Bandwidth Efficient All-reduce Operation on Tree Topologies[1]

Pitch Patarasuk and Xin Yuan
Department of Computer Science, Florida State University,
Tallahassee, FL 32306
{patarasu, xyuan}@cs.fsu.edu

Abstract

We consider efficient implementations of the all-reduce operation with large data sizes on tree topologies. We prove a tight lower bound of the amount of data that must be transmitted to carry out the all-reduce operation and use it to derive the lower bound for the communication time of this operation. We develop a topology specific algorithm that is bandwidth efficient in that (1) the amount of data sent/received by each process is minimum for this operation; and (2) the communications do not incur network contention on the tree topology. With the proposed algorithm, the all-reduce operation can be realized on the tree topology as efficiently as on any other topology when the data size is sufficiently large. The proposed algorithm can be applied to several contemporary cluster environments, including high-end clusters of workstations with SMP and/or multi-core nodes and low-end Ethernet switched clusters. We evaluate the algorithm on various clusters of workstations, including a Myrinet cluster with dual-processor SMP nodes, an InfiniBand cluster with two dual-core processors SMP nodes, and an Ethernet switched cluster with single processor nodes. The results show that the routines implemented based on the proposed algorithm significantly outperform the native MPI_Allreduce *and other recently developed algorithms for high-end SMP clusters when the data size is sufficiently large.*

1

1 Introduction

The all-reduce operation combines values from all processes and distributes the results to all processes. It is one of the most commonly used collective communication operations. In the Message Passing Interface (MPI) standard [9], the routine that realizes the all-reduce operation is *MPI_Allreduce*.

We consider efficient implementations of *MPI_Allreduce* with large data sizes on tree topologies, focusing on operations with commutative operators. We prove a tight lower bound of the amount of data that must be transmitted to carry out the all-reduce operation and use it to establish the lower bound of the communication completion time. We develop a topology specific all-reduce algorithm that is bandwidth efficient in that (1) the amount of data sent/received by each process is minimum; and (2) the communications do not incur network contention on the tree topology. Hence, our bandwidth efficient algorithm can theoretically achieve the lower bound on the communication time for the all-reduce operation when the bandwidth term dominates the communication time. Using the proposed algorithms, the all-reduce operation can be realized on the tree topology as efficiently as on any other topology when the data size is sufficiently large. The algorithm can be directly applied to several contemporary cluster environments, including high-end clusters with SMP and/or multi-core nodes and low-end Ethernet switched clusters.

We evaluate the proposed algorithm on various clusters of workstations, including high-end SMP clusters with Myrinet and InfiniBand interconnects and low-end Ethernet switched clusters. The results show that the routines based on the proposed algorithm significantly out-perform the native *MPI_Allreduce* and other recently developed algorithms for SMP clusters [17] when the data size is sufficiently large, which demonstrates the effectiveness of the proposed algorithm.

The rest of the paper is organized as follows. The system model is described in Section 2. Section 3 proves the theoretical lower bound of the communication time for the all-reduce operation and presents the topology specific algorithm that can achieve this lower bound on tree topologies. Section 4 reports the results of our experiments. The related work is discussed in Section 5. Section 6 concludes the paper.

[1]This research was funded in part by National Science Foundation grants, CCF-0342540 and CCF-0541096.

2 The system model

The tree topology is a connected graph $G = (V, E)$ with no circle, where V is the set of nodes and E is the set of edges. There is a unique path between any two nodes. Since we use the tree topologies to model cluster systems, the leaf nodes in the topology are processing elements (processors, cores, or machines) and the intermediate nodes are switching elements. We will call leaf nodes *machines* and intermediate nodes *switches*. Let S be the set of switches and M be the set of machines. $V = S \cup M$. Since links in the cluster systems that we considered are bidirectional, we will use two directed edges to model a link in the tree topology. Figure 1 shows an example cluster. Notion $u \rightarrow v$ denotes a communication from node u to node v. $Path(u \rightarrow v)$ denotes the set of directed edges in the unique path from node u to node v. For example, in Figure 1, $path(n0 \rightarrow n3) = \{(n0, s0), (s0, s1), (s1, s3), (s3, n3)\}$. Two communications, $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$, are said to have contention if they share a common edge, that is, there exists an edge (x, y) such that $(x, y) \in path(u_1 \rightarrow v_1)$ and $(x, y) \in path(u_2 \rightarrow v_2)$. A *pattern* is a set of communications. A *contention-free pattern* is a pattern where no two communications in the pattern have contention. We will use the notion $u \rightarrow v \rightarrow w \rightarrow ... \rightarrow x \rightarrow y \rightarrow z$ to represent pattern $\{u \rightarrow v, v \rightarrow w, ..., x \rightarrow y, y \rightarrow z\}$.

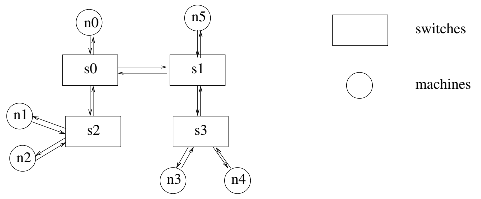


Figure 1. An example tree topology

The tree topology can be used to model high-end SMP clusters with high speed interconnects. The interconnect in such a system is usually a fat-tree, whose performance is very close to a cross-bar switch. The communication between processors in different SMP nodes (inter-node communication) goes through the interconnect while the intra-node communication is performed within an SMP node, typically through memory operations. Assuming that each SMP node is equipped with one network interface card, such a system can be approximated by a two-level tree topology as shown in Figure 2: with the interconnect being modeled as the root and the processors (cores) modeled as leaves. This paper only considers the case when one network interface card is used. Note that in a large cluster, how the SMP nodes are allocated for an MPI job is system dependent. Hence, the exact two-level tree topology for a job is undetermined until the SMP nodes are allocated. However, in most SMP clusters, the default process assignment for running an MPI program is to assign MPI processes with consecutive ranks to processors (or cores) within each SMP node. As we will show later, using the two-level tree abstraction, an efficient all-reduce algorithm can be developed for this default process assignment scheme.

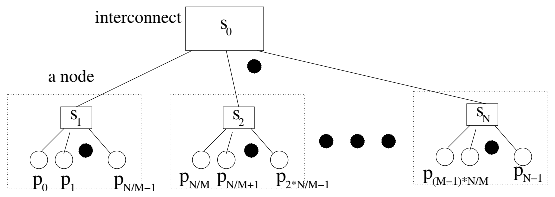


Figure 2. The two-level tree model for SMP clusters

Another type of cluster that can be modeled by the tree topology is an Ethernet switched cluster. We assume that each workstation is equipped with one Ethernet card. In such a cluster, links operate in the duplex mode that supports simultaneous communications on both directions of each link with the full bandwidth. Communications in such a system follow the 1-port model [2], that is, at one time, a machine can send and receive one message. The switches may be connected in an arbitrary way. However, a spanning tree algorithm is used by the switches to determine forwarding paths that follow a tree structure [14]. As a result, the physical topology of the network can be modeled as a tree $G = (V, E)$ with switches being the internal nodes and machines being leaves.

3 The all-reduce operation

We consider all-reduce operations with commutative operators. In this operation, a global reduction is performed on data of all processes and the reduction results are put in all processes. In terms of operating results, an all-reduce operation is equivalent to a reduction operation (that reduces the results to one process) followed by a broadcast operation that distributes the results to all processes. More specifically, let the N processes be denoted as p_0, p_1, ..., p_{N-1}. Before the all-reduce operation, each process p_i, $0 \leq i \leq N-1$, has X data items a_0^i, a_1^i, ..., a_{X-1}^i. Let us denote $\oplus$ the commutative reduction operator. At the end of the operations, all processes have all X-item results r_0, r_1, ..., r_{X-1}, where $r_j = a_j^0 \oplus a_j^1 \oplus ... \oplus a_j^{N-1}$, $0 \leq j \leq X-1$. The j-th item in the result data (r_j) is equal to the reduction of the j-th items in the source data in all processes.

3.1 The lower bounds

Let us first consider the required communications to complete the one-item all-reduce operation on N processes. In this case, let the N processes be p_0, ..., p_{N-1} and the initial value of the element in process p_i, $0 \leq i \leq N-1$, be a_i. At the end of the all-reduce operation, all processes will contain the result $r = a_0 \oplus a_1 \oplus ..... \oplus a_{N-1}$. Let partial results $b_{i,j} = a_i \oplus a_{i+1} \oplus ... \oplus a_j$, $0 \leq i \leq j \leq N-1$. Since $\oplus$ is commutative, the reduction result r can be obtained by applying $\oplus$ on multiple partial results. Obviously, $a_i = b_{i,i}$: the initial data items a_i are also partial results that can be used to compute the final reduction result. By the definition of the reduction operation, the size of the partial results $b_{i,j}$ is the same as the size of a data item. We will use the term *elements* to refer to both the partial results $b_{i,j}$ and the initial data items a_i and assume that *the smallest unit for computation and communication in the all-reduce operation is an element*. This assumption holds for almost all practical cases.

Lemma 1: To complete a *one-element* all-reduce operation on N processes, the minimum number of elements to be communicated is $2 \times (N-1)$.

Proof: For easy exposition, let us assume that each message transfers one element between two processes (a message with multiple elements can be treated as multiple messages). We will prove by induction on N that in order for all processes to have the result $r = a_0 \oplus a_1 \oplus ..... \oplus a_{N-1}$, at least $2 \times (N-1)$ messages are needed.

Base case: $N = 2$. With one message, at most one process can send its data to the other process. Hence, at most one process can compute $a_0 \oplus a_1$, which cannot complete the all-reduce operation. Hence, at least $1 + 1 = 2 = 2 \times (N-1)$ communications are needed to complete the operation.

Induction case: The induction hypothesis is that the minimum number of elements to be communicated in order to complete a one-element all-reduce operation with N processes is $2 \times (N-1)$. Using this hypothesis, we will prove that the minimum number of messages in order to complete a one-element all-reduce operation on $N+1$ processes is $2 \times ((N+1)-1) = 2N$.

Let p_l be the process that receives the last message in the operation on $N+1$ processes. Hence, before the last message, all other processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N have sufficient information to compute the reduction result $r = a_0 \oplus a_1 \oplus ..... \oplus a_N$. We will show that at least $2N-1$ messages have been communicated before the last message.

Let Z be the minimum number of messages for processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N to have the result $r = a_0 \oplus a_1 \oplus ..... \oplus a_N$. Let messages m_0, m_1, ..., m_{Z-1} be the Z messages. We will show that to compute the all-reduce results ($r' = a_0 \oplus ... \oplus a_{l-1} \oplus a_{l+1} \oplus ... \oplus a_N$) on processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N ($N-1$ processes), only $Z-1$ messages are needed. Notice that process p_l does not involve in the all-reduce operation on processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N. Let messages m_{i_1}, m_{i_2}, ..., m_{i_x} be messages in $\{m_0, m_1, ..., m_{Z-1}\}$ that are sent to p_l and messages m_{j_1}, m_{j_2}, ..., m_{j_y} are messages that are sent from p_l. Since the result $r = a_0 \oplus a_1 \oplus ..... \oplus a_N$ includes the term a_l, there exists at least one message sent from p_l (so that other processes can compute the result with the term a_l). Hence, $y \geq 1$. Consider the messages sent to p_l, there are two cases: (1) there exist such messages; and (2) there does not exist such a message.

In the case when there does not exist any m_i sending to p_l, the element sent in m_{j_1} must be a_l (p_l does not have any other information to send). Actually, all messages in $\{m_0, m_1, ..., m_{Z-1}\}$ that are sent from p_l must be a_l since p_l does not have the information about the items in other processes. We can construct the sequence of at most $Z-1$ messages to compute the all-reduce result ($r' = a_0 \oplus ... \oplus a_{l-1} \oplus a_{l+1} \oplus ... \oplus a_N$) among processes p_0,..., p_{l-1}, p_{l+1}, ..., p_N as follows: for each m_i, $0 \leq i \leq Z-1$, (1) remove m_i if the element in m_i is a_l; (2) change the message to be $C \oplus D$ if the message in m_i is $C \oplus a_l \oplus D$, where C and D are either elements or empty. Since at least one message, m_{j_1}, is to be removed, the new sequence has at most $Z-1$ messages. Since the original message sequence m_0, ..., m_{Z-1} allows $r = a_0 \oplus a_1 \oplus ..... \oplus a_N$ to be computed in processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N, the new sequence will allow $r' = a_0 \oplus ... \oplus a_{l-1} \oplus a_{l+1} \oplus ... \oplus a_N$ to be computed in processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N. Note that p_l is not involved in the new message sequence.

In the case when there exist some messages m_{i_1}, m_{i_2}, ..., m_{i_x} that are sent to p_l, let us denote $src(m_{i_k})$ be the source process of m_{i_k}. We can construct the sequence of at most $Z-1$ messages to compute the all-reduce result ($r' = a_0 \oplus ... \oplus a_{l-1} \oplus a_{l+1} \oplus ... \oplus a_N$) among processes p_0,..., p_{l-1}, p_{l+1}, ..., p_N as follows: (1) for each m_i, $0 \leq i \leq Z-1$, change the message to be $C \oplus D$ if the element in m_i is $C \oplus a_l \oplus D$ (remove the message if the element is a_l); (2) change the receiver of message m_{i_k}, $k < x$, from a_l to $src(m_{i_x})$; (3) remove message m_{i_x}; and (4) change the source node of message m_{j_k}, $1 \leq k \leq y$, from p_l to $src(m_{i_x})$. Basically, all the messages relayed by p_l in the original sequence is now relayed by $src(m_{i_x})$ and a_l is erased from all messages. The new sequence should allow the all-reduce operation to be completed on processes p_0, ..., p_{l-1}, p_{l+1}, ..., p_N. This sequence has at most $Z-1$ messages since m_{i_x} is removed.

Hence, the all-reduce operation among processes p_0,..., p_{l-1}, p_{l+1}, ..., p_N can be done with $Z-1$ messages. From the induction hypothesis, $Z-1 \geq 2(N-1)$. Hence, the number of communications before the last communication is at least $2 \times (N-1)+1$ and the total number of messages to complete the all-reduce operation on $N+1$ processes is at least $2 \times (N-1)+1+1 = 2N$. □

Lemma 2: Assume that (1) data are not compressed during the all-reduce operation; and (2) source data items are independent of one another. To perform an all-reduce operation of X items of $itsize$ bytes on N processes, there exists at least one process that must communicate at least $2 \times \frac{N-1}{N} \times X \times itsize$ bytes data.

Proof: From Lemma 1, the minimum amount of data to be communicated in order to complete the one-element all-reduce operation is $2 \times (N-1) \times itsize$ bytes. Since every source data item in the all-reduce operation is independent, the minimum amount of total data to be communicated is $2 \times (N-1) \times itsize \times X$. Since there are N processes that carry out the communications collectively, the communications can be distributed among the N processes. Hence, at least one process needs to communicate $\frac{2\times(N-1)\times X\times itsize}{N} = 2 \times \frac{N-1}{N} \times X \times itsize$ byte data. □

Let the *communication completion time* for the all-reduce operation be defined as the duration between the time when the operation starts and the time when the last process finishes the operation. Let us model the time to send an n-byte message between any two processes as $T(s) = \alpha + s \times \beta$, where α is the startup overhead and β is the per byte transmission time. When s is sufficiently large, the bandwidth term $s \times \beta$ dominates the total communication time ($s \times \beta >> \alpha$) and $T(s) \approx s \times \beta$. In this case, $T(m \times s) \approx m \times T(s)$.

Lemma 3: Under the assumptions stated in Lemma 2, the communication completion time for the all-reduce operation is at least $T(2 \times \frac{N-1}{N} \times X \times itsize)$.

Proof: Straight-forward from the previous discussion and definition. □

Lemma 3 indicates that the minimum communication time for the all-reduce operation of $msize = X \times itsize$ data on N processes is $T(2 \times \frac{N-1}{N} \times msize)$. When $msize$ is sufficiently large such that $msize \times \beta >> \alpha$, $T(2 \times \frac{N-1}{N} \times msize) \approx 2 \times \frac{N-1}{N} \times T(msize) \approx 2T(msize)$. This indicates that the best of any all-reduce algorithms can do is to achieve roughly 2 times the time to send an $msize$ data. It must be noted that this analysis focuses on communication time with the assumption that the communication time is dominated by the bandwidth term. In practice, the communication start-up overhead and the computation in the operation may be significant.

3.2 Bandwidth efficient all-reduce algorithm

Following the scheme proposed in [12], our bandwidth efficient algorithm realizes the all-reduce operation by a reduce-scatter operation followed by an all-gather operation. The challenges lie in how to realize these two operations efficiently on the tree topology. The reduce-scatter operation computes the reduction results for all items; and the results are distributed among the N processes, each process having $\frac{msize}{N}$ byte data. After the reduce-scatter operation, an all-gather operation is performed to gather all of the reduction results to all processes.

To achieve the best performance, both the reduce-scatter and the all-gather operations must be performed efficiently. In particular, network contention can significantly degrade the communication performance on the tree topology when a collective operation is performed. Hence, one must carefully select the communication patterns for realizing the operations without introducing contention.

Our algorithm performs the reduce-scatter using a logical ring communication pattern. Let the N processes be p_0, p_1, ..., p_{N-1}. Let $F : \{0, ..., N-1\} \rightarrow \{0, ..., N-1\}$ be a one-to-one mapping function. Thus, $p_{F(0)}$, $p_{F(1)}$, ..., $p_{F(N-1)}$ is a permutation of p_0, p_1, ..., p_{N-1}. The logical ring pattern contains the following communications:

$$p_{F(0)} \rightarrow p_{F(1)} \rightarrow p_{F(2)} \rightarrow ... \rightarrow p_{F(N-1)} \rightarrow p_{F(0)}$$

Using the logical ring pattern, the reduce-scatter operation is performed as follows. First, the $msize$ source data in each process is partitioned into N segments, each segment having $\frac{msize}{N}$ bytes of data. For easy exposition, we will assume that $msize$ is divisible by N. Let us number the segments by SEG_0, SEG_1, ..., SEG_{N-1} The reduce-scatter operation is carried out by performing the logical ring pattern $N-1$ iterations. In the first iteration (iteration 1), process $p_{F(i)}$ sends segment $SEG_{(i-1)\ mod\ N}$ to $p_{F((i+1)\ mod\ N)}$. After each process receives the data, it performs a reduction operation on the received data segment with its corresponding data segment (the segment with the same segment index), and replaces its own data with the (partial) reduction results. For each remaining iteration $j : 2 \leq j \leq N-1$, each process $p_{F(i)}$ sends the newly computed $SEG_{(i-j)\ mod\ N}$ to $p_{F((i+1)\ mod\ N)}$. After receiving the data communicated in each iteration, each process performs the reduction operation on the data received with the corresponding segment in the local array and replaces the partial reduction results in the array. At the end of the $N-1$ iterations, $p_{F(i)}$ holds the reduction results in SEG_i, $0 \leq i \leq N-1$. Figure 3 shows the logical ring implementation of reduce-scatter on four processes.

The remaining question is then how to embed a logical ring on a tree topology without introducing contention. This problem has been solved in [4]. For completeness, we will briefly describe the algorithm in [4] that finds a contention-free logical ring on a tree topology.

Let $G = (S \cup M, E)$ be a tree graph with S being switches, M being machines (each MPI process is mapped to one machine), and E being the edges. Let $G' = (S, E')$ be a subgraph of G that only contains switches and links between switches. A contention-free logical ring can be com-

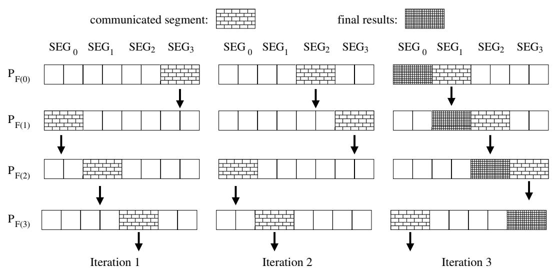


Figure 3. Logical ring reduce-scatter algorithm

puted in the following two steps.

• Step 1: Starting from any switch, perform Depth First Search (DFS) on G'. Number the switches based on the DFS arrival order. We will denote the switches as s_0, s_1, ..., $s_{|S|-1}$, where s_i is the ith switch arrived in the DFS traversal of G'.

• Step 2: Let the X_i machines connecting to switch s_i, $0 \leq i \leq |S|-1$, be numbered as $n_{i,0}$, $n_{i,1}$, ..., n_{i,X_i-1}. $X_i = 0$ when there is no machine attaching to s_i. The following logical ring is contention-free:

$n_{0,0} \rightarrow ... \rightarrow n_{0,X_0-1} \rightarrow n_{1,0} \rightarrow ... \rightarrow n_{1,X_1-1} \rightarrow ... \rightarrow n_{|S|-1,0} \rightarrow ... \rightarrow n_{|S|-1,X_{|S|-1}-1} \rightarrow n_{0,0}$.

Lemma 4: The logical ring found by the above algorithm is contention-free.

The formal proof of Lemma 4 can be found in [4]. The algorithm described can be directly applied when the tree topology is determined. As we mentioned earlier, in a high-end SMP cluster, even though the system can be approximated with a two-level tree, the exact tree topology is unknown until the SMP nodes are allocated. Hence, this algorithm cannot be directly used in a high-end SMP cluster. However, by default, most SMP clusters assign MPI processes with consecutive ranks to processors (or cores) in each SMP node. For such clusters, using the 2-level tree representation, a contention-free logical ring for the default process assignment scheme can be built without knowing the exact topology as shown in the following lemma.

Lemma 5: Under the assumption that MPI processes with consecutive ranks are assigned to processors (or cores) in each SMP node, logical ring pattern, $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_{N-1} \rightarrow p_0$, is contention free.

Proof: Assume that there are M SMP nodes in the system, each having $\frac{N}{M}$ processors (cores). Under the process assignment assumption, p_0, p_1, ..., $p_{\frac{N}{M}-1}$ are assigned to one SMP node; $p_{\frac{N}{M}}$, $p_{\frac{N}{M}+1}$, ..., $p_{2\times\frac{N}{M}-1}$ are assigned to another SMP node; and so forth. An SMP node SMP_i, $1 \leq i \leq M$, contains processes $p_{(i-1)\times\frac{N}{M}}$ to $p_{i\times\frac{N}{M}-1}$. The process assignment is depicted in the two-level tree in Figure 2. Let switch s_0 corresponds to the root (interconnect) and switch s_i corresponds to node SMP_i. Clearly, the sequence s_0,..., s_N is a DFS traversal sequence. Thus, from Lemma 4, the logical ring $p_0 \rightarrow p_1 \rightarrow ... \rightarrow p_{N-1} \rightarrow p_0$ is contention-free. □

Now, let us consider the all-gather operation. The all-gather operation gathers $\frac{msize}{N}$ data from each process to all processes. We use the algorithm developed in [4] for performing this operation. In this algorithm, the all-gather operation is also carried out with a logical ring communication pattern, which is exactly the same as the communication pattern used to realize the reduce-scatter operation.

$$p_{F(0)} \rightarrow p_{F(1)} \rightarrow ... \rightarrow p_{F(N-1)} \rightarrow p_{F(0)}.$$

In the first iteration, each process $p_{F(i)}$, $0 \leq i \leq N-1$, sends its own data to $p_{F((i+1)\ mod\ N)}$ and receives data from $p_{F((i-1)\ mod\ N)}$. In subsequent iterations, each process $p_{F(i)}$ forwards what it received in the previous iteration to $p_{F((i+1)\ mod\ N)}$ (and receives from $p_{F((i-1)\ mod\ N)}$). After $N-1$ iterations, all data from all processes reach all nodes in the system and the all-gather operation is complete. Note that in each iteration, each process must copy the data it receives into the right place of its output buffer.

Both the reduce-scatter operation and the all-gather operation are performed in $N-1$ steps with each process sending and receiving $\frac{msize}{N}$ data in each step. Hence, the total communication time is $2 \times (N-1) \times T(\frac{msize}{N})$. When $\frac{msize}{N}$ is sufficiently large, $2 \times (N-1) \times T(\frac{msize}{N}) \approx T(2 \times \frac{N-1}{N} \times msize)$, which is optimal (Lemma 3). Note that "sufficiently large" is a relative term: if $\frac{msize}{N}$ is close to infinity, the performance of the proposed algorithm will be close to optimal. However, the exact threshold when the proposed algorithm is better than other algorithms is system dependent. As will be shown in the next section, the threshold value is around $\frac{msize}{N} = 256B$ for the proposed algorithm to be more efficient than existing algorithms in some contemporary high-end SMP clusters, and $8KB$ in a 32-node cluster connected by Gigabit Ethernet switches.

4 Experiments

Based on the algorithm described in the previous section, we implement routines that perform the all-reduce operation in two forms. For the high-end cluster whose topology can be approximated by a two-level tree, we implement a stand-alone *MPI_Allreduce* routine based on Lemma 5. For clusters with a physical tree topology, we develop a routine generator that takes a tree topology as input and automatically produces an all-reduce routine that uses the topology specific algorithm for the input topology. The generated routines are written in C and are based on MPICH point-to-point primitives. These routines are available at http://www.cs.fsu.edu/∼patarasu/ALLREDUCE.

We compare the proposed algorithms with the native MPI implementations on various clusters. In addition, for high-end SMP clusters, we also compare the proposed scheme with two algorithms that were developed specifically for SMP clusters [17]. Both SMP specific algorithms [17] take advantages the two-level structure of an SMP cluster and separate intra-node communications from inter-node communications. One algorithm, denoted as *SMP-rdb*, uses the recursive doubling scheme for inter-node communication. The other one, denoted as *SMP-binomial*, uses a pipelined binomial tree.

Three clusters are used in the evaluation: (1) the NCSA Teragrid IA-64 Linux cluster [11], (2) the TACC Lonestar cluster [13], and (3) an Ethernet switched cluster. The NCSA Teragrid IA-64 Linux cluster is a Myrinet cluster with dual 1.5GHz Intel Itantium 2 SMP nodes and 4GB memory per node. The system runs Linux 2.4.21-SMP operating system and uses mpich-gm-1.2.5.10 library. The TACC Lonestar cluster is an InfiniBand cluster with Dell PowerEdge 1955 blade as compute nodes, each node having two dual-core Xeon 5100 processors (4 cores per node) and 8GB memory. The system runs Linux 2.6.12 x86_64 operating system and uses mvapich-0.9.8 library. The Ethernet switched cluster consists of 32 compute nodes connected by Dell Powerconnect 2724 Gigabit Ethernet switches. The nodes are Dell Dimension 2400, each with a 2.8GHz processor and 640MB memory. All nodes run Linux (Fedora) with the 2.6.5-358 kernel.

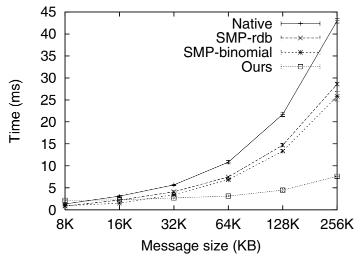
The code segment for measuring communication completion time is shown in Figure 4. The reduction operator is *MPI_BOR*. Twenty iterations are measured for all-reduce communications with data sizes less than 64 KB, and ten iterations for communications with larger data sizes. Within each iteration, a barrier is added to prevent pipelined communication between iterations. Since we only consider communications with reasonably large data sizes, the barrier overhead is insignificant. For each experiment, we run the benchmark 32 times, compute the confidence interval with 95% confidence level from the 32 samples, and report the results using the 95% confidence interval. Most confidence intervals from our experiment results are very small, which indicates that the results from the experiment are re-producible with small errors. We only report the results with the data size larger than 8KB since our algorithm is designed to achieve high performance for large data sizes.

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
for (count = 0; count < ITER_NUM; count ++) {
  MPI_Allreduce(..., MPI_BOR, ...);
  MPI_Barrier(...);
}
elapsed_time = MPI_Wtime() - start;
```

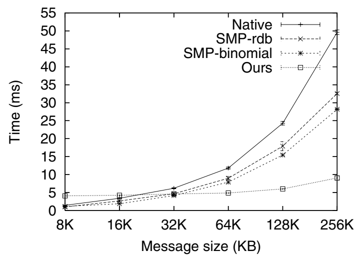
Figure 4. Code segment for measuring $MPI\_Allreduce$ performance.

NCSA Teragrid IA-64 Linux cluster results

Figure 5 show the performance of different all-reduce algorithms on the NCSA Teragrid cluster. The native MPI library on this system is mpich-gm-1.2.5.10. All programs are compiled with 'mpicc -lm' command with no other flag (mpicc invokes the Intel compiler in the system). Figure 5 (a) shows the results on 64 processors (32 nodes) and Figure 5 (b) shows the results on 128 processors (64 nodes). On the 64-processor system, our routine out-performs the native routine when the data size is 16KB; on the 128-processor system, our routine out-performs the native routine when the data size is 32KB. Notice that $\frac{16KB}{64} = \frac{32KB}{128} = 256B$: the threshold value of $\frac{msize}{N}$ for our bandwidth efficient algorithm to be more efficient is around $256B$ on this cluster. As the data size increases, the performance difference is very significant. This is due to the fact that our logical ring based algorithm is bandwidth efficient while the native MPI implementation uses a different algorithm that is not bandwidth efficient in the SMP cluster. Our routine also performs significantly better than the two algorithms designed for SMP clusters (SMP-rdb and SMP-binomial), both of which are better than the native all-reduce implementation. This is because our algorithm integrates (overlaps) the inter-node and intra-node communications seamlessly while SMP-rdb and SMP-binomial artificially separate inter-node communications and intra-node communications and have logically three phases.



(a) 64 processors

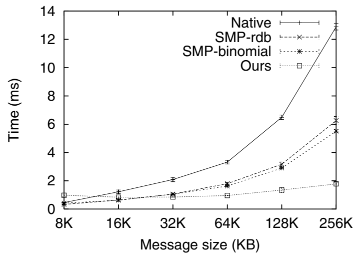


(b) 128 processors

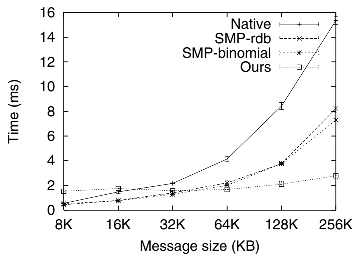
Figure 5. Results for the NCSA cluster

TACC Lonestar cluster results

Figure 6 shows the results on the TACC Lonestar cluster whose native MPI library is *mvapich 0.9.8*. All programs are compiled with 'mpicc -lm' command with no other flag (mpicc invokes the Intel compiler in the system). Figure 6 (a) shows the results on 64 cores (16 nodes) and Figure 6 (b) shows the results on 128 cores (32 nodes). The trend in the relative performance between our routine and the native *MPI_Allreduce* routine is very similar to that in the NCSA cluster except that the Lonestar cluster is a newer cluster and the communication is much faster. Interestingly, the threshold value of $\frac{msize}{N}$ is also around 256B in this cluster. The MPI libraries on both the Lonestar cluster and the NCSA cluster are based on MPICH, whose all-reduce algorithm is not bandwidth efficient on the SMP clusters. On both clusters, our bandwidth efficient implementation provides much better performance when the data size is large.



(a) 64 cores



(b) 128 cores

Figure 6. Results for the TACC cluster

Ethernet switched cluster results

For the Ethernet switched cluster, we show the results on the 32-node cluster with the two topologies shown in Figure 7. We performed experiments on other topologies and the trend is similar to those shown in the results for these two topologies. We will refer to the topologies in Figure 7 as topology (1) and topology (2). We use our routine generator to produce topology specific routines for these two topologies and compare the automatically generated routines with the routines in LAM/MPI 7.1.2 [6] and MPICH 2.1.0 [10].

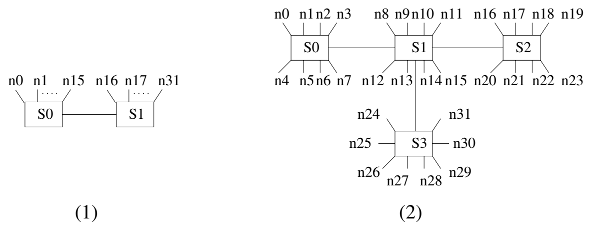
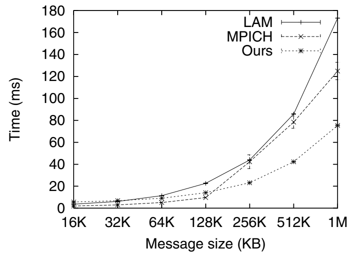
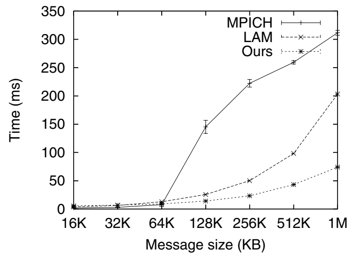

n0 n1 n15 n16 n17 n31
S0 S1
(1)
n0 n1 n2 n3 n8 n9 n10 n11 n16 n17 n18 n19
S0 S1 S2
n4 n5 n6 n7 n12 n13 n14 n15 n20 n21 n22 n23
n24 n31
n25 S3 n30
n26 n27 n28 n29
(2)

Figure 7. Ethernet Topologies used in the experiments



(a) Topology (1)



(b) Topology (2)

Figure 8. Results for the Ethernet switched cluster

Figures 8 (a) and (b) show the results for topology (1) and topology (2) respectively. For the Ethernet switched clusters, the proposed algorithm is more efficient when the data size is significantly larger than those in the high-end clusters (the NCSA and TACC clusters): on Topology (1), the proposed algorithm is the best among the three routines when the data size is more than 256KB; on Topology (2) the proposed algorithm is better when the data size is more than 128KB. Hence, the threshold value of $\frac{msize}{N}$ for the proposed algorithm to be more efficient is $\frac{256KB}{32} = 8KB$, which is much larger than the $256B$ in the high-end NCSA and TACC clusters. This reflects the fact that on Ethernet switched cluster with Gigabit switches, the communication start-up overhead is much larger than that in high-end clusters. Thus, even for reasonably large data sizes (e.g. 64KB), it is still more important to reduce the communication start-up overheads. Nonetheless, when the data size is larger, network contention becomes a problem and our contention-free algorithm is much more efficient. Notice that in Topology (2), the MPICH algorithm introduces serious network contention in the system when the data size is larger than 128KB.

5 Related Work

The all-reduce operation is one of the collective operations supported in the MPI standard [9], and thus, all MPI libraries such as LAM/MPI [6] and MPICH [10] support this operation. Many efficient platform independent algorithms for this operation have been proposed [15, 16, 12]. In [12], Rabenseifner proposed to realize the all-reduce operation by a reduce-scatter operation followed by an all-gather operation and gave various algorithms for the reduce-scatter and all-gather operations. Our proposed scheme combines this idea with contention-free realization of reduce-scatter and all-gather operations. Various architecture specific all-reduce schemes have also been developed [1, 3, 5, 7, 17]. In particular, all-reduce algorithms were developed specifically for SMP clusters in [17]. Our algorithm is more efficient on SMP clusters than those in [17]. Our all-reduce algorithm uses a contention-free all-gather algorithm for the tree topology [4]. In this paper, we combine the ideas from various papers [4, 12, 16] to develop an efficient algorithm for the all-reduce operation with large data sizes on the tree topology. The new contributions include the following. First, we establish a tight lower bound for the amount of data to be transmitted in order to complete the all-reduce operations. Second, we develop a topology specific algorithm for the all-reduce operation on the tree topology, which happens to be similar to that in [4]. Third, we empirically demonstrate that the proposed algorithm is efficient on high-end clusters with SMP and/or multi-core nodes as well as low-end Ethernet switched clusters.

6 Conclusions

In this paper, we investigate an efficient all-reduce implementation for large data sizes on the tree topology. We derive a theoretical lower bound on the communication time of this operation and develop an all-reduce algorithm that can theoretically achieve this lower bound. We demonstrate the effectiveness of the proposed algorithm on various contemporary clusters, including high-end clusters with SMP and/or multi-core nodes connected by high-speed interconnects, and low-end Ethernet switched clusters.

References

[1] G. Almasi, et.al. "Optimization of MPI Collective Communication on BlueGene/L Systems." *International Conference on Supercomputing* (ICS), 2005.

[2] O. Beaumont, L. Marchal, and Y. Robert, "Broadcast Trees for Heterogeneous Platforms." *The 19th IEEE IPDPS*, 2005.

[3] L. Bongo, O. Anshus, J. Bjorndalen, and T. Larsen. "Extending Collective Operations With Application Semantics for Improving Multi-cluster Performance." *Proceedings of the ISPDC/HeteroPar*, 2004.

[4] A. Faraj, P. Patarasuk and X. Yuan. "Bandwidth Efficient All-to-all Broadcast on Switched Clusters." In *IEEE Cluster Computing* , Boston, MA, Sept., 2005.

[5] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. "Efficient collective operations using remote memory operations on VIA-based clusters." In *IPDPS*, April 2003.

[6] LAM/MPI. http://www.lam-mpi.org

[7] A. Mamidala, J. Liu, D. Panda. "Efficient Barrier and Allreduce on InfiniBand Clusters using Hardware Multicast and Adaptive Algorithms." *Cluster*. 2004.

[8] MATHWORLD. http://www.Mathworld.com

[9] The MPI Forum. *The MPI-2: Extensions to the Message Passing Interface*, July 1997. Available at http://www.mpi-forum.org/docs/mpi-20-html/ mpi2-report.html.

[10] MPICH - A Portable Implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.

[11] NCSA Teragrid IA-64 Linux Cluster. http://www.ncsa.uiuc.edu/UserInfo/Resources /Hardware/TGIA64LinuxCluster.

[12] R. Rabenseifner. "Optimization of Collective Reduction Operations" *International Conference on Computational Science*, 2004.

[13] The Lonestar cluster. http://www.tacc.utexas.edu/services/userguides/lonestar.

[14] Andrew Tanenbaum, "Computer Networks", 4th Edition, 2004.

[15] R. Thakur and W. Gropp. "Improving the Performance of Collective Operations in MPICH." In *EuroPVM/MPI*, Oct. 2003.

[16] R. Thakur, R Rabenseifner, and W. Gropp. "Optimization of Collective Communication Operations in MPICH." *International Journal of High Performance Computing Applications*, 2005.

[17] V. Tipparaju, J. Nieplocha, and D. Panda. "Fast Collective Operation Using Shared and Remote Memory Access Protocols on Clusters." *IEEE IPDPS*, 2003.