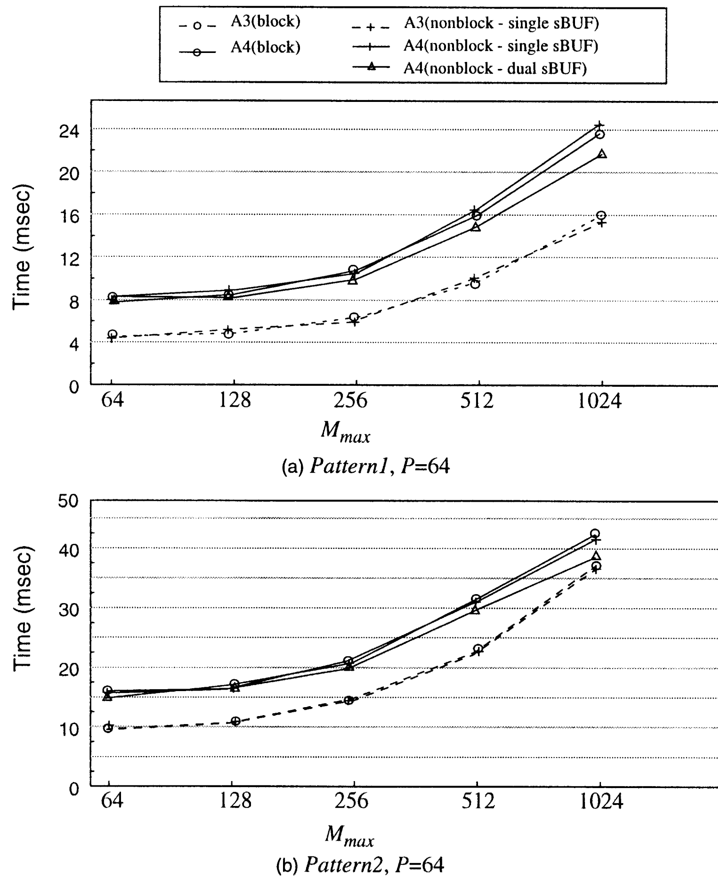


FIG. 19. Comparison of A4 with A3 on T3D when $M_{max} \leqslant 1k$. (a) *Pattern*1, $P = 64$. (b) *Pattern*2, $P = 64$.

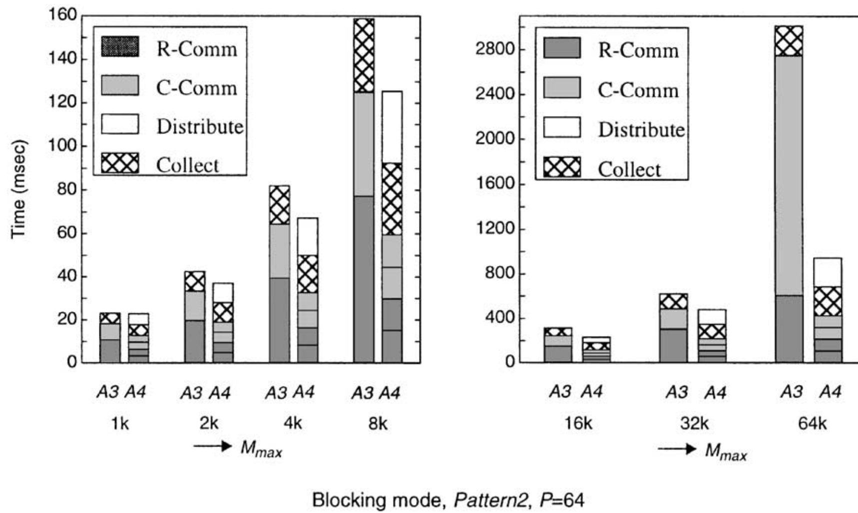
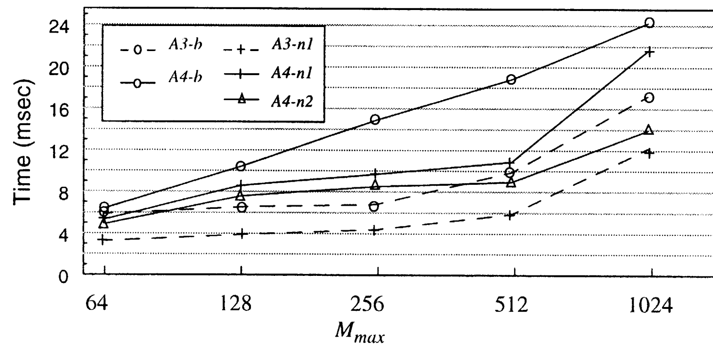
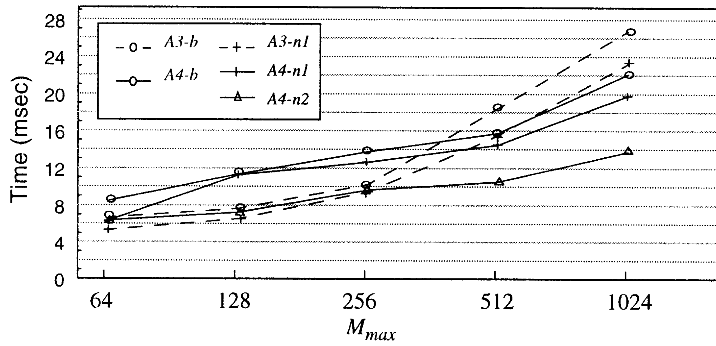


FIG. 20. Comparison of A4 and A3 on T3D when $M_{max} \geqslant 1k$.



(a) *Pattern1*, P=64



(b) *Pattern2, P*=64

FIG. 21. Comparison of A4 and A3 on SP2 when $M_{max} \leqslant 1k$.

communication network can be modeled as a "flat" network in which the software overheads dominate the hardware latencies. As shown in Table 1, currently, the ratio of $\frac{T_s}{t_c+t_d}$ is in the range of 1000. Note that $t_c + t_d$ depends on the processor speed as well as the network transfer rate. Thus, increase in processor and network

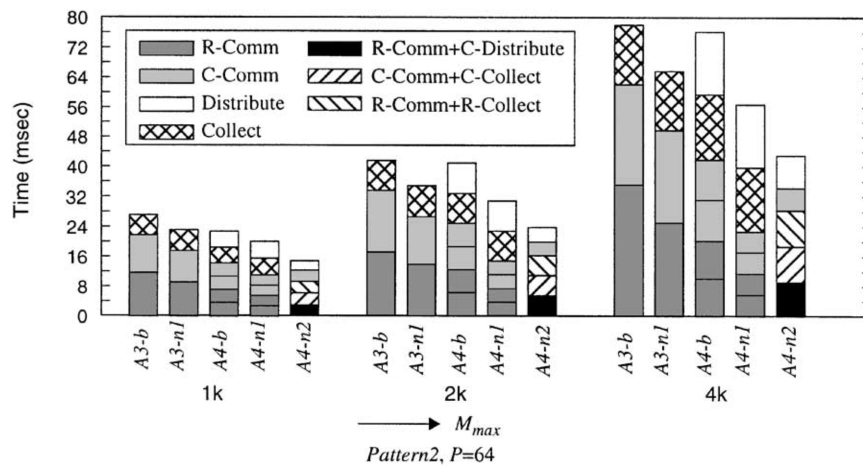


FIG. 22. Comparison of A4 and A3 on SP2 when $M_{max} \geqslant 1k$.

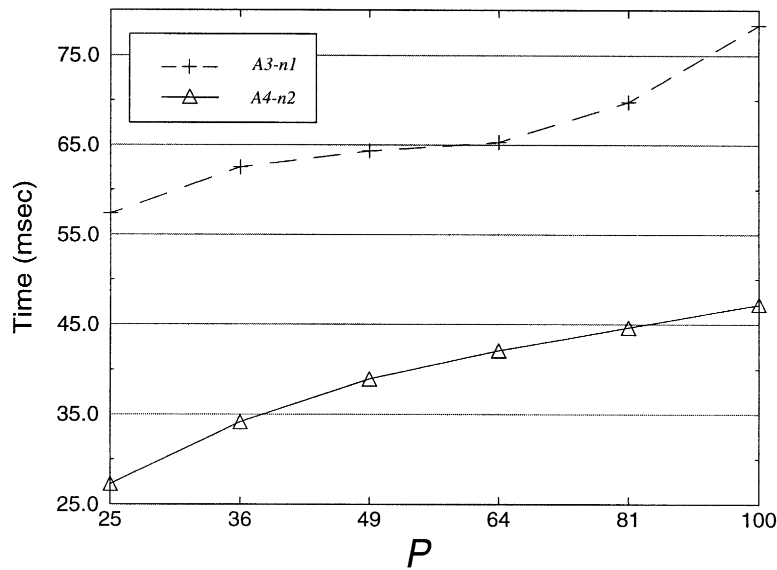


FIG. 23. Comparison of A4-n2 and A3-n1 on SP2 when P varies from 25 to 100 and $M_{max} = 4k$.

performance will improve $t_c + t_d$. Since reducing the software overhead in communication seems to be relatively difficult as compared to reducing the network hardware latencies, this ratio is likely to increase. Our algorithmic techniques will become even more useful with the increase in this ratio in the future.

Our algorithm is suitable for performing all-to-all communication with high variance in message size, and is particularly attractive as the number of nodes increases. The communication time is reduced by reducing the number of message start-ups and by smoothing out the variance of the message size. The performance can be further improved by overlapping the message composition with actual data transfer if message passing activity is partially off-loaded from the main processor. In addition, our algorithm is superior to the previous algorithms with respect to buffer space requirement.

Our algorithm can be generalized to perform other collective communication primitives such as multiple multi-casting [17, 19]. It can also be modified to perform all-to-all communication when senders and receivers belong to distinct groups and the number of senders is different from the number of receivers. When the number of nodes participating in the communication increases, the number of messages can be further reduced by increasing the number of stages. In each stage, all-to-all communication is performed in each group with a smaller number of nodes. There is, however, a penalty because more data rearrangement is required, resulting in the increase of the total traffic.

The overheads due to local memory accesses adversely affect our algorithm and it masks the potential performance gains. As shown in Section 3, non blocking message passing primitives can only partially reduce the latency on SP2 and T3D. The key to reduce the latency induced by memory accesses is to reduce the memory copy times encountered in the end-to-end communication path. On some HPC platforms,

message decomposing and coalescing can be performed directly between the local memory and the interface buffer. If such feature is available, then the message rearrangement in the local memory can be skipped.

Our algorithm was implemented using MPI. Thus, the code is portable to other platforms. We employed the MPI point-to-point communication primitives only. Therefore, the performance of the implementation is independent of the efficiency of the current MPI collective communication primitives. Our algorithm can be easily ported onto, other distributed systems such as NOWs and Myrinet-based multicomputers [13] that are being developed for embedded signal processing applications.

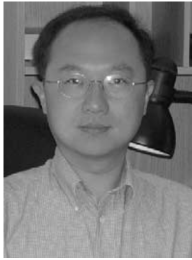
ACKNOWLEDGMENTS

The implementations reported here were performed on the SP2 at the Maui High Performance Computing Center (MHPCC). This research is sponsored in part by the Phillips Laboratory, Air Force Material Command, USAF, under cooperative agreement number F29601-93-2-0001. Implementations on T3D were performed at the Pittsburgh Supercomputing Center (PSC).

REFERENCES

1. D. Adams, "CRAY T3D System Architecture Overview Manual," Technical Report, CRAY T3D Research Inc., 1993.
2. D. Bader, D. Helman, and J. JáJá, "Practical Parallel Algorithms for Personalized Communication and Integer Sorting," Technical Report, University of Maryland, -CS-TR-3548, UMIACS-TR-95-101.
3. L. S. Blackford *et al.*, ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance, *in* "Proceedings of the Supercomputer 96 Conference," IEEE Computer Society Press, Los Alamos, CA, U.S.A., November 1996.
4. N. J. Boden *et al.*, MYRINET: A gigabit per second local area network, *IEEE Micro* **15**(1) (February 1995) 29–36.
5. S. Bokhari, Multiphase complete exchange on Paragon, SP2, & CS-2, *IEEE Parallel Distrib. Technol. Systems Oper.* **4**(3) (1996).
6. Y. Chang, N. Golmie, and D. Su, Study of interoperability between EFCI and ER switch mechanism for ABR traffic in an ATM network, *in* "Proceedings of the Fourth International Conference on Computer Communications and Networks (ICCCN95)," September 1995.
7. S. Chaudhuri, S. Ghandeharizadeh, and C. Shahabi, "Avoiding Retrieval Contention for Composite Multimedia Objects," Technical Reports 95-618, *USC Computer Science Department.*
8. A. Choudhary, G. Fox, S. Hiranadani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz, Classification of irregular loosely synchronous problems and their support in scalable parallel software systems, *in* "1992 DARPA Software Technology Conference Proceedings," pp. 138–149, Syracuse Technical Report SCCS-255.
9. J. Demmel and S. Smith, Parallelizing a global atmospheric chemical tracer model, *in* "Symposium of High Performance Computing and Communications," May 1994.
10. T. Eicken, A. Basu, V. Buch, and W. Vogels, U-Net: A user-level network interface for parallel and distributed computing, *in* "Proceedings of the 15th ACM Symposium on Operating Systems Principles," Copper Mountain, Colorado, December 1995.

11. R. Felderman *et al.*, ATOMIC: A high speed local communication architecture, *J. High Speed Networks* **3**(1) (1994), 1–30.
12. A. Gerasoulis, J. Jiao, and T. Yang, Experience with scheduling irregular scientific computation, *in* "Proceedings of the First IPPS Workshop on Solving Irregular Problems on Distributed Memory Machines," Santa Barbara, CA, April 1995.
13. R. Graybill, Embedded high performance scalable computing tech. dev., Presented at the "ARPA Embedded Systems P1 Meeting," January 1996.
14. W. Gropp, Ewing Lusk, and Anthony Skjellum, "Using MPI," MIT Press, Cambridge, MA, ISBN 0-262-57104-8.
15. M. Gupta and P. Banerjee, Compile-time estimation of communication costs in programs, *J. Program. Lang.* **2** (1994), 191–225.
16. S. Hambrusch, F. Hameed, and A. Khokhar, Communication operations on coarse-grained mesh architectures, *Parallel Comput.* **21** (1995), 731–751.
17. S. Hambrusch, A. Khokhar, and Y. Liu, Scalable S-to-P broadcasting on message-passing MPPs, *in* "25-th International Conference on Parallel Processing," 1996.
18. V. Karamcheti and A. A. Chien, Comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D, *in* "Proceedings of the ISCA'95," June 1995.
19. R. Kesaven and D. K. Pands, Minimising node contention in multiple multicast on wormhole k-ary n-cube networks, *in* "Proceedings of the International Conference on Parallel Processing," August 1996.
20. V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to Parallel Computing, Design and Analysis of Algorithm," Chap. 3. Benjamin/Cummings Publishing Company, Inc., New York, 1994.
21. J. Lee, S. Ranka, and R. Shankar, "Communication-Efficient and Memory-Bounded External Redistribution," Technical Report, Computer Science Department, Syracuse University, 1995.
22. Y. W. Lim, P. B. Bhat, and V. K. Prasanna, Efficient algorithms for block-cyclic redistribution of arrays, *IEEE Symp. Parallel Distrib. Process.* (1996).
23. W. Liu, C. Wang, and V. K. Prasanna, Portable and scalable algorithms for irregular all-to-all communication, *in* "Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)," Hong Kong, May 1996.
24. D. Nicol and S. Bokhari, Optimal multiphase complete exchange on circuit-switched hypercube architectures, *in* "Proceedings of the 1994 ACM SIGNETRICS Conference," Nashville, TN
25. R. Ponnusamy, Y. Hwang, R. Das, and J. Saltz, Supporting-irregular distributions using data-parallel languages, *IEEE Parallel Distrib. Technol.* **3** (1) (1995), 12–24.
26. S. Ranka, R. Shankar, and K. Alsabti, Many-to-many communication with bounded traffic, *in* "1995 Symposium on Frontiers of Massively parallel Computations."
27. S. Ranka, J. Wang, and G. C. Fox, Static and runtime algorithms for all-to-many personalized communications on permutation networks, *in* "Proceedings of the 1992 International Conference on Parallel and Distributed Systems," pp. 211–218, Hsin Chu, Taiwan, December 1992.
28. S. Ranka, J. C. Wang, and M. Kumar, Personalized communication avoiding node contention on distributed memory systems, *in* "International Conference on Parallel Processing," August 1993.
29. C. Stunkel, D. G. Shea, *et al.*, "The SP2 Communication Subsystems," IBM Highly Parallel Supercomputing Systems Laboratory, August, 1994.
30. C.-L. Wang, V. K. Prasanna, H. Kim, and A. Khokhar, Scalable data parallel implementations of object recognition using geometric hashing, *J. Parallel Distrib. Comput.* (March 1994), 96–109.
31. C.-L. Wang, V. K. Prasanna, and Y. Lim, Parallelization of perceptual grouping on message-passing machines, *in* "1995 Workshop on Computer Architecture for Machine Perception," September 1995.



WENHENG LIU is an assistant professor of electrical and computer engineering in the California State University, Los Angeles (CSULA). His research interests include parallel architecture, embedded high performance computing, parallel algorithms, and wireless telecommunication architectures. He received his B.S. in electrical engineering from the National Taiwan University, M.S. and Ph.D. in computer engineering from the University of Southern California. He leads a telecommunication research group in CSULA to develop a parallel environment for a suite of telecommunication applications. Also, he leads the parallel computing multimedia researches in the Structures, Pointing and Control Engineering (SPACE) lab at CSULA. He is the Co-PI of the NSF sponsored CSULA/USC Collaborative to Integrate Research & Education.

CHO-LI WANG received his B.S. in computer science and information engineering from National Taiwan University in 1985. He obtained his M.S. and Ph.D. degrees in computer engineering from the University of Southern California in 1990 and 1995, respectively. He is currently an associative professor with the Department of Computer Science and Information Systems at The University of Hong Kong. His research interests are in high-speed cluster networking, distributed Java virtual machine, operating systems, and Web and Internet computing. He is a member of the executive committee for IEEE Task Force on Cluster Computing (TFCC) and also a core member of Asia-Pacific Grid (ApGrid).

VIKTOR K. PRASANNA (V. K. Prasanna Kumar) received his B.S. in electronics engineering from the Bangalore University and his M.S. from the School of Automation, Indian Institute of Science. He obtained his Ph.D. in computer science from the Pennsylvania State University in 1983. Currently, he is a professor in the Department of Electrical Engineering as well as in the Department of Computer Science at the University of Southern California, Los Angeles. He is also an associate member of the Center for Applied Mathematical Sciences (CAMS) at USC. He served as the Division Director for the Computer Engineering Division from 1994–1998. His research interests include parallel and distributed systems, embedded systems, configurable architectures, and high performance computing. He has published extensively and consulted for industries in the above areas. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He is the Steering Co-chair of the International Parallel and Distributed Processing Symposium (merged IEEE International Parallel Processing Symposium (IPPS) and the Symposium on Parallel and Distributed Processing (SPDP)) and is the Steering Chair of the International Conference on High Performance Computing (HiPC). He serves on the editorial boards of the *Journal of Parallel and Distributed Computing* and the *Proceedings of the IEEE*. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a Fellow of the IEEE.