Collective Operations for Wide-Area Message Passing Systems Using Adaptive Spanning Trees

Hideo Saito
University of Tokyo
h_saito@logos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
University of Tokyo/JST
tau@logos.ic.i.u-tokyo.ac.jp

Takashi Chikayama
University of Tokyo
chikayama@logos.ic.i.u-tokyo.ac.jp

***Abstract*— We propose a method for wide-area message passing systems to perform collective operations using dynamically created spanning trees. In our proposal, broadcasts and reductions are performed efficiently using topology-aware spanning trees constructed at run-time; processors autonomously measure latency and bandwidth to create latency-aware trees for short messages and bandwidth-aware trees for long messages. Our spanning trees adapt to topology changes due to the joining or leaving of processors; when processors join or leave a computation, processors repair the spanning trees so that effective execution of collective operations can continue. With 128 to 201 processors distributed over 3 to 4 clusters, the latency of our broadcast was within a factor of 2 of a static topology-aware implementation, and our broadcast achieved 82 percent of the bandwidth of a static topology-aware implementation. Moreover, when some processors joined or left a computation, our broadcast temporarily performed poorly for about 8 seconds while the spanning trees adapted to the new topology, but completed successfully even during this time.**

I. Introduction

Improvements in wide-area networks (WANs) have made it feasible to perform message passing using computational resources located in multiple local-area networks (LANs). In such an environment, however, systems designed for single LANs do not perform well. In particular, collective operations designed for local-area message passing systems perform horribly—they assume that latency and bandwidth between processors are uniform, but the wide-area links have latency and bandwidth that are orders of magnitude worse than those of the local-area links. Thus, collective operations for WANs must consider the topology of networks to avoid sending messages through links with high latency or low bandwidth.

Methods to perform collective operations efficiently in WANs have been proposed in the past [1]–[6]. Such methods assume that the computational environment (i.e. the processors involved in the computation and the performance of the network) does not change, and rely on manually supplied topology information to achieve high performance. However, manual configuration is impractical when the topology is not a simple two or three-level hierarchy. Moreover, manual configuration requires extensive knowledge of the underlying network, but such knowledge may not be readily available; in large-scale computations performed in WANs, knowledge of the underlying network will become harder and harder to obtain as it becomes more and more common to use different resources upon each invocation of an application and to use resources automatically allocated by middleware. Most importantly, large-scale computations performed in WANs last a long time and require that the topology be dynamic—not only is it necessary to adapt to changes in network performance, but it is also necessary to be able to release computational resources for other users, to use resources that become available after application startup, and to continue execution even when a part of the resources fail.

In the context of Content Delivery Networks (CDNs), many methods to perform application-level multicasts using topology-aware overlay networks have been proposed [7]–[9]. However, such multicast mechanisms, allowing data loss or being single-source, do not meet the needs of message passing applications.

Thus, we propose a configuration-free method for message passing systems to perform collective operations efficiently in dynamically changing topologies. Ultimately, we plan on providing support for fluctuating network performance and for fault-tolerance, but in this paper, we only deal with the intentional joining and leaving of processors.

In our proposal, we create latency-aware spanning trees and bandwidth-aware spanning trees, and perform collective operations along the generated trees. Our collective operations satisfy the following properties:

- They perform well when the processors taking part in the computation are not changing.
- They complete successfully even while processors are joining or leaving.
- They adapt to topology changes due to the joining or leaving of processors.

We have implemented the two spanning tree algorithms along with broadcast and reduction operations that use the generated trees on the Phoenix Message Passing Library [10], and performed experiments using real processors distributed over several clusters.

The rest of this paper is organized as follows. We discuss related work in Section II and describe the Phoenix Message Passing Library in Section III. We then explain our proposal in Section IV and present our experimental results in Section V. Finally, we conclude in Section VI.

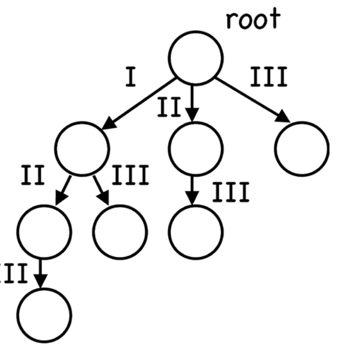


Fig. 1. The binomial tree that MPICH uses for broadcasts of short messages. An 8-processor broadcast is performed in 3 steps.

II. Related Work

A. The Collective Operations of MPICH

MPICH [11] is the most widely used implementation of the Message Passing Interface (MPI) [12]. Although there are many implementations that have extended MPICH for Grid execution, the original MPICH implementation is designed for LANs. Thus, its collective operations are optimized under the assumption that latency and bandwidth between processors are uniform. While MPICH uses different algorithms for different collective operations, message sizes, and numbers of processors, the basic idea in all of the algorithms is to forward messages along a tree [13]. Latency-aware trees are used for short messages, and bandwidth-aware trees are used for long messages. For example, short messages are broadcast along a binomial tree (Figure 1). Meanwhile, a binomial tree is bandwidth-inefficient, because the same processor must send the same data multiple times. Thus, long messages are broadcast by first scattering the data among all processors, and then collecting the data to all processors.

B. Efficient Collective Operations in WANs

Collective operations designed for LANs, such as those of MPICH, do not perform well in WANs, because the latency and bandwidth of the wide-area links are orders of magnitude worse than those of the local-area links. Thus, a number of other projects have studied the use of topology information to perform collective operations efficiently in WANs [1]–[6] .

Kielmann et al. have proposed MagPIe, in which collective operations minimize the use of wide-area links by using different trees for intra-cluster and inter-cluster communication [2]. One node called the *coordinator node* represents each cluster in inter-cluster communication. In a broadcast, for example, the root node sends a message to each of the coordinator nodes, which in turn performs an MPICH-like broadcast within its own cluster using a binomial tree.

While MagPIe focuses only on latency and thus short messages, Kielmann et al. have also proposed bandwidth-efficient collective communication [1]. In this work, measured values of latency and bandwidth are given as input to the system so that the proper tree shape and message segment size can be determined from the message size.

MPICH-G2 is an MPI implementation that extends MPICH for Grid execution [4]. It provides multi-level topology-aware collective operations that perform better than earlier topology-aware approaches, such as MagPIe, that distinguish only between intra-cluster and inter-cluster communication.

These works differ from our proposal in two ways: they assume that the computational environment does not change, and they require information concerning the network to be supplied manually.

C. Application-Level Multicast for CDNs

In the context of Content Delivery Networks (CDNs), many methods to perform application-level multicasts using topology-aware overlay networks have been proposed [7]–[9]. However, such multicast mechanisms, being designed for content delivery, do not meet the needs of message passing applications.

Banerjee et al. have proposed a low-overhead method to create and maintain a latency-aware overlay network that can be used to efficiently multicast short messages [9]. An extension to this work that uses the latency-aware trees for long messages has also been proposed, but its performance is not as good as those that focus specifically on bandwidth—it does not actually measure the bandwidth of links, and the fanout of the tree that it uses is too large. Moreover, it allows data loss because it is designed for data stream applications, but message passing applications cannot tolerate data loss.

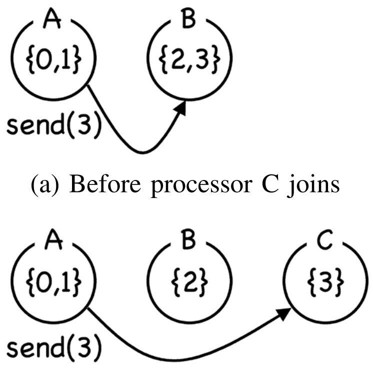
Overcast efficiently multicasts large data by creating and maintaining a bandwidth-aware overlay network [7]. It only supports single-source multicasts because it is designed for video streaming, but in message passing, every processor must be able to become the root of a collective operation.

Moreover, while multicast is a *1-to-N* operation, message passing systems need to support not just *1-to-N* operations, but also *N-to-1* operations and *N-to-N* operations.

III. Phoenix

We have implemented our collective operations in the context of the Phoenix Message Passing System [10], which defines semantics of message passing programs in the presence of dynamically joining/leaving processors. Although this paper can mostly be understood without knowing Phoenix specifics, this section describes the salient features of Phoenix in order to more precisely state our problem setting as well as our motivation. In particular, Phoenix defines what each type of collective operation (e.g. broadcast and reduction) should accomplish when the processors participating in a computation change dynamically.

In Phoenix, messages are not addressed to processors, but to one of the *virtual nodes* in a virtual node name-space. This virtual node name-space is divided up among the processors by an arbitrary mapping between processors and virtual nodes. Phoenix maintains a routing table for point-to-point communication, and messages are delivered to the processor to which



(a) Before processor C joins

(b) After processor C joins

Fig. 2. Virtual nodes in Phoenix. In (a), virtual node 3 is mapped to processor B, so a message sent to virtual node 3 arrives at processor B. In (b), processor C has joined the computation and acquired virtual node 3 from processor B—now, a message sent to virtual node 3 arrives at processor C.

the destination virtual node is mapped at the time of message delivery.

Applications can support the dynamic joining and leaving of processors during computation by changing the processor-virtual node mapping (Figure 2). To join a computation in progress, a processor connects to another processor already involved in the computation and receives a set of virtual nodes. To leave a computation, a processor first gives away its virtual nodes to another processor, then leaves.

A *broadcast* in MPI is an operation that delivers a message to every processor in a communicator. Meanwhile, a broadcast in Phoenix is an operation that delivers a message to every virtual node in a virtual node name-space. Thus, when a virtual node migrates from one processor to another upon the joining or leaving of a processor, a broadcast message must arrive at one of the two processors, but not both.

A *reduction* in Phoenix is an operation that performs some mathematical operation, such as computing the sum or finding the maximum, on data mapped to all virtual nodes, and delivers the result to one virtual node. When a virtual node migrates from one processor to another upon the joining or leaving of a processor, one of the two processors, but not both, must contribute data to the reduction.

As Phoenix allows the processors involved in a computation to change dynamically, existing static strategies to perform collective operations efficiently need to be extended. With a static tree, processors that join after application startup and the children of processors that leave after application startup are not able to participate in collective operations. Thus, a dynamic strategy, such as the one we propose in this paper, is required. While our strategy is most useful in the presence of joining and leaving processors, the self-tuning property of our strategy makes it valuable even in the context of MPI, in which processors neither join nor leave.

IV. Our Proposal

A. Overview

In our proposal, we dynamically create and maintain topology-aware spanning trees, and perform broadcasts and reductions along the generated trees. It is difficult to uniformly handle short messages and long messages; the performance of short messages is determined by latency, while that of long messages is determined by bandwidth. Thus, we create latency-aware spanning trees for short messages and bandwidth-aware trees for long messages. Moreover, each processor needs to be able to become the root of a broadcast or a reduction, so we create a tree for each processor.

Spanning tree creation takes the form of each processor's measuring the latency and bandwidth between itself and randomly-selected other processors to find a suitable parent for each tree. While each spanning tree is independent, running a separate spanning tree protocol for each tree would be too expensive. Thus, when a processor selects another processor at random, it measures the latency and the bandwidth between the two processors, and then uses the result to judge the suitability of the randomly-chosen processor for every tree. Latency is measured by actually sending a 1-byte message, and bandwidth is measured by sending a 128-kilobyte message.

Processors perform measurements and search for suitable parents at application startup and any other time that processors join or leave, but perform no measurements when processors are neither joining nor leaving.

When a processor joins, a new tree needs to be constructed with the joining processor as the root, and the joining processor must find a suitable parent for each of the other trees. Thus, upon the joining of a processor, each processor begins performing measurements with randomly-selected processors. Each processor continues performing measurements until it has found any parent for each of the trees, and then performs a certain number of extra measurements to improve the quality of the trees. While increasing the number of extra measurements will yield trees that resemble the topology better, decent trees will be created if each processor can select several processors within the same LAN. Therefore, in our implementation, each processor performs 10 extra measurements.

Moreover, when a processor leaves, it may leave some of the other processors parent-less for some of the trees. Such processors need to update their trees, so they begin performing measurements with randomly-selected processors. As in the case of a joining processor, each of these processors continue performing measurements until it has found any parent for each of the trees, and then performs ten more measurements. Processors that did not have the leaving processor as a parent for any of the trees are unaffected by the leaving processor, so they do not perform any measurements.

We present the two spanning tree algorithms in Subsections IV-B and IV-C, and then describe how broadcasts and reductions are performed using the generated trees in Subsections IV-D and IV-E.

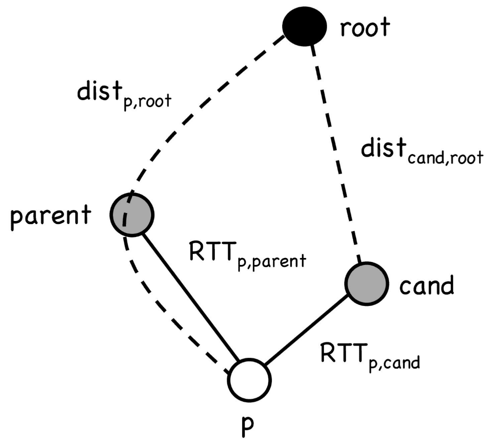


Fig. 3. The RTT between a processor and its parent or a parent candidate is determined by directly performing a ping-pong. A processor's distance from the root is computed by adding the RTT of each hop from the root to that node along the spanning tree.

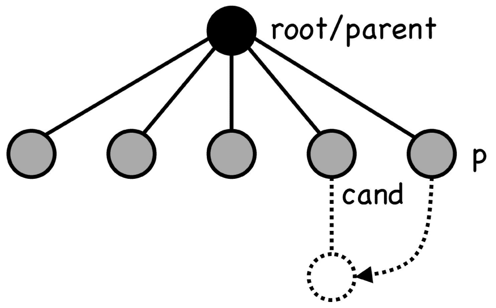


Fig. 4. In a shallow tree, a processor in a shallow position is able to move to a deeper position by becoming a child of one of its children.

B. Latency-Aware Spanning Tree

The goal of the latency-aware spanning tree algorithm is to create and maintain trees that overlap as much communication as possible. To achieve this goal, we construct trees with the following properties:

I) Few parent-child relationships between LANs
II) Random parent-child relationships within a LAN

Property I prevents the chaining of wide-area latencies, and property II keeps trees from becoming too deep or too shallow within a LAN.

For each tree, each processor begins by making any processor connected to that tree its parent. Once it has found a parent, it changes parents every time that it finds a more suitable parent. A processor judges the suitability of a parent candidate using four values: the round-trip time (RTT) between itself and its current parent, the RTT between itself and the parent candidate, its distance from the root along the tree, and the parent candidate's distance from the root along the tree. That is, processor p becomes a child of parent candidate $cand$ if $cand$ satisfies both the following conditions:

I) $RTT_{p,cand} < RTT_{p,parent}$
The RTT between itself and the parent candidate is shorter than the RTT between itself and its current parent.
II) $dist_{cand,root} < dist_{p,root}$
The distance between the parent candidate and the root processor is shorter than the distance between itself and the root processor.

Here, $RTT_{p,cand}$ and $RTT_{p,parent}$ are measured directly by performing ping-pongs, while $dist_{cand,root}$ and $dist_{p,root}$ are computed by adding the RTT of each hop along the spanning tree (Figure 3). P must have $dist_{cand,root}$ sent to it by $cand$, so p has $cand$ include $dist_{cand,root}$ in the 128 kilobytes used to measure bandwidth between p and $cand$.

When some processors join or leave, a spanning tree may end up with many parent-child relationships that stride two LANs, and it may become excessively deep or shallow within a LAN. However, the tree is quickly repaired as processors search for more suitable parents.

Parent-child relationships striding LANs are quickly replaced by those within a single LAN, because condition I causes processors to change parents to closer processors.

Trees that become excessively deep within a LAN quickly become shallower. To understand how this works, we assume that all the processors within a LAN have equal latencies. Then, if a processor selects a parent candidate within its own LAN, condition I will be satisfied with 1/2 probability. Moreover, if this processor is in a deep position, most parent candidates will be in a shallower position, and condition II will be satisfied with a high probability as well. Thus, a processor that ends up in an excessively deep position in the tree can move to a shallower position by selecting a few parent candidates within the same LAN.

Trees that become excessively shallow within a LAN quickly become deeper. As in the previous paragraph, if a processor selects a parent candidate within its own LAN, condition I will be satisfied with 1/2 probability. Moreover, in a shallow tree, a processor in a shallow position has many siblings, and if a processor selects one of its siblings as a parent candidate, condition II will be satisfied with 1/2 probability as well. Thus, a processor in a shallow position in a shallow tree can move to a deeper position by selecting a few parent candidates within the same LAN (Figure 4).

Each processor holds the following information for each of the spanning trees:

- Its parent processor
- Its child processors
- $RTT_{p,parent}$
- $dist_{p,root}$
- For each of the child processors, the set of virtual nodes mapped to it and its descendants

Information concerning the child processors is not necessary for creating the spanning trees, but is necessary for actually performing broadcasts and reductions.

C. Bandwidth-Aware Spanning Tree

The goal of the bandwidth-aware spanning tree algorithm is to create and maintain trees that use bandwidth efficiently for long broadcasts. To achieve this goal, we construct trees with the following properties:

I) High point-to-point bandwidth between the root and each node along the tree
II) Few branches

While property I alone results in high bandwidth point-to-point communication, high-bandwidth broadcasts require property II as well. This is because when a parent forwards a message to many children, network resources (i.e. the interface of the parent as well as the links and switches in between the parent and children) must be shared among the children.

As in the latency-aware spanning tree algorithm, each processor judges the suitability of parent candidates, changing parents every time that a parent candidate turns out to be more suitable than its current parent. While the best measure of suitability could be obtained by actually downloading a message from the root along the tree, such a method would stress the root too much. Thus, in our algorithm, a processor judges the suitability of a parent candidate based solely on information obtained from interaction between itself and the parent candidate. That is, processor p estimates $est_{p,cand}$, the rate at which it would receive data from parent candidate $cand$ during a broadcast, using the following information:

- bw_{p2p}: The point-to-point bandwidth between p and $cand$
- $n_{children}$: The number of children that $cand$ has (the number of siblings p would have if it became a child of $cand$)
- est_{cand}: The estimated rate at which $cand$ receives data from its parent during a broadcast

P obtains bw_{p2p} by measuring the time that it takes to download 128 kilobytes from $cand$. Moreover, it obtains est_{cand} and $n_{children}$ by having $cand$ include them in the 128-kilobyte message. Using this information, p computes $est_{p,cand}$ as follows:

$$est_{p,cand} = min(est_{cand}, bw_{p2p}/n_{children})$$

If $est_{p,cand}$ is greater than $est_{p,parent}$, the estimated rate at which p receives data from its current parent, p changes parents to $cand$.

Our estimation scheme of dividing the point-to-point bandwidth by the number of children is fairly accurate within a LAN, where the interface of the parent is likely to become the bottleneck. It is not so accurate in a WAN, where a wide-area link is likely to become the bottleneck, but dividing by the number of children still helps keep the number of branches small.

Each processor holds the following information for each of the spanning trees:

- Its parent processor
- Its child processors
- $est_{p,parent}$
- For each of its child processors, the set of virtual nodes mapped to it and its descendants

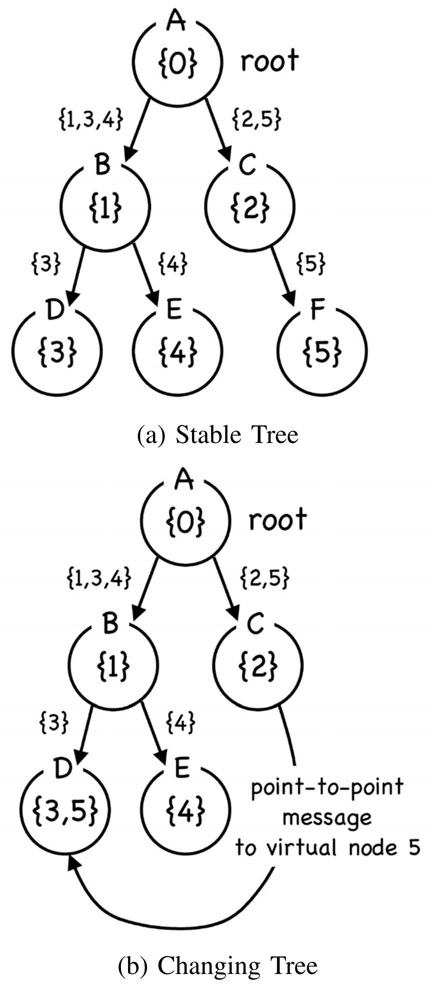


Fig. 5. In (a), the spanning tree is stable, so messages are forwarded efficiently along the spanning tree. In (b), processor F has given virtual node 5 to processor D just after processor A forwarded a message to processor C. As processor C does not have a child to forward virtual node 5 to, it sends a point-to-point message to virtual node 5.

D. Broadcast

As described in Section III, a broadcast in Phoenix delivers a message to every virtual node. In our broadcast, we allow messages for multiple virtual nodes mapped to the same processor to be delivered as a single message.

Short messages are forwarded along the latency-aware spanning trees described in Subsection IV-B, and long messages are pipelined along the spanning trees described in Subsection IV-C. Ultimately, we plan on using the measured values of latency and bandwidth to determine the message size at which to switch from the latency-aware trees to the bandwidth-aware trees. Each processor p can use the size of the message, $dist_{p,root}$, and $est_{p,parent}$ to estimate the time it would take a broadcast message to reach it along a given tree. The estimate can then be used to determine whether the latency-aware tree or the bandwidth-tree would minimize the completion time of the broadcast for a given message size. In this paper, however, we use the latency-aware trees for messages shorter than 256 kilobytes and the bandwidth-aware trees for messages longer than or equal to 256 kilobytes.

When the spanning trees are stable, a message is simply forwarded along a tree from the root to the leaves. When the spanning trees are changing, however, forwarding a message along a tree does not guarantee that all virtual nodes will receive the message. For example, if a processor leaves during a broadcast, the message will not reach its descendants.

Thus, when a processor forwards a message to a child, it includes in the header of the message the set of virtual nodes that are to be reached through that child. When a processor receives a message from its parent, it looks at the set of virtual nodes included in the header of the message and performs the following:

I) If any of the virtual nodes are mapped to itself, it delivers the message to the local user program.
II) For virtual nodes that can be reached through a child, it forwards the message along the spanning tree.
III) For virtual nodes that apply to neither I nor II, it forwards the message using point-to-point communication.

When the spanning trees are stable, message headers are consistent with the spanning trees, so broadcasts are performed efficiently using just I and II (Figure 5 (a)). Meanwhile, when the spanning trees are changing, inconsistencies between message headers and the spanning trees cause III to be used as well (Figure 5 (b)). Resorting to point-to-point communication lowers the performance of a broadcast, but guarantees that a message will reach all virtual nodes. When the spanning trees stop changing, message headers once again become consistent with the spanning trees, and III stops occurring.

E. Reduction

As described in Section III, a reduction in Phoenix performs some mathematical operation, such as computing the sum or finding the maximum, on data mapped to all virtual nodes, and delivers the result to one virtual node.

Messages are forwarded along the spanning trees described in Subsection IV-B, with leaf processors beginning a reduction by each sending its own data to its parent; a processor with multiple virtual nodes mapped to itself performs a local reduction first. The specified operation is performed on the data as they are forwarded along the tree from the leaves to the root, and the reduction completes when the result arrives at the processor to which the root virtual node is mapped. As the order in which data are operated on differ depending on the tree used, we only consider operations that are both associative and commutative.

When the spanning trees are stable, each processor simply waits for data from each of its children, performs the specified operation, and then sends the result to its parent. Meanwhile, when the spanning trees are changing, a processor may end up waiting for a child that has already sent data to a different processor.

Thus, we use a timeout mechanism in which a processor waits for only a certain amount of time for data from its children. Each processor sets a timeout when the reduction function is called, and when the timeout expires, the processor forwards whatever data it has received by that time. Any late data and data from non-child processors are forwarded as they arrive.

The timeout must be long enough so that it does not cause data to be forwarded prematurely when the topology is stable. One way to enforce this is to make the timeout longer than the time between the reduction root and the farthest leaf, but each processor does not know this value, and it would require much communication to collectively compute this value and deliver it to each processor. However, each processor does know the time between itself and the root of each spanning tree, so it uses the maximum of that value as an estimate.

V. Experimental Results

In this section, we present our experimental results. Each of the following experiments was performed using 128 to 201 real processors distributed over 3 or 4 clusters. The RTT of point-to-point messages using Phoenix was several hundred microseconds within a cluster and several milliseconds between clusters. We evaluate the performance of our collective operations when the spanning trees are stable in Subsections V-A and V-B, and the behavior of our collective operations when processors join and leave in Subsection V-C.

A. Stable-State Broadcast Performance

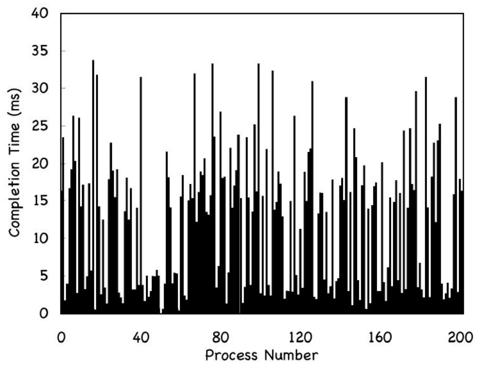
In our first experiment, we evaluated the stable-state performance of our broadcast by performing broadcasts after all processors had finished measuring latency and bandwidth with other processors and the spanning trees had become stable. One virtual node was mapped to each processor.

1) Short Messages: To evaluate the performance of short broadcasts, we measured the time it took a 1-byte message to reach each processor. We compared our implementation with two other implementations: a topology-unaware implementation that used the algorithm of MPICH, and a static topology-aware implementation that used the algorithm of MagPIe. Figure 6 shows our results for 201 processors distributed over 3 clusters.

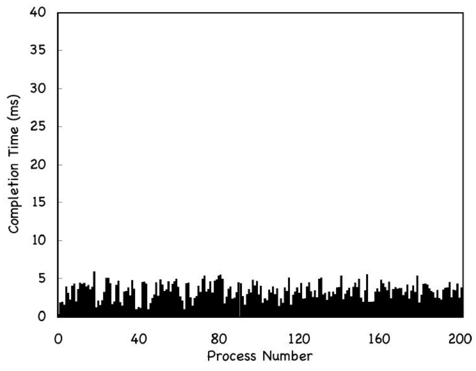
In the topology-unaware implementation, messages that stayed within a cluster arrived within several milliseconds, while messages that were forwarded between clusters many times took over 30 milliseconds to arrive (Figure 6 (a)).

Meanwhile, in the static topology-aware implementation, messages did not travel through wide-area links unnecessarily, so messages arrived at all processors within 6 milliseconds (Figure 6 (b)). This is 7 times faster than the topology-unaware implementation, which confirms the results of Kielmann et al.

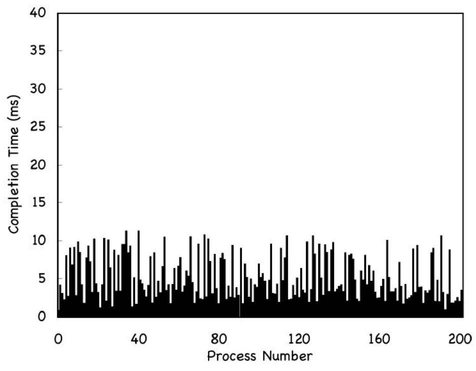
In our implementation, every processor received its message within 11 milliseconds (Figure 6 (c)). This is 3 times faster than the topology-unaware implementation, but almost twice as slow as the static topology-aware implementation. One reason for the slowdown is that our spanning tree algorithm did not make binomial trees within LANs—all leaves receive messages at the same time with a binomial tree, but leaves receive messages at different times with our tree. Another



(a) MPICH-like Implementation



(b) MagPIe-like Implementation



(c) Our Implementation

Fig. 6. 1-byte broadcast using 201 processors distributed over 3 clusters

reason is the overhead incurred by listing virtual nodes in the message header.

2) Long Messages: To evaluate the performance of long broadcasts, we measured the bandwidth of 32-kilobyte to 64-megabyte broadcasts. To compute the bandwidth, we divided the product of the number of bytes sent and the number of processors used (the minimum amount of communication required) by the completion time.

We compared our implementation (*Dynamic*) with a topology-unaware implementation (*MPICH-like*) and two static topology-aware implementations (*MagPIe-like* and *List*). In *Dynamic*, the latency-aware tree was used for messages shorter than 256 kilobytes, and the bandwidth-aware tree for messages longer than or equal to 256 kilobytes. In *MPICH-like*, the long-message algorithm of MPICH was used. In *MagPIe-like*, the root sent the message to the coordinator node of each cluster as in MagPIe, and then the long-message algorithm of MPICH was used within each cluster. In *List*, the processors were lined up in a list that made the bottleneck as wide as possible, and messages were pipelined along that list.

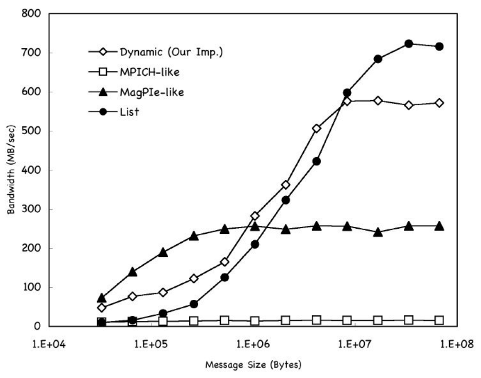


Fig. 7. Broadcast using 137 processors distributed over 4 clusters

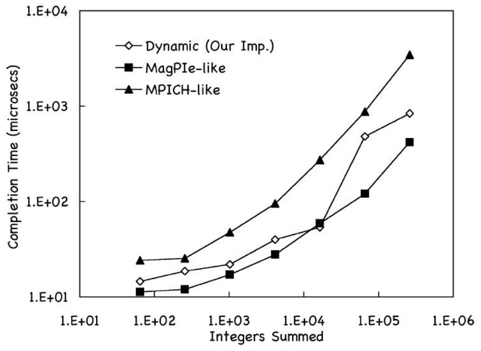


Fig. 8. Reduction using 128 processors distributed over 3 clusters

Figure 7 shows our results for 137 processors distributed over 4 clusters. As expected, *MagPIe-like* and *List* performed much better than *MPICH-like*; *List* outperformed *MagPIe-like* for very long messages, because a processor never sends the same data more than once in *List*. Our implementation was able to achieve 82 percent of the bandwidth of *List*, even though it was not branch-free.

B. Stable-State Reduction Performance

In our next experiment, we evaluated the stable-sate performance of our reduction. We mapped one virtual node to each processor, gave an array of integers to each processor, and measured the time to add the arrays together. Figure 8 shows the results for 128 processors distributed over 3 clusters. Just as with our broadcast, our reduction performed better than the topology-unaware implementation, but not quite as well as the static topology-aware implementation.

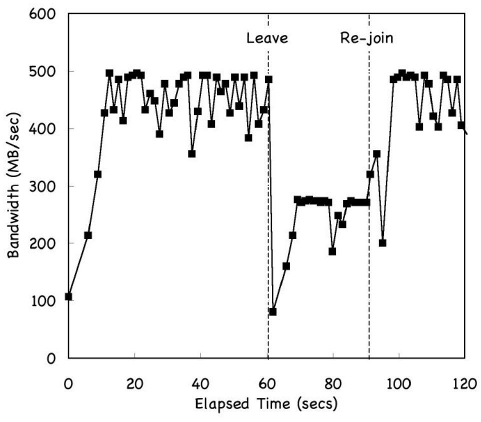


Fig. 9. Broadcast bandwidth when some processors left and then re-joined while 4-megabyte broadcasts were continually being performed

C. Transition-State Behavior

In our last experiment, we evaluated the transition-state behavior of our broadcast by having some processors leave and then re-join a computation while we repeatedly performed 4-megabyte broadcasts. We mapped one virtual node to each processor in the beginning, and then changed the mapping during computation as follows:

I) 60 seconds after application startup, half of the processors gave away their virtual nodes to the remaining processors and then left the computation.
II) Another 30 seconds later, the processors that gave away their virtual nodes in I re-joined the computation and re-acquired their original virtual nodes.

Thus, every processor had one virtual node during the first 60 seconds, and half of the processors each had two virtual nodes during the next 30 seconds.

Figure 9 shows the results for 160 processors distributed over 4 clusters. Immediately after application startup, broadcasts performed poorly because the spanning trees were under construction, but 11 seconds after application startup, processors finished searching for parents, and the performance of broadcasts stabilized.

When half of the processors left, the spanning trees broke, causing the performance of broadcasts to fall sharply. However, the spanning trees were quickly repaired, and the performance of broadcasts stabilized once again after 8 seconds. Similar behavior was seen when the processors that left earlier re-joined the computation.

In this way, our broadcast temporarily performs worse when processors join or leave, but messages are guaranteed to reach all virtual nodes, and our broadcast resumes effective execution once the spanning trees stabilize.

VI. Conclusion

In this paper, we proposed a method for wide-area message passing systems to perform collective operations using dynamically created spanning trees. We presented algorithms to create spanning trees using latency and bandwidth measured at run-time, and explained how to perform broadcasts and reductions using the generated trees.

In our experiments, we used 128 to 201 real processors distributed over 3 to 4 clusters and showed the following:

I) In stable-state, our broadcast and reduction performed much better than those of a topology-unaware implementation, although not quite as well as those of a topology-aware implementation. In particular, the latency of our 1-byte broadcast was 3 times as short as that of a topology-unaware implementation, and within a factor of 2 of that of a static topology-aware implementation. Moreover, our 64-megabyte broadcast achieved 82 percent of the bandwidth of a static topology-aware implementation.
II) When some processors joined or left a computation, our broadcast temporarily performed worse for about 8 seconds while the spanning trees adapted to the new topology, but completed successfully even during this time.

In the future, we plan on optimizing our broadcast and reduction to close the gap between our implementation and static topology-aware implementations. In our current broadcast, there is no given order in which each processor forwards messages to its children. However, if we send to the child that has the most descendants first, we will be able to overlap more communication. Moreover, we only consider RTT in the latency-aware spanning tree algorithm, but we can model the network better by also considering forwarding overhead. Finally, we can decrease the forwarding overhead in broadcasts by compressing the virtual nodes listed in the message header.

Moreover, we plan to implement collective operations besides broadcast and reduction using our spanning trees. Other *1-to-N* operations and *N-to-1* operations can be implemented like the broadcast and reduction operations, but *N-to-N* operations, such as *all-to-all* and *barrier*, require a new strategy.

Finally, in the spanning tree algorithms proposed in this paper, processors do not share much information with each other. However, by sharing more information, processors may be able to create the spanning trees more efficiently. For example, instead of selecting parent candidates randomly, a processor can use hints given by other processors to preferentially select nearby processors.

Acknowledgements

This work is partially supported by “Precursory Research for Embryonic Science and Technology” of Japan Science and Technology Corporation and by “Grand-in-Aid for Scientific Research” of Japan Society for the Promotion of Science.

We would like to thank the anonymous reviewers for their comments, which helped us improve our paper. We are also grateful to the reviewers of the HPDC-14 conference, who guided us to relevant literature and insights concerning bandwidth-aware broadcasts.

REFERENCES

[1] T. Kielmann, H. E. Bal, and S. Gorlatch, “Bandwidth-efficient collective communication for clustered wide area systems,” in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, 2000, pp. 492–499.

[2] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, “MagPIe: MPI’s Collective Communication Operations for Clustered Wide Area Systems,” in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999, pp. 131–140.

[3] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, “Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance,” in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, 2000, pp. 377–384.

[4] N. T. Karonis, B. Toonen, and I. Foster, “MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface,” *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551–563, 2003.

[5] E. Gabriel, M. Resch, T. Beisel, and R. Keller, “Distributed Computing in a Heteregeneous Computing Environment,” in *Proceedings of the 5th European PVM/MPI User’s Group Meeting*, 1998, pp. 180–187.

[6] P. Husbands and J. C. Hoe, “MPI-StarT: Delivering Network Performance to Numerical Applications,” in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998, pp. 1–15.

[7] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. James W. O’Toole, “Overcast: Reliable Multicasting with an Overlay Network,” in *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, 2000, pp. 197–212.

[8] A. Bozdog, R. van Renesse, and D. Dumitriu, “SelectCast - A Scalable and Self-Repairing Multicast Overlay Routing Facility,” in *Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems*, 2003, pp. 33–42.

[9] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, “Scalable Application Layer Multicast,” in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002, pp. 205–217.

[10] K. Taura, T. Endo, K. Kaneda, and A. Yonezawa, “Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources,” in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003, pp. 216–229.

[11] MPICH-A Portable Implementation of MPI, “http://www-unix.mcs.anl.gov/mpi/mpich.”

[12] Message Passing Interface (MPI) Forum, “http://www.mpi-forum.org.”

[13] R. Thakur and W. Gropp, “Improving the Performance of Collective Operations in MPICH,” in *Proceedings of the Euro PVM/MPI*, 2003, pp. 257–267.