社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－HPC－99 (38)
2004／8／1

Phoenixプログラミングモデル用の集団通信

斎藤秀雄[††] 田浦健次朗[††] 近山隆[†,††]

本稿ではグリッド環境において集団通信のために用いるスパニングツリーを構築するアルゴリズムを提案する。スパニングツリーを動的に構築することによってネットワークのトポロジーが変化しても効率の良い集団通信を行い続けることができる。実験では、3つのクラスタにまたがる168台の計算機において、レイテンシーを考慮したスパニングツリーを構築することに成功した。提案するアルゴリズムを用いてPhoenixメッセージ・パッシング・ライブラリに実装したブロードキャスト関数はトポロジーを考慮しないブロードキャストより明白に良い性能を示したが、事前に静的に構築されたツリーを用いたブロードキャストには及ばなかった。

Collective Operations for the Phoenix Programming Model

HIDEO SAITO,[††] KENJIRO TAURA[††] and TAKASHI CHIKAYAMA[†,††]

In this paper, we propose an algorithm to create spanning trees for use in collective operations in a Grid environment. By creating the spanning trees dynamically, we are able to continue performing collective operations efficiently even when the topology of the network changes during program execution. In our experiments, we succeeded in constructing latency-aware spanning trees with as many as 168 nodes in 3 clusters. The broadcast function that we implemented on the Phoenix message passing library using this algorithm decisively outperformed a topology-unaware broadcast, although it did not perform quite as well as a broadcast using static trees that were created a priori.

1. Introduction

1.1 Background

Collective operations are a vital part of parallel programming based on message passing. Compared with their counterparts using send-receive primitives, broadcasts and reductions provided as a single function are not only easier for the programmer to read and write, but also perform better.[1] Collective operations perform better because they can be implemented with knowledge of the underlying network. In particular, latency and bandwidth can be used to construct spanning trees in such a shape that messages are forwarded efficiently from one process to another.

Early work on collective operations sought to create operations that were optimal under the assumption that all nodes were equidistant from one another. This assumption does not hold in Grid environments—two nodes within the same cluster may be just a few microseconds apart, while two nodes located on opposite sides of the globe will be separated by over 100 milliseconds. Some implementations of MPI, such as Magpie[2] and MPICH-G2[3),4)], have implemented efficient collective operations for wide area networks. These implementations require advance information about the topology of the network to aid the creation of efficient spanning trees. Yet, in some programming models for the Grid, resources may join or leave in the middle of a computation, making it impossible for topology information to be provided in advance. The Phoenix message passing library[5] follows such a programming model.

In this paper, we propose an algorithm to create spanning trees for use in collective operations in a Grid environment. Unlike the spanning trees used in Magpie or MPICH-G2, our spanning trees are not created statically a priori. Instead, our algorithm uses latency measured at run-time to create the spanning trees dynamically during program execution. By creating the spanning trees dynami-

† Department of Frontier Informatics, Graduate School of Frontier Sciences, the University of Tokyo
†† Department of Information and Communication Engineering, Graduate School of Information Science and Technology, the University of Tokyo

cally, we are able to continue performing collective operations efficiently even when the topology of the network changes during program execution. We implemented a broadcast function that uses this algorithm on the Phoenix message passing library. Our experiments showed that broadcasts using our algorithm decisively outperformed topology-unaware broadcasts, although they did not perform quite as well as broadcasts using static trees that were created a priori.

1.2 Organization of this Paper

The rest of this paper is organized as follows. In Section 2, we discuss related work. We then describe our algorithm in Section 3 and discuss our experiments in Section 4. Finally, we summarize in Section 5.

2. Related Work

Topology-aware collective operations have been implemented in the past. These implementations, however, have all required advance information about the topology of the network to achieve efficiency.

Magpie[2] is a library of collective operations optimized for wide area systems. It achieves high performance by using algorithms that are wide area optimal—every data item incurs only a single wide area latency, and every data item is sent at most once across each wide area link. To use these wide are optimal algorithms, Magpie must be told how many clusters there are and which process is located in which cluster. Yet, in some programming models for the Grid, this information changes over time, and we cannot rely on it to optimize our algorithms. Phoenix, the message passing library that we are targeting, follows such a programming model. Thus, in our work, we aim to create trees that are wide area optimal, without being explicitly told the topology of the network.

MPICH-G2[3),4)] implements multilevel topology-aware collective operations. While Magpie only distinguished between two levels of communication, "intracluster" and "intercluster," MPICH-G2 aims to achieve higher performance by introducing more levels of communication. In order to accomplish this, each process is assigned a topology depth and a color. The topology depth is the number of network levels that a process is associated with (wide area, local area, system area, etc.). Using these topology depths, MPICH-G2 groups processes at a particular level through the assignment of colors. Two processes are assigned the same color at a particular level if they can communicate with each other at that network level. Topology depths and colors are told to MPICH-G2 via a script called the Resource Specification Language (RSL) script.[6] Thus, once again, in cases where the topology of the network may change, this approach cannot be used.

3. Algorithm

In this section, we will describe our spanning tree algorithm. The idea is for each process to be the root of a spanning tree. For each tree, each node remembers its parent and children. In 1-to-N operations, such as broadcasts☆1 and scatters☆2, messages are sent down the tree from the root to the leaves. In N-to-1 operations, such as gathers☆3 and reductions☆4, messages are sent up the tree from the leaves to the root. We have not yet considered how N-to-N operations, such as all-to-alls and barriers, fit into our work.

3.1 The Main Algorithm

The spanning trees are created and updated in a distributed manner. For each spanning tree, each node looks for a suitable parent, changing parents as it finds more suitable ones. The criteria for suitability are described in Fig. 1.

Fig. 1 (1) and (2) together keep the spanning tree wide area optimal. At first, the spanning tree may have many connections between the same clusters. Yet as it is updated, all but one of those connections will be replaced by ones within the same cluster, because nodes within the same cluster have lower latency.

Within a cluster, a binomial tree, such as the one used in MPICH[7], offers optimal performance by

☆1 A broadcast is an operation in which one node sends the same data to all other nodes.

☆2 A scatter is an operation in which one node sends different data to each of the other nodes.

☆3 A gather is an operation in which one node receives data from each of the other nodes.

☆4 A reduction is an operation in which one node receives the result of a computation, such as summation, on data held by each of the nodes.

Given a node n with a parent p and a parent candidate c for the spanning tree with r as the root, c is a more suitable parent than p if it satisfies all four of the conditions below:

(1) c is connected to the spanning tree with r as the root

(2) $latency_{n,c} < latency_{n,p}$

(3) $latency_{c,root} < latency_{n,root}$

(4) $serial_{c,root} = serial_{n,root}$

Here, $latency_{x,y}$ denotes the latency between x and y, and $serial_{x,y}$ is the largest serial number that x has received from y.

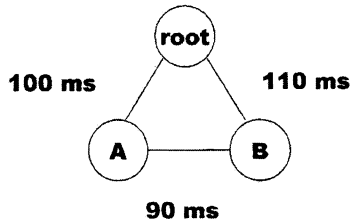
Fig.1 Criteria for choosing whether a parent candidate is more suitable than the current parent.

maximizing the amount of overlapped communication. Meanwhile, Fig. 1 (1) and (2) alone create trees that are too sparse and deep. Fig. 1 (3) prevents a node from choosing a parent that is farther from the root than it is, thereby keeping the spanning tree from becoming too deep.

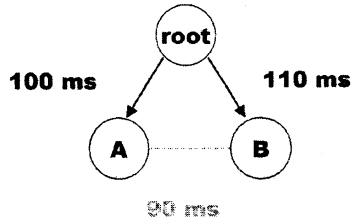
Fig. 1 (4) prevents parts of the spanning tree from forming isolated cycles. Each node occasionally broadcasts a serial number down the spanning tree for which it is a root. Non-root nodes forward the serial number along with updated latency information. This ensures that if two nodes have the same serial number, the node with higher latency cannot be a descendant of the other node. Without these serial numbers, a node may temporarily have higher latency than its descendant—after it has changed parents to a node that raises its latency, but before that information has arrived to one of its descendants.

3.2 Shortcuts

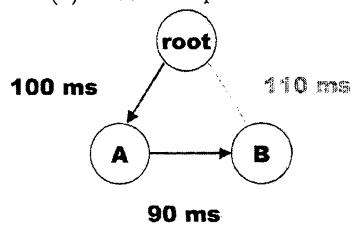
The algorithm described thus far is able to construct spanning trees given any network topology. For some topologies, however, it may produce spanning trees that are not wide area optimal. We show an example in Fig. 2. We show the latency between 3 nodes in Fig. 2 (a), and a wide area optimal tree in Fig. 2 (b). Our algorithm, however, would create the tree shown in Fig. 2 (c), which is not wide area optimal. Such a tree is created, because B chooses A as its parent based on the criterion that it is closer than other parent candidates, in this case the root node. The problem is that in doing so, we



(a) Latency between 3 nodes



(b) Wide area optimal tree



(c) Tree created without shortcuts

Fig.2 An example where shortcuts are needed to retain wide area optimality.

If a node n with a parent p finds a parent candidate c that would shorten $latency_{n,root}$ by over 10 milliseconds, n can make c its parent, even if $latency_{n,c} > latency_{n,p}$.

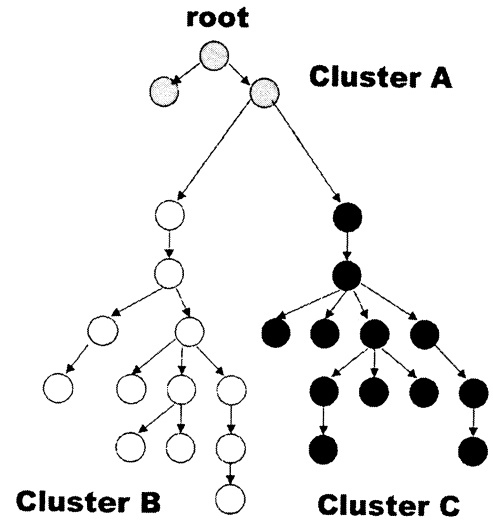
Fig.3 Shortcut mechanism to preserve wide area optimality.

ignore the fact that B places itself 80 milliseconds farther from the root node than if we were to choose the root node as B's parent.

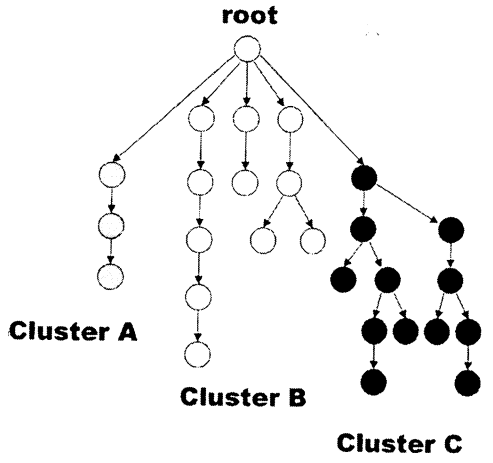
In order to keep the tree wide area optimal, we introduce the shortcut mechanism described in Fig. 3. Using this mechanism, B in Fig. 2 would make the root node its parent, preserving wide area optimality.

4. Experiments

In this section, we will discuss the experiments that we performed to evaluate our tree-creation algorithm and broadcast implementation.



(a) One of the nodes of Cluster A as the root



(b) One of the nodes of Cluster B as the root

Fig.4 Two of the trees created using 27 nodes in 3 clusters.

4.1 Spanning Tree Creation

First, we took a look at the shape of the spanning trees created by our algorithm. We used 27 nodes in 3 clusters (3 nodes in the first cluster and 12 nodes each in the other two clusters) and created a spanning tree with each node as the root.

Fig. 4 (a) shows the tree created with one of the nodes in the 3-node cluster as the root. We can see that the tree is not optimal, but that it is wide area optimal. We say that it is not optimal, because the intra-cluster communication deviates from a binomial tree, and because the root node fails to deliver data to the other clusters directly.

Fig. 4 (b) shows the tree created with one of the nodes in one of the 12-node clusters as the root. Once again, we can see that the tree is not optimal, but that it is wide area optimal.

Other trees with other nodes acting as the root had similar shapes and characteristics—none were optimal, but all were wide area optimal.

4.2 Low-Latency Broadcast

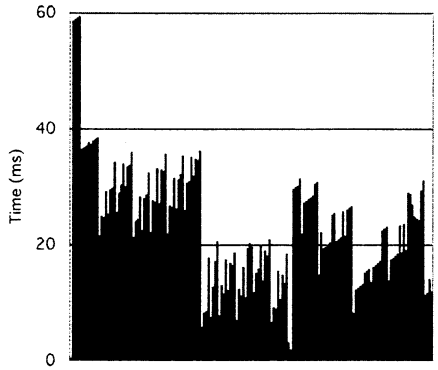
Next, we modified the Phoenix message passing library by adding a broadcast function that sends data down spanning trees created by our algorithm, and compared it with two other implementations: a topology-unaware broadcast and a broadcast that uses a static tree determined a priori.

The topology-unaware broadcast simply imitated the Grid-unaware approach taken by MPICH—it used a binomial tree, but as it did not take network topology into consideration, it was an inefficient one. Data could travel to the same cluster multiple times—worse yet, data could travel back and forth between the same clusters.

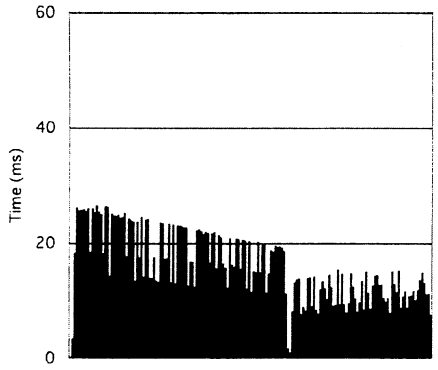
For our Grid-aware static tree, we used the strategy employed by Magpie. A coordinator node represented each cluster, and data was delivered to each coordinator node in a flat tree (in sequential sends). Inside clusters, a binomial tree was used to broadcast messages from the coordinator node to all other nodes.

We used 168 nodes in 3 clusters (100 nodes in the first cluster, 65 nodes in the second cluster, and 3 nodes in the last cluster), and our broadcast message size was 1KB.

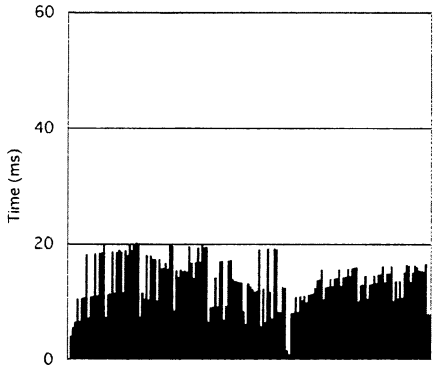
Fig. 5 shows the time before each of the 168 nodes returned from the call to the broadcast function. Our implementation decisively outperformed the topology-unaware implementation, although it did not perform quite as well as the implementation using static trees determined a priori. Our implementation was able to outperform the topology-unaware implementation because it was wide area optimal. It was not able to match the performance of the static/a priori implementation because its intracluster trees strayed too far from the optimal binomial trees.



(a) Topology-Unaware



(b) Our Implementation



(c) Static/A Priori

Fig.5 Broadcast over 168 nodes in 3 clusters.

5. Summary and Future Work

In this paper, we have proposed an algorithm to dynamically create topology-aware trees that can be used for collective operations in a Grid environment. We modified the Phoenix message passing library to create such trees, and implemented a broadcast that uses these tress. Our experiments showed that broadcasts using our algorithm decisively outperformed topology-unaware broadcasts, although it did not perform quite as well as broadcasts using static trees that were determine a priori.

This work is in its early stages, and much remains to be done. Our broadcast operation could be optimized by using the spanning tree in a smarter way. Currently, each node forwards to its children in the order that it encounters them. If we send to the children that have the most descendants first, we may be able to overlap more communication with each other.

We can also improve our work on collective operations by taking the size of messages into account. This entails not only switching algorithms for small and large messages as MPICH does, but also measuring bandwidth and using that information to create trees.

One immediate area of concern is implementing a complete set of collective operations, not just broadcast: scatter, gather, reduction, and all-to-all. Scatters, gathers, and reductions can immediately make use of the topology-aware trees proposed in this paper. All-to-alls may require a different strategy.

Another area of concern is extending our tree-creation algorithm for use in other Grid-related applications. We believe that other message passing models, such as MPI, can also take advantage of our algorithm to reduce the amount of necessary configuration.

References

1) Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, 2004.
2) Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. Magpie: Mpi's collective commu-

nication operations for clustered wide area systems. In *PPoPP'99*, pages 131–140, 1999.
3) Nicholas T. Karonis, Brian Toonen, and Ian Foster. Mpich-g2: A grid-enabled implementation of the message passing interface.
4) N. Karonis, B. deSupinski, 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 Parallel and Distributed Processing Symposium*, 2000.
5) Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix: a parallel programming model for accomodating dynamically joining/leaving resources. In *PPoPP'03*, pages 216–229, 2003.
6) K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.
7) MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.